Chapman & Hall/CRC
Computational Science Series

INTRODUCTION TO Concurrency in Programming Languages

Matthew J. Sottile
Timothy G. Mattson
Craig E Rasmussen

CRC Press
Taylor & Francis Group
A CHAPMAN & HALL BOOK

INTRODUCTION TO

Concurrency in Programming Languages

INTRODUCTION TO

Concurrency in Programming Languages

Matthew J. Sottile
Timothy G. Mattson
Craig E Rasmussen

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
A CHAPMAN & HALL BOOK

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Version Date: 20131120

International Standard Book Number-13: 978-1-4200-7214-3 (eBook - PDF)

Matthew J. Sottile and Timothy G. Mattson and Craig E Rasmussen

Introduction to Concurrency in Programming Languages

CRC PRESS
Boca Raton London New York Washington, D.C.

Contents

List of Figures

Chapter 1

Introduction

Objectives:

- Introduce the area of concurrent programming and discuss the relevance and timeliness of the topic.
- Investigate some areas where concurrency appears, both in programs and real-world situations.

This book is intended to give an introduction to thinking about concurrency at the level of the programming language. To date, the vast majority of mainstream programming languages are designed to express sequential programs. Current methods for writing concurrent programs in these languages exist primarily as add-on libraries that encapsulate the expression of concurrency in a form that, to the compiler or language, remains sequential. Techniques such as OpenMP provide more information to the compiler in the form of code annotations, but they still suffer from the fact that they are inherently bound to the sequential languages that they are used with. Java is one of the only mainstream languages with concurrency concepts defined as part of the language itself, and even then, most of the facilities for concurrent programming are provided within the standard library and not the language definition. Given that these facilities most often exist outside the scope of language definitions, the acts of optimization, efficient code generation, and correctness checking by compilation and source analysis tools prove to be very difficult.

Language-level constructs created specifically for concurrent programming are necessary to exploit parallel processing capabilities in a general sense that facilitates automatic compilation and analysis. We will discuss a set of these language constructs in this text based on those that have proven successful in the decades of parallel languages research, in the hope that programmers will understand and use them to best take advantage of current and upcoming architectures, especially multicore processors. These will be demonstrated in the context of specific concurrent languages applied to a set of algorithms and programming patterns. Using these constructs requires an understanding of the basic concepts of concurrency and the issues that arise due to them in constructing correct and efficient concurrent programs. We will cover this topic to establish the terminology and conceptual framework necessary to fully understand concurrent programming language topics.

Readers should come away from this book with an understanding of the effect of concurrency on programs written in familiar languages, and the language abstractions that have been invented to truly bring concurrency into the language and to aid analysis and compilation tools in generating efficient, correct programs. Furthermore, the reader should leave with a more complete understanding of what additional complexity concurrency involves regarding program correctness and performance.

Concurrency and parallelism: What's the difference?

Before we dive into the topic, we should first establish a fundamental piece of terminology: what does it mean to be concurrent versus parallel? There is a great deal of press currently about the advent of parallelism on the desktop now that multicore processors are appearing everywhere. Why then does the title of this book use the term *concurrency* instead of *parallelism*? In a concurrent system, more than one program can appear to make progress over some coarse grained unit of time. For example, before multicore processors dominated the world, it was very common to run multitasking operating systems where multiple programs could execute at the same time on a single processor core. We would say that these programs executed *concurrently*, while in reality they executed in sequence on a single processing element. The illusion that they executed at the same time as each other was provided by very fine grained time slicing at the operating system level.

On the other hand, in a multicore system (or, for that matter, any system with more than one processor) multiple programs can actually make progress at the same time without the aid of an operating system to provide time slicing. If we run exactly two processes on a dual-core system, and allocate one core per process, they will both make progress at the same time. They will be considered to be executing in *parallel*. Parallelism is a realization of a concurrent program. Parallelism is simply the special case where not only do multiple processes appear to make progress over a period of time, they *actually* make progress at the same time. We will discuss the details of this and other fundamental concepts further in Chapter 2.

1.1 Motivation

Concurrency is a core area of computer science that has existed for a long time in the areas of high-performance and scientific computing, and in the design of operating systems, databases, distributed systems and user interfaces. It has rapidly become a topic that is relevant to a wider audience of programmers with the advent of parallel architectures in consumer systems. The presence of multicore processors in desktop and laptop computers, powerful programmable graphics processing unit (GPU) co-processors, and specialized hardware such as the Cell processor, has accelerated the topic of concurrency as a key programming issue in all areas of computing. Given the proliferation of computing resources that depend on concurrency to achieve performance, programmers must learn to properly program them in order to fully utilize their potential.

