András Vajda

Programming Many-Core Chips

Springer

Programming Many-Core Chips

András Vajda

Programming Many-Core Chips

With Contributions by Mats Brorsson
and Diarmuid Corcoran

Foreword by Håkan Eriksson

Springer

András Vajda
Oy L M Ericsson Ab
Hirsalantie 11
02420 Jorvas
Finland
andras.vajda@ericsson.com

ISBN 978-1-4419-9738-8 e-ISBN 978-1-4419-9739-5
DOI 10.1007/978-1-4419-9739-5
Springer New York Dordrecht Heidelberg London

Library of Congress Control Number: 2011930407

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Dedicated to the loving memory of my grandmother.
Mama, without your dedication and sacrifices, I would have never been in a position to accomplish this.

Thank you.
András Vajda

Foreword

Parallel computing has been confined, for much of its over 40 year history, to highly specialized, technology-wise advanced domains such as scientific computing or telecommunications. There were only a few experts who had the background and experience to write efficient, robust and scalable programs for these parallel machines.

A few years ago all that suddenly changed with the emergence of multi-core and many-core processors, as we reached the end of seemingly endless single-core performance scaling. While Moore's Law still provides chip designers with ever-increasing amount of transistors for building chips, these chips will now all have multiple cores and there's no end in sight for the scalability of computing power through the integration of ever more processor cores on the same piece of silicon. Practically overnight parallel programming skills became mandatory for most experts working in the field of computer programming.

Ericsson has always been at the forefront of parallel programming. We have been delivering best in class massively parallel telecommunication systems for several decades and providing our customers with quality software running on systems with hundreds of processors has always been one of our core assets. Continued leadership in this area is essential for our future success: the amount of traffic in telecom networks and consequently, the amount of computation required to sustain it, is expected to grow orders of magnitude faster than what the chip industry can match, even if it continues to deliver according to Moore's law. I believe we are well positioned to take advantage of emerging chips with hundreds or even thousands of cores in order to sustain our technology leadership in ICT.

This book is the result of the experience we as a company and our experts as individuals have accumulated over the past decades. It is driven by our desire to share with the software community the knowledge we built the hard way: learning from our mistakes and building on our successes. It is our strong belief that co-operation is essential in order to spark and sustain innovation that can generate the cutting

edge technologies from which we all—the computing community and the society in general—can ultimate benefit.

Håkan Eriksson
Senior Vice President, Chief Technology Officer
Head of Group Function Technology & Portfolio Management
Head of Ericsson in Silicon Valley

Contents

Chapter 1
Introduction

Abstract The primary goal of this chapter is to introduce the background and historical reasons that led to the emergence of multi-core chips. We look at some of the most important challenges—power constraints, memory latency and memory bandwidth issues—that accompanied the development of computing environments during the past two decades. Within this context we introduce the purpose and the content of this book: a survey of the state of the art of practice and research into programming chips with tens to hundreds of cores. The chapter is concluded with the outline of the rest of the book.

1.1 The End of Endless Scalability

Ever since the microprocessor made its debut at the dawn of the eighth decade of the last century, the computer software industry has enjoyed something of a free ride—free lunch, if you wish [1]—in terms of computing power available for software programs to use. Moore's law [2], driven to self-fulfillment by the hardware industry's desire to deliver better and better chips, secured doubling number of transistors—and consequently, at least till recently, hardware performance—every two years or so; if you ran into performance problems, those were solved within a few months through the release of newer, faster, better hardware. Writing parallel programs was something that only the high performance computing experts or domain experts working in similar, highly specialized fields of computing had to worry about.

This free ride however came to a sudden halt at around 2004, when the speed of a single processor core topped out at about 4 GHz. Since then, while Moore's law [2] still held (as exemplified in Fig. 1.1) and there were more and more transistors crammed on smaller and smaller areas, the performance per single processor core, measured as the amount of instructions per second it could execute, remained constant or actually started to decline, along with the frequency at which the core was run. All major hardware vendors started shipping processors with multiple proces-

A. Vajda, *Programming Many-Core Chips,*
DOI 10.1007/978-1-4419-9739-5_1, © Springer Science+Business Media, LLC 2011

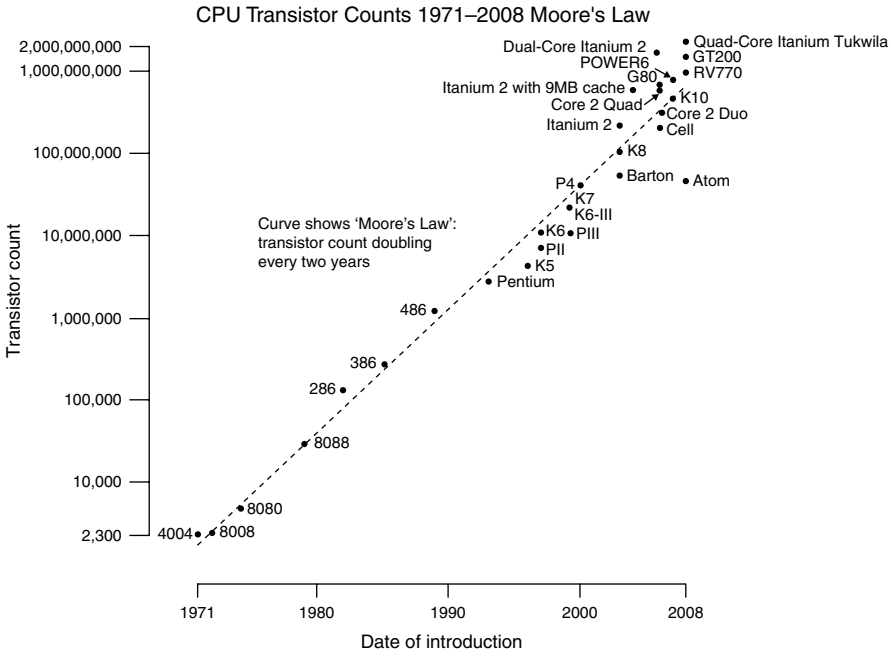


Fig. 1.1 Moore's law. (Source: Wikipedia, under Creative Commons license)

sor cores, finding thus a new use for the transistors churned out in accordance with Moore's law.

Why did this happen?

The prime reason was power consumption and power dissipation. The extra power consumption and dissipation resulting from pushing up the frequency of a single core made this path simply not sustainable—neither from economical nor from technological perspective. In terms of instructions per chip per second, two or more cores running at lower frequencies could deliver significantly better performance within the same power budget than a single core running at a higher frequency. Following the rule of diminishing returns, it was simply not viable to improve the performance of a chip with some low percentage points at the expense of dramatic increase in power consumption—sometimes toppling 100%. From hardware perspective, Moore's law still guaranteed higher transistor densities and all those transistors could be used to execute more instructions per second, without an unsustainable increase in the power consumption of chips.

There was another, more subtle observation that triggered the push for chips with large numbers of simpler cores. It was most concisely formulated in Ref. [3] as the KILL rule (stands for Kill if Less than Linear). Essentially, it argues for omitting architectural innovations—such as deeper pipelines, speculative threads etc—if the performance gain is less than linear (with respect to number of transistors and power required). It is one of the fundamental principles underpinning the design of chips such as Tilera's family of Tile-architecture based, massively multi-core chips.

1.2 The Trouble with Software

This approach however shifted the focus to software. Suddenly, software writers found themselves in the situation of seeing the performance of their products stagnate or even run slower on newer hardware. For many, this was—and is—a major shock: one had to rethink the software architecture and go through the painful process of re-architecting a working piece of software in order to keep up with the performance requirements that never ceased to come, while re-educating programmers so used to single-threaded coding and free performance boost from newer hardware. Needless to say, this adjustment came at a cost far greater than the Y2K problem; even worse, we are just at the beginning of this process, as for many applications the problem of dealing with multi-core processors is yet to come.

The challenges for the programming community do not stop here, however; there are two other factors that will have a decisive impact on the way we will build software: memory interface and scaling.

It's one of the inconvenient truths of the computing industry that on-chip computing power kept increasing much faster than the speed of access to memory. Hardware providers tried to address this yawning gap through ever more sophisticated caching architectures, smart predictors and pre-fetch engines, with varying degrees of success for single-core applications. The emergence of multi-core chips adds however an extra dimension to this problem, *concurrent and competing access* to the same scarce resource. This brings a whole new set of issues that cannot be solved anymore in hardware alone and hence will impact the software as well: how to access a shared memory location concurrently without a major impact on performance; how to negotiate, prioritize and manage concurrent and competing requests to different locations in the memory; how to guarantee integrity and consistency over time of data logically belonging to different cores; in general, how best to exploit a scarce resource (memory bandwidth and on-chip memory) that just got a lot more scarcer.

The second challenge, with a much more far-reaching consequence is that of scale. The chip industry has recently coined the 'new' Moore's law, predicting that the number of cores per chip wills double every two years over the next decade, leading us into the world of *many-core processors*—loosely defined as chips with several tens, but more likely hundreds, or even thousands of processor cores. Depending from where we start (there are today general purpose chips with 4 to up to 100 cores), this law, if proven to be true, will mean chips with 32–512 cores by 2014 and between 256 and 4000 cores by 2020. Indeed, looking back and setting 2004 as the base year (the last year with just one core per chip), we did see chips with dual cores in 2006 (in fact, already during 2005) and with 4 cores in 2008, chips with at least 8 cores being delivered by all major chip vendors during 2010; Intel has recently announced [4] a new chip with more than 50 cores, scheduled for introduction in 2012 or 2013.

All this data backs up, albeit within a limited time-span, the claims of the new Moore's law. At this scale, constructs and techniques that work well for processors

with just a few cores—shared memory, locks, monolithic OS kernels, cache systems—have been shown to become bottlenecks that are increasingly more difficult to tackle with every step in chip development.

1.3 The Book

This is the setting where this book fits in. It is based on two beliefs: first, that we will indeed see widespread, even dominating, presence of chips with hundreds of cores within five years; second, that using and programming these chips will require fundamental shifts in how all the layers of computing—from the hardware, through the virtual machine monitor, operating system and middleware to the application software, including programming models, paradigms and programming languages—will be constructed, adapted, re-invented and connected. We do not aim at introducing some revolutionary new way to address these issues; rather, our goal is to present the state of the art of research and practice of designing the various layers of the software stack for chips with tens to hundreds of processor cores, along with our critical analysis of the suitability of technologies and approaches for various problem domains. Our deep conviction is that there is no silver bullet—but there is a consistent body of high quality research that every programmer using the chips of the future shall be aware and familiarized with and use it for the domain he or she works with. We set out to provide the tool for this, through the book you are now holding in your hands.

Consequently, this book is structured in four main parts. In our view, no programmer can claim to be an expert without a good understanding of the hardware on which his or her software will execute: how the hardware is organized, how programming concepts are mapped to the hardware, how the way hardware works can influence the correctness and performance of the software. This is the focus of the first part of the book, which consists of only one chapter: *Multi- and Many-core Processor Architectures.* In this chapter we evaluate different technologies related to instruction set architectures, cache and shared memory structure and scalability, on-chip interconnects, power design, but also novel ideas such as 3D stacking and innovative usage of frequency and voltage scaling.

The second part introduces the current state of the art in existing technologies for programming multi-core processors, covering all layers from operating systems to the application programming models. It lays the foundation for part three, through the detailed discussions of the concepts related to parallel programming in general. Besides introducing the terminology that we will use in the second part of the book, we also take a critical look at the bottlenecks and limitations with current solutions, thus setting the frame for the ideas presented in the third part. This part consists of four chapters:

- *Chapter 3, Multi-core Operating Systems: State of the Art* introduces the concept of operating system, the roles of an operating system as well as the various

architecture choices of main-stream operating systems. The chapter also surveys the concrete implementation mechanisms behind the most important operating systems of today: Linux, Solaris and Windows, with respect to issues relevant from many-core programming perspective

- *Chapter 4, Fundamental Laws of Parallelism* focuses on the fundamental laws of parallel programming: Amdahl's [5] and Gustafson's [6] law, the Karp-Flatt metric as well as the relationship between these. We also describe some of the more controversial observations and qualitative laws, such as Gunther's conjecture and the KILL rule [3] we already mentioned
- *Chapter 5, Fundamentals of Parallel Programming* describes the basic concepts of parallel computing—decomposition and synchronization—as well as the ways to realize these. This chapter also covers the main patterns of parallel programs, captured in the unified pattern language for parallel programs called OPL [7]
- *Chapter 6, Debugging and Performance Analysis of Many-core Programs* is an introduction to the concepts of debugging, analysis, profiling and performance tuning of programs targeting many-core processors

The third part of the book targets novel approaches to programming true many-core chips (with hundreds to thousands of cores). This part includes three chapters:

- *Chapter 7, Many-core Virtualization and Operating Systems* discusses the scalability issues of current operating systems, such as limitations of symmetric multi-processing approaches, current resource management and scheduling policies, as well as approaches to synchronization, computation and data locality; it describes the principles that shall underpin the architecture of operating systems for many-core chips: space-shared scheduling, support for heterogeneity, power awareness and the role of virtualization. We also introduce a conceptual architectural model for operating systems for many-core chips, based on research results from the academic and industrial community, as basis for building programming models and programming languages. We will illustrate these concepts through some of the most promising research operating systems emerging from the OS research community
- *Chapter 8, Introduction to Programming Models* focuses on the models we believe are best suitable for many-core chips: communicating sequential processes, the actor model and task-based parallelism, along with a comparison of process versus task based parallelism. We illustrate each of these models through representative programming languages and libraries, such as
  - Occam and the Go language for the communicating sequential processes model
  - Erlang, Haskell and Scala for the actor model
  - X10 and Chapel for the task based model
- *Chapter 9, Practical Many-core Programming* is a comprehensive analysis of the principal frameworks for task-parallel, data-parallel and actor based

programming. We will illustrate, through concrete examples and an in-depth description of the internals, the following:

- Cilk, Apple's Grand Central Dispatch, Intel's Thread Building Blocks, Microsoft's Task Parallel Library and OpenMP 3.0 for task-parallel programming, including a comparative analysis of the strengths and weaknesses of these libraries
- OpenCL for data-parallel and—to some extent—task parallel programming
- Erlang for actor-based programming

No book on many-core programming would be complete without a peek into the crystal ball for what the future holds in terms of technological evolution and challenges—so we'll end with our take at how the computing industry will look like within ten years. This is the focus of the final part of the book, consisting of *Chap. 10, Looking Ahead.* We'll do this despite the warning from Winston Churchill, that "it's difficult to make predictions, especially about the future" and all the failed prophecies of the computing industry over its more than 60 years of existence that proved to be over-cautious understatements. We will analyze the problem area of scaling programs with limited parallelism; the need for programming at higher abstraction level; as well as the interaction with some related domains such as cloud computing, mobile computing and exascale computing.

1.3.1 Applications

We took a decision early on to exclude discussion of specific application domains. Our goal is to describe and compare the programming mechanisms in general; it's beyond our intention or, indeed, capabilities, to make qualified recommendations for different domains. It would also be difficult to cover all important applications; leaving some out would result in an incomplete work.

It's our strong belief that eventually most application domains will face the challenge of adapting to multi-core chips—consequently, our duty is to address as big part of the community as possible.

1.4 Summary

We have a confession to make: after concluding the writing of the rest of the book, we had to re-visit the content of this chapter. Over the months of working on the book, our field has changed so much that we had to adapt "on the fly" and consequently, the content of the book is different compared to what we originally envisioned and initially described in the first version of this chapter.

In this quickly changing domain, we tried to avoid insisting on transient "fashion concepts" and instead focus on the important fundamentals required to understand

the big picture of programming many-core chips—principles and practices that have proven to be sound over and over and are unlikely to change. Whether we succeeded or not, it's up to you, dear reader, to decide. We can only hope you will find the content of this book insightful and useful for your work or passion.

We cannot conclude this chapter without thanking all those who supported us in this work: first of all our families who endured the extended periods of time dedicated to writing; Anders Caspár at Ericsson for his continued support and encouragement; our editor at Springer, Chuck Glaser, who initiated this book in the first place and whose discrete yet efficient shepherding made sure that we actually deliver it; last but not least, all those colleagues and friends who, through their input, comments or simply encouragement facilitated the writing of this book.

References

1. Sutter, H (2005) The Free Lunch is Over: A Fundamental Turn toward Concurrency in Software. Dr. Dobb's Journal 30(3)
2. Moore G (1965) Cramming More Components onto Integrated Circuits. Electronics 38(8), Available from Intel's homepage: ftp://download.intel.com/museum/Moores_Law/Articles-Press_Releases/Gordon_Moore_1965_Article.pdf. Accessed 11 January 2011
3. Agarwal A, Levy M (2007) The KILL Rule for Multicore. Design Automation Conference 2007: 750-753
4. Intel Corporation (2010) Petascale to Exascale: Extending Intel's HPC Commitment. http://download.intel.com/pressroom/archive/reference/ISC_2010_Skaugen_keynote.pdf. Accessed 11 January 2011
5. Amdahl G (1967) Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. American Federation of Information Processing Societies (AFIPS) Conference Proceedings 30:483-485
6. Gustafson J L (1988) Reevaluating Amdahl's Law. Communications of the ACM 31(5): 532-533. Online at http://www.scl.ameslab.gov/Publications/Gus/AmdahlsLaw/Amdahls.html. Accessed 11 January 2011
7. Mattson T (2010) Our Pattern Language (OPL). http://parlab.eecs.berkeley.edu/wiki/_media/patterns/opl_pattern_language-feb-13.pdf. Accessed 11 January 2011

Chapter 2
Multi-core and Many-core Processor Architectures

Abstract No book on programming would be complete without an overview of the hardware on which the software will execute. In this chapter we outline the main design principles and solutions applied when designing these chips, as well as the challenges facing the hardware industry, together with an outlook of promising technologies not yet in common practice. This chapter's main goal is to introduce the reader to the most important processor architecture concepts (core organization, interconnects, memory architectures, support for parallel programming etc) relevant in the context of multi-core processors as well the most common processor architectures available today. We also analyze the challenges faced by processor designs as the number of cores will continue scaling and the emerging technologies—such as transactional memory, support for speculative threading, novel interconnects, 3D stacking of memory etc—that will allow continued scaling of processors in terms of available computational power.

2.1 Overview

In 2001 the first general-purpose processor that featured multiple processing cores on the same CMOS die was released: the POWER4 processor of IBM. Since then, multi-core processors have become the norm and currently the only way to improve the performance for high-end processors is to add support for more threads, either in the number of cores, or through multithreading on cores to mask long-latency operations.

There are multiple reasons why the clock rate gains of the past cannot anymore be continued. The unsustainable level of power consumption implied by higher clock rates is just the most obvious and stringent reason; equally important is the fact that wire delays rather than transistor switching will be the dominant issue for each clock cycle.

With Contribution by Mats Brorsson

A. Vajda, *Programming Many-Core Chips,*
DOI 10.1007/978-1-4419-9739-5_2, © Springer Science+Business Media, LLC 2011

Compared to single-threaded processors, multi-core processors are highly diverse and the design space is enormous. In this chapter we present the architectural principles of multi-core chip architectures, discuss some current examples and point out the critical issues with respect to scalability.

2.2 Architectural Principles

Many of the technologies behind current multi-core architectures were developed during 1975–2000 for parallel supercomputers. For instance, directory-based cache coherence was published in 1978 by Censier and Featrier ([1]) who then referred to snoopy cache coherence (although it was not called so) as "the classical solution". The development of semiconductor technology has permitted the integration of these ideas onto one single-chip multiprocessor, most often called a multi-core processor. However, as always, the trade-offs become radically different when integrated onto the same silicon die.

In addition, recent advances in semiconductor technology have permitted greater control and flexibility through separate voltage and frequency islands which provide system software the possibility to control performance and adjust power consumption to the current needs.

This section is divided into two parts: *established concepts*, which focuses on the architectural ideas forming the basis of all multi-core architectures, one way or the other, and *emerging concepts* describing ideas that have only been tried experimentally or implemented to a limited extent in commercial processors.

2.2.1 Established Concepts

The very concept of multi-core architecture implies at least three aspects that will be the main topics of this chapter:

- There are multiple computational cores
- There is a way by which these cores communicate
- The processor cores have to communicate with the outside world.

2.2.1.1 Multiple Cores

The concept of multiple cores may seem trivial at first instance. However, as we will see in the section about scalability issues there are numerous tradeoffs to consider. First of all we need to consider whether the processor should be homogeneous or expose some heterogeneity. Most current general-purpose multi-core processors are homogeneous both in instruction set architecture and performance. This means

that the cores can execute the same binaries and that it does not really matter, from functional point of view, on which core a program runs. Recent multi-core architectures, however, allow for system software to control the clock frequency for each core individually in order to either save power or to temporarily boost single-thread performance.

Most of these homogeneous architectures also implement a shared global address space with full cache coherency (which we discuss later), so that from a software perspective, one cannot distinguish one core from the other even if the process (or thread) migrates during run-time.

By contrast, a heterogeneous architecture features at least two different kinds of cores that may differ in both the instruction set architecture (ISA) and functionality and performance. The most widespread example of a heterogeneous multi-core architecture is the Cell BE architecture, jointly developed by IBM, Sony and Toshiba [2] and used in areas such as gaming devices and computers targeting high performance computing.

A homogeneous architecture with shared global memory is undoubtedly easier to program for parallelism—that is when a program can make use of the whole core—than a heterogeneous architecture where the cores do not have the same instruction set. On the other hand, in the case of an application which naturally lends itself to be partitioned into long-lived threads of control with little or regular communication it makes sense to manually put the partitions onto cores that are specialized for that specific task.

Internally the organization of the cores can show major differences. All modern core designs are today *pipelined*, where instructions are decoded and executed in stages in order to improve on overall throughput, although instruction latency is the same or even increased. Most high-performance designs also have cores with speculative dynamic instruction scheduling done in hardware. These techniques increase the average number of instructions executed per clock cycle but because of limited instruction-level parallelism (ILP) in legacy applications and since these techniques tend to be both complicated and power-hungry as well as taking up valuable silicon real estate, they are of less importance with modern multi-core architectures. In fact, with some new architectures such as Intel's Knights Corner [3], the designers have reverted to simple single-issue in-order cores (although in the Knights Corner case augmented with powerful vector instructions, in order to reduce the silicon footprint and power consumption of each core).

A counter measure for limited instruction level parallelism added to the most advanced cores is simultaneous multithreading (SMT), perhaps better known by its Intel brand name *hyper-threading*. This is a hardware technique that allows better utilization of hardware resources where a multi-issue pipeline can select instructions from two or more threads. The benefits are that for applications with abundant ILP, single-thread performance is high, while with reduced ILP, thread-level parallelism can be utilized. Simultaneous multithreading has the nice property that it is relatively cheap in terms of area (the state for each thread and a simple thread selection mechanism) and extra power consumption.

2.2.1.2 Interconnection Networks

Having multiple cores on a chip requires inter-core communication mechanisms. The historical way that the individual processors in a shared memory multiprocessor communicate has been through a *common bus* shared by all processors. This is indeed also the case in the early multi-core processors from general purpose vendors such as AMD or Intel. In order not to flood the bus with memory and I/O traffic, there are local cache memories, typically one or two levels, between the processor and the bus. The shared bus also facilitates the implementation of cache coherency (more about that in the section about shared memory support below) as it is a broadcast communication medium which means that there is always a global order of shared memory operations.

More recent designs are based on the realization that shared communication mediums such as buses are problematic both in latency and bandwidth. A shared bus has long electrical wires and if there are several potential slave units—in a multi-core processor all cores and memory sub-system are both slaves and masters—the capacitive load on the bus makes it even slower. Furthermore, the fact that several units share the bus will fundamentally limit the bandwidth from each core.

Some popular general purpose multi-core processors at the time of this writing use a cross-bar interconnect between processor modules—which include one or two levels of cache memories—and the rest which includes the last-level cache and memory interfaces. Other technologies—such as multiple ring busses, switched on-chip networks—are emerging and gaining traction due to either lower power consumption, higher bandwidth or both. As the number of cores will increase, on-chip communication networks will increasingly face scalability and power constraints.

Figure 2.1 provides an overview of the most common designs of on-chip interconnect used in today's systems.

2.2.1.3 Memory Controllers

The memory interface is a crucial component of any high-performance processor and all the more so for a multi-core processor as it is a shared resource between all the cores on the chip. In recent high-end chips from both AMD and Intel, the memory controller was moved onto the chip and is separated from the I/O-interfaces in order to increase the memory bandwidth and to enable parallel access to both I/O devices and memory.

Of particular interest is the DRAM controller where the focus during the last few years has been to provide for increased throughput rather than low latency. DRAM request schedulers do not maintain a FIFO ordering of requests from processors to the DRAM. Instead, they try to combine accesses to the same DRAM page if possible, in order to best utilize open pages and avoid unnecessary switching of DRAM pages [4]. For a multi-core system, one can get both synergies as well as interferences in these aggressive DRAM schedulers [5].

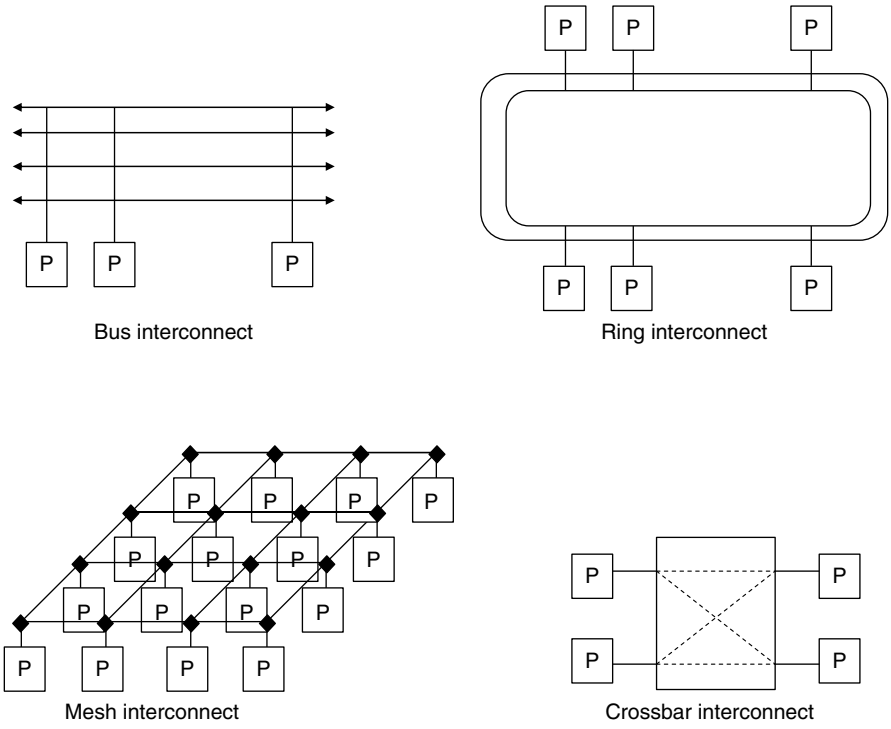


Fig. 2.1 Types of on-chip interconnect

One would expect synergies in the case of parallel programs that divide the work in a data-parallel fashion. In such programs, threads executing on different cores tend to work on the same instructions and with data located close to other threads' working-set and therefore the likelihood that they will access the same DRAM-pages is increased. On the other hand, when multi-core processors become many-core, we will see space-sharing in conjunction with the time-sharing that takes place in today's operating systems. This also happens if multi-core processors are used for throughput computing with multiple sequential applications rather than using them for parallelism. In these cases, the different applications will fight for the shared resources and may severely interfere with each other.

The memory controller is also the place where sophisticated pre-fetch mechanisms are usually implemented.

On the other hand, better utilization of increased memory bandwidth also leads to increased thermal development in the shared DRAM. Therefore, recent advances have brought forward both specialized DRAM DIMM modules for improved power consumption performance and memory controllers with DRAM throttling to reduce excessive temperature [6].

2.2.1.4 Shared Memory Support

With shared memory we mean that the processing cores can communicate with each other storing data in shared memory locations and subsequently reading them. The actual communication takes place over the interconnection network as discussed earlier.

A shared address space facilitates migration from a sequential programming model to a parallel one, in that data structures and control structures in many cases can be kept as those were in the sequential program. Programming models such as OpenMP [7] and Cilk [8] also allow for an incremental parallelization of the software, rather than the disruptive approach needed for a message-passing approach. That said, shared memory does not come for free, particularly when the core count goes up.

In the high-performance computing area, shared memory has generally not been used since its current implementations do not scale to the thousands or even hundreds of thousands of nodes used in the top-performing compute clusters today. These clusters, however, are built out of shared memory nodes and although programmers may or may not use shared programming models within a node, the hardware of these nodes typically implement a shared address space. The best performing programs in these machines typically use a hybrid programming model using message-passing between nodes and a shared memory within a node.

For these reasons, all general-purpose multi-core processors today support a shared address space between cores and maintain a *cache-coherent memory system*. By definition, a cache system is said to be coherent if and only if all processors, at any point in time, have a consistent view of what is the last globally written value to each location.

Over the time, various cache organization architectures have been proposed, relying on private, shared or mixed, flat or hierarchical cache structures. Figure 2.2 gives a schematic overview of various cache architectures.

The cache coherency mechanism allows processors fast access to commonly used data in their own private caches while still maintaining consistency when some other processor updates shared variables. The most common implementation uses invalidation mechanisms where local copies are invalidated if a core updates a shared variable. Figure 2.3 illustrates this basic principle of cache coherence through a simple example of access to shared memory.

In the example in Fig. 2.3 we have three cores, each with a private level 1 (L1) cache. For simplicity we only show the data cache, although the cores also typically have separate private instruction caches as well. There is also a shared level 2 cache unified for data and instructions. For now we assume that the cores and the L2 cache are interconnected through a shared bus.

Consider the case where both core 0 and core 1 have read the value of address A and thus both have a copy of A in their L1 caches. The need for cache coherence arises when either core 0 and 1 needs to update the value of A and the other core subsequently wants to read it. The example at the bottom of Fig. 2.3 illustrates

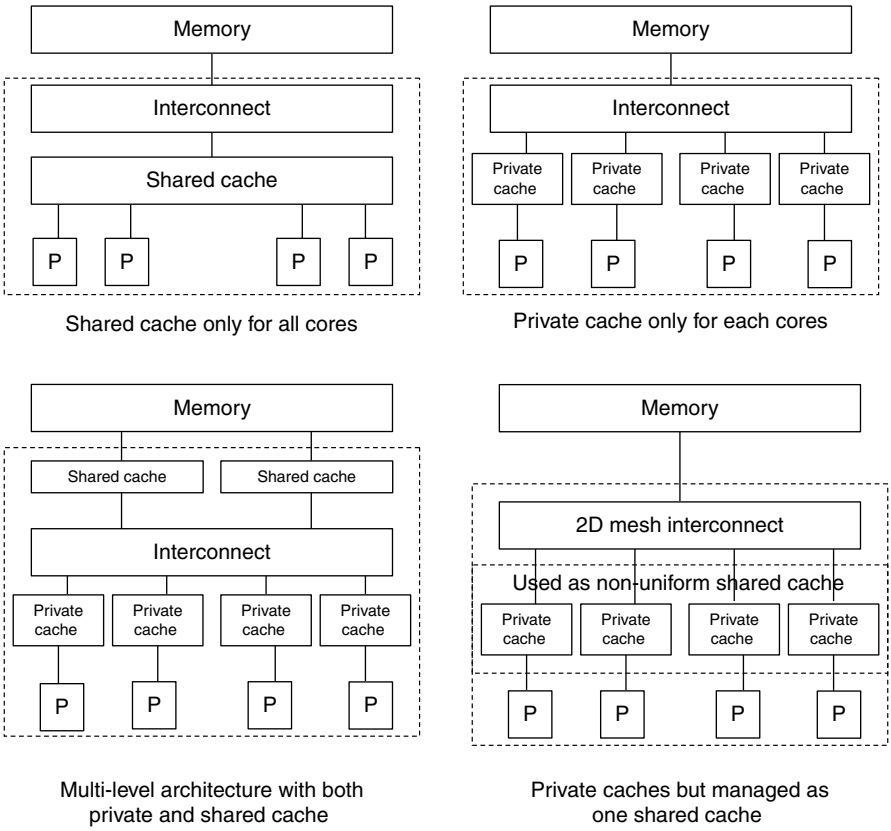
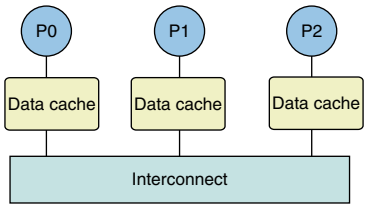


Fig. 2.2 Examples of cache architectures

a sequence of accesses to block A that will lead to some coherence actions to be performed.

The coherence actions taken on a cache block is governed by a set of state-machines. The different coherence protocols employed are often named after these states. An example of a commonly used cache coherence protocol is the MESI protocol, deployed in Intel processors. MESI stands for the four states a cache-block can be in according to below:

- **M**odified—A block in this state is the only valid copy of the block. The memory does not hold valid information and no other cache may have a valid copy. The core that owns this block can write to it without notifying any other core.
- **E**xclusive—The first core to read in a block from memory will load it into the Exclusive state. This means that if the same core later modifies the block, it can silently be upgraded to the modified state without notifying anyone else. This is beneficial for the vast majority of data which is not shared between threads or processes. If a block is in the exclusive state we know that



Initial state of block A is un-cached.

Time	P0	P1	P2	Comments
t1	Load A	Load A		A copy in both cache 0 and 1. The block is shared.
t2	Store A			The copy in cache 1 needs to be invalidated. The block is exclusive in cache 0.
t3	Load A			This load will hit in the cache and no coherence action is needed.
t4		Load A		The newly updated value in cache 0 is transferred to cache 1. The block is shared. Memory may or may not be updated depending on cache coherence protocol used.
t5			Load A	Block is now also cached in cache 2.
t6		Store A		The ownership is transferred to P1. The block is now exclusive in cache 1. The copies in both cache 0 and cache 2 are invalidated.

Fig. 2.3 Example of cache coherence in practice

  - the memory has an up-to-date copy
  - there are no other cached copies in the system.

- **S**hared—As soon as a second core is reading the same block, it will be loaded to the cache of that core and will be marked Shared in all caches
- **I**nvalid—As soon as one of the copies is modified by one of the cores, all other copies will be marked invalid and will need to be refreshed at the next access

MESI is just one of many cache coherence mechanisms proposed and implemented over time. Our goal is just to introduce the basic concepts; for a thorough overview of the area, we recommend one of the well-established books in computer architecture, such as Ref. [9] and [10].

The concept of cache coherence makes it easy to write efficient parallel programs using threaded or task-centric programming models. However, it also adds to the complexity of the memory hierarchy and power consumption. Clearly, for a parallel program to be scalable, the use of shared resources—be it memory locations, interconnects or memory interfaces—leads to performance bottlenecks. We will elaborate on this in the section dedicated to scalability issues later on.

2.2.1.5 Memory Consistency

The problem of consistent view of the memory is one of the fundamental problems that need to be tackled in any multi-core design. The existence of multiple copies of the same physical memory location—at various levels of caches but also within processor cores—requires a consistent and easy to understand model of how concurrent loads and stores are coordinated in order to maintain a consistent view of the content of the memory.

One of the important concepts related to memory consistency is *store atomicity*. In the presence of multiple copies of the same memory location, a store to that memory location needs to be propagated to all cores in *zero time*—something that's impossible to achieve in practice. However, the *appearance of instantly propagated store* can be achieved if a global order of store operations to the same memory location is enforced and a load (use of the value) is globally performed before the value it returns is used, meaning that the store providing the new value is performed with respect to *all cores* in the system.

The strictest memory consistency model available today is called *sequential consistency*. Formally, it is defined as follows:

> A multiprocessor is sequentially consistent if the result of any execution is the same as if the memory operations of all threads were executed in some sequential order and the operations of each individual thread appear in thread order.

Intuitively, it means that, while the order of memory accesses with respect to one core is maintained, accesses from different cores may be interleaved in any order. If one would perform a poll among programmers, this would be the most popular choice of memory model—but it requires significant implementation effort in hardware.

Several other models have been proposed over time, with weaker consistency requirements. Here are a few examples:

- allow a load to bypass a store within a core, if it addresses a different location in the memory
- allow stores issued from a core to be used by subsequent core-local loads even before it's globally visible
- rely on the use of atomic sections when accessing shared memory areas, thus enforcing mutual exclusion; all other loads and stores are performed without any consistency enforcement, as these are considered core local
- processors relying on out of order execution sometimes deploy speculative violations of memory orders: the core will *assume* that the values read or written will not conflict with updates from other cores; if this assumption turns out to be wrong, the operation is rolled back and re-executed; however significant performance gains can be achieved in case no conflicts are detected

While taken for granted by programmers, memory consistency is one of the most critical and complex issues to be considered when designing multi-core systems. Understanding the fundamentals of how memory access works is therefore essen-

tial when dealing with tricky concurrency bugs or implementing support for basic synchronization primitives.

2.2.1.6 Atomic Operations

As we will see further on in this book, *partitioning* into parallel tasks and *synchronization* between tasks are the fundamental activities that are required when designing software for many-core systems. *Synchronization* is very hard to realize in software alone, thus support in hardware is required; however, hardware only synchronization is usually hard to scale and has limited flexibility. Therefore the most common solutions rely on software with basic support from the hardware. We will look at the software solution in Chap. 5; here we will focus on the support available in hardware.

The mechanism that modern processors provide in order to support synchronization mechanisms are so called *read-modify-write (RMW)* or *conditional stores*. The basic principle behind these instructions is to provide the *smallest critical section* that guarantees conflict free access to a specific memory location that will contain the value used for synchronization. The most commonly used RMW instructions are listed below:

- *Test and set (T&S):* it reads a memory location, sets it to 1 and returns the value read in a register atomically (thus no other core can perform any store action on this memory location while the T&S instruction is being executed). Using this instruction, the mechanism to acquire/release a lock would look like this:

```
Locking:
   register = test_and_set (lock);
   while (register != 0)
         register = test_and_set (lock);
Unlocking:
   lock = 0;
```

- *Compare and swap (CAS):* atomically compares the value in the memory to a supplied value and, if these are equal, the content of the memory is swapped with the value stored in a register. Lock implementation using CAS looks like this (quite similar to T&S based implementation):

```
register = 1;
compare_and_swap (lock, 0, register); // if lock is 0, swap its value to 1
while (register != 0)
    compare_and_swap (lock, 0, register); // if lock is 0, swap its value to 1
```

- *Load-linked and store-conditional:* this is a de-coupled version of T&S, which is more pipeline-friendly. While there are two instructions in this case, the two are linked: after a load-linked, a store-conditional will only succeed if no other operations were executed on the accessed memory location since the execution

of the load—otherwise will set the register used in the store-conditional to 0. Here's how locking can be implemented with these mechanisms:

```
register = 0
while (register == 0) {
  dummy = load_linked (lock);
  if (dummy == 0) {
    register = 1;
    register = store_conditional(lock, register);
  }}
```

The presence of one of these ISA level constructs is essential for implementing higher level synchronization mechanisms. However, just one of these basic mechanisms is sufficient for most flavors of software synchronization primitives as well as for realization of lock-free data structures.

2.2.1.7 Hardware Multi-threading

As the capabilities and speed of cores kept improving at a more rapid pace than that of memory, the net result was that cores were idling for a significant share of the time while waiting on high latency memory operations to complete. This observation led to the implementation of *hardware multi-threading,* a mechanism through which a core could support multiple thread contexts in hardware (including program counter and register sets, but sharing e.g. the cache) and fast switching between hardware threads whenever some of the threads stalled due to high latency operations.

Various implementations of this concept have been provided over time, but most present systems use a fine-grained multithreading approach in the context of out-of-order cores called *simultaneous multi-threading.* In each cycle, instructions from different hardware threads are dispatched, or—in case of super-scalar processors where multiple instructions can be dispatched per core cycle—even instructions from different threads may be executed in parallel. As instructions from different threads use different registers, the latency of detecting where out-of-order execution can be performed is significantly reduced, leading to a higher density of instructions effectively executed per cycle. The goal of this technique is ultimately this: assuming that the software is multi-threaded, exploit it to mask the slowness of fetching data from memory and improve core utilization.

This technique has been implemented by most major vendors, including Intel (through the Hyper Threading Technologies (HTT) concept), IBM (which also supports thread priorities) and Oracle Sun (where as much as eight hardware threads are supported on each core).

Few years ago, there were arguments brought forward [11] that suggested that hardware threads actually consume more power than similar designs based on multiple cores, while also increasing the frequency of cache trashing. By 2010

however most chip vendors either already supported it or announced plans to do so.

The usefulness of this technology also has its limit however. Recently, the gap between the speed of memory access and speed of cores started to narrow due to the decline in processor core frequencies; therefore latency-hiding techniques will yield smaller benefits. Considering the performance gain per watt, it's likely that these techniques will be replaced by other mechanisms, such as increased cache sizes.

2.2.1.8 Multi-processor Interconnect

Multi-processor, cache coherent interconnect is an important building block for today's multi-processor, NUMA systems: it allows linking multiple processors together in order to provide a single logical processing unit. There are two major technologies found in today's systems:

- *HyperTransport,* a standardized low-latency packet-oriented point-to-point link supported by many major vendors (except Intel); its latest specification allows for speeds up to 51.2 Gbps/s, running at 3.2 GHz
- *Intel's QuickPath Interconnect (QPI),* used in most Intel chips

These types of interconnects are also used for connecting with I/O devices; however, in the context of this book, their importance lies in the possibility of creating more complex, higher core count logical processors from simpler building elements, with just a few cores.

2.2.1.9 Summary

In this section we introduced the well established concepts used in the design of the state of the art multi-core systems. There's a red line throughout these concepts that can be summarized in a few principles and assumptions:

- The only way forward is to start adding—albeit slowly—more cores to future chip designs
- These cores shall still be quite complex and high performance to support single threaded applications
- Shared memory on hardware level will still be required, even as we scale up the number of cores per chip
- The main issues to address are still memory access latencies, hence we need continued support for cache hierarchies and mechanisms that can improve efficiency of core usage

Only recently the community started questioning some of these principles and assumptions, which we will discuss later on in this chapter. It's therefore useful to look at some of the other ideas that are slowly making their way into the design of multi-core processors.

2.2.2 Emerging Concepts

As multi-core chips became ubiquitous, several new issues emerged that required new solutions, while some existing, well-known problems have found novel solutions. In this chapter we will look at some of these concepts.

The problem of *memory wall,* initially defined as the gap between the speed of processors and the speed of memory access, has slowly morphed into a different problem: the latency gap became smaller, but with the increase in the number of cores, the need for memory bandwidth has increased. Interconnects became the other issue that needed attention: existing solutions became either too power hungry as the transistor count went up (the case of bus or ring solutions) or led to higher latency (mesh networks), as the number of cores continues to increase. Finally, the larger number of cores starts to question the emphasis on core usage efficiency and ideas relying on using some of the cores (or at least hardware threads) as *helpers* have popped up recently.

On software side, as the number of cores increased, the cost of pessimistic, lock-based synchronization of access to shared memory got higher, prompting the search for new mechanisms, leading to the emergence of the concept of *transactional memory.*

2.2.2.1 Scouting and Helper Threads

The idea of scouting hardware threads was promoted by Sun as part of the design of their new processor called Rock (canceled since). The idea is to let the execution of a hardware thread continue even if normally it would stall due to e.g. a memory access: while waiting for the data to be fetched, the processor switches to a mode called *scouting* and would execute those instructions—out of order—that are not dependent on the result of the instruction that caused the stalling. Once the execution of the high latency instruction completes, the execution is re-synchronized and continued from the next instruction that was left out by the scouting process (due to data dependency or latency of execution). This technique could be augmented with speculative execution: the processor could speculate on the likely outcome of the stalled instruction and use this assumption for the scout thread's run-ahead execution.

The scout thread idea clearly targeted the traditional memory wall problem, trying to mask the latency of memory accesses. We believe that in this form and under this assumption (latency is the primary concern) it is a technique of diminishing returns; however the idea itself—using helper threads to speed up execution—has its merits and alternative use cases have been proposed.

Paper [12] proposes the usage of helper threads to improve the efficiency of cache usage: a separate core (or hardware thread) would monitor the memory traffic to and from a specific core, recording memory access patterns; using this information, whenever the same pattern is observed again, the helper core would initiate fetching of data from memory to cache, thus pre-loading the cache ahead of time. If the data is already in the cache, the helper core could make sure that it stays there and no unnecessary write-backs would occur. This technique tends to reduce both

latency, but also optimize memory bandwidth usage: if the prediction is correct, useful memory traffic is prioritized and unnecessary one can be avoided.

A similar idea is proposed in Ref. [13] to cache invalidation (trashing). In this case, the execution of the OS would be offloaded to a special core, so that the content of the cache relevant for the application would not be destroyed through the loading of the data needed by the OS. This idea is similar to the factored OS principle that we will present in Chap. 7.

Thread Level Speculation

Thread level speculation support in hardware is quite similar to the scouting thread concept: the processor is capable of performing run-ahead execution on certain branches, using private copies of data and registers, at the end either validating or discarding the result. This idea may get more traction as the number of cores will continue to increase: some of the cores can be used to perform speculative execution on behalf of the main branch of execution. Such an approach suits well heterogeneous single-ISA multi-core architectures: the main thread of execution is placed on the high-capability core, while the simpler cores will be responsible for low power speculative pre-execution.

These ideas have not yet made their way into actual products, but as we will see later on in this book, such an approach could provide a way forward for scaling hard to parallelize applications on many-core processors.

2.2.2.2 Memory-centric Architecture

It's becoming clearer that memory bandwidth rather than memory latency is the next bottleneck that needs to be addressed in future chips. Consequently we have seen a dramatic increase in the size of on-chip caches, a trend that is likely to continue as the amount of logical gates on a chip will keep increasing for some time to come, still following Moore's law. There are however several other directions based on novel technologies that are currently pursued in the quest of improving memory bandwidth:

- Embedded DRAM, already in production in commercial chips such as IBM's Power7 processor
- Use of 3D stacking of memory with short, high bandwidth links called vias between processor cores and memory
- Use of memristors, perhaps the most radical departure from today's designs

Embedded DRAM

DRAM has the benefit of having much higher density and lower power consumption than the traditional SRAM technology used for on-chip memory, thus integrating it on the same silicon with the processing logic holds the promise of higher

on-chip memory density and higher bandwidth access than through external interfaces. Novel developments in CMOS technology allow today the usage of the same process to manufacture both memory and processing units.

Perhaps the most well known usage of embedded DRAM is in IBM's Power7 processor, packing a 32 Mb L3 cache built using eDRAM; however the technology has now been deployed in various gaming and embedded processors as well.

3D Memory Stacking

Packaging memory on top of the processor cores allows the creation of very high speed, dense interconnects (*vias*) between the two layers, achieving two goals at the same time: dramatically increasing the amount available memory (to the order of gigabytes) with access speeds comparable to cache access today.

Beside practical implementation of layering itself, this technology has a number of challenges related to cooling the processor cores hidden under the memory layer. The concept is still in research phase with a number of prototyping activities ongoing at universities and industrial research laboratories. If realized, it will allow a dramatic improvement in memory performance.

Memristor-based Technologies

The concept and theory of *memristor* (short for memory resistor) has been formulated almost four decades ago [14], but only as recently as 2008 the first practical realization was announced by HP Labs. In short, it's a circuit element with "memory": the resistance is a function of past current, in other words, a memristor can remember a voltage applied to it even after the current is cut off. This property makes it a prime candidate for building more compact and faster storage devices, while computer designs based on memristors replacing transistors have also been proposed. If it will materialize, it will represent a unified design base with significant possibilities: 3D designs based on memristors with up to 1,000 layers have been designed.

Memristor based commercial processors and memory systems are still some time—perhaps a decade—off, but currently this is one of the most promising basic technologies in the quest for improving memory performance.

2.2.2.3 Future Interconnects

As the number of cores on a chip will continue to increase, the on-chip interconnect is quickly becoming the bottleneck, both in terms of throughput and power consumption. This is especially true for bus and ring based interconnects: the power needed to drive high-speed networks across long wires is quickly becoming the largest contributor to overall power consumption, outstripping all other components (such as cores and peripherals). Mesh/switch based interconnects, due to shorter wiring were able to keep power at manageable levels; however, as the size of the

mesh will continue to increase, the latency and delay of transporting data across the chip will become the limiting issue.

There are two significant new ideas that aim at tackling the power, latency and delay related issues with the on-chip interconnect: 3D stacking coupled with mesh interconnect and optical interconnects. 3D stacking of cores, not just memory would help limit the number of hops that would need to be used to interconnect the cores. Consider for example the case of a 1,024 core machine: in a 2D mesh of 32×32, the cost of reaching across the chip is 62 hops (31 in both directions); if it's placed in a 3D configuration with 4 layer of 256 cores, this value drops to only 33 hops (3 to cross the layers and maximum of 30 in the lower layer), a factor of almost 3× improvement. Obviously, 3D stacking of cores has to solve the same issues that have so far hampered efforts of stacking memory on top of processor cores: cooling and heat removal from the lower layers.

Optical On-chip Interconnect

Optical on-chip networks are in the focus of recent research efforts at both universities and chip manufacturing companies such as IBM or Intel. Besides offering higher speeds, having lower power consumption and occupying less chip real-estate, optical interconnects have the possibility to span multiple physical chips, thus creating one logical chip from smaller physical chips.

Optical interconnect solutions are likely to use *Wavelength Division Multiplexing (WDM)* as underlying transmission technology, which allows a single optical waveguide to simultaneously carry multiple independent signals on different wavelengths. Optical signals use significantly less power because of the relatively low loss even across long distances.

One interesting approach was put forward by researchers at MIT [15], illustrated in Fig. 2.4. Their approach is to use a broadcast optical interconnect (ONet in the figure) between islands of several cores (16 each, in their largest configuration) and rely on island-local electric broadcast of information received over the optical network. In addition, an island-local electric mesh network would be preserved for local communication (*ANet* in the figure). This hierarchical architecture helps simplifying the architecture of each network: the optical network would have only 64 endpoints in the largest (1,024 cores) architecture while the electric networks would be a 16-way star and 4×4 mesh network, respectively.

On-chip optical interconnects have yet to make their ways into commercial chips. However, in our assessment, this technology represents the best bet for improving performance of on-chip interconnects.

2.2.2.4 Transactional Memory

Transactional memory emerged as an alternative to traditional locking based synchronization. We will discuss it in more details in Chap. 5; here we will focus on the hardware aspects of its realization.

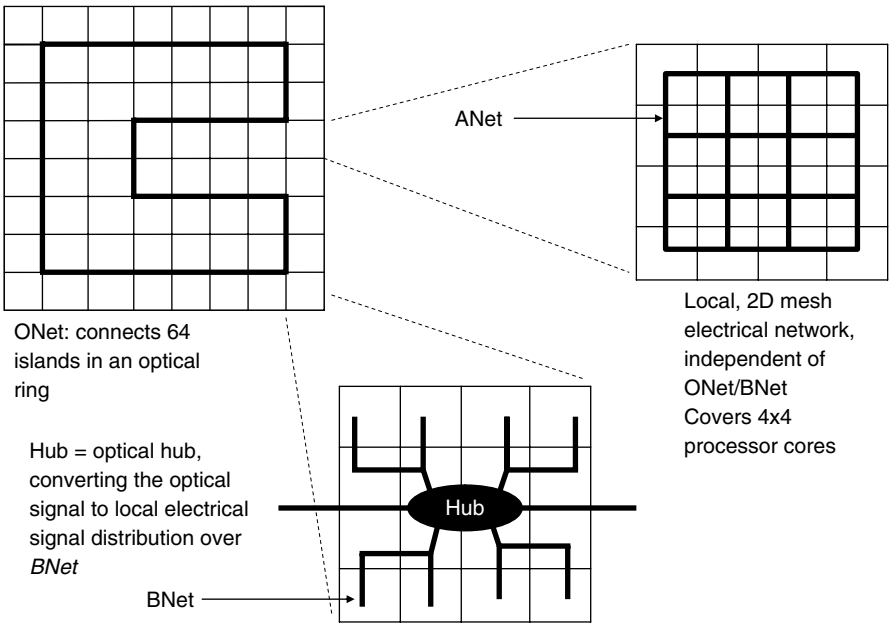


Fig. 2.4 Architecture of a combined optical/electrical on-chip interconnect

In short, transactional memory is a mechanism to enable optimistic concurrent updating of shared memory locations: a thread is allowed to progress with its own updates until it completes the sequence of actions (called *transactions*: the changes performed during a transaction are not visible to the rest of the system); at that point, it is checked if any other concurrent accesses took place on the same memory locations: if not, all the changes are *atomically committed* and made visible to other cores/threads; if the memory areas modified by the transaction have been modified elsewhere, the complete transaction is *rolled back* and re-attempted. All these mechanisms—private updates, check for concurrent access, commit/rollback/retry—are transparent to the application and handled by the run-time system. Transactional memory is most useful in situations where the probability of contention is very low.

There are three ways to implement the concept of transactional memory: in hardware only (called Hardware Transactional Memory—HTM), in software only (Software Transactional Memory—STM) or a hybrid solution, combining both hardware and software solutions.

Hardware transactional memory requires significant changes to the memory management and thread management system of the processor. Each core will have a new state—the *transactional state*—in which handling of the active thread and the memory will be different. The ISA of the processor shall support new commands for indicating the start and end of a transaction.

How will hardware transactions work?

At the beginning of a transaction, the processor has to checkpoint the state of the executing thread, i.e., the content of the registers and the memory. Registers are

easy to manage by supporting two register files: the "regular" one and the "transactional" one; during transactions, a copy of the regular one is created in the transactional one which is then either committed (copied back to the regular one) or discarded (in case of rollback) at the end of transactions.

There are multiple solutions proposed for handling memory during transactions. Here we will describe a simple method, based on a mechanism called *transactional cache state.* Basically, there shall be two copies of the blocks being edited: the version before the transaction (the stable copy) and the one being modified by the transaction (the transactional copy). The simplest way to manage the two copies is to hold the transactional copy in the L1 cache and the stable copy in L2 cache. This will require a new state of the block (beside the MESI ones): *Transactional,* indicating that the block is being modified as part of an ongoing transaction and shall not be shared with other cores. With the method described here, this new state is needed for blocks cached in L1 cache.

There are two methods to detect conflicting updates to the same shared memory: the eager method detects the conflict and aborts one of the transactions immediately; the lazy method does so only at the end of transaction. Using the modified MESI coherence protocol, the eager implementation is the more handier one: when creating the L1 copy of the block, the core will be notified if it's already under editing on an other core; in this case it can immediately abort the transaction (and retry later on).

One of the major issues with this method based on L1 caching of transactional blocks is the limited size of the L1 cache: for long running transactions or transactions that modify a lot of data, the size of the L1 cache may become insufficient to hold all transactional blocks. There's no easy way around this limitation—hence keeping the size of transactions (both time- and space-wise) short is a pre-requisite for this method to work.

Despite extensive research—the first paper on transactional memory was published already back in 1993 [16]—transactional memory support has yet to become part of modern processors. Many of the major players in the industry chose to rely on STM implementations first—the only processor that was planned to support hardware transactional memory was the now canceled Rock processor from Sun.

2.3 Scalability Issues for Many-core Processors

So far in this chapter we looked at the current well established technologies used in building multi-core processors as well as at the emerging technologies that are targeting some of the shortcomings of current technologies.

On manufacturing technology level, major chip manufacturers estimate that we will be able to continue with current CMOS technologies down to approximately the 6 nm manufacturing process (as of 2010, the most advanced technology node is the 32 nm node). However at that level at least two major challenges will emerge:

- As the size of transistors will be measured in just a few tens of atoms at most, quantum effects will have to be taken into account and consequently we will see

an increased *unreliability of the hardware*, with components failing more often and—more importantly—intermittently
- There will be so many transistors on the chip that it will be, power wise, impossible to switch all of these at the same time; this phenomenon—called the *dark silicon* problem—will have a significant impact on how we will build future processors

The unreliability of future hardware will likely lead to the implementation of redundant execution mechanisms. Multiple cores will perform the same computation, in order to increase the probability that at least one will succeed; in some cases a voting scheme on the result (verifying if all the computations yielded the same result) may be used to guarantee correctness of the calculation. Such mechanisms will likely be invisible to the software, but will impact the complexity of the design of logical processor cores.

The dark silicon problem is trickier to address. By lowering the frequency at which chips operate, we can push the limit further [17], but eventually it will become an issue, no matter how low we go with the frequency. Alternative solutions include the adoption of memristors as building blocks (instead of transistors) and chip designs where, at any given time, just a subset of components would be active, depending on the type of application. Optical interconnects within and between chips could also allow building smaller chips, interconnected to build larger, cache coherent logical chips.

Another scalability bottleneck relates to the design of cache coherency protocols. Synchronizing access to the same memory area from large number of cores will increase the complexity of coherency design and will lead to increasing delay and latency in accessing frequently modified memory areas. In our view, it will be an uphill battle to maintain a coherent view across hundreds, let alone a thousand cores; even if we will be able to do this, it will be hard to justify the cost associated with it.

Memory bandwidth will be another scalability bottleneck. The leveling out of the core frequency will lead to reduced latency, but the increase in the number of cores will multiply the amount of data required by the cores and thus the aggregate memory bandwidth that future chips will require. If we will indeed see the continuation of Moore's law, translated into an ever-increasing number of cores—perhaps following the same trend of doubling core-count every two years—a similar trend would need to be followed by memory bandwidth, something the industry failed to deliver in the past.

Based on these observed or predicted developments and bottlenecks, we believe the following trends will dominate the design of future processors:

- Shift towards *simple, low frequency, low complexity* cores, coupled with an increase of the core count to the level of several hundreds within five to ten years; heterogeneity—not in ISA, but rather in core capabilities—will play a role simply because it's an easy optimization gain
- Focus on *novel memory technologies* that can deliver higher bandwidth access; technologies such as 3D stacking, optical interconnects and perhaps memristors

will see an accelerated uptake by mainstream processor designs; especially optical interconnects have the potential of easing some of the pressure on how chips are structured and interconnected
- The size of *on-chip memory* will also see a dramatic increase and we will see innovations emerging that will reduce the footprint, power consumption and complexity of designing such solutions, similar to the development of the embedded DRAM technology; once again 3D stacking and memristors may be some of the technologies to watch
- *HW accelerators* will be abundant: these have low footprint and low power consumption, thus we will see realizations in HW of an increasing array of algorithms
- *Aggressive power optimization mechanisms*—near threshold operation, power gating, frequency and voltage scaling—will be pervasive not only in traditionally low power domains, but also in most areas where processors are used

Some of these predictions may fade away, but, in the absence of a revolutionary new method of designing processors, increasing core count, heterogeneity and reliance on aggressive power optimization methods will likely dominate the chip industry for the coming five to ten years.

2.4 Examples of Multi-core Processors

After following the path of predictable design—faster, more complex, single-core RISC (Reduced Instruction Set Computer) or CISC (Complex Instruction Set Computer) machines—for about 30 years, the chip industry has seen an explosion of wildly differing designs over the past seven years. If anything, this trend will continue well into the second decade of our century—thus any attempt to survey the landscape of representative designs is likely to lead to outdated results very quickly.

Therefore in this chapter we will try to showcase the major trends followed by processor vendors rather than individual incarnations of certain processor families; whenever we delve into the details of a specific processor, the goal is to highlight and analyze a certain design choice, rather than upholding that specific version of the chip.

There are essentially three different philosophies followed by multi-core chip designs:

- fewer, but more complex, high performance cores with large, shared on-chip caches and cache coherence mechanisms; this design is represented by most server chip vendors (IBM, Intel, AMD and SUN to some extent) but also companies focusing on the mobile and low power design space
- large number of simple, low frequency, sometimes specialized cores with distributed on-chip memories (with or without cache coherence mechanisms) and scalable interconnects; these chips usually target data-parallel applications such

as graphics processing or networking, with the notable exception of Tilera which targets a broad range of applications, including data-centers
- integration of multiple cores of different capabilities and potentially supporting different instruction set architectures

The main argument for the first type of design is support for single-threaded applications. Multi-core is regarded a "necessary evil"—the next best thing once performance gains through higher frequencies could not be sustained. On the other hand, the advocates of the second approach rely on the argument that overall performance per watt will be higher using simpler but more cores. The third type of design usually targets specific domains where some measure of performance—overall, per watt or per real estate—dominates all other factors.

