

Concurrent CS: Preparing Students for a Multicore World

Daniel J. Ernst

Daniel E. Stevenson

Department of Computer Science

The University of Wisconsin – Eau Claire

Eau Claire, WI, USA 54702-4004

ernstdj@uwec.edu

stevende@uwec.edu

ABSTRACT

Current trends in microprocessor design are fundamentally changing the way that performance is extracted from computer systems. The previous programming model of sequential uniprocessor execution is being replaced quickly with a need to write software for tightly-coupled shared memory multiprocessor systems. Academicians and business leaders have both challenged programmers to update their skill sets to effectively tackle software development for these newer platforms [2].

At the University of Wisconsin – Eau Claire, we have taken steps early in our curriculum to introduce our students to concurrent programming. Our approach is not to add parallel programming as a separate class, but to integrate concurrency concepts into traditional material throughout a student’s coursework, beginning in CS1. Our goal is for students to gain both familiarity and confidence in using parallelism to their advantage. This paper describes the programming process we seek to introduce to our students and provides example assignments that illustrate the ease of integrating this process into a typical curriculum.

Categories and Subject Descriptors

K.3.2 [Computers and Education]: Computer and Information Science Education – *Curriculum*

General Terms:

Design, Experimentation

Keywords:

parallel programming, curriculum design

1. INTRODUCTION

The current trends in computer hardware are changing the needs of our students. Numerous performance and power limitations have prompted both academic and industry computer architects to abandon the quest for increased single-core performance [1,2]. Instead, the industry has turned to chip multiprocessing, allocating new die space gained from technology advances to replicating processors, thereby increasing theoretical throughput. Intel, among others [4], has put a large stake in the success of multicore architectures, as evidenced by their unveiling of an 80-core prototype chip last year [6]. Already, these major

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ITiCSE '08, June 30–July 2, 2008, Madrid, Spain.

Copyright 2008 ACM 978-1-60558-115-6/08/06...\$5.00.

manufacturers are offering multicore processors almost exclusively, even in low-end and laptop systems.

Unfortunately, extracting increased single-program throughput from this newly provided hardware will require far more programmer intervention than any other recent processor advancement. While many advances have been made over the years in parallel programming tools, it is still necessary for programmers to have a working knowledge of fundamental concurrency concepts to make productive use of multiple processors.

Parallel programming practitioners in the past have been limited to a small, specialized community of computer scientists, as access to multiprocessor systems was only available to those working in the research departments of large corporations, universities, or national laboratories. Because of this, the curricula of most computer science departments have only taught parallel programming as an advanced elective course, if at all.

However, our current undergraduate students will likely spend their entire career working on multiprocessor machines. The current CS workforce is not yet equipped to handle this shift in skills, and industry recognizes this. As part of their “Terascale Initiative”, Intel has put out a call to programmers and educators alike to expand their skill sets and their curricula to include multiprocessing. These concepts are becoming increasingly important in ensuring the short-term and long-term success of our students.

Our approach is to give students practice with the concepts behind parallel programming early and often by integrating them into our existing coursework. We have developed modules for CS1, CS2, and an algorithms course which introduce students to some of the basic concepts of parallelism in a way which is straightforward, interesting, and interwoven into other course topics.

Instead of spending time on teaching students specialized parallel programming languages, we focus on having students learn and apply principles of concurrency within languages with which they are already comfortable, such as Java and its object-oriented use of threading. Our goal is to have students emerge from the early part of our program being comfortable leveraging concurrency in their programs.

2. MULTIPROCESSING BACKGROUND

The introduction of threads into early CS coursework is not a new concept. Like many other institutions, we have been teaching the Java threads package for years. However, threads are usually introduced in the context of rotational multithreading. That is, the ability to switch among tasks running on the same processor. Some common examples are:

- Threading event-handlers in GUI development so the GUI does not become inactive during processing
- Handling blocking I/O in reading from sources such as sockets or hard drives
- Creating a server-type process that handles multiple users at the same time using independent threads

These are practical applications that would certainly gain some benefit from running on multicore processor designs. They should certainly be kept in any curriculum that wants to focus on multicore programming. However, the reason threads were introduced in these applications is because they need them in order to perform properly. Without the threads, multicore or not, the programs would cease to function as desired: the GUI would freeze, processing would halt while waiting for I/O input, or users would be queued up waiting for service.