Accompanying this proliferation of hardware support for parallelism, there is a proliferation of language tools for programming these devices. Mainstream manufacturers support existing techniques such as standardized threading interfaces and language annotations like OpenMP. Co-processor manufacturers like Nvidia have introduced the Compute Unified Device Architecture (CUDA) language for GPUs, while libraries have been created by IBM to target the Cell processor found in the Playstation game platform and various server-class machines. The Khronos group, an industry standards organization responsible for OpenGL, recently defined a language for programing heterogeneous platforms composed of CPUs, GPUs and other processors called OpenCL. This language supports data parallel programming models familiar to CUDA programmers, but in addition OpenCL includes task parallel programming models commonly used on CPUs.

In the high-performance computing industry, Cray is developing a language called Chapel, while IBM is working on one called X10 and Sun is working on a language called Fortress. The 2008 standard for Fortran includes primitives specifically intended to support concurrent programming. Java continues to evolve its concurrency primitives as well. The Haskell community is beginning to explore the potential for its language in the concurrent world. How does one navigate this sea of languages, and the tide of new ones that is surely to continue to follow?

1.1.1 Navigating the concurrency sea

Unfortunately, there is only a small amount of literature available for programmers who wish to take advantage of this capability that now exists in everyday devices. Parallel computing has long been primarily the domain of scientific computing, and a large amount of the educational material available to programmers focuses on this corner of the computing world. Computer science students often are introduced to the issues that arise when dealing with concurrency in a typical curriculum through courses on the design of operating systems or distributed systems. Similarly, another area of likely exposure to issues in concurrency is through databases and other transaction-oriented server applications where software must be designed to deal with large numbers of concurrent users interacting with the system. Unfortunately, literature in each of these fails to provide readers with a complete and general introduction to the issues with concurrency, especially with respect to its impact on designing programs using high-level languages.

Operating systems typically are implemented with lower level languages for performance reasons and to facilitate fine grained control over the underlying hardware. Concurrency can be quite difficult to manage at this low level and must be planned very carefully for both performance and correctness reasons. On the other extreme are database-specific languages, such as SQL. In that case, programmers work at a very high level and are concerned with very specific types of concurrent operations. Concurrency in both of these cases is very specific to the problem domain, and looking only at these cases fails to paint a complete picture of the topic. This book aims to provide an introduction for computer scientists and programmers to high-level methods for dealing with concurrency in a general context.

The scientific computing community has been dealing with parallelism for decades, and many of the architectural advances that led to modern concurrent computing systems has grown out of technological advances made to serve scientific users. The first parallel and vector processing systems were created to fulfill the needs of scientists during the Cold War of the twentieth century. At the same time, mainframe manufacturers such as IBM and systems software developers at AT&T Bell Laboratories sought to design systems that supported concurrent operation to service multiple users sharing single, large business computing resources. The work in scientific computing focused on concurrency at the algorithmic level, while designers of multiprocessing operating systems focused on concurrency issues arising from the interactions of independent, concurrently executing programs sharing common resources.

The problem most modern readers will find with much of this literature is that it focuses on programmers using parallelism by working at a very low level. By low level, we imply that the programmer is given direct and detailed control over how computations are split across parallel computing elements, how they synchronize their execution, and how access to shared data is managed to ensure correctness in the presence of concurrent operations. This focus on low-level programming is, in comparison to traditional sequential programming techniques, equivalent to teaching programmers to use techniques close to assembly language to write their programs. In sequential programming, one considers assembly language to be at the "low level" of the hierarchy of abstraction layers. Assembly language programmers control the operations that occur in the computer at a very fine grain using relatively simple operations. Operations are specified in terms of individual arithmetic operations, basic branches that manipulate the program counter to move between relevant regions of executable instructions, and explicit movement of data through the memory hierarchy from the main store to the CPU.

Many decades ago, with the advent of programming languages and compilers, programmers accepted that higher level abstractions were preferable to utilize hardware in a portable, efficient, and productive manner. Higher levels of abstraction associated with a language construct are related to the amount of insulation and protection it provides to the programmer above the underlying hardware. Higher yet are constructs that not only insulate programmers from the machine, but attempt to encapsulate concepts and techniques that are closer to the human description of algorithms and data structures. In addition, higher-level abstractions give the compiler more freedom in emitting code that is specific to a given computer architecture.