We tend to agree with the second line of thinking—the first type of approach is limited to just incremental improvements; without novel programming models that would allow running even single threaded applications on multiple cores, this approach will eventually not be sufficient.

2.4.1 Processors Based on Low Number of Cores

In this chapter we will cover the most representative chip families where the single-core performance still is the primary design concern: Intel's server processors (based on the Nehalem microarchitecture), IBM's Power series of processors, Oracle/Sun's SPARC ISA-based processors as well as designs based on ARM's Cortex-A15 core design.

2.4.1.1 Intel's Server Processors

Intel's current (as of 2010) line-up of 64 bit server processors relies on the *Nehalem* micro-architecture, first introduced in 2008 and revised in early 2010 (under the code-name *Westmere*). The successor of this microarchitecture will be released in 2011, under the code name *Sandy Bridge*; the major change will be the integration of dedicated graphics cores. Today chips based on this micro-architecture are manufactured using 45 and 32 nm processes (which will be used for the *Sandy Bridge* line as well, at least initially).

The main features of this line of processors include:

- Native support for up to 8 cores/die (typical configurations have 4 or 6 cores)
- Integrated memory controller
- Support for shared L3 (missing from previous generations) with sizes up to 12 Mb, in addition to per core L1 and L2 caches
- Support for virtualization (dedicated protection ring)
- Support for Intel's Turbo Boost, Hyper-Threading, Trusted Execution, and SpeedStep technologies

- Core speeds of up to 3.33 GHz, with a maximum turbo frequency of about 3.6 GHz
- Maximum power consumption (TDP—thermal design power) of 130 W, including on-chip, but off-processor GPU cores, with 6 processor cores running at 3.33 GHz.

Clearly this family of processors is targeting server workloads with special emphasis on single-threaded applications and power management. From technology point of view it's important to understand the background of different technologies supported by Intel's server chips, the focus of the following sub-sections.

Turbo Boost

Turbo boost is Intel's implementation of dynamic frequency scaling. This technology allows cores to run faster than the base operating frequency, if the overall power and temperature specifications allow it. It must be explicitly activated, but the actual operating level is determined by three factors:

- Number of active cores
- Estimated power and energy consumption
- Processor temperature

The operating frequency of the core is raised stepwise, with 133 MHz at a time, at regular intervals. The number of steps depends on the conditions (especially the number of active cores) and the base frequency, the maximum range is typically 10–15% compared with the core base frequency.

Hyper Threading

The hyper-threading technology is essentially the implementation of hardware threading, provided by Intel all the way from the low power Atom processor till the top tier server chips. The support is limited however to only two threads per core. There's no priority assignment to threads and the hardware will make sure that no thread gets starved.

Trusted Execution

Intel's trusted execution technology provides hardware support for enhanced security of program execution. Specifically, it allows for

- Creation of multiple, isolated execution environments (partitions) that are guaranteed to be tamper proof by any other application running in an other partition
- Secure key generation and storage
- Remote attestation of the security level of the machine

These features are usually realized as a combination of processor features and support in chipset and firmware.

2.4.1.2 IBM's Power Processors

IBM's line of POWER processors—now at its 7th generation—is based on the Power Architecture instruction set architecture, driven primarily by IBM and Freescale, within the Power.org industrial consortium. The ISA has a RISC architecture and it's open for licensing. Despite the pervasive nature of the X86 ISA (on which Intel and AMD processors are based) in the server space, roughly half of the Top 50 supercomputers are based on processors using the Power Architecture (all built by IBM). Additionally, the Power architecture is available in many specialized chips, including almost all game console processors.

The main characteristics of the Power 7 family of processors are:

- 4, 6 or 8 cores per chip, with execution frequencies exceeding 4 GHz; a more conventional limit is around 3–3.5 GHz—which still puts power consumption over the 200 W bar
- 4-way SMT, resulting in 32-way SMT per processor, with aggressive out of order execution
- 32 Mb on chip, embedded DRAM based shared L3 cache, on top of the 64 kb/core L1 and 256 kb/core L2 cache, tightly coupled with the cores
- Scalability up to 32 sockets, with "near linear" performance scaling claimed by IBM
- Advanced power optimization designs
- Distributive resource management which allows re-allocation of resources (cache and external memory bandwidth) between cores, depending on the application that is being executed

Power7 processors stand out—in our opinion—with two design choices: the usage of eDRAM (which we have discussed elsewhere in this chapter) as basis for L3 cache and the advanced power management features.

The cores in the Power7 processors support two idle modes:

- *nap mode*, optimized for wake-up time: clocks are turned off towards the execution units, frequency is reduced, but caches and virtual memory management tables remain coherent, thus the core can be brought back to full speed quicker
- *sleep mode,* optimized for power reduction: clocks are fully turned off, caches are purged and voltage reduced to a level where leakage current is substantially lowered; however, at wake-up still no re-initialization of the core is required

The processors also support active energy management, commercially called Energy Scale, consisting of the following technologies:

- DVFS (dynamic voltage and frequency scaling) within the range of −50% to +10% of the nominal core frequency

- Turbo mode, similar to Intel's solution
- Power budgeting for different parts of the system: performance will be optimized within pre-configured power limits

The Power 7 family of processors is a fine example of the balancing act server chip vendors have to perform in order to deliver chips that can support both single-threaded as well as multi-threaded workloads. Features such as high clock rate (the highest of any chip delivered during 2010), turbo mode, distributive resource management are geared towards supporting single-threaded performance; DVFS (Dynamic Voltage and Frequency Scaling), the relatively large number of cores and the distributed cache architecture have as a primary goal support for parallel scalability.

2.4.1.3 SPARC Processors

Beside the two X86-based vendors and IBM, Sun Microsystems (acquired by Oracle) was one of the long-standing leaders in the area of server chips and high end servers, with designs based on the SPARC instruction set architecture, initially developed at Sun. Recently however Sun also diversified itself into supporting X86 based architectures.

SPARC (Scalable Processor Architecture) is an open, RISC type of instruction set architecture, licensable in a similar manner as the Power specification; several of the processors designs based on SPARC have actually been released under open source license (such as OpenSPARC T1 and OpenSPARC T2). One of the SPARC architecture based processors also serves as the baseline for the widely used SPEC (Standard Performance Evaluation Corporation) CPU benchmarks: all benchmark results represent a relative speed compared to that of the basic SPARC processor; for more details on the SPEC benchmarks [18]. As of 2010, there are essentially only two large vendors providing chips based on the SPARC specification: Fujitsu and Oracle/Sun.

An interesting feature of the SPARC ISA is the support for *register windows*. The processor may have up to 160 general purpose registers, but at any time only 24 of these are visible to the software; whenever a sub-routine call is performed, the register window is shifted by 8, so that the new procedure gets 8 local registers and shares 8 registers with its caller as well as another 8 with any other sub-routine it may call.

The latest processor released by Sun, based on the SPARC version 9 architecture is the SPARC T3, code-named Rainbow Falls or Niagara 3 (all UltraSPARC processors used the Niagara code-name—the tag *Ultra* was dropped in this case). It's the server chip with most parallelism: it contains 16 cores running at 1.65 GHz, each with support for 8 hardware threads per core, thus the chip supports 128-way SMT. The schematic layout of the chip is shown in Fig. 2.5.

What sets the SPARC T3 apart is its on chip and off chip interconnect solution. The basic connectivity between cores and L2 caches is through a cross-bar type of

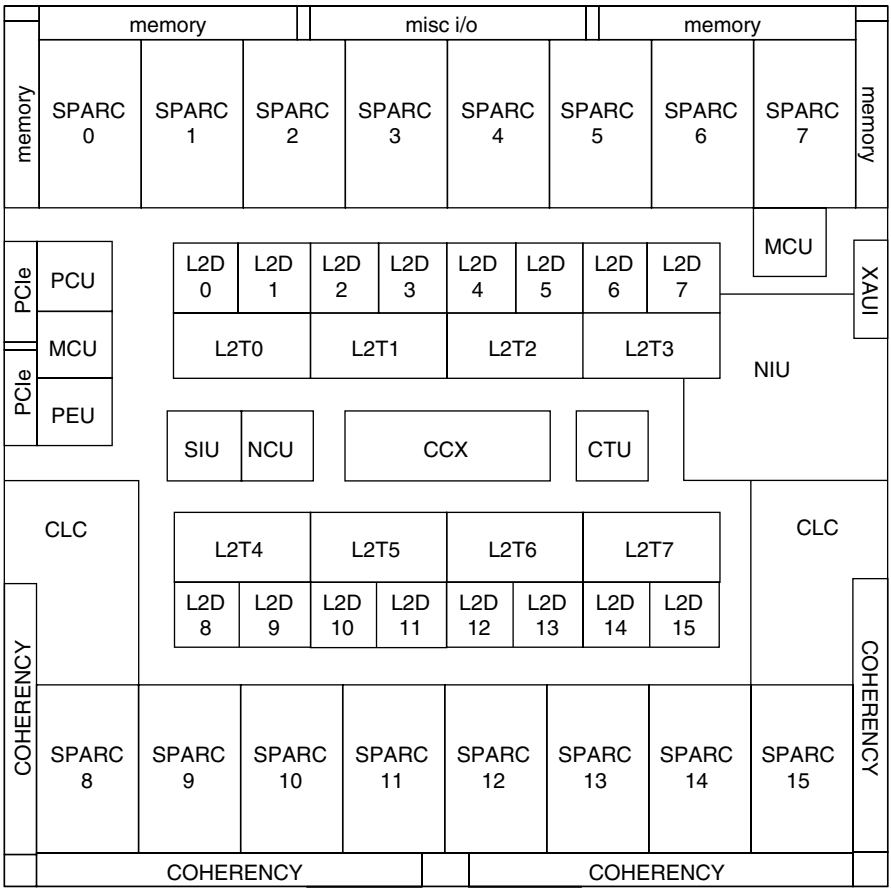


Fig. 2.5 SPARC T3 architecture. (Source: Wikipedia, under Creative Commons Attribution 3.0 license)

interconnect that connects the cores to a total of 6 Mb of L2 cache (organized into 16 banks). On the external side, there's a wide array of connectivity options: two PCIe interfaces, 2 Ethernet interfaces with a bandwidth of 10 Gbps each and 6 coherence links per core, each with a bandwidth of 9.6 Gbps. This architecture allows gluing together, without any additional hardware, four T3 chips in a cache coherent, SMP structure—resulting in a system with 64 cores and 1,024-way simultaneous multi-threading support.

The SPARC T3, while clearly a server chip, is targeted to a specific type of workload: large number of relatively simple parallel tasks that require a lot of traffic, typical for web servers and database systems (it's not surprising that offerings based on T3 are tightly integrated with Oracle's database). Its focus is a different market from Intel's and IBM's, whose chips are primarily geared towards more computationally intensive applications with less inherent parallelism.

2.4.1.4 ARM-based Multi-Core Processors

If X86 and Power architecture based systems dominate the server and desktop space, the mobile and low power computing space has its own pervasively present player: most mobile phones and a lot of embedded devices today use a chipset based on the ARM architecture, licensed by the UK based company of the same name.

ARM itself does not develop chips: its business relies on designing and licensing aggressively power optimized core designs that can then be integrated into actual chips by one of its licensees. Consequently, there's a large variety of chips based on ARM cores, all sharing the same basic ISA and core design.

Traditionally ARM cores were simple, low power designs unsuitable for desktop, let alone server usage; recently however ARM entered a new territory with its Cortex A9 and A15 designs that added capabilities previously only seen in high end desktop and server products. For this reason we include ARM into the category of chips with fewer, yet more complex cores.

ARM Cortex A15 Architecture

The ARM ISA is a 32 bit RISC architecture with fixed instruction width and mostly single cycle execution. It features some more peculiar features such as support for conditional execution of instructions (the result of the last comparison can be used as conditional for subsequent instructions) and support for folding bit-shifting instructions into arithmetic ones. It has been extended with optional ISAs covering floating point operations, SIMD instructions and native execution of Java byte-code (called *Jazelle*).

The Cortex A15 core design (announced in 2010, expected to ship in 2012) integrates all these features into an out-of-order, speculative issue, superscalar architecture with cores clocked between 1 and 2.5 GHz. The licensable solution will support up to 4 cores in a cluster, with the possibility to link two clusters into one cache coherent SMP system. For the first time for ARM, A15 has virtualization support (with a new protection ring) and support for addressing more than 4 Gb of memory (the address space is extended to 40 bits). The cache architecture will feature 64 kb L1 cache and up to 4 Mb shared L2 cache for the 4-core cluster.

It's yet unclear how chips based on the A15 core design will measure up against competing products. ARM claims a significant improvement compared to the previous generation and academic evaluations have shown Cortex A9 (the predecessor of A15) based chips to outperform—in terms of performance per watt—competing Intel server chips by a factor of about 5–7× ([19]). It all depends however on the type of the application: ARM based designs will likely perform much better with tasks that are highly parallel and do not require execution of lengthy single-threaded code.

The main differentiating factor for ARM based processors is power efficiency. While other vendors are trying to retrofit their high-performance designs with power efficient solutions, ARM cores are trying to ramp up performance on a basic

design that is eminently power efficient. So far, for embarrassingly parallel applications, the second approach seems to be gaining the upper hand.

2.4.2 Processors Based on Large Number of Cores

When looking at raw performance alone (total number of instructions executed within a unit of time), more but less powerful cores clearly outperform chips with few but powerful cores, within a set power budget. This is a tempting prospect, especially in domains with abundant parallelism such as networking, graphics, web servers etc. Accordingly, this category of chips—with many, but simpler cores—is usually represented by processors targeting a specific domain.

In this chapter we survey some of the best known representatives of this category: Tilera's Tile GX family, NVIDIA's Graphics Processing Units (GPUs), picoChip's 200-core DSP as well as Intel's recently announced Many Integrated Core (MIC) architecture.

2.4.2.1 The Tile Architecture

The Tile architecture has its origins in the RAW research processor developed at MIT and later commercialized by Tilera, a start-up founded by the original research group. Chips from the second generation are expected to scale up to 100 cores based on the MIPS ISA and running at 1.5 GHz, within a power budget of under 60 W.

The key differentiating technology of the Tile architecture is the on-chip interconnect and the cache architecture. As the name implies, the chip is structured as a 2D array of *tiles,* where each tile contains a processor core, associated L1 (64 kb) and L2 cache (256 kb per core) and an interconnect switch that can connect the tile to its 3 (on the edges) or 4 (inside the mesh) neighboring tiles. This way, the interconnect is essentially a switched network with very short wires connecting neighboring tiles linked through the tile-local switch.

The interconnect network—called *iMesh* by Tilera—actually consists of five different networks, used for various purposes:

- application process communication (UDN)
- I/O communication (IDN)
- memory communication (MDN)
- cache coherency (TDN)
- static, channelized communication (STN)

The latency of data transfer on the network is 1–2 cycle/tile, depending on whether there's a direction change or not at the tile. The overall architecture of the Tile processor concept is shown in Fig. 2.6

This network based architecture allows for some innovative solutions. The network of L2 caches is organized into a non-uniformly structured L3 cache, using a

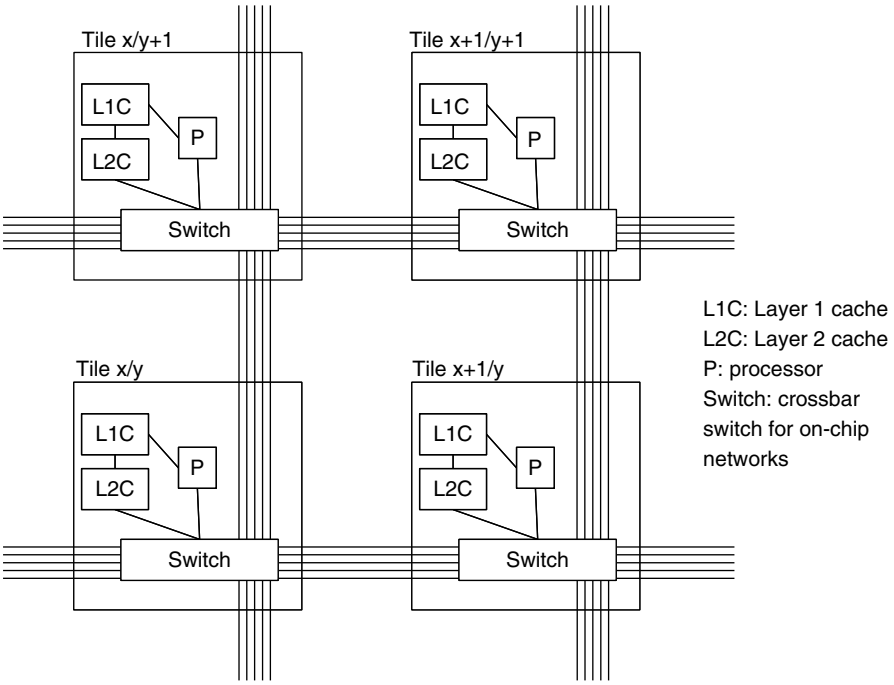


Fig. 2.6 The Tile architecture

directory-based coherence mechanism and the concept of *home tile* (the tile that holds the master copy) for cached data. While the access latency to various parts of this virtual L3 cache varies according to the distance between tiles, this mechanism provides an efficient (space and power wise) logical view to the programmer: a large on-chip cache to which all cores are connected. The TDN network is exclusively dedicated to the implementation of the coherency protocol.

Another mechanism implemented in the Tilera architecture using the communication networks is the *TileDirect* technology that allows data received over the external interfaces to be placed directly into the tile-local memory, thus bypassing the external DDR memory and reducing memory traffic.

In our view, the scalable, low power, high bandwidth on-chip mesh interconnect as well as the mechanism that allows cache coherence on such scale (up to 100 cores) are the technologies that make the Tile processor concept unique and clearly differentiates it from other chip designs.

2.4.2.2 Graphics Processing Units

The term Graphics Processing Unit (GPU) was first used back in 1999 by NVIDIA. The introduction of the OpenGL language and Microsoft's DirectX specification resulted in more programmability of the graphics rendering process; eventually this evolution led to the introduction of a GPU architecture (by the same com-

pany, NVIDIA) that dramatically improved programmability of these chips using high level (C like) languages and turned this type of processors into an interesting choice for supercomputers targeting massively data parallel applications. NVIDIA's CUDA (Computer Unified Device Architecture) programming model was the first that enabled high level programming of GPUs; more recently the OpenCL language (which we will also cover in this book) is set to become the de facto standard for data-parallel computing in general and for programming GPUs in particular (NVIDIA itself adopted it as a layer above CUDA). On hardware side, the most radical new chip design to date by NVIDIA is the Fermi family of chips.

The Fermi architecture crams an impressive 3 billion transistors into a system with 512 CUDA cores running at 700 MHz each. A CUDA core has a pipelined 32 bit integer arithmetic logic unit (ALU) and floating point unit (FPU) with fused multiply-add support, being capable to execute an integer or floating point operation in every cycle. The CUDA cores are grouped into 16 *streaming multi-processors,* each featuring 32 CUDA cores and 16 load/store units allowing memory access operations for 16 threads per each clock cycle. There are also four special function units (SFU) that can calculate values of functions such as sin or cosine. For a schematic view of the SM architecture, see Fig. 2.7

Regarding memory, the chip supports 6 GB of ECC protected DRAM memory. For the first time, the GPU has support for cache coherency; each SM has 64 kb local memory that can be configured in a 16/48 split as cache and local store; there's also 768 kb of shared L2 cache. The chip also has a unified address space spanning both private and global memory addresses.

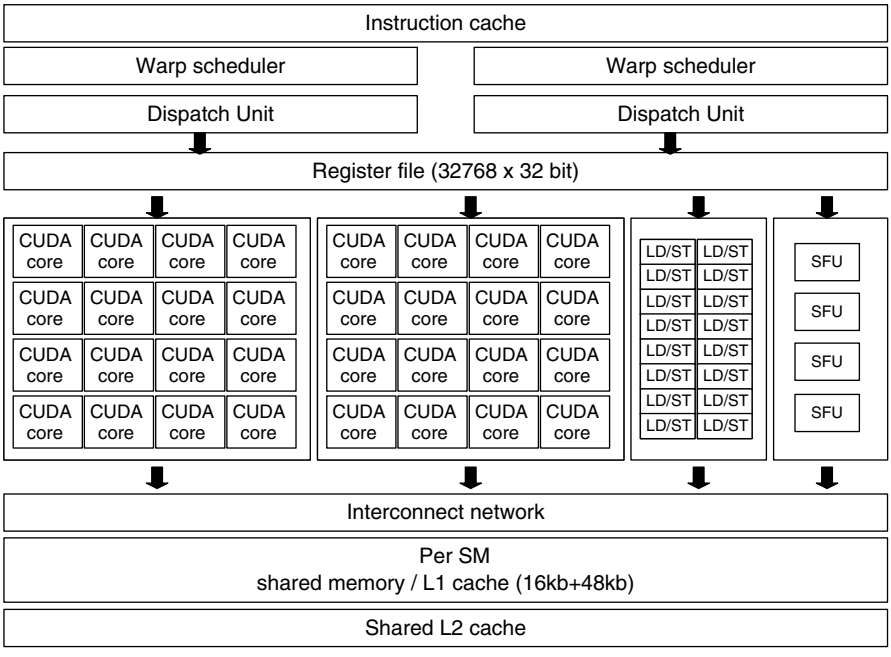


Fig. 2.7 NVIDIA streaming multiprocessor architecture

The Fermi architecture based chips have a two-level thread scheduler called GigaThread. On the chip level, an engine distributes thread blocks to different streaming multiprocessors; on each streaming multiprocessor, the so called dual warp scheduler distributes groups of 32 threads (which are called warps) to the available CUDA cores. Each streaming multiprocessor has two instruction dispatch units, thus at each cycle two warps are selected and an instruction from each warp is issued to 16 cores.

GPUs have clearly outgrown their roots in graphics processing and are now targeting the broader field of massively data-parallel applications with significant amount of scientific and floating point calculations. While a particular programming model must be followed (which we discuss later on in this book), the programmability of these chips has increased to a level that is on par with regular CPUs, turning these chips into a compelling choice for general purpose utilization.

2.4.2.3 PicoChip Architecture

We included the DSP architecture developed by picoChip because it's a prime example of successfully addressing a particular application domain with a massively multi-core processor.

The basic architecture of a picoChip DSP consists of a large number of various processing units (more than 250 in the largest configuration), connected in a mesh network (called the *picoArray*) using an interconnect resembling Tilera's iMesh technology. However, the communication is based on time division multiplexing mechanisms with the schedule decided deterministically at compile time, thus eliminating the need for execution time scheduling and arbitration. While this may be a big advantage in some cases, it will also limit the usability of the technology for more dynamic use-cases.

The processing elements in the *picoArray* can be

- *proprietary DSP cores:* 16 bit Harvard architecture processors with three-way very long instruction words, running at 160 MHz
- *hardware accelerators* for functions such as Fast Fourier Transformation (FFT), cryptography or various wireless networking algorithms that are unlikely to change and hence can be hard-coded into hardware
- *ARM core* for more complex control functions (such as operation and maintenance interface)

The DSP cores come in three variants, differentiated by target role and, as a consequence, available memory:

- *standard:* used for data-path processing with very low amount (< 1 kb) own memory
- *memory:* used for local control and buffering with approx. 8 kb of memory
- *control:* global control functions, with 64 kb of memory

Chips based on the *picoArray* architecture—due to the low amount of memory and very simple cores—have very low power consumption. For example, a model with 250 DSP cores running at 160 MHz, one ARM core at 280 MHz and several accelerators consumes just around 1 W, while capable of executing 120 billion basic arithmetic operations per second.

The picoChip architecture is a prime example of massively multi-core processors found in the embedded domain, especially telecommunication products (wireless infrastructure nodes and routers). These nodes usually perform the same, relatively simple sequence of operations on a very large amount of entities (packets, phone calls or data connections), hence such architectures are the best match in terms of balance between programmability and efficiency.

2.4.2.4 Many Integrated Core Architecture

In 2010 Intel announced their first commercial many-core chip code-named Knights Corner that will be manufactured in 22 nm and will integrate more than 50 x 86 cores. While not publicly stated, it's most likely the continuation of the Larrabee program that aimed at developing a GPU-like device.

There are few details available publicly about this device that is planned to be released in 2011. Based on an Intel presentation [3], it will be based on "vector Intel Architecture cores", which indicates an emphasis on data parallel processing. The cores will support hardware threading (at least 4 threads per core) and will share a tiled, coherent on-chip cache. The chips will likely include hardware accelerators, called *fixed function logic* in Intel's presentation. A schematic view of the architecture is shown in Fig. 2.8.

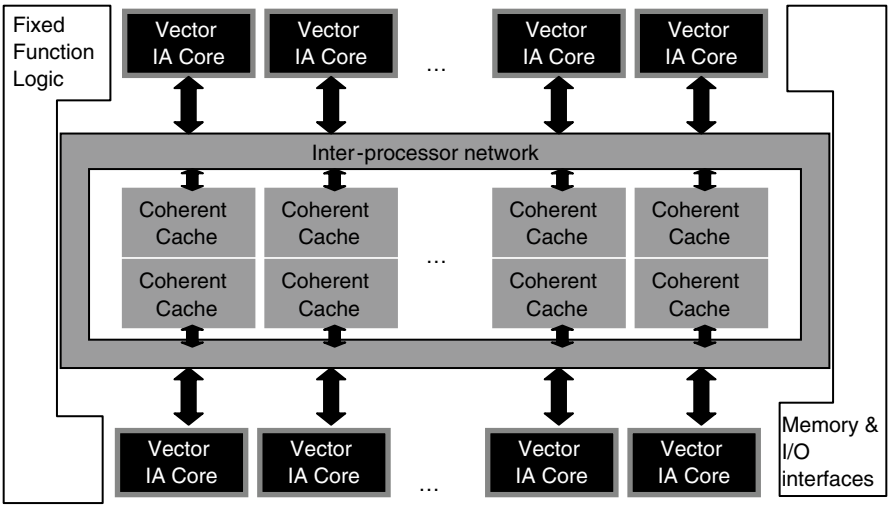


Fig. 2.8 Schematic architecture of Intel's Knights Corner chip design

In many ways this chip seems to be Intel's combined answer to the advancement of GPUs and the Tilera-type many-core architectures. Intel brands this type of chip a co-processor that a traditional server chip can use to offload parts of the application—enabled by sharing the same ISA.

2.4.3 Heterogeneous Processors

From purely hardware technology point of view, using dedicated cores for specific tasks is the best choice in terms of efficiency, as it will yield a simpler structure and less power consumption, while will deliver higher performance. As an extension, for a more complex problem, with sub-problems that require specific type of functionality, chips that combine cores with different capabilities are the optimal choice. Processors of this type are called *heterogeneous processors.*

Obviously there are other factors that make this rather simplistic line of thinking hard to argue for in full. Developing a chip has a significant cost, which is proportional to the complexity of the chip; on the other hand, the more specialized a chip is, the smallest its addressable market and hence the higher the per unit cost will be; additionally, higher complexity inherently will lead to increased software development cost. There is a trade-off point beyond which specialization cannot be anymore justified, hence the addressable market shall be sufficiently large and/or the chip design sufficiently generic so that the chip will be economically sustainable.

There are a few good examples of such chips, especially in high volume markets such as mobile computing or gaming. We will exemplify this type of processors through the Cell BE, used in game consoles as well as in supercomputers.

2.4.3.1 The Cell Broadband Engine

The Cell processor is the result of the co-operation between Sony, Toshiba and IBM and originally it was targeting gaming and other entertainment devices (including HD television sets).

The basic architecture of the Cell processor is shown in Fig. 2.9. The processor has nine cores, inter-connected through a circular bus called Element Interconnect Bus (EIB):

- one *Power Processing Element (PPE):* two-way SMT processor core running at 3.2 GHz, based on the Power Architecture ISA and equipped with 32 kb L1 and 512 kb L2 cache; it usually runs a main-stream operating system such as Linux
- eight *Synergistic Processing Elements (SPE):* DSP-like SIMD processors equipped with 256 kb of local memory (marked with LS—local store—in the figure), that can only be accessed from outside the SPE using DMA through a specialized memory flow controller unit (MFC)

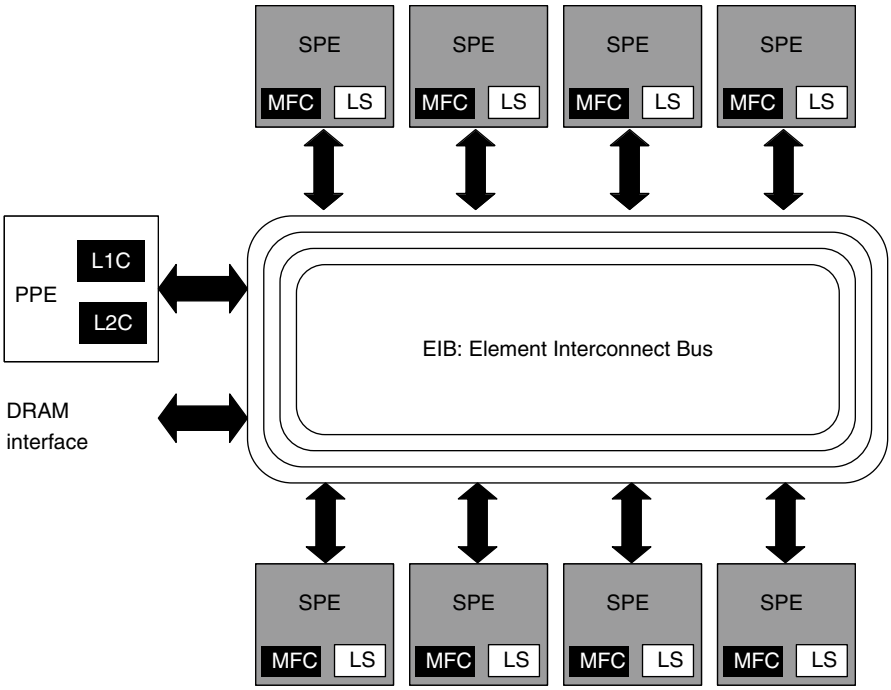


Fig. 2.9 The architecture of the Cell Broadband Engine chip

Beside the basic Cell processor, there's a special version used in supercomputers with about an 8× improvement in performance (to over 100GPFLOPS aggregated throughput).

The design of the Cell processor clearly prioritized performance over programmability: the SPEs need to be managed, scheduled and configured explicitly by the PPE, which adds significantly to the cost of software development. The division of tasks is clear: the SPEs are the number-crunching workers under the strict and necessary supervision of the PPE that takes care of all the other functions. This kind of chip architecture bolds well for game consoles (the primary users of the Cell processor), but also for compute-intensive supercomputers, where the PPE is the "visible" part of the processor capable of high-speed data crunching; internally, it can offload the work to the SPEs.

2.5 Summary

Any book on programming would be incomplete without a basic understanding of the concepts and design choices behind the actual hardware on which the software will have to execute. The goal of this chapter is to lay the (hardware) foundation

for discussing the software stack in the context of many-core programming: we surveyed the main design concepts available today, the challenges faced by chip designs when scaling up to tens or hundreds of cores as well as the emerging technologies that can help mitigate these challenges.

We believe that in the future we will see a continued increase of the amount of cores integrated on one chip—in the absence of some break-through technology, there are very few options available that can drive the performance of individual cores. This trend is already visible today: the success of GPUs and Tilera's architecture and Intel's plans for integrating more than 50 cores on a single chip all point to this direction.

The real question however is how to make use of this parallel computing power—the subject of the remaining part of this book.

References

1. Censier L M, Featrier P (1978) A New Solution to Coherence Problems in Multicache Systems," IEEE Transactions on Computers, 27(12):1112-1118
2. Gschwind M, Hofstee H P, Flachs B, Hopkin M, Watanabe Y, Yamazaki T (2006) Synergistic Processing in Cell's Multicore Architecture. IEEE Micro 26(2):10-24
3. Intel Corporation (2010) Petascale to Exascale: Extending Intel's HPC Commitment. http://download.intel.com/pressroom/archive/reference/ISC_2010_Skaugen_keynote.pdf. Accessed 11 January 2011
4. Sonics MemMax Scheduler. http://www.sonicsinc.com/uploads/pdfs/memmaxscheduler_DS_021610.pdf. Accessed 10 January 2011
5. Mutlu O, Moscibroda T (2009) Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems. IEEE Micro Special Issue 29(1):22-32
6. Ahn J H, Leverich J, Schreiber R S, Jouppi N P (2009) Multicore DIMM: an Energy Efficient Memory Module with Independently Controlled DRAMs. Computer Architecture Letters 8(1): 5-8
7. The OpenMP Architecture Review Board (2008) The OpenMP Application Program Interface. http://www.openmp.org/mp-documents/spec30.pdf. Accessed 10 January 2011
8. Frigo M, Leiserson C E, Randall K H (1998) The implementation of the Cilk-5 Multithreaded Language. Proceedings of the ACM SIGPLAN 1998 conference on Programming Language Design and Implementation, 212-223
9. Culler D E, Gupta A, Singh J P (1998) Parallel Computer Architecture: A Hardware/Software Approach. Morgan Kaufmann
10. Hennessy J L, Patterson D A (2006) Computer Architecture: A Quantitative Approach 4th Edition, Morgan Kaufmann
11. Wikipedia article Hyper-threading. http://en.wikipedia.org/wiki/HyperThreading. Accessed 10.1.2010
12. Mars J, Williams D, Upton D, Ghosh S, Hazelwood K (2008) A Reactive Unobtrusive Prefetcher for Multicore and Manycore Architecture. Proceedings of the Workshop on Software and Hardware Challenges of Manycore Platforms 2008, 41-50
13. Nellans D, Sudan K, Balasubramonian R, Brunvand E (2010) Improving Server Performance on Multi-Cores via Selective Off-loading of OS Functionalility. Proceedings of the 10th Workshop on Interaction between Operating Systems and Computer Architecture
14. Chua L O (1971) Memristor—the Missing Circuit Element. IEEE Transactions on Circuit Theory 18(5):507-519

15. Kurian G, Miller J E, Psota J, Eastep J, Liu J, Michel J, Kimerling L C, Agarwal A (2010) ATAC: a 1000-core Cache Coherent Processor with On-Chip Optical Network. Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques: 477-488
16. Herlihy M, Moss J E B (1993) Transactional Memory: Architectural Support for Lock-free Data Structures. Proceedings of the 20th International Symposium on Computer Architecture: 289-300
17. Falsafi B (2009) Energy-Centric Computing & Computer Architecture. Proceedings of the 2009 Workshop on New Directions in Computer Architecture
18. SPEC (2008) SPEC CPU2006. http://www.spec.org/cpu2006/. Accessed 11 January 2011
19. Åbo Akademi (2010) Cloud Software Program: SIP-Proxy and Apache Running on traditional X86 vs ARM Cortex-A9. https://research.it.abo.fi/projects/cloud/posters/POSTER_A3_Demo_2_ARM-SIP_2.pdf. Accessed 11 January 2011

Chapter 3
State of the Art Multi-Core Operating Systems

Abstract Operating systems represent the software foundation that enables applications to make use of the hardware resources of the computer in an efficient way. In this chapter we introduce the main roles of an operating system and will discuss in detail the two features most relevant from programming perspective: scheduling of threads for execution on available processor cores and resource management, primarily virtual and physical memory management in a multi-core context. We illustrate these concepts through the three dominant operating systems in use today: Linux, Solaris and Windows, with particular emphasis on thread and memory handling.

3.1 Definition of an Operating System

The origins of the term operating system are lost in the mist of the sixties of the last century, as no one seems to know for sure who first used it to describe the basic software of a computer. We do know though that IBM used it to describe the system software of their System/360 in 1964—called OS/360—famous for the huge problems its development has encountered, leading to the no less famous book on software project management, *The Mythical man-month* [1].

If the origins of the term are not entirely clear, the definition of what an operating system really is made also a good subject for lengthy and passionate technical—and lately, legal—debates (just think of the browser war and whether the browser shall be part of the OS or not). For the purposes of this book however we will limit ourselves to a basic definition that can be used as a clear basis for the discussion of the subjects that require operating system service.

So, what is an operating system?

To be able to answer this question, it's useful to take a brief look at the evolution of computers. Back in the early fifties, only one program was executing at any time and had full control of the complete computer. The programmer had to take care of every single detail needed to execute the program, manage hardware events and getting the data in and out of the system. Then, with the development of more compli-

A. Vajda, *Programming Many-Core Chips,*
DOI 10.1007/978-1-4419-9739-5_3, © Springer Science+Business Media, LLC 2011

cated peripherals (such as displays, printers, magnetic tapes etc.) in the later part of the fifties the need arouse for maintaining these and providing a translation function for the actual application—hence the first drivers packaged as libraries appeared, combined gradually into an application that had to be loaded on the computer before any useful task could be executed and which provided a minimal environment for the application itself to rely on. These libraries went by names 'master control program', 'executive' and the like and were essentially resource managers for I/O resources spiced up with some application start/stop functions.

The next big thing that really led to the development of operating systems was the emergence of multitasking in the sixties, prompted by the dramatic improvement in the performance of the processors and the increasing role of input/output devices, running at significantly lower speed—which would have left the processor unused for large periods of time, something totally unacceptable in an era when computers still cost tens to hundreds of thousands of dollars. It's probably fair to say that multitasking was the last spark that triggered the emergence of modern operating systems and—even though many would argue against this claim—we believe that OS/360, or rather the family of operating systems it has originated, was the first truly modern operating system to see a widespread use. The rest, as the saying goes, is history—Windows, various flavors of Unix, MacOS, Linux and the rest, with their ever increasing range of features appeared as the range of hardware diversified, and became personal and commoditized.

What is then the core and soul of an operating system?

First of all, the operating system is, from the application perspective, the *abstraction layer on top of the underlying hardware*. It shall take care of interrupts, processor management, low level interaction with peripherals and other similar tasks, while providing a unified interface and abstraction for the applications to use. This role is probably the oldest role of any operating system, tracing back to the beginnings of modern computers—and is usually one of the least disputed ones when it comes to drawing the limits of operating systems.

Though we will discuss the issue of virtualization further on, it's important to make a clarification of what hardware means in the context of the previous paragraph, in a virtualized environment. To put it simply, it means anything that the operating system perceives as hardware—it may be real hardware or an abstraction presented as hardware—virtual hardware—by some underlying virtualization software. It is the *totality of* (real or virtual) *hardware* that this specific (real or virtual) computer may use and therefore the operating system needs to manage in order to perform its function of abstraction layer towards the applications. The same reasoning still holds—the applications shall not be exposed to the nitty-gritty details of hardware, rather shall focus on the tasks that they were designed to perform.

The second role of an operating system is that of *resource owner and resource manager*. While resource may mean many things, in this context we define it as three specific entities:

- Processor resources that may execute application code
- Memory, either on-chip or off-chip

- Peripherals, such as network interfaces, hardware accelerators, and any other input or output devices that multiple applications may share.

In this context the role of the OS is that of resource broker and supervisor, with the goal of maximizing resource exploitation while meeting requirements from the applications. This role originates from the ever-growing gap between the speed of processors and other resources such as memory and peripherals: in order to keep the processors as busy as possible, multiplexing of tasks became mandatory.

In our view these two roles—hardware abstraction layer and resource owner/manager—are at the heart of any operating system. Other functions typically associated with operating systems, such as graphical interfaces or networking stacks, while essential for many applications are not mandatory for *all* applications executing on a certain computer. As said before, this distinction is the subject of a long-running debate in the academic, industrial and legal sphere; our delimitation here has the mere goal of setting the focus area for further discussions on operating systems in the context of multi-core and many-core chips.

The rest of this chapter is structured as follows. The first sub-chapter deals with various architectures for operating system kernels; the second sub-chapter discusses scheduling, while the third one handles issues related to memory and peripheral management. Finally, we will exemplify all these aspects through mainstream operating systems: Windows, Linux and Solaris.

3.2 Operating System Architecture: Micro-Kernels and Monolithic Kernels

There's such a diversified offering of operating systems, that it's hard to establish a simple set of classification criteria. From this book's perspective however, there are fundamentally two dimensions along which operating systems can be classified: the architecture of the kernel and the approach to task scheduling. We'll briefly discuss these different categories in this chapter, but for a thorough presentation we recommend one of the books listed under Refs. [2–4]. Our goal here is to merely introduce the fundamental ideas to be used in subsequent discussions.

The *kernel* of an operating system is usually defined as the part of the operating system that is mandatory and common to all other software that executes on that specific computer. What exactly shall be 'mandatory and common' is at the core of the micro-kernel versus monolithic kernel divide and it revolves around the all-important concepts of kernel space versus user space. Essentially, a piece of software is considered to execute in kernel space if it has direct, unrestricted access to all protected data structures of the operating systems kernel (of all the components considered mandatory and common), without any other specific and explicit security restrictions—hence it has all the same rights as any other piece of software executing in kernel space. On the other hand, software executing in user space has no direct access to any of the kernel space data structures; it may influence the behavior of the kernel only through a set of well defined, restricted interfaces. These

interfaces may be implemented as system calls or as a messaging interface, but the clear distinction between the two modes lies at the type of access that is granted to kernel-space data structures.

From the kernel architecture perspective there are two main categories of operating systems, with a number of variations in between. *Micro-kernel* operating systems are characterized by running most of their services in user mode as user processes and keeping only the very basic scheduling and hardware management mechanisms in the protected kernel space. *Monolithic kernels* on the other hand are characterized by incorporation of most of the operating system services into the kernel space, sharing the same memory space.

The fundamental principle micro-kernel designs aim to follow is the principle of separation of mechanism and policy. In a micro-kernel, the kernel's role is to provide the minimal support for executing processes, inter-process communication and hardware management. All the other services—and indeed, policies—are then implemented as servers in user space, communicating with applications and with other components through inter-process communication mechanisms, usually messages. Micro-kernel based operating systems are usually characterized by strong modularity, low kernel footprint and increased security and robustness, as a consequence of the strong isolation of the components and the execution of most services in user-space. On the other hand, micro-kernels traditionally require a higher overhead for access to operating services, as there will be more context switches between user-space and kernel-space modes.

Monolithic kernels excel primarily at speed as all the services of the operating system execute in kernel mode and hence can share memory and perform direct function calls, without the need for using inter-process communication mechanisms, such as message passing. As most of the OS functions are packed together, monolithic kernels tend to be bigger, more complex and hence more difficult to test and maintain, also requiring careful, holistic approach to overall design.

There have been several long debates around these two different approaches, most famously between Linus Torvalds, the inventor and gate keeper of Linux (on the monolithic side) and Andrew Tannenbaum, a respected professor and author (advocate of micro-kernel architectures), dating back to 1992 with a revived exchange in 2006. The essence of the debate revolves around maintainability, security, efficiency and complexity of operating systems, with valid arguments brought forward by both camps. The argument for micro-kernels is primarily based on the emphasis on reliability and security, supported by as little data sharing as possible and strict decomposition and isolation of operating system components. The counter-argument brought forward by Torvalds builds on the fact that algorithm design for distributed, share-nothing systems is inherently more complex and hence micro-kernels, with their emphasis on isolation would suffer on the maintainability and performance front. It's fair to say that on the theoretical level, the debate is undecided; however, on the practical side, mainstream general purpose operating systems tend to have a monolithic architecture, while operating systems targeting specific application domains have, in many cases, a micro-kernel based approach. Examples of micro-kernel based operating systems include some real-time operating systems

(such as QNX [5] and OSE [6]) and Mac OS, which is in fact a hybrid built around a micro-kernel, but using shared memory space with other OS components.

Before concluding our brief introduction to operating system kernel types, it's fair to mention a third form of kernels, called *exokernels* [7]. An exokernel's primary design principle is to limit the HW abstractions it defines to the bare minimum—focusing primarily on the multiplexing task of the operating system, offering 'real' HW to its users. All the other typical tasks of an operating system are then offered through library operating systems, which can provide specific abstractions to the applications while using the services of the exokernel. In many cases, such an approach offers even greater flexibility in building the 'just right' operating system, tailor made for each application—but at the cost of placing some of the typical OS functionality in the domain of user-space applications.

From the perspective of multi-core operating systems this debate is only relevant to the extent it brings forward potential bottlenecks that only become visible as we try to scale up to more cores. As we'll elaborate on this further on, widely shared data structures tend to become a bottleneck at some point, hence designs that limit large-scale sharing tend to perform better. However, this argument is essentially orthogonal to the discussion about micro-kernels versus monolithic kernels; it has to do more with the design of internal data structures and scheduling philosophies than with how the software components are structured.

3.3 Scheduling

One of the fundamental concepts in any operating system is the scheduling of applications that are using the computer under the control of the operating system. Task scheduling—also referred to as thread or process scheduling—is at the heart of any operating system and largely defines how well the operating system is capable of dealing with various workloads and various types of applications. In the context of this discussion, we define *scheduling* as the mechanism the operating system uses to execute available programs on the available hardware with the dual goal of maximizing processor usage and minimizing execution time for any of the programs that shall execute on the computer it controls. In this chapter we will use the term task to denote schedulable entities—traditionally called programs, processes or threads; a task is, in this context, a certain piece of computation that shall be executed on a computer.

In traditional, single processor systems, the goal of the scheduling was to maximize the utilization of the CPU, traditionally considered the key resource in any computer. As the gap between the performance of the CPU and that of other components—memory, peripherals, networking infrastructure—was widening, it became obvious that the best way to keep the CPU occupied was to allocate multiple tasks (programs) to the processor and execute these in a time-multiplexed manner. Multiplexing of tasks in time—while implemented according to a bewildering number of scheduling policies—relies traditionally on two very simple principles: the readiness principle and the priority principle.

The readiness principle simply states that a task will be considered for execution as soon as it has all the data and input that it needs to run available—i.e., it does not wait on a memory access, peripheral device or user interaction. The readiness principle simply defines the range of tasks that the operating system shall consider when deciding what the processor shall execute.

The priority principle sets the high level policy for how to share the computing power of the processor among available tasks. In fact, the priority of a task has been—until recently—the only indication applications were able to provide to the operating system as to the importance of a specific task and how it shall be treated in relation to other, competing tasks. While there are several strategies for dealing with tasks at various priority levels, the fundamental, underlying principle remained the same: the sharing of computing capabilities and the multiplexing in time of tasks shall be guided by the priority assigned to each task.

It's important to emphasize that time multiplexing and priority scheduling were driven by two factors: the gap between the performance of the CPU and peripherals and by the much larger number of tasks to be executed, as opposed to the amount of available processors. This approach is essentially pervasive—there's practically no operating system today that does not follow these two basic principles.

The emergence of shared memory multi-processors and multi-core processors added an extra layer to the scheduling of tasks by operating systems: for the first time, the operating system was facing multiple processors, capable of executing tasks—and in fact, operating systems—truly in parallel. Two new principles emerged, adding extra complexity the scheduling problem: the principle of load balancing and the principle of non-uniformity.

The load balancing principle simply states that the operating system shall aim at balancing the load on available processors, by choosing a suitable deployment and scheduling strategy. This principle has to be balanced however in relation to the principle of non-uniformity: as not all resources will have equal access times from all processors, task scheduling shall take into account the resource needs of tasks—primarily, memory access—when choosing a certain scheduling policy. The concept of Non-Uniform Memory Architecture (NUMA) is at the core of this principle, with regards to memory access.

Depending on the strategy chosen for handling multiple processor cores, today's operating systems mostly fall in one of the following three categories: symmetric multi-processing (SMP), asymmetric multi-processing (AMP) or bound-multiprocessing (BMP). We'll briefly introduce these categories—but for a broader discussion of the subject, we recommend going through some of the well-established text books on the subject, such as Refs. [2–4]. Figure 3.1 gives an overview of the three architectures.

3.3.1 Symmetric Multi-Processing

In a symmetric multi-processing context, the operating system treats all available processor cores as equal resources, hence putting emphasis on the load balancing principle: available tasks will be allocated to cores in such manner that all cores are

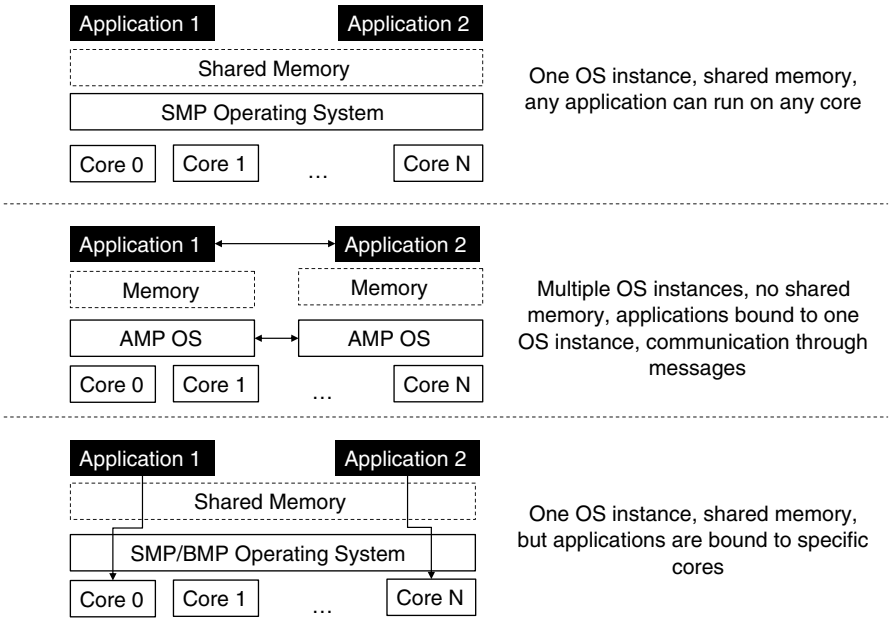


Fig. 3.1 Operating systems architectures

kept busy as much as possible. Typically there is one single operating system image managing all cores and the memory is globally shared—a key enabler for scheduling any task on any available processor core.

In case of NUMA architectures, the SMP model is usually refined with the notions of *clustering* of cores and *affinity*. All cores in a cluster share the same memory access characteristics, in the sense that every part of the globally shared memory can be accessed with similar latency and access time from each core in the same cluster. The affinity of a task defines on which cluster it shall execute in order to improve its performance—however this affinity may be overruled by the operating system in order to follow the load balancing principle.

SMP architectures are widely used today on homogeneous architectures where programmability and shared memory semantics are the main concerns. Practically all major operating systems—including Windows, Linux and different variants of Unix—offer an SMP approach. As we'll see in Chap. 7, however, SMP introduces several issues related to cache utilization and operating system kernel scalability that may limit its applicability to large number of cores located on the same chip.

3.3.2 Asymmetric Multi-Processing

As opposed to symmetric multi-processing, asymmetric multi-processing (AMP) emphasizes partitioning and role specialization for available processor cores. On the hardware level, asymmetric multi-processing usually means cores with differ-

ent capabilities—e.g. different instruction set architectures or different execution speeds—that essentially precludes the usage of an SMP model, for the simple reason that tasks may not be allocated on any of the available processor cores. Even in case of homogeneous hardware architectures, there may be reasons for choosing an asymmetric multi-processing approach: memory partitioning, explicit and restrictive affinity management or simply lack of scalability of the operating system across multiple processor cores.

An AMP operating system is usually characterized by multiple operating system domains executing on different sub-sets of the available processor cores. Applications are locked to one of the domains and may not transparently migrate to another domain; at the same time, communication between applications running in different domains is restricted and strictly controlled—in many cases, it's only possible through messages. In some cases, there is a master–slave relationship between different domains, in the sense that 'slave' domains can only execute tasks dispatched from the master domain.

Asymmetric multi-processing operating systems are usually deployed on heterogeneous hardware architectures or in cases when isolation brings additional benefits, such as real-time systems.

3.3.3 Bound Multi-Processing

A special class of operating systems is using a specialized version of symmetric multi-processing, called bound multiprocessing (BMP). BMP provides similar scheduling semantics as an asymmetric multiprocessing model, however in the context of a single copy of the OS that maintains an overall view of all system resources, similarly to the symmetric multi-processor model. The key differentiating feature is that in bound multiprocessing operating systems, the location (binding) of each task is explicitly defined; hence all tasks are locked to a specific core—in practice, placing the task of load-balancing in the hands of the programmer. Compared to conventional SMP operation, this approach offers several advantages: it improves cache performance, by providing the programmer with the tools needed to optimize placement with respect to cache and it provides a framework for executing legacy applications with poor multi-processing behavior. Most SMP systems today offer the tools necessary to use them in a BMP manner, such as the core affinity attribute that can be used in Linux to steer the placement of tasks.

3.4 Memory Management

Resource (other than the processor core resources) management in general—and memory management in special—is one of the key roles for any operating system. Any application requires memory to execute, in the form of stack and heap and the

task of coordinating and servicing requests coming from various applications is naturally handled by the operating system kernel.

The fundamental principle any OS shall adhere to is that of *isolation*. The operating system shall guarantee that no application can access the memory belonging to any other application or to the kernel. Any such attempt shall result in an exception that the OS shall handle in an orderly way (e.g. by terminating the offending application).

Another fundamental principle is the principle of *continuous memory*. The memory allocated to an application as part of one request shall be continuous, thus addressable through a single pointer, irrespective of how it is actually laid out in the physical memory. Main-stream, general purpose operating systems usually complement this with another principle, that of *exclusiveness:* any application shall be presented with the view of having exclusive access to the complete physical memory (and, for some operating systems, to the full range of addressable memory), irrespective of how much physical memory is actually installed.

As we will elaborate on further in this chapter, these three principles are at the foundation of the concepts of virtual memory and associated concepts and techniques such as memory pages, swapping and address mapping.

An important principle observed to various extents by modern operating systems is that of *locality*. Especially in multi-processor and multi-core systems it is very important that memory is allocated as close to the processor core on which it will be used as possible, in order to avoid the cost of transferring memory content in terms of latency and core stalling. Equally important is the knowledge of whether a certain memory area (memory page) is in the core's local cache or not; this knowledge can steer the placement of newly allocated memory as well as steer the swapping of pages as function of memory access patterns.

Besides these principles, an important phenomenon memory managers have to deal with is that of *memory fragmentation*. Repeated allocation and de-allocation of memory can lead to a high degree of fragmentation, where the size of the largest continuous free memory block is significantly smaller than the total amount of free memory. There are several techniques that were developed to deal with this issue (most prominently the method of buddies), but so far no simple and efficient solution was proposed that would address the problem in its entirety.

In multi-core systems, the scalability of memory management is one of the major concerns. Traditionally, all data structures dealing with memory management are centralized, in order to keep one consistent view of the memory. However, in multi-core systems such an approach requires either a central singleton memory manager that can quickly become the bottleneck, or a synchronized access to the global memory-related data structures that, as we will see in Chap. 7, will lead to increased latency of memory management operations.

In the following sub-sections we will take a closer look at some of the established concepts and methods related to memory management: paging and virtual memory, memory allocation techniques and methods for preventing and managing memory fragmentation.

3.4.1 Virtual Memory and Memory Pages

The concept of virtual memory is in no ways new: it's a well established technique to provide the appearance of much larger memory than what is physically available. It relies on three techniques:

- memory content swapping to and from external storage
- statistical multiplexing of currently accessed memory areas in the main physical memory
- HW support for mapping virtual addresses to physical ones in the memory. This is also the prime enabler for providing each application with a continuous memory address space that is automatically mapped to potentially discontinuous physical (or swapped out) memory areas.

In an operating system using virtual memory, each application is provided with a continuous virtual address space that it can access as if it would indeed be a continuous physical memory area. This space is divided into virtual memory pages, blocks of few kilobytes each, managed as one entity. Memory pages are the primary instrument to manage the mapping of virtual address spaces to physical ones, as pages are the smallest entities that are mapped into the physical memory or external storage.

Each virtual memory page is either mapped to a page in the physical memory—in which case the hardware resolves automatically the mapping through the page mapping directory managed by the operating system—or it is swapped out into external storage. In this case, a page fault is generated by the HW and the operating system will try to swap it back into a physical page, potentially swapping out another, recently unused page in the process. This chain of mapping is illustrated in Fig. 3.2.

Swapping pages in and out of the physical memory obviously adds a significant performance penalty for applications, thus intelligent management of what gets swapped out of the cache and main memory is extremely important. Most modern operating systems implement some form of page usage monitoring so that situations where frequently used pages are swapped out in favor of other, perhaps just occasionally accessed pages can be avoided. A more subtle, yet very important issue is the mapping of pages to cache-lines: the operating system shall try to avoid putting pages that are accessed together into physical memory areas that are mapped to the same cache line, as such a situation would result in frequent cache misses and swapping of data in and out of the cache memory.

Usage of virtual memory constructs usually requires a centralized management of page faults and swapping of data to and from external storage. While such a system works well for single-core systems or for processors with just a few cores, it quickly becomes an unwanted contributor to the slowdown of the system as the number of cores increases: it will be directly reflected in the contention for the global data structures related to the management of virtual memory.

All in all, the concepts of memory pages and virtual memory are fundamental components of most modern operating systems and we believe it's fair to conclude

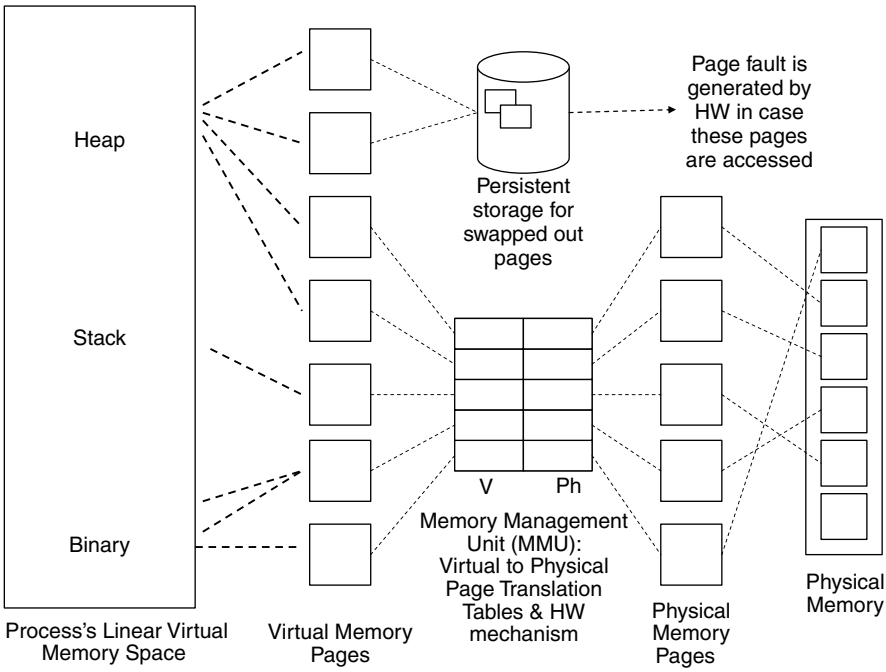


Fig. 3.2 Memory management concepts

that such mechanisms would anyway be implemented by applications, in case none would be provided as a system service. However, as the core count increases, there's a need to address scalability issues that can become a performance bottleneck due to the need to access centralized data structures from multiple cores.

3.4.2 Memory Allocation and Fragmentation

Memory allocation techniques are some of the most studied subjects in operating system research. In this chapter we cannot and will not attempt to cover all the results—instead we will focus on some of the fundamental principles as well as the challenges that need to be addressed in multi-core systems.

The first principle we'll call the *coarse-grained allocation principle*. In most operating systems, the OS deals with memory pages only; any finer granularity allocation is usually left to language run-time systems (such as the malloc/free methods of the C standard library). This approach simplifies to a large extent the task of the OS and helps mitigate the issue of memory fragmentation.

The second principle is the *locality principle,* especially important in distributed SMP systems. The operating system shall always service memory allocation requests as close to where the application is expected to execute as possible, in order

to minimize the memory access latency. This requires co-operation with the scheduler, both at memory allocation and re-scheduling, in order to avoid the migration of application threads away from cores which have the fastest access to the most used memory pages.