Our concern is with code that runs perfectly well using a single thread, but that will never take advantage of the performance gains of current and future multiprocessor systems without being explicitly written for concurrency. In recent history, programmers could expect the performance of their software to scale up with the help of Moore’s Law. Under these conditions, a well-designed single-threaded program would likely be sufficient. However, as processor technology advances along its new multicore route, these programs will never run any faster (or, more likely, will actually get slower). As the number of cores increases, single-threaded performance will become relatively more unacceptable.

3. CREATING PARALLEL ALGORITHMS: STEP-BY-STEP

Before asking students to write parallel programs, they need to be given some straightforward structured processes to help them approach the problem with confidence. Below, we outline the parallel program design process [3] we introduce to our students through our curriculum. During early courses, we stick with the most basic versions of each step; gradually working up to a more complete model as they progress through the major.

3.1 Decomposition

The first step in writing parallel programs is for students to identify where there is parallelism in a program and to break it down into tasks. In early coursework, students are given simple problems with obvious data-parallelism (such as image convolution), which lends itself well to visual instruction. Later, students are shown ways to extract parallelism in slightly more abstract ways, such as within loops (in the encryption cracking assignment) and recursion.

3.2 Assignment

Once the student has identified the concurrency, they need to assign their tasks to individual threads. As part of this step, students are taught early in their education to be intelligent with their thread allocation. For example, a simple experiment of varying the number of threads generated can be used to show that, while performance may scale upwards at first, making a huge quantity of threads can cause performance problems due to excessive overhead.

3.3 Orchestration

Students will need to arrange (or “orchestrate”) tasks logically, especially with dependent tasks that require a particular sequencing. Again, in the early courses, students ignore synchronization issues during this step. Later in the curriculum, as they gain confidence in task assignment, partial ordering and basic synchronization constructs such as semaphores are added.

3.4 Mapping

In advanced parallel processing applications, the final step in this process is mapping the threads to hardware processors. In our assignments, this step is handled by the Java Virtual Machine, which schedules the threads. Because the students have very little control over this process, we do not cover this step in our modified curriculum.

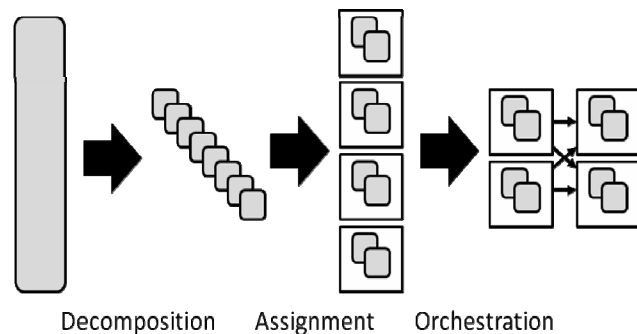


Figure 1. The parallel programming process, described in [3], as presented to the students. The mapping step is not included as it is handled by the Java Virtual Machine.

4. CURRICULAR ADDITIONS

Thus far, we have begun integrating these parallel programming concepts into three courses in our core curriculum: CS1, CS2 and Algorithms. All of these classes are currently taught in Java, so we use Java threads and synchronization primitives in accomplishing concurrent tasks. However, similar constructs exist in most major programming languages so the examples we describe below could be easily adapted. Additionally, our changes have an impact on courses that may have traditionally contained topics related to parallel computation already, such as Operating Systems.

Each of the following sections describes the particular curricular content of the concurrency modules.

4.1 Introducing Concurrency in CS1

In CS1 we want to keep our focus on the decomposition step. We would like to avoid any major issues with orchestration of the threads or results. Thus, we will look for problems that are ‘embarrassingly parallel’ and have few synchronization issues. The processing of the data in these problems simply needs to be broken up in a meaningful way, but there will be no major dependencies between the pieces. Thus, the concurrent solutions to these problems are similar to the single threaded solutions, but with an extra layer of threads added in. Students are shown that with a little extra work they can get 2 or 4x the performance on the same types of assignments we have traditionally given in the course. Below we give two examples of CS1 assignments we have adapted to incorporate concurrency.

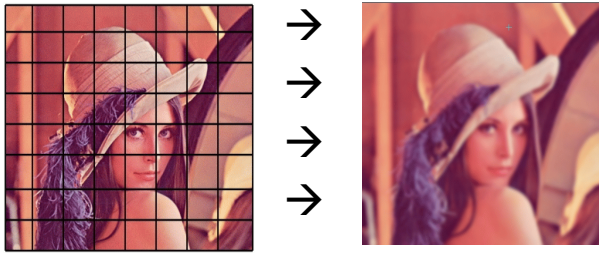


Figure 2. Convolution algorithms, like *blurring*, can be easily partitioned into independent tasks.