Complex arithmetic expressions, such as the computation of a square root or transcendental function, both of which could require detailed sequences of assembly instructions, could be expressed as atomic operations translated to machine code by a compiler. Similarly, the flow of control of a program could be abstracted above rudimentary program counter manipulations to operations that were closer to those used to design algorithms, such as `do-while` or `for`-loops and `if-then-else` conditional operations. Programmers realized, and quickly accepted, that algorithmic tools could assist them in translating algorithms from an abstraction layer where they could comfortably think and design programs to a lower one where the machine could execute equivalent machine code.

The basis of this text is that a similar set of abstractions exists for concurrent programming, and that programmers should start to think at this higher level with respect to concurrency when designing their programs to both achieve portability, high performance, and higher programmer productivity. In a world where concurrency is becoming pervasive, abstractions for concurrency will become increasingly important.

Unfortunately, parallel programming languages that provide this higher level of abstraction to programmers have yet to gain any acceptance outside of niche academic circles. There has not been a parallel equivalent of Fortran or C that was accepted by the broader computing community to build the critical mass of users in order for production-grade compilers and educational literature to be created. Java is the closest that can be found in popular usage, but it has not proven to be generally accepted in contexts where other languages have traditionally been used. Parallel languages inspired by Fortran, C, Lisp, Java, and many others have been proposed, but all have failed to gain widespread usage and acceptance. Parallel languages often have been relegated to the shelves of libraries in high-performance computing literature, and have lived most often in academic circles as research topics with the sole purpose of providing research contexts for Ph.D. dissertations. With parallelism clearly on the desktop, it is vital to change this state of affairs for the benefit of programmers everywhere. Programmers must be provided with tools to manage concurrency in general purpose programs. Language designers have responded in recent years with a variety of exciting concurrent languages.

The key is to focus on the language constructs that provide high level mechanisms for the programmer to utilize concurrency with the same amount of intellectual effort required to design traditional sequential programs. Ideally, building a program that uses concurrency should be only slightly more difficult than building a sequential one. It has become clear that, although many languages for parallel computing have come and gone over the years, there are successful abstractions that persist and appear over and over in languages to reduce the complexity of using concurrency to the developer's advantage. Educating programmers about these successful and persistent abstractions, and teaching them how to apply them to algorithms that are relevant to their work, is key in providing them with the background to be ready to use new languages as they arise. Furthermore, users who choose to remain in traditional sequential languages augmented with add-on tools for concurrent programming who understand these abstractions can implement them in their own programs when true parallel languages and compilers are absent.

1.2 Where does concurrency appear?

Programmers have adapted to concurrency whether or not they appreciate it in many contexts. A working programmer is hard pressed to claim that he or she has not had to deal with it at some point in his or her career. What are the contexts where this may have occurred?

Operating systems

The operating system of a computer is responsible for managing hardware resources and providing abstraction layers that give programmers a consistent and predictable view of the system. Few application programmers are ever concerned in modern times with the details related to how one sends a text file to a printer, or how a block of data traverses a network connection to another computer. The operating system provides interfaces to drivers that abstracts this from the programmer. Similarly, all modern operating systems[1] allow multiple programs to execute concurrently, each accessing these resources in a coordinated way. This has been especially useful with single processor systems where the single processing unit has been multiplexed to provide the illusion of parallel execution in modern multitasking operating systems.

[1]Embedded operating systems are the most common modern exception to this.

Concurrency arises in many contexts within operating systems. At a fundamental level, the operating system manages programs that execute at the same time through some form of time-sharing or multitasking. One can take a standard operating system today, run two programs, and they will happily coexist and run independently. The operating system ensures that when these programs are sharing resources (such as I/O devices and memory), they run in a predictable way in the presence of other concurrent processes. By "predictable," we imply that the programs execute in the same way regardless of whether they execute alone on the system or concurrently with other processes. The only significant difference that is likely to occur is an increase in the runtime due to contention for shared resources (such as the processor). From a correctness sense, the system will ensure that programs not sharing and modifying data (such as a file in the file system) will produce the same output regardless of the other processes executing concurrently on the machine.

Correctness is not the only concern in operating system design however. Given a set of processes that are sharing resources, each expects to make some reasonable amount of progress in a given amount of time. This is handled by scheduling algorithms that control the sharing of these resources between concurrently executing processes. In this context, it is important to provide some guarantee of service, or fairness, to all processes. As a result, operating system designers must also be concerned with the performance characteristics of the concurrent parts of their system in addition to their correctness.

