The Challenges of Parallelism

Curt Hill

Mathematics and Computer Science Department

Valley City State University

101 College St SW

Valley City, ND 58072

701 845-7103

Curt.Hill@vcsu.edu

Abstract:

This paper is a call to give increased attention to parallel programming in the Computer Science Curricula. This emphasis needs to occur in comparatively low level classes considering our historical treatment of concurrent programming. Moreover, it should not be an optional topic, but every student should have sufficient exposure to the issues.
The software industry is presented with a public relations problem. The accustomed increase of computer speed will only be sustained by the used of further parallelism, including multi-core processors. The current generation of software is poorly equipped to exploit the soon to arrive multi-core processors. The public will know that their machine is faster but see no performance increase and justifiably blame the software manufacturers. Hence, the public relations problem will eventually arrive at the doorstep of Computer Science educators. The time to do something about this is now.
The paper discusses the historical reasons for the scant emphasis on parallel programming and considers some of these types. Certain types of parallelism, such as Single Instruction Multiple Data, has been important for the solution of certain types of problems, but will not be of much help in this case, while multiple thread programming will become very important. The paper then asserts that the advent of multi-core CPUs in desktop and server computers alike will obligate a change and document that these CPUs will become common in 2005.
The paper claims that part of the problem in teaching parallelism in the undergraduate curriculum is the support given in various programming languages. Therefore, several common programming languages as well as supplemental libraries are briefly considered in regards to this support. Concurrent programming is also complicated by the new types of errors that are added to the testing problem. These types of problems will also be considered.
The paper concludes with some recommendations concerning programming languages and the curricula.

Introduction.

Like any discipline, the advancements in Computer Science occur when someone has a good idea to solve a problem. That idea may be completely new or the collation of previous work in a new form. However, the importance of the problems to motivate these ideas should not be underestimated. The area of Programming Languages serves as an example in this area. The problem is a tremendous resource that is difficult to use in its native form. Hence, numerous programming languages have been proposed that make the exploitation of the resource much easier. The sub-problems include parsing, code generation and the like. Operating Systems is a different kind of a solution to nearly the same problem.

Mainly this problem has not changed much since the 1940s. The only substantial change is the speed and capability of the computer has increased dramatically. The justly famous Moore's Law has been in effect for several decades now and the advent of the microprocessor was not the beginning of the increase of speed and capability. This increase in capacity has allowed solutions to change in that we now tolerate much more overhead than formerly. Imagine the scenario if Java had been proposed in the 1970s rather than the 1990s. Although its class structure would have been well ahead of its time, the whole concept of interpreting a general purpose language in that day would have killed the project. Java is clearly the result of a time when CPU cycles are inexpensive.

One thing that has not changed is the dominance of the single processor system. However, this too will pass and very soon now[1]. As Computer Science Educators we can clearly see the change in the near future and now is the time to start preparing for the inevitable. This paper is largely a call to action.

The Triumph of Parallelism.

The bulk of our efforts have been devoted to sequential problems, because our machines have been largely sequential. A variety of parallel approaches dating back to the 1960s have been attempted some with considerable success. However, these have comprised a very small part of the discipline as these machines have comprised a small part of the CPU population.

Hidden parallelism crept into each generation of processors as the speed increases that could be accomplished in other ways diminished. One example of this is transformation of the Intel 80386 to the Pentium. The 80386 was essentially a simple processor. The 80486 introduced caching but received most of its performance improvements from a decrease in component sizes. The Pentium then was a fully pipelined super scalar device.

These changes have not had a large effect in the curriculum since they are mostly transparent to most programming. We have dealt with pipelining and caching in an architecture course or the like. There has also been an improvement in compiler design to

optimize output code for a pipelined machine, but even these changes were usually at the graduate level. The undergraduate curriculum was still mostly safe.

The advent of the World Wide Web promoted the use of multi-processor servers and distributed systems. Very early in the Pentium product line the processors could be easily configured into a multi-processor system that shared memory. However, like pipelined processors this did not have a large effect in the undergraduate curriculum. The hardest hit class was the database class. A DBMS is typical server system with large amounts of traffic and even on a uni-processor it must deal with separate requests independently. Often this was the first exposure to concurrent processing for the undergraduates students.

The advent of Computational Science seems to be the simulations of the nuclear explosions in government research labs. However, since that time simulations of complex natural systems has become increasingly important. This in turn has been the driver for even larger computing systems, including grid computing and distributed computing. One important project that demonstrated the concept was the Search for Extra Terrestrial Intelligence. It developed a screen saver that was able to exploit countless idle personal computer cycles for the processing of the vast quantities of data that it had accumulated. This was a success because it cost the user nothing and made them part of what was seen as a noble effort. SETI@Home has consumed in excess of 6×10^{21} floating point operations[2]. It also demonstrated a proof of concept that has been followed by several other projects. However, it also demonstrated in practice what we were seldom teaching in the classroom. This was not necessarily a problem, how many programmers do we need for such projects?