Note: there is another type of memory manager—process scheduler interaction that is used when the amount of available physical memory drops below certain level. In these cases, the memory manager may decide to temporarily swap out some memory-resident processes in order to conserve memory—irrespective of the scheduling decisions taken by the main process scheduler. This feature is available in most modern operating systems.

The third principle is the *cache collision avoidance principle,* which aims at minimizing the overlap of pages frequently used together in the core-local caches. In short, each physical memory page is mapped to a specific cache-line; if two pages are mapped to the same cache-line, these will compete for that part of the cache and will repeatedly invalidate that part of the cache, resulting in constant reloading of the cache content. Several algorithms—mostly based on page coloring according to cache line—are available to support this principle; it's very much dependent on the application's memory usage pattern which one performs best.

The fourth principle relates primarily to the management of executable code. Most modern operating systems implement some form of the *single code image principle,* i.e., having just one copy of the same program code in the memory for all applications. This principle may not apply globally—especially in distributed SMP systems—for performance reasons; in these systems, it's common to have one memory-resident image of each code segment per node (as a direct application of the locality principle).

One of the key design choices for memory management is the selection of memory allocation algorithm with respect to the sub-slicing of memory. The driving constraint is *memory fragmentation,* manifested through scattered, small sized segments of memory which make allocation of large segments impossible, even though the total amount of free memory is sufficient. As one would expect, there are plenty of algorithms developed over the years; here we will focus on the most widely used ones: the buddy allocator, the slab allocator and Solaris' Vmem allocator.

3.4.2.1 The Buddy Allocator

The buddy allocator [8] is one of the most widely used methods for managing memory with the goal of minimizing memory fragmentation. It adheres to the coarse grained allocation principle, in the sense that it's targeted at allocating larger chunks of memory (on the level of few hundred bytes to few kilobytes), instead of fine grained management of memory.

The basic idea of buddy allocator is to allocate memory in chunks of sizes of power of two (e.g. 1024, 2048 etc.). The system keeps a free list for each chunk size and implements a chunk slicing and merging algorithm as described below:

1. Initially, the complete memory is in one free list, as one large chunk
2. When a new request is received, the system will search for a free list with available chunks that are same sized or larger than the requested amount of memory
3. If the available chunk size is less than twice the requested size, one of the chunks is allocated
4. Otherwise, one chunk is selected and repeatedly sliced up in halves until the condition from step 3 is met. The resulting, unused chunks are put in the corresponding free lists
   - *Example:* A memory block of size 512 bytes is requested; as the smallest available chunk is of 2048 bytes, one of these chunks will be sliced up into two chunks of 1024 bytes each, one is put on the 1024 bytes free list, while the other one is further sliced up into two chunks of 512 bytes each: one is allocated while the other one is stored on the 512 byte free list
5. When a block of memory is released, the system will check if the any of the two neighboring chunks (buddies) are free; if yes, the chunks are merged and the process is repeated; finally, the resulting chunk is put in the corresponding free list

This method is illustrated in Fig. 3.3.

Clearly, this algorithm is just an *opportunistic method* to grab the low-hanging fruits whenever possible: it can re-merge chunks of memory, but this is only

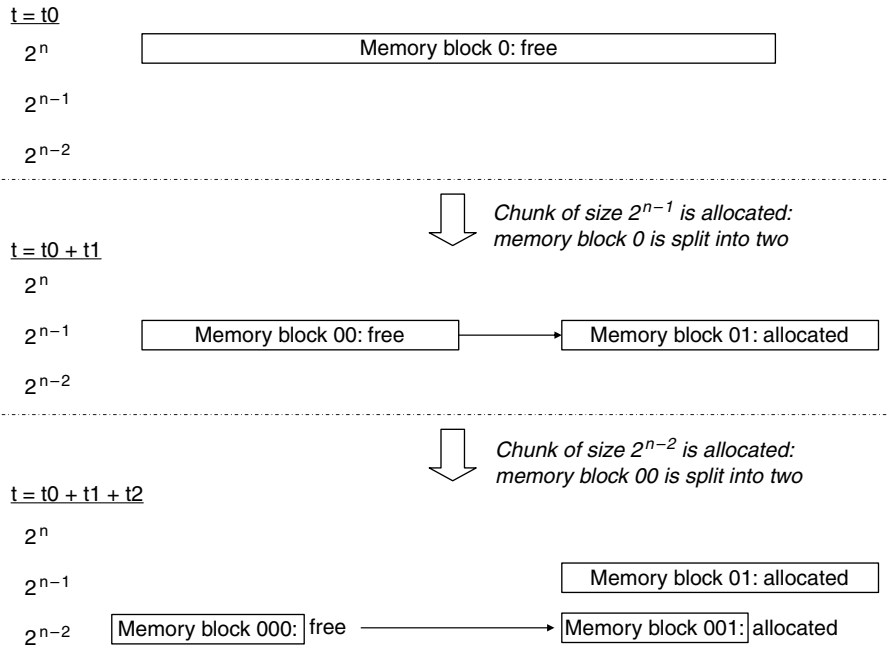


Fig. 3.3 The buddy allocator

possible if the overall allocation/de-allocation sequence has a pattern that results in neighboring chunks being released roughly one after the other.

The basic algorithm can be further improved through additional chunk selection mechanisms such as:

- Allocate chunks for one application from the same region of the memory: if the application terminates, a large, compact chunk of memory can be freed up
- Choose "isolated" chunks, in order to increase the continuous allocated memory areas

The buddy method has been significantly improved recently in the Linux kernel by grouping the memory pages according to whether the page can be moved (the case for most user space applications and for file caches) or not (kernel memory). By keeping these pages separate, file-system type of de-fragmentation can be performed for the movable pages (at the additional cost of content copying and re-mapping).

3.4.2.2 The Slab Allocator

The slab allocator [9] was designed for the Solaris kernel at Sun, but made its way into several other mainstream operating systems (such as Linux). While Linux uses it as a small scale allocator, in Solaris it's the main memory management algorithm, replacing the buddy allocator altogether.

The basic concepts of the slab allocator are those of *object,* to describe a memory allocation unit, *cache,* i.e., set of objects of similar sizes and *slab,* a sub-set of objects located within the same cache and stored in a continuous group of pages. Thus a slab represents a continuous set of pages that contain multiple objects of the same class (similar sizes), while a cache is a collection of slabs.

Consequently, the slab allocator is made up of two main layers. The *backend* (*lower*) *layer* is responsible for creating slabs using several subsequent pages allocated through a simple page manager; the *front end* (*higher*) *layer* takes care of servicing memory requests from clients. To put it simply, the backend supplies the raw material in which to place the objects, while the front-end is responsible for actually creating and maintaining the objects. The architecture of the slab allocator is shown in Fig. 3.4.

In practice however, especially for multi-processor systems, the slab allocator is usually implemented as a three layer system. Here we describe briefly the architecture of the Solaris slab allocator, however most implementations are likely to follow a similar of even more refined approach.

The lowest, CPU level layer implements a processor-level caching mechanism in order to minimize the overhead—and especially global contention—for object allocation and de-allocation. The mechanism used for this is that of *magazines,* groups of objects assembled by the higher levels, possibly from different slabs and put at the disposal of the CPU. The CPU level layer will use the magazines available to it to service local requests and will only interact with the higher layers when it needs more magazines.

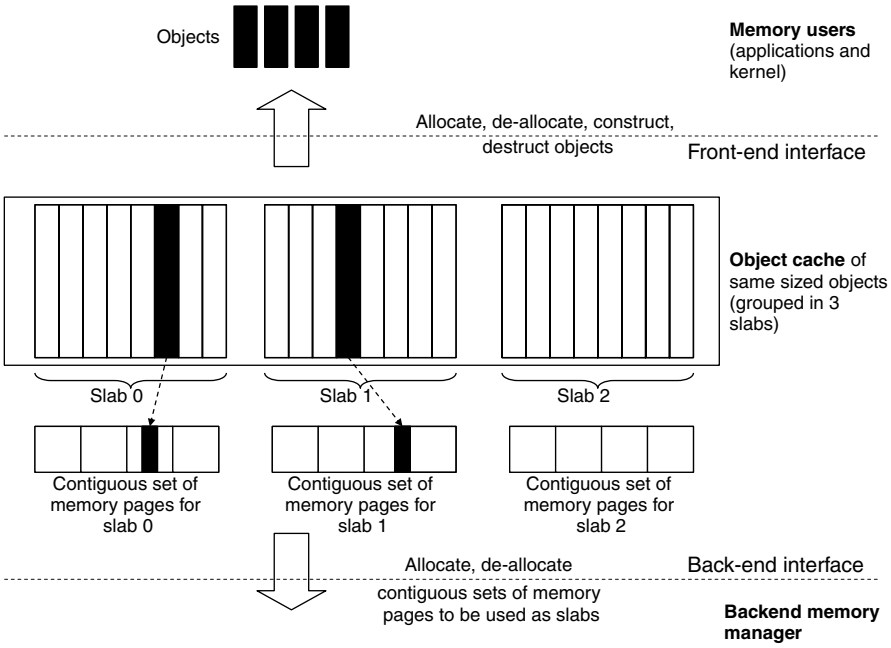


Fig. 3.4 The slab allocator

The intermediate layer is that of the depot manager. The depot manager takes care of constructing and allocating magazines as well as defining the appropriate size of the allocated magazines (based on regularly checked and evaluated usage patterns).

The highest layer is the global slab layer that manages the allocation of slabs from contiguous pages of memory for the different caches and passing these down to the depot manager. It also implements a slab coloring mechanism in order to reduce the probability of cache line overlap; the 'color' is simply an offset to the start of each slab.

One of the important characteristics of the slab allocator is that it can allocate memory areas of different sizes, by using a 'best fit' cache and a 'best fit' available object approach. Caches are usually created explicitly, normally at initialization phase of a kernel module or applications. Objects and slabs can also be created explicitly, hence de-coupling the memory object *construction* and *effective utilization* phases. This approach also allows the usage of *object caching* techniques: if the usage of objects in a cache is known a priori (e.g. as file descriptor), the file descriptor object can be pre-created in advance (in the construction phase) and its content preserved after de-allocation, keeping it in *hot or immediately usable* state. This way, especially for frequently used object types, the construction phase—which may be quite expensive—can be factored out and performed only once or just rarely; the client (user of the object) then can focus on just creating the differentiating content of the object. In order to improve the performance of the object caching system,

the applications may provide object constructor and destructor methods that will be invoked by the slab allocator whenever an object needs to be modified (this is the approach used in Solaris).

The slab allocator has two major benefits over the buddy method. First, it is significantly faster, not least due to the object caching functionality (in Solaris' case, the performance difference factor is about 2.5 ×); equally importantly, as objects are not clustered around addresses that are powers of 2, it tends to have a better cache behavior and hence improved application performance. Second, due to the clustering of similar sized objects, it reduces the fragmentation of the memory, again with a factor of approximately 2.5 × compared to the buddy allocator (expressed as the ratio of wasted memory: while for the buddy allocator it's above 40% in many cases, it stays under 20% for the slab allocator, according to Ref. [10]).

3.4.2.3 The Solaris Vmem Allocator

The Solaris Vmem allocator [10] was designed to handle virtual address allocation, however it evolved to become a general purpose resource manager. In the context of the Vmem allocator, a resource is anything that can be described by a set of integers: for example, a memory area is defined by two integers (start address and size) while process ids are single integers.

The Vmem allocator has a very simple external interface: it allows the creation of *arenas of resources,* modification of arenas by dynamically adding more resources and allocation/de-allocation of individual resources. An arena is defined as one or multiple integer ranges (spans) that represent resources (e.g. memory address areas), together with some characteristics that drive the resource management process: the *quantum,* which defines the resource allocation alignment (e.g. page size for memory allocation) and *cache size* which defines the maximum small integer multiplier of quantum to be reserved for caching. These two parameters define the sizes of object caches that the arena will have and which will be used to quickly allocate resources of those sizes: there will be object cache for object sizes equal to small integer (usually up to 5) multiples of quantum. The allocation of memory to these caches will be done in slabs of sizes equal to the next power of 2 above 3*cache size—a value that allows good fit for cache size values up to 5 and helps avoiding fragmentation within the arena.

Resource allocation is done in segments, defined by a start address and a length. The arenas will keep lists of segments grouped according to size, thus the list of best fit segments can be quickly identified and the right sized free segment selected. Using boundary tags, the Vmem allocator will essentially implement a buddy allocator mechanism in constant time and will try to merge segments whenever possible. Using the caching mechanism outlined above, segments of specific sizes can be allocated directly from caches, hence speeding up the process.

The Vmem allocator has proved to be a highly scalable general purpose resource management framework that has been reported to improve system level performance by up to 50% and help to eliminate several existing resource allocators in the Solaris system.

3.5 Current Main-Stream Operating Systems

Even though there are dozens or perhaps hundreds of operating systems targeting specialized domains such as automotive, real-time or telecommunications, when it comes to systems based on multi-processor architecture and multi-core processors, there are just a few well established operating systems that have seen a widespread usage. Of these, without the intention of completeness, we will briefly describe Linux, Solaris and the Windows family of operating systems; our intention is not to provide an extensive description, but rather focus on the main elements that are relevant from a multi-core perspective: scheduling policy and memory/resource management.

3.5.1 Linux

Linux was developed by Linus Thorvalds at the Helsinki University of Technology and first released as an open source operating system in 1992. Since then, it has undergone major redesigns and today is one of the most widely used operating systems, especially in server installations and supercomputers (about 80% of the top 500 supercomputers use Linux as the operating system [11]).

Linux is an open source SMP system with a monolithic kernel, a feature famously highlighted in the debate between Thorvalds and Tannenbaum, the leader of the MINIX project (a micro-kernel based Unix variant). Its design has undergone several major redesigns, most recently through the swap of the core scheduler in release 2.6.23 (currently—2010—the Linux kernel has a 2.6.3x release version), thus it is quite difficult to describe the features of Linux in general; the description given in this chapter is based on version 2.6.33.

3.5.1.1 Scheduling

Scheduling in Linux is structured on two levels: the core scheduler and the scheduler classes. The task division between the two layers is quite well defined: while the scheduler classes' layer takes the scheduling decisions according to various policies, hence deciding which task to run next, the core scheduler takes care of the general task management and task switching activities, independently of how—according to which policy—the next task to be executed has been selected. The overall architecture and relationships between layers is shown in Fig. 3.5.

The core scheduler is activated in two ways: either a task yields the CPU or through a mechanism that is run periodically and decides whether task switching is necessary. It consists of two main functions: the periodic scheduler and the main scheduler. The periodic scheduler performs—besides the collection of scheduling specific statistics—two major tasks: it decides and if needed, performs load rebalancing between CPUs in an SMP system and activates the periodic scheduling

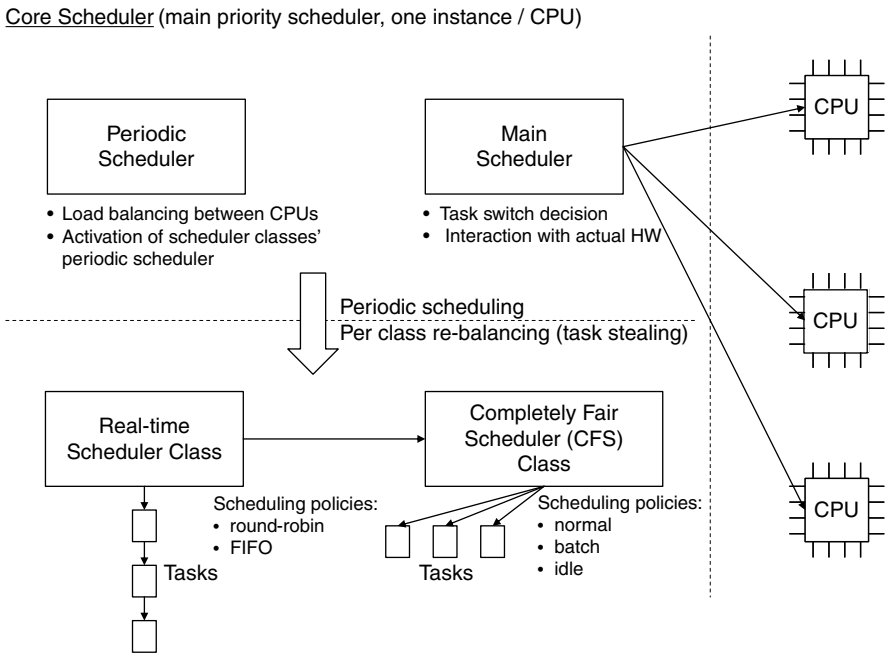


Fig. 3.5 The architecture of the Linux scheduler

method of the scheduling classes to which the currently running task belongs. This, in turn, may decide that a switch of active tasks is needed and indicates this through setting a kernel flag that signals the need to run the main scheduler.

The main scheduler is invoked at many points throughout the kernel—for example, after system calls, if the kernel flag indicating the need to reschedule is set—and is the entity that actually decides and triggers a task switch. At this point it's necessary to clarify the relationship that exists between different scheduling classes (or scheduling policies): from the main scheduler's perspective, these are organized in a strict hierarchy and no task belonging to a lower priority scheduling class will be selected as long as there are any runnable tasks belonging to higher priority scheduling classes. On the other hand, each task belongs to one scheduling class only (and each task *must* belong to a scheduling class). Due to this algorithm, the current main scheduler is also called the *priority scheduler,* as it selects the next task based on the priority of the scheduling class to which it belongs; for the same priority level, however, the decision—task selection policy—is delegated to the scheduler class.

It is important to clarify what exactly is being actually scheduled by the Linux kernel. We used so far the term task; in reality however the kernel works with *schedulable entities,* where a schedulable entity may be simply a thread but also a group of processes. From the overall scheduling architecture point of view this is practically irrelevant, hence our choice of using the term task.

Linux today supports, by default, two scheduler classes, each with different scheduling policies. The (soft) real-time scheduling class has the higher priority and supports the round-robin and FIFO scheduling policies; for a thorough discussion of this scheduling class, we recommend one of the many books and sources available, such as Ref. [12]. Herein we will focus on the second, more widely used general purpose scheduling class that supports the normal, batch and idle scheduling policies.

The general Linux scheduling class used prior to kernel version 2.6.23 was based on the O(1) scheduler [13], capable of scheduling processes within a constant amount of time, but this was replaced with a new, O(log n) scheduler called the *Completely Fair Scheduler* (*CFS*). It's based on the Rotating Staircase Deadline scheduler, also developed within the Linux kernel community. Consequently, today this main scheduling class is also called the completely fair scheduling class.

The basic principle of CFS is to provide as close to ideal as possible fairness to each task, with respect to the computational power that it gets allocated. In a simplified case of N tasks with the same priority, it would mean equal processor time. The core method to approximate such a situation is to keep an ordered list of tasks (using a red–black tree as implementation method), so that the tasks with the longest waiting times are at the head of the list and will execute next. In fact, the quantity by which tasks are ordered is modified to take into account the time it *should* receive and hence simulate the ideal case more precisely; this modification is based on the *virtual clock* of the CFS, which weights the wall clock by the number of available tasks.

Once a task is selected for execution, its waiting time will be decreased periodically with the amount of time it was allowed to run and hence eventually it will not be at the head of the ordered list—triggering the selection of another task for execution, the task that becomes first in the list.

In practice, this algorithm has to be fine-tuned in order to cater for several constraints: different priority levels shall be factored into the waiting time of tasks (tasks with higher priority shall have higher 'fair share' times than lower prioritized tasks), fairness shall be weighed against the cost of switching tasks too often: the overhead of doing so may outweigh the benefits, if done too often. In fact, the Linux kernel has two built-in parameter that controls the latency of scheduling; the first one indicates the time period within which *all tasks* must get the chance to execute at least once (default value is 20 ms); the second one sets the maximum number of tasks that are supposed to be handled within this time period (if this configured value is exceeded, the interval will be extended linearly).

3.5.1.2 Multi-Processor and Multi-Core Support

The multi-processor support of Linux is exclusively symmetric multi-processing centered, with support for non-uniform topologies (such as NUMA systems), but assuming CPUs with equal capabilities.

SMP support in Linux is an extension of single-processor scheduling. Each CPU has its own scheduler, but this is coupled with a periodic re-balancing between CPUs. Essentially, at each periodic invocation (tick) of the scheduler, on each CPU, the need for rebalancing is checked—in practice, this means that if sufficient time has elapsed since the last rebalancing, a new rebalancing procedure is initiated.

The rebalancing is done per scheduling class and always within scheduling domains. A scheduling domain is a set of CPUs that define a domain within which rescheduling can be performed (e.g., share the same card, processor socket or NUMA domain). In a perfectly flat SMP, there would naturally be only one scheduling domain.

The rebalancing is based on a task stealing mechanism. When rebalancing is decided, the thief CPU will identify the CPU with the busiest run queue and, if the load on that CPU is higher, it will attempt to move tasks from that CPU to itself (only tasks that are currently not executing may be moved). In case the move of tasks cannot be done for some reason, the CPU with the busiest queue will be triggered to perform itself the off-loading of some of its tasks; this will be achieved by a special thread called the migration thread, associated with every CPU and the procedure is called active balancing: tasks will be moved forcefully if it's deemed necessary.

The distributed nature of this rebalancing method is both its strength and weakness. As it's not a synchronized activity, it can work autonomously e.g. across multiple scheduling domains; for the same reason however it may be the source of contention between CPUs as several may attempt to move tasks from the same busiest CPU. The run-queue of any CPU may be inspected at any time by any other CPU and attempts to modify it may happen at any time and concurrently with local scheduling decisions. While careful locking schemes may make the mechanism work smoothly for reasonable number of CPUs, it may become impractical for large scale SMP systems, where it's often either disabled or confined to reasonably-sized scheduling domains.

3.5.1.3 Memory Management

Memory management in Linux is based on the buddy allocator for general purpose memory management and the slab allocator for kernel-specific memory allocation. Physical memory is divided, usually in a 3:1 ratio, between applications and the kernel.

From multi-processor and multi-core processor support point of view, Linux offers support for NUMA systems and memory page migration. For multi-processor systems in general and NUMA architectures in particular, memory is organized around the concepts of *node* and *CPU sets*. Each node owns part of the physical memory and in general corresponds to one CPU (or several processor cores that share part of the memory). A CPU set groups several CPUs (nodes) into one group that has a uniform access to part of the memory.

Applications may be assigned to certain CPU sets and the OS supports automatic migration of memory pages in case the application is re-assigned; the same effect

can be achieved by defining memory policies that force the allocation of memory for a specific application from a specific set of nodes. The memory policy manager can be configured to perform automatic page migration by redefining the set of nodes on which the pages for that specific application may be allocated, coupled with an explicit request for complying with the new policy (which in practice means page migration).

Beside these two methods—assignment to CPU sets and memory management policy—the Linux kernel does not support any additional automatic page migration policies, as the fundamental principle guiding memory management design has always been to leave the control of it to the application, for lack of a good method to predict best fit memory page localization.

3.5.1.4 Linux: Summary

Linux is a monolithic kernel, SMP-based multi-processor and multi-core enabled operating system. It has been used for very large SMP deployments, but usually with quite strong constraints in order to prevent automatic task migration and cross-node memory allocations: experiments have shown that these are the areas where Linux faces the most severe scalability issues. Research has also shown [14] that Linux is quite intrusive with regards to cache behavior: system calls tend to destroy application-specific cache, impacting application performance.

3.5.2 Solaris

Solaris was introduced in 1992 by Sun Microsystems (acquired by Oracle) as a replacement of their earlier SunOS operating system. It was based on Unix System V Release 4 and it's certified against the Single UNIX Specification. Solaris is an SMP operating system, supporting cache-coherent NUMA architectures. Traditionally, it has been renowned for its scalability and in many ways it's regarded as the basis to measure up against when it comes to operating systems for massively multi-processor computers.

Our discussion of Solaris is based on Solaris 10, first released in 2005 but regularly updated throughout the years. As with Linux, we focus on task scheduling, memory management and support for large-scale multi-processor and multi-core systems.

3.5.2.1 Scheduling and Multi-Processing Support

It's practically impossible to de-couple the subjects of scheduling and multi-processor support, as the Solaris scheduler—called the dispatcher—inherently takes multi-processing into account when making scheduling decisions. Unlike Linux,

load balancing and task management are inter-related and whenever a task needs to be modified in terms of its position in the scheduling hierarchy, CPU placement and load balancing will always be considered. This way Solaris has a very fine-grained task migration and load balancing mechanism embedded with the very fabric of task scheduling.

There are two components of the dispatcher functionality that needs to be understood in order to get an overview of how task scheduling works in Solaris: the abstract system model and the dispatcher architecture.

The abstract system model provides a logical view of the hardware in terms of processing cores, CPUs, groups of processors, resource grouping and latency hierarchy; this logical view is the one that guides the dispatcher's decisions with regards to thread placement. The basic concepts used in the abstract system model are the following:

- *CPU:* the CPU is the abstraction of a processor, where a processor is viewed as an execution resource for a thread; the meaning of the term is different depending on the underlying hardware: in some multi-core chips, it's the representation of a core, while for chips with hardware threads there will be one CPU object for each HW thread (e.g. to represent a 8 core machine where each core has 4 hardware threads, 32 objects of type CPU are needed)
- *Chip:* a chip is the representation of a physical processor chip; its exact semantics is dependent on the actual HW: it may represent just one core if that core has multiple hardware threads or a complete single- or multi-core processor in case each core has just one HW thread. From the abstract system model perspective, a *Chip* is always made up of one or several objects of type *CPU*
- *Processor sets, resource pools* and *CPU partitions:* a processor set is a user-level grouping of one or more processors (CPUs), used for binding tasks (processes) to a subset of the available processors; the CPU partition is the kernel representation of a processor set with one real-time queue (called kernel preempt dispatch queue) per CPU partition. A resource pool is essentially a stateful processor set (we will not deal with it in the following discussion of the dispatcher)
- *Locality groups (lgroups):* provide a mechanism for grouping resources—primarily CPUs and physical memory pages, but also I/O devices—together in order to express the cost in terms of latency of accessing certain resources from threads scheduled on specific CPUs. It's important to make the distinction between processor sets and locality groups: two CPUs may belong to the same processor set (a choice made by the system administrator) while being part of different locality groups (due to the way the actual hardware has been built). We'll discuss locality groups in more detail when presenting the memory management features of Solaris.

These concepts and their relationships are illustrated in Fig. 3.6.

The dispatcher uses the abstract system model defined using these concepts to optimize the placement of threads at creation (each thread is assigned to a *home lgroup* when it's created, based on the load level of different lgroups) and at every rescheduling decision.

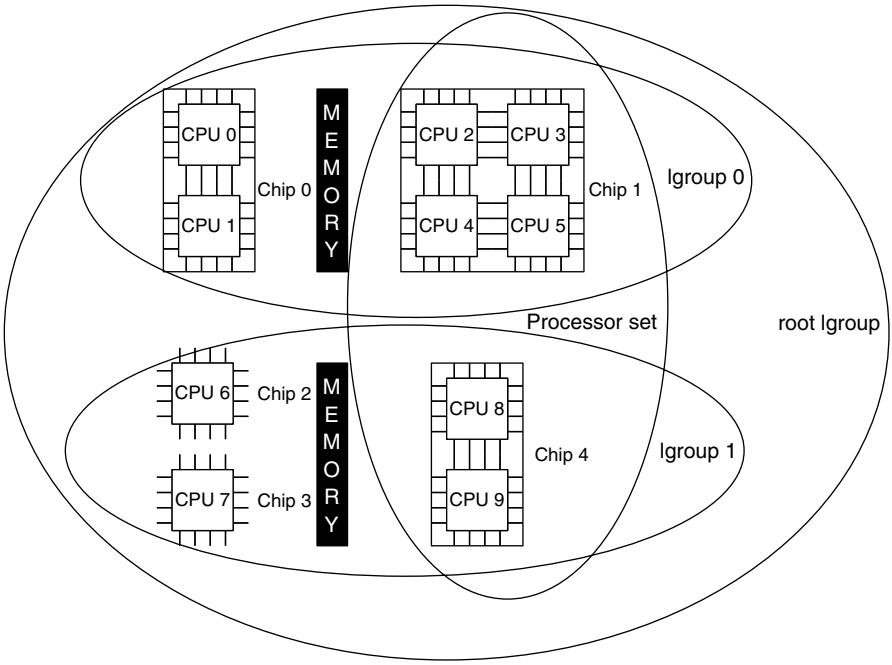


Fig. 3.6 Solaris hardware related concepts

Internally the dispatcher is structured on two layers: the core dispatcher and the scheduling classes. At conceptual level, this is very similar to the approach implemented in Linux (or rather the Linux approach is similar to the Solaris approach): scheduling classes provide the various policies for setting thread priorities while the core dispatcher takes care of allocating tasks to different CPUs, task switching and load balancing.

Solaris has 170 global priority levels of which some are reserved for specific scheduling classes (such as real-time or system) while the others (lower than 60) are managed jointly by multiple scheduling classes. On the core dispatcher level, there's a dispatch queue per CPU and each dispatch queue has an ordered sub-queue for each priority level (except for the real-time priorities, managed separately). Scheduler class level decisions will decide on which priority level a thread will be placed; the core dispatcher will decide the CPU and will take care of task switching.

Solaris supports the following scheduler classes:

- *Timeshare (TS):* the global priority of the task (and hence its chance to execute) are adjusted based on the time it spent waiting for the CPU and time spent using the CPU
- *Interactive (IA):* same as TS, coupled with a mechanism to prioritize tasks coupled with windows performing active user interaction
- *Fair Share (FS):* available processor time is allocated as shares to tasks through administrative means; this scheduling class enforces that allocation

- *Fixed Priority (FX):* tasks placed in this scheduler class will not see their global priority adjusted, hence will more-or-less execute according to a best effort approach, whenever other tasks with varying priorities make this possible
- *Real Time (RT):* have the highest priority range and guarantees the lowest possible dispatch latency
- *System (SYS):* used exclusively for OS functions and have the highest priority besides the RT class

As mentioned before, re-balancing between CPUs and task (thread) migration may happen anytime a thread's global priority is changed and it's placed in a new dispatcher queue. The high level algorithm for doing so is as follows (it's only applicable if the task is not pinned to a specific CPU):

- if the task has been waiting less than the 'cache keep warm' value of the CPU where it last executed, it will stay on the CPU (there's a good chance it still has its data in the CPU cache)
- otherwise, a CPU is chosen, taking into account the last CPU where the task was executed, its home lgroup, its CPU partition, its priority as well as which CPU is currently running the lowest prioritized task (in order to boost the chances of getting executed soon); locality is favored over priority: CPUs in the task's partition are considered in latency distance order. As we will show in the next chapter, this is a key feature that can help optimize memory access latencies as well.
- once a CPU is selected, load balancing may be performed, if the run queues on different CPUs for the task's priority level do not have their lengths within a certain span; the scope of load-balancing is also driven by the system's architecture: it may be limited to just one chip, one lgroup or it may be performed globally.

In summary, Solaris provides a scheduling framework specifically targeting large scale SMP systems and various workloads, with good support for multi-core and multi-processor systems as well as some consideration for cache performance. As it has a global approach to scheduling and load balancing, it inherently faces scalability issues as well; traditionally this has been addressed through careful configuration of processor sets and locality groups.

In the next chapter we'll briefly discuss how the concept of locality groups can help improving memory performance in distributed NUMA systems.

3.5.2.2 Memory Management and Support for NUMA Systems

Solaris' memory management is based on the Vmem allocator described earlier in this chapter as well as the slab allocator, specifically designed for Solaris.

What makes the Solaris infrastructure for memory management interesting is its built in optimization support for cache-coherent non-uniform memory access (ccNUMA) architectures as well as explicit support for chip multi-processing (both HW threads and multi-core chips). The foundation for this support is built on the concepts of locality groups and the abstract hierarchical latency model.

Conceptually, a locality group consists of hardware that is 'close' to a specific reference point, i.e., HW that is tightly integrated and has access latencies within certain limits. Hardware in this context may be processors, physical memory pages, devices or other locality groups that are included in the current locality group. This way, a latency hierarchy can be built, with the root latency group—representing the complete system—at the top of the hierarchy.

The home locality group concept of threads is used as the basic guiding principle when allocating memory in ccNUMA systems, implemented as part of the Memory Placement Optimization (MPO) Framework. There are two physical memory allocation policies: first touch and random, each targeting different memory usage cases.

First touch allocation chooses the physical memory from the home locality group of the thread that first uses it, thus assuming that this specific thread will be the main user of the memory. First touch is the default policy for private memory and underlines the importance of attempting to schedule threads on CPUs belonging to there home lgroup as often as possible: it's a simple way to improve memory access latency and bandwidth.

The random allocation policy is suitable for large shared memory allocations where the memory is likely to be accessed by multiple threads having different home locality groups. The goal is to spread out the usage of memory bandwidth, while obtaining a statistically average access time and latency. The policy can be made processor set aware, so that memory is still allocated from locality groups that are spanned by the processors in a specific processor set—this feature is useful e.g. in cases processor sets are used to isolate applications, as it will lead to a more predictable memory performance, without the applications impacting each other.

3.5.2.3 Solaris: Summary

Solaris provides a tightly integrated scheduling, memory management and SMP system management kernel architecture with a high degree of configurability and flexibility which, no doubt, contributed to its reputation of highly scalable operating system. We believe the abstract system model and the latency model are the key abstractions that make Solaris perform well on different hardware architectures.

Solaris still suffers from the same issues as any other pre-emptive, fine grained time-sharing based operating systems: potential for user task pre-emption that leads to cache trashing and degraded performance and the non-constant overhead of large scale co-ordination for systems where partitioning is not carefully planned and enforced.

3.5.3 Windows

The current Windows family of operating systems originates from two sources: the Microsoft DOS based GUI systems for personal computers and the Windows NT 32-bit operating system. Starting from NT kernel version 5.0, all Windows branded operating systems are based on the NT technology (these include Windows 2000,

XP, Windows Server 2003 and 2008, Windows Vista and Windows 7). Since 2007, with the launch of Windows Vista and Windows Server 2008 (in March 2008), all the variants—both for client and server usage—share the same core system files, including the kernel, HAL (Hardware Abstraction Layer) libraries, device drivers and basic system utilities. In our discussion we will focus on this unified version, with special priority given to the server versions, as the ones supporting multi-processor and NUMA systems. Windows Server today supports configurations with 32 or 64 processors: these limits are only related to the usage of word-sized bitmasks in certain parts of the scheduler; the largest deployment can support up to 2048 gigabytes of physical memory.

Internally, the Windows kernel is structured as a micro-kernel based hybrid architecture: the actual kernel has a limited set of functions, with most of the services grouped into a layer above called the Executive. However, these two layers share the same kernel address space and are delivered as part of the same binary. According to a recent interview with one of the key kernel engineers at Microsoft [15], in Windows 7 a new re-layering and restructuring took place in order to recapture some of the original modular design.

3.5.3.1 Scheduling and Multi-Processor Support

Windows is a *symmetric multiprocessing* (SMP) operating system with support for multi-processor and multi-core architectures and NUMA systems. Scheduling is performed by a collection of routines jointly called the *dispatcher* and it's based on a pre-emptive, priority-driven policy. Windows treats all processor cores equally in the sense that all cores capable of executing software are represented by the same abstraction, called *processor*.

There are 32 priority levels in Windows: 16–31 are reserved for real-time applications, 1–15 for normal applications while 0 is reserved for special system tasks. The kernel makes no distinction between kernel-space or user-space threads, the only deciding factor is the priority of the thread.

The priority of a thread is a combination of a base priority associated with the process to which it belongs and a relative priority assigned based on the role of the thread. The priority of a thread may be changed for various reasons, such as completion of an I/O task, after waiting on a synchronization object such as a semaphore, in case the thread has been waiting for a resource for too long, GUI threads woken up due to user activity etc. Thread boosting is also used to resolve priority inversion cases due to deadlocks.

Scheduling in Windows is based on *quantum allocation*. Each thread is allowed to run for a certain pre-configured quantum of processor cycles (the value of quantum depends on the type of Windows deployment; it's typically longer for server configurations). A context switch will occur if the quantum allocated to a thread expires, but also in cases another thread with a higher priority becomes runnable or the current thread voluntarily gives up control (due to I/O operation or synchronization). In case of pre-emption, the pre-empted thread is put at the head of the queue of runnable threads so that it can resume its execution—and consume its remain-

ing quantum—once the higher priority thread completes. In case the quantum of a thread expires, the next thread on the same priority level is chosen; if no such thread is available, the thread will receive another quantum to run.

The Windows dispatcher uses a number of structures collectively called the *dispatcher database*. Each processor has a set of local dispatcher queues (one per priority level) each containing the threads on that specific priority level that are ready to execute; access to these queues is protected by a per-processor spin lock. There's a 32-bit (or 64-bit) mask that indicates which queue has at least one thread ready (called the *ready summary*), thus a scheduling decision on which thread to run next can be done in constant time. In addition, the dispatcher database maintains some global structures such as information on which processors in the system are usable and which processors are idle; these structures are protected by a system-wide global dispatcher spinlock.

In a multi-processor system, for each thread there are a number of configuration parameters stored in the dispatcher database. A bitmask defines the processor affinity of the thread, i.e., the processors on which the thread *must* execute; in addition, each thread has two special processors associated that are defined and managed by the dispatcher: the *ideal processor,* that indicates the 'home processor' of the thread where it will be scheduled whenever it's possible and the *last processor,* indicating the processor where the thread was last executed.

Figure 3.7 gives an overview of the main elements of the dispatcher database and the principal scheduling information stored for each thread.

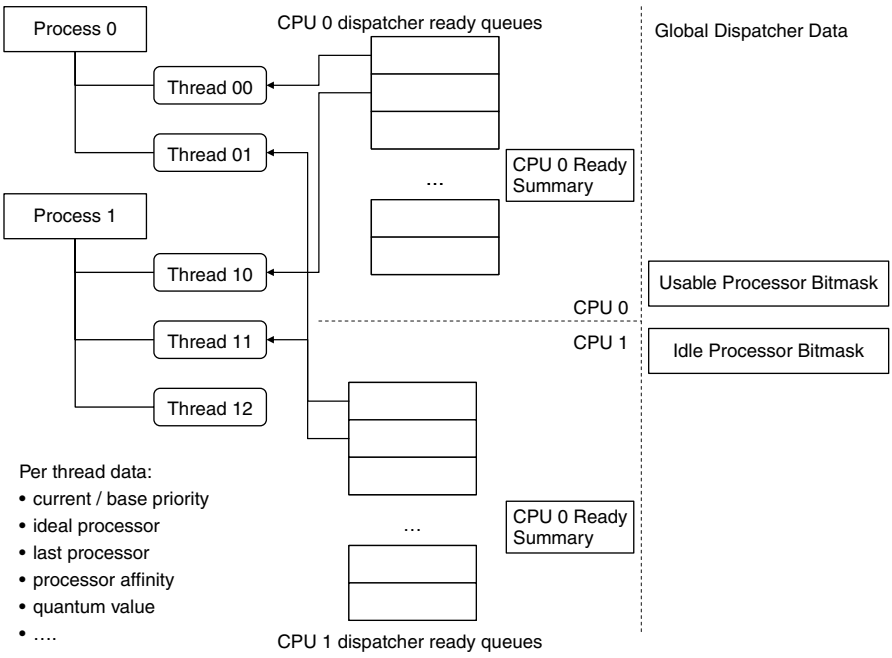


Fig. 3.7 Windows dispatcher data structure

The selection of ideal processor happens at process and thread creation, according to the following algorithm:

- In a NUMA system, an *ideal NUMA node* is selected for the process when it's started; two processes started one after the other will be allocated to different nodes
- Each thread will have an ideal processor allocated from the ideal NUMA node (as a first choice); Windows will attempt to spread the threads belonging to the same process out evenly on all available processors (the program has the option to specifically indicate a desired ideal processor)

Scheduling decisions in an SMP system are reduced in the Windows kernel to two choices:

- for a runnable thread, choose a processor to run on;
- for a given processor, choose the next thread to execute

Whenever a thread is ready to run, the Windows kernel executing on one of the cores—usually the one where the state change occurred—will attempt to choose an idle processor for the thread to execute on; if there are multiple eligible idle processors (e.g. within the ideal NUMA node associated with the thread's process), preference is given to the thread's ideal processor, then to the thread's previous processor and then to the processor where the scheduling decision is being made. The thread is inserted in the corresponding queue and the processor is notified so that the execution can be started.

If there's no idle processor, the thread will be added to the corresponding dispatcher queue on its ideal processor and that processor is notified that pre-emption is needed, in case a lower priority thread is executing. Due to this policy, Windows' scheduler does not guarantee that *all* the threads with the highest priority are executed if there are a sufficient number of processors; but it guarantees that *at least one* of the highest prioritized threads will execute. Also, Windows will never preempt and move a thread in order to allow another thread that could execute only on that specific processor to run: the thread will be queued and will have to wait until the currently executing thread is de-scheduled.

A processor will normally execute threads available on its local dispatcher queues—either placed there by the kernel executing on that processor or by kernels on other processors, as described above. If no threads are available, the processor is marked as idle and will schedule a special thread, called the system *idle thread* that will search for threads that could be taken over: it will first search within the same NUMA node, then globally. The mechanism is very similar to the one implemented in Linux and it's based on a task stealing mechanism.

Windows offers a special mechanism called *job objects* that can be used for controlling multiple processes as a group. The main purpose of job objects is to allow the definition and enforcement of resource usage limits and scheduling constraints for a set of processes: CPU time, processor affinity and scheduling class; the scheduling class associated with a job object defines the length of the quantum for each thread belonging to a process in that job object.

Over the different releases there were multiple refinements implemented in Windows' scheduler to improve scalability; however the need to use the global dispatcher spinlock for some of the operations that can occur quite frequently—e.g. for modifying a thread's priority—can have a significant performance impact in large SMP systems with a large number of threads with dynamic behavior.

3.5.3.2 Memory Management and Support for NUMA Systems

The memory management system of Windows is structured into two main layers: the *virtual memory manager*, responsible for coarse-grained allocation and management of virtual memory pages and the *heap manager*, responsible for the fine-grained allocation of memory buffers.

On the virtual memory level, Windows supports two types of pages: small ones (4–8 kb, dependent on the underlying HW) and large ones (4–16 Mb, also dependent on the HW), but the allocation granularity is always at least 64 kb. The memory manager supports page priorities—to drive the policy of which page gets swapped out if more physical memory is needed—but also explicit locking of pages to physical memory, in order to improve the performance of critical applications. On the kernel level, a similar effect is achieved by having two types of memory pools: *non-paged pool,* with all the pages resident in the memory at all times and *paged pool,* that will be managed as any other pool of virtual pages.

Windows also provides a variant of the slab allocator, called *look-aside lists*. Each list contains same-sized memory objects that can be quickly allocated whenever needed; the supply of memory is achieved by allocating new sets of pages whenever needed (similarly to the slab-based approach of the Solaris method).

The heap manager acts as a layer on top of the virtual memory manager and provides support for allocating memory at byte granularity level; obviously there will be one heap manager for each heap (there may be several heaps created by each process). The default heap manager implementation offered by Windows supports multithreading, however, it uses a global locking scheme that limits its scalability in case of large number of threads distributed across several processors. To mitigate this issue, there's a special heap manager front end offered together with the default manager, called *Low Fragmentation Heap (LFH):* it organizes buffers in buckets of different sizes and allocates memory from the best fit bucket. In order to improve scalability, LFH has a special method of placing internal structures—and thus logically splitting buckets—into so called slots (there are twice as many slots as processors). Threads are allocated to a slot by an internal component called the affinity manager and may be moved to another slot (and thus, use different set of data structures) in case resource contention is detected. More details on this mechanism can be found in Ref. [16].

In order to support NUMA systems, Windows' memory manager creates during startup a *cost graph* describing the cost of accessing memory allocated on different nodes from all other nodes. Whenever a page needs to be allocated or swapped into physical memory, the memory manager will ensure that the node with the lowest

access cost is chosen, based on the information stored in the cost graph. As a basic principle, memory needed by a thread is always prioritized from the ideal processor, both at allocation and at fault cases, even if the thread is at that point executing on another processor; the scheduling mechanism described above makes sure that the thread will eventually end up on its ideal processor and thus it will benefit from having its memory paged into that processor's RAM. Only if the ideal node is out of resources will another node be chosen—again, based on the cost graph.

The allocation of heaps and look aside lists—including the non-paged kernel pool—is also spread across multiple NUMA nodes in order to improve access time for kernels running on different processors; allocation from these pools will also ensure that the virtual address mapped closest to the allocating core will be chosen.

Beside this default behavior, the memory manager also provides support for application managed allocation of memory, by enabling explicit specification of the node from which a certain allocation request shall be serviced.

Windows' memory manager is fully re-entrant and mostly distributed, with just a few system-wide resources that need global synchronization such as the page numbering database and operations on page files. The biggest bottleneck resides within the heap manager which still relies on a global lock, not suitable for large, distributed applications. Some of the impact is mitigated through the Low Fragmentation Heap solution.

3.6 Summary

In this chapter we looked at the roles usually ascribed to operating systems: hardware abstraction and resource management. Of the resource management tasks, handling of processor resources and memory are the most important ones and we briefly described the main techniques in general, as well as the particular implementations in three main-stream operating systems: Linux, Solaris (representative for the UNIX family) and Windows Server. All these systems implement variants of the symmetric multi-processing paradigm which, as we'll see in more details in Chap. 7, fails to meet the challenges of the radically different scale characteristic to many-core systems: operating system activities related to co-ordination, scheduling and resource management can have significant negative impact on applications, not least due to the high importance of cache memory in massively multi-core systems.

References

1. Brooks F P (1995) The Mythical Man Month and Other Essays on Software Engineering, Addison Wesley
2. Tanenbaum A S, Woodhull A S (2008) Operating Systems Design and Implementation. Pearson Education

3. Silberschatz A (2009) Operating System Concepts. John Wiley & Sons
4. Tanenbaum A S (2007) Modern Operating Systems: International Version. Pearson Education
5. Krten R (1998) Getting Started with QNX 4: A Guide for Realtime Programmers. Parse Software Devices
6. Enea (2010) Enea OSE: Multicore Real-Time Operating System (RTOS). http://www.enea.com/Templates/Product____27035.aspx. Accessed 11 January 2011
7. Engler D R, Kaashoek M F, O'Toole Jr J (1995) Exokernel: An Operating System Architecture for Application-Level Resource Management. Proceedings of the 15th ACM Symposium on Operating Systems Principles: 251-266
8. Knowlton K C (1965) A Fast Storage Allocator. Communications of the ACM 8(109:623-625
9. Bonwick J (1994) The Slab Allocator: an Object-caching Kernel Memory Allocator. Proceedings of the Usenix Summer 1994 Technical Conference: 6
10. McDougall R, Mauro J (2007) Solaris Internals. Solaris 10 and OpenSolaris Kernel Architecture. Prentice Hall
11. Operating System share for 06/2010. http://www.top500.org/stats/list/35/os. Accessed 11 January 2010
12. Mauerer W (2008) Professional Linux Kernel Architecture. Wiley Publishing
13. Aas J (2005) Understanding the Linux 2.6.8.1 CPU Scheduler. http://joshaas.net/linux/linux_cpu_scheduler.pdf, Accessed 11 January 2011
14. Nellans D, Sudan K, Balasubramonian R, Brunvand E (2010) Improving Server Performance on Multi-Cores via Selective Off-loading of OS Functionalility. Proceedings of the 10th Workshop on Interaction between Operating Systems and Computer Architecture:
15. Microsoft Corporation (2009) Mark Russinovich: Inside Windows 7 (video interview). http://channel9.msdn.com/shows/Going+Deep/Mark-Russinovich-Inside-Windows-7. Accessed 11 January 2011
16. Russinovich M, Solomon D A (2009) Windows Internals 5th Edition (PRO-Developer). Microsoft Press

Chapter 4
The Fundamental Laws of Parallelism

Abstract This chapter introduces the basic laws of parallelism that have influenced the research and practice of parallel computing during the past decades. We discuss Amdahl's and Gustafson's law as well as the equivalence of the two laws; we also analyze how Amdahl's law can be applied to multi-core chips and what implications it can have on architecture and programming model research. Finally we present some of the more controversial rules and conjectures, such as the KILL rule and Gunther's conjecture.

4.1 Introduction

The theory of parallel computing looks back to a tradition of over four decades. Many of the parallel algorithms, but especially the fundamental concepts and laws of parallelism were defined as part of this effort—however remained confided to the niche domain of high performance computing for many years. All this body of research was brought to the mainstream once the inevitability of multi-core chips became clear; many of these laws guide today the practice of computer architecture and parallel programming and set the framework for scientific research.

No serious attempt at covering the field of many-core programming would be complete without a critical survey of these laws and the implications for the domain of programming many-core chips.

4.2 Amdahl's Law

Amdahl's law is probably the best known and most widely cited law defining the limits of how much parallel speedup can theoretically be achieved for a given application. It was coined by Amdahl in 1967 [1], as a supportive argument for the continuing scaling of the performance of a single core rather than aiming for massively parallel systems.

A. Vajda, *Programming Many-Core Chips,*
DOI 10.1007/978-1-4419-9739-5_4, © Springer Science+Business Media, LLC 2011

In its traditional form it provides an upper bound for potential speedup of a given application, as function of the size of the sequential fraction of that application. The mathematical form of the law is

$$Speedup = 1/(1 - P + P/N)$$

where P is the fraction of the parallelized portion of the application and N is the number of available cores. It is immediately clear that, according to Amdahl's law, any application with a sequential fraction will have an upper bound to how fast it can run, independent of the amount of cores that are available for its execution. When N approaches infinite, this upper bound will be

$$Speedup = 1/(1-P)$$

The speedup curve for various levels of parallelism is shown in Fig. 4.1.

Intuitively, Amdahl's law is quite easy to deduct. Having N processors available, the time needed for executing the parallel portion in

$$P/N,$$

the total time on N processors is then

$$T(N) = (1 - P) + P/N$$

assuming

$$T(1) = 1$$

we will have

$$Speedup = 1/(1 - P + P/N)$$

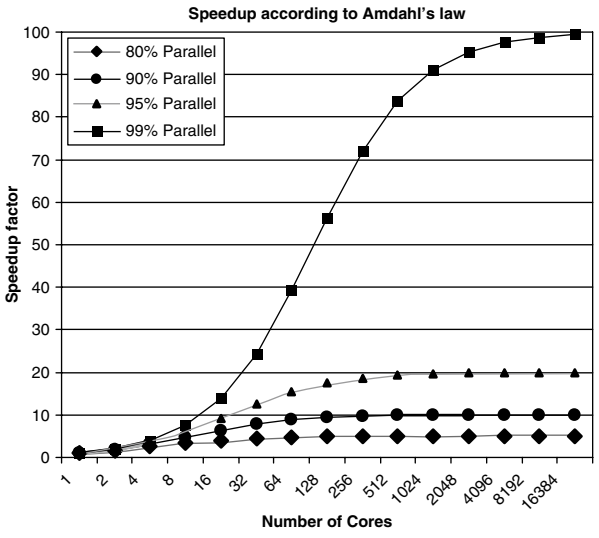


Fig. 4.1 Speedup curves according to Amdahl's law

The implications of Amdahl's law were profound. Essentially it predicted that the focus shall be on getting single cores run faster—something that was within reach for the past four decades—instead of the costlier approach of parallelizing existing software which would have anyway limited the scalability, in accordance with Amdahl's law, as long as some part of the software remained sequential.

This law also triggered groundbreaking research that resulted in innovations such as out of order execution, speculative execution, pipelining, dynamic voltage and frequency scaling and, more recently, embedded DRAM. All these techniques are primarily geared towards making single threaded applications run faster and consequently, push the hard limit set by Amdahl's law.

As multi-core chips were becoming mainstream, Amdahl's law had the clear merit of putting the sequential applications—or sequential portions of otherwise parallelized applications—into the focus. A large amount of research is dealing with auto-parallelizing existing applications; the research into asymmetric multi-core architectures is also driven by the need to cater for both highly parallelized applications and applications with large sequential portions.

As we will see in the following sub-chapter, Amdahl's law can point into interesting new research directions when applied to many-core chips. However, we cannot conclude this chapter without highlighting one of the shortcomings of Amdahl's law: it assumes that the fraction of the sequential part of an application stays constant, no matter how many cores can be used for that application. This is clearly not always the case: more cores may mean more data parallelism that could dilute the significance of the sequential portion; at the same time an abundance of cores may enable speculative and run-ahead execution of the sequential part, resulting in a speedup without actually turning the sequential code into a parallel one. The first example lead to Gustafson's law, while the second one to a new way of using many-core chips, covered in Chap. 10.

4.2.1 Amdahl's Law for Many-core Chips

The applicability of Amdahl's law to many-core chips has first been explored, on theoretical level, in Ref. [2]. It considered three scenarios and analyzed the speedup that can be obtained, under the same assumptions as in the original form of Amdahl's law, for three different types of multi-core chip architectures. The three scenarios were:

- Symmetric multi-core: all cores have equal capabilities
- Asymmetric multi-core: the chip is organized as one powerful core along with several, simpler cores, all sharing the same ISA
- Dynamic multi-core: the chip may act as one single core with increased performance or as a symmetric multi-core processor; this is a theoretical case only so far, as no such chip was designed

In order to evaluate the three scenarios, a simple model of hardware is needed. Let's assume that a chip has a certain amount of resources expressed through an

abstract quantity called *base processing unit (BCU)*. The simplest core that can be built requires at least one BCU and speedup is reported relative to execution speed on a core built with one BCU; multiple BCUs can be grouped together—statically at chip design time or dynamically at execution time as in the dynamic case—in order to build cores with improved capabilities. In Ref. [2], the performance of the core built from n BCUs was approximated using a theoretical function *perf(n),* expressed as the square root of n. It's clearly just an approximation to express the diminishing returns from adding more transistors (BCUs in our terminology) to a given core design.

In the symmetric multi-core processor case, the results were equivalent to the original scenario outlined by Amdahl's law—essentially, using homogeneous architectures, for the canonical type of applications, the speedup will be limited by the sequential fraction.

The more interesting results were obtained for the other two cases. In the single ISA asymmetrical multi-core processor case, the number of cores is reduced in order to introduce one more powerful core along a larger set of simpler cores. Mathematically, Amdahl's law for asymmetric multi-core processors has the following form:

$$Speedup(p, n, r) = 1/((1 - p)/perf(r) + p/(perf(r) + n - r))$$

Where

- n is the total number of BCUs
- r is the number of BCUs used for the more powerful core which has a performance of *perf(r),* estimated as square root of r
- p is the share of the program that is parallelized

Figure 4.2 shows the speedup curves for programs with different degrees of parallelism on a chip with 64 BCUs, organized into different asymmetric configurations. There are two conclusions that may be drawn from this chart:

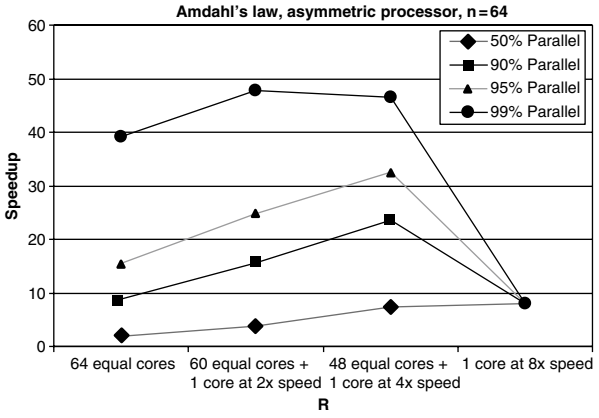


Fig. 4.2 Amdahl's law for asymmetric multi-core processors

- The speedup is higher than what Amdahl's classical law predicts: the availability of the more complex (faster) core makes it possible to run the sequential part of the program faster
- There's a sweet spot for each level of parallelism beyond which the performance will decline; for example for the 95% parallel type of application this sweet spot is reached with 48 equal cores and one core running at four times higher speed

The implication of this law is that asymmetric chip designs can help mitigate the impact of Amdahl's law; however the challenge is that different applications may have their sweet spots at different configurations. In our example (assuming this particular *perf()* function) the 48 equal cores + 1 core at 4× speed is a good approximation of the ideal sweet spot, but the result may be different for different organization of the chip and/or different application profiles.

This is what makes the third case, the fully dynamic scenario really interesting. In this scenario the HW can either act as a large pool of simple cores—when the parallel part is executed—or as a single, fast core with performance that scales as function of the number of simple cores it replaces, according to the *perf()* function. Obviously, this mode of operation is targeted at sequential portions of applications.

For this scenario, Amdahl's law has the following form:

$$Speedup(p, n, r) = 1/((1 - p)/perf(r) + p/n)$$

with the same notations as in the asymmetric case. The speedup for applications with various degrees of parallelism on a 64 BCU chip is shown in Fig. 4.3, assuming the performance of the chip—when used for the sequential part of the code—to be in the range of 1× till 8× of a simple core.

It's clear from the chart that if such a chip would be built, it could provide a speedup approaching linear scaling with the number of cores, thus essentially removing

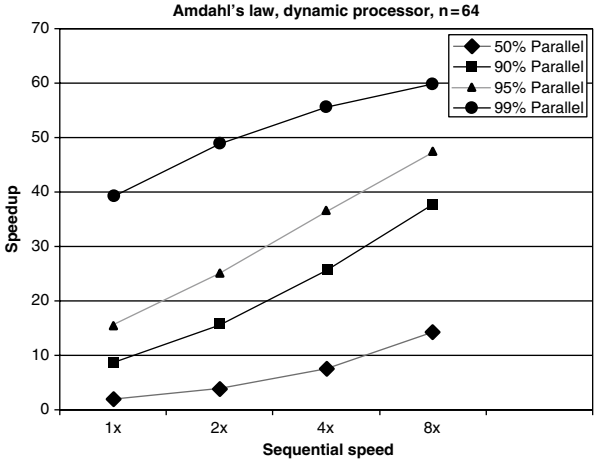


Fig. 4.3 Amdahl's law for dynamically organized multi-core chips

the limitations set by Amdahl's law. However, we don't yet know how to build such a chip, thus at first sight this application of Amdahl's law seems to be a mere theoretical exercise.

There are however two techniques that could, in theory, make N cores look like behaving as one single powerful core. The first, albeit with a limited scope of applicability, is dynamic voltage and frequency scaling, applied to one core, while the others are switched off (or in very low power mode); the second, with a theoretically better scalability is speculative, run-ahead execution.

We will elaborate on the run-ahead speculative execution in Chap. 10. Briefly, it aims at speeding up execution of single threaded applications, by speculatively executing in advance the most promising branches of execution on separate cores; if the speculation is successful, the result is earlier completion of sequential portions and thus a speedup of execution. The grand challenge of speculative execution is the accuracy of prediction: too many mis-predictions decrease the ratio of useful work per power consumed, making it less appealing while delivering a limited amount of speedup. On the other hand, it does scale with the number of cores, as more cores increase the possibility of speculating on the right branch of execution. As we'll show in Chap. 10, the key to efficient speculation is to dramatically strengthen the semantic link between the execution environment and the software it is executing.

Amdahl's law, along with Gustafson's law that we will introduce next, is still at the foundation of how we reason about computer architecture and parallel programming in general and the architecture and programming of many-core chips in special. At the core of it, it defines the key problem that needs to be addressed in order to leverage on the increased computational power of many-core chips: the portions of the program that are designed to execute within one single thread.

4.3 Gustafson's Law

John L. Gustafson first described the law that became to be known as Gustafson's or Gustafson-Barsis' law in1988, in a paper published in the *Communications of the ACM* [3]. Mathematically, it states that the speedup that can be obtained on N processors, in case the sequential fraction of the application is *np,* is

$$Speedup = N - (N - 1) * np$$

Apparently, this result is in conflict with Amdahl's law, as it may suggest that applications can actually benefit from unlimited parallelism, without an upper bound. In Gustafson's original paper [3], the law was presented as an exception, linked to the diluted role of the sequential portion in relation to the parallel portion as the size of the problem and the number of available processors. As we'll show in this chapter, the two laws are in fact equivalent, thus at the core of it we really have just one fundamental law describing how performance scales up depending on the nature of the application and the share of the sequential portion.