Distributed systems

The majority of computer users today take advantage of distributed, network-based services, such as the world wide web or e-mail. Servers on the Web, especially those that are popular, have to deal with the fact that many users may be attempting to access them at any given time. As users expect a high degree of responsiveness, server designers must deal with concurrency to ensure a good user experience on their sites. This requires server programmers to pay attention to details such as:

- How to efficiently service requests as they arrive.
- Preventing connections with varying performance characteristics from interfering with each other.
- Managing data to cache frequently accessed information, and preventing conflicts between clients that make modifications on shared data.

In many instances, network software has performance as a top priority. This performance is defined in terms of responsiveness and performance as seen by clients. Furthermore, most servers manage some form of state that clients gain access to for reading and modification. Correctness of these concurrent operations is critical to maintaining the integrity of this state.

User interfaces

The user interface has long been an area where concurrency has been a core topic. Concurrency is an unavoidable feature of any interface because there are always at least two concurrently operating participants in an interface — the user and the program he or she is interacting with. In most graphical interfaces, the system allows the user to see and interact with two or more programs that are executing simultaneously. Managing the interactions of these programs with the shared graphical hardware and input from the user are areas where the user interface programmer must consider the implications of concurrency.

The user interface system must manage shared resources (such as the display and input device) and ensure that the proper programs receive the appropriate input and that all programs can display their interfaces without interfering with each other. Similarly, programs that are run in a graphical environment must be written to deal with the concurrency that arises in their internal logic and the logic necessary to interact asynchronously with the user.

Databases

Databases commonly are available to more than one client. As such, there are expected instances in which multiple clients are accessing them at the same time. In the case of databases, the system is concerned with providing a consistent and correct view of the data contained within. There is a delicate balance between providing a responsive and well performing system, and one that guarantees consistency in data access. Clever techniques such as transactions can minimize the performance overhead with many synchronization primitives by taking advantage of the relatively rare occurrence of conflicts between concurrently executing client interactions.

We will see that the successful abstraction provided by database transactions has inspired more general purpose methods that can be integrated with programming languages for writing concurrent code. This will be discussed when we cover software transactional memory schemes.

Scientific computing

Scientific computing has long been the primary domain of research into methods for expressing and implementing concurrency with performance as the key metric of success. Unlike other areas, performance was often the *only* metric, allowing metrics such as usability, programmability, or robustness to be pushed to a secondary status. This is because scientific computing often pushes machines to their extreme limits, and the sheer size of scientific problems, especially in their time to execute, makes performance exceptionally important.

An interesting side effect of this intense focus on performance is that many algorithmic techniques were developed in the scientific computing community for common, complex operations that exhibit provably efficient performance. For example, many operations involving large sets of interacting concurrent processing elements that are naively coded to take $O(P)$ time on P processors can actually achieve a preferable $O(\log(P))$ performance with these sophisticated algorithms. This has only really mattered in very large scale systems where the difference between P and $\log(P)$ is significant. We will see that efficient implementations of these operations exist, for example, in libraries like the popular Message Passing Interface (MPI), to provide high performance algorithms to programmers.

1.3 Why is concurrency considered hard?

The obvious question a programmer should be asking is why is taking advantage of concurrency more difficult than programming sequential programs. Why is concurrency considered to be hard? Concurrency is often treated in computing as a topic that is difficult to express and manage. It is regarded as a constant source of bugs, performance issues, and engineering difficulty. Interestingly, this belief has existed since parallel computers were first proposed. In his 1958 paper, Gill states that:

> Many people with whom the author has spoken have expressed the opinion that programming under such circumstances will be impossibly complicated and will never be worth while. The author feels strongly that this is not so. [40]

Fortunately, Gill was correct in his estimation of the utility of parallel systems and the possibility of programming them. As we will argue in this text, many difficulties programmers face in building programs that use concurrency are a side effect of a legacy of using primitive tools to solve and think about the problem. In reality, we all are quite used to thinking about activities that involve a high degree of concurrency in our everyday, non-computing lives. We simply have not been trained to think about programming in the same way we think about activities as routine as cooking a full meal for dinner. To quote Gill again, he also states this sentiment by pointing out that

> There is therefore nothing new in the basic idea of parallel programming, but only in its application to computers.

1.3.1 Real-world concurrency

Let's consider the activities that we take up in cooking a full meal for ourselves working from a simple recipe. We'll look at how one goes about the process of preparing a simple meal of pasta with home-made marinara sauce.