Most of these advances could be marginalized. This may be seen rather clearly in the 2001 Computing Curricula of ACM and IEEE [3]. Parallel programming is not mentioned at all in the Programming Fundamentals core, it is only optional and last topic in the Algorithms and Complexity core and gets somewhat more attention in the Architecture and Organization core. This observation of light coverage is not intended as a criticism of the Joint Task Force on Computing Curricula. At the time of publication the report reasonably reflects the state of the art.

However, our dear friend Moore's Law was on a collision course with the laws of physics, with no doubt about the winner. The increase in speed due to the decrease in size caused an increase in the production of heat. This was no surprise to the integrated circuit manufacturers, but the press releases started in 2004. AMD [4] demonstrated a dual-core CPU and anticipates delivery in mid to late 2005. The reigning champion, Intel [5] threw in the towel as well, announcing that it was to implement multi-core processors throughout their product line. The determination is that a multi-core processor could deliver more speed with less heat. Both of these manufacturers will offer these products for the desktop market as well as the server market.

This causes the software industry a public relations problem, which will become a Computer Science problem. The hardware manufacturers will say that their next generation of products is still in conformance with Moore's Law, but the consumers will

say that their new PC is not substantially faster than their last one. They will both be right. One important problem is that the current crop of software is not up to the task of exploiting the parallelism in such machines. (Another of the problems is that disks and memory have not increased at nearly the rate of the CPU, but that problem is not the focus of this paper.) We do not have to wait for the multi-core processors to appear in desktops, the Hyper-Threading processors are already demonstrating it. Since this is not a server problem but a desktop problem it becomes increasingly clear that the exploitation of parallelism needs to appear earlier and more often in the undergraduate curriculum.

The Language Problem of Parallelism

One of the problems presented by parallelism includes expressing multiple threads of execution in our current programming languages. Since parallelism has not been a burning issue, most programming language designs have tended to ignore the problem. Now consider programming language support for multiple threads of execution.

Java and Ada seem to have the most thorough approach to concurrency among popular languages. The syntax of Java has only the *synchronized* keyword in the basic language but standard class libraries provide most of the concurrent capabilities. The Remote Method Interface gives capabilities for distributed computing and the Thread class and Runnable interface provide the parallelism. The Java Virtual Machine is itself multi-threaded so there is a good capability for exploiting multiple CPUs. The Java approach seems to be well thought out, which makes Java a good choice of language for instruction in concurrent programming. However, Java has not gained preeminence for high performance computing for a variety of reasons.

Ada concurrency is largely built into the language itself, with a significant portion of the syntax devoted to concurrency issues. These include the reserved words task, entry, accept, requeue, and delay. This provides for multiple threads and synchronization between these threads. The most significant problem with Ada is the lack of general acceptance. It has many features to commend it but neither the industry nor academia have made dramatic moves towards the language.

Most programming languages have no real provision for parallel execution. C and C++ are typical. The default approach is to merely give access to the application program interfaces of the underlying operating system. This forces the language to be dependent on the concurrency model presented by the operating system and prevents machine independence. Platform independence is one of the reasons to use high level languages in the first place. However, it should be noted that neither language provides for files either, leaving that to the libraries as well.

There are several parallel extensions to C or C++ that have gained some following. These include OpenMP and HPC++ and these two will be briefly considered.

The OpenMP[6] system enhances the use of C or C++ with specialized *#pragma* statements. These *#pragma* statements give a parallel interpretation to otherwise sequential statements. These may be implemented by modification of the preprocessor

which is generally much easier than the compiler itself. Moreover the language definition of C insists that a *#pragma* statement not understood by a compiler is merely ignored. Thus, an OpenMP compliant program is merely sequential on a non-compliant system. The OpenMP system uses a fork-join parallelism model, which works well for large array manipulation. However, this may not be the best model for exploiting the parallelism of future desktop computers.

HPC++[7] takes a somewhat different approach. It is a class library that provides multiple thread as well as distributed computation capabilities. The thread model is based upon the Java Thread class and Runnable interface. The code is designed for a variety of UNIX systems, but since the source code is provided it may be ported to any C++ environment. This project appears to currently be inactive.

Both OpenMP and HPC++ were designed for somewhat specialized computations, usually involving large-scale numeric calculations, simulations and the like. This is not the domain of the single desktop, but the multi-threading capability is what is required to exploit the desktop of the future.

The Errors of Parallelism

Concurrent programs have all the usual problems of sequential programming, but add some new ones as well. These problems include race errors and deadlocks. Unfortunately, both of these may be non-deterministic, since they are sensitive to machine load and other unpredictable factors. In summary the program may run the same test data multiple times and give differing answers. This can be particularly frustrating when the use of a debugger causes the errors to cease, since this also affects timing. The testing phase is challenging enough without the introduction of heisenbugs, that only manifest themselves sporadically. A review of these two types of errors will be considered next.

A race error occurs when two or more threads update a variable without any serialization mechanism. Suppose that two threads desire to add different values to the same shared variable:

Thread A: `share = share + 2;`

Thread B: `share = share + 5;`

Figure 1 shows typical X86 machine language.