Intuitively, Gustafson's law can be obtained through a simple deduction. Assuming that the parallel execution of a program on N processors takes one unit of time and it's made up of

$$np(N)\text{: sequential portion}$$
$$p(N)\text{: parallel portion}$$

thus we have

$$T(N) = np(N) + p(N) = 1$$

The same program, executed on one processor, will have an execution time equal to

$$T(1) = np(N) + N * p(N)$$
$$T(1) = np(N) + N - N * p(N)$$

The speedup is given by

$$Speedup = T(1)/T(N)$$

which yields the following result

$$Speedup = N - (N - 1) * np(N)$$

which is exactly Gustafson's law. Figure 4.4 shows the speedup curves for applications with various degrees of parallelism.

The apparent contradiction between Amdahl's law and Gustafson's law results from the subtle difference in how the fraction of sequential part is defined. This subtle difference is the root cause around much of the confusion surrounding the two laws; as we'll elaborate in the following sub-chapter, both laws are expressions, written in different terms, of the same inherent law of parallelism.

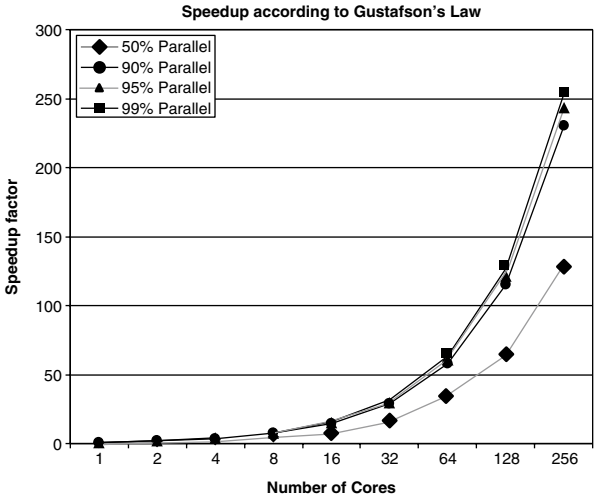


Fig. 4.4 Gustafson's law

4.4 The Unified Amdahl-Gustafson Law

One of the often omitted facts—and a source of some confusion—is the mathematical equivalence of Amdahl's and Gustafson's laws, even though the first proof of this equivalence was provided back in 1996 [6]. The fundamental observation that enabled resolving the apparent conflict between the two laws was that the *fraction of sequential part* is defined differently for the two laws, however these two approaches can be linked by a simple formula.

To illustrate these differences, let's define the following quantities:

- *ts(1):* the execution time of the sequential portion of the application on one processor
- *tp(1):* the execution time of the parallel portion of the application on one processor
- *tp(N):* processing time of the parallel portion of the application on N processors; it's equal to *tp(1)/N*
- *T(N):* total execution time on N processors $(N \geq 1)$

With these definitions, we obviously have

$$T(1) = ts(1) + tp(1)$$
$$T(N) = ts(1) + tp(N)$$

The key difference between Amdahl's law and Gustafson's law is how the share of the sequential part is defined. Amdahl's law uses the *non-scaled share* quantity defined as

$$np\_ns = ts(1)/T(1) = ts(1)/(ts(1) + tp(1))$$

i.e., the ratio between the execution time of the sequential part on a single processor and the total execution time on a single processor. By contrast, Gustafson's law uses the *scaled share,* defined as

$$np\_sc(N) = ts(1)/T(N) = ts(1)/(ts(1) + tp(N))$$

i.e., the share of the execution time of sequential part of the application of the total execution on N *processors* (hence it's called the scaled share).

Both laws obviously define speedup as

$$speedup = T(1)/T(N)$$

Expressing this formula as function on *np_ns* we get

$$speedup = 1/(np\_ns + (1 - np\_ns)/N)$$

which is the classical form of Amdahl's law.

Expressing the same formula in terms of $np\_sc(N)$, the result will be

$$speedup = ts(1)/(ts(1) + tp(N)) + tp(1)/(ts(1) + tp(N))$$

rewritten in terms on *np_sc(N)*:

$$speedup = np\_sc(N) + N * (1 - ts(1)/(ts(1) + tp(N))$$
$$speedup = np\_sc(N) + N * (1 - np\_sc(N)) = N - (N - 1) * np\_sc(N)$$

which is the traditional formulation of Gustafsson's law.

As the two speedup factors—obtained through the two laws—shall be mathematically equivalent, we can establish the relationship between np_ns and np_sc(N). Indeed, starting from

$$1/(np\_ns + (1 - np\_ns)/N) = N - (N - 1) * np\_sc(N)$$

we obtain:

$$np\_ns = np\_sc(N)/(np\_sc(N) + N * (1 - np\_sc(N)))$$
$$np\_sc(N) = np\_ns/(1/N + np\_ns * (1 - 1/N))$$

For a practical illustration, let's assume that we have $N=10$, and an application or which $np\_sc(10)=0.3$, i.e., 30% of the time is spent on executing the sequential part of the application. If we assume $T(10)=10$, then $T(1)$ will be 3 (sequential part) + 10 × 7 (10 processors and parallel time was 7), so $T(1)=73$. In consequence, *np_ns* will be

$$np\_ns = 3/73 = 0,0411$$

Using *np_sc*(10) in Gustafson's law and *np_ns* in Amdahl's law we will obtain

$$speedup = 10 - (10 - 1) * 0.3 = 7.3 \ (according\ to\ Gustafson's\ law)$$
$$speedup = 1/0.0411 + (1 - 0.411)/10 = 7.3 \ (according\ to\ Amdahl's\ law)$$

The major question raised by the proven equivalence of the two laws is whether the scale-up potential is limited (as implied by Amdahl's law) or not (as proposed by Gustafson's law) by the sequential portion of the application. The answer, unsurprisingly, is application dependent: for some applications, the time share of the sequential part will stay constant as the number of available processors increases—as assumed in Amdahl's law—and thus these applications will have an upper bound for their scalability; for others, the time share of the sequential portion will diminish as the number of available cores increases and consequently, will continue to gain speedup as more cores are added.

An interesting class of applications is those where *supra-linear speedup* can be obtained, at least in comparison with a specific sequential implementation. This however is a bit of catch 22: while it may be possible to prove supra linear speedup, the newly parallelized version will perform better on a single processor than the original sequential version. These types of sequential implementations are sometimes called *non-structure persistent algorithms.*

The simple conclusion of our analysis is that there is really just one law—we call it the Amdahl-Gustafson law—that sets un upper bound on how much an application may be sped up using N cores; as N increases, this upper bound may or

may not be limited to N, depending on the nature of the application and how the sequential portion compares with the parallelized part. The basic fact remains also in place: ultimately, it's the sequential part that needs to be tackled in order to bend the speedup curve closer to linear.

4.5 Gunther's Conjecture

Gunther's law [4]—called Universal Scalability Law by it's proponent, Neil Gunther—is actually a conjecture (empirical formula) based on measurement data that aims at modeling performance scalability, taking into account also the impact of synchronization and other sources of delay related to maintaining a coherent state of the system, in order to give a more accurate picture of the scalability curve of a given piece of software on a number of processors. Its mathematical formula is

$$C(N, s, k) = N/(1 + s * (N - 1) + k * N * (N - 1))$$

Where

- N is the number of processors
- s is a parameter modeling contention (such as the sequential portion of an application), with values between 0 and 1
- k is a parameter modeling coherence, i.e., the extra cost due to synchronization and any other coordination activity that is not strictly related to the implementation of the algorithm.

If k is zero, Gunther's conjecture defaults to Amdahl's law, thus it can be perceived as a generalization of Amdahl's law that takes into account the overhead cost due to the need to synchronize among multiple cores. It is capable of modeling the retrograde performance phenomena that is often observed in practice: many applications stop gaining performance after a certain number of cores, even decreasing in performance as more cores are available. This 'sweet spot' happens, according to Gunther's conjecture for

$$NLimit = sqrt\ ((1-s)/k)$$

i.e., for any $N > NLimit$, the performance will be lower.

Clearly, for $k=0$, we will not see retrograde performance scaling (the law falls back on Amdahl's law); for fully parallelized applications ($s=0$), the coherency parameter will alone define the point where the performance will turn retrograde.

Intuitively, applications can be classified according to the values of s and k specific to them. The main categories are

- $s=0$, $k=0$: ideal case, with linear scaling of performance with the number of cores
- $s>0$, $k=0$: contention limited scalability due to serialization (according to Amdahl's law)
- *any* s, $k>0$: coherency limited scalability due to the overhead of synchronization between cores

The major drawback of Gunther's conjecture is that it does not give any indication with respect to *what exactly* is modeled through *k,* what the absolute values of *k* may mean and *how* the value of *k* may be estimated for a given architecture and possibly a given software architecture. Large sets of empirical data have indicated that it indeed approximates very well the performance curve of different applications (in the sense that a value pair (*s, k*) can be found that will produce the same curve as the one drawn based on the empirical data); it remains an interesting research subject to design a method that can estimate these values *a priori* and provide a framework for reasoning on how architecture and software need to be changed so that the values of *s* and *k* will be as close to zero as possible.

4.6 The Karp-Flatt Metric

The Karp-Flatt metric [5] was proposed in 1990 by Alan H. Karp and Horace P. Flatt as a practical way to estimate how well an application scales with the number of processor cores available to it. It assumes that the speedup *S* on *N* processor cores versus the execution on one core can be measured and consequently used to estimate the sequential, non parallelized part of the application.

Mathematically, it provides the following formula:

$$np = (1/S - 1/N)/(1 - 1/N)$$

where *np* is the estimated non-parallelized fraction of the application; obviously, the lower the value of *np*, the better. A simple deduction shows that the Karp-Flatt metric is consistent with Amdahl's law, in the sense that extracting the sequential fraction as function of speedup and number of cores from Amdahl's law yields the same result.

The Karp-Flatt metric provides a simple yet powerful tool to assess the reasons for limited scalability. For any application, running in an ideal environment, with no memory, cache, communication or synchronization overhead, the sequential fraction shall be lower or equal than the fraction of sequential portion defined for Amdahl's law; any value larger than that points to some sort of inefficiency in the way the hardware is used or how the synchronization mechanisms are designed.

While not a law in the proper sense of the term, the Karp-Flatt metric is an essential quantitative measure of parallelism, with wide applicability in the domain of multi-core programming.

4.7 The KILL Rule

The KILL rule [6] is a qualitative empirical law—rather, a rule of thumb—introduced by MIT researchers (the same group that developed the Tile concept and multi-core architecture) to guide the development of multi-core hardware. It stands for Kill If Less than Linear and has the following definition: any architectural feature—such

as out-of-order execution, pre-fetching schemes etc—shall only be implemented on a processor core level if it yields a linear speedup for some relevant application in relation to chip area, power budget or transistor count required for implementing it. Consequently, if this requirement is not fulfilled, the feature shall be scrapped and the resources used for other features or addition of more cores.

This rule was clearly one of the guiding principles behind the development of the Tilera chip. It is, however, not universally endorsed by everyone, as it does not answer the all important question of how the performance of single threaded applications can be boosted. In the context of applications with limited parallelism, even innovations with sub-linear performance gains may be highly desired, for lack of alternative solutions.

We believe however that, in the context of applications that can make use of massive amount of cores, the KILL rule is an important guiding principle for designing many-core chips and deciding on the right dimensioning of cores, on-chip communication mechanisms and memory architectures.

4.8 Summary

This chapter's goal was to familiarize the reader with the fundamental laws of parallelism and the relationship between these. Some of these laws—for example the KILL rule—are not universally accepted, yet it provides a good practical guidance for understanding how some of the many-core chip designs emerged. Others, such as the two fundamental laws have had and continue to have a defining impact and shall always be kept in mind when programming many-core chips. To paraphrase an IBM engineer: *everyone knows Amdahl's law, but conveniently pretends it does not exist*.

References

1. Amdahl G (1967) Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. American Federation of Information Processing Societies (AFIPS) Conference Proceedings 30:483-485
2. Hill M D, Marty M R (2008) Amdahl's Law in the Multi-core Era. IEEE Computer
3. Gustafson J L (1988) Reevaluating Amdahl's Law. Communications of the ACM 31(5): 532-533. Online at http://www.scl.ameslab.gov/Publications/Gus/AmdahlsLaw/Amdahls.html. Accessed 11 January 2011
4. Gunther N J (2002) A New Interpretation of Amdahl's Law and Geometric Scaling. http://arxiv.org/PS_cache/cs/pdf/0210/0210017v1.pdf. Accessed 11 January 2011
5. Karp A H, Flatt H P (1990) Measuring Parallel Processor Performance. Communications of the ACM 33(5): 539 - 543
6. Shi Y (1996) Reevaluating Amdahl's Law and Gustafson's Law. http://www.cis.temple.edu/~shi/docs/amdahl/amdahl.html. Accessed 11 January 2011

Chapter 5
Fundamentals of Parallel Programming

Abstract In this chapter we will look at some of the fundamental concepts and techniques specific to parallel programming. We do not aim at providing an all-inclusive detailed description of parallel computing as that would require much more than the total length of this book; rather we aim at presenting a theoretical framework on which later chapters, dealing with programming many-core chips can build on. We will consider, on conceptual level, the fundamental types of parallelism, the synchronization concepts as well as how these are implemented and supported in modern computing systems. Finally, we describe the basic patterns for architecting and designing parallel software.

5.1 Introduction

The history of programming parallel computers—systems with multiple processors—goes back several decades. For much of this time, it was confined to the domain of high performance and scientific computing: the author of this chapter remembers how surprised some of his colleagues were when he took an optional undergraduate course in parallel programming, as the whole area was considered a sort of overspecialized, almost esoteric field. Understanding the basic concepts of parallel programming—decomposition and synchronization—and the techniques to achieve these has however became an indispensable body of knowledge for any programmer planning to write software for multi-core chips.

5.2 Decomposition and Synchronization

The execution of any software on multiple processors requires some form of *decomposition* of the computation into multiple chunks of smaller entities that can be executed in parallel on the available processors. Correspondingly, these chunks shall in some way *synchronize* with each other in order to generate the final result

A. Vajda, *Programming Many-Core Chips,*
DOI 10.1007/978-1-4419-9739-5_5, © Springer Science+Business Media, LLC 2011

of the computation. Without decomposition and synchronization, parallel programming—save for certain forms of speculative execution—would be inconceivable; therefore the methods of decomposition and synchronization are at the foundation of parallel programming. In this section we will focus on the main types of decomposition and synchronization and how these relate to the basic concepts provided by the hardware.

There are fundamentally just two types of decomposition—all other flavors ultimately fall back on these two. *Functional decomposition* aims at dividing up the overall computation along functional dimensions; for example, in order to calculate something like $h(x)=f(x)+g(x)$, the execution of f and g may be done in parallel on two processors (assuming that no other dependencies exist). *Data-based* decomposition on the other hand targets the parallel execution of the same computation but on multiple data instances; the classical example is the application of the same function f on all elements of an array in parallel.

Obviously, these two methods of decomposition have different applicability, depending on the problem domain. When working with massive amount of data, or designing a system based on independent events (e.g. a telecommunication system), data based decomposition is the natural first choice; for computations made up of many simple computations that may be executed independently—at least most of the time—functional decomposition is probably the right path to go.

5.2.1 Functional Decomposition

Functional decomposition can be of two distinct types: static and dynamic. For static decomposition, the example given above is the typical one—it's known at design time that h is easily decomposed into f and g and the system is designed accordingly. The primary issue related to static decomposition is the lack of scalability when the number of processor cores is increased. Staying with the same example, it's impossible to use more than two cores, unless f or g is further decomposed—a task that would have to be performed every time more cores become available. As a consequence, while widely used in embedded systems, static functional decomposition is rarely used for more than as first step break down of applications before other techniques are applied.

Dynamic functional decomposition on the other hand has good speedup potential and it's one of the widely used methods to gain core count agnostic decomposition. The core technique is to generate new *tasks* whenever possible, tasks that can then be executed by any available processor core. Dynamic functional decomposition is thus essentially one flavor of the task based model that we will discuss later on: whenever possible, instead of executing the chunk of code directly, a new task is created that can be picked up and executed by any of the available cores. This model is essentially independent of the number of available processor cores, as the partitioning of computation is done independently of where and when the tasks will be executed and thus can result in better speedup. It's important to note however that

dynamic functional decomposition is not the only form of the task based model: data parallelism can also yield task based models.

While static functional decomposition has limited applicability, dynamic decomposition has recently enjoyed renewed interest, primarily due to the popularity of the task based models, such as OpenMP [1], Cilk [2] or Intel's thread building blocks [3]. We will explore these models in more details in Chap. 9.

5.2.2 Data-Based Decomposition

The basic idea of data-based decomposition is to perform the same computation on different sets of data in parallel, on multiple processor cores. The typical example is the mapping of a function onto each element of an array: as long as there are no dependencies between the different elements, the function may be calculated in parallel for each of these, on different processor cores.

Another, less obvious, case of data-based decomposition is certain type of event-driven systems, where events may trigger computations that may take longer time to execute. A typical example of this case is a telecommunication system, where an event (initiation of a call) may trigger data processing (e.g., speech compression) that will last as long as the call is ongoing. At the core of it, this case is an example of the task based model applied in a data parallel context, where the event (initiation of the call) triggers the task (speech compression).

A special case of data-based parallelism that incorporates certain elements of static functional decomposition is the *pipeline model*. The main characteristic of the pipeline model is the shifted processing of data: while processing does happen in parallel, it is functionally decomposed into several steps and different data elements may be at different stages of the processing. The key characteristic of the pipeline model is that each processor core is executing just one type of processing (while there may be several cores executing the same function), data is handed over to the next stage and the same computation is performed on the next available chunk of data. The concept of pipelined execution is shown in Fig. 5.1.

5.2.3 Other Types of Decomposition

For some problem domains, neither functional, nor data-based decompositions provide acceptable results. Typical examples include finite state machines or the more common Huffman decoding algorithm: in both cases, there's a chain of dependency in how and on which data the next computation shall be performed, hence parallel execution becomes difficult or impossible.

The most common form of parallelization for this class of problems is the usage of what commonly is referred to as *helper computation* (see Refs. [4, 5] for examples). A helper computation is usually not part of the main computation, but it

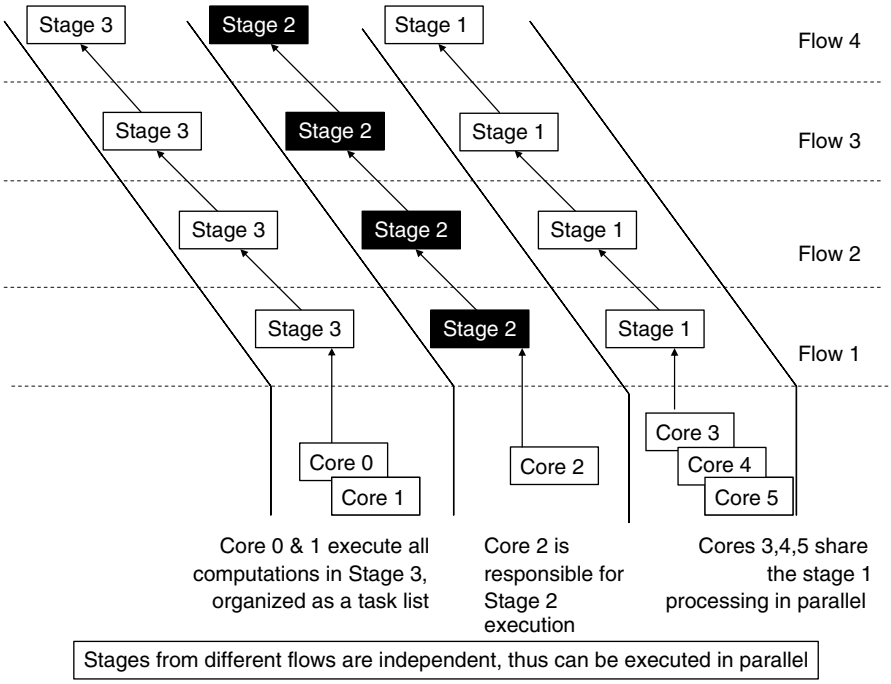


Fig. 5.1 Pipelined execution and scheduling

helps the main computation progress faster; examples include data usage prediction and pre-fetching (in order to minimize memory access latency), pre-computation of the most probable computations as well as all variants of speculative and run-ahead execution. Especially the later is believed to hold great potential, which however largely failed to materialize so far.

5.2.4 Synchronization

Independent of how a program is decomposed into computations that can be executed in parallel, there is usually a need for the parallel computations to interact: access the same data, exchange information, wait on each other—in general, to synchronize. More than anything else, minimizing the cost of synchronization is one of the central issues that the parallel programming community has to tackle.

There are basically two reasons why parallel computations need to synchronize. The first one stems from *functional dependency*, also called *data dependency* in some contexts: in order to continue, some of the computations *need the results of other computations*—hence need to wait and synchronize on the termination of the computations these computations depend on. This type of dependency naturally

stems primarily from functional decomposition—Google's MapReduce algorithm is a prime example.

The second reason computations need to synchronize is *resource contention*. Except for some special cases, computations belonging to the same program will share some resources—primarily data—and are likely to need access to system resources (such as peripherals) where may compete with other computations as well. Synchronizing access to shared memory areas is one of the most researched and debated subjects in the computer science community and a bewildering variety of solutions have been proposed; the bottom line however is that eventually access to the resource needs to be serialized, which implies that computations will have to queue and wait—an unwanted delay that needs to be minimized in order to improve performance.

5.2.5 Summary

Using multiple processor cores requires decomposition of the computation, while decomposition generates the need to synchronize the parallel computations and may give rise to resource contention situations. In this sub-chapter we briefly explored the three main classes of decomposition as well as the main sources for the need to synchronize. In the following sections we will briefly survey how these fundamental techniques are materialized in practice and which hardware and software concepts are used to support the implementation.

5.3 Implementation of Decomposition

The goal of decomposition is to create tasks that can be executed in parallel on multiple processor cores. When discussing how decomposition is mapped to the hardware and system software (operating system and potential middleware), it's important to distinguish between how the operating system views the application and how the application can act within that frame.

The basic concepts on the operating system level, when it comes to application level parallelism, are those of *process* and *thread*. Both are used to model parallel computations, but there are two fundamental characteristics along which these two concepts are differentiated: hierarchy and isolation. First of all, there is a hierarchical dependency between processes and threads: in most operating systems, processes are made up of multiple threads that share the same virtual address space (the address space of the process), while processes are strictly isolated in terms of address space and are handled as equal entities by the operating system. In most cases, each process is required to contain at least one thread and thus the process is viewed—and used—as the unit of isolation only, while parallelism related mechanisms are implemented primarily based on the concept of threads. For most operat-

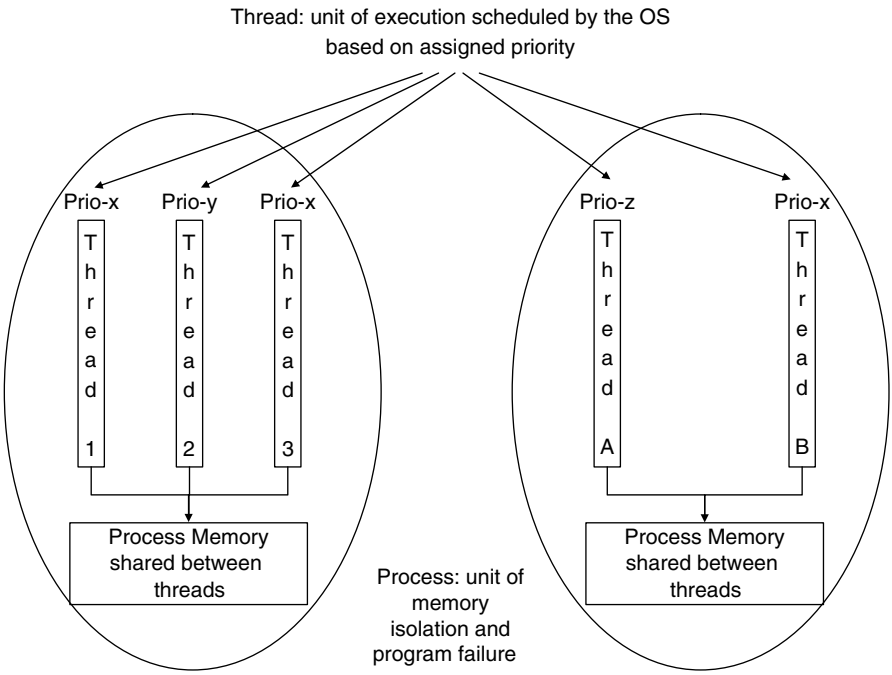


Fig. 5.2 The concepts of process, threads and priorities

ing systems, processes are also the unit of failure, as the process is the smallest isolated entity that can be safely and predictably managed by the operating system. In summary, processes can be regarded as units of isolation, especially when it comes to memory and fault management, while threads are the operating system level units of concurrency, where threads belonging to the same process share resources, primarily the same memory space. Processes and threads also have an associated *priority*, which guides the operating system in the prioritization of the access to the computing resources. Figure 5.2 illustrates the relationship between threads, processes and priorities.

Any application level partitioning strategy shall be mapped to these three fundamental concepts; applications may use the concepts of thread, process and priority to implement various approaches including synchronization and scheduling mechanisms.

Before discussing how different types of partitioning approaches could be mapped to threads, it's important to clarify one important terminology aspect. The concept of *task* has been used for several purposes, both on the operating system and application level. In this chapter we will use the term task to denote an application level chunk of computation that *may* be executed in parallel with other tasks. Thus, a task is an application level unit of concurrency, as opposed to threads which are operating system level constructs, used to implement tasks.

Static functional decomposition can directly be mapped to threads: each chunk of computation—task—will be assigned to a thread and operating system level scheduling will make sure that the threads will be executed in parallel whenever possible. In this case the number of threads is defined in the design phase by the number of computation chunks, irrespective of the number of available cores. We will call this approach *application driven thread pool sizing*.

Dynamic functional decomposition on the other hand demands a different kind of threading strategy. The number of tasks may not be known at design time and it also may be higher than the number of available cores. Thus the only viable approach in this case is to apply a *target driven thread pool sizing,* in the sense that the number of threads will depend on the number of available cores (normally, the number of threads shall be equal or higher than the number of cores).

This strategy leads however to the question of who will take care of allocation (or mapping) of the tasks to threads. There are multiple examples of addressing the task scheduling problem, of which we will survey some of the more representative ones in Chap. 9; essentially the most common approaches are based on either operating system level solutions (such as in MacOS' Grand Central Dispatch [6]) or language based middleware (such as the Cilk [2] run-time system).

For both types of functional decomposition, the thread pool sizing is essentially static or semi-static (decided at the start of program execution). For the data-based decomposition method however the thread pool sizing tends to be much more dynamic: a new thread is usually created as new data becomes available (up to an architecture dependent limit). A notable exception is the pipeline model, where threads may be pre-allocated for each pipeline stage; however, even in this case new threads may be created for each pipeline stage as multiple data elements become available. We will call this threading strategy *semi-dynamic thread pool sizing.*

The speculation based partitioning schemes usually require a completely dynamic thread pool sizing policy, as threads will need to be created quickly and will probably have a very short lifespan. Speculative execution techniques usually require very specialized middleware, operating system or even hardware level solutions with tailor-made threading mechanisms.

5.3.1 Summary

Threads, processes and priorities are the fundamental operating system level mechanisms that the programmer may use to map decomposition techniques to concrete implementation. As we saw, the difference really lies in the granularity and dynamicity of thread management, with different decomposition approaches requiring different dimensioning and timing of threads and thread creation. We would like to emphasize that the methods described in this chapter are solely general rules of thumb that may or may not be always implemented as such in practice. We urge the reader to use this chapter as a guidebook, rather than as a strict specification of threading rules.

5.4 Implementation of Synchronization

This chapter is dedicated to the most commonly used techniques of inter-thread synchronization. As application level tasks are mapped, one way or the other, to threads, the problem of task synchronization is consequently mapped to the problem of inter-thread synchronization.

At the foundation of any type of synchronization there will be a computational resource to which all the threads that need to synchronize have access. This computational resource may have different forms, but the lack of it would mean that the threads in question would not be able to communicate and hence any synchronization would be impossible.

As we saw earlier, there are two basic reasons why application level tasks—and consequently, threads—need to synchronize: functional dependency and resource contention. There is a third, more subtle form of concurrency control that is however extremely important in some cases: the need to express the need for certain chunks of code to execute completely, uninterrupted by other tasks (these chunks of code are usually called *critical sections* or *atomic sections*). While the solutions to the three cases of synchronization may share common elements, the logical constraints are different.

For functional dependency synchronization, the threads need to *communicate* and inform each other when a certain milestone has been reached by one of the threads. Handling of resource contention on the other hand requires *guardian* functionality: threads do not necessarily need to communicate, but a mechanism needs to be in place to guarantee that only the allowed amount of resource user threads have access to the guarded resource. *Critical sections* require a global synchronization mechanism that prevents other threads to access the same resources while the code in the critical section is being executed.

While there is a wide range of communication methods between threads, essentially all of these can be reduced to a message-based approach coupled with some form of the *Observer* pattern: the observed thread will notify its observer(s) whenever certain conditions are met. This notification may be done in several ways, we will list here the most common ones—all others will implement a variant of these:

- *Using an OS or middleware level message passing scheme, such as MPI* [7], *signals or similar mechanisms*. In this case, the notification may include the actual information the observer is interested in or could point to a shared storage where the information is available (it may be a shared memory area or a file on persistent storage)
- *Using a synchronization mechanism, such as a lock or similar construct*. In this method, the observed thread shall hold the ownership of the synchronization object until its results are available; the observers shall regularly attempt to acquire ownership of the synchronization object (poll it) until successful—possible only if the observed thread released its hold, indicating that the required information is available. As in the message passing case, the actual information will be stored in some additional form (shared memory or file on persistent storage)

Thus, communication between threads is implemented either through messaging with 'share nothing' semantics or some form of shared memory semantics coupled with synchronization primitives. This underlines the observation we made at the beginning of this sub-chapter: at implementation level, synchronization solutions due to functional dependencies and resource contention will share some of the mechanisms.

Handling of resource contention requires a *guardian function* that can police the access to the guarded resource. The concrete implementation will depend on the cardinality of the resource, in terms of the number of threads that are allowed to access the resource in parallel: for a cardinality that's higher than one, special solutions are needed; for a cardinality of one, several methods can be used, as we'll explain briefly.

It's important to discuss the fundamental functionality that is required of the hardware in order to be able to provide an efficient implementation. Essentially, there shall be a way to *atomically execute at least two operations on a certain memory location*, as a mandatory pre-requisite for being able to safely verify and set the value of a memory location in a parallel environment. Atomicity is required for two reasons: to prevent pre-emption during the operation of the guardian functionality and to safeguard against concurrent execution on multiple processors. The typical atomic operation supported by modern hardware is *test-and-set*: atomically verify the value of a memory location and, if the condition is true, update it to a new value. Using such an atomic operation, safe checking and update of variables used for implementation of synchronization primitives can be implemented. Other similar atomic operations are *fetch and add* and *compare and swap*. For a more detailed discussion of the hardware level synchronization mechanisms, see Chap. 2.

The most common constructs used for implementing resource guardian functions are various types of locks, semaphores, monitors, conditional variables and transactional memory. Together with critical sections, message passing mechanisms and the shared memory paradigm, these form the foundation of inter-thread synchronization and we'll describe briefly each of these in the following chapters.

5.4.1 Locks

Locks are probably the most used and most debated synchronization mechanisms; in fact, locks are at the foundation of many other synchronization primitives.

A lock is essentially a resource that can only be owned simultaneously by just a few execution threads (typically just one). It can be viewed as an object that exposes two mandatory and one optional method: *acquire, release* (mandatory) and *test* (optional), with the following semantics:

- *acquire* sets the owner of the lock to the calling thread if the lock is free, otherwise will block the calling thread
- *release* releases the ownership of the lock and consequently other threads waiting to acquire the lock may gain ownership of it

- *test* checks—without blocking the caller thread—whether the lock is acquired by a thread or not

The main usage of locks is *mutual exclusion*: the goal is to secure exclusive access to the protected resource. For this reason, a lock is sometimes also called *mutex,* but essentially it's just a different name for the same concept.

One feature of locks is that a lock is usually not associated by default with any resource—the scoping is left for the application to decide. The usual way to use locks is to allocate a lock as a guardian for each logically related set of resources (such as memory) and require that each thread that needs to access that set of resources acquires the associated lock. The scoping of locks is sometimes refined also along the lines of operations that are protected: it may be that only writes shall be protected, while any thread is allowed to access for reading the protected resource (in this case the lock is called a write lock).

There are several special cases of locks. *Spin-locks* have the following semantics: attempts to acquire the lock will not block the calling thread, instead return an indication of failure—then the thread may decide to attempt acquiring the lock later—spin until it succeeds (the same effect could be achieved through the use of the *test* and *acquire* methods, but these will have to be implemented as one atomic action). Spin locks have the advantage of preventing pre-emption by the OS (due to blocking while waiting for the lock), but it may have adverse effects if the thread has to spin too long and prevent other threads to progress, by keeping the processor for itself. *Reader/writer locks* allow multiple readers to access the protected resource for reading, while there may be only one thread modifying the resource (while there are no threads holding a reader lock); these types of locks provide better concurrency behavior as multiple threads could access the resource for reading, as long as no write is being performed (think of a rarely updated, but often used configuration parameter).

One of the basic problems with locks relates to the *lack of composability*. Given two pieces of software that individually perform well, these may fail to do so when combined in certain ways. Consider the case in Fig. 5.3: if function *f1* acquires locks A and B in this order, while function *f2* acquires locks A and B in reverse order (B then A), the two functions executed in parallel may lead to a situation where *f1* has acquired lock A, *f2* has acquired B, but both fail to continue, as will be mutually waiting on the other function to release the acquired lock; this situation is called *dead-lock* and it's one of the trickiest issues to handle in a parallel program. Several dead-lock resolution methods have been proposed but all of these tend to have hefty overhead and often fail to resolve all cases. In the chapter on shared memory we'll describe such an algorithm, applicable for the cases when locks are used to protect access to shared memory areas (the dominating case).

Another issue characteristic to locks is *priority inversion,* which occurs in cases a lower priority thread holds a lock needed by a higher priority thread. The common way to deal with such cases (e.g. in Solaris) is to *temporarily lend* the lower prioritized thread the same priority as the higher prioritized thread competing for the same lock and thus guarantee that the lock will be released as soon as possible.

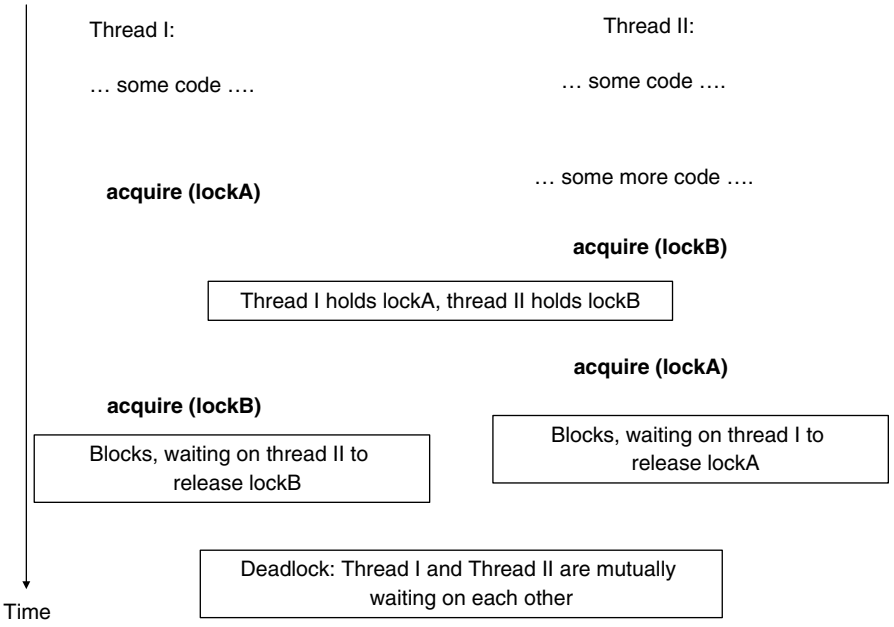


Fig. 5.3 Dead-lock example

5.4.2 Semaphores

Semaphores were first proposed by Dijkstra [8] and are in fact a generalization of locks. In its simplest form as a binary semaphore with two values (allowed/denied), a semaphore is equivalent to a lock: it either allows a thread to pass it (if it's in the allowed state) or blocks the thread until its state changes to allowed. There are opinions claiming otherwise based on established usage patterns, but in our view, there's no semantic difference between the two concepts that would justify such an arbitrary distinction.

The more interesting variant of semaphores is the *counting semaphore.* A counting semaphore allows *up to N* threads to pass it (the binary semaphore has $N=1$) and it's primarily used for controlling access to a resource that can accommodate up to N users at a time. A simple analogy to describe the meaning of semaphores is that of a restaurant with N tables: when all tables are occupied, newly arrived guests have to queue at the front desk until one of the previous guests leaves—the restaurant's tables are the protected resources, while the counter at the entrance is the semaphore protecting the resources. The two operations characteristic to semaphores—passing the semaphore and acquiring the resource and release of the resource—are traditionally called P and V (names given by Dijkstra, based on Dutch words) and implementations usually follow these conventions.

Semaphores, while more general and flexible than locks, suffer from much the same composability and priority inversion issues as locks. Nevertheless,

semaphores are the synchronization method of choice for most modern operating systems and are at the foundation of other, higher level constructs, such as monitors.

5.4.3 Condition Variables and Monitors

Condition variables were proposed by Hoare and Hansen in 1974–1975 [9, 10] as a mechanism for operating systems. Essentially, it's a way to wait on and signal the fulfillment of a *condition.* Consequently, a conditional variable *c* is associated with an assertion *P* and it provides two operations: *wait c* and *signal c*. A thread may block and wait on a condition variable, but it will not block it; another thread may in turn signal it, thus waking up one of the waiting threads. For example, a semaphore can be implemented using a condition variable associated with the value of the semaphore: if the value is not positive, the thread issuing a *P* operation will wait; every time a *V* operation is performed, the condition variable is signaled (the condition is fulfilled, the associated value is guaranteed to be positive).

There is a long standing academic debate on who shall own a condition variable (i.e., continue progressing) once a *signal* operation is performed. In the original proposal, the issuing thread is blocked and any waiting thread will progress; this approach is called *blocking condition variables*. In case the signaling thread is allowed to continue, the condition variable is called non-blocking, but also *Mesa style* (for the language with the same name) or *signal and continue* condition variable.

Monitors provide a primarily language level mechanism for implementing objects that can be used safely by multiple threads A monitor is thus an object with both mutual exclusion property (only one thread at a time may execute its methods—implemented through the use of semaphores or mutexes) and support for conditional access, implemented through condition variables. A good example is an object modeling a bank account: only one thread at a time may perform operations on it; at the same time, threads that want to withdraw money have to wait until the condition variable expressing the constraint of having sufficient funds is signaled.

Condition variables and monitors have been extensively implemented and used in multiple languages and operating systems such as Ada, Java, .NET languages, Python and the Solaris operating system (for implementation of the sleep/wake-up mechanism).

5.4.4 Critical Sections

As we mentioned briefly in the discussion of the sources of synchronization, one of the more subtle ways of synchronization relates to the need to make sure a certain piece of code, once started, is not interrupted by other threads until it fully completes; in a more relaxed form, the requirement is to guarantee that no other thread

can enter that piece of code while it's being executed. Using the banking example again, executing an inter-account transfer is a prime candidate to be handled in this way.

Such pieces of code are called *critical sections* and are said to execute *atomically*. In order to be able to efficiently manage critical sections, there shall be a way to mark these accordingly; this is usually implemented in language runtimes, rather than through operating system support. Under the hood, the most common implementation is through a combination of a mutex or a semaphore and special handling in the operating system scheduler; however critical sections are a higher level, handy way to express atomicity, consistency and isolation requirements in parallel programs, even though same effect can be obtained through locks and manipulation of the OS scheduling.

Several operating systems provide pre-defined critical sections that can guarantee atomic execution of some simple operations; for example, Linux has a library of atomic integer operations including arithmetic and set-and-test type of operations.

Critical sections are also extensively used in the implementation of transactional memory mechanisms.

5.4.5 Transactional Memory

Transactional memory was proposed in an ISCA paper in 1993 [11] as a method to support efficient implementation of lock-free data structures. It was proposed as a generalization of the wide-spread HW level load-link/store-conditional instruction pair, which guaranteed that once a value was read from a memory location, it was only written back if the content of the memory location was not changed in-between.

Transactional memory aims at removing at least some of the problems inherent to locking based solutions for concurrent access to memory. The basic idea of transactional memory is to treat a set of memory access operations as a transaction that either succeeds fully or it's rolled back completely. To achieve this, all memory accesses need to be cached until the transaction completes and committed (made visible to other threads) if and only if none of the accessed memory locations was modified by any other thread, otherwise, the transaction is rolled back and re-executed later.

On application level, transactional memory is usually implemented using critical sections to mark the code that shall be treated as a transaction. Additionally, some implementations also require that memory areas that may be accessed from within transactions are explicitly marked, in order to provide a mechanism for monitoring those areas for concurrent access. Beside this, normally no other action is required from the programmer: in fact, the strength of transactional memory lies in the built in mechanism to transparently re-try transactions until these succeed.

Recently [12], transactional memory was proposed as a method for guaranteeing safe execution of critical code in safety-critical systems: the same transaction is ex-

ecuted on multiple processors and only if the same result is obtained, the execution declared correct and used further on.

On implementation level, transactional memory may be implemented either in software (in which case it's called Software Transactional Memory—STM) or hardware (Hardware Transactional Memory—HTM). STM is usually implemented through a combination of compiler support and run-time library; during run-time, all changes to memory are logged and either committed (executed within one critical section) or just removed, if other transactions modified the same shared area. While there are several STM libraries available, at the time of writing this chapter (2010) there was no commercially available hardware platform that would support HTM.

One of the reasons behind this is that after the initial hype, transactional memory is traversing a period characterized by reports on lack of scalability, lack of predictability (essential for some problem domains such as real-time applications) and complexity of handling some of the consistency and tracking problems, especially in case of STM. While some of these issues are due to the very nature of shared memory, other stem from the complexity of efficiently tracking memory updates, maintaining thread local copies and re-executing failed transactions in a transparent way.

5.4.6 Shared Memory

The concepts discussed above are the basic building blocks allowing implementation of the shared memory model. The shared memory paradigm is the most used and easiest to grasp method for inter-thread communication. Its main attractiveness is that it feels like a natural extension of single-threaded applications and does not require major re-thinking of program structure.

The concept itself is simple and straightforward: threads that need to communicate all hold a reference to a location in the memory that can be used to place information into which then can be accessed by anyone else holding a reference to the same memory location. Usage of shared memory is one of the main drivers behind the SMP paradigm: most SMP machines strive to provide the appearance of a single memory space, uniform computing environment, where any processor can access any memory location in the system.

There are however two fundamental issues with shared memory: in order to maintain the *consistency* of the memory content, access to the memory must be *policed* and all threads *need to synchronize;* for the same consistency reasons, local copies of the shared memory held in processor caches must be synchronized and updated, while the hardware must provide mechanisms for keeping the logical ordering of memory writes, the primary focus of memory consistency models, discussed in Chap. 2.

The first issue—policing access to shared memory—lies entirely in the application developers' hands. Traditionally, locks and/or semaphores were used, with a synchronization object associated with each logically related group of shared mem-

ory areas; any thread wanting to access the shared memory would need to acquire the ownership of the synchronization object before accessing the shared memory area. More recently, transactional memory was proposed as a method for policing access to shared memory, through automatic commit or re-execution of critical sections accessing it.

The two methods—locks and transactional memory based—differ in their approach to concurrency. Lock based solutions are inherently pessimistic and defensive: assume that conflict *will* occur and hence make sure the program will progress only if it's guaranteed to have exclusive access. Transactional memory takes a more optimistic approach: it assumes that the access to the shared memory areas will succeed, but provides a graceful exit if it doesn't, through rollback and re-execution of transactions. It also guarantees that *at least one* thread will make progress, eliminating the possibility of dead-locks. This comes however at the cost of potentially wasted computing resources: while in locking schemes the OS may schedule other threads while some of the threads are waiting to acquire a certain lock, with the transactional memory based approach, the processor is kept busy re-executing—potentially many times—the same code.

Both cases however will lead to delayed execution and slowed down progress of threads in case some of the shared memory is highly contested: threads will either be blocked waiting to acquire a lock or will be busy re-executing rolled-back transactions. This slow down and delay is not due to locking or transactional memory; it's rather the manifestation of a symptom often overlooked by practitioners of parallel programming: as the number of parallel tasks (threads) increases, any shared resource will eventually become a bottleneck and a point at which the tasks need to execute sequentially. Based on Amdahl's law discussed in Chap. 4, this sequential execution will define the scalability (or rather, the lack of it) for the entire program.

The second issue—cache coherence and hardware ordering—is usually solved entirely in hardware through the implementation of memory consistency and cache coherence and validation protocols; however, these will have impact on software performance: the number of processor cycles needed for keeping the caches coherent and the size of the logic required to implement it (consequently, the power consumption) will increase with the number of cores that are concurrently accessing the same memory location. Optimizing for cache behavior requires extremely good understanding of the underlying hardware and often leads to application level solutions that are not immediately obvious, such as 'un-natural' partitioning or sizing of data, bundling of memory updates etc. We'll take a look at these techniques in Chap. 6.

5.4.7 The Follow the Data Pattern

One of the promising ideas that have recently been put forward and aim to tackle both the cache coherence and concurrent access policing issues focuses on enforced *execution locality* [13, 14]. The idea is to prioritize the placement of data, rather than

that of the program: instead of replicating shared memory content across multiple CPUs' caches, the data shall be locked to and accessed from just a few (or, ideally, just one) processor cores; consequently, threads would need to be migrated instead, whenever access to the shared memory area is performed (e.g. whenever a critical section is entered). On application level, this method requires only that portions of the program that access shared memory are marked as critical sections and, at least in the general implementation, that shared memory areas are explicitly marked.

In its simpler version, all shared memory accesses are performed from just one core, called the resource guardian core, thus no other core is allowed to access any shared memory areas. In this approach, threads that need to access a shared memory area (e.g. by entering a critical section) are blocked, migrated to the resource guardian core and will be executed there. As the core will only execute one single thread at a time, using the critical section paradigm (i.e., non-interrupted execution of the complete critical section), this approach guarantees by design that all accesses to shared memory are serialized and thus no dead-locks will occur and no explicit locks are needed.

In fact, the resource guardian core (the only one allowed to execute critical sections) implements one big system wide lock: threads that will get to execute on that core are in fact acquiring this global lock. Such a design obviously limits scalability: even un-related accesses to disjoint shared memory areas will have to wait on each other, as all will execute on that single processor core.

The generalized version of the Follow the Data pattern groups critical sections into critical section groups and assigns a resource guardian to each of the groups. A critical section belongs to a specific group if there's at least one other critical section in that group with which it shares at least one shared memory area (in the sense that both access that specific shared memory area). Critical section groups are essentially clusters of inter-related critical sections and represent the concurrency dependencies between different critical sections, hence these critical sections shall execute on one single resource guardian that will own all the shared memory areas accessed by the critical sections in that group. The conceptual architecture is shown in Fig. 5.4.

This approach obviously improves the scalability of the system, as only critical sections with the potential of resource contention are serialized on one single resource guardian. However, this method brings in the potential of dead-locks: in a modular system, not all dependencies may be known a priori, hence nested critical sections may be allocated to different groups and execute on different resource guardians, thus dead-lock may occur. There's however an elegant solution for resolving such cases by having resource guardians share information on which critical sections triggered which other critical sections and on which resource guardian. Based on this information, a *dependency graph* of the critical sections can be built which will contain a loop if and only if there's a dead-lock in the system. To resolve it, at least one critical section will be rolled back (using e.g. a simplified form of transactional memory) and the execution can resume by skipping the critical section (and re-executing it later). In addition, this dead-lock resolution algorithm provides a method for merging the resource guardians frequently involved in dead-locks, in

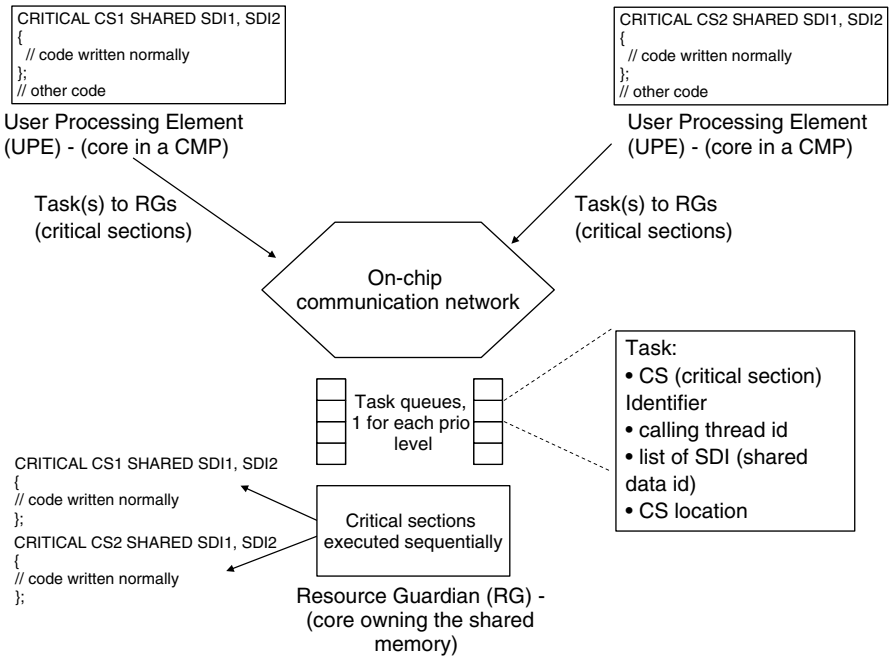


Fig. 5.4 Conceptual architecture of the implementation of the Follow the Data pattern

order to prevent future dead-locks; in fact this method implements an elastic and automatic lock-scoping strategy (as we mentioned earlier, resource guardians are in fact a different implementation of locks).

The Follow the Data pattern has some significant benefits. First, both in the simplified and in the generalized version provides a dead-lock free solution for implementing shared memory semantics (either by construction or through a built-in dead-lock detection, resolution and avoidance mechanism).

Secondly, it also has significant benefits from memory access point of view: as shared memory areas are only accessed from one single core, the content of these areas in the memory can be kept in the core's fast L1 or L2 cache, thus reducing the cost of memory access. For cases when the size of data is much larger than that of the code accessing it, such an approach improves memory performance significantly, by reducing the amount of memory-to-cache traffic to code only—experiments have shown that the benefits easily compensate for the cost of thread migration [14].

There are two subtle consequences for hardware as well from using the Follow the Data pattern. In the simplified case, the system will perform better if the resource guardian is implemented as a fast core with a large local cache: the critical sections will execute faster and hence will reduce the weight of the sequential portion in relation to the parallel part (and result in better performance according to Amdahl's law). The same is true for the generalized case as well, but in this case more than one fast core could be used efficiently. All in all, the Follow the Data

pattern performs better on single ISA, asymmetric processors, where a large number of simpler cores are coupled with few fast cores with large caches.

The second consequence has to do with the way shared memory is accessed. As any shared memory area is accessed from one single core, on hardware level there is actually *no memory sharing between cores:* any memory location is either private to a thread or it's shared among several threads, but accessed from just one core. Either way, cache coherence and memory consistency mechanisms become superfluous and consequently could be removed, reducing either the power consumption of the chip or using the freed-up logic for other functions [15].

In summary, the Follow the Data pattern is a way to provide shared memory semantics on application level without actually implementing shared memory on middleware, OS or hardware level. It has a number of clear benefits in terms of programmability, memory performance and support required from the hardware and it's a clear candidate to replace both lock and transactional memory based solutions.

5.4.8 Message Passing Based Communication

The message passing paradigm is considered by many the complete opposite of the shared memory model, as it usually relies on a 'share nothing' approach. In a message passing based system, threads will have no shared memory areas, all information exchange happening through messages that are passed between the threads. Message passing is usually considered the foundation for any thread isolation solution: shared memory means that if one thread fails, the other threads may fail as well, as the content of the shared memory may be corrupted; the same situation will not occur in a message passing based system, as threads can protect themselves through analysis and possible discarding of erroneous messages. Due to this characteristic, message passing is common in real-time systems, such as telecommunication software.

There are many implementations of message passing systems, both as part of operating system kernels, various middleware and language systems. These methods can be classified along several criteria: reliability of message delivery, delivery order guarantees, communication channel architectures (one-to-one, many-to-one or various multi-casting solutions), and whether messaging happens in a synchronous or asynchronous fashion. Most systems provide reliability and delivery order guarantees and support for at least one-to-one channels. Asynchronous communication is more wide-spread.

There are several well known and used message passing solutions. TIPC (Transparent Inter-Process Communication, [16]) is an open-source messaging stack aiming at distributed systems and integrated with the Linux kernel as well as the VxWorks and Solaris operating system. MCAPI (Multi-core Communication API, [17]) was developed specifically for multi-core chips and aims at hiding the hardware variations under one unified communication system. MPI (Message Passing Interface, [7]) is the long standing de facto standard for message-based communication in high performance computing.

On the language side, several high level languages such as Java support messaging interfaces. The one most intimately associated with the share nothing, message passing based style is Erlang, specifically developed for implementing massively parallel systems which however require strong isolation between threads. We'll discuss Erlang in more detail in Chaps. 8 and 9.

5.4.9 Partitioned Global Address Space (PGAS)

The *Partitioned Global Address Space* (*PGAS*) model was proposed by the high performance computing community as a way to tackle the inherent scalability issues of the shared memory model. With a grain of oversimplification, it's a combination of the traditional shared memory model and a variant of the *Follow the Data* pattern.

The PGAS model has a locally coherent, globally de-coupled view of the memory. From an application perspective, it maintains a global address space view of shared memory: all memory areas are accessible and addressable from any processor core (many of the systems and languages built on the PGAS model also allow threads to have private local storage, unavailable to other threads). However, each memory area has a *home location,* i.e., a set of processor cores from which actual manipulation of the content of that memory area is possible. If an application task (or thread) needs to access that specific memory area, its execution must be shifted to one of the cores belonging to the home location of the memory, in a very similar manner to how the *Follow the Data* pattern implements shared memory access. Some implementations of the PGAS model also deploy data shifting techniques but these are usually less efficient than techniques relying on shifting of computation to the location of the memory that needs to be addressed.

Recently, as the limitations of the globally shared and coherent memory models became hard to circumvent, PGAS has gained a wider acceptance, with several new languages having built in support for this model (such as X10, Chapel etc.). We will take a closer look at these languages in Chap. 8.

5.4.10 Future Constructs

Future as a synchronization construct was proposed in 1976 [18] and it's also called in some cases *promise, delay* or *eventual.* It's a construct that represents a placeholder for the result of a computation that may not have been yet performed—a task may declare it as an indication that at some point it may need this specific result, but the execution of the computation may be delayed until the task will actually need the result. Due to this property, futures may be used as input to scheduling tasks in an operating system or a language middleware; this feature is exploited by some language runtimes (such as that of language *E*) and it's also called *promise pipelining*.

There are several choices related to the implementation of futures. The usage of the future may be *implicit* if it is resolved automatically the first time it is used or *explicit* if the user has to call a get-like method of the future to receive its value. The behavior of the receiver task may also differ according to the specific implementation: the receiver task may block waiting for the future to be resolved or may be terminated in case the result is not readily available. Finally, the evaluation of the futures may be done eagerly—initiated as soon as the future is created—or lazily, only when it's actually needed.

Futures are available in most mainstream languages, either as part of the language itself or as a separate library. Java, C++0x, OCaml, C##, Ruby or Python all support different variants with different semantics, usually implemented as standard libraries or modules (in the case of C++0x it will be part of the standard).

5.4.11 Summary

Decomposing a program into parallel tasks inherently means that the resulting tasks will have to communicate in order to synchronize and share results. While over the past decades many synchronization and communication primitives have been proposed, these can ultimately be reduced to just a few basic ones: mutual exclusion primitives for synchronization, shared memory and message passing for communication. In fact, there is a layering of these primitives, where higher level methods—such as transactional memory, condition variables, critical sections etc.—can be built using simpler constructs such as locks, shared memory or message passing. The ultimate choice between these will depend on the characteristics of the application, the nature of the hardware as well as the support available in the underlying operating system.

5.5 Patterns of Parallel Programs

So far, we looked at the fundamental methods of decomposition and synchronization characteristic to any parallel program, as well as how these can be implemented as atomic primitives. However, in order to give a more detailed picture of parallel programming, we have to look at the higher level issue of architecting parallel programs (which then can make use of the techniques addressed earlier in this chapter).

Patterns have been used to capture best practices ever since the book of the Gang of Four [19] made the concept popular within the computer science community. For a deeper understanding of what patterns are, we recommend Ref. [20], which also lists the main conferences in this area.

When it comes to parallel programming, there were two main attempts at defining a pattern language (a collection of patterns and associated rules to combine them into an architectural style): the Pattern Language for Parallel Programs (PLPP),

described in Ref. [21] and Our Pattern Language (OPL), an effort led by the ParLab at Berkeley. Recently, the two proposals were merged into one language—called OPL—debated at ParaPLOP 2010 and available in Ref. [22]. In this chapter we'll provide an outline of this unified pattern language, as we believe it's one of the most comprehensive and structured approaches for describing the most important architectural choices a programmer will have to make.

OPL is structured into layers, which loosely correspond to the typical design steps; Fig. 5.5 gives an overview of these layers. The top-most layer comprises the structural and computational patterns, at the foundation of the high level software architecture of any parallel program. Structural patterns capture the overall architectural styles of programs (the 'box and arrows' view), while computational patterns describe the different classes of computations (the 'inside the box' view), loosely based on the dwarf concept introduced by Berkeley researchers [23]. These two categories of patterns—structural and computational—are strongly coupled, as choices within one category will influence the other, thus an iterative approach is needed to settle the architecture and computational model for a specific application.

The following three layers relate directly to the design of parallel software. The top-most comprises the *parallel algorithm strategies patterns*, methods for mapping the high level software architecture onto parallel algorithms. *Implementation strategies patterns* capture the mapping of parallel algorithms into program and

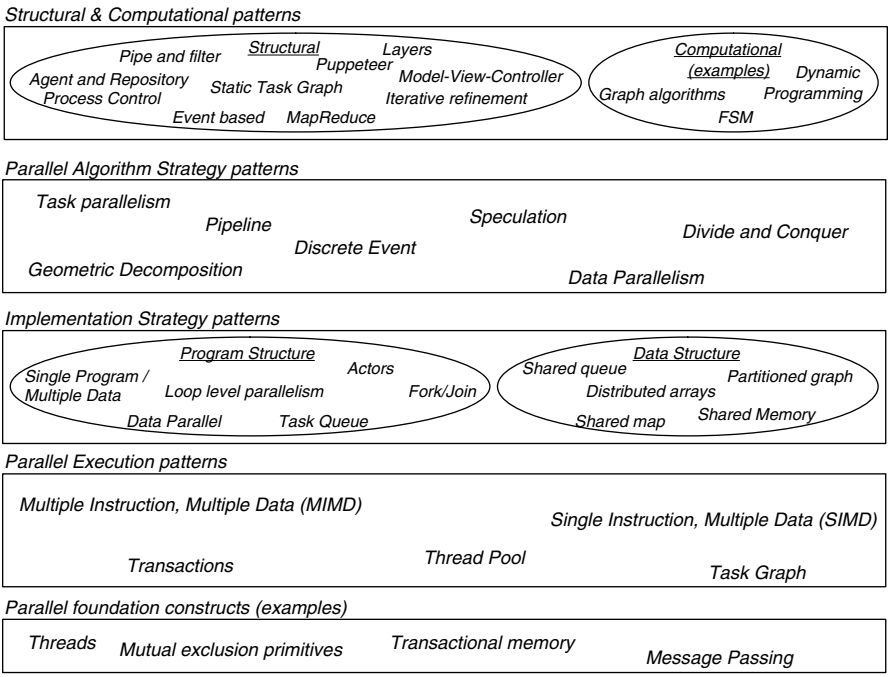


Fig. 5.5 The high level structure of the OPL pattern language

data structures, i.e., realization of selected algorithm strategies in source code. Finally, the *parallel execution patterns* capture the most common methods of supporting the implementation strategies in e.g. language run-time systems or middleware that supports concurrency.