In the preparation phase, we first sit down and prepare a shopping list of ingredients, such as pasta, herbs, and vegetables. Once the list is prepared, we go to the market and purchase the ingredients. Computationally speaking, this is a sequential process. Enumerating the ingredients is sequential (we write them one by one), as is walking through the market filling our basket for purchase. However, if we happen to be shopping with others, we can split the shopping list up, with each person acquiring a subset of the list which is combined at the checkout for purchase.

The act of splitting the list up is often referred to in computational circles as *forking*. After forking, each person works autonomously to acquire his or her subset of the list without requiring any knowledge of the others during the trip through the store. Once each person has completed gathering his or her portion of the shopping list, they all meet at the checkout to purchase the goods. This is known as *joining*. As each person completes his or her trip through the store, he or she combines his or her subset of the shopping list into the single larger set of items to finally purchase.

After we have purchased the products and returned home to the kitchen, we begin the preparation process of the meal. At this point, concurrency becomes more interesting than the fork-join model that we would utilize during the shopping process. This is due to the existence of *dependencies* in the recipe. Assuming that we are familiar with the recipe, we would likely know that the pasta will take 8 minutes to cook once the water is boiling, while it takes at least 20 minutes to peel the tomatoes for making the sauce. So, we can defer preparing the pasta to prevent overcooking by aiming for it to finish at the same time as the sauce. We achieve this by waiting until the sauce has been cooking for a few minutes before starting the pasta. We also know that the pasta requires boiling water, which takes 5 minutes to prepare from a pot of cold tap water. Deciding the order in which we prepare the components requires us to analyze the dependencies in the recipe process: cooking pasta requires boiled water, pasta sauce requires crushed tomatoes, pasta sauce requires chopped herbs, etc.

This chain of dependencies in this simple cooking problem is illustrated in Figure 1.1. We can observe that the dependencies in the recipe do not form a linear chain of events. The pasta sauce is not dependent on the pasta itself being cooked. The only dependency between the pasta and the sauce is the time of completion — we would like the sauce to be ready precisely when the pasta is, but they have different preparation times. So, we start preparing the ingredients and start subtasks when the time is right. We'll chop the herbs, and start warming the olive oil while we mince the garlic that we'd like to saute before combining with the tomatoes. While the sauce is warming up, we can start the water for the pasta, and start cooking it while the sauce simmers. When the pasta finishes, the sauce will have simmered sufficiently and we'll be ready to eat.

The point of going through this is to show that we are quite used to con-

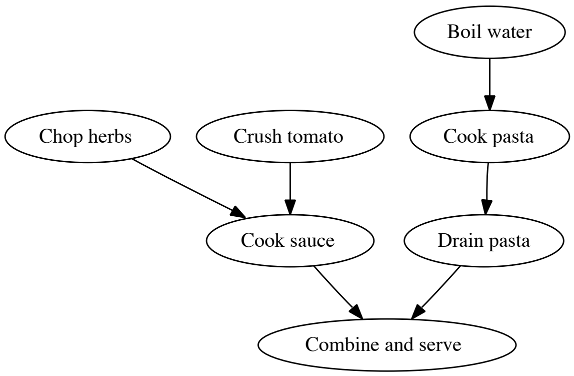

FIGURE 1.1: Dependencies in a simple cooking activity. Arrows point from activities to those that depend on their completion in order to proceed.

currency. Boiling a pot of water at the same time that we prepare part of the meal is just a natural thing we do in the kitchen, and it is an inherently parallel operation. The fact that we can understand, follow, and occasionally write recipes means we understand basic issues that arise in concurrent tasks where dependencies are important and sequential subsets of operations are used. Concurrency really isn't as bad as it is made out to be, *if we have the right methods to express and manage it.*

1.4 Timeliness

As we stated already, the advent of multicore and other novel parallel processors available in consumer hardware has made parallel programming an area demanding significant attention. As of the writing of this text, the state of the art with respect to programming tools for these architectures is primitive at best. Utilizing low core-count processors such as the Intel dual or quad-core processors is best achieved by explicit use of threads in libraries like POSIX threads or compiler extensions like OpenMP. In cases such as the IBM Cell, which represents a potential route that multicore processors will take in the future, programmers are required to work at an even lower level. In the worst case, they must program in assembly or using C library APIs that hide low-level instructions. Furthermore, users must define memory access scheduling explicitly — something that most programmers have assumed to be handled at the hardware level for many years due to the prevalence of cache-based processors.