4.1.1 Convolution

Convolution is an image processing application in which an image is filtered by some mask (e.g. blur, distort, sharpen). Students generally have seen this process in action though playing with various filters in Photoshop, although they have no idea how it works. They are excited to see the details and will often spend extra time learning about the different types of masks on their own.

As a CS1 assignment, the implementation deals mostly with arrays and nested loops as the fundamental concepts. The single-threaded solution has students walk a rectangular mask over every pixel in the image and use the mask values in conjunction with the image values to compute a new value for the pixel. The new value is then stored in a modified image that is created as the result of the convolution.

When adding concurrency, we want the students to determine which parts of the computation can be performed in parallel. For convolution, it turns out that each pixel computation can be performed independently. Students quickly recognize that each pixel can be processed in parallel with no dependencies between them.

There is a small amount of task assignment to threads that needs to occur in this assignment as well. In theory, each pixel could be done in its own thread in parallel. However breaking it down to this level produces a large overabundance of threads – over 300,000 for a 640x480 image! Since we are only working with dual and quad-core processors at the moment, it makes more sense if we break the image up into multiple rectangular sized chunks instead. In general, matching the number of chunks to the number of processors in the core would be an ideal solution, as it keeps the cores busy with the least amount of thread creating overhead.

4.1.2 Encryption Cracking

Security topics have become important in computer science education in recent years. We have previously found ways to bring some of these topics into our introductory courses. One of the assignments we use has students build a simple version of the RSA encryption algorithm. While the mathematics behind the algorithm are difficult to understand, the algorithm itself that computes the values necessary for encryption is remarkably easy to implement. This gives us the chance to discuss some high-level issues involved in security such as key size choice and the general concept of exponential algorithms.

While we will not go into all the details of RSA here, the basic idea is that one must find the prime factors of a large number. These prime numbers could be anywhere in a large range of numbers. The top end of the range is dependent on the size of the

number being factored. And that is in turn is dependent on the key length.

As part of this assignment, we include an RSA cracking task to illustrate the impact of key size on cracking time. The assignment involves attempting to crack a given RSA key. Students produce keys of various lengths and then measure how long their crackers take to find the key. By increasing the key length, we can illustrate the nature of exponential algorithms to the students.

Once the single-threaded version is written, students begin the process of decomposition. After a little looking it becomes clear that each attempt at factoring is independent of every other attempt. Thus, all attempts at factorization can be done in parallel, with one of them producing a positive result. Note, again, that all these tasks are completely independent and there is no orchestration necessary in terms of scheduling or posting of results. Like the convolution assignment, there is just a little bit of assignment of tasks to threads that needs to be done. As the range of numbers increases it makes the most sense to divide up the range into a fixed number of sub-ranges, ideally matching the number of processors with which you are working.

4.2 Reinforcing Concurrency in CS2

During this course, we begin to introduce the students to a bit more orchestration of concurrent processes. We delay significant implementation of more advanced orchestration topics until at least sophomore year.

4.2.1 Raytracing

Raytracers are fairly straightforward to understand and are a wonderful application of recursion. One simulates shooting a ray through each pixel in the image under construction (i.e. the film) and out into the 3D world behind the image. The ray eventually intersects some object in the world. It picks up some of the objects color and then bounces off that object and continues in the same fashion recursively until some set number of bounces has been reached or it shoots off into space. There is obviously considerable physics involved in modeling of the rays and color components, but in terms of concurrency we can ignore those details.

In analyzing this problem we find that each of the rays is independent of each other. Because of this, there is again very little in the way of orchestration needed. Task-to-thread assignment is straightforward as well in that each ray can be handled by its own thread or a number of rays can be handled by the same thread. These are all similar choices as in the examples given for CS1, only the underlying problem being solved is harder because it involves recursion.

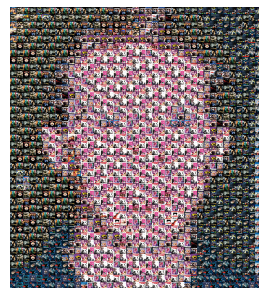


Figure 3. One of the authors immortalized in an image mosaic.