All these layers will make use of some fundamental services—essentially the same we have been discussing so far: thread and process management, communication as well as synchronization primitives. The authors of OPL call these *parallel foundation constructs* and consider these beyond the scope of OPL.

In the following discussion of the patterns that comprise each layer, we will skip the computational patterns and focus instead on the lower three layers of parallel design patterns and briefly look at the structural patterns.

5.5.1 Structural Patterns

OPL includes the following structural patterns:

- *Pipe and filter*: the program can be organized as data streams flowing through a chain of operations (filters) connected through communication channels called pipes
- *Agent and repository*: the program consists of a collection of data elements (repository), accessed asynchronously by a number of agents that need to be scheduled so that the consistency of the data is maintained
- *Process control*: analogous to industrial control processes, where the process is monitored and based on the status, actions are decided and performed by controllers
- *Event based systems*: programs are modeled as a collection of tasks that respond to events by performing some computations and issuing new events as a response
- *Model-view-controller*: essentially the classical GUI design pattern
- *Iterative refinement*: the result of the program is obtained through repeated (statically or dynamically dimensioned) application of a set of operations until a predefined goal is realized
- *MapReduce*: the result is obtained through the mapping of the same processing function onto multiple subsets of data, followed by a reduce operation that combines the results of the sub-computations into one final result
- *Layers*: the classical layered system pattern
- *Puppeteer*: programs are realized as interacting software agents, with interactions managed by a central entity called the puppeteer
- *Static task graph*: the program is structured into independent tasks whose interaction is described as a graph

Of these patterns, *Layers, Iterative Refinement, Process* Control and *Model-view-controller* are general patterns that just happen to be applicable to parallel software. *Pipes and filters* is primarily applicable to data stream processing and we have

already discussed it in our introduction to decomposition; *Static task graph* is the high level expression of static functional decomposition. The *Agent and Repository* and the *Puppeteer* patterns are essentially the manifestation of the same basic pattern, the only difference is the scope of the co-ordination (access to the shared data or inter-agent communication). *Event based systems* are found in e.g. telecommunications; the Erlang language is a prime example for the implementation of support for this pattern.

5.5.2 Parallel Algorithm Strategy Patterns

Parallel algorithm strategies are deployed to map the structural patterns in the context of parallel programs. Without the aim of completeness, we will discuss the most important ones in this chapter.

Task parallelism addresses the scheduling of tasks resulting from high level decomposition on available parallel processing units (e.g. cores). It builds on the dynamic decomposition method and the main challenges relate to load balancing and inter-task dependency management.

Pipeline is the parallel mapping of the *pipes and filters* and *process control* structural patterns; as we discussed in the data parallel decomposition chapter, the opportunity for parallelism results from having multiple data items flowing through the series of pipeline stage computations—as soon as the previous data item was processed, any stage can start working on the next item; if the pipeline stages are allocated on different processing elements (cores), parallel execution will be obtained.

The *discrete event* pattern represents the event handling functionality at the basis of multiple structural patterns, such as *event based systems*, *model-view-controller* and in some cases *process control*. It provides the infrastructure for delivering events generated by tasks to tasks that can process it; it does not require (nor exclude) that tasks know the source or destination of received (or sent) events.

The *speculation* pattern captures the speculative execution type of decomposition. To be effective, it requires support for two essential features: method for reliably and deterministically determine whether speculation in general or on some branches in particular were successful or not and support for rolling back and re-executing unsuccessful speculations in case execution conflicts have been detected. The pattern can be applied in combination with several structural patterns, such as *static task graph* or *agent and repository*.

Data parallelism is the direct implementation of the data-based decomposition method: it implements support for applying concurrently the same set of operations on multiple elements of data.

Divide and conquer is the parallel version of the classic decomposition method. As opposed to task parallelism, it tends to be synchronous or at least more strictly orchestrated in a hierarchical fashion: a specific task generates multiple sub-tasks and then combines the results into a higher order result.

The *geometric decomposition* pattern deals with the issue of data parallelism in the presence of dependencies between data chunks. Thus, computations will be made in three phases: exchange chunk boundary information with neighboring chunks; perform computation in the interior of the chunk; update boundary information. The process is repeated until a satisfactory result is obtained; obviously, operation on different chunks can be done in parallel on different processors. This pattern is typically used in scientific computing on large arrays or matrices.

The relatively low number of parallel algorithm strategy patterns underlies the important observation that when everything is considered, there are basically very few ways to map a program onto a parallel architecture: different types of task decomposition, message passing, shared memory, scheduling techniques. The secret sauce comes with the art of combining these techniques and finding the right level of abstraction for each problem domain and target computing environment.

5.5.3 Implementation Strategy Patterns

Implementation strategy patterns capture the most common methods for mapping higher level constructs into concrete code and the associated data structures that can support concurrent access efficiently. OPL groups the implementation strategy patterns into two categories: those related to program structure and those related to supporting data structures; there are six program structure patterns (*single program/multiple data, data parallel, fork/join, actors, loop level parallelism* and *task queue*) and five data structure patterns (*shared queue, shared map, partitioned graph, distributed array and shared data*). These will not cover all the possible structures one might find in a real program, but should rather be regarded as the most common basic building blocks from which other structures may be created.

The *single program/multiple data* pattern represents the approach where the threads that make up the program will decide based on their identity the actual functionality to be performed as well as the actual data set on which to act. Still, the same program is available on all processor cores—hence this pattern is primarily a method for managing one set of source code when different functions need to be performed on different types of data.

The *data parallel* implementation strategy pattern is essentially the realization of data parallelism on source code level. Typically the data is organized in arrays or matrices and partitioning is done using index ranges relative to that structure (hence the pattern is sometimes called *index space* as well). The computations are executed in parallel on all of these disjoint index ranges and results are then potentially merged at the end. This pattern is the implementation level mapping for the *data parallel* and *MapReduce* patterns from the layers above.

Fork/join models the organization of computations along spawning new threads coupled with join operations (wait synchronization) for co-ordination, usually within a shared address space. It's the typical implementation level realization of the

Divide and Conquer and sometimes of the *Task Parallelism* pattern (for example, the OpenMP library), using generally available operating system primitives.

The *Actors* pattern is the realization of the programming model with the same name. Computations are organized into objects that communicate primarily asynchronously (through messages/events, though remote procedure call based implementations are also available) with each other and hence can be deployed on different processor cores. It's very popular in soft real-time, event based systems as well as in data-flow programming and it's usually at the basis of the implementation of event based patterns.

Loop level parallelism is probably the most researched parallelization technique as it is at the basis of many algorithms (usually from the scientific domain) and provides plenty opportunities for parallelization. The idea is to structure loops in such a way that there are no dependencies between successive iterations and thus different iterations can be executed in parallel. To achieve such a structure, usually several structuring techniques need to be deployed: loop splitting, loop un-rolling or loop alignment (increasing the number of iterations in order to eliminate carried dependencies between different iterations). Ref. [24] has a good discussion of these techniques.

Realization of task parallelism requires an implementation level mechanism for *task queuing*. The idea is to organize available tasks into one or several task queues from which processor cores can pull tasks—concurrently—for execution. In its simplest form, the pattern proposes a single task queue; however, for massively parallel systems this is likely to be a bottleneck. Hence most realization of the task parallelism pattern—such as Cilk or Intel Thread Building Blocks—use multiple, one per-core, queues coupled with task stealing: when a core's queue runs out of tasks, it may take over some of the tasks from another queue. We will discuss these libraries in depth in Chap. 9.

The practical realization of these patterns requires specific data structures—the scope of the data structure implementation strategy patterns. We would like to emphasize again that this list is not comprehensive: these structures are likely to be useful in many cases, but sometimes different or further specialized structures will be needed.

The *shared queue* and *shared map* patterns provide the mechanisms for implementing the basic structures of queues and maps so that these can be accessed concurrently from multiple processor cores. The synchronization mechanisms are built into the implementation of the typical operations (such as *put* and *get*) so that any user of the data structure can access it as if it would be the only user. There are many implementations of such structures, e.g. in Ref. [3].

Partitioned graph provides a recipe for enabling parallel computations on different regions of a graph structure so that the need for mutual exclusion is minimized. Such data structures are useful for some problem domains where patterns such as *geometric* decomposition are used.

In many cases there is a need to *distribute arrays* across multiple cores or processors: either because of the size of the structure or simply for partitioning reasons (such as usage of the *data parallel* pattern). The most practical way to achieve such

a distribution is to provide a data structure that can hide and efficiently manage mapping of global indices to local indices and provide support for boundary overlaps between cores(such as in the case of the *geometric decomposition* pattern). In many ways, the distributed array concept mirrors the virtual memory mechanisms found in most modern operating systems.

Last but not least, *shared memory* is a well known, used and often abused mechanism. OPL proposes a mechanism through which access to shared data structures is wrapped into a strict API backed up by synchronization mechanisms that can guarantee safe concurrent access. There are plenty of ways to implement this: transactional memory and the *follow the data* pattern we discussed earlier in this chapter are good examples.

5.5.4 Parallel Execution Patterns

The lowest layer in the pattern hierarchy of OPL consists of patterns that are usually input to the implementation of language run-time systems, middleware solutions and occasionally operating systems. These five patterns represent recipes for using the basic concepts offered by the HW and the OS (processes, threads, synchronization and communication primitives) in order to support implementation of the patterns from the higher level layers.

Multiple Instruction, Multiple Data (*MIMD*) captures the wide-spread paradigm of executing different programs working on different data sets on different processor cores; these streams of computations will occasionally synchronize through e.g. message passing or shared memory. This pattern is the natural choice for several types of applications.

Single Instruction, Multiple Data (*SIMD*) captures a special class of processing environments, commonly found in Graphics Processing Units (GPUs): all the cores (in practice, a subset of cores) execute in lock step the same instruction flow, but act on different data sets. Such execution environments are best suited for problem domains were data parallelism is the natural choice.

Thread pool is a method for guaranteeing fast allocation of threads to fulfill the application's needs. Threads are pre-allocated and managed in an idle thread pool; whenever the application needs a new thread of execution, one of the pre-allocated threads is woken up and given the task indicated by the application; when the task is completed, the thread is returned to the pool. This pattern is often used with dynamic functional decomposition methods and it's at the basis of task based models. The same effect can be obtained—and hence we consider it an implementation of this pattern—through specific language run-time systems that can quickly create and destroy user-space threads, without actually maintaining a thread pool; the Erlang run-time system is a prime example, with thread creation times far lower than through any OS level primitive.

The *Task Graph* pattern captures the mechanism through which task dependencies can be expressed as a directed acyclic graph and presented to the run-time

system for scheduling and execution on a machine with multiple processor cores. Such mechanisms are typical to data-flow type of applications found in e.g. the signal processing parts of mobile communication systems.

Transactions are the prime mechanism for implementing speculative execution. The run-time system has to provide and support the mechanism for implementing units of execution (chunks of programs and memory these are accessing) that either complete without conflicts with other units of execution or need to be rolled back and re-executed at a later time. It is sometimes implemented using transactional memory as a vehicle for detecting and rolling back conflicting memory operations.

5.6 Summary

In this chapter we set out to describe the fundamentals of designing software for parallel systems, i.e., systems with multiple execution units (either on one chip or across a communication network). There are two fundamental procedures for designing parallel software: decomposition into multiple, parallel tasks and design of synchronization/communication mechanisms through which the tasks resulting from decomposition can interact in order to achieve the ultimate goal of the program. We also introduced OPL, probably the most ambitious attempt at capturing the best practices through which parallel software can be architected, designed, implemented and deployed over an execution environment, for multiple application domains.

Is OPL the Answer for the big question of how to program multi-processor and multi-core system? Obviously not. It's merely a comprehensive, structured snapshot of the current best practices that proved to be useful; it's the result of a close co-operation between academic and industrial partners and thus captures the experiences of both practitioners and researchers.

The goal of this chapter is to introduce the basic concepts and methods of designing parallel software and hence it's the foundation on which other chapters—primarily Chaps. 8 and 9—can build. In those chapters we will return to some of the techniques introduced here and detail those in the context of many-core chips.

References

1. The OpenMP Architecture Review Board (2008) The OpenMP Application Program Interface. http://www.openmp.org/mp-documents/spec30.pdf. Accessed 10 January 2011
2. Frigo M, Leiserson C E, Randall K H (1998) The implementation of the Cilk-5 Multithreaded Language. Proceedings of the ACM SIGPLAN 1998 conference on Programming Language Design and Implementation, 212-223
3. Reinders J (2007) Intel Thread Building Blocks: Outfitting C++ for Multi-core Processor Parallelism. O'Reilly Media
4. Mars J, Williams D, Upton D, Ghosh S, Hazelwood K (2008) A Reactive Unobtrusive Prefetcher for Multicore and Manycore Architecture. Proceedings of the Workshop on Software and Hardware Challenges of Manycore Platforms 2008, 41-50

5. Nellans D, Sudan K, Balasubramonian R, Brunvand E (2010) Improving Server Performance on Multi-Cores via Selective Off-loading of OS Functionalility. Proceedings of the 10th Workshop on Interaction between Operating Systems and Computer Architecture
6. Apple Corporation (2009) Grand Central Dispatch : A Beter Way to Do Multicore. http://images.apple.com/macosx/technology/docs/GrandCentral_TB_brief_20090903.pdf. Accessed 11 January 2011
7. Gropp W, Lusk E, Skjellum A (1994) Using MPI: Portable Parallel Programming with the Message Passing Interface. MIT Press Scientific And Engineering Computation Series, Cambridge MA, USA
8. Dijkstra E (1974) Over seinpalen (in Dutch). EWD 1974. http://www.cs.utexas.edu/users/EWD/ewd00xx/EWD74.PDF. Accessed 11 January 2011
9. Hoare, C A R (1974) Monitors: an Operating System Structuring Concept. Communications of the ACM 17(10):549-557
10. Hansen B P (1975) The Programming Language Concurrent Pascal. IEEE Transactions on Software Engineering 2:199-206
11. Herlihy M, Moss J E B (1993) Transactional Memory: Architectural Support for Lock-free Data Structures. Proceedings of the 20th International Symposium on Computer Architecture: 289-300
12. Yalcin G, Unsal O, Hur I, Cristal A, Valero M (2010) FaultTM: Fault Tolerance using Hardware Transactional Memory. Proceedings of the 3rd Workshop on Parallel Execution of Sequential Programs on Multi-Core Architectures
13. Vajda A (2010) Handling of Shared Memory in Many-Core Processors without Locks and Transactional Memory. Proceedings of the 2010 Workshop on Programmability Issues for Multi-Core Computers
14. Suleman M A, Mutlu O, Qureshi M K, Patt Y N (2009) Accelerating Critical Section Execution with Assymetric Multi-Core Architectures. Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems
15. Vajda A (2010) The Case for Coherence-less Distributed Cache Architecture. Proceedings of the 4th Workshop on Chip Multi-processor Memory Systems and Interconnects
16. TIPC Working Group (2010) TIPC: Transparent Inter Process Communication Protocol. http://tipc.sourceforge.net/doc/draft-spec-tipc-07.txt. Accessed 11 January 2011
17. Multicore Association (2010) Multicore Communication API, available from http://www.multicore-association.org/workgroup/mcapi.php. Accessed 11 January 2011
18. Friedman D, Wise D (1976) The impact of Applicative Programming on Multiprocessing. Proceedings of the International Conference on Parallel Processing: 263-272
19. Gamma E, Helm R, Johnson R, Vlissides J M (1994) Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley Professional
20. http://www.hillside.net
21. Mattson T G, Sanders B A, Massingill B L (2004) Patterns for Parallel Programming. Addison Wesley
22. Mattson T (2010) Our Pattern Language (OPL). http://parlab.eecs.berkeley.edu/wiki/_media / patterns/opl_pattern_language-feb-13.pdf. Accessed 11 January 2011
23. Asanovic K et al (2006) The Landscape of Parallel Computing Research: A View from Berkeley. University of California, Berkeley Technical report
24. Tasharofi S, Johnson R (2010) Patterns of Optimized Loops, Proceedings of the Workshop on Parallel Programming Patterns

Chapter 6
Debugging and Performance Analysis of Many-core Programs

Abstract Testing, debugging, maintenance and performance tuning of software applications usually take up more than 50% of the total effort that goes into designing software systems. The introduction of parallelism, the challenges posed by the complexity of memory systems and cache hierarchies exacerbates the already complex issues surrounding this area: concurrency bugs are notoriously hard to reproduce or detect through intrusive debugging techniques; cache misses and resource contention usually require deep understanding of the hardware and operating system. In this chapter we survey the most important issues in this area, the specific challenges in the context of software systems designed for many-core hardware as well as the most promising approaches for tackling correctness and performance problems.

6.1 Introduction

The major challenge verifying, debugging and profiling applications on many-core processors comes from the non-intuitive nature of the program flow: in fact, there are at least as many independent, yet interacting program flows as there are cores (or even more, when multi-threading within a core is used). While these program flows are executing independently, synchronization will introduce dependencies on the software level; access to shared resources—such as memory interfaces, on-chip cache or I/O devices—even if there's no apparent sharing on the application level, can introduce subtle dependencies that are hard to detect. Consider the simple example of a cache shared between two cores: even if the programs running on the two cores may access different, logically unrelated areas in the main memory, if those two areas happen to be mapped to the same cache line, the two cores will compete for the same resource and the result will be a constant trashing of the shared cache. Such errors and performance bottlenecks are extremely hard to detect even in such a simple case—let alone in a case where hundreds of cores are involved.

Traditionally, the area of verifying and tuning software systems is sub-divided into three main areas, briefly discussed here and shown in Fig. 6.1.

A. Vajda, *Programming Many-Core Chips,*
DOI 10.1007/978-1-4419-9739-5_6, © Springer Science+Business Media, LLC 2011

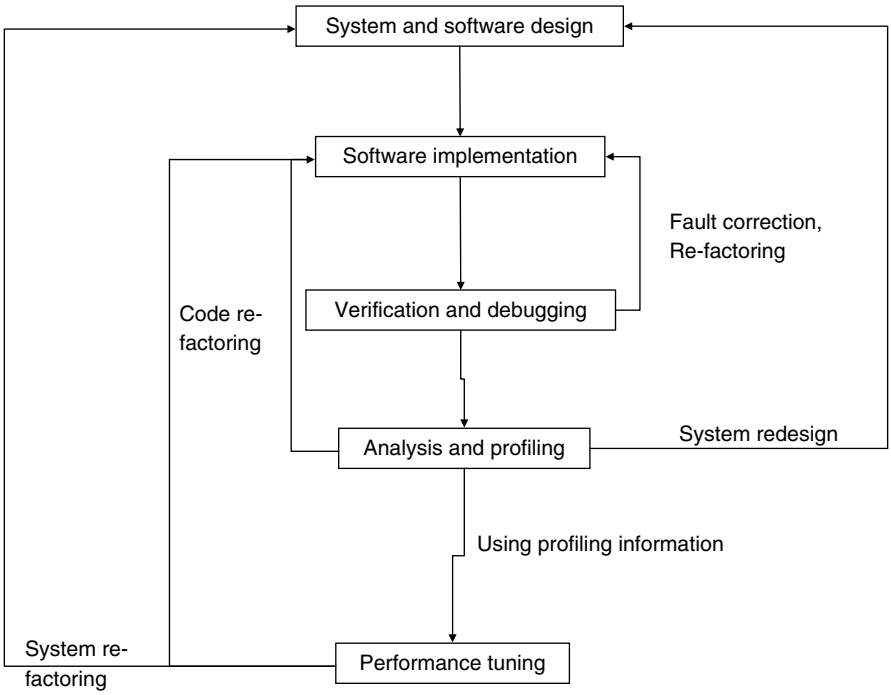


Fig. 6.1 Main areas of software verification and tuning

- *Debugging:* Making sure that the software is logically correct is obviously the first step taken once the software is designed and it can be executed—either in a simulated environment or directly on the target hardware. There are essentially two main areas that fall under debugging: *static analysis* of the source code and *dynamic analysis* of target code under execution. While static analysis methods have shown some promising results, reasoning about complex interactions and race conditions usually requires observation of executing code; identifying resource usage integrity issues (on software and hardware level) and detecting race conditions (such as deadlocks) are the primary concerns, specific to many-core software.
- *Analysis and Profiling:* once the software system is considered logically correct, the next step is to understand the runtime characteristics. This aspect of software tuning relies heavily on sampling and collecting information about execution times of different parts of the system, memory usage, memory access patterns, cache usage patterns, idling times due to synchronization or I/O operations etc. In the context of many-core systems, it's critical to record per-thread and per-core behavior as well as the interaction between different parts with respect to access to shared resources and impact on the usage of shared hardware components.
- *Tuning:* the goal of software tuning is to adjust the software so that—while preserving correctness—some specific metric of quality is improved. In the context

of software for many-core systems such metric may be idle time due to synchronization and/or resource access, average resource utilization, especially how well the software can exploit the availability of processor cores etc. An important aspect is the *scalability* of the software: how well the availability of more resource can be exploited by the system?

In the following chapters we will look at each of these areas in more depth. Our goal is not to pinpoint specific tools, but rather to highlight those aspects that need to be covered in the process of verifying, debugging and performance tuning applications in general and on many-core processors in particular. Occasionally however we will highlight certain tools that we found particularly useful or interesting in the context of software for many-core processors.

6.2 Debugging

Debugging is arguably as old as programming itself; still, finding faults in software is usually considered the costliest component of the whole design chain. Consequently, there's a bewildering array of hardware and software tools available—both open source and commercial—targeting the area of software correctness.

As mentioned in the introduction, there are two broad categories of debugging methods. *Static analysis* relies on—automatic or programmer-guided—analysis of the source code in order to detect omissions by the programmer. We consider the errors found by such tools *mechanical errors,* as usually these highlight coding errors rather than system or algorithm design flaws. Typical faults identified by such methods include

- *Un-allocated or un-released memory* as well as *buffer overflows*: the programmer failed to properly allocate memory for a pointer variable before accessing the content
- *Un-initialized variables*
- *Infinite loops* and *un-reachable code segments*
- *Data structures* that are inefficient with respect to memory usage and cache behavior, in the light of how those data structures are used. For example, instead of an array of structures, a structure of arrays with the different fields of the original structure may be more appropriate, if the data is mostly accessed in loops that only touch a few of the structure members. While detecting such issues is not really in the realm of debugging—more relate to profiling and tuning—static analysis tools routinely identify such bottlenecks

In the context of software for many-core processors, static analysis is most efficient in revealing unprotected/unsynchronized (through a lock, atomic section, transactional memory or similar) access to shared memory as well as potential race conditions and, occasionally, potential deadlocks. While such violations are relatively easy to detect through static analysis, dynamic execution may simply fail to create

the race condition within a reasonable time; hence we believe static analysis of the source code for potential synchronization issues shall always be the first step in the process of securing correctness of the software.

Dynamic debugging relies on observing the software system under execution. In order to be useful, dynamic debugging has to fulfill at least one of the following conditions:

- It shall allow *recording of execution information* through the software for post-run analysis and potentially replay. The recording shall include sufficient information in order to deterministically identify the path through the code, timing and interaction between different threads and the data context (which memory areas were accessed and the content of those areas)
- It shall allow *step-wise* or *slowed-down execution* of the software in order for the programmer to inspect the dynamic state of the executing software and potentially modify it in order to either correct an observed fault or induce a potential fault

Arguably the trickiest issues in debugging multi-core and multi-threaded applications are related to *timing* and *intrusiveness*. Determining accurate timing of events across multiple processor cores—let alone across multiple timing domains—is extremely difficult without intrusion; on the other hand, intrusive techniques (including breakpoints and recording of debugging information) tend to alter the behavior of the system to an extent where 'real' behavior cannot be observed anymore.

There are two promising techniques emerging that may address these issues, both relying on the abundance of computing power brought about by many-core chips. The first one relies on allocating some of the cores on the chip purely to monitoring, unobtrusively, the other cores' behavior as well as memory accesses, caches etc. The usage of such *helper cores* was explored in other contexts as well, such as support for memory pre-fetching, off-loading of certain functions (such as OS services), see e.g. Refs. [1, 2]. Such a technique has a negligible impact in case of many-core chips, as long as just a few cores are sufficient for the task.

The second technique relies on using full-scale hardware simulators, where the complete hardware, including cores, caches, interconnects and memory interfaces are modeled and simulated in software. In such a simulated environment, as long as it's very closely replicating the behavior of the real hardware, it becomes possible to completely halt the complete system—including the clock—inspect any aspect of it, simulate hardware faults, induce specific behaviors (especially delays and synchronization between cores) and much more in a completely unobtrusive manner. Obviously, the downside of such an approach is the drastically reduced speed of execution, sometimes by as much as two orders of magnitude; nevertheless, we found such techniques really useful in tackling some of the trickiest synchronization or hardware sharing problems. An added benefit of large scale simulators is that these allow easy tweaking of hardware in order to evaluate how changes in hardware may impact the behavior of the software—a tool that, beyond debugging, is of great use for a software company wanting to steer the development of next generation hardware.

Of the available simulation platforms, we would like to highlight two. Virtutech's Simics [8] (now acquired by WindRiver) is a commercial product with wide-ranging support for various processors that allows execution and inspection of unchanged binaries, as well as definition of new hardware simulators to fit the needs of the user. Beside simulating single-processor machines, complete cards and networks can be defined and simulated. For more commercial details on Simics please see Ref. [3].

A similar, but more research focused tool called Graphite was developed at MIT's CSAIL laboratory [4]. It is a parallel, distributed, software only infrastructure running on commodity Linux machines and it supports both functional and performance modeling for processor cores, on-chip networks, memory systems and cache hierarchies with full coherence. It's not cycle accurate, but it has built in mechanisms to generate reliable estimates of expected performance; these shortcomings are offset by a relatively high speed of simulation. The simulator is available for free download from the research group's web page.

Recording and reliable replay of software behavior is another area that has been extensively researched. The goal of the research is to design solutions that are capable of recording sufficient behavioral information about running applications so that a particular execution can later on be repeated or even automatically re-executed. Unfortunately, we are still quite some effort away from a solution that is capable of non-intrusive recording: today's solutions have such a significant overhead, that the intrusive impact obscures many of the fine-grained synchronization and race issues and may even be impractical for applications with tight timing requirements.

6.3 Analysis and Profiling

Software analysis and profiling are obviously the required activities in order to support tuning of software performance. In many respects, static analysis for debugging purposes and static analysis for performance purposes are overlapping activities, albeit with different goals; we already mentioned some of the aspects of static analysis with respect to performance in the previous chapter.

One of the key characteristics of analysis and profiling is the dependence on two parameters: the *hardware architecture* on which the software will execute and the *language* in which it was developed. Very few performance issues can be resolved in a hardware independent manner: the pipeline architecture of the processor, the cache hierarchy, bus speed, memory interfaces all have an impact on the software and hence need to be considered when analyzing an application. Similarly, the language in which the application was implemented will impact the nature of the analysis and the feedback that a tool can provide: a Java application will be dependent on the JVM; a C program with extensive use of pointers leaves too little room for automatic actions; the semantic link from a very high level functional language down to machine code may be very hard to maintain or follow—just to name a few of the issues that need to be considered.

Analysis and profiling usually requires some form of *instrumentation* or at least *configuration* of the monitoring environment. More often than not, some form of instrumentation of the source code or the binary—e.g. by using a special compiler or a post-compilation tool—is required. Instrumentation of the binary is usually automatic, with no programmer intervention; however techniques that require instrumentation of the source code usually demand active involvement of the programmer in identifying those critical paths in the software that can most benefit from the collection of profiling information.

Alternative techniques to instrumentation include *simulation in a virtual environment,* similarly to the method described in the previous chapter. Despite the speed penalty, such methods provide the programmer with a wealth of opportunities to monitor many aspects of the software, perform on the fly adjustments and quickly evaluate how minor changes can impact performance—hence we regard it as one of the most efficient and productive techniques. A similar technique is *statistical sampling:* based on pre-calculated sampling rates deemed efficient for certain performance factors, the analysis technique can periodically sample some characteristics of the running software and infer conclusions on the overall performance. Such techniques proved to be surprisingly accurate, while having comfortably low overhead (in the order of 5–20%), especially with respect to cache and memory performance. In our view, combinations of statistical sampling techniques with virtual execution environment and/or helper core techniques are the least intrusive while sufficiently productive methods for collecting relevant profiling information on running software.

What are the most important aspects that effective analysis and profiling methods for software deployed on many-core processors shall support?

The obvious challenge is dealing with the ever increasing number of threads and cores. Analysis and profiling techniques have to keep a constant overhead while dealing with exponentially increasing number of cores—in other words, the profiling overhead has to be constant in time. Recording thread interactions, contention points between threads and cores—both on application and hardware level—synchronization points and involved threads and resources are critical in profiling applications on many-core platforms with respect to synchronization bottlenecks that eventually will translate to scalability issues. Monitoring the useful and idle load each thread generates is also critical in assessing the usefulness of a particular threading model. Similarly, profiling access patterns to shared resources can yield useful information on the threading model's performance as well as how this may be improved.

Optimizing the usage efficiency of the cache hierarchy can also yield significant performance gains: in many cases simple improvements of access patterns and data re-structuring led to performance gains of up to 4×. As we briefly touched upon already, the following factors are important in this context:

- *Usage of cache lines*: profiling shall identify overlaps between cache lines mapping that can hamper the performance of concurrently executing threads
- *Cache usage and usage density*: the higher the usage rate of the cache (the percentage of cache content that is relevant from the code currently executing and the proximity of code fragments acting on the same data in the cache), the lower

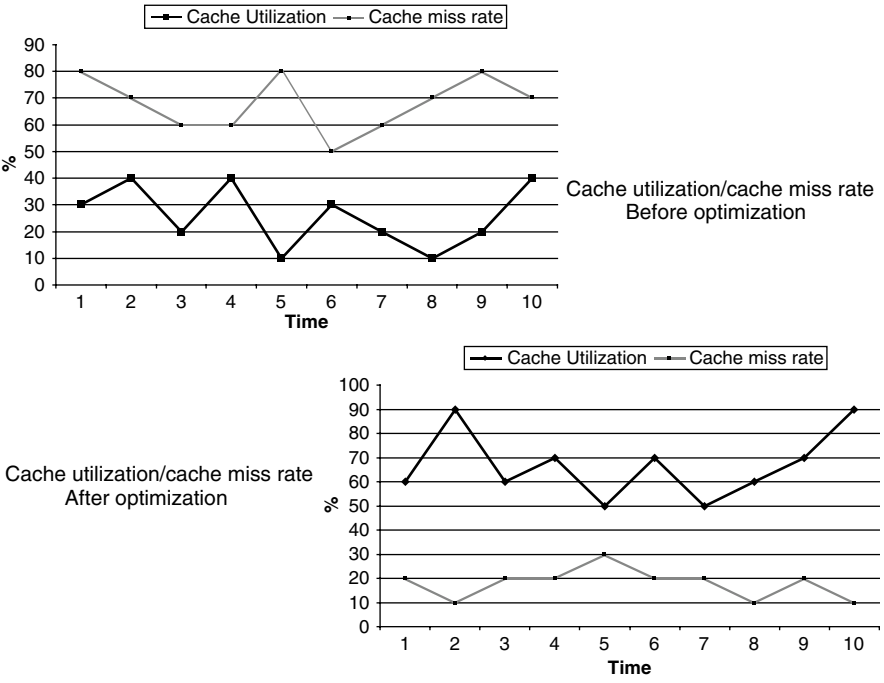


Fig. 6.2 Example of cache behavior of an application before and after optimization

the over head due to memory accesses will be; profiling tools shall be able to monitor and present the usefulness and timing of access to data in the cache

- *Cache miss rate*: how often and why the data needed by the executing code is not available in the cache and what techniques can be used to reduce it

Figure. 6.2 shows an example of cache usage level and cache miss rate level for an application before and after bottlenecks were identified—and corrected—using a cache usage profiling tool.

In this context, we believe one of the most productive technology is provided by Acumem (now acquired by RogueWave), through its cache and memory usage analysis and profiling tools. Acumem's tools deploy low overhead statistical sampling techniques coupled with tuning suggestions for C programs. For more details, please see Ref. [5].

6.4 Performance Tuning

Tuning the performance of an application running on a many-core processor is arguably the trickiest and most human-centered activity in the area of verification and performance modeling. Tuning usually requires an informed human decision

relying on deep semantic understanding of the software that is being tuned and careful analysis of data provided by analysis and profiling activities.

The actual actions performed as part of performance tuning can vary widely depending on the metric that is being improved. In our view the most important ones in the context of many-core deployments are the following:

- *Processor load, load distribution* and *idle processor time*: the deciding factor may be load balancing and distribution, scalability or power efficiency (achieving the same performance at lower energy consumption). Optimizing this metric may imply better balancing of tasks between threads and cores, loop optimizations and unrolling, reduction of contention for shared resources but it could result in a re-design of the application with respect to the deployed parallel programming model
- *Memory and cache performance:* this is arguably the trickiest and least understood aspect of performance tuning. Optimizing cache utilization usually requires deep understanding of cache coherency mechanisms, the concept of sharing on the hardware level—all besides the usual application level memory performance issues

An interesting technique emerging recently in the research community is *auto-tuning* [6]. The technique is based on two principles:

- *Acknowledge that no technique and optimization works best for all platforms*: instead of creating a single variant of a program that can execute efficiently on any platform, rely on searching for the best solution through generating and evaluating multiple variants of the software
- *Avoid optimizations in the code provided by the programmer*: the programmer shall design the correct algorithm, but let the technique of auto-tuning generate the best variant for the platform at hand.

So how does auto-tuning work? It starts with the premise that the software is written in some high level language where the focus is on *what* shall be achieved, rather than *how to achieve it efficiently.* Starting from this source code and equipped with knowledge of both the application domain and especially the target hardware, the auto-tuning infrastructure will generate a multitude of variants of the software, with variations of some key parameters and deploying pre-configured optimization techniques. All the variants are compiled to target code, executed and profiled automatically on the target hardware (the auto-tuning infrastructure can also generate the instrumentation required for profiling); finally the variant with best performance is chosen as the result of auto-tuning. The process is illustrated in Fig. 6.3.

This technique resembles closely the technique we will discuss in Chap. 10: auto-parallelization based on software written in high level domain specific languages. The key insights in both cases are

- Deep knowledge of the application domain: what are the characteristics of the application that we are tuning?
- Un-obstructed source code: no hand-written optimizations that may hamper automatic tuning
- Built in knowledge of the target hardware platform

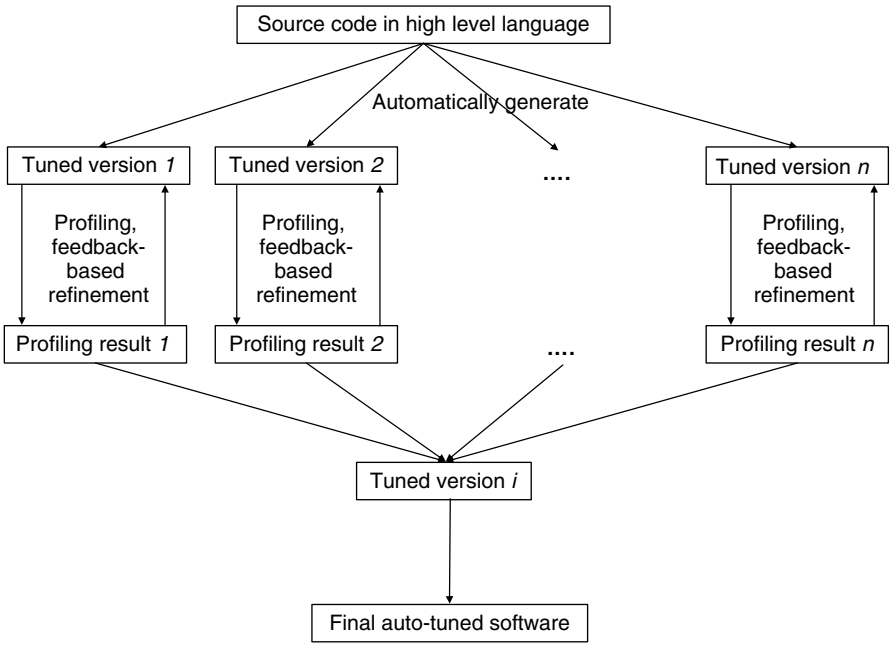


Fig. 6.3 The process of auto-tuning an application

This technique is still in early research state and large scale deployment is some time away; however, we believe a combination of high level and domain specific abstraction for software design, coupled with automatic search of possible solution domain, offers the best hope for less programmer-reliant and more tool based performance tuning. However, for some time to come, the state of the art of commercially available performance tuning tools will still be based on hints generated based on profiling information collected from previous executions of compiled software.

6.5 Summary

In this chapter we surveyed the three typical correctness assurance and performance improvement techniques—debugging, analysis and profiling, tuning—in the context of software deployed on many-core hardware systems. We highlighted the most important metrics and bottlenecks that need to be addressed as well as the characteristics that tools supporting these activities will have to fulfill. While our goal is certainly not to promote specific vendors' solutions, we briefly described those commercially and freely available tools—developed by companies and research institutions—that in our opinion have the potential of yielding the highest benefits for debugging, profiling and tuning software on many-core chips. Finally, we introduced some of the emerging tuning techniques—based on high level domain spe-

cific languages and automatic search of the potential optimization space—that have the potential for radically improving the efficiency and accuracy of performance tuning on many-core platforms.

As further reading, we recommend interested readers to consider the study listed in Ref. [7] which summarizes the key benefits and features of some of the most well established tools suitable for multi-core software development, debugging, profiling and performance tuning.

References

1. Mars J, Williams D, Upton D, Ghosh S, Hazelwood K (2008) A Reactive Unobtrusive Prefetcher for Multicore and Manycore Architecture. Proceedings of the Workshop on Software and Hardware Challenges of Manycore Platforms 2008, 41-50
2. Nellans D, Sudan K, Balasubramonian R, Brunvand E (2010) Improving Server Performance on Multi-Cores via Selective Off-loading of OS Functionalility. Proceedings of the 10th Workshop on Interaction between Operating Systems and Computer Architecture
3. Magnusson P S et al (2002) Simics: A Full System Simulation Platform. Computer, 35(2):50-58
4. Miller J E et al (2010) Graphite: A Distributed Parallel Simulator for Multicores. IEEE International Symposium on High-Performance Computer Architecture.
5. Acumem/RogueWave (2007) Performance Analysis for HPC/Multi-core Systems. http://www.acumem.com/images/stories/AcumemSlowSpotter.pdf. Accessed 11 January 2011
6. Samuel W, Datta K, Carter J, Oliker L, Shalf J, Yelick K et al (2008) PERI: Auto-tuning MemoryIntensive Kernels for Multi-core. Journal of Physics.
7. Spath D, Weisbecker A, Hebish E (2010) Market Overview of Tools for Multi-core Software Development. Fraunhofer Verlag
8. Windriver (2010) White Paper: WindRiver Simics for Multi-core Systems Development. http://www.windriver.com/whitepapers/whitepaper.php?f=WP_Multi-core_Bug_Simics_0410.pdf. Accessed 11 January 2011

Chapter 7
Many-core Virtualization and Operating Systems

Abstract In this chapter we focus on the design of future operating systems that can support, in a scalable manner, chips with hundreds of cores. We evaluate the fundamental design principles—space sharing, support for heterogeneity, power efficiency, virtualization and layering—and the reasons behind these principles, as well as the forces driving a departure from today's established concept. In the second part of the chapter we introduce the most promising experimental operating systems—such as Corey, fOS, HeliOS, Tessellation and Barrelfish—to exemplify how these principles can and most likely will be implemented in commercial operating systems. Finally, we shortly look at emerging weak signals that may add further features to operating systems that can support many-core chips.

7.1 Introduction

Suitability of current mainstream operating systems for chips with potentially hundreds and thousands of cores is one of the most hotly debated subjects in the academic and industrial community. The debate is fueled partly by the uncertainty surrounding how exactly such chips would look like, but most prominently by findings that suggest that some of the concepts at the foundation of current operating systems may face challenges when extrapolated to large number of cores and still limited amount of on-chip memory and external memory bandwidth.

A recent analysis [1] of the most contended data structures in the Linux kernel indicates that these are related to I/O operations (about 50%), scheduling (about 25%) and timer management (about 9%). This observation is consistent with other measurements of the cost of updating shared data structures from multiple cores [2], indicating close to linear increase of latency with the number of cores. The visible impact, as we scale the number of cores up, is reduced responsiveness of the operating system—in fact, the ratio of OS cycles (number of cycles spent performing operating system tasks) versus useful cycles (cycles spent executing user programs) is increasing, reducing the overall usage level of many-core processors.

A. Vajda, *Programming Many-Core Chips,*
DOI 10.1007/978-1-4419-9739-5_7, © Springer Science+Business Media, LLC 2011

There are fundamentally two issues limiting scalability of current operating systems to many-core platforms. The first is related to memory access cost: while the number of cores and total on-chip memory may be increasing, the amount of memory bandwidth per core is actually decreasing; however, at their core, current operating systems still try to maximize utilization of the processor, at the expense of cache and memory performance. Time-shared scheduling inherently means pre-emption, with drastic consequences for the cache miss rate and cache utilization metrics—often resulting in more wasted cycles, rather than improved utilization of the processor.

The second reason is related to the scalability of the shared memory model. As the measurements of the contention for Linux kernel data structures show, sharing information on tasks and scheduling—a must in any symmetric multi-processor system—adds significant overhead in the kernel that increases with the number of cores, thus increasing the share of processing performed in the kernel. This data actually show a more subtle underlying problem: simultaneous access from multiple cores to the same memory locations increases the cost associated with maintaining cache coherence to levels that directly influence the performance of the software (in our case the performance of the operating system).

Some of these bottlenecks can be addressed through careful tuning, cache-aware optimizations and ever finer scoping of contention management techniques such as various types of locks. We believe however that experimental data from many types of applications show that the scalability of any software is eventually limited by the resources it shares globally—much the same way as the sequential portion of any parallel software limits its scalability. This, in fact, is a corollary to Amdahl's law: since shared memory access requires serialization, shared memory areas will define, when pushed to the extreme (everything else is parallelized), the sequential portion of the software—and, consequently, the limit to its scalability.

In this chapter we will analyze some of the emerging concepts in operating system design that indicate the direction operating systems will likely take in the future. At the time of writing this chapter, few of these are mainstream, established concepts, though some ideas—such as task-based and work-stealing based mechanisms—have made their way into current systems.

7.2 Fundamentals for a New Operating System Concept

There are three trends that are observable in the design of massively multi-core chips, all driven by the need to maximize the raw processing capability of modern chips within a certain power budget. While all these pose extra challenges and create new constraints, all create opportunities for new approaches to operating system design as well.

The first trend is the shift to *more but simpler*, rather than *fewer and bigger cores*. This trend is fueled by power constraints, as lowering frequency and removing logic allows more cores to be instantiated within the same power budget and transistor count (albeit with lower complexity). The primary reason this did not

happen faster is the large body of legacy, non-parallelized applications that simply cannot make use of more (and even less, slower) cores. However, the trend is identifiable in most new chip designs and is in fact one of the underlying assumptions throughout this book.

The second trend is the emergence of *heterogeneous architectures*. Heterogeneity is driven both by legacy, non-parallelized applications (that require bigger, faster, more complex cores) as well as by the significantly higher power efficiency of specialized hardware solutions compared to general purpose cores. As long as hardware accelerators can outperform software by at least one order of magnitude, we will see specialized hardware components; similarly, as long as there's no universally applicable method for running sequential applications faster, there will be cores with different capabilities, grouped together on the same chip.

The third trend is connected to recent advances in chip design in general and memory design in particular that enable *more memory to be placed on chips*. There are two major technologies that hold the promise of better memory structure: three-dimensional stacking of processor cores and memory cells as well as novel memory designs—such as embedded DRAM—that allow more capacity at lower cost and lower footprint. We don't believe these advances will reverse the trend towards less memory bandwidth per core within modern chips; however, it shall be possible to slow down or even stop this trend, hence stabilizing new designs at some sustainable levels.

What do these three tends mean for operating system design?

An interesting insight based on the first trend is that it is likely that the average ratio of threads per core will continue to decline and eventually approach the ultimate limit of one thread per core; this insight is supported by the observed limit on scalability in terms of threads in existing applications, coupled with the trend of increasing amount of cores. If this indeed will be the case, it will have a dramatic impact on how we think about computers, something we did not experience ever since the birth of modern computers: processing cores will become a resource where limited waste actually is possible and even beneficial, as we will not be able to benefit from trying to get the cores do something else useful in case the threads currently running will stall—for the simple reason that *there will not be sufficient threads.*

An immediate consequence will eventually be that time sharing of cores will not be needed anymore. This idea has only recently surfaced in the academic research and in presentations given by e.g. members of the Microsoft operating system architecture group [3]; it is, at first, so counter-intuitive and against the conventional wisdom that it's hard to accept by a community that was so used to trying to squeeze the last bit of performance out of processor cores. However, as we'll detail in subsequent chapters, the benefits are immediate, even at intuitive level: less complexity in the operating system; better—and user-controlled rather than OS dictated—usage of on-chip memory and cache; significantly improved scalability of operating systems.

The second trend—shift towards heterogeneity—introduces its own set of constraints and opportunities. Operating systems cannot rely anymore on a pure or

semi-pure (augmented with NUMA semantics) symmetric multi-processing architectures, instead will be forced to deal with a wide range of combinations of cores with different characteristics. Different parts of applications will have different requirements as to where should be executed (different ISAs, different speeds etc) and the operating system has to provide a mechanism for the applications to convey this information to the OS, as priorities will not be sufficient. In addition, operating systems will have to become much more adaptable, being capable of executing on different types of core architectures and providing abstraction levels to pure, non-programmable hardware accelerators. We will evaluate some of the most promising approaches—such as architecture signatures, affinity information, user-driven adaptability etc—in a dedicated sub-chapter.

The third trend is likely the most controversial, as a large slate of the academic and industrial community will still argue that the trend of declining memory bandwidth per core will continue. We believe however that we have reached a level where no further decrease is acceptable without such a dramatic loss in performance that the chips would actually become unusable; therefore, either the pace of increasing the number of cores shall slow down, or new ways of incorporating memory shall be implemented. We believe the second approach will eventually prevail, based on recent developments in technology, primarily 3D stacking with high-speed short interconnects between cores and memory and novel ways to build smaller, more power-efficient embedded memories (embedded DRAM or emerging memristor-based solutions are just a few examples). The consequence for OS design is that, though memory efficiency will remain extremely important, there is a possibility to shift focus slightly and optimize within the same memory constraints that we are seeing today.

Summarizing the design constraints on operating systems for many-core chips, we believe operating systems will have to undergo a significant change in how resources are managed. We believe there shall be a gradual shift away from time-shared approaches towards novel trends, such as space-sharing that we'll elaborate on in subsequent chapters. At the same time the operating systems will have to embrace pervasive heterogeneity in hardware which will directly impact the hardware abstraction concepts as well as the interaction between applications and operating systems; applications will have to provide much more details and with increased dynamicity on their resource and processing needs. How exactly this can be achieved has been the subject of several research projects recently and we'll evaluate the most promising ones.

Beside these hardware driven trends, the emergence of *power-aware computing* will put additional constraints and requirements on how operating systems work. Today's SMP systems have very limited support for power-saving modes at system level, even in cases the available workload would not require full utilization of the hardware. This phenomenon will be even more visible when systems comprise hundreds to thousands of cores, as the probability of not needing all cores at all times will increase. Hence, operating systems for many-core processors will have to incorporate strategies and methods for managing power at the system level, through dynamic on/off powering of processing elements.

7.3 Space-shared Scheduling

As the number of threads per core will decrease, while the number of cores will increase significantly, time-shared scheduling will increasingly become a bottleneck and source of unnecessary complexity in the operating system kernel. The fundamental reason behind increasing amount of operating system cycles that produce diminishing returns is the complexity of the optimization problem that the OS needs to address: mapping a very large number of threads onto a large number of cores, while considering constraints related to memory locality, affinity, cache usage etc. This has been recently recognized as a potential bottleneck, both in the academic and commercial community.

The most significant proposals yet center on the idea *of space-sharing computing resources* as opposed to time-sharing. It rests on a simple basic design choice, with far reaching consequences: in a many-core chip, memory access characteristics are far more important than close to 100% utilization of computing resources, especially if managing cores at fine grained level increases the share of the cycles consumed by the operating system; in other words, many-core operating systems shall optimize for memory usage and latency and reduction of OS complexity, instead of traditional core usage efficiency.

The essence of the space-shared approach is to allocate several cores completely to one application and let the application manage any finer grained scheduling itself, thus making sure that the operating system will not interfere with applications, neither will it try to micromanage all the threads available. In this context, some of the cores will be reserved for the operating system where no other applications will execute, hence simplifying the overall design and making the approach to memory and kernel integrity protection more robust.

This approach makes a space-shared operating system somewhat more similar to hypervisors than to traditional time-shared operating systems: it 'hands out' computing resources (cores) to applications, but it is not involved in how the applications use those cores. The analogy to hypervisors is obvious: a hypervisor hands out computing resources (virtual machines) to applications (in that case, operating system instances), but does not schedule those instances for the applications executing within the guest operating systems.

Figure 7.1 gives a comparative overview of the concepts of space-shared and time-shared scheduling.

7.3.1 Architecture of a Space-shared Operating System

It's very likely that we will see a large number of different space-shared OS designs, each with its own architectural flavor. This trend is already visible today: the most well known experimental OSes—Corey [4], Tessellation [5], fOS [6], Barrelfish [7] etc—all have a slightly different take on the same fundamental principles.

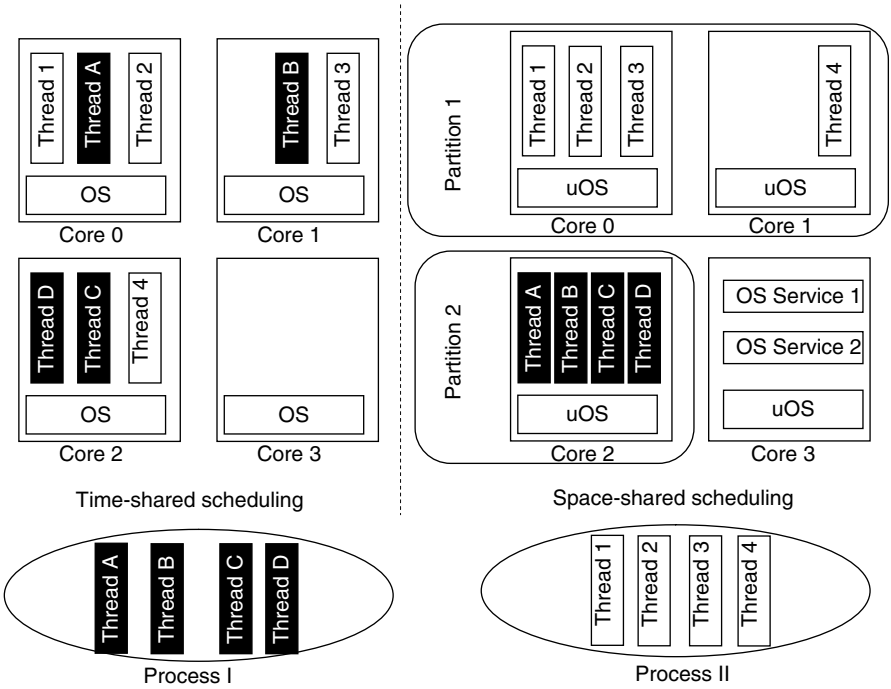


Fig. 7.1 Time-shared and space-shared scheduling

There are however a few basic concepts that will underpin any architecture: space-based separation of applications and space-based separation of operating system and application; no pre-emption and no central operating system scheduler. All space-shared OSes available today share these traits, with the notable exception of Tessellation when it comes to scheduling.

A generic architecture of a space shared operating system is visible in Fig. 7.1. The operating system kernel and all its services are executing on a set of special cores, dedicated to the operating system alone. Internally, on the operating system cores, the operating system may still have either a micro-kernel or monolithic kernel architecture, completely invisible to the applications—as noted previously, the internal architecture of the operating system is orthogonal to how resources are managed.

An interesting design choice, from an operating system architecture perspective, is the handling of OS executives located on each application core. These extremely simple kernels—in fact libraries—will provide a very basic local hardware abstraction layer and the gateway to all the other OS services, including memory management and peripherals. It's an important design choice how these executives will interact with the other parts of the OS: may execute in the same kernel space as the central OS functions—hence resulting in a monolithic structure—or in separate address spaces, only interacting with the central functions through explicit messaging—which would give a micro-kernel approach. We believe that in the long term, taking also heterogeneity into account, a micro-kernel, separate address-space

approach will win out as it will provide better isolation between applications and OS and will support heterogeneous processors where a shared address space may not even be possible. Another important aspect to consider will be the likely unreliability of many-core chips, where core failures will be more common; providing strict isolation of central OS functions from potentially faulty application cores will enhance the reliability of the overall system.

Scheduling—one of the fundamental tasks of any OS—is greatly simplified in a space-shared architecture. Essentially, the operating system's role is reduced to that of a resource manager, very similar to the memory manager: allocating cores to the applications and performing the basic hardware house-keeping (interrupt handling, core restart and initialization etc). As we'll discuss in subsequent chapters, while this removes one of the scalability bottlenecks of operating systems, it also increases the potential for resource waste, leaving the control in the hands of the applications.

A special type of scheduling was implemented in Tessellation. This experimental operating system, developed at Berkeley, uses space-sharing as the foundation of the computing resource management; however, in addition, it also retains the time-sharing approach to scheduling. At any given time, an application has complete control over a set of cores; however, the OS may decide to swap out the current application and let another application use exclusively the same set of cores for a certain period of time. It's important to highlight that this is not traditional time-sharing, as it's operating on a set of cores at a time, as opposed to the traditional approach of managing the resources of each core individually; in addition, Tessellation treats all the threads of an application together—another difference to the individual, thread-based scheduling choice of traditional operating systems.

In case the number of threads within an application is greater than the number of cores allocated to that application, there will be a need for application level scheduling. In theory, from the operating system perspective, this is just another application function, completely invisible to the OS—the application is fully in charge. In practice however concrete realizations may range from completely application level scheduling—similar to today's user level threads such as those of the Erlang language—to an application decided, but OS performed task switch. In this later model, the OS provides no pre-emption support, just system calls that can perform a switch to another, application specified thread, much in the similar way as collaborative multi-tasking was implemented in Windows 3.1. We would like to emphasize however, that conceptually the task still lies with the application; the OS may provide pre-implemented schedulers and system calls, but these are more a 'good will' service than a regular, mandatory OS function.

7.3.2 Benefits and Drawbacks of Space-shared Operating Systems

A space shared approach to computing resource management simplifies greatly the task of an operating system: instead of dealing with thousands of threads and at-

tempting to schedule those threads on tens or hundreds of processor cores, it manages processing cores as a bulk resource, similarly to memory. In fact, what the operating system will do is to allocate cores much like any OS does with memory; even the enforcement of protection of resources between applications is very similar to protecting memory from access from other processes. If previous experience with memory management is anything to go by, such an approach is guaranteed to scale to thousands of cores and most probably well beyond that—hence eliminating one of the key bottlenecks in today's time-shared operating system.

The impact on memory performance is also significant. A space shared OS will never pre-empt an application executing on any core, as any application core is dedicated to that single application and hence it will not compete with other applications. This has dramatic impact on cache behavior, as it will now be completely under the control of one application that will never be pre-empted; consequently, application-specific cache usage strategies can be implemented, enhancing the application's performance.

The "one core, one application" principle allows deterministic power management, both globally and locally on a core; globally all non-allocated cores can be switched off (or put to a very low power state) to save power; locally, on a core, especially in the case of a "one core, one thread" model, the core can be deterministically put into low power modes when the thread is idling and waken up when the thread again has work to do. This way a globally optimal power management can be achieved with very little overhead.

Improved security and better support for managing un-reliable hardware are additional assets. As the OS will execute on just a few cores, isolated from the other applications and cores, itself will not be impacted by any HW fault experienced by an application core. The OS will have the means to detect such failures and safely take the core out of service, without impact on other applications (as the failed core was used by just one application) or on the OS itself.

There are two fundamental arguments that are raised against space-shared operating systems. The first one has to do with the perceived waste of resources; the second with the shift of operating system functions to applications.

Since cores are allocated to applications, the OS has no direct control over how efficiently—or badly—the cores are used, hence an application that does not "behave" properly will drain system resources leading to inefficient usage. To add to this issue, I/O intensive applications may have some of their cores idling for a significant amount of time waiting for external operations to complete.

The first of these problems may be addressed by giving the OS the possibility to monitor core usage and revoke and re-use cores that have been idling for a long time. In such cases, the impacted application will have to either re-apply for the cores that were reclaimed by the OS or implement internal scheduling and re-locate computations to other cores. While this is not an optimal solution, it may help in some resource-constrained cases; future research may yield more sophisticated algorithms. A second approach to the problem is Tessellation's solution to also time-multiplex the allocation of cores, hence reducing the idle time of under-utilized cores.

For the moment however, this remains indeed one of the drawbacks of space-shared operating systems that needs to be addressed.

The second problem—cores idling around waiting for external events or I/O operations—basically reflects a trade-off between OS design simplicity and scalability on one hand and maximizing core usage on the other. It's not at all clear that micro-managing *all* cores would actually improve the usage significantly; as we described in the beginning of this chapter, it means a significant overhead in the OS. In addition, in truly many-core systems, a couple of idling cores translates to only a low percentage of non-utilized resources which may be an acceptable solution. All in all, we believe this issue is not significant and it will be even less so as the number of cores will continue to increase.

7.3.3 Summary

We believe the space-shared approach represents the likely direction mainstream operating systems will take in the future, as it provides a way to build simpler, more scalable and less intrusive operating systems. Space shared OSes will increase the flexibility for applications, but will also shift some of the OS responsibilities (e.g. thread scheduling) to the application domain. Some issues remain to be addressed—fairness and better control of computing resource allocation and porting of legacy applications are the chief challenges that the OS community will need to address.

7.4 Heterogeneity

Heterogeneity in processors can take multiple forms, along three dimensions: instruction set architecture and core capabilities, programmability and on-chip memory.

Cores on a chip may share exactly the same instruction set architecture (ISA), may have partly overlapping ISAs or could have different (disjunctive) ISAs. In addition, some cores may have enhanced capabilities: may operate at different frequencies or have other enhancements that make them more suitable for specific tasks: deeper pipe-lines, out of order execution capabilities, higher number of hardware threads etc. While the supported ISA determines the portability of a specific piece of code from one core to the other, the capabilities will define the efficiency of execution of that code on different types of cores. In state-of-the-art processors is increasingly a common practice to mix cores capable of executing software with hardware accelerators for various well defined and usually hard-coded tasks. Supporting these different types of components with different capabilities in a transparent way is one of the issues modern many-core operating systems will have to address.

As the number of cores will increase, on-chip memory will increasingly be distributed and processor cores will experience different access speeds to different parts of the memory, even if the hardware will present a uniform address space.

In addition, some cores may have larger local on-chip memories, giving these an advantage over the other cores when executing code which requires access to larger memory areas. Many-core operating systems will have to consider memory locality and access latencies in order to maximize the overall efficiency of program execution.

This double challenge—heterogeneity of core capabilities and non-uniformity of on-chip memory architecture—is one of the key issues that many-core operating system need to address. In this chapter we'll present and evaluate some of the promising approaches that have been proposed so far.

7.4.1 Managing Core Capabilities in Single-ISA Chips

The approach to tackling heterogeneous core capabilities depends to a large extent on whether the cores support the same ISA or not, as this defines to what extent applications may be moved from one core to the other.

Single-ISA, heterogeneous systems—where some cores have improved capabilities—have been shown to be more suitable for applications with a varying mix of sequential and parallel sections [8–12]. The basic idea is rather simple: as soon as a sequential portion of the code is reached, the execution shall move to one of the higher performance cores in order to reduce execution time; whenever a parallelized section of the code is executed, the execution can be moved back to the simpler cores.