It is quite likely that as the industry converges on architectures that meet the demands of general purpose computing, tools will arise to assist programmers in using these processors. These tools will provide abstractions above low-level memory access scheduling and the decomposition of instruction streams to various heterogeneous processing elements. We are seeing this today with a rising interest in using graphics co-processors as general purpose processors to achieve higher performance for some algorithms than the traditional main processor alone is capable of.

One area where tools will become available will be in language extensions and the use of novel parallel language constructs. We are seeing a move in this direction with tools like Nvidia's CUDA, the Khronos OpenCL definition, and revisions of existing standards such as OpenMP. The fact that parallel language constructs are designed to allow compilers to easily target parallel systems will mean that we will begin to see technologies from previously obscure languages become integrated with mainstream languages. For this reason, we feel that readers will benefit from learning how these language-level features can be used to construct algorithms in anticipation of tools that will employ them for these new architectures.

1.5 Approach

In this text, we start by laying out for the reader the set of issues that programmers face when building concurrent programs and algorithms. These topics in concurrency are widespread and applicable in many domains, ranging from large scale scientific programming to distributed internet-based systems and even into the design of operating systems. Familiarity with these topics and the consequences they have in design decisions and methods is vital for any programmer about to embark on any task where concurrency is a component. In order to ground this topic in areas with which programmers are already familiar, we will discuss the relationship that concurrency has to traditional sequential models used frequently in everyday programming tasks. Most importantly, the reader should understand precisely why traditional programming techniques and models are limited in the face of concurrency.

Before delving into the description of the high-level language constructs that programmers should begin to think about algorithms in terms of, we will discuss the current state of the art in concurrent programming. This discussion will not only serve as a summary of the present, but will expose the flaws and limitations in the current common practices that are addressed by higher-level language constructs.

In introducing high-level parallel programming techniques, we will also discuss notable milestones in the hardware evolution that occurred over the past forty years that drove the developments in linguistic techniques. It has repeatedly been the case that new hardware has been preceded by similar hardware introduced decades earlier. Multimedia extensions in consumer level hardware derive their lineage from vector processing architectures from the late 1970s and 1980s. The first dual and quad core multicore processors that arrived a few years ago are essentially miniaturized versions of shared memory parallel systems of the 1980s and 1990s. More recently, these multicore processors have taken on what can best be described as a “hybrid” appearance, borrowing concepts tested in many different types of architectures. We will discuss this historical evolution of hardware, the corresponding high-level techniques that were invented to deal with it, and the connection that exists to modern systems. Much of our discussion will focus on shared memory multiprocessors with uniform memory access properties. Many of the concepts discussed here are applicable to nonuniform memory access shared memory machines and distributed memory systems, but we will not address the nuances and complications that these other architectures bring to the topic.

Finally, we will spend the remainder of the text on a discussion of the various high-level techniques that have been invented and their application to a variety of algorithms. In recent years, a very useful text by Mattson et al. emerged to educate programmers on patterns in parallel programming [69]. This text is similar to the influential text on design patterns by Gamma et al., in that it provides a conceptual framework in which to think about the different aspects of parallel algorithm design and implementation [37]. Our examples developed here to demonstrate and discuss high-level language techniques will be cast in the context of patterns used in Mattson’s text.

1.5.1 Intended audience

This book is intended to be used by undergraduate students who are familiar with algorithms and programming but unfamiliar with concurrency. It is intended to lay out the basic concepts and terminology that underly concurrent programming related to correctness and performance. Instead of explaining the connection of these concepts to the implementation of algorithms in the context of library-based implementations of concurrency, we instead explain the topics in terms of the syntax and semantics of programming languages that support concurrency. The reason for this is two-fold:

1. Many existing texts do a fine job explaining libraries such as POSIX threads, Windows threads, Intel’s Threading Building Blocks, and compiler directives and runtimes such as OpenMP.

2. There is a growing sense in the parallel programming community that with the advent of multicore and the now ubiquitous deployment of

small-scale parallelism everywhere there will be a trend towards concurrency as a language design feature in languages of the future.

As such, it is useful for students to understand concurrency not as a library-like add on to programs, but as a semantic and syntactic topic of equal importance and treatment as traditional abstractions that are present in the programming languages of today.