4.2.2 Image Mosaics

We have used the image mosaics [5] assignment several times and the students find it motivates their creative spirit. The assignment asks students to take a large image and first break it into small chunks. Then each small chunk is replaced with a thumbnail image from another source. For example, you might use thumbnail images of things that smell bad to replace sections of an image of your professor's face. The key being that each thumbnail needs to approximately match the same colors found in the chunk it is replacing. Thus, an algorithm is run to determine which thumbnails best match the color of the chunk being replaced. When completely done, it will still look like the original image from far away, but from up close the small images can be seen (like a mosaic).

There is obviously some setup work in sizing and cataloging the "average" color of the thumbnails. But after that, students can simply go through the chunks of the original image and run the color matching algorithm on each to obtain a local best match. This is the simplest version of this program.

Again, each small chunk of the image can be run independently and thus in its own thread. This essentially decides the assignment step as well. There is almost no orchestration involved either as long as we are working with the simplest version of the program.

However, this version of the assignment sometimes produces mosaics that look blocky as large areas of the original image are often replaced with the same thumbnail. To prevent this from happening we need to make sure that each thumbnail is used only once in the image. If the best match is already taken then we can pick the next best match, etc.

This affects the concurrent parts of the program by making the thumbnail selections not completely independent from one another. The threads can still be run without knowledge of the other threads until it is time to make a final selection. Then there must be some final orchestration of the selections to ensure there are no duplicates. The easiest way to approach this is to create another class that is in charge of making the final selections. We call this the "selector". The threads are no longer in charge of making the final selection of thumbnail, but instead need to produce a ranked list of the best thumbnails for each chunk. The selector then simply waits for all the threads to report in with their ranked lists. Once all have responded, it is up to the selector to ensure that highly ranked thumbnails are picked and that no two thumbnails are the same. There are several different version of how the selector might go about doing such as task, but that is beyond the scope of this paper.

Also note that we setup the solution in this manner so as to avoid any race conditions. The threads could potentially make their own selections rather than return a ranked list. In this case, they would need to check with some clearing house object to make sure the one they were selecting was not already taken. The problem this gets us into is that multiple threads might be accessing this clearing house object at the same time. Some will simply be reading values, but other changing values. This can lead to race conditions if not properly handled. Java does have an easy to use synchronization primitive to handle this case, but there is some thought that must go into exactly what to synchronize. We have not given this version of the assignment as we wanted to avoid the race condition, but it is certainly an option

if instructors want to introduce these synchronization methods in this assignment.

4.3 Concurrency Limitations in Algorithms

As students mature they can begin to handle more complex designs and abstract thought processes. Our Algorithms class is taken in the sophomore year and we find that this is usually when students are ready to start tackling more abstract ideas. This class is the perfect place to introduce more complex issues in parallelism as well. In particular, we start to deal with the orchestration issues that were ignored the previous year.

Traditionally, an algorithms course might cover parallel algorithms towards the end of the term in a special unit, if at all. Our approach is not to create or expand a section on parallel algorithms, but instead focus on how to integrate parallelism into the other sections of the course. One place we have found that works particularly well is the topic of dynamic programming.

Dynamic programming naturally lends itself to parallel implementation since in the building of tables from the bottom up each entry only requires some small number of other table entries to be completed. One of the common topics discussed in single-thread dynamic programming is how to traverse the table structure so you can get from what is known initially (i.e. base conditions) to the solution position. This needs to be done in such a way so that each time an entry in the table is filled, all the required sub-values have already been computed. This is not a serious issue for 1d tables as they are normally handled linearly, but comes into play with 2d and 3d table structures.

Computing table values in parallel is a simple extension of this discussion. However, there are also some definite orchestration issues involved in making sure the threads wait until they have the data necessary before they run.

We present an example below of using a dynamic programming chained matrix multiplication algorithm to introduce these concepts.

4.3.1 Chained Matrix Multiplication

Chained matrix multiplication is the problem of figuring out the best way to parenthesize a sequence of matrix multiplication such that the minimal number of scalar additions and multiplications are performed. This is a standard optimization problem and has a well-known dynamic programming solution.

The single-threaded version of this problem is a standard algorithms assignment and will not be revisited here. The important portion is that a 2d table of values is constructed. As seen in Figure 4a, the initial known values (all 0s) are along the main diagonal. The value we want to obtain is in the upper right (location 1-6). Each entry depends on the values below and to the left of it. This can be seen in Figure 4c, where the number 64 in box 1-3 is dependent on boxes 1-1, 1-2, 2-3, and 3-3. To reach the goal position, we therefore only need to fill in the upper-right half of the matrix. Thus, in the single-threaded version of the solution, one normally proceeds down each diagonal of the upper triangle one at a time until you reach the upper-right corner. See Figure 4a-d for the standard progression.