There have been several methods proposed for determining when to move the execution of programs from one core to the other. A key characteristic of all these methods is that it relies on the application to provide additional information, beyond the 'traditional' thread priority in order to enhance the operating system's ability to decide the best thread placement.

The most explicit approach is to mark all parts of the code that need to execute on special hardware—such as higher capability cores—as special sections, akin to critical sections. This will allow the compiler to generate code that will explicitly trigger the OS to migrate the execution of the special section to the specific type of core (and back, once the special section completes). This approach also enables further optimizations, such as pre-loading of data needed by the special section to the cache of the target core, advance scheduling etc.

Using architectural signatures to convey information about the characteristics of applications is a simplified version of the approach based on explicit markings. An architectural signature describes the nature of the application (e.g. amount of parallelism, non-parallelized parts) as a recipe for the OS to decide where to execute which part of the application. The architectural signature can be provided either by the programmer or can be generated by the compiler, for example relying on explicit markings in the code.

Implicit exploitation of the benefits of heterogeneous systems has been researched from the perspective of access to shared memory. Instead of replicating

memory content to the cores that need to access it, an approach based on migration of the execution to the location of data—fixed to a specific core or set of cores—is used (also called the *Follow the Data pattern*, see e.g. Refs. [11, 12]). The fundamental observation is again that accessing shared memory areas requires sequential execution of the program code; hence the execution shall be moved to a core with enhanced capabilities. This approach has the added benefit of reducing or completely eliminating the potential for dead-locks by guaranteeing that shared memory areas are only accessed from one core and improving cache behavior, as all accesses to a certain memory area are always from the same core and consequently, the probability of having the data in the cache is higher, while the probability of cache line invalidation is practically zero.

7.4.2 Managing Core Capabilities in Multi-ISA Chips

Multi-ISA chips have emerged as viable alternatives for highly cost-sensitive or power-constrained systems. Typically such processors will need to execute software composed of software entities requiring specialized architectures—such as digital signal processing, network processing, graphics and similar—and components best executed on general purpose hardware. Example domains include mobile devices and networking equipment.

Multi-ISA chips pose a different set of challenges for many-core operating systems. One of the key requirements from application point of view is same or similar interface to the operating system from all the cores in order to simplify the task of designing application software. A shared memory space or some uniform inter-core communication mechanism—such as message passing—is also desirable in order to remove the complexity of managing heterogeneous computing environments. Ideally, the only difference to homogeneous computing shall be in the *affinity* of particular pieces of software for a specific ISA and the different compilers used to generate code for the different targets.

One of the promising approaches builds on the concept of *satellite operating system* [13]. In this architecture, the main processor cores—typically general purpose cores based on e.g. X86, Power, MIPS or similar instruction set architectures—will execute the main operating system (which may be an SMP or space-shared OS). The other processor cores will run specialized operating systems, but exposing the same kernel interface, called satellite operating systems. The main and satellite OS instances will form an asymmetric multi-processing system, but exposing a unified OS interface towards the applications.

On the application side, heterogeneity is managed through *ISA affinity*. Specific parts of the application will be marked as ‘would best execute on x ISA core’, using e.g. the concept of special sections that we introduced in Chap. 7.4.1. These parts will be compiled for both the main processor cores and the preferred core; which version will get executed will be decided during run-time and will depend on the scheduling decision that the operating system will take.

This approach therefore requires a specific scheduling mechanism in the operating system. Essentially, the OS will need to take into account the affinity of different parts of the application and try to move the execution to those cores whenever possible. From the application perspective, this will be visible as a simple inter-core thread migration; however the main and satellite operating systems will have to make sure that the right ISA version will be executed. A shared address space between cores of different ISAs will contribute to the impression of homogeneity, as memory can be addressed equally from the different cores.

Such an approach is commercially available as part of the OSE [14] real-time operating system from Enea, supporting chips with e.g. ARM and DSP cores and shared memory space. On the research side, HeliOS, a prototype OS from Microsoft advocated the concept of satellite OS and showcased how such an architecture can be implemented in practice.

7.4.3 Summary

Heterogeneous ISA many-core processors will likely have a niche appeal due to the associated software complexity. However, operating systems can significantly help reducing this complexity by providing a uniform operating system view of the system, providing a mechanism for expressing ISA-affinity of the software and supporting multi-target compilation and creation of executables. These concepts have already been proven in commercially available operating systems.

7.5 Power-aware Operating Systems

Traditionally, operating systems aimed at balancing the computational load among available processing elements in order to avoid overloading certain cores. The benefits of such an approach are obvious when the load is high, even approaching the maximum capacity of the chip; however, at low load levels, this approach is actually counter-productive as it triggers undue re-scheduling of tasks and keeps unnecessary amount of processor cores busy. In fact, SMP systems have largely failed in exploiting the capabilities of modern chips simply because the OS architecture makes it inevitable that the largest possible amount of cores are active—instead of the lowest possible amount, the goal of power aware scheduling.

Recently, power awareness has been steadily climbing up on the list of important features for computational systems. The exploding cost of powering data centers, the infrastructure limitations as well as environmental awareness are all contributing factors to the drive for lower values of the watts/computation metrics. Essentially, there are three levels at which power consumption can be regulated in a computational system: HW/firmware level; operating system level; and application level.

At HW level, several techniques—such as aggressive power gating and fine-grained adaptation of performance levels—have shown potential for savings without

compromising performance, as long as these are supported by the operating system as well. Essentially, all techniques focus on using 'just enough' of the chip's transistors at 'just enough' performance level to meet the application requirements; therefore input from OS and applications is crucial for the performance of such approaches.

On the OS level, power efficiency can be improved by taking into account the capabilities of the underlying hardware, but, more importantly, through careful resource management techniques. In order to save power, the computations shall be merged to as few cores as possible, in order to be able to set the other cores into low-power modes—this however requires a radical departure from the traditional load-balancing based approach.

Space shared operating systems naturally support such designs: cores are either allocated to applications or are 'free'—and free cores can safely be put into a low power mode without impacting the performance of the system. In a space-shared OS further techniques, such as frequency scaling of cores can contribute to power saving: for a highly parallel application, it may be more power efficient to allocate double amount of cores, but with each core running at just half of maximum frequency. In such a scenario, overall power consumption can be reduced by at least 50%, while still delivering the same performance.

7.6 Virtualization in Many-core Systems

Virtualization as a concept dates back to the seventies of the last century [15], but it entered the mainstream of computing only during the last decade. The primary promise of virtualization is *efficiency* in the usage of computation resources, achieved through de-coupling of the real hardware view from the view offered to the operating systems and the applications running on top of these.

The main driver for virtualization was—and still is—efficient exploitation of the computational power of chips that would otherwise idle, for a significant share of the time, without doing useful work. Virtualization's answer to this problem is the creation of *virtual machines* that look and behave like real hardware machines for the operating system and applications running on top of it. Multiple virtual machines are managed and time-multiplexed onto the real hardware by a small executive called the *hypervisor* (or virtual machine monitor, VMM), that makes sure that all virtual machines get a fair share of the processor's time and—ideally—the processor will never have to idle while waiting for an application to be executed. The key feature of virtualization is the possibility to create, stop, resume and delete virtual machines easily, as well as the possibility to move VMs around to processors with lower load.

At a closer look, however, operating systems and hypervisors share a lot of common traits. Both manage the computing power of the underlying processor; both deal with scheduling of applications (either user applications or complete stacks of operating system and applications); both aim at balancing the computational load of the system. Essentially, hypervisors add an extra layer of scheduling and resource management in order to manage the diversity in the requirements for operating systems of different applications—to fill the gap a single OS cannot do.

What will be the role of virtualization in many-core chips?

We believe operating systems will move to a space-sharing based scheduling of computations, as opposed to today's time-sharing methods. In this context, the need for time-sharing a core's resources at a lower level than the operating system is basically removed, as cores will be so abundant that such an approach would be inefficient. What will not disappear however is the need to manage a diversity of applications with different requirements on the underlying operating system and hence the need to support multiple operating systems on the same chip.

There is an interesting trend in the operating systems for many-core systems that can be observed in most of the experimental OSes we will explore. Essentially, all the operating systems targeting many-core systems have a layered architecture: the lower layer takes care of the space-sharing and basic management of resources, while the actual scheduling of application threads is left to the higher layer—usually called the *library operating system*—that tends to be application dependent. In many ways, the lower level corresponds to traditional hypervisors, while the higher layer resembles closely today's operating system.

Consequently, we believe we will see a double trend when it comes to virtualization and hypervisors: the disappearance of the need for time-sharing individual cores between operating systems, coupled with an increasing integration of and blurring of the dividing line between functionalities offered by hypervisors and operating systems. Space-shared operating systems may take over the remaining role of hypervisors—management of OS diversity—while hypervisors could evolve into something similar to the basic layer of a space-shared OS. This trend is also underlined by the strategies outlined by leading virtualization providers: growing the role of the hypervisor, that can execute 'just-enough operating systems' (JE-OS) [16], tailored to the needs of individual applications and only providing the services needed by that specific application.

7.7 Experimental Many-core Operating Systems

In recent years several experimental operating systems focusing on many-core chips have been proposed. Most of this research has either been performed by commercial operating system vendors or has been supported by these—an acknowledgement, in our view, of the need to reconsider the current OS principles and shift the OS design focus to new directions. None of these experimental OSes made their way yet into mainstream, commercial use; however, several ideas—related to managing heterogeneity and intra-chip communication—have been incorporated into commercially available operating systems.

In this chapter, we'll survey the most promising experimental OSes. We will focus primarily on the strengths of the technologies implemented in these operating systems, especially in the areas of scheduling, management of heterogeneity and power efficiency.

7.7.1 Corey

Corey [4] was developed at MIT's Parallel & Distributed Operating Systems Group, the same group that initially developed the concept of exokernel. Corey's basic principle centers on the idea of shifting the control of what exactly gets shared between cores—both in user space and kernel space—to the application.

There are three fundamental concepts in Corey: shares, address trees and kernel cores (mapped to the concept of 'pcore', explained in subsequent chapters). Shares provide a mechanism for applications to control precisely how kernel structures are shared; address trees control which page-table entries are private to a core and which ones are really shared; kernel cores allow space-separation of kernel functions and applications, by defining a set of cores dedicated exclusively to the kernel. In addition, Corey supports the concept of library operating systems, first introduced in connection with the exokernel concept.

In the following sub-chapters we examine these concepts more closely.

7.7.1.1 Shares

Fundamentally, Corey is an exokernel operating system and hence applications are required to use 'library operating systems' to implement functionality beyond the basic management and protection of hardware resources. By default, he Corey kernel ensures that cores use only local resources and share no state between cores—it's the responsibility of the applications to create sharing whenever needed using the shares kernel interface.

Hardware resources are modeled in Corey as five classes of objects: shares, segments, address trees, pcores, and devices. Each object has a unique 64-bit id, called object ID, stored in a lookup table. Every core has access to a set of lookup tables, called *shares,* which define the range of objects it may access: a core can never access any object that is not included into one of its shares. Library operating systems have the possibility to define the shares of each core, and explicitly, the amount and limits of sharing among cores: core local objects will be mapped to private shares, while shared objects will belong to one of the global shares, mapped to each of the cores participating in the sharing.

7.7.1.2 Memory Management and Address Trees

In Corey, physical memory is represented through the segment (defined as a memory area of specific size) abstraction. By default, only the core allocating the segment may access it—unless it is added to one of the global shares. A key concept in Corey is that of *address trees*. An address tree is used to define the address space for each core: it maps virtual addresses to segments and maps virtual addresses to internal address trees that can be shared (referenced from core-local address trees) between cores.

This approach gives a fine-grained, dynamic mechanism for controlling which memory address areas get shared and how those are mapped to each core's address range. The key insight is that there is no need for a global memory structure in the OS, which limits the potential contention cases to explicitly defined areas (where possibility for contention is really due to application characteristics).

The concept of segments and shares is also used for communicating with devices. When an application opens a devices (requests access to it), it will get access to a segment belonging to the device; any read or write from or to the device will then happen through this shared memory segment.

7.7.1.3 Core Management and Scheduling

Physical cores are represented in Corey using *pcore* objects. Applications—or kernel services—are started on specific cores by requesting the kernel to execute a specific piece of software, configured using a stack pointer, a set of shares (to define what that specific piece of software will share with software executing on other cores) and an address tree (to define the initial address space of the software and core).

This design essentially implements a space-shared scheduling mechanism. Finer-grained scheduling policies—such as traditional time-sharing—may be implemented through library operating systems, instantiated to execute on application cores; however, from the Corey kernel point of view, the library OS is just a support function of the application.

In the standard library OS implementation (called libOS) Corey provides a many-core enabled variant of the traditional UNIX *fork* system call: *cfork* will start a new thread on a new core, using a copy (copy-on-write, to be precise) version of the parent's address tree. Sharing can be achieved again by passing a set of global shares to the new core's pcore object.

7.7.1.4 Summary

We believe Corey's approach—giving applications much more fine-grained control—is an incarnation of the principle we base our argumentation for a new OS approach on: the need for more fine-grained semantic information sharing between applications and operating systems. Shares, address trees and pcores are essentially mechanisms through which applications can control what gets shared at different levels and how cores are best mapped to the problem at hand.

7.7.2 fOS

Factored operating system (fOS) [6] is another prototype to emerge from the Massachusetts Institute of Technology, this time from the group that initially devel-

oped the concepts behind the RAW and Tile architectures (commercially available through Tilera). The fundamental concepts behind fOS are strikingly similar to those of Corey: space sharing as opposed to time sharing; separation of operating system and application domains; strict isolation and drastic limitation of sharing memory between cores.

fOS however takes the concept of isolation even further. All the operating system services are strictly factored out and communicate with other services and applications purely through messaging—in fact, operating system services may execute, in theory, anywhere, even on other processors, completely transparently for the applications. This strict enforcement of separation and 'share nothing' actually allows fOS to scale from single chips to large-scale data centers and cloud computing environments.

Operating system services are designed with scalability as a primary design requirement. Internally, each service (such as file system or memory manager) are implemented using a co-operative set of cores (called servers), invisible to the application. The OS design also makes sure that individual server faults are invisible outside the service domain, by automatically re-routing communication to server cores executing correctly.

fOS is still in early design phase, but it holds the promise of an operating system concept that can scale from individual many-core chips to cloud computing environments. The key to such scalability lies in the four design principles:

- share nothing, communicate through messaging
- space-share computing power instead of time-sharing
- design operating system services so that performance can be scaled up by adding more cores to the service group of cores implementing the service
- fault tolerance through automatically re-routing tasks handled by failed cores to other cores belonging to the same service group.

7.7.3 Barrelfish

Barrelfish [7] is the result of co-operation between ETH in Zürich and Microsoft Research. It is officially called a *multikernel*, i.e. an operating system that views a chip as a network of independent cores, assumes no inter-core sharing and moves traditional OS functions to a set of distributed processes communicating solely through message passing. It also aims at supporting heterogeneity in HW, primarily with respect to core capabilities within a single ISA domain. There are three design principles at the foundation of Barrelfish:

- make all inter-core communication explicit
- make the OS structure HW neutral
- view state as replicated instead of shared.

Barrelfish is essentially a micro-kernel based operating system. On each core, the OS kernel is factored into a privileged mode CPU driver and a user mode monitor

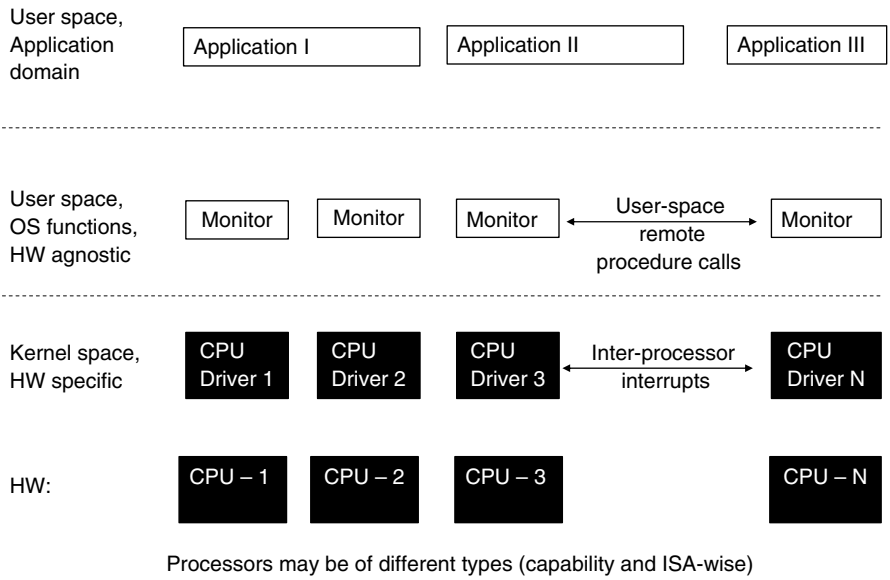


Fig. 7.2 Barrelfish operating system architecture

process; the rest of OS functionality—device drivers and system services—are run in user-level processes. Interrupts are routed to the appropriate core, handled by the CPU driver and delivered as messages to the driver process. Figure 7.2 shows the high level architecture of Barrelfish.

The CPU driver's main task is to enforce protection, authorization, time-slicing (if needed) and HW abstraction; it also provides same-core, fixed size message based inter-process communication facility. The CPU driver of any core shares no state with any other core and it's fully event-driven, single-threaded and non pre-emptive. In many ways it resembles an exokernel and has such an extremely small footprint that allows it to be entirely located in the core's local memory. It is the part of the Barrelfish operating system that is heavily hardware specific and needs to be customized for each new hardware version.

Monitor processes are responsible for system wide co-ordination and also execute as single core user-space processes. All global kernel data structures—such as memory allocation tables—are distributed and replicated in Barrelfish; monitors secure that these structures are kept globally consistent using an OS-specific agreement protocol. Consequently, monitor processes mediate application requests that require access to global data structures.

In addition, monitors also have a housekeeping function, implementing power save modes of operation, setting the core in low power mode when there's no application to be run. This becomes possible as inter-core communication is managed by the local monitor, hence it has a consistent and comprehensive view of the status of all tasks executing locally.

The concept of process is implemented differently in Barrelfish compared to traditional operating systems. On each core where the process might execute, it is represented by a local dispatcher object, which is invoked by the CPU monitor whenever it shall execute; each dispatcher object typically implements a core-local, user-space thread scheduler. Shared space semantics can be provided for threads within the same process—even if executing on different cores—either through actual sharing of the relevant hardware page table among the dispatchers or replication and consistency protocols between the dispatchers. Thread migration and load balancing is achieved through message based communication between dispatchers.

It's important to note in this context that Barrelfish is only responsible for time-multiplexing the dispatchers on a specific (and all) cores; all the mechanisms leading to shared memory semantics and thread balancing are implemented in user space. This way, Barrelfish actually offers a space-time multiplexed approach to scheduling applications.

An interesting concept in Barrelfish is that of the *system knowledge base*. It's essentially a database of hardware information implemented as a subset of first-order logic. It enables the OS to make efficient decisions with regards to e.g. allocation of drivers to cores, selection of message passing mechanisms or NUMA-aware memory allocation.

One of the weak spots of Barrelfish is the memory management sub-system. It is implemented as a capability system: all memory management is explicit, done through manipulation of capabilities, which are essentially just references to physical memory regions. In practice, this removes dynamic memory management from the kernel, which merely checks the correctness of memory operations, and moves it to user processes; while this approach results in a decentralized memory management, it puts unnecessary complexity into the user-space applications, without actually obtaining significant performance gains in comparison to optimized global memory managers.

All in all, Barrelfish is an interesting OS that proposes a de-coupled model, where core-local OS components only communicate with their peers on other cores through messages. It also advocates placing more responsibility into applications, relegating the kernel to basic HW abstraction and communication mechanisms. One of the strengths of the Barrelfish model is the potential to support different—potentially heterogeneous—hardware architectures, as it models the chip as a network of nodes, a model easily mapped to most types of available processors.

7.7.4 Tessellation

Tessellation [5] is an experimental client-side operating system developed at the Parallel Computing Laboratory at Berkeley. Its fundamental principle is space-time partitioning (STP), a combination of space-shared scheduling (slicing of available

resources into a number of entities called *partitions* in Tessellation) and time-sharing, whereas applications may be swapped out to give way to new applications, executing in the same partition. In many ways, a combination of application and space partition and time-shared scheduling is very similar to the concept and management of virtual machines, making Tessellation similar in some functionality to hypervisors.

In its approach to space-sharing and structuring of operating system services, Tessellation is very similar to fOS: it builds on a micro-kernel approach with scheduling responsibility delegated to the applications; it treats OS services as special applications that have their own partition allocated; it relies on message passing for communication between partitions. The key differentiating factor is time-multiplexing: applications have exclusive control over the resources in their partitions only for the duration of a scheduling quantum (which is likely to be much larger than in traditional SMPs), however may be suspended and replaced with other applications once their quantum elapsed.

Time-multiplexing in Tessellation is largely event driven: as applications only communicate with the outside world through messages, an application is ready to be scheduled when it has at least one pending, unprocessed external message. Some applications with high levels of QoS requirements or real-time characteristics may be pinned to a particular set of resources (primarily cores) and will never be swapped out in favor of other applications.

Quality of service management is one of the interesting features of Tessellation. It relies on monitoring and policing message flows between partitions and the outside world, approach made possible by having message passing as the sole communication mechanism between cores. For instance, in case two applications share equal parts of some global resource or service and there is an imbalance in the access pattern to the resource, superfluous access requests to that global resource or service are suppressed at the source—the offending application—in order to prevent contention and overloading of the resource or service.

Architecture-wise, the key components of the Tessellation kernel are the *Partition Mechanism Layer* and the *Partition Manager.* The Partition Mechanism Layer is essentially the HW abstraction mechanism that provides a machine-independent partition management interface to the policy implementation layer. It also provides the abstract messaging infrastructure for communication between the partitions.

The Partition Manager represents the policy layer that allocates and schedules resources to applications and partitions, based on the abstract, machine independent model provided by the Partition Mechanism Layer. It supports dynamic adjustment and resizing of application partitions, based on requests from applications. It is responsible for time-multiplexing as well, based on application priority and ready state (i.e., an application can be scheduled if it has at least one message that it needs to process).

Partitions in Tessellation can be fixed sized or dynamically sized. An application allocated to a dynamically sized partition may add, remove or modify the resources it controls through kernel APIs and it's the more common type recommended by the

OS. Fixed sized partitions are recommended for static applications and applications with high predictability requirements.

In summary Tessellation, while still in early phases, proposes an interesting model that aims at combining traditional—time-shared based—techniques with novel, space-shared approaches. It aims at mitigating the inherent problem of time-sharing, its collaborative nature, with coarse-grained timing control on the operating system level. While it's likely to remain an experimental OS, the principle of space-time sharing is one that will be re-encountered in future OS designs.

7.7.5 HeliOS

HeliOS [13] was developed at Microsoft Research, based on a previous research OS called Singularity. Its main goal is to address the issue of heterogeneous hardware by providing a single set of operating system abstractions across cores of different architectures or characteristics, coupled with a re-targeting mechanism that allows applications to be run on cores with ISAs best matching the performance requirements.

HeliOS aims to meet four fundamental design goals in the context of heterogeneous processors: heterogeneity in kernels, but with one set of abstractions; transparent inter-process communication; simplified deployment and tuning; encapsulation of multiple hardware architectures.

The solution to meet these goals in HeliOS relies on two basic concepts: satellite kernels and affinity metric supported re-targeting. We'll explain these concepts in the context of the four fundamental design goals of HeliOS.

7.7.5.1 Heterogeneity in Kernels

In order to manage cores with disparate characteristics, HeliOS requires that each type of core has an OS kernel which in turn supports a basic, common set of OS abstractions. These kernels are implemented as micro-kernels, and are called *satellite kernels* in HeliOS terminology.

Satellite kernels are executing largely autonomously of each other and are required to implement only a few APIs, primarily memory, process and thread management. The reference satellite kernel of HeliOS requires access to CPU, memory, a timer, an interrupt controller and the ability to catch an exception—which makes it a good candidate to run on most modern cores, including even specialized ones.

The key benefit of satellite kernels is the uniform view of the hardware that is exposed to the programmer; cores may differ in the *speed* at which applications of certain type will be executed; but otherwise all look similar from an application point of view. This is further enhanced through the usage of an intermediate representation of the applications.

7.7.5.2 Transparent Inter-process Communication

In order to provide efficient inter-process communication, the operating system must provide an abstraction of where applications are executing and, at the same time, guarantee that the communication will be as efficient as possible, especially if the two processes are co-located on the same core.

The main mechanism provided by HeliOS for application registry and discovery is a global namespace. It provides a unified mechanism to advertise services, both from applications and the operating system. Drivers may use it to publish peripherals and the services supported by these; the OS may advertise basic services such as the file system; applications can use it to discover each other. This mechanism is not new for HeliOS; it has been previously implemented in other contexts, such as the real-time operating system OSE [14].

Efficiency is achieved through the usage of the best communication mechanism available, behind the same uniform messaging APIs. For core-local messaging, a zero-copy, memory based mechanism may be used; for inter-core communication shared memory, on-chip networks or even network protocols such as Ethernet/IP are valid options. In fact, this allows HeliOS to scale across chips, similarly to fOS.

7.7.5.3 Simplified Deployment, Tuning and Hardware Encapsulation

Traditionally, heterogeneous architectures pose a number of challenges for application developers: the need to choose the placement of each computation; the difficulty of re-allocating (migrating) threads; the difference in performance for certain computational paradigms (such as shared memory or message passing) when deployed on different targets.

HeliOS attempts to simplify development and efficient execution of applications on heterogeneous hardware platforms through two mechanisms: affinity and two-phase compilation. Each process is allowed to express its—positive or negative—affinity to other processes or types of cores as a metadata managed by the OS and used to schedule applications. A *positive affinity* between two processes indicates a benefit from running the two on the same satellite kernel; similarly, positive affinity to a certain type of core gives a hint on the type of architecture the application could best exploit (e.g. speech encoding fits best on a DSP). In contrast, *negative affinity* indicates that two applications might interfere with each other when run on the same kernel or that a particular type of architecture is not optimal for a certain type of application; an interesting sub-case is the negative affinity towards itself: it indicates, in HeliOS terminology, that multiple instances of the same application shall not be co-located.

Affinities for each application are expressed in XML format in a *manifest* that shall accompany any executable. It is automatically generated during compilation, but it can be edited by the programmer in order to fine-tune the application. When processing a manifest, HeliOS prioritizes positive affinities, with CPU affinity be-

ing the tie-breaker and the one with highest priority. It's important to note that affinities are not constraints that will always be met, rather preferences that the OS will try—but may not always succeed—to accommodate.

Reliance on metadata alone is not sufficient however, especially in the case of chips with multiple ISAs. This is the main reason why HeliOS supports a two-phased compilation: applications are first compiled into a representation in CIL (Common Intermediate Language, the byte-code of Microsoft's .NET platform); when the target where the application will be run is decided, the CIL representation is transformed into executable code, through a translation into the ISA of the target processor core.

This approach has clear advantages in terms of flexibility over the *fat binary* approach, where versions for different ISAs are readily included into the delivered binary: it allows applications to take advantage of newer architectures without modification and allows deployment on any platform using HeliOS for which a CIL to native ISA compiler is available.

7.7.5.4 Programming Model and Handling of NUMA Domains

A process in HeliOS can only execute on one single satellite kernel, thus all its threads must execute on that kernel; communication between kernels is through message passing only. However, HeliOS does not mandate that one satellite kernel may only execute on one core only: the current design of the OS supports the concept of NUMA domain satellite kernel that we describe in this sub-chapter.

A NUMA domain satellite kernel presents a uniform memory access island of cores as one device to the rest of the system, essentially executing an SMP-type OS on those cores. As processes are locked to satellite kernels and communication between different satellite kernels only happens through message passing, this method essentially guarantees locality of data and makes remote memory accesses explicit, even on cache coherent systems. This design essentially reflects a design choice for non-shared memory space and enforcement of locality—some of the issues relevant from many-core systems' point of view.

7.7.5.5 Summary

HeliOS is a good example of OS design that embraces heterogeneity while using space-partitioning (through the concept of NUMA domain satellite kernels) to mitigate the challenge of scalability for many-core operating systems. Some of the ideas prototyped in HeliOS have proven to be successful in commercial offerings as well, especially in the embedded domain.

We believe the concept of affinity and its usage to guide the OS with regards to the specific needs of applications is a promising approach, which however will have to be made much more fine-grained—down to individual sections of code—in order to be usable in a truly heterogeneous context.

7.8 Possible Future Trends: the Return of Speculative Execution

Before we conclude this chapter on future many-core operating systems, we shall return to one of the ideas whose time came, passed but may be returning again: speculative execution [17].

Increased number of cores is good news for applications with lots of inherent parallelism and for applications that were carefully engineered to take advantage of multiple cores. However, there is still a large class of applications that have defied attempts at efficient parallelization and are the prime candidates to be executed on complex, high-speed cores. Precisely for this kind of applications, the idea of speculative execution has been extensively researched, but with very limited results. A major reason for that is that it is very difficult to decide at run-time when speculation will be successful. Relying on run-time information has not offered much gain. On the other hand, limit studies have shown that there is indeed a wealth of parallelism to be exploited [18], therefore speculation as an approach should not be abandoned. Provided that we can apply speculation where it pays off, the upside is that we can use the higher amount of transistors to perform speculative execution of sequential software at radically larger scale than attempted before.

The pre-requisite however, in our opinion, is to strengthen the interface between high level software and hardware/operating system, so that the operating system and the hardware can utilize detailed semantic knowledge at large scale to support execution of single threaded applications. One approach proposed recently [19] uses the possibilities offered by the design by contract paradigm—in terms of possible and/or realistic execution paths—coupled with speculative, run-ahead execution of sequential programs. Such approach can exploit chips with thousands of simple cores, running at lower speed for improving the performance of applications with limited amount of parallelism, without the need for rewriting the actual software.

Obviously, such an approach requires support from the operating system. Fast creation of speculative threads, efficient replication of data, detection and squashing of wrong speculative threads will all require mechanisms provided by the operating system. How this will be implemented is still an open research question; however this technology holds the promise of mapping and speeding up sequential applications on massively multi-core processors.

7.9 Summary

In this chapter we surveyed the most pressing challenges that operating systems will face when scaling up to massively multi-core processors. We focused on the two most important scalability issues: firstly, the increasing overhead of time-shared scheduling fueled by the cost of pre-emption and cache misses; secondly, the unsustainable nature of shared memory models, both in applications and in the operating system kernel itself.

We argue that the emergence of chips with large number of cores will trigger a re-structuring of operating systems and hypervisors along two dimensions: the approach to scheduling and the approach to the internal architecture of the operating system. Lessons from experimental operating systems indicate that the core of the operating system is likely to become much more like a virtual machine monitor focusing on the management of HW resources, isolation and very few basic services. Unlike VMMs however, this lower layer of an operating system for many-core processors is likely to have distributed, micro-kernel like architecture, with controlled, limited or no sharing between instances on different cores. This basic OS core will likely implement a space-shared scheduling strategy and have support built in for power efficient operation—made possible precisely by the usage of space shared scheduling methods. It's main goal is partitioning—a form of *divide et impera*—and support for diversity in higher layers of the software stack.

On top of this low level, basic OS we will see a proliferation of application tailored '*user (or library) operating systems*' characterized by strong semantic integration with the application (through the usage of affinities as in HeliOS or some other similar mechanism). The key trend here is that applications will get better—almost OS level—control of the resources needed for execution, a far cry from today's simplistic, thread priority-based approach.

References

1. Gough C, Siddha S, Chen K (2007) Kernel Scalability – Expanding the Horizon Beyond Fine Grain Locks. Proceedings of the Linux Simposium 1:153-166
2. Baumann A et al (2009) The Multikernel: a New OS Architecture for Scalable Multicore Systems. Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles: 29-44
3. Multicore requires OS rework, Windows Architect advises. http://www.networkworld. com/news/2010/031910-multicore-requires-os-rework-windows.html. Accessed 11 January 2011
4. Boyd-Wickizer S et al (2008) Corey: An Operating System for Many Cores. Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation: 43-59
5. Colmenares J A et al (2010) Resource Management in the Tessellation Manycore OS. Proceedings of the 2nd USENIX Workshop on Hot Topics in Parallelism
6. Wentzlaff D et al (2010) An Operating System for Multicore and Clouds: Mechanisms and Implementation. ACM Symposium on Cloud Computing.
7. Baumann A et al (2010) The Multikernel: A New OS Architecture for Scalable Multicore Systems. Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles: 29-44
8. Hill M D, Marty M R (2008) Amdahl's Law in the Multi-core Era. IEEE Computer
9. Shelepov D, Fedorova A (2008) Scheduling on Heterogeneous Multicore Processors Using Architectural Signatures. Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture
10. Saez J C, Prieto M, Fedorova A, Blagodurov S (2010) A Comprehensive Scheduler for Asymmetric Multicore Processors. Proceedings of the 5th ACM European Conference on Computer Systems
11. Vajda A (2010) Handling of Shared Memory in Many-Core Processors without Locks and Transactional Memory. Proceedings of the 2010 Workshop on Programmability Issues for Multi-Core Computers

12. Suleman M A, Mutlu O, Qureshi M K, Patt Y N (2009) Accelerating Critical Section Execution with Assymetric Multi-Core Architectures. Proceedings of the International Conference on Architectural Support for Programming Languages and Operating SystemsFollow the data
13. Nightingale E B, Hodson O, McIlroy R, Hawblitzel C, Hunt G (2009) HeliOS: Heterogeneous Multiprocessing with Satellite Kernels. Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles: 221-234
14. Enea (2010) Enea OSE: Multicore Real-Time Operating System (RTOS). http://www.enea.com/Templates/Product____27035.aspx. Accessed 11 January 2011
15. Popek G J, Goldberg R P (1974) Formal Requirements for Virtualizable Third Generation Architectures. Communications of the ACM 17(7):412-421
16. Krishnamurti S (2007) Get Juiced! VMWare's Executive Blog, http://blogs.vmware.com/console/2007/07/get-juiced.html. Accessed 11 January 2011
17. Ohsawa T, Takagi M, Kawahara S, Matsushita S (2005) Pinot: Speculative multi-threading processor architecture exploiting parallelism over a wide range of granularities. Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture: 81–92
18. Steffan J G, Mowry T C (1998) The potential for using thread level data speculation to facilitate automatic parallelization. Proceedings of the 4th International Symposium on High-Performance Computer Architecture: 2–13
19. Vajda A, Stenström P (2010) Semantic Information based Speculative Parallel Execution. Proceedings 3rd Workshop on Parallel Execution of Sequential Programs on Multi-Core Architectures

Chapter 8
Introduction to Programming Models

Abstract In this chapter we introduce the foundations for the programming models we consider most suitable for programming many-core chips: communicating sequential processes (CSP), the Actor model and the task based model. For each of these, we shortly survey and compare the most commonly available programming language implementations: the goal is to lay the groundwork for the more in-depth analysis and presentation in the next chapter. An important part of the chapter will deal with the comparative analysis of thread/process based and task based parallelism, as well as an analysis of shared memory based models.

8.1 Introduction

The computer science community was quite successful in proposing a large number of programming models and paradigms for tackling the issue of writing software for parallel computers. Each of these has its own merits as well as drawbacks; some have become widely adopted and used, some are confined to specific problem or computing platform domains. In the following two chapters we'll try to elaborate and illustrate the most promising ones, from the perspective of designing software for many-core processors.

At the root of them all, there are fundamentally two basic paradigms that define where a certain programming model will fit in: the *global state* based paradigm and the *de-centralized* (without any global state) paradigm. In a global state based paradigm, the computational system can be fully characterized, at any point in time, through a global, finite set of values and the system will evolve from one state to the next through a computational step, the execution of a finite set of instruction. These models are the natural evolution of the abstract Turing machine [1] based computational model and are a direct extension of the sequential, single processor model. It's extremely important however to make the distinction between the concepts of global state and shared memory, as these are not equivalent; as we'll see in this chapter, a programming model may be global state based while not sharing

A. Vajda, *Programming Many-Core Chips,*
DOI 10.1007/978-1-4419-9739-5_8, © Springer Science+Business Media, LLC 2011

any data between execution threads and similarly, shared memory may be used in a de-centralized paradigm.

In the *de-centralized paradigm*, there's no global state that would characterize the whole system: the system is made up of autonomous un-synchronized entities that may react in any order to external stimulus or requests coming from the other components of the system. The best known example of such a system is the Actor model [2], where the system is made up of a variable number of independent, un-bound entities that may communicate with each other asynchronously through messages and there's no pre-defined logic as to in which order the tasks will be executed.

In this chapter we will look at the three basic models of concurrency that influenced most of the programming models and languages we detail in the next chapter: the theory of *communicating sequential processes,* the *Actor model* and the *task based model.* While these are not the only possible fundamental models—general, shared memory based programming is a notable one—we believe that these shall be at the foundation of any model that aims at scaling to hundreds of cores. We show how these are at the basis of more elaborate models and how these basic models of concurrency influenced the development of support for concurrency in programming languages.

8.2 Communicating Sequential Processes (CSP)

The concept of *communicating sequential processes (CSP)* was proposed as a new language by Hoare in his seminal paper with the same title [3] published in 1978. In it Hoare defined a new concept for modeling concurrency that was later at the basis of a process calculus theory aimed at reasoning mathematically about concurrent programs. Initially however, the model was presented as a language for describing concurrent systems and introduced a number of concepts still found in many modern programming languages.

The language proposed by Hoare built on seven principles:

1. It adopted Dijkstra's *guarded command* concept [4] as a sequential control structure and the sole source of non-determinism. Briefly, a guarded command is a combination of conditional statements and program code that shall be executed if the conditional statement is true
2. It introduced a mechanism to indicate that sets of sequential commands (called sequential processes) shall be *executed in parallel;* these *processes are statically defined*
3. It defined the concept of input and output operations as *message sending* and *receiving* primitives that shall be used for communication between processes and between the system and its environment; messages are the sole way for processes to communicate
4. *Communication paths are explicitly named* and communication will only occur if a process names another process as a destination for a message and that process names the first process as a sender

5. *Incoming messages were used in guards* to steer execution; if multiple messages were available, the semantics of the language required that one was chosen, thus creating a source of nondeterministic behavior
6. *Guards were used as synchronization primitive* for a repetitive command: if all the sources named by the guards have terminated, the repetitive command also terminated
7. *Pattern matching scheme* was used to discriminate the structure of messages and facilitate selection of the right type of message to be processed next

According to these principles, the model of communicating sequential processes is based on the global state paradigm: the configuration of processes and their interaction is defined statically and any nondeterminism has a finite set of choices (thus the system exhibits bound nondeterministic behavior). This restriction has been lifted in later versions of the CSP model, but it's of little interest from our discussion's point of view.

The CSP model also introduces several important concepts. The *share nothing* principle is enforced by requiring processes to communicate exclusively through messages, without sharing any storage among them. Communication between processes is elevated to the status of first class primitive concept, to the same level as traditional arithmetic operations, emphasizing the overarching importance of messaging in a concurrent system without shared storage (memory). In the same context, usage of messages as guards is also a novel way for providing program control, as the primary mechanism to steer the program's control flow. By construction, CSP didn't need any of the other synchronization mechanisms—the lack of shared memory and the usage of messaging for synchronization purposes made all other forms of synchronization redundant.

One important characteristic of the CSP model is the *rendez-vous nature* of inter-process communication: a process may only transmit a message if the corresponding receiver process is ready to accept it; otherwise it will be delayed until the receiver is ready to process the message. This is an important restriction with important implications for how the processes will be scheduled on a real hardware.

The original language proposed by Hoare was later developed into full-blown algebraic process calculi with support for formally specifying, verifying and reasoning about the concurrency behavior of software systems. While it has found some applicability at small scale, especially in safety critical systems, it largely fails at large complex systems, thus remaining in the realm of theoretical research. The main practical merit of CSP, in our opinion, is the introduction of a compelling model for designing concurrent systems, based on small-scale, internally sequential processes that communicate with each other and their environment in a predictable and well specified way, without using any explicit sharing of data.

8.2.1 Practical Implementations of CSP

Probably the most well publicized concrete implementation of CSP was through the *occam* language [5], the programming language of choice for the transputers [6],

developed during the 80s. Recently the *Go language* [7], proposed by Google, revived some of the ideas from CSP in the context of a C-like language. In this chapter we review the main concepts—relevant from many-core programming point of view—offered by these two languages.

8.2.1.1 Occam

Occam was developed as the main language for programming the transputer family of processors, directly based on the CSP model (the original design was actually advised by Hoare himself). In its latest form—occam-pi [8]—it is supported through a university-developed compiler and encompasses major additions beyond the scope of CSP, such as dynamic process creation.

An occam application is defined as a collection of processes that may execute concurrently and can communicate with each other through channels or restricted shared memory. The simplest process in occam is an action which may be an assignment, an input or an output operation; processes can be organized hierarchically into higher level processes using so called *construction primitives*. Occam supports the following type of constructions:

- SEQ: builds a sequential process where component processes shall be executed sequentially in the order listed by the construction
- IF: builds a combination of processes where each process is guarded by one conditional, evaluated in sequence; the process associated with the first true conditional will be executed (there may be any number of conditionals)
- CASE: acts similarly to traditional CASE constructs, the process corresponding to the first matching conditional is executed
- WHILE: repeats a process while the associated Boolean expression is true
- PAR: executes the component processes in parallel with each other
- ALT: just one of component guarded processes will be executed; if two or more guards are true, the language run-time will select one arbitrarily or according to a priority hierarchy that may be optionally specified; a guard can be a combination of Boolean expressions and message reception

As mentioned above, occam allows shared memory between processes, but in a restricted manner. Basically, sharing is read-only: if a variable or array element is written in one process (e.g. through assignment of message reception), it cannot be assigned or used in any expression in any other process executing in parallel. For arrays, occam provides a mechanism for partitioning so that different partitions of the same array can be accessed freely in different processes.

Communication between parallel processes is through messages passed along *communication channels*. A channel is a unidirectional, un-buffered, named and typed point to point communication mechanism between two parallel processes (thus, for bi-directional communication, two channels are needed). The type of messages that can be sent and received over a particular channel are defined in the type of the channel (called protocol) and may be just a simple type, arrays or composite structures.

Today occam is largely relegated to university research and it's currently maintained by the Programming Languages and Systems group at the University of Kent, UK. Its main legacy is that it was the first language to show how the concepts of CSP can be successfully implemented in practice, which was the baseline for later realizations, such as Google's Go language.

8.2.1.2 The Go Language

The Go language [7] was developed at Google and released as open source in 2009. It is a Java/C++ like imperative language and it relies heavily on the concepts pioneered by the CSP model.

The concurrency model of Go relies on two concepts: *goroutines* and *communication channels*. A goroutine is a process spawn off by another process that may communicate with other processes through communication channels. A communication channel is a mechanism that can be used to transfer data between different processes in a half synchronous manner: the receiver will block on a receive statement until there is something to receive on that particular channel. Channels are first class data types in Go and thus may be passed around and even sent over another channel.

While shared memory is supported in Go, messages sent over channels are the primary synchronization mechanism between goroutines. The authors of the language use the principle "*don't communicate by sharing memory; share memory by communicating*"; in practice it means explicit hand-off of data between goroutines through passing around references to data that needs to be accessed. In fact, Go implements also many of the specifics of the Actor model and task based model that we will describe later in this chapter, but the overwhelming use of named channels as the primary mechanism for synchronization puts Go closer to the CSP model.

The Go language development is still work in progress with many features such as atomicity not yet fully defined. There are indications however that its usage is taking off, primarily within Google, its originating company.

8.3 Communicating Parallel Processes and the Actor Model

The Actor model was first proposed in 1973 by Hewitt, Bishop and Steiger [9] and was intended for usage in Artificial Intelligence research. It was motivated as a model for systems with thousands of independent processors, each with local memory and communicating via a high performance network, a model that has become again relevant in the context of many-core processors.

The basic *structuring principle* of the Actor model is centered on the concept of Actor. An actor is an entity that

- interacts with its surrounding *through messages*: it performs a certain computation based on its internal state and the message received from outside (other ac-

tors) and, as a response, may generate (finite number of) new messages sent to other actors
- can create a finite number of *new actors*

The *principle of interaction* within the Actor model is based on asynchronous, unordered, fully distributed, address-based messaging. An actor may send messages only to actors it already knows about, but there's no built in guarantee for preserving the order of messages or for providing implicit feedback on message delivery. While the original Actor model does not provide for buffering of messages, in practice such buffering is a must in order to support the asynchronous model of communication.

Inherently, there's no upper bound on when the system will settle on a stable state (all messages have been delivered and processed and no messages are pending)—the Actor model only guarantees that all messages will eventually be serviced. It's theoretically proven that the Actor model provides an implementation for *unbounded nondeterminism* (as opposed to the CSP's *bounded nondeterminism,* where it can be guaranteed that the system will finish in one of a finite set of states).

There are several fundamental differences between the Actor model and the CSP model. The original CSP model is based on parallel composition of a fixed set of sequential processes and synchronous communication between these (a message will be sent only if the receiver is ready to receive it), based on pre-defined channels. By contrast, in the Actor model actors (processes) can be created dynamically, any actor can communicate with any other actor whose address is available to it (received through e.g. a message) and communication is asynchronous. While a CSP system can be characterized by a global state, this is usually not the case for Actor model based systems: it's fully dynamic, distributed and without any central entity that can provide the full view of the system. In practice, a CSP-based application can be implemented using actors (by restricting e.g. actor creation, introducing actors to model synchronous communication etc); hence the CSP model is in fact a simplified, restricted incarnation of the Actor model.

Figure 8.1 gives a comparative example of two systems, one based on the CSP model and one based on the Actor model.

8.3.1 Definitions and Axioms of the Actor Model

In a paper published in 1977 [10], Hewitt and Baker laid down some basic definitions and axioms (also called laws, but without a formal proof) for the actor model. For a better understanding of the philosophy behind this model as well as the terminology associated with it, it's worth revisiting these axioms as a clarification of some of the underlying behavioral aspects of the model.

One of the fundamental aspects of the Actor model is the emphasis on local state, time and name space instead of global equivalents, found in other models. There's

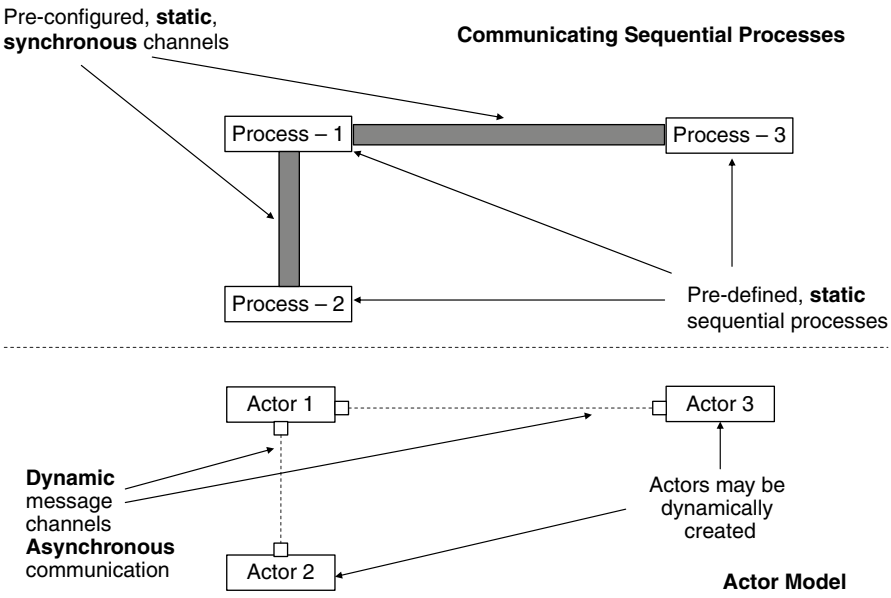


Fig. 8.1 Comparative overview of CSP and Actor model

no central entity managing the whole system—it's fully distributed without any notion of a central scheduler. In fact, a computation is not a sequence of global states, but a partially ordered set of events where each event is a transition from one *local state* to another and unordered events may proceed in parallel, creating multiple new local states asynchronously (an event is defined as receipt of a message by a target actor).

Actor computations are *constructed inductively*, by adding events in discrete steps and each event has a well defined set of immediate predecessors and a finite set of well defined set of immediate successors—except for the initial event which sets the actor system in motion.

The flow of events in an actor system form an *activation suborder tree* expressing for every event a causality order with a finite path back to the initial event; different branches of the tree represent parallel event chains.

The Actor model postulates a *total ordering of the events* that have the same target. In practice this means that every actor will have an ordered list of received but not yet processed messages. The rules of ordering are however not defined—it's totally up to an arbiter function associated with every actor (which acts a message queue manager).

The Actor model enforces *strict locality*: every information exchange occurs exclusively through messages which may contain information on other actors as well. In fact an actor A can get knowledge about the existence of another actor B if it has created actor B or has received the name of that actor from a third actor (which may

be actor *B*). At every point in its timeline, an actor has a local list of acquaintances: actors it has created or received information about. Initially this list contains only the actor that created this actor.

The most common types of messages in an actor system are requests and replies to requests; in this context an *activity* is a set of events that take place between the reception of a request and the sending of the reply. Two activities are considered *concurrent* if their request events occur in parallel, even if the activities may have some overlap (e.g. activate the same actor, the Actor model's equivalent of shared resource contention). An activity corresponding to a request is considered *determinate* if exactly one reply is generated (and non-determinate otherwise). The concept of activity also allows grouping of actors in layers: an actor that simply generates a reply to a request without activating any other actors is defined as *primitive* and belongs to the lowest layer of the system; all other actors are composite and are placed in higher layers. Obviously, the same actor may be primitive within one activity and composite in others—but this classification offers a basis for reasoning about the structure of interaction within an actor-based system. For example, it can help determine if the system follows the principles of proper layering, where actors in a given layer only activates actors in lower layers.

One important aspect that is *not* considered part of the Actor model is the reliability of message delivery. Hewitt et al. considered this an orthogonal issue with impact on how the behavior of the actors is implemented: in an unreliable network environment the actors (or some specifically designed actors) need to take care that proper re-transmission and fault handling protocols are in place.

8.3.2 Practical Realizations of the Actor Model

Besides direct implementations in various languages, the Actor model is at the basis of many derivative programming models. For example, the pipeline model or flow based model [11] can be viewed as a restricted instantiation of the Actor model.

In object oriented programming, the concept of *active object* [12] has been proposed as the mechanism and pattern to model an actor's semantics. An active object is an object that executes in its own thread of control, can receive calls of its methods asynchronously and notify its caller, upon completion of the request, through a callback function or a shared memory area where it will write its result.

The Actor model is also widely used in event-driven soft real-time systems—such as telecommunications—where, coupled with message and/or actor priorities and "run to completion" semantics, has proven to be a successful model. Most real-time operating systems implement some form of the Actor model, augmented with priorities (OSE [13] and Qnx [14] are some of the better known examples) for managing parallel computations (threads or processes). The Actor model is also at the basis of the Real-time Object Oriented Modeling (ROOM) methodology [15], today commercialized by IBM as part of their Rational Rose Real-time product suite and used in large scale systems at e.g. Ericsson or NASA.

On the language side, some well known languages draw inspiration or implement directly the Actor model, such as Erlang, Scala or Haskell; more recently Microsoft also released a new language based on the Actor model called Axum. In this chapter, we will briefly introduce the concurrency support found in some of these languages.

8.3.2.1 Erlang

Erlang [16] was developed in the later part of the 1980s at Ericsson's Computer Science Laboratory, as a method for programming software for telecommunication equipment. According to its main developer, Joe Armstrong [17], the main inspiration came from Prolog, augmented with parallel processes. After being released to the open source community in 1998, Erlang slowly gained acceptance in the programming community, a trend accelerated with the emergence of multi-core processors and the adaptation of the Erlang run-time system, OTP (Open Telecom Platform) to multi-core processors.

There are two main characteristics of Erlang that stand out at first sight: it's a *functional programming language* and it has built in constructs for *process-based parallelism* and *fault tolerance.* Though it may sometimes be confusing, these two characteristics are actually orthogonal to each other as the parallelism constructs may very well be implemented in an imperative language as well. In fact, the functional nature of Erlang simply reflects the preference of its developers and any attempt to link it to the powerful support for parallelism is based on false assumptions. There are however very compelling reasons for designing Erlang as a language with its own runtime system: many of the key features of Erlang (such as lightweight processes or fault tolerance mechanisms) were difficult to map to existing languages or operating systems—so faced with the choice between designing a new language or a new operating system, the original design team choose the first option (as a matter of fact, Ericsson later pursued the other track as well).

There are a few basic principles that underpin Erlang's design, collectively defining what Armstrong calls *Concurrency Oriented Programming (COP)* [17]. These are:

- The basic element for building programs is the concept of *process.* Any program is built as a collection of interacting processes. Each process is uniquely identifiable and addressable, but processes are strongly isolated and may implicitly share no information or state. The run-time system of Erlang implements a lightweight mechanism for quickly creating and destroying processes, an essential feature for a language relying on processes as a basic building block. It's important to highlight that an Erlang process is *not equivalent* to an operating system process; it's rather a language construct to model concurrency and isolation.
- The sole way of interaction between processes is through *asynchronous messages*. The delivery of messages is not guaranteed, irrespective if the destination of the process is running on the same physical machine as the originator or not (sometimes called a *send and pray* approach). Processes may choose to process the messages they receive in any order.

Based on these two principles, an Erlang process is essentially equivalent to an actor in the Actor model, turning Erlang into one of the cleanest implementation of the Actor model.

- *Fault isolation:* no failure in any process shall impact any other process, unless this is a conscious decision by the programmer. Hence, processes shall be able to detect the failure in any other process, including the reason for failure. This principle of fault isolation provides the foundation for Erlang's *let it fail* approach: each process shall have a supervisor process monitoring its behavior and, in case of encountering a fault, it shall simply fail and rely on its supervisor process to take the necessary recovery actions. This philosophy proved to be a powerful one, leading to much simpler, more compact and more resilient programs.

As implied by the principles above, Erlang naturally supports distribution across multiple cores, processors and physical machines, as long as there is an underlying communication mechanism between these. The unreliability of message delivery is justified exactly by this support for distributed systems (and reflects the origins of Erlang as a language for programming telecommunication systems).

It is also important to note that the "share nothing" semantics of the language does not imply a similar implementation in the run-time system; in fact, message passing between Erlang processes executing on the same processor is indeed done without explicit copying—but this is strictly an implementation feature of the run-time system and does not violate the principle itself.

As Erlang supports virtually thousands of simultaneous processes distributed across multiple processors, it provides a powerful environment for programming many-core processors. However, the exclusive reliance on process-based parallelism is also one of the limiting factors for some problem domains: Erlang does not support implicitly a de-coupling between application specific parallelism and available parallelism in the underlying hardware, as for example in task based models. Processes and their cardinality must be identified during design time and further partitioning—in order to cater for an increased number of available processor cores—requires a redesign of the application.

8.3.2.2 Haskell

Haskell [18] is the result of several decades of research into functional languages. It is a purely functional language with powerful constructs for expressing and containing side effects as well as concurrency. While it's a standardized language, the open source compiler for Haskell—Glasgow Haskell Compiler [19]—is defining the de facto standard version of the language.

Haskell incorporates today multiple flavors of concurrency, implemented as add-on packages; however, the basic model relies on lightweight threads that communicate either through special synchronization variables (called *MVar*) or through messages over named channels, in CSP style. The concept of *synchronizing variable—Mvar*—provides a single-element message box semantic: such a variable may either be empty or may contain a single value; if it's not empty, any attempt to put a new

value into it will trigger a suspension of the thread attempting it. When this mechanism is not sufficient, named communication channels may be used, quite similarly to the communication channels of the Go language.

The Haskell community also produced two interesting packages worth mentioning in this context. The first one is *Data Parallel Haskell,* adding support for parallel arrays and operations on these arrays in UMA (Uniform Memory Access) systems. The second package supports *Software Transactional Memory (STM),* a mechanism we described in Chap. 5. It's also worth mentioning that the basic concurrency mechanisms were expanded—through new libraries—to support CSP-style concurrency (by the group that developed Occam) as well as the Actor model in a more explicit way. These are not part of the 'standard' Haskell release, but provide powerful mechanisms for writing CSP- or Actor-style programs in a functional language.

In our opinion, Haskell is a good candidate for systems where productivity is the primary concern and a functional style of programming is acceptable. In our view, Haskell offers the most complete set of basic building blocks for writing parallel programs using various styles of concurrency—but at the cost of somewhat lower performance and the usage of a purely functional style of programming.

8.3.2.3 Scala

Scala (short for **Sca**lable **La**nguage) [20] was developed from 2001 onwards at EPFL in Switzerland. It has a unique blend of object oriented and functional style programming and it's primary goal was to provide a language environment that's suitable for both small scale and large scale systems, as well as one that's easily extendable—a language that can act as a workbench for creating new, domain specific languages. Recently Scala was embraced by researchers at Stanford [21] who aim at creating domain specific languages that allow automatic extraction of parallelism during compile time.

Scala is fully compatible with Java—it is compiled to Java byte code and is executed within a JVM (support for the .NET Common Language Run-time (CLR) platform was also implemented). The design philosophy of Scala is however to focus on abstraction and composition capabilities that enable building just about any programming flavor through libraries rather than basic language constructs. It has a uniform object model, support for pattern matching on objects and support for higher order functions; each function is considered a value and each value is modeled as an object; functions are implemented as objects with an apply method. A further example of the blend of object oriented and functional paradigm is the support for both parameterization of data types and for abstract members, expanding the possibilities for re-use; Scala even allows operations on parameterized types without instantiating the parameter—as long as the computation is consistent.

The support for concurrency in Scala is very similar to the Erlang style actor based model (in fact, the Scala community readily admits copying Erlang's solution). However, the Actor model is not part of the basic language in Scala; true to its principles, it's implemented as a bolt-on library. In its basic form, Scala's actors are implemented as sub-classes of the abstract *Actor* class, with behavior provided

through overriding the *Actor.act* method; even the send operator is the same as Erlang's default "*!*" send operator. Scala supports Erlang-style selective receive as well: all messages not matching any of the case statements of the receive block are kept in the mailbox until a matching receive is executed. As a small—but useful—extension, Scala supports replies to messages; instead of extracting the sender and sending an explicit message, all incoming messages support the *reply* method for sending a quick answer back to the sender.

On top of the basic support however, Scala adds a number of additional features. The first one is support for synchronous messages: using the "*?!*" operator, the sender is blocked until a *reply* is received on the sent message, basically implementing the semantics of a blocking remote procedure call.

By default, Scala actors are mapped to Java threads and are scheduled by the JVM; these actors are called *thread based actors*. This approach however highlights the very reasons why Erlang has its own process management mechanisms: operating system or virtual machine level threads are just too heavy-weight (in terms of memory and computational overhead) and will quickly become a bottleneck in case a large number of actors are used. To mitigate this issue, Scala also supports *event based* or *reactive actors*: these actors are not run on their own thread, but rather are multiplexed over the same thread; whenever an actor blocks waiting on a message, the Scala run-time system will create a closure that will be activated once a message is available. Syntactically, reactive actors are described using the statement *react* (hence the name) instead of normal *receive.*

The code snippets below show how *receive* and *react* constructs would look like in Scala. Using *receive*:

```
while (true) {
        receive {
           case Deposit(x) =>
              account += x
           case saldo =>
              println("Saldo is "+account)
              exit()
        }
}
```

Using *react:*

```
while (true) {
    react {
      case (x: String, actor: Actor) =>
        actor ! show(x)
    }
  }
```

Scheduling of actors in Scala relies on dynamically sized thread pools: the run-time system will adjust the amount of threads allocated for actor processing based on the amount and type of actors: with event based actors, the thread pool size will stay constant; with thread-based actors it will expand or shrink based on how many actors are concurrently active (processing messages).

In summary, Scala is an interesting language that has seen some significant take-up recently, both in industry (Scala's web page reports production usage by a fair amount of companies, both small and large) and in academia (primarily as a language creation workbench). While the basic support for concurrency is fairly straightforward, the possibilities offered by Scala for extending the semantics of the language and creating new, more restricted languages make it an attractive basis for novel research into using domain specific languages to create an environment where automatic mapping of functionality to parallel hardware can be achieved.