1.5.2 Acknowledgments

This book is the result of many years of work on the part of the authors in the parallel computing community, and is heavily influenced by our experiences working with countless colleagues and collaborators. We would like to specifically thank David Bader, Aran Clauson, Geoffrey Hulette, Karen Sottile, and our anonymous reviewers for their invaluable feedback while writing this book. We also are very grateful for the support we have received from our families who graciously allowed many evenings and weekends to be consumed while working on this project. Your support was invaluable.

Web site

Source code examples, lecture notes, and errata can be found at the Web site for this book. We will also keep a relatively up-to-date list of links to compilers for languages discussed in this book. The URL for the Web page is:

`http://www.parlang.com/`

1.6 Exercises

1. Describe an activity in real life (other than the cooking example in the text) that you routinely perform that is inherently concurrent. How do you ensure that the concurrent parts of the activity are coordinated correctly? How do you handle contention for finite resources?

2. List a set of hardware components that are a part of an average computer that require software coordination and management, such as by an operating system, due to their parallel operation.

3. Using online resources such as manufacturer documentation or other information sources, write a short description of a current hardware platform that supports parallelism. This can include multicore processors, graphics processing units, and specialized processors such as the Cell.

4. Where in the design and implementation of an operating system do issues of concurrency appear?

5. Why is concurrency important to control in a database? Can you think of an instance where improper management of clients can result in problems for a database beyond performance degradation, such as data corruption?

6. Where does concurrency appear in the typical activities that a Web server performs when interacting with clients?

7. Beginning with all of the parts of a bicycle laid out in an unassembled state, draw a chart (similar to Figure 1.1) that shows the dependencies between the components, defining which must be assembled before others. Assume a simple bike with the following parts: wheels, tires, chain, pedals, seat, frame, handlebars.

Chapter 2

Concepts in Concurrency

Objectives:

- Establish terminology for discussing concurrent programs and systems.
- Discuss dependencies and their role in concurrent programs.
- Define important fundamental concepts, such as atomicity, mutual exclusion and consistency.

When learning programming for sequential machines, we all went through the process of understanding what it means to construct a program. The basic concept of building solutions to problems within a logical framework formed the basis for programming. Programming is the act of codifying these logic-based solutions in a form that can be automatically translated into the basic instructions that the computer executes. To do so, we learned one of a variety of programming languages.

In doing so, we learned about primitive building blocks of programs, such as data structures, statements and control flow primitives. We also typically encountered a set of fundamental algorithms, such as those for searching and sorting. This entailed learning the structure and design of these algorithms, syntax for expressing them in a programming language, and the issues that arise in designing (and more often, debugging) programs. Along the way, convenient abstractions such as loops and arrays were encountered, and the corresponding problems that can arise if one is not careful in their application (such as errors in indexing).

As programs got more complex, we began to employ pointers and indirection to build powerful abstract data structures, and were exposed to what can go wrong when these pointers aren't maintained correctly. Most programmers have at some point in their careers spent a nontrivial amount of time tracking down the source of a segmentation fault or null pointer exception. One of the driving factors behind high-level languages as an alternative to working with lower-level languages like C is that programmers are insulated from these issues. If you can't directly manipulate pointers, the likelihood of using them incorrectly is drastically reduced. Convenient higher-level language constructs abstracted above this lower-level manipulation of raw memory layouts and contents, and removed many of the tedious details required when building correct and efficient programs by hand.

Practitioners in programming rapidly learn that it is a multifaceted activity, involving gaining and mastering an understanding of:

- How to codify abstract algorithms in a human readable form within a logical framework.
- How to express the algorithm in the syntax of a given programming language.
- How to identify and remedy correctness and performance issues.

Much of the latter part of this text will focus on the first two points in the context of parallel programming: the translation of abstract algorithms into the logical framework provided by a programming language, and the syntactic tools and semantics provided by the language for expressing them. This chapter and the next focus on the third point, which as many working programmers know from experience is a crucial area that requires significant thought and practice even when the act of writing code becomes second nature. Furthermore, the design aspect of programming (an activity under the first point) is intimately related to the considerations discussed in this chapter.

The operations from which sequential programs are constructed have some assumed behavior associated with them. For example, programmers can assume that from their perspective one and only one operation will occur on their data at any given time, and the order of the sequence of operations that does occur can be inferred directly from the original source code. Even in the presence of compiler optimizations or out-of-order execution in the processor itself, the execution can be expected to conform to the specification laid out in the original high-level source code. A contract in the form of a *language definition*, or *standard*, exists between the programmer and language designer that defines a set of assumptions a programmer can make about how the computer will behave in response to a specific set of instructions.