	1	2	3	4	5	6
1	0					
2		0				
3			0			
4				0		
5					0	
6						0

(a)

	1	2	3	4	5	6
1	0	30				
2		0	24			
3			0	72		
4				0	168	
5					0	336
6						0

(b)

	1	2	3	4	5	6
1	0	30	64			
2		0	24			
3			0	72		
4				0	168	
5					0	336
6						0

(c)

	1	2	3	4	5	6
1	0	30	64	132	226	348
2		0	24	72	156	268
3			0	72	198	366
4				0	168	392
5					0	336
6						0

(d)

Figure 4: The progression of a standard table building for chained matrix multiplication.

Obviously, the table entries can be computed concurrently. However, many of the entries have dependencies upon each other. Certain ones can still be computed concurrently as long as they have the required data to do so. This requires significant orchestration of the threads. We have worked with two different orchestration approaches.

The easiest orchestration method is to compute all elements in a diagonal in parallel as none of them will depend on each other. All the sub-values they need have already been computed in the previous diagonals. Then once all the elements in that diagonal have been computed, proceed to the next diagonal. This is a simple addition to the standard method that proceeds diagonal by diagonal. It requires little addition to the code in terms of synchronization as we only need to wait until all our current threads for the diagonal have finished before we fire up the next set of threads. This approach is conceptually simple, easy to implement, and yet makes use of multicore processors effectively.

A second approach is more complex. Each element knows the other elements it depends on to perform its computation. The trick is to block computation until all necessary elements are ready. While this task can be implemented with Java's basic *wait* and *notify* methods, it is not straightforward and we have found students struggle with this implementation. However, Java now provides several additional synchronization primitives such as *counting semaphores* and *blocking queues*. Counting semaphores can be used to simply control the synchronization in the chained matrix multiplication problem. Each element is guarded by a semaphore which allows no access at the start. That is, when another element tries to acquire its value, it will block automatically. When the element's value is finally computed, it then simply releases permits which allow all blocked threads to proceed and obtain its value. The addition of these more advanced synchronization primitives makes this problem as well as many others more accessible to students at an early point in their education.

4.4 Operating Systems

The operating systems course is where complex synchronization concepts are traditionally discussed in a computer science curriculum. However, the focus is usually on the details of primitives – why they are needed, what situations call for which approaches, and how they are implemented. There is not normally any focus on the parallel programming aspects of these synchronization devices.

We do not provide more parallel programming content in operating systems than we did previously. In fact, the effect is that some of the synchronization content can be compressed. While we still need to cover the trickier synchronization issues in this course, the amount of time needed on these topics is reduced since the students have already been exposed to several of the concurrency constructs. This frees up time to work with other interesting topics, such as distributed systems or virtual machines.

5. CONCLUSIONS

The goal of our curriculum changes is to provide our students with the skills they will need to succeed as we enter the multicore era. By giving our students a clear process for exploiting parallelism and practice at implementing it, they have gained confidence in taking advantage of new hardware. The integration of these topics into standard CS course topics has also caused our students to start thinking about concurrency in all of their programs, not just those given with an explicit directive to parallelize. In a world where every machine is a multiprocessor, we consider it a good trait for our students to be always thinking about ways to take advantage of concurrency.

6. REFERENCES

- [1] Vikas Agarwal, M.S. Hrishikesh, Stephen Keckler, Doug Burger. "Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures." Appears in *the Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA)*, 2000.
- [2] Krste Asanovic et al. *The Landscape of Parallel Computing Research: A View from Berkeley*. Electrical Engineering and Computer Sciences, University of California, Berkeley. Technical Report No. UCB/EECS-2006-183, Dec. 18, 2006; www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html.
- [3] David E. Culler and Jaswinder Pal Singh. *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaufmann Publishers, Inc., 1999.
- [4] D Geer. "Chip makers turn to multicore processors". *IEEE Computer* 38, 5, May 2005.
- [5] Richard E. Pattis. "Photomosaics." Appears in *the Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*, 2005.
- [6] Sriram Vangal et al. "An 80-Tile 1.28TFLOPS Network-on-Chip in 65nm CMOS." Appears in *the Proceedings of the 2007 International Solid State Circuits Conference (ISSCC)*, 2007.