8.4 Task-Based Programming Models

The origins of the task based programming model are much more murkier than those of CSP or the Actor model; however in our view some of the fundamental research results were published in 1991 by Erich Mohr, David A. Krantz and Robert H. Halstead Jr. in their paper about lazy task creation [22] and in 1998 by Matteo Frigo, Charles E. Leiserson and Keith H. Randall, reporting there work on the Cilk extension to the C language [23].

The core idea of the task based model was expressed in Ref. [22]: the programmer's task is to *identify* parallelism (*what can be* computed safely in parallel), while the run-time system takes care of *how* the exposed parallelism can best be exploited—usually *limiting* the parallelism to a manageable level that can be efficiently executed on the available hardware. The obvious way to do this is to provide the programmer with simple means to expose computations (called tasks)—no matter how simple—that may be executed in parallel and the necessary synchronization mechanisms that can guarantee that the task execution will yield correct results. Everything else—mapping to available hardware resources, load balancing, scheduling in general—is handled by the run-time system and is transparent to the programmer.

Cilk provides an elegant implementation of this concept, using just three keywords: *cilk, spawn* and *synch.* The keyword *cilk* is used to mark functions that may generate tasks (a marking useful for the compiler); *spawn* creates tasks that may be executed in parallel with their creator task; *synch* simply waits until all the tasks spawned by the current task finish their job. These concepts are best illustrated with the un-optimized implementation of a function that calculates the *n*th Fibonacci number:

```
cilk int fib (int n)
{
      if (n<2) return n;
      else {
                int x, y;
                x = spawn fib (n-1);   // x may be calculated in parallel
                y = spawn fib (n-2);   // y may be calculated in parallel
                sync;   // wait until both tasks spawned above complete
                };
return (x+y) ;
}
```

The strength of this model is that removing the three keywords—*cilk, spawn* and *sync*—results in an equivalent sequential program, allowing an easy fallback to sequential execution. As an extension to the C language, the Cilk model—as most of the task based parallel programming models—is implicitly supporting shared memory, but requires the programmer to take care of access to shared resources; this inherently means that generic task based models suffer from the same issues as any shared memory model based programming models.

The key to the performance of task based programming models is the underlying *scheduling policy.* Most task based model implementations today use a variant of the *work stealing method,* which is based on three simple principles:

1. All tasks generated by tasks under execution on a specific core are placed in the core local task queue of the processor core where the originating task was executing
2. All processor cores will execute tasks from their local queue as long as the queue is not empty
3. When the core local queue is empty (there are no tasks left to execute), the core will *steal* some tasks from another core; the choice of the victim core may be based on various policies, but most often a random selection is used

This approach has the benefit of minimizing contention on task queues: as long as each core has some work to do, there will not be any interaction between the run-time system instances running on each core (other than possible application level synchronization) and hence no synchronization bottlenecks will occur.

In Ref. [23] the authors established upper bounds for execution time, memory need and communication overhead for work stealing algorithms. Regarding execution time, on P processors, the upper bound is

$$T_1/P + O\,(T_\infty)$$

where T_1 is the minimum serial execution time (on one processor) and T_∞ is the minimum execution time with an infinite number of processors. Similar upper bounds were established for memory and communication; all are within a constant factor multiple of optimal values (this factor is 2 for execution time, hence work stealing guarantees execution times within 2× of optimal).

Task based models (scheduled using a variant of the work stealing algorithm) provide an elegant de-coupling of application level parallelism from the parallelism available in the underlying hardware. These models also map naturally to space shared operating systems introduced in Chap. 7: an application, provided with P cores, can create one single thread on each core that will execute tasks generated by the application, based on work-stealing, hence the task-scheduler becomes the application level operating system library.

As we already touched upon, task based models implicitly support shared memory based systems or rather are orthogonal to the use (or not) of shared memory (task based parallelism implemented in the context of a functional language is a good example of task based parallelism without shared memory). In the context of shared memory systems, most languages and run-time environments supporting the task

based model, used in large scale computer systems rely on the *Partitioned Global Address Space (PGAS)* mechanisms that we introduced in Chap. 5.

8.4.1 Practical Realizations of the Task Based Model

The task based model is at the foundation of several well established libraries and languages. We will deal with the libraries in Chap. 9; here we will focus on the languages relying on the task based model for their parallelism constructs, namely X10 and Chapel. Obviously, these are not the only ones supporting the paradigm; however we believe these are the ones that were purposefully built around the concept and hence most suitable for practical usage.

8.4.1.1 X10

X10 [24] is a language developed by IBM, as part of a DARPA-financed program aimed at creating high productivity programming environments for the US' national security and industry (the other language coming out of the same effort is Chapel, covered in the next sub-chapter). X10 is targeted specifically to parallel computing, primarily in the context of high-performance systems. It is an object oriented language similar to C++ and Java, compiled to Java byte code and run within a Java Virtual Machine. In addition, X10 supports transformation to various backend systems, including C++.

X10 is heavily relying on PGAS. The logical model of the target environment is made up of a number of *places,* an abstract model of a collection of locally resident objects and *activities* (equivalent to the concept of task). In practice a place is likely to be a processor core or complete processor with own share of memory.

Any object may be referenced from any place, but any read/write access to the object has to execute at the *home place* of the object, requiring that any activity doing such access must shift its execution to the specific place where the object resides. The X10 run-time is globally asynchronous: there is no ordering guarantee for activities between places; however, locally within a place, synchronous behavior is guaranteed. The concepts of *places, activities* and *object location* are shown in Fig. 8.2.

The model of concurrency in X10 relies—beside the concept of place—on activities, i.e. tasks. Concurrency is expressed basically through six constructs:

- *asynch:* creates a new child activity that cannot be aborted or canceled and is executed in parallel with the parent activity; essentially it's equivalent with cilk's *spawn* statement. The new activity may reference any of the enclosing block's objects
- *finish:* the statement marked with the keyword is executed, but the system will not continue beyond the scope of the statement before all transitively spawned

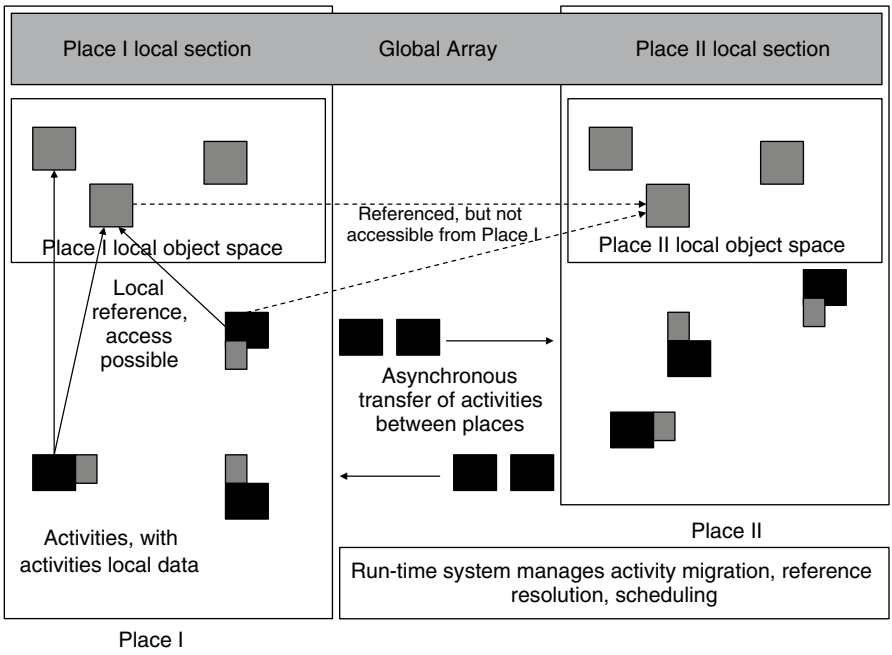


Fig. 8.2 X10 concurrency concepts

activities (using *asynch*) have terminated; it will also aggregate all exceptions potentially thrown by spawned activities

- *atomic:* the statement marked with *atomic* is executed without interruption, introducing this way a sequential ordering of activities; the body of the *atomic* statement must be *non-blocking, sequential* (may not create new activities) and *local* (may not access objects located at other places)
- *when:* the current activity is suspended until the Boolean expression associated with it becomes true
- *at:* specifies explicitly at which place the statement associated with it shall be executed—it is primarily used to access objects located at some other places; the current activity will block until the statement is executed at the remote place
  - *Note:* in many ways, this is an implementation of the Follow the Data pattern that we discussed in Chap. 5, with explicit movement of the computation
- *clocks:* X10 supports step-wise synchronization of multiple activities using a concept called *Clock* (multi-stop barrier) and the construct *next*. Each participating activity will execute one phase of its execution then call *next* to synchronize with all other activities on the same clock; once all activities reached the same phase, all of them are allowed to proceed—concurrently—with the next phase of the execution.

The code snippet below illustrates some of these concepts, based on examples from the X10 reference manual [25]:

```
finish async{
        val clockAB = Clock.make();
        async clocked(clockAB) {
                say("A step 1");
                next;
        say("A step 2");
        next;
        say("A step 3");
        }
async clocked(clockAB) {
        say("B step 1");
        next;
        say("B step 2");
        next;
        say("B step 3");
        }
}
```

In addition to these constructs, X10 also supports mechanisms for data parallelism, through the concept of distributed arrays, where parts of an array (called *regions*) are allocated to different places, in order to facilitate parallel execution of some algorithms and better management of locality for large data sets.

X10 is still very much a work in progress, but gaining momentum especially within the academic community. Its simple syntax, similar to C++ and Java, as well as the powerful yet simple concurrency support make it an attractive choice for designing software for many-core systems. However, as most imperative languages, it requires that programmers identify and express parallelism.

8.4.1.2 Chapel

In many ways, Chapel [26], developed by Cray in co-operation with academic partners, is the sister language of X10, developed with the same goal and DARPA funding: making programming large parallel systems easier. Consequently, the basic concurrency model in Chapel is based on tasks and a PGAS model very similar to X10.

Concurrency, synchronization and distribution are supported in Chapel through the following constructs and mechanisms:

- *begin, cobegin* and *coforall* statements are used to start a single new task (*begin*), multiple tasks (*cobegin*) or a task for each iteration of a loop (*coforall*)
- *sync:* all tasks started within the scope of this statement must be finished before continuing to the statement after *sync*

- *sync variables* are special variables that, beside value, can store a full/empty state, allowing multiple tasks to synchronize with each other on writes and reads to and from variables
- *atomic* sections are supported with the same basic syntax and semantics as in X10

PGAS implementation in Chapel relies on the concept of *locale,* the equivalent of *place* in X10. The concept of locale is however in many ways more powerful than place in X10: tasks may be run on a locale explicitly described (using the statement *on,* similar to *at* in X10), but allocation may be based on the locale of a data set as well.

Chapel also has powerful mechanisms for data parallelism, through the concepts of domains and distributions. An array may be partitioned into multiple domains and domains may be mapped to distributions that specify locality of data and implicitly, locality of data access (where computations accessing the data will execute).

The example below illustrates some of the concepts of Chapel through a simple tree traversal algorithm (taken from the Chapel tutorial material):

```
def search(N: TreeNode, depth = 0) {
        if (N != nil) then
        serial (depth > 4) do cobegin {
                search(N.left,  depth+1);
                search(N.right, depth+1);
        }
}
search (root);
```

Despite the similarities and in many ways competing solutions, Chapel seems to be more geared towards high performance computing, while X10 is apparently aiming at a broader audience, a claim backed up by their choices of syntactic sugar and backend support. How and where these two promising languages will eventually find their niche will depend a lot on the strength and nature of communities building up around the two solutions, as well as the long term commitment of their parent companies, IBM and Cray respectively.

8.5 Process Versus Task-Based Parallelism and the Usage of Shared Memory

In this chapter we introduced three parallel programming models which we believe are the most widely used and also most promising ones in the context of programming chips with large number of cores.

There are several important differences between these models that are important to understand when making a choice for the design of an application. The most important issues revolve around the nature of the parallelism that can be extracted

from the application and the need (or desire) to use any sharing of data between the tasks that need to execute in parallel.

CSP and the Actor model are most suitable for applications where parallel tasks tend to have a longer lifespan and are relatively independent of each other—i.e., can communicate solely based on messages. Consider the case of a telecom node, for which Erlang was originally developed: each call—lasting from several seconds to many minutes or even hours—can be modeled as a process that has a fairly long execution time and only interacts sparsely with other processes (likely some sort of database and possibly related calls). For such a case, the Actor model is a natural choice: the system is distributed and loosely coupled, thus a model based on share nothing and loose interaction is the best bet. CSP and the Actor model also tend to suit applications with a slower pace in terms of generating parallel workloads as well as applications where the nature of the hardware—the amount of parallelism available—is fairly well known.

Task based parallelism on the other hand emphasizes quick dispatch of usually short tasks that may interact with each other through shared memory. As we mentioned in the previous sub-chapter, the basic philosophy behind the task based model is to expose as much parallelism as possible from the application point of view and then let the hardware limit it if necessary. Task based parallelism is more opportunistic as well and hence its performance may be less predictable: creating too much parallelism may add unwanted overhead that could adversely impact performance. On the other hand, task based implementations are less coupled with actual hardware and my scale better as more processor cores are added. These conflicting forces must be carefully judged when deciding on how much parallelism shall actually be exposed through tasks.

Regarding usage of shared memory, it's our firm belief that large scale usage will eventually lead to bottlenecks and hence it shall be limited. However, there are cases when shared memory is simply the most convenient approach with limited performance impact. To use a real life analogy, consider the social network of a person, usually consisting of at least two layers. First, it's his or her immediate family with whom there's a frequent interaction, sharing of and (friendly) competition for resources (such as the bathroom); then there's the second layer of friends with whom messaging based interaction (e-mails, Facebook updates, phone calls) may be just fine. The first layer corresponds to a shared memory model in programming terms, where sharing may be inevitable, but it's restricted to a small amount of competing tasks (persons); the second layer represents a model of fairly large amount of parallel entities with sparse, message-based interaction. For the first case, the task model with memory sharing may be the most suitable one; the second case is a clear candidate for a CSP/Actor model based approach.

Finally, as we also touched upon, there's a potential for layering the two models (process and task based) of parallelism. The threading model can be used to provide an abstract model of the underlying HW, while the task model exposes the amount of parallelism from the application. The modeling of the HW can be extremely simple: just allocate one thread (or lightweight process) to each processor core or HW thread to model workers waiting to execute tasks that are delivered by previous

tasks executed by some of these worker threads. This model suits perfectly a many-core system with a large pool of simple cores; it can also incorporate heterogeneous systems comprising high capability cores as well by just modeling those as multiple threads. The interface between these two layers—HW modeled as threads and applications modeled as graphs of tasks—is obviously the scheduling policy. How tasks are managed (queuing)? How are constraints (e.g. timing) taken into account? How is load balancing—or power management—handled? We believe these are the fundamental issues that will need to be addressed and will serve as the foundation of future many-core programming models.

8.6 Summary

In this chapter we introduced the theoretical background of the most promising models for programming many-core processors: communicating sequential processes, the Actor model and the task based model. We also looked at the programming languages implementing these models directly, as a basis for the more in-depth coverage in the next chapter. Finally, we compared these models with a focus on where the usage of each of these makes more sense.

In the next chapter we'll cover the most important libraries implementing these models as well as the practical usage of some of the languages introduced in this chapter.

References

1. Turing A M (1936) On Computable Numbers, with an Application to the Entscheidungsproblem. Proceedings of the London Mathematical Society 42:230-265
2. Agha G (1986) Actors: A Model of Concurrent Computation in Distributed Systems. Doctoral Dissertation, MIT Press
3. Hoare C A R (1978) Communicating Sequential Processes. Communications of the ACM 21(8):666-677.
4. Dijkstra E W (1975) Guarded Commands, non-determinacy and formal derivation of programs. Communications of the ACM 18(8):453-457
5. Ericsson-Zenith S (1988) occam 2 Reference Manual. Prentice Hall
6. Barron I M, Aspinall D (1978) The Transputer. The Microprocessor and its Application: an Advanced Course. Cambridge University Press
7. Google Inc (2011) A Tutorial for the Go Programming Language. http://golang.org/doc/go_tutorial.html. Accessed 11 January 2011
8. Barnes F, Welch P (2006) occam-pi: blending the best of CSP and pi-calculus. http://www.cs.kent.ac.uk/projects/ofa/kroc/. Accessed 11 January 2011
9. Hewitt C, Bishop P, Steiger R (1973) A universal modular ACTOR formalism for artificial intelligence. Proceedings of the 3rd International Joint Conference on Artificial Intelligence: 235-243
10. Baker H, Hewitt C (1977) Laws for Communicating Parallel Processes. MIT Artificial Intelligence Laboratory

11. Morrison J P (1994) Flow-based Programming: A New Approach to Application Development. van Nostrand Reinhold
12. Schmidt D, Stal M, Rohnert H, Buschmann F (2000) Pattern-Orineted Software Architecture Volume 2: Patterns for Concurrent and Networked Objects. John Wiley & Sons
13. Enea (2010) Enea OSE: Multicore Real-Time Operating System (RTOS). http://www.enea.com/Templates/Product____27035.aspx. Accessed 11 January 2011
14. Krten R (1998) Getting Started with QNX 4: A Guide for Realtime Programmers. Parse Software Devices
15. Selic B, Gullekson G, Ward P T (1994) Real-Time Object Oriented Modelling. John Wiley & Sons
16. Armstrong J (2007) Programming Erlang: Software for a Concurrent World. Pragmatic Bookshelf
17. Armstrong J (2003) Making Reliable Distributed Systems in the Presence of Software Errors. PhD thesis, Royal Institute of Technology, Stockholm, Sweden. http://www.erlang.org/download/armstrong_thesis_2003.pdf. Accessed 11 January 2011
18. O'Sullivan B, Goerzen J, Stewart D (2008) Real World Haskell. O'Reilly Media.
19. The Glasgow Haskell Compiler. http://www.haskell.org/ghc/. Accessed 11 January 2011
20. Odersky M, Spoon L, Venners B (2008) Programming in Scala: A Comprehensive Step-by-Step Guide. Artima Inc
21. Chafi H, Sujeeth A K, Brown K J, Lee H J, Atreya A R, Olukotun K (2011) A Domain-Specific Approach to Heterogeneous Parallelism. Proceedings of the 16th Annual Symposium on Principles and Practice of Parallel Programming
22. Mohr E, Kranz D A, Halstead R H Jr (1991) Lazy task creation: a technique for increasing the granularity of parallel programs. IEEE Transactions on Parallel and Distributed Systems 2(3):264-280
23. Frigo M, Leiserson C E, Randall K H (1998) The implementation of the Cilk-5 Multithreaded Language. Proceedings of the ACM SIGPLAN 1998 conference on Programming Language Design and Implementation, 212-223
24. Murthy PVR (2008) Parallel Computing with X10. Proceedings of the 1st International Workshop on Multicore Software Engineering: 5-6
25. Saraswat V, Bloom B, Peshansky I, Tardieu O, Grove D (2010) X10 Language Specification Version 2.1. http://dist.codehaus.org/x10/documentation/languagespec/x10-latest.pdf. Accessed 11 January 2011
26. Cray Inc (2010) The Chappel Parallel Programming Language. http://chapel.cray.com/. Accessed 11 January 2011

Chapter 9
Practical Many-Core Programming

Abstract In the previous chapters we introduced the foundations for programming many-core chips: current and expected hardware architectures; operating system designs; foundations of parallel programming; the basic programming models that we believe are the most promising ones in the context of many-core chips. This chapter focuses on the concrete technologies available today which we believe will endure the test of time, providing a solid background for programming the many-core chips of the future. Specifically, we cover several task based models (such as Cilk, Grand Central Dispatch, OpenMP, Thread Building Blocks, Microsoft's Task Parallel Library), data parallel models (illustrated through OpenCL which also supports the task model) and a well established representative of the actor model (Erlang). The goal of this chapter is to provide a guiding map for the choice of the most suitable programming model and implementation library when addressing the challenge of 'best solution' for a specific application domain.

9.1 Introduction

It should be pretty clear by now that, at least in the view of the authors of this book, there are essentially two basic models of parallelism that we believe will eventually prevail:

- *Task based parallelism:* The concept of exposing abundant parallelism independently of the underlying hardware is the most promising approach for supporting scale-up of applications with sufficient amount of parallelism. Unsurprisingly, just about every emerging parallel programming language or library supports some form of this model
- *Actor model:* The actor model's attractiveness comes from its distributed, loosely coupled approach, which bolds well for chips where co-ordination and synchronization is likely to be the primary source for bottlenecks

With Contribution by Diarmuid Corcoran

A. Vajda, *Programming Many-Core Chips,*
DOI 10.1007/978-1-4419-9739-5_9, © Springer Science+Business Media, LLC 2011

These two models are fairly low level and require the programmer to identify parallelism and expose it in a suitable form to the underlying run-time system and hardware. This is exactly the focus of the current chapter; we look at two other promising, yet higher level approaches (domain specific languages and semantic speculation) in the final chapter of this book.

9.2 Task-Based Parallelism

This chapter focuses on the most promising task-based libraries and solutions available today: Cilk and Thread Building Blocks (TBB) from Intel, Task Parallel Library (TPL) from Microsoft, Grand Central Dispatch (GCD) from Apple and the OpenMP v3 library. While these are not the only libraries and languages available, we believe that these are the most representative ones, not least due to the sizable community already using or supporting these technologies.

9.2.1 Cilk

Cilk [1] emerged as a spin-off from research done at MIT, which led to the design of the work stealing task scheduler. It was recently acquired by Intel and included into Intel's parallel programming offering, alongside the more established Thread Building Block library (that we will cover in the next sub-chapter).

Cilk is a multi-core programming model based on a simple extension to the C (and more recently, C++) programming language (through the introduction of a three new keywords), together with a sophisticated work-stealing scheduler. By design all Cilk programs preserve the semantics of their C equivalent, in the sense that when the specific Cilk keywords are removed from the program, it should be an exactly equivalent C program, minus the multi-task capabilities.

Let's look again at the pedagogical implementation of the calculation of the nth Fibonacci number (first introduced in Chap. 8, shown here in Fig. 9.1). It is the most commonly cited example in the Cilk research literature and we follow that tradition here, as it exposes the most important keyword additions, namely *spawn* and *sync*. The *spawn* keyword tells the Cilk run-time that the function may, but doesn't have to, run in parallel with its parent caller. The *sync* keyword is a barrier that indicates that control cannot pass this point until all spawned children have returned. This is about the extent of language features one needs to know in order to write useful Cilk programs. One of the characteristics of Cilk is that both the compiler and run-time, with supporting scheduler, bear the responsibility for efficient execution of Cilk programs. The Cilk scheduler's algorithmic design makes certain guarantees, which can be proven about the achievable efficiency of a program run on it.

Figure 9.1 also illustrates the concept of a Cilk thread. In effect a Cilk thread is the maximal sequence of statements that execute until the next *spawn* or *sync* state-

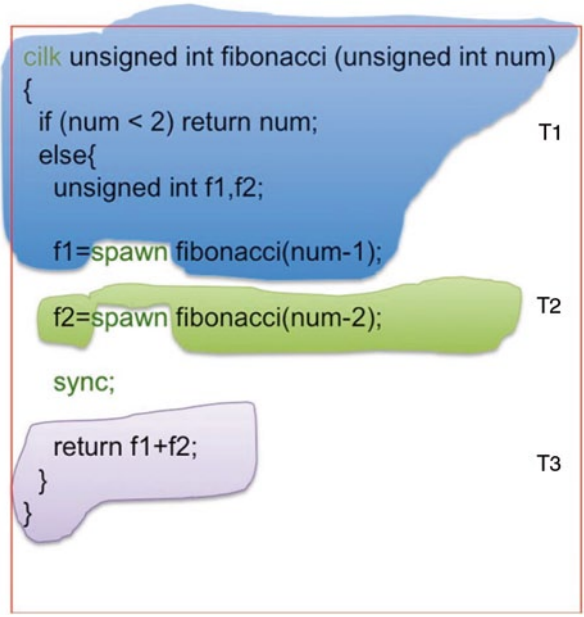


Fig. 9.1 Fibonacci numbers using Cilk

ment. The T1 strand runs in some initial worker thread. On reaching the first spawn statement, a new possible parallel thread of execution is created. The Cilk model of execution is such that the spawned thread continues to execute using its parent worker thread context. The parent or spawning thread is then placed on a queue of possible thread work items. If more then one processor is available and this processor has nothing to do then it will steal this work item; thus the parent has a continuation context in a new thread. The same happens when the parent context reaches the second spawn statement: the spawned thread continues using the parent thread context, then the parent thread is placed on the work list queue and if there is an idle processor available it will again steal from this work list.

It is interesting to note that the concept of a parent continuation being popped onto a queue of available work items is rather analogous to the semantics of the C equivalent program where the parent frame or context would be popped onto the stack while execution continues in a branched child function. The *sync* statements function as a barrier and guarantee that all spawned children have returned after the statement. This is important as the values of partial Fibonacci sums *f1* and *f2* in the example cannot be relied upon until after the sync statement.

Figure 9.2 shows yet another view of what happens at execution time. The figure captures what happens on execution of fibonacci(4). In effect a DAG (directed acyclic graph) of the program execution unfolds as the program executes. This DAG represents a graph of control and data flow and can be defined formally as $G=(V,E)$ where each vertex v is an element of the set of vertices V and each edge e is an element of the set of edges E. A vertex then represents a sequence of instructions that

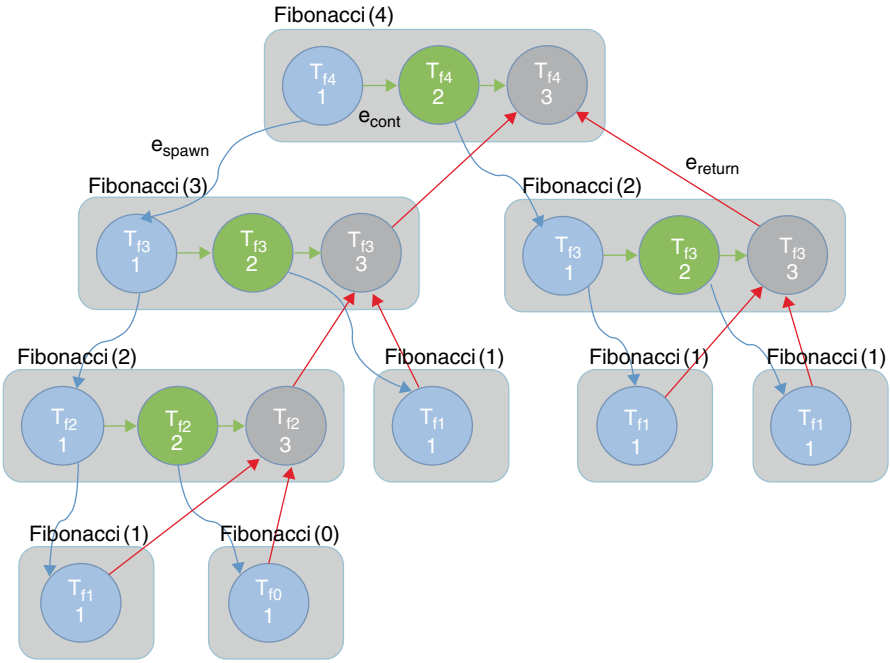


Fig. 9.2 The execution of fibonacci(4)

does not contain a Cilk language statement and an edge represents a Cilk *spawn*, a function *return* or a *continuation*.

In Fig. 9.2 the edge e_{spawn} represents a branch in the execution graph through the creation of a new Cilk thread. After the e_{spawn} branch the parent has a continuation edge e_{cont} that may result in continued parallel execution, through the process of work-stealing, in a free processor. The upward edges, shown by e_{return}, represent a flow of data back to the creating context. Finally execution completes in a final thread.

9.2.1.1 The Cilk Scheduler

The Cilk scheduler uses the concept of work-stealing in which a processor, often called a thief, who runs out of work steals work from the task queue of some victim processor, who has more tasks waiting to execute that it can currently service. The strategy in Cilk is to select the victim processor at random.

The theory behind the Cilk scheduler states that the performance of a Cilk computation is related to two quantities. The first is *work*, which is the cost of the computation when run serially on one processor. The second is termed *critical-path* length, which is the cost of the computation when run on a theoretically infinite number of processor. Figures 9.3 and 9.4 illustrate these concepts on the Fibonacci example using a value of 1 per task computation cost.

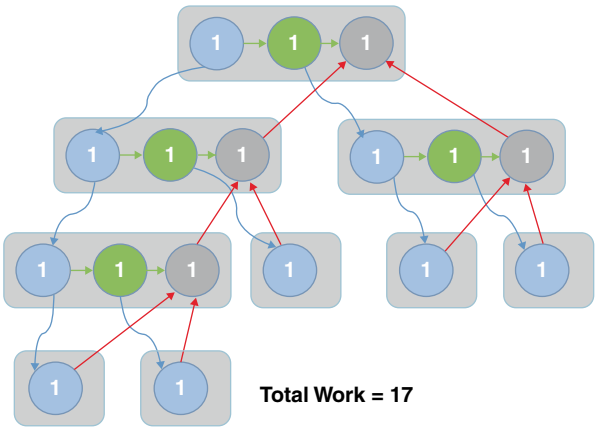


Fig. 9.3 Total work in case of fibonacci(4)

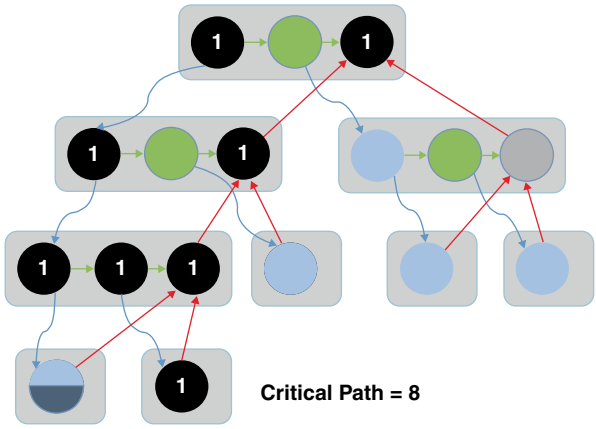


Fig. 9.4 Critical path for fibonacci(4)

The term T_1 is the same as the work, the time to execute on one processor. T_p is the time to execute on P processors. Then $T_p \geq T_1/P$ since P processors can do at most P work in one step.

Another important principle of the Cilk scheduler is the *work-first principle*. This principle states that the scheduling overhead should not be borne by the work of a computation but, rather, moved onto the critical path to the point where stealing is necessary. The Cilk scheduler is termed a "greedy scheduler" as it attempts to do as much work as possible at each step. There are basically two types of schedule steps. The first is called *complete:* in this step there are at least P tasks ready and P processing resources available. In this case the scheduler selects any P and runs them. The second is called *incomplete:* in this step there are less than P tasks available to run; the scheduler will then select all for execution on subset of processing resources. Stealing—and consequently, scheduling—will only occur when there is at least one processor out of work.

In their paper [1], the original designers of Cilk and its scheduling algorithm have shown that its performance is within a constant multiplier of optimal execution time, for execution time, required memory and communication. This theoretical underpinning—verified in practice as well—is the main reason for the enduring success of the work stealing scheduling method.

9.2.2 Grand Central Dispatch

Grand Central Dispatch (GCD) [2] is Apple's solution to the many-core programming problem. The source code for GCD has been released as open source under the Apache license version 2, in the hopes that it will spread as a general solution to many-core programming problems.

The programming model recognises the fact that building software on many-core environments using the traditional techniques of operating system threads, locks and semaphores is very difficult to get right. The mechanism builds upon using the *queue* as an abstraction to control access to resources, be they processors, or some other shared application or system level resource. As in the Cilk programming model, the approach builds upon extensions to the compiler called *blocks* as well as a scheduler which distributes tasks to processing resources through the GCD queues. In general, the programming model is very simple to understand if you are comfortable with using the concept of a queue to control access to some resource. This is a pattern which is widely used within the software world, especially within deeply embedded technical domains. Outside the software world, the queue is of course well know to all and sundry as a technique to regulate access to some limited resources, be it bank machine or a bartender (if we were to extend the concepts of locks and semaphores to our real world example it would mean passing around some kind of "token" in a crowded bar to gain access to the bartender—indeed without queues, our world would be a chaotic place).

9.2.2.1 Blocks

In order to understand how GCD works it is necessary to understand what *blocks* are. Blocks are a semantic and syntactic addition to the C and C++ (also Objective-C) languages to allow the concept of closures, found in many other languages as well. The block abstraction has been added to certain release branches of the *gcc* compiler, in addition to the clang compiler chain by Apple. Apple has also submitted the block additions to C and C++ for future standardisation into the languages; however, for now blocks should be considered an Apple OS X specific extension.

The syntax additions are very straight forward. A block is declared by using the ^ caret symbol and then surrounding the "code block" by curly braces. A simple example is shown below:

$$\text{int}(^{\wedge}\text{squareBlock})(\text{int}) = {}^{\wedge}(\text{int x})\{\text{return x} * \text{x}\}$$

This rather simple example misses some of the powerful advantages of blocks: blocks can refer to variables within the same lexical scope from within which they are created.

The next example below highlights this: the variable *solution* is copied from the *solveEverything()* function context. This variable now lives within the block context as a copy. It is important to note that the block context can outlive the context of its creating scope, in this case the function *solveEverything()*. Blocks can also be used to describe what are often termed *anonymous* functions in other languages. In general, a framework like GCD would traditionally have been implemented using call-back functions and indeed the API still allows for usage of these. Implementation issues are considerably simplified by the usage of blocks however.

```
void solveEverything()
{
    int solution = 42;
    int (hitchhiker)(int) = ^(int n) {return n*solution}
    solverEverythingQueue(hitchhiker)
}
```

9.2.2.2 Grand Central Dispatch Queues

Grand Central Dispatch relies on management of a number of queues. Some of these queues are pre-determined by the GCD run-time, while others are defined by the application and their meaning is up to each application design to determine. Jobs or tasks are dispatched to these queues and it is then up to the GCD run-time to utilize the underlying processor resources in best possible way by distributing the tasks to the processor pool.

The scheduler builds upon the thread pool pattern where a number of threads are fired up in advance of their use in order to build up a resource pool. A task on one of the dispatch queues that is ready to run is then paired up with a free resource from the thread pool. If there are many processing resources available then, depending on which queues the tasks have been dispatched on, these will be distributed in parallel across the processor pool.

The GCD programming model has three classifications of queues:

- Global asynchronous queues
- Main queue
- User defined serial queue

Each of these queues has differing behavioral properties, described in the following sub-chapters.

GCD queues can be arranged in a system of higher order queues giving arbitrarily complex directed acyclic graphs. Queue hierarchies can be used to funnel tasks from disparate subsystems into a narrower set of centrally controlled queues. GCD also supports suspending/resuming individual queues: no block on a suspended queue will be executed. These mechanisms are asynchronous and take place be-

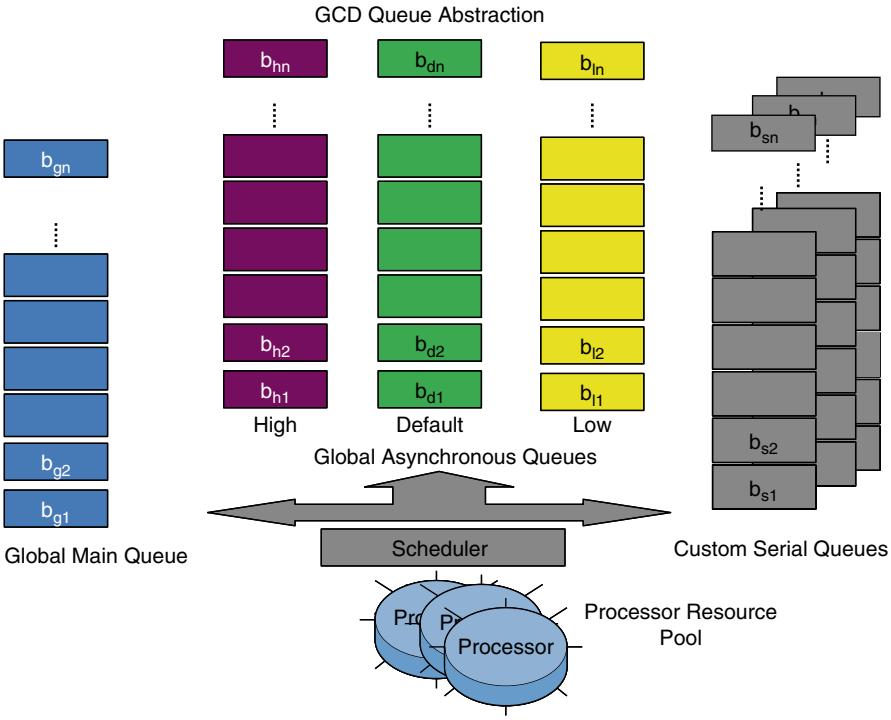


Fig. 9.5 GCD Queue architecture

tween blocks (i.e., the currently executing block will be completed before suspend takes effect).

Figure 9.5 shows an overview of the GCD architecture and its respective queues.

The *global asynchronous queues* are a set of three queues of different priority levels (1 is High, 2 is Default and 3 is Low). Tasks dispatched towards these queues will be removed from the queues in order of priority. Also these queues have the property that many tasks may be removed from the queues, in parallel, and dispatched to available processing resources with the prerequisite that the priority order is preserved.

An application can create any number of *custom serial queues*. Task dispatched to these queues will be executed serially per queue instance but if there are many serial queues instances jobs across these will be executed in parallel. Dispatching jobs to a serial queue instance implies some dependency either in terms of control or data. The serial queues can be used to control access to some global resources instead of a lock or other mutual exclusion mechanism.

The *process main queue* is in some ways analogous to the *main* program in C: it can be used as a global synchronization point. An important property of the main queue is that it is not drained automatically by the scheduler, but rather must be drained as part of the program or application design.

The example below shows a code snippet of how a block is added as a task to the default asynchronous queue. In this case the block is anonymous. The snippet also illustrates how a reference is found to the global default queue. The *dispatch_async()* call dispatches asynchronously, that is the dispatching context continues to run after the task is placed on the appropriate queue.

```
dispatch_queue_t aQueue = dispatch_get_global_queue
                              (DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
dispatch_async(aQueue , ^{SomeSignificantTask});
// …….continue……
```

It is also possible to do a synchronous dispatch to global queue. In this case the calling task waits until the dispatched task has completed before continuing. The next example shows how two new user serial queues are created using the *dispatch_queue_create()* call:

```
dispatch_queue_t aQueue1;
dispatch_queue_t aQueue2;
aQueue1 = dispatch_queue_create(“aSillyOldQueue1", …..);
aQueue2 = dispatch_queue_create(“aSillyOldQueue2", …..);
// …….
dispatch_async(aQueue1 , ^{SomeSignificantTask});
dispatch_async(aQueue1 , ^{SomeSignificantTask});
dispatch_async(aQueue2 , ^{SomeSignificantTask});
```

An application using blocks dispatched to queues might want to be notified when the block completes; the best way to achieve this is to let the block dispatch itself a special block—called *completion block*—to a queue supplied by the application. The code snippet below exemplifies this:

```
void async_block_with_completion(dispatch_queue_t queue, void (^block)(int))
{
  // queue is where the completion block shall be dispatched
  // block is the completion block itself
  dispatch_retain(queue);  // makes sure the queue does not disappear
  // Do the work on the default concurrent queue
  dispatch_async(
            dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0),
             ^{
                  // useful work comes here
                  // then dispatch the completion block:
                 dispatch_async(queue, ^{ block(avg);});
  // Release the user-provided queue when done
  dispatch_release(queue);
 });
}
```

GCD also supports a special version of the *dispatch* operation for parallelizing loops. It is essentially a C implementation of an iterator, but it generates asynchronous tasks (blocks), and waits for all to complete before returning. The code example below shows two loops, one implemented with a classic *for* and the second one using *dispatch_apply*, the parallel iterator provided by GCD:

```
for (i = 0; i < n; i++)
  // do some useful work, sequentially
dispatch_queue_t queue =
      dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
dispatch_apply(n, queue, ^(size_t i) {
  // do the same useful work, in parallel
});
```

9.2.2.3 Synchronization in GCD

GCD supports two methods of synchronization: *dispatch semaphores* and *waiting on a group of queued blocks*. We'll briefly describe and exemplify both in this section.

Dispatch semaphores are essentially equivalent to regular semaphores but are primarily implemented in the GCD library rather than in kernel, hence takes less time to pass an open semaphore; the only case when the kernel is invoked is when the semaphore has no available resources: in this case the thread trying to pass it must be suspended. Otherwise the regular methods are supported:

- *dispatch_semaphore_create:* create a dispatch semaphore with a pre-defined maximum number of resources
- *dispatch_semaphore_wait:* wait on the semaphore
- *dispatch_semaphore_signal:* signal the semaphore (the resource controlled by the semaphore is released)

GCD supports mechanisms to wait on the completion of multiple blocks. This is achieved through the use of *dispatch groups*. A dispatch group is a set of blocks dispatched to the same queue, with the property that the application may wait on the group to complete, i.e., wait until all tasks associated with the group are completed. The main primitives involved in this mechanism are:

- *dispatch_group_create:* creates a new dispatch group
- *dispatch_group_async:* dispatches a block on the specified queue and adds it to the group
- *dispatch_group_wait:* waits until all blocks in the group are completed
- *dispatch_release:* releases the group when it's no longer needed

This mechanism is quite similar to Cilk's *spawn/sync* pair, but it's more flexible: while *sync* will synchronize on all tasks spawned within the scope of its block (including nested ones), dispatch groups allow more flexible control (the programmer can pick and choose the tasks/blocks), but also require more manual, explicit specification by the programmer.

The example below shows a simple usage example for dispatch groups:

```
dispatch_queue_t queue = dispatch_get_global_queue(
                DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
dispatch_group_t group = dispatch_group_create();
// Add a task to the group
dispatch_group_async(group, queue, ^{
  // Some work
});
dispatch_group_async(group, queue, ^{
  // Some more work
});
// Continue execution
dispatch_group_wait(group, DISPATCH_TIME_FOREVER);
// Release the group
dispatch_release(group);
```

9.2.2.4 GCD Summary

GCD is an attractive task management library, more flexible than Cilk, but still building on the same foundations. As with Cilk, it requires modified language syntax which makes it less portable without compiler modification. Still, the real power of GCD lies in the possibility of creating multiple hierarchies of queues with different priorities, an interesting approach to managing complex systems that can be described through task graphs.

9.2.3 Intel Thread Building Blocks

It may be somewhat surprising, but Intel has lately invested massively into building up its software portfolio, both in terms of tools (compiler, profiler etc) and libraries. The acquisition of Cilk is one example; the continuous development of the Threading Building Blocks (TBB) library is another proof point.

TBB [3] is essentially a collection of various C/C++ library functions and templates that provide mechanisms for building software that can execute efficiently on multi-core hardware. TBB is not bound to Intel processors only; ports to other processors are also available. It is integrated with Microsoft's Visual Studio and Intel's own development tools; supported operating systems (by Intel) include Windows, Linux and MacOS.

The main components of TBB are:

- Support for *loop parallelism:* TBB supports various constructs such as *parallel_for* (for parallel version of *for* loops), *parallel_reduce* (for implementation of the Map-Reduce algorithm), *parallel_do* (parallel version of *while*), along

with support for affinity and cache-aware partitioning. TBB also has support for lambdas, part of the upcoming C++ standard.
- *Parallel containers:* TBB includes parallelism-safe templates for implementing hash tables, vectors and queues which may be accessed concurrently by multiple threads. These containers rely on either lock-free data structures or fine-grained locking in order to improve scalability and are a useful basis for building software which requires concurrent access to such shared structures
- *Synchronization mechanisms* and *atomic sections:* TBB has its own mutex implementation and support for defining atomic sections
- *Task-based programming support:* TBB has its own task management mechanism and scheduler implementation, quite similar to the one offered by Cilk

These components together offer a coherent and useful set of abstractions that contain all the ingredients needed to build parallel software according to various models (loop parallelism, shared memory based parallelism, task-based parallelism etc). We found the parallel containers especially useful; through their support across multiple operating systems, these are good candidates for cross-platform, scalable applications.

In this chapter we will focus primarily on the task based programming support.

9.2.3.1 TBB Task Model Overview

You may be forgiven for wondering what the exact difference between Cilk and TBB's task model really is: the fundamentals are the same; the only real difference is that TBB has a much more elaborate and flexible model. Supporting both has historical reasons: TBB was developed long before the Cilk technology was acquired by Intel.

Let's start with the conventional example of calculating Fibonacci numbers. The example below is loosely based on the one provided by Intel in the TBB tutorial [4], simplified a bit in order to aid understanding of the basic concepts.

```
// the task class
class Fibonacci: public task {
public:
   const long n;
   long* const result;
   Fibonacci( long n_par, long* result_par ) : n(n_par), result(result_par) {}
   task* execute() { // Overrides virtual function task::execute
    if( n<2 )
      { *result = n;}
    else {
        long x, y;
        Fibonacci& x_task = *new( allocate_child() ) Fibonacci(n-1,&x);
        Fibonacci& y_task = *new( allocate_child() ) FibTask(n-2,&y);
        set_ref_count(3);
        spawn( y_task );
```

```
        spawn_and_wait_for_all(x_task);
        // Both tasks have completed, generate the result
        *result = x+y; }
    return NULL; }
};
// …
// invocation and result
// …
long fib_n;
Fibonacci& x = *new(task::allocate_root()) Fibonacci(n,&fib_n);
task::spawn_root_and_wait(x);
// fib_n now contains the result
```

The most important parts in the code are highlighted with **bold**.

Any task in TBB is a subclass of the library class *task.* This class contains a virtual function called *execute* that any task needs to override—it will be the method that's invoked when the task will execute. In our case, we created a new class called *Fibonacci* and have overridden the *execute* method to calculate the required Fibonacci number. Please note, the *execute* method cannot return its result, hence a pointer to the place where the result shall be stored is added to the task class (*long* const result*).

There are two basic ways to create a task: either using the static method *task::allocate_root*, which creates the root of the task graph or using the non-static *task::allocate_child* method which creates a sub-task of the current task. This way, an acyclic graph of tasks is built up, something that the scheduler will exploit when scheduling the tasks for execution. In our example, when the calculation of the Fibonacci number is triggered, the root task is created and spawned; all other tasks will be its sub-tasks (created within the *execute* method).

There are two ways to spawn a task: either using the basic *task::spawn* method (used in our example to trigger the task *y_task*) or using a spawn-combined-with-wait method. In our example, there are two such cases:

- *task::spawn_root_and_wait* for triggering the scheduling of the root task
- *task::spawn_and_wait_for_all:* used in the *execute* model to trigger task *x_task* and wait until all sub-tasks have finished executing

A peculiar feature of TBB is the need to set the reference count of the tasks that need to complete, using *set_ref_count.* The documentation does a pretty poor job explaining the reasons behind this, other than offering a simple rule of thumb: if no task continuation shall be used, the value shall be set to *number of tasks + 1;* for continuation style task management (explained later on in this chapter), the value shall be *number of tasks.*

9.2.3.2 Continuation Passing and Scheduler Bypass

TBB support *continuation style* task synchronization. The goal is to free the parent task from having to wait on its children; instead children tasks are allocated on a

special type of task called *continuation task.* Again, loosely based on the tutorial example, here's the Fibonacci number calculation using continuation tasks:

```
struct FibonacciCont: public task {
  long* const result;
  long x, y;
  FibonacciCont( long* result_par ) : result(result_par) {}
  task* execute() { *sum = x+y; return NULL; }
};

struct Fibonacci: public task {
  const long n;
  long* const result;
  FibTask( long n_par, long* result_par) : n(n_par), result(result_par) {}
  task* execute() {
    if( n<2 ) { *result = n; return NULL;}
    else {
        FibonacciCont& cont = *new( allocate_continuation() ) FibonacciCont(result);
        Fibonacci& x_task = *new( cont.allocate_child() ) Fibonacci(n-2,&cont.x);
        Fibonacci& y_task = *new( cont.allocate_child() ) Fibonacci(n-1,&cont.y);
        cont.set_ref_count(2);
        spawn( x_task );
        spawn( y_task );
        return NULL;} }
};
```

First of all, there's a new task created using *task::allocate_continuation* which takes on the role of concluding the work of the spawned sub-tasks (in our case, generating the sum of the results). Two important issues need to be highlighted: the reference count is set to 2 (there will be a continuation of tasks *x_task* and *y_task*) and—not obvious from the code—while the *execute* method of the *Fibonacci* class may return, the task itself will only be considered completed when the continuation task allocated by it will also complete—hence the original invocation of the *Fibonacci* task will still work: *task::spawn_root_and_wait* will only return when the continuation task itself is completed. One final note: there's no need for calling *task::spawn_and_wait_for_all:* once all sub-tasks (*x_task* and *y_task*) completed their execution, the continuation task will be automatically invoked.

TBB offers two additional features that may be useful under certain circumstances. The first one is called *scheduler bypass:* the *task::execute* method has a return value a pointer to an object of type *task*; if it return a valid pointer, the task pointed to will be the next task that will execute on the current thread, hence bypassing any scheduler decision. For the Fibonacci example this is actually a useful feature, as it reduces the overhead that may be generated by scheduler decisions.

The second feature is called *task recycling.* A parent task, instead of completing its execution may *recycle itself* (using a library call like *task::recycle_as_child_of*) to become the child of another task. Again, in our example, the parent task my re-

cycle itself as e.g. *x_task*; returning itself can guarantee that it will be called again and the execution will continue as the new child task.

9.2.3.3 TBB Task Scheduler

The TBB task scheduler—as most task schedulers—is essentially evaluating a task graph that is constructed by the active tasks as these go along and generate more children, continuation or even root tasks.

TBB's scheduler is sometimes characterized as a *breadth-first theft and depth-first work* scheduler. In practice it's a variant of the work stealing algorithm: each thread has its own queue of tasks that have priority of execution; if no tasks are available, stealing from other tasks will occur. The simplified algorithm can be summarized in three steps:

1. If the *execute* method of the current task returns a pointer to a task, that task will be executed next
2. Otherwise, take the youngest task from local queue (the one added last)
3. If no tasks are available, randomly choose another thread and steal the oldest task from its queue

Rules 1 and 2 guarantee a depth first execution, as long as this is feasible. This approach can have beneficial impacts in terms of cache reuse (child tasks are likelier to act on similar data) and tend to avoid an exponential explosion of the amount of tasks. Stealing (rule 3), on the other hand obviously aims at balancing load.

9.2.3.4 TBB Summary

TBB offers a useful toolbox of components and templates that can simplify the task of writing parallel programs in C++ (its target language). Its task-based programming support partly overlaps with Cilk, but it's much more feature rich and, through the complete exposure of the task interface, it allows a much more flexible usage, including the possibility of building new schedulers as well. Thus, in case the goal is to get a simple, C-based task support library, Cilk shall be the primary choice; in case a more sophisticated solution is needed, TBB might be the right solution (of course, in the context of using Intel-only libraries).

9.2.4 Microsoft Task Parallel Library

Microsoft first released the Task Parallel Library (TPL) for the .NET framework as a community technology preview in 2007. Since then, it became an integral part of the .NET framework (starting from release 4.0). It's once again a task-based programming framework geared towards CLR (Common Language Run-time) based

languages (such as C# or Visual Basic). Consequently, we will use C# throughout the examples in this chapter. For an in-depth discussion of TPL, we recommend Ref. [5].

Microsoft's support for parallel programming in the .NET framework obviously incorporates other mechanisms as well. These mechanisms are grouped in the *Parallel* class and include the following models:

- Support for *loop level parallelism: Parallel* provides a wide range of *For* and *ForEach* methods with support for various features, such as external control, monitoring, alteration and parameterization
- Support for *task execution:* the method *Invoke* is the basic mechanism for executing multiple tasks in parallel

In this chapter we will focus on the features of the task based model and the task scheduler as well as the *Future* concept supported by TPL.

9.2.4.1 TPL Task Model

The concept of task is modeled in TPL through the class *Task* in the namespace *System.Threading.Tasks.* Tasks are created using the *task factory* available through the static *Task.Factory* property, which is an instance of the *TaskFactory* class. The method *TaskFactory.StartNew* is responsible for creating a new task consisting of the method passed to it. Execution of tasks is done through the method *Parallel.Invoke*, waiting on completion of tasks is done with *Task.Wait* (for single specific task), *Task.WaitAny* (for any single task) and *Task.WaitAll*.

In the example below, we rewrite the TBB version of the Fibonacci number generator using TPL, in C#:

```
static void Fibonacci(int n, int& result),
{
  int x, y;
  if ( n < 2)
    result = n;
  else {
      Task x_task = Task.Factory.StartNew(() =>
              Fibonacci(n-1, x));
      Task y_task = Task.Factory.StartNew(() =>
              Fibonacci(n-2, y));
      Parallel.Invoke(x_task, y_task);
      result = x + y; }
}
```

The code is fairly straight-forward: two tasks are created (please note the use of lambda expressions) and then invoked (*Parallel.Invoke* will only return once both tasks have completed—internally, it used *Task.WaitAll* for this). The code is quite similar to Cilk code; but, as we'll see later on, it can be made even simpler using the concept of *Future.*

TPL supports mechanisms for task cancellation and task error handling. Task cancellation is a co-operative feature: the task itself has to support it and willingly obey to a cancel request. The mechanism to support this is called *cancellation token:* it is created outside of the task; passed on to the task when it's created (as a second parameter to *Task.Factory.StartNew*); the task then can check if a cancel was requested through the token. If this is the case, it shall throw a special exception that contains that specific token (*OperationCanceledException*)—this way the .NET run-time system will take the required actions for canceling the task and setting its status to *Canceled* (in which state then can be inspected elsewhere in the code). *Note:* the token object actually supports a method that checks the token and throws the exception automatically.

The code snippet below illustrates this behavior:

```
CancellationTokenSource ct_source = new CancellationTokenSource();
CancellationToken ct = ct_source.Token;

Task myTask = Task.Factory.StartNew(() =>
{     // some work, then occasionally:
      token.ThrowIfCancellationRequested(); }, ct);

// The token is signaled like this:
ct_source.Cancel();
```

The possibility to cancel tasks is a powerful feature that can support various execution models, including application level speculative execution.

In TPL, tasks are allowed to throw exceptions which can be handled by the thread that has invoked the task. The .NET framework provides a mechanism to integrate all the exceptions thrown by multiple tasks invoked together or waited on together; the exception handler can then filter these exceptions and act on each one independently.

By default, tasks are independent of each other and bound only through being invoked or waited on together. TPL supports a mechanism to bind a child task to its parent task explicitly, in which case the parent task will only complete once all its children tasks have completed. This can be achieved through an explicit binding option when the child task is created using *Task.Factory.StartNew*.

Figure 9.6 shows the different states a TPL task may be in during its lifecycle.

9.2.4.2 Futures, Continuations and Pipelines

TPL also supports the concept of *futures* and *task continuations*. A future is modeled as a task with a result value of a specific type; the task is the stand-in for the actual result. When an attempt is made to access the result stored in the future, one of the following four cases may be valid:

- The task has already completed successfully: the result will be read and used
- The task is under execution: the reader thread will block until the result is available

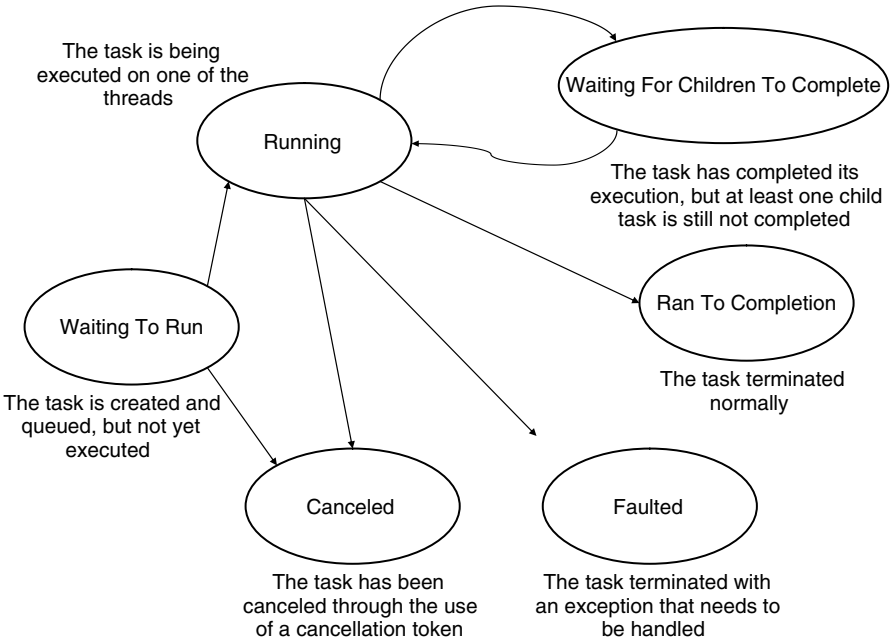


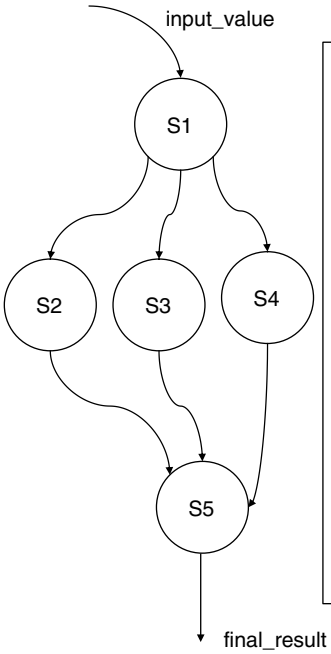
Fig. 9.6 Task state machine in TPL

- The task did not start yet: it will be executed immediately within the current thread (save for some very special situations)
- The task failed—the attempt to read the result will throw an exception that will need to be handled

Futures are created through instantiation of the parameterized class *Task<TResult>* and invoking its *StartNew* method. In the code example below, we calculate the *n*th Fibonacci number using futures:

```
    public static int Fibonacci(int n)
    {
    if (n < 2) return n;
    Task<int> future_x = Task.Factory.StartNew<int>(() => Fib(n-1));
    int y = Fib(n-2);
    return future_x.Result + y;
}
```

Clearly, the use of future task made the code cleaner and almost equivalent with the sequential one, while still executing in parallel. Please note, this code is not exception safe—*future_x.Result* should be read inside a *try* statement to catch possible exceptions and/or cancellations.




```
int s1 Result = S1(input_value);

// three futures are launched
// one for S2, S3, S4, respectively

Task<int> s2 Result = Task.Factory.StartNew<int>(
                          () => S2(s1Result));

Task<int> s3 Result = Task.Factory.StartNew<int>(
                          () => S3(s1Result));

Task<int> s4 Result = Task.Factory.StartNew<int>(
                          () => S4(s1Result));

// final result is calculated once all input is available
// for S5 to execute

int final_result = S5(s2Result.Result,
                      s3Result.Result,
                      s4Result.Result);
```

Fig. 9.7 Example of task graph executed with futures

Futures can be used to model task graphs, where some steps may execute in parallel. Consider the small example in Fig. 9.7, where *S2, S3, S4* can all execute in parallel and *S5* will use the value of all three (all are functions). For sake of simplicity, we assumed that all data dependencies are integers.