Line number	Thread A	Thread B
1	`;share = share + 2;`	`;share = share + 5;`
2	`mov  eax,share`	`mov  eax,share`
3	`add eax,2`	`add eax,5`
4	`mov share,eax`	`mov share,eax`

Figure 1. Machine language demonstrating a race error.

This code may result in three possible results in variable `share`. The desired effect is for `share` to increase by 7, which would be the most frequent occurrence. However, the add of 5 may be lost if Thread A executes line 2 and then is suspended, at which time Thread B executes lines 2-4. When Thread A resumes it's EAX register holds a value that it not updated. Reversing the roles of A and B produces an update of `share` increasing it by only 2. This effect may be seen on any single CPU computer where an update requires multiple machine language instructions. In this case single instructions, such as these, are generally indivisible. On hyper-threading machines and multi-core processors this error may occur on even single instructions. If the multiple processors have separate caches for data the problem is exacerbated. Moreover, if the update is to a file or database the problem is also compounded by the long instruction sequences and I/O delays.

There are a number of ways to resolve such a problem. Such a portion of code is a critical section and solving this problem requires serializing the two pieces of code so that when one starts the sequence the other is effectively prevented from starting until the first is finished. A number of mechanisms have been suggested for this serialization, including semaphores and monitors.

Java has a synchronized reserved word[8] that may be added to a method declaration, which forces threads to serialize as they call the method. Hence, the method is presumed to be a critical section. This technique would not help the above example unless both pieces of code were in the same synchronized method. In that case a synchronized statement obtains a lock on an object and then executes a block of code. If the object is locked by another thread this instance is suspended until the previous lock is released.

A deadlock situation occurs when two threads each have a lock on some resource.

Thread A	Thread B

`synchronize(s){`
`  statement 1b;`
`  synchronize(t){`
`  …`
`  } // inner`
`  } //outer`
	`synchronize(t){`
	`  statement 1b;`
	`  synchronize(s){`
	`  …`
	`  } // inner`
	`  } //outer`

Figure 2. Recipe for deadlock.

In figure 2 a deadlock situation becomes possible. If both threads can start statement 1 before the other arrives at the second synchronize then a deadlock situation occurs. Each holds one of two locks and then asks for the remaining. The delightful part of concurrency is that in most cases this will very likely work without a problem. However, as the timing changes for subtle and unpredictable reasons the threads freeze.

Most operating systems provide this type of functionality through their APIs as well as others. However the use of APIs may be efficient in terms of machine resources but diminishes the portability to other platforms.

Recommendations.

The introductory programming class should be taught with a language possessing the following characteristics:

- Simple enough for introductory students.
- Contains the features that should be exposed to students. This paper argues that concurrency is one of many such features.
- Good acceptance within the industry.
- Standard versions are available on all platforms.

Unfortunately, no such language currently exists! Instead, Computer Science departments and instructors wrestle with the question of what is the lesser of the possible evils. Therefore it seems that those who design languages still have work to do.

Given the reality of no perfect language the early programming curriculum must be changed to incorporate parallelism. Even in a language with no standard support for concurrency, such as C++, it is far better to use the operating system's APIs then to miss the topic altogether. Languages, like Java and Ada which have adequate facilities have even less excuse. It is typical for the first programming class to be rather full, but this topic needs to be considered very shortly thereafter. In the future concurrency may be as important a topic as recursion, so needs to be covered early and often.

References.

[1] Fordahl, Matthew (2004). Next-generation computer chip to hold 2 engines. http://seattlepi.nwsource.com/business/205282_yechips27.html. Date accessed 28 December 2004.

[2] SETI (2004). Current statistics. http://setiathome.ssl.berkeley.edu/totals.html. Date accessed 24 December 2004.

[3] Chang, Carl and Peter J. Denning (2001). Computing Curricula 2001 Computer Science. Final Report of the Joint Task Force on Computing Curricula. http://www.computer.org/education/cc2001/final/index.htm Date accessed 22 December 2004.

[4] AMD (2004). AMD Demonstrates World's First x86 Dual-Core Processor. http://www.amd.com.hk/us-en/0,,3715_11787,00.html. Date accessed 22 December 2004.

[5] Jajeh, Daniel P (2004). Intel accelerates in a new direction. http://www.intel.com/employee/retiree/circuit/righthandturn.htm Date accessed 22 December 2004.

[6] OpenMP (2002). OpenMP C and C++ Application Program Interface. http://www.openmp.org/specs/mp-documents/cspec20.pdf. Date accessed 10 March 2005.

[7] Extreme! Computing (1999). High Performance C++. http://www.extreme.indiana.edu/hpc++/index.html. Date accessed 22 December 2004.
[8] Sun Microsystems (2004). JavaTM 2 SDK, Standard Edition, Documentation. Version 1.4.1. http://java.sun.com/products/archive/j2se/1.4.1_07/ Date accessed 10 March 2005.