Unfortunately, most languages do not take concurrency into account. Concurrent programming complicates matters by invalidating this familiar assumption — one cannot infer relative ordering of operations across a set of concurrent streams of instructions in all but the most trivial or contrived (and thus, unrealistic) circumstances. Similarly, one cannot assume that data visible to multiple threads will not change from the perspective of an individual thread of execution during complex operation sequences. To write correct concurrent programs, programmers must learn the basic concepts of concurrency just like those learned for sequential programming. In sequential programming, the core concept underlying all constructs was logical correctness and operation sequencing. In the concurrent world, the core concepts are not focused solely on logic, but instead on an understanding of the effect of different orderings and interleavings of operations on the logical basis of a program.

These initial chapters introduce the reader to the problems that can arise in this area in the form of non-determinism, and the conceptual programming constructs that exist to manage and control non-determinism so that program correctness can be inferred while benefiting from the performance increases provided by supporting the execution of concurrent instruction streams. We will also see that a whole new class of correctness and performance problems arise if the mechanisms to control the effect of non-determinism are not used properly. Many of these topics are relevant in the design of network servers, operating systems, distributed systems and parallel programs and may be familiar to readers who have studied one or more of these areas. Understanding these concepts is important as a basis to understanding the language constructs introduced later when we discuss languages with intrinsically parallel features.

2.1 Terminology

Our first task will be to establish a terminology for discussing parallel and concurrent programming concepts. The terminology will be necessary from this point forward to form a common language of discussion for the topics of this text.

2.1.1 Units of execution

In discussing concurrency and parallelism, one must have terminology to refer to the distinct execution units of the program that actually execute concurrently with each other. The most common terms for execution units are *thread* and *process*, and they are usually introduced in the context of the design and construction of operating systems (for example, see Silberschatz [87]).

As we know, a sequential program executes by changing the *program counter* (PC) that refers to the current position in the program (at the assembly level) that is executing. Branches and looping constructs allow the PC to change in more interesting ways than simply regular increasing increments. Similarly, the program is assumed to have a set of values stored in CPU registers, and some amount of memory in the main store available for it to access. The remainder of this context is the operating system state associated with the program, such as I/O handles to files or sockets. If one executes the same program at the same time on the machine, each copy of the program will have a distinct PC, set of registers, allocated regions of memory and operating sys-

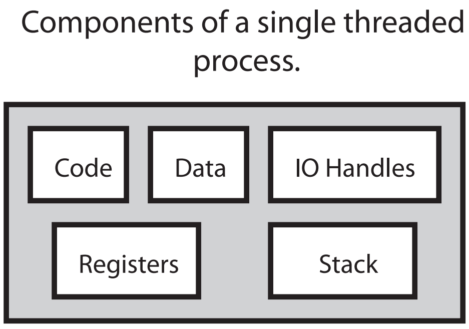

FIGURE 2.1: An operating system process and its components.

tem state. Programs, unless explicitly coded to do so, cannot see the state of other programs and can safely treat of the machine as theirs exclusively. The only artifact of sharing it with other programs would most likely be observed as performance degradation due to contention for resources. This self contained encapsulation of the environment and control state of the program is what we refer to as the execution unit or execution context.

In modern operating systems, these encapsulations of data and control state are referred to as *processes*. For the most part, they exist to enable multitasking operating systems by providing the necessary abstraction of the machine from the program, and also some measure of protection such that poorly behaving programs cannot easily disrupt or corrupt others.

One of the simplest methods to implement concurrent programs is by creating a set of processes that have a means to communicate with each other by exchanging data. This is precisely how many concurrent systems work. It is safe to assume many readers familiar with C will have seen the `fork()` call at one point, which allows a single program to create (or "spawn") multiple instances of itself as children that can execute concurrently. Simple server applications are built this way, where each forked child process interacts with one client. A parent process will be run, and as jobs arrive (such as requests to an HTTP server), child processes are created by forking off instances of the parent that will service the operations associated with each incoming task that arrives, leaving the parent to wait to service new incoming tasks.

The term *fork* typically has the same high-level meaning in all cases, but can be interpreted slightly differently depending on the context. When referring to the flow of control within a program, if this flow of control splits into two concurrently executing streams of execution, we would say that the flow of control had forked. In this usage, the implementation of the concurrent control flow isn't specified — it could be via threads or processes. On the other hand, the `fork()` system call typically has a specific meaning that is