The concept of futures is also used in TPL to model continuations (tasks that can be triggered only when one or several previous tasks have completed). Continuations can be expressed in one of two forms:

- Using a special task factory method, *Task.Factory.ContinueWhenAll* which can take as argument a list of tasks that must be completed before the current task that's being created can be executed
- For a specific task a new task is being defined that will be its continuation, using the method *Task.ContinueWith*

For the example given in Fig. 9.7, execution of *S5* could have been expressed using the following mechanism:

```
var S5Result = Task.Factory.ContinueWhenAll<int, int>(
        new[] { s2Result, s3Result, s4Result },
        (tasks) => S5(s2Result.Result, s3Result.Result, s4Result.Result));
```

The final feature of TPL that is interesting in the context of programming many-core chips is the support for *blocking collections.* A *BlockingCollection<T>* is a data structure that blocks any thread that tries to access it until the structure is ex-

plicitly de-blocked by the thread accessing it, through calling a special method (*BlockingCollection<T>.CompleteAdding*). Using this particular data structure and tasks that take as arguments such structures, one can create automatic pipelines whose stages will be triggered automatically once the previous stage is complete (signaled through the call of *CompleteAdding* on its output data structure). The code fragment below illustrates a simple $S1 \rightarrow S2 \rightarrow S3$ pipeline:

```
var buff1 = new BlockingCollection<string>(1024); // 1024 is the size of the buffer
var buff2 = new BlockingCollection<string>(1024); // 1024 is the size of the buffer
var buff3 = new BlockingCollection<string>(1024); // 1024 is the size of the buffer

var st1 = Task.Factory.StartNew(() => S1(buff1));
var st2 = Task.Factory.StartNew(() => S2(buff1, buff2));
var st3 = Task.Factory.StartNew(() => S3(buff2, buff3));

Task.WaitAll(st1, st2, st3);
```

Note: for sake of simplicity, the default task factory was used in this example; this however may not be the best choice in case the different stages have a long execution time. TPL does support custom task factories; for more information please see Ref. [5].

9.2.4.3 The Default TPL Task Scheduler

The TPL task scheduler—unsurprisingly—relies on a task-manager thread pool, thread local queues and a variant of the work stealing algorithm for managing tasks. However, the concept of global queue is also kept, but it has a specific role: it is used to store tasks generated by threads that are not member of the thread pool that are responsible for task execution. The framework also provides a mechanism to force *any* task onto the global task queue.

The global task queue has FIFO semantics while thread local queues are LIFO (similar to the TBB task scheduler). If a thread runs out of tasks (the thread local queue is empty), it will attempt to fetch a task from the global queue; if there are no tasks in the global queue an attempt to steal from other threads in the thread pool will be performed.

The TPL scheduler also supports *in-lining* of tasks. If a task is waiting on another task *and* that task is in the local queue of the current thread, it will be immediately executed, so that the original task can make progress. Long running tasks and tasks in the global queue are not in-lined.

The scheduler automatically manages the thread pool by injecting or removing threads at regular intervals, based on the observed behavior: if there's no progress made with existing queues, new threads may be added; the same will happen if some cores experience lengthy idle times due to I/O operations generated by the

tasks. Frequent work stealing attempts—especially failed ones—are an indication that a reduction of the thread pool size may be appropriate.

9.2.4.4 Summary

TPL is clearly Microsoft's solution for parallel programming in .NET/CLR based languages. It is quite similar—architecture and functionality wise—to Intel's TBB, but it addresses a different application domain, namely, those written in managed languages, primarily for the Windows platform. We found especially the concept and implementation of futures quite powerful and useful for a large class of applications, resulting in a clean and easily comprehensible source code.

9.2.5 OpenMP 3.0

OpenMP (Open Multi-Programming) [6] is a standardized programming model specification for shared-memory based multiprocessing. It's backed by most major hardware and software vendors and consequently it has been successfully used, primarily in the high-performance computing domain. The latest version of the standard [7], released in 2008, added support for explicit tasks and it's supported for C, C++ and Fortran.

OpenMP itself is a collection of compiler directives (of the type *#pragma omp*), library routine API definitions and environmental variables that shall obey to a standard-defined set of semantic constraints. Beyond the standard definitions, vendors are free to provide their own library and compiler implementations.

OpenMP defines support for the following features:

- *Parallel regions:* sections of code that can be executed in parallel by multiple threads
- *Synchronization primitives:* co-ordination between parallel regions
- *Data-sharing attributes*: OpenMP supports both thread-private and shared memory constructs, as well as directive-driven partitioning of data and variable values among threads
- *Nested parallelism,* including task nesting
- *Thread pool dimensioning and management*

These features are usually realized through a combination of directives, libraries and environmental variables.

While our goal is to describe OpenMP's support for task-based parallelism, it's important to understand the logic behind the structuring of OpenMP programs (the concepts of parallel regions, thread teams and data-sharing attributes are of particular interest). Hence, we'll start with a brief overview of the key features of OpenMP (memory model, execution model, parallel regions and data-sharing mechanisms) before moving on to the in-depth analysis of the support for explicit tasks and task scheduling.

9.2.5.1 Structure of an OpenMP Program

An OpenMP program is a multi-threaded, shared-memory based application, but with support for thread-private data structures as well. There are five principles underlying the memory model of OpenMP:

- All threads have access to the same, globally shared address space: there are no address space islands
- Data can be shared or private to a thread
- Shared data is accessible by all threads: there's no concept of process, i.e., group of threads that can share memory just among themselves
- Private data is strictly local to one thread
- Transfer of data is transparent to the programmer; synchronization is mostly (but not always) implicit.

Consequently, all data needs to be, implicitly or explicitly, labeled. While there are some variations, there are essentially two basic labels: *shared,* with only one version of the data available globally and *private,* where all relevant threads have their own private copies. As we will see, threads may access the private data through the *same* variable—the compiler and run-time system make sure that the specific variable has a different meaning for each thread. OpenMP supports variations of the private data declaration in order to support specification of how data shall be privatized.

The basic execution model of Open MP programs is based on a *fork-join model.* Each program starts with one master thread; as parallelization directives are encountered, more threads—forming a *thread team*—will be spawned that will share the work that is marked as possible to execute in parallel. Here's the perhaps simplest OpenMP program, where a loop is marked to execute over multiple threads:

```
n = 2000;
#pragma omp parallel for
for (i = 0; i < n; i++)
   z[i] = x[i] * y[i]
```

which could be compiled and launched like this:

```
setenv OMP_NUM_THREADS 10
cc –xopenmp mycode.c
a.out
```

The *#pragma omp* statement instructs the compiler that the subsequent statement is a candidate for loop parallelization. As the number of threads is set to 10 (through an environmental variable), the *for* loop will be distributed over 10 threads; depending on the library implementation, the distribution could be so that values of *i* between 0 … 199 are handled by thread 0, 200 … 399 is handled by thread 1 and so on. What's important to note that the compiler and the run-time system will implicitly create a private version of *i* for each thread (and a private loop as well). All this happens behind the scene, without any programmer intervention.

A *parallel region* is thus a block of code executed by all threads simultaneously; the threads working on the same region are said to form an *OpenMP (or thread) team*. Parallel regions are defined using *#pragma omp parallel;* the directive allows for specifying the type of parallelism (e.g. loop parallelism), a condition for when parallel execution shall be performed as well as explicit specification of which variables shall be shared and which shall be private. Considering our simple example above, it may be rewritten as follows:

```
#pragma omp parallel if (n > 10) /
shared (z, x, y, n) private (i)
{
  #pragma omp for
  for (i = 0; i < n; i++)
      z[i] = x[i] * y[i]
} // end of parallel region
```

In this example parallel execution will only happen is the loop is big enough (more that 10 iterations) and the programmer explicitly specifies that *z, x, y* and *n* shall not be privatized, while *i* must be (again, that means that each thread will have its own private *i,* initialized to one of the possible values).

Within a parallel region, some code may require that only one thread will execute it. Such areas shall be marked using the *#pragma omp single* directive (later on we will show a practical usage of this feature).

By default, the end of a parallel region constitutes a barrier, where all threads shall wait before the execution can be continued by the master thread (in the example above, the closing '}' represents such a barrier). Explicit barriers can be added with the directive *#pragma omp barrier*; on the other hand, a parallel region may be marked as non-synchronized (no barrier at the end) by adding the keyword *nowait* to the *#pragma omp parallel* directive. In this case, the execution of the master thread would continue without waiting for all threads to finish.

OpenMP also supports *critical sections* and *atomic sections*. While *single* code sections can be executed by only one thread overall, a critical section (or an atomic section) will be executed by *all threads,* but only one at a time; thus a critical section is another form of synchronization between members of the thread team. The syntax for defining critical sections and atomic sections is

```
#pragma omp critical
#pragma omp atomic
```

The only difference between the two is that only a single statement (e.g. assignment) can be marked with *atomic*, while *critical* can be used for code sections of any size. The reason for this distinction is purely optimization of execution.

9.2.5.2 Explicit Tasks

Arguably, OpenMP has always had the implicit concept of tasks: when a parallel construct is encountered, an implicit task per thread is created and executed. The

task is *tied* to that specific thread and—unless *nowait* is used—all threads would synchronize at the end of the parallel region.

In OpenMP 3.0 the concepts of *explicit task* and *decoupled execution* was introduced. Any thread encountering a task will package a new instance (code and possible private data), but the task may be executed by a thread in the same team at some later time. Nesting of tasks is also permitted, i.e. tasks may themselves generate new tasks.

Tasks are defined using the *#pragma omp task* directive, with syntax similar to the *#pragma omp parallel* directive. Here's the complete definition:

#pragma omp task [*clause*[[,] *clause*] …]
  structured-block

where clause can be

  if (*expression*)
  untied
  shared (*list*)
  private (*list*)
  firstprivate (*list*)
  default (**shared** | **none**)

The semantics of the clauses are slightly different and require some explanation. The *if* clause controls the execution of the task: if *expression* is false, the encountering thread will execute the task immediately, thus in-lining the task. We already looked at *shared* and *private*, however at this point it's important to understand the difference between the semantics of *private* and *firstprivate*. The clause *private* is primarily relevant for parallel execution of *for* loops: each thread (or task) will get a private version of the variable that is initialized to one of the possible values that is encountered during loop execution (the typical example is a loop control variable that will be set to one of the values in the loop range); *firstprivate* on the other hand initializes the privatized variable to the value that is encountered when creating the private copy.

To illustrate the meaning of these clauses, let's look at a concrete example. The code snippet below performs the traversal of a list using tasks that would act on each element of the list in parallel:

```
#pragma omp parallel
{
 #pragma omp single private (list_p)
 {
  list_p = list->head;
  while (list_p != NULL) {
   #pragma omp task firstprivate (list_p)
        visit_node (list_p);
   list_p = list_p->next;
  }
 }
}
```

Here's what happens through the execution of this code:

- *#pragma omp parallel* will trigger the creation of a thread team that will execute the enclosed code in parallel
- *#pragma omp single* will make sure that the list traversal will happen only within one thread. This is required, otherwise all threads within the thread team would traverse the list and, consequently, each node would be visited by as many times as the number of threads in the pool.
- *#pragma omp task* will generate a new task for each element of the list. By using *firstprivate* each task will have its own local *list_p* pointer copy, set to the value available when the task was created—i.e, each task will have a *list_p* pointing to the specific element that the function *visit_node* will act upon. The generated tasks are not executed by the thread that performs the list traversal, but rather are put in a task queue and executed by the threads in the thread team

The final clause available in *#pragma omp task* is *untied.* This clause controls how a task may be scheduled: a tied task can only be executed by the thread that first 'grabbed' it (started executing it, at which point it becomes tied to that thread), while the execution of an untied task can be handed over to any other thread. Tasks are by default tied because it's likely that their execution may rely on some thread local data; if this is not the case and only task local and shared data is accessed from a task, *untied* will give more freedom to the run-time system in scheduling tasks. We'll look at this feature in the context of task scheduling.

A final aspect that needs to be clarified is *task synchronization* and *task completion.* All tasks generated within a parallel region will be synchronized (waited upon) at implicit and explicit barriers; in addition, OpenMP 3.0 supports the directive *#pragma omp taskwait* which introduces an explicit synchronization point for all child tasks—but only tasks on the first level, *not* further descendents. This leaves an important aspect undefined: if a function generates a task tree, what will happen to private (and shared) data structures accessed by those child tasks that did not complete, if the original function returns meanwhile? As the behavior is undefined—and similar to the dangling pointer problem—best approach is to make sure that such situations will not occur through explicit synchronization and wait for completion mechanisms.

Here's a code snippet to illustrate *#pragma omp taskwait*, through a post-order traversal of a tree:

```
#pragma omp parallel
void tree_postOrder (tree_element *p) {
 if (p == NULL)
   return;
 #pragma omp task
  tree_postOrder(p->left);
 #pragma omp task
  Tree_postOrder(p->right);
 #pragma omp taskwait
  visit_node (p); }
```

Here the *#pragma omp taskwait* directive guarantees that the sub-trees are visited first, before the current node is accessed; as there will be a synchronization point at each level, it's guaranteed that the processing of the complete sub-trees is completed before visiting the root node.

9.2.5.3 Task Scheduling

According the OpenMP specification [7], scheduling decisions can occur at *task scheduling points,* where task switching may be performed. The specification defines the following locations as task scheduling points:

- At the generation of an explicit task
- The last instruction of a task region
- At *#pragma omp taskwait* directives
- At explicit and implicit barrier regions
- Optionally, anywhere in *untied* tasks

Task switch means beginning or resuming execution of a different task bound to the current thread team. Specifically, the following actions may be performed by threads:

- Start execution of a tied or untied task bound to the current thread team
- Resume any suspended task to which the thread is tied or resume any suspended, untied task that is bound to the same thread team

The order of choice between these options is left unspecified and thus is implementation specific. Finally, the specification defines two scheduling constraints that any scheduling algorithm must follow:

- If the *if* clause of an explicit task evaluates to *false,* the task shall be executed immediately after generation
- Scheduling of new tied tasks is constrained by the availability of other, not suspended tied tasks: if any such tasks are available, these shall have scheduling priority from the current thread's perspective. The only exception is if the new task is a descendent of all tasks pending execution

These constraints leave quite some freedom for practical implementation of task scheduling, in terms of task selection, queuing and load balancing. As we will see when we compare the performance of the different task based models, this leads to significant differences in the performance of practical implementations.

A particular feature of OpenMP's task model is that it allows for suspension of an untied task and thread switching: a task may be suspended in the middle of its execution and resumed by another thread. This feature is missing from other task based libraries: once a task is started, it is either canceled (possible e.g. in Microsoft's TPL) or it completes; the only case when a task may be suspended is when the OS may decide to suspend the thread that executes it. However, from the thread's perspective, the task will be run to completion.

Figure 9.8 illustrates the parallel execution framework of OpenMP and the possible sequential execution on just one thread.

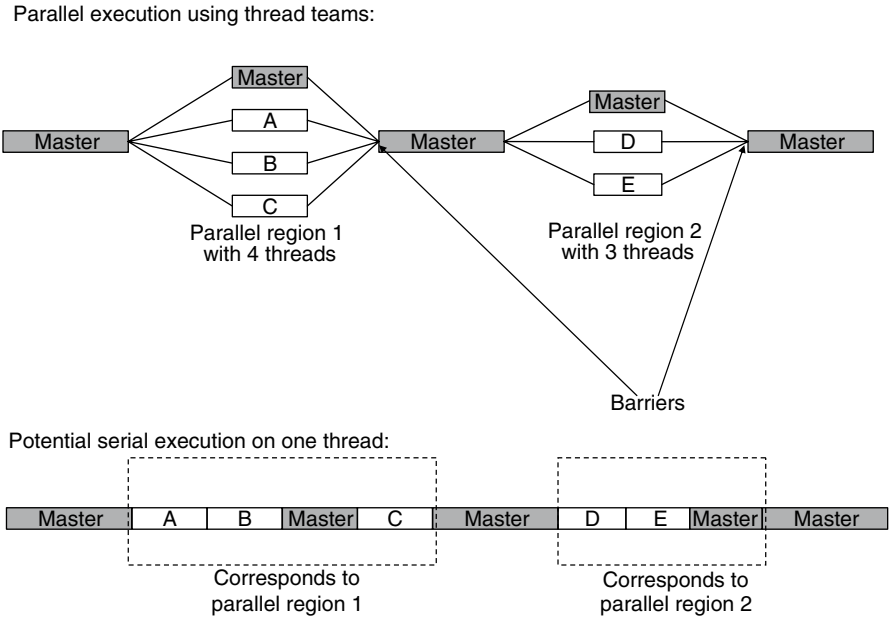


Fig. 9.8 Parallel execution of OpenMP programs and the concept of thread teams

9.2.5.4 Summary

OpenMP is perhaps the best known and most widely used standardized solution for writing shared memory based parallel programs. It provides powerful features—such as automatic privatization of data, flexibility in defining parallel regions etc—that make it a preferred choice for cases when shared memory is of prime interest. However, as we will see through empirical results, in terms of performance it lags behind quite significantly when compared with some of the other task-based libraries. The flexibility it provides seems to be its weak point as well: variations in implementation question its portability.

9.2.6 Comparison of Task Based Programming Libraries

As the parallel computing community seems to be settling on the consensus that task based models are a promising way forward (and most vendors are jumping on the train and churning out their own implementations), the performance of different flavors becomes ever more relevant. Obviously portability and support for multiple OS and language environments is of prime interest; but performance clearly is an important factor when deciding for a particular solution.

Quite recently there were several empirical results published that aimed directly at comparing various task based models in the context of different applications. Our

analysis is based on the studies available in ref. [8–10]. The first one is focusing on the impact of task granularity on the performance; the second one was evaluating performance while taking into account development effort; the third one evaluated the different models when applied to executing unbalanced task graphs (specifically, the problem of unbalanced tree search). All studies included OpenMP 3.0 variants, Cilk (or Cilk++) and TBB, thus we ill focus on these ones. One of the studies also included plain pThread-based implementations [9], while another one [8] also evaluated a highly optimized Cilk-like task based library called Wool (for more information on Wool [11]).

The first conclusion—consistent across the three studies—is that OpenMP seems to yield the lowest performance, while Cilk and TBB are competing for the top spot (Cilk seems to be coming out on top slightly more often). It's also important to emphasize that Cilk and TBB based implementations were able to show consistently improving performance as the number of cores kept increasing, often expressible as first order polynomial equations. OpenMP's performance on the other hand was also degrading as the granularity of tasks became higher (more fine-grained tasks), consistent with the high overhead due to task scheduling reported by one of the studies (ranging from 4% till as much as 80%, with a typical value hovering around 20%). This wide variation in overhead also highlights the importance of actual implementations: the variance was between different tools, not within a single tool applied to different applications.

Interestingly, the speedup obtained in Ref. [10] with hand-crafted, optimized, manually load-balanced, non-task based OpenMP implementations was only marginally higher than the speedup measured with Cilk and TBB (11× on 16 cores versus 9×). In our view this shows the efficiency of the simple model behind both Cilk and TBB.

The benefits of hard-optimizing the *spawn* mechanism were reported in Ref. [8], where the Wool implementation often outperformed Cilk with as much as 50%, when applied on problems with a very large number of fine-grained tasks. This indicates that indeed the cost of generating a task seems to be dominant factor, consistent with the observations regarding OpenMP as well.

It is also interesting to note the experiences reported in Ref. [9] with respect to development efficiency and cost of faults when using different libraries. The development times with TBB and Cilk were comparable, but OpenMP had a 50–100% penalty (still significantly lower compared to pThreads). The same pattern was observed also with respect to the time needed to correct bugs, with Cilk significantly outperforming even TBB.

9.2.7 Summary

While there are significant variations between the different task based models, the basic principles are the same: large number of tasks generated independently of the capabilities of the underlying machines; distributed scheduling; work stealing as a

load balancing mechanism. The performance evaluations we analyzed show that the task based paradigm has the potential for supporting scalability as the number of cores is increasing, making it a prime candidate for programming many-core systems.

OpenMP deserves a particular mentioning. While its performance with respect to explicit tasks still needs significant improvements, it provides an interesting approach that has the potential of gaining acceptance in the industry (due to standardization and the significant amount of tools supporting it).

Finally, we cannot conclude without noting that the best performing libraries (Cilk and TBB) seem to be coming from the largest hardware vendor—Intel. In our view this underlines once again the importance of coupling the development of hardware with the development of supporting software models; this coupling will be even more important as the scale of modern chips will pose new software challenges.

9.3 Data-Parallel Model

Data-parallel programming models look back to a long history, especially in scientific and graphics computing, where the typical computational problem involves applying the same algorithm on a (very) large data set. More recently, as parallel computing was emerging once again as the prime candidate for programming new chips, the usage of Graphics Processing Units (GPUs) expanded well beyond traditional domains, including the world's top supercomputers. This success was also due to the emergence of programming models and libraries that made these chips programmable even by programmers without special training in GPUs—if any, the CUDA (Computer Unified Device Architecture) model [12] has done the most in bringing GPUs and data-parallel programming into the main-stream.

In this chapter we detail one of the promising approaches to bridging the gap between main-stream CPU and GPU programming, OpenCL [13]. Our choice of OpenCL is motivated by multiple reasons: it's standardized and backed by most GPU and CPU vendors; it's GPU-independent; it provides a uniform model for writing portable programs for both CPU and GPU processors using both the data-parallel and task parallel model. Other frameworks for data-parallel programming—such as CUDA—have pretty much the same characteristics and use similar constructs and concepts.

9.3.1 OpenCL

Open Computing Language (OpenCL) was initially promoted by Apple (who still holds the copyright on the name), but it's now backed by most major chip vendors through the Khronos Compute Working Group, responsible for its standardization.

At the writing of this chapter, release 1.1 [14] was just made available, but the main-stream release 1.0 was already supported by most CPUs and GPUs as well as Apple's MacOS X v10.

OpenCL is based on C, but it excludes support for some features (such as function pointers, recursion, variable length arrays, bit fields etc) while adding some features to support specific concepts such as work items, workgroups, synchronization and new data types. In addition it has a powerful library of built-in functions, e.g. for image processing.

9.3.1.1 OpenCL Architecture

The execution platform on which an OpenCL program is executed is modeled as a collection of *OpenCL devices,* managed by a *host processor,* responsible for dispatching work to the devices. Internally an OpenCL device consists of one or several *compute units (CU),* each divided into one or more *processing elements (PE)* that may have SIMD or SPMD characteristics. For a CPU, cores will usually be reported as compute units.

The basic execution model of OpenCL relies on the concepts of *kernel* and *program.* A *kernel* is the basic unit of executable code, similar to a function in another language which can be applied over a data set (for data parallel execution) or as one instance, to model a task. A *program* is a collection of kernels and supporting functions for managing kernels.

When a kernel is executed on a device, an *N*-dimensional index space is associated with it and for each point in the index space an instance of the kernel will be launched. This instance is called a *work item* and it's identified by its position in the index space. Work items may be organized into *work groups*, also identified with unique ids. All work items within a work group will execute concurrently on the processing elements of one compute unit. For example, a 9×9 array may be subdivided into 9 work groups, each consisting of 3×3 work items.

Execution of an OpenCL program is based on the concept of a *command queues*. The host will place commands into one of the command queues (of which there may be several) which are then scheduled for execution on the available devices. There are three types of commands supported in OpenCL:

- *Kernel execution*: a kernel is scheduled for execution using an index space
- *Memory management*: transferring data between memory objects and mapping memory objects between address spaces
- *Synchronization*: constrain the execution order of commands

Commands are executed asynchronously between host and devices. OpenCL supports two styles of command execution:

- *In order*: commands are executed serially, the next command cannot start before the previous one completes
- *Out-of-Order*: commands are still issued in order for execution, but do not wait on each other, thus any synchronization requires the use of explicit mechanisms

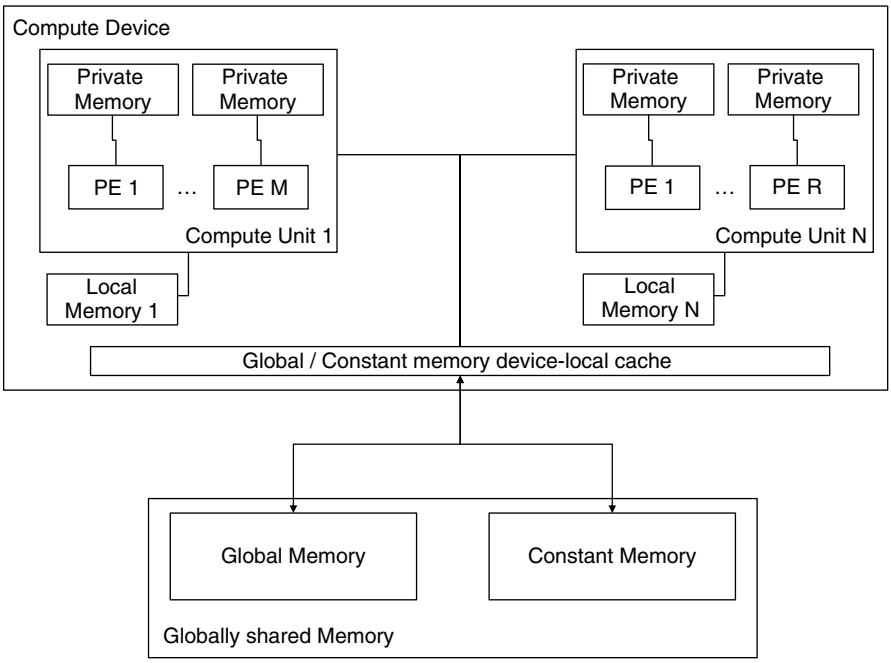


Fig. 9.9 OpenCL memory hierarchy

OpenCL also defines a hierarchical memory model, shown in Fig. 9.9. There are four main types of memory:

- *Global memory* areas are read/write accessible from any work item executing anywhere in the system
- *Constant memory* is initialized once and remain constant with read access from anywhere in the system (it's a write-once-read-many type of global memory)
- *Local memory* is only accessible from within one work group and is shared between work items. It can be allocated by the host or the devices, but it's only accessible from the devices executing the kernels associated with the work items
- *Private memory* is local to a single work item and may not be accessed from anywhere else in the system

Note: memory allocated on the host and used exclusively on the host is not included into these categories.

Memory management is explicit, including access, thus applications must make sure that no concurrent update of the same global memory is done without proper control. OpenCL offers library functions for explicitly mapping memory between devices and global memory; the usage of these is essentially required if global sharing needs to be achieved.

As work groups are essentially isolated, synchronization only makes sense within work groups and not between work groups. In addition, synchronization between commands is supported.

OpenCL provides two synchronization mechanisms for work groups:

- *Work group barriers* require that all work items execute the barrier before any is allowed to continue
- *Memory fences* guarantee that all load/store operations issued before the fence are committed before any subsequent memory operation will be executed

Similarly, there are two synchronization mechanisms for commands:

- A *command queue barrier* ensures that all previously queued commands will be completed before any new commands will be executed
- *Event-based synchronization* allows a command to wait on the completion event of a previously issued command. This mechanism can be used to model dependencies within a task graph

It shall be clear from this very brief and condensed description that OpenCL is primarily targeted at writing programs that can be distributed across multiple GPU-like devices; however it takes the design of such programs to a higher abstraction level and allows for a single environment to be used to design software spanning multiple processors with different characteristics.

9.3.1.2 OpenCL for Many-Core Programming

As mentioned earlier, the power of OpenCL lies in the support for both the data parallel and the task parallel model, across multiple types of hardware, in a transparent manner. However, to be fair, there are very few actual usage examples of OpenCL outside the GPU world: the primary usage area was and still is the programming of SIMD machines.

The basic abstractions that allow support for both models of programming are *kernels* and *command queues*. Kernels represent tasks that are either executed as one work item (in which case it defaults to a single task) or as a collection of work items, across an index space (in a SIMD fashion). For each type of device in a system (GPU/SIMD machine or normal CPU) there will be at least one command queue where kernels will be queued for execution; the runtime system will be responsible for dispatching the kernel instances (tasks) from the command queue to the available compute units in that specific device. The run-time system will also make sure that SIMD kernels can execute in parallel and can synchronize using one of the methods described in the previous chapter.

Besides support for modeling single tasks as kernels with one work item, barriers and command completion events allow modeling of task graphs: a task dependent on another task will synchronize on the completion event of the command that contains that specific task. However OpenCL has a very important limitation with respect to task parallel programming: kernels are simple calculations that cannot act as host functions and enqueue new kernels for execution—in other terms, dynamic, task generated tasks are hard to realize in OpenCL. This is the main reason the Fibonacci example is missing from this sub-chapter: it cannot be implemented efficiently in OpenCL.

9.3.1.3 Summary

OpenCL is a good candidate for data-parallel applications and for applications that can be expressed as static task-graphs; but it's not suitable—as it is today—for applications with dynamically unfolding task graphs, where tasks would need to be generated by previously executed tasks. We believe that allowing dynamic task management on CPU type of processors would be a welcome addition to the OpenCL standard that would put this language on par with other task-based libraries, while preserving the powerful data parallel and cross-platform capabilities.

9.4 The Actor Model

We introduced and discussed the Actor model extensively in Chap. 8. In this chapter, our focus is on how programs that can exploit many-core hardware capabilities can be written using this paradigm; we will use Erlang to showcase these capabilities.

9.4.1 Erlang's Actor Model

The basic unit of parallelism in Erlang is the *light-weight process,* which shares no data with any other process and can only communicate with other processes through asynchronous messages, whose delivery is not guaranteed by the run-time system. An Erlang process—other than a process spawn to perform a simple calculation (which is the Erlang way to implement the concept of a task or OpenCL kernel)—is a *reactive* parallel entity, waiting for external messages, reacting on those by performing some computations, optionally triggering other processes to perform calculations and usually replying to the received message with an answer (a message sent to the originator of the received message).

Erlang has a powerful mechanism for abstracting out concurrency and letting the programmer focus on implementing the logic of the application called *behaviors*. A behavior is an abstract implementation of a certain type of application, encompassing all the generic mechanisms required by that type of application, while the functional behavior is left for implementation by the user of the behavior, in the form of a set of callback functions that the behavior requires to be present in order to function correctly. Some examples of default behaviors provided as part of OTP (Open Telecom Platform), the run-time system of Erlang, include:

- *gen_server:* implements a generic server to be used in client-server type of applications
- *gen_fsm:* implements a generic Finite State Machine (FSM) that can instantiated for any type of FSM
- *supervisor:* implements the concept of process supervision tree, used in Erlang's fault management system

We will exemplify how behaviors can be used to separate concurrency concerns from actual functional implementation through a simplified implementation of a server that performs authentication of users and then hands off the handling of requests from clients to a function that is provided by the programmer instantiating the server. This example is based on the generic server implementation in Ref. [15] and the ftp example program on the main Erlang site, www.erlang.org.

We define the server as a module that exports two functions. Here's the source code:

```
-module(ex_server).
-export([start/3, stop/1]).

start(Name, Users, F, State) ->
    register(Name,
        spawn(fun() ->
                loop(Name, Users, F, State)
            end)).

stop(Name) -> Name ! stop.

loop(Name, Users, F, N) ->
    receive
      {connect, Pid, User, Password} ->
        case member({User, Password}, Users) of
        true ->
            Max = max_connections(),
            if
              N > Max ->
                  Pid ! {Name,
                        {error, too_many_connections}},
                  loop(Name, Users, F, N);
              true ->
                  New = spawn_link(fun () ->
                              F(Pid)
                          end),
                  Pid ! {Name, {ok, New}},
                  loop(Name, Users, F, N + 1)
            end;
        false ->
            Pid ! {Name, {error, rejected}},
            loop(Name, Users, F, N)
        end;
    {'EXIT', Pid} ->
        loop(Name, Users, F, lists:max(N-1, 0));
    stop ->
        void;
    Any ->
        loop(Name, Users, F, N)
end.
```

What happens in this code?

We define a module called *ex_server* which exports two functions: *start* and *stop*. The function *start* is responsible for initializing the server and it requires from its user

- the *Name* of the server, that can be used later to connect to the server
- the database of *Users*, which contains the list of users allowed to connect to the server
- the function *F* that implements the functionality of the server and which accepts one argument, the *pid* (process identifier) of the client that has connected to the server; the function is expected to terminate when the handling of the user is completed.

The main implementation of the server is the function *loop (Name, Users, F, N)* which loops indefinitely. It performs the following actions:

- accepts connect requests from users and verifies that the user is member of the database *Users* and that the server has not reached its maximum capacity (if the user is unknown or maximum capacity has been reached, a negative response is sent back)
- if the user is authenticated and the server has not reached the maximum capacity, a new process is spawned and linked to the server process (using *spawn_link*), which will handle this user; the core of the process will be the function *F* received in the *start* function; the pid of the newly spawned process is returned to the user
- as the spawned process is linked to the server, the server will receive an EXIT signal when it terminates which will be used to keep track of the connected clients

Note: this example uses a so-called *tail recursive implementation*: the last action in the *loop* function is always a call to itself, with new arguments; as this is the last action, the Erlang run-time system will execute the new instance in the *same stack space* as the caller instance of the function; hence no additional memory will be used. This is an often used technique in functional languages.

What we achieved here is a complete de-coupling of concurrency implementation and actual functional content: the user of our simple *ex_server* module will not have to worry about how concurrency and scalability will be handled; the only thing it has to supply is a function (*F*) that implements the protocol logic (probably as a state machine) and can interact with a single client. Internally, the server is a typical realization of the actor model: it creates a new Erlang process (a new actor instance) with functionality *F* for every interaction and lets the originating actor know about this new instance; the two instances will communicate using messages; when the interaction is concluded, the actor destroys itself and its creator is notified. The server implementation then will perform the cleanup that is required (in our case an update of the number of active users).

A possible implementation of the protocol logic (pseudo-ftp, taken from Erlang's home page) could look like this:

```
handler(Pid) ->
   receive
      {Pid, quit} ->
         Pid ! {ftp_server, ok};
      {Pid, Op} ->
         Pid ! {ftp_server, do_op(Op)},
         handler(Pid)
   end.
```

(For this case, the argument *Name* in the *ex_server:start* function is set to *ftp_server*). Clearly this is a clean protocol implementation, with absolutely no logic for distribution or scalability. In fact, this function simply implements the internal logic of an Actor instance.

Within this model, there may be thousands of users and for each of these there will be a new process spawned. In our simple example, all processes are created on the current Erlang node (roughly equivalent to a processor) and thus—assuming the use of the SMP version of Erlang—will seamlessly make use of all the cores available on this particular node. This approach is reminiscent of the principle underlying the task model: expose as much parallelism as possible and let the run-time system limit it; as Erlang processes are light-weight—not much more costlier to create and destroy than tasks in some task based libraries—we believe this approach offers the possibility to scale up as the number of cores will increase.

It's important however to understand that the Actor model is not equivalent to the task based model. While implementation of some form of task model would be possible in Erlang (or in any other Actor library), the Actor model is clearly targeted at applications with different type of parallelism: coarser grained and interaction based, where the parallel activities have to co-operate in order to achieve their goals. Neither is better than the other—just simply target different type of applications.

9.5 Summary

In this chapter we illustrated how the different parallel programming models that we believe are prime candidates for many-core processors—task model, data parallel and Actor model—are implemented and how these can be used in practice. This presentation is by no means exhaustive and some other important environments (such as the Haskell language or MPI for message passing/Actor style programming or CUDA for data parallel programming—and the list would be very long) were not covered simply for lack of space. Our goal was to illustrate the mechanisms underlying these models and how these can be applied in real world applications; consequently, many of the principles and solutions we presented here can be applied directly when using other languages or libraries.

References

1. Frigo M, Leiserson C E, Randall K H (1998) The implementation of the Cilk-5 Multithreaded Language. Proceedings of the ACM SIGPLAN 1998 conference on Programming Language Design and Implementation, 212-223
2. Gruman G, Hattersley M, Butler T R (2009) Mac OS X Snow Leopard Bible. Wiley & Sons
3. Reinders J (2007) Intel Thread Building Blocks: Outfitting C++ for Multi-core Processor Parallelism. O'Reilly Media
4. Intel Corporation (2010) Intel Threading Building Blocks. http://www. threadingbuildingblocks.org/uploads/81/91/Latest%20Open%20Source%20Documentation/Getting_Started.pdf. Accessed 11 January 2011
5. Campbell C, Johnson R, Miller A, Toub S (2010) Parallel Programming with Microsoft .NET: Design Patterns for Decomposition and Coordination on Multicore Architectures (Patterns & Practices). Microsoft Press
6. Chapman B, Jost G, van der Pas R, Kuck D J (2007) Using OpenMP: Portable Shared Memory Parallel Programming. The MIT Press
7. The OpenMP Architecture Review Board (2008) The OpenMP Application Program Interface. http://www.openmp.org/mp-documents/spec30.pdf. Accessed 10 January 2011
8. Podobas A, Brorsson M, Faxén K-F (2010) A Comparison of some recent Task-based Parallel Programming Models. 3rd Workshop on Programmability Issues for Multi-core Computers
9. Ravela S C (2010) Comparison of Shared Memory Based Parallel Programming Models. PhD Thesis, School of Computing, Blekinge Institute of Technology.
10. Olivier S L, Prins J F (2010) Comparison of OpenMP 3.0 and Other Task Parallel Frameworks on Unbalanced Task Graphs. International Journal of Parallel Programming 38(5-6):341-360
11. Faxén K-F (2008) Wool: a Work Stealing Library. SIGARCH Computer Architecture Newsletter 36(5):93-100
12. Sanders J, Kandrot E (2010) CUDA by Example: An Introduction to General-Purpose GPU Programming. Addison-Wesley Professional
13. Khronos Group (2010) OpenCL: Introduction and Overview. http://www.khronos.org/developers/library/overview/opencl_overview.pdf. Accessed 11 January 2011
14. Khronos OpenCL Working Group (2010) The OpenCL Specification Version 1.1. http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf. Accessed 11 January 2011
15. Armstrong J (2003) Making Reliable Distributed Systems in the Presence of Software Errors. PhD thesis, Royal Institute of Technology, Stockholm, Sweden. http://www.erlang.org/download/armstrong_thesis_2003.pdf. Accessed 11 January 2011

Chapter 10
Looking Ahead

Abstract This final chapter summarizes the experiences of the first decade of multi-core programming and looks at the challenges the research and practitioner community will face in the next decade. We discuss how we can tackle the issue of scaling the performance of single threaded applications, how we can deal with the bottlenecks that emerge as the level of parallelism increases and what higher abstraction level of software design will mean for deployment on many-core hardware; in general, how the hardware and software environment may look like in a few years' time. Finally we discuss how related fields of computing technology—such as exascale computing, cloud computing and pervasive mobile computing—may influence (and be influenced in the process) by advancements in programming many-core processors.

10.1 What We Learned Until Now

When the computing community realized that single-core performance scaling is coming to an end, it didn't have to start from scratch: it was looking back at over three decades of experiences from programming large scale parallel computers. It would have been fair to assume that the transition will be rather smooth and the whole issue will be remembered just as a minor bump in a triumphal march forward.

Yet this is not what happened. Companies were postponing their transition to multi-core processors as long as they could, but when they finally had to do it, it was rarely painless. Researchers were pouring out large quantities of papers, yet very few of the results were really groundbreaking and good solutions remained few and far between.

Why did this happen? In our view, there are two broad categories of reasons:

- *Human reasons:* most of the programmers were ill-prepared to make the switch from multi-threading to true parallelism and when they did, the results were sub-optimal; some of the root causes were related to technologies used for decades, but we simply cannot avoid the brutal fact that the competence of most was not sufficient to tackle the tricky issues that were emerging; in addition, the co-oper-

A. Vajda, *Programming Many-Core Chips,*
DOI 10.1007/978-1-4419-9739-5_10, © Springer Science+Business Media, LLC 2011

ation with the high performance computing community often lacked the breadth and depth that would have been required to facilitate hand-over of accumulated experiences
- *Technology reasons:* firstly, the industry—and research community—were trying to preserve as much as possible of the conventional wisdom of the single-core era and thus focused on approaches that were trying to maintain the well known and understood shared memory model in a completely new context; secondly, we underestimated the additional challenges of integrating parallelism on a single chip, both from hardware and software perspective; thirdly, we had to deal with a large legacy that was still required to continue scaling in performance

This is by no means and exhaustive list, but in our view captures the main issues that we faced. These are significant barriers in isolation; but in combination it posed a challenge far larger than anticipated.

What kind of conclusions can we draw from the past decade?

It became painfully clear that we reached the end of single core (and, implicitly, few-threaded) scalability: recently released chips were not delivering more than 20–30% improvement in the performance of single-threaded applications in the time-span of 2 years, a paltry annual improvement of around 10–14% (and declining). We can also be pretty confident that we *will not be able* to extract sufficient parallelism from *all* applications to enable these to improve performance on a multi-core hardware—neither manually nor—even less—automatically; while this may not matter for most applications (think of desktop applications such as text editors or spreadsheet managers), there will be sufficient amount of applications where failure will not be an option. Hence, *we must continue our quest for the holy grail of parallel computing:* improving performance of hard to parallelize applications on parallel machines with better overall performance. This is one of the central themes we address in this chapter.

We also learned that automatic parallelization of software written in traditional imperative languages has a limit that we are fast approaching or have even reached. In practice this means that the only option for extracting more parallelism from applications written in traditional imperative languages is *manual rewriting,* every time performance improvements are required. As a consequence, we need to look into new ways of expressing software that can help tools perform better when it comes to extracting parallelism from software. In general, we need to improve radically the *amount of semantic information that is transmitted from software down to execution environment*.

During the past few years, a large number of new computational models were proposed; however, we seem to slowly converge around a model based on generating a large number of tasks that—when executed in parallel—can help the application progress make progress. These tasks may share memory on a small scale, but we have learned that large scale sharing quickly becomes a limiting factor. At the same time, we have seen innovation in how cores are used: we don't have to execute the actual application on a specific core to make use of it; we can use the *additional cores to perform helper functions* that will help the main application progress faster.

As we will see, this technique can be used in many different contexts to help applications scale better on many-core processors.

During the past decade, the computing industry was able to deliver the first supercomputers capable of petaflops performance and the quest now is on for building exascale machines. One important lesson from these efforts that will apply to many-core programming as well is that such performance will not be possible with pure, homogeneous architectures: we will have to accommodate heterogeneity and we must find ways *to design software that can exploit heterogeneity—without losing its portability or future scalability*.

The first decade of multi-core programming also coincided with the first decade of *cloud computing:* the promise of unlimited, on a need basis and eventually very cheap computing power delivered as a utility. The pre-requisite of "free" computing power is starting to impact the way software is designed for cloud; we see similar trends in many-core computing that we have yet to fully understand and absorb into our ways of building software.

Finally, it's hard to ignore the unprecedented rise of *mobile computing*. Today smart-phones are ubiquitous in the developed world and will soon reach the same status in developing countries; at the same time the mobile industry already talks about 50 billion interconnected devices [1], the so-called *mobile swarm*. The resource management challenges of this field can certainly influence strategies for increasingly unreliable many-core chips.

These are the basic elements on which the rest of this chapter is built. Unlike the rest of the book, this chapter is highly speculative and, consequently, potentially contradictory or controversial: it will introduce novel concepts and ideas that have yet to become mature, but which may change the way we design software in the next 5 to 10 years.

10.2 Scalability Bottlenecks

We have discussed this subject throughout the book, in the chapters dedicated to hardware, operating systems and programming models. The red thread throughout the discussion was the impact of *sharing resources* on different levels:

- *Hardware:* explicit sharing (of memory and execution resources) adds overhead through synchronization mechanisms in general and cache coherence implementation in particular; implicit sharing (e.g. two un-related data structures mapped to the same cache line) adversely impacts performance, ruining the performance of the cache system
- *Operating system:* time sharing of core resources and in general sharing of operating system central resources will lead to an increased share of the time being used up by the operating system; in many cases, the OS is optimizing the wrong resource: core resources, ignoring e.g. memory access
- *Programming models:* memory sharing on application level quickly becomes the bottleneck both in terms of productivity (it's much harder to get synchronization

right) and performance: the larger the contention, the more time is wasted for access to resources

In our view the single most important task of the coming decade is to gradually shift away from sharing based models to more decoupled (hardware and software) systems that work autonomously, and interact sparsely and asynchronously. By now we have most of the basic technologies in place (e.g. 3D stacking of core-dedicated memory, space-shared operating systems, Actor/task based models), but we need to integrate these into a working stack of hardware and software.

10.3 Scaling Hard to Parallelize Applications

Allowing applications with just a few threads to make use of new machines where single-core performance has increased just moderately, yet there are more cores is one of the hardest research questions facing the research community. Automatic parallelization yielded modest results, even when guided by the programmer; for a significant amount of applications we have yet to find novel algorithms that would scale reasonably on a parallel machine.

Some of the approaches that have been extensively researched were *instruction* and *thread level parallelism* (*ILP* [2] and *TLP* [3]) and *speculative execution* [4]. ILP support is found today in most modern chips; however applications have been shown to have fairly limited amount of parallelism on machine code level that can be exploited by the hardware. Speculative execution—while intuitively appealing—failed because of the difficulty in predicting the likely paths in the code on which to speculate, thus leading to a large amount of squashed speculative execution paths, without any sizable gains.

The root cause for limited success or failure with both of these techniques is that the hardware simply *lacks sufficient information* to perform efficiently. It's very much like trying to find a needle in a haystack in the dark: it's essentially random experimentation that will rarely succeed.

Recently at least three research results were presented in Refs. [5–7] that revisited speculative execution from a different perspective and have shown encouraging results. All three approaches relied on the same idea: the programmer (and the software) *has to provide more information on what it tries to achieve.* In Ref. [5] this information is limited to indicating when the execution of two functions is *commutative* (may be performed in any order); in the second paper [6] the programmer is required to provide a *predictor function* that can be used to decide how to perform speculative execution; finally in the third paper [7], this conveying of semantic information is made pervasive, providing a wealth of information to the execution environment and the hardware to help speculative execution. This approach also enabled speculative execution on a much larger scale: the programmer is able to tell where it makes sense—not just on a short sequence of machine instructions, but on the level of whole functions.

Interestingly, papers [6] and [7] report results on the same application, namely Huffman decoding. It's considered embarrassingly sequential because the encoded data cannot be easily split up in the middle: it's hard to predict where compressed code boundaries are. In Ref. [6], an intelligent predictor is presented that has a fairly good accuracy; using this predictor, the run-time system can speculatively pre-execute decoding of the stream from the position predicted by it; if it was accurate, by the time the main execution path reaches this point, the result is already available—hence a speedup was obtained.

Paper [7] uses the same pre-execution techniques, but it makes a decision on where to speculate based on a different information: by analyzing the Huffman code tree, it can establish that, with a certain probability (say, 95%), there's a code boundary within 8 consecutive bits; by starting 8 parallel speculative threads, it can be reasonably certain that one of those will speculate on the right position (in our case, the certainty level is 95%). Hence, a speedup may be obtained—something confirmed by simulations, in both cases.

This may seem a waste of computing resources: especially in Ref. [7], 87% percent of the effort is always wasted (while 13% will succeed with a high probability). This is however, in our opinion, a false argument: the same paper has shown that speedup can be obtained even within the same power budget: replacing one high performance core with many simpler cores, speculative execution still resulted in $8\times$ speedup on 64 simple cores—within the same power budget. The gain obviously comes from increased use of parallelism: while decoding individual chunks will be much slower, this slow-down is more than compensated for through increased successful parallelism.

There are two conclusions that can be drawn from these research results that show a potential way forward for applications with limited amount of 'traditional' parallelism.

First, the amount of semantic information exploited in parallelizing the Huffman decoding algorithm is beyond reach for hardware or software that is just observing a running application. The speedup was possible exactly because this information was conveyed to the run-time system in a form that it was able to make use of.

Second, especially Ref. [7] has shown that applications with limited amount of parallelism can still benefit from an increased amount of cores: the performance gains (while nearly not linear) followed an upward trend as the number of cores kept increasing. In fact speculative execution—and a related technique called speculative pre-execution—will benefit from the increased core count, even if these are simpler cores. This aspect is especially promising, as it provides a path forward for hard to parallelize applications deployed on many-core processors.

In summary, we believe applications with limited traditional parallelism can benefit from many-core processors by deploying speculative execution enhanced with dramatically increased amount of information on the nature of the application and especially where speculation is likely to yield results. The more cores there will be, the less important it will be if the work of a single core is wasted—as long as *one* will succeed in its speculation *and* the space of speculation can be limited using programmer provided information to a level that ensures faster execution than on a

single core. Much research remains to be done in this area, especially on the interface between software and hardware as well as on which type of semantic information is likely to be most beneficial.

10.4 Programming at Higher Abstraction Level

Efficiency of compiled code on one hand and portability, ease of programming, coupled with productivity on the other hand have been the major drivers behind development of new programming languages, libraries and programming models, from the first version of Fortran all the way to today's scripting languages. However these three characteristics—efficiency of code on one hand, portability and productivity on the other hand—were more than often at odds with each other: efficiency usually required languages close to the actual hardware (most imperative languages), while productivity and portability usually meant the usage of high level languages, often coupled with automatic code generation (through model compilation, for example). General purpose very high level languages almost always carry with them a sizable performance penalty.

We already discussed the issues around auto-parallelizing compilers and the impact the lack of semantic information has on performance. The same is valid with programming using high abstraction general purpose languages: the compiler still lacks *domain knowledge:* about what is it that the programmer really wants to model?

The tension between the need for higher productivity and lower performance fueled fresh research into using *high level, yet domain specific languages* (*DSLs*). In theory at least these shall combine the best parts of the two worlds: be sufficiently high level to enable higher productivity of even non-programmer experts while restricting the domain sufficiently for tools to be able to reason about the semantics of programs. By restricting the problem domain, DSLs aim at achieving two targets: provide constructs that are easily recognizable by experts and restrict the domain to such an extent, that tools can reason about the concepts and take efficient decisions on how to transform these into target code within the given context.

This line of research—at least in the context of programming many-core processors—is still in early stages. Ericsson is working on a Haskell-embedded domain specific language for programming massively multi-core DSPs, with encouraging results [8]; a more holistic approach is being pursued at the Pervasive Parallelism Laboratory (PPL), part of Stanford University [9]. Their vision of a holistic parallel computing stack is shown in Fig. 10.1; the main components of this vision—on the software side—are:

- A family of *embedded domain specific languages* targeting different well defined domains (such as scientific computing, robotics etc.)
- A *virtualized embedding language,* that provides an infrastructure which enables domain specific optimizations without the need to re-write the environment for each and every new DSL

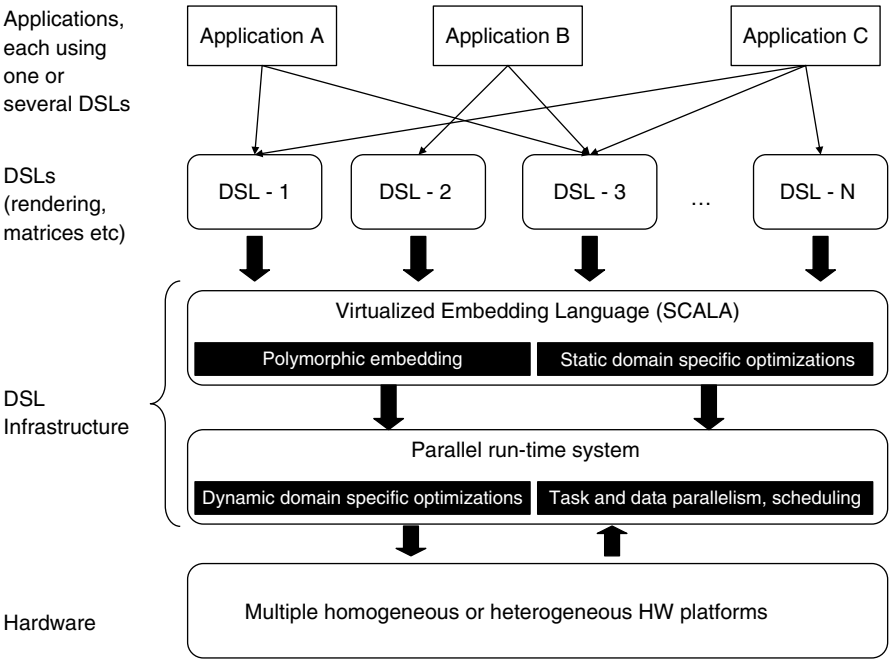


Fig. 10.1 DSL based parallel computing software infrastructure

- A *parallel run-time* that enables dynamically defined domain specific optimizations and supports both data- and task-based parallelism. It's important to note that this run-time environment, in the Stanford vision, executes on top of a Java Virtual Machine, thus somewhat limiting its applicability

The most interesting concept put forward by PPL is that of *virtualized host language* [10]. It is defined as a language that can provide an environment to a series of embedded languages in which the implementation of the embedded languages is not distinguishable from an equivalent stand-alone implementation, in terms of expressiveness (including syntax, semantics, ease of use by domain experts), performance (support for static and dynamic optimizations and code optimizations similar to stand-alone languages) and safety (it shall not be possible to use constructs not included in the embedded language). The choice for such language is *Scala* [11], which we also covered in this book; it allows overloading of all language constructs and polymorphic re-definition of basically any language construct. Using these mechanisms, the DSL-specific implementation of certain operations can be re-defined to take into account the knowledge about the domain itself. For example, a simple DSL for matrix operations may figure out that a sum of multiplications can be re-written as a multiplication with the result of a sum.

In our view DSLs offer powerful new opportunities for both raising the abstraction level at which software is designed and automating the extraction of parallel-

ism. The key insight is that by restricting the domain, specific domain knowledge can be built into the analysis, compilation and optimization phases—yet another way to improve the amount and quality of semantic information available to the run-time system and hardware.

10.5 Interaction with Related Domains

The field of computing has witnessed four important trends during the first decade of the 21st century: the rise of cloud computing, the rise of mobile computing, the push for exascale computing farms and multi-core programming. In this chapter we look at the interdependencies between these domains and how these will interact during the next decade.

10.5.1 Cloud Computing

Simply put, cloud computing is the remote delivery of IT resources (processing and storage) on a pay-per-use basis; it is built on a few fundamental principles and technologies:

- *Virtualization*, which enables on-demand instantiation of computing resources, de-coupled of the physical location; it is the cornerstone of most cloud computing offerings
- *Pay per use model*, which allows users to *lease (virtual) resources* instead of *buying physical computers*
- *Elastic capacity management*, which allows scaling up or down the allocated resources, depending on the needs
- *Self-service, remote interface:* the user can manage the leased computational resource himself, remotely, usually through a shell or web interface.

The concept of cloud computing is changing in a significant manner the way we reason about software: the cost of computational and persistent storage resources has declined to a level where it's not anymore a defining factor; other characteristics—such as the cost of networking and security—started playing an important role. This is quite similar to the trend we see in many-core chips: the level of utilization of cores is becoming less important than the cost of moving data to and from memory as well as inside the chip. Optimizing data movement is one of the primary concerns in both cases; speeding up execution by deploying speculation on a large scale—see e.g. Ref. [7] for many-core programming and Ref. [12] for cloud computing—is another one.

In our view there's a good opportunity for cross-fertilization and re-use of results between these two areas, as both face similar issues (large amount of communica-

tion constrained computing resources); thus we believe the two communities shall co-ordinate their efforts more closely.

10.5.2 Exascale Computing

Building a data center capable of exaFLOPS performance is still perhaps a decade away, however there are compelling reasons for doing so—some domains, such as global weather forecast or biological simulations demand processing power at this scale.

There are interesting similarities between the challenges faced by exascale computing and many-core chips. *Power* is the evident one: the problem of dark silicon (the impossibility to power up all transistors within a chip) is actually quite similar to the problem of supplying sufficient electricity to power a 1,000 petaflops machine—and so could be the solutions. Utilization of hardware accelerators, low power cores and advanced power management schemes and algorithms are applicable to both domains.

Eliminating *scalability* bottlenecks and mitigating *reliability* at such a scale are also related problems in the two domains. Experts agree that exascale machines are unlikely to be shared memory based, thus a more decoupled approach is needed; on the reliability side, there's a need to design mechanisms that can deal with failures that will occur at much higher rates—in absolute terms—simply due to the sheer size of the system. Here again the two fields shall co-operate much closer.

Some people are still doubtful that exascale machines will ever be built—and the camp of those claiming that a 1,000 core chip will never be practical is equally large. We believe however that there's really no alternative in sight—solving our most stringent problems *will* require machines at such scale—and the two domains of macro-computing and micro-computing can for sure co-operate on these important issues: power efficiency, scalability and reliability.

10.5.3 Mobile Computing

Just ten years ago mobile phone were used for little more than making calls and sending text messages. By contrast, today's mobile phones are more capable internet connected devices than the PCs of the 1990s of the last century and we are fast approaching the point in time when virtually anyone on this planet who wants to will have access to a mobile device. The industry is thinking even further, predicting 50 billion connected devices within a decade [1].

Managing such a computationally and network-wise complex, distributed, unreliable (in terms of accessibility) system—the mobile computing swarm—poses major challenges in terms of resource allocation, fault detection, containment and management. This is another domain that can positively influence the domain of

many-core programming, primarily with respect to resource management techniques and fault tolerance.

10.6 Summary: the World of Computing in 2020

As Winston Churchill, the wartime British Prime Minister once noted, "it's difficult to make prediction, especially about the future". Predictions about the future of computing have been proven wrong more than once in the past: just think about IBM's estimate back in 1950 of how many computers would be sold worldwide (under 10) and the estimate of reaching speeds of 10 GHz+ well before 2010. However, if the current trend of shift to multi-core parallel computing will continue, we believe there are a few likely developments we will witness during the next decade:

- The emergence of what could be called *semantic computing:* provisioning sufficient amount of semantic information to the run-time system that would allow it to autonomously decide how best to execute the given program, including speculative execution and pre-calculation techniques
- The pervasive importance of *power-conscious computing*: if anything, power will be the dominant cost factor and constraint during the coming years and will influence design choices on hardware, operating system and application level
- Computing will evolve towards a much *less centralized, distributed, loosely coordinated model*, on chip, data center and mobile computing level; everything will be interconnected, but software will likely consist of loosely coupled threads of execution with as little sharing with other entities as possible.

This is the place where the famous final word shall be placed. Luckily, there's no such thing in the ever-changing, ever-evolving, ever-surprising yet wonderful world of computer science: it's just a series of amazing achievements and new challenges that have transformed the world we live in—and surely will do so in the future as well.

References

1. Ericsson (2010) CEO to shareholders: 50 billion connections 2020. http://www.ericsson.com/thecompany/press/releases/2010/04/1403231. Accessed 11 January 2010
2. Hung P, Flynn M J (1999) Optimum Instruction-level Parallelism (ILP) for Superscalar and VLIW Processors. Technical report, Stanford University
3. Steffan J G, Mowry T C (1998) The Potential for Using Thread Level Data Speculation to Facilitate Automatic Parallelization. Proceedings of the 4th International Symposium on High- Performance Computer Architecture: 2–13
4. Liu S, Gaudiot J-L (2008) The Potential of Fine-Grained Value Prediction in Enhancing the Performance of Modern Parallel Machines. 13th Asia-Pacific Computer Systems Architecture Conference: 1-8

5. Bridges M, Vachharajani N, Zhang Y, Jablin T, August D (2007) Revisiting the Sequential Programming Model for Multi-Core. Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture: 69-84
6. Prabhu P, Ramalingam G, Vaswani K (2010) Safe Programmable Speculative Parallelism. Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation: 50-61
7. Vajda A, Stenström P (2010) Semantic Information based Speculative Parallel Execution. Proceedings 3rd Workshop on Parallel Execution of Sequential Programs on Multi-Core Architectures
8. Axelsson E, Claessen K, Devai G et al (2010) Feldspar: A Domain Specific Language for Digital Signal Processing algorithms. Proceedings of the 8th ACM/IEEE International Conference on Formal Methods and Models for Codesign
9. Chafi H, Sujeeth A K, Brown K J, Lee H J, Atreya A R, Olukotun K (2011) A Domain-Specific Approach to Heterogeneous Parallelism. Proceedings of the 16th Annual Symposium on Principles and Practice of Parallel Programming
10. Chafi H, DeVito Z, Moors A, Rompf T, Sujeeth A K, Hanrahan P, Odersky M, Olukotun K (2010) Language Virtualization for Heterogeneous Parallel Computing. Onward! '10: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications
11. Odersky M, Spoon L, Venners B (2008) Programming in Scala: A Comprehensive Step-by-Step Guide. Artima Inc
12. Cadar C, Pietzuch P, Wolf A L (2010) Multiplicity Computing: A Vision of Software Engineering for Next-Generation Computing Platform Applications. FSE/SDP Workshop on the Future of Software Engineering Research: 81-86

Index

A. Vajda, *Programming Many-Core Chips,*
DOI 10.1007/978-1-4419-9739-5, © Springer Science+Business Media, LLC 2011