Parallel Computing

Daniel S. Priece

Villanova University
Computer Science Department
Villanova, PA 19085
daniel.priece@villanova.edu

Abstract

The rapid rate at which technology is evolving has led to a need for more sophisticated techniques that can take full advantage of the hardware available. Traditionally, software has lagged behind hardware and there is a continuous need to maximize software performance on multiprocessor systems. This paper reviews the fundamentals of parallel computing with a focus on implicit, or automatic, parallelization. We discuss different software techniques as well as restrictions that arise when creating parallel code.

1. Introduction

Parallel computing involves the use of several processes working together on a single set of code as the same time [1]. The purpose of exploring parallel computing is to cut down on the execution time of processor intense code by distributing the work among several processors. Sets of instructions are run simultaneously and then synchronized back together before producing output, instead of running serially one after the other.

The very first example of parallel computing was found on a tablet dated back to 100 B.C. [2]. The tablet consisted of three calculating areas where the user was able to compute at higher speeds. It has been an idea since the earliest digital computers were built in the 40's and developed multi-processor machines were developed in the late 60's. Today the most powerful computer is able to use 20 teraflops of processing power [1]. Rapid hardware advance has created a need to write faster and better performing software. The code that one machine could execute in 8 hours could be executed on a parallel system of 8 processors in one hour greatly reducing the actual run time, with each processor running a smaller section of the input code. Each processor is doing the same amount of work as the single processor machine but they each processor could be running a different section of the code at a time.

2. Parallel Hardware Setups

Today there are four standard setups for parallel systems differing in how the memory and communication between processors are handled [1].

The first is a shared-memory multiprocessor setup which is found in a standard multi-processor machine. The machine uses multiple processors for computing while sharing one common memory area. There is a single set of memory addresses that every

processor knows so that no copying and moving of data is required. This method is easier for programmers but because the data can be updated by one processor, if another processor is not aware of the data change it could cause problems with output.

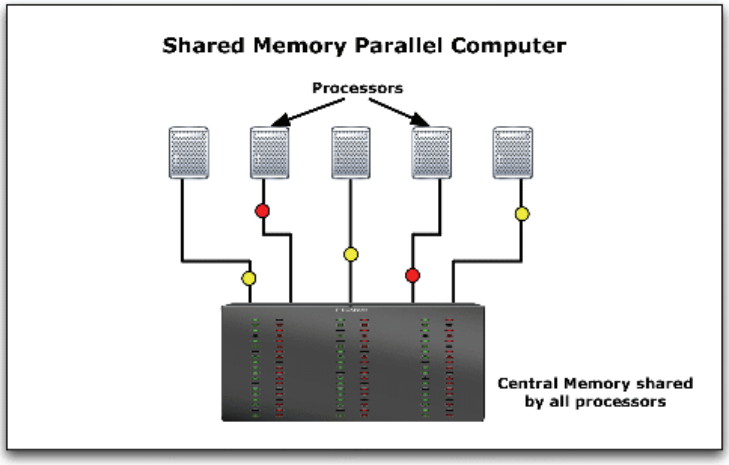

Figure 1: Shared Memory Multi-processor [14]

The next most commonly thought of parallel system is network computing. Network computing is more commonly referred to as a large network system usually operating over an internet connection. The problem with network computing in this fashion is that different computers have different processor and internet setups thus making the computations both inefficient and slow. A slow connection between computers creates a problem for an efficient parallel system. Network computing is also referred to as a distributed memory, multi-computer system.

A smaller computer network, known as cluster computers, work over a local area network connection and although they are faster than a larger internet based network, they are still limited by their communication speeds. The basic idea is still the same and is easier to manage than network computing due to the smaller number of computers in the system and fast, efficient connections between cluster computers.

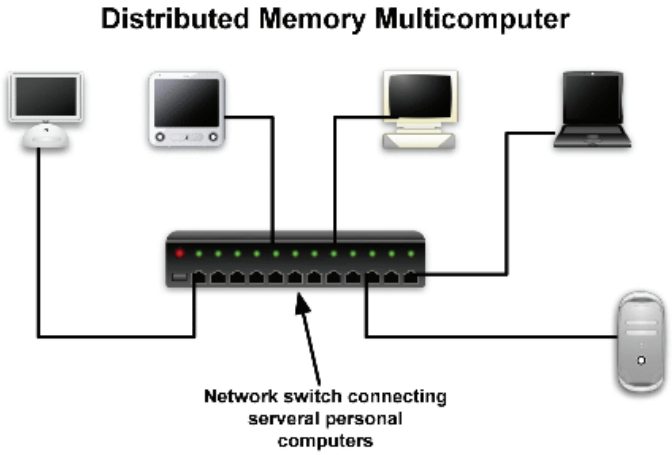

Figure 2: Network Computing [14]

The final type of parallel system is a parallel vector processor. A vector processor is a type of CPU that is able to run simultaneous operations [3]. A standard serial processor is known as a scalar processor. Instead of processing just instructions in its pipeline, a vector computer processes the data as well as the instructions. For example, if a scalar processor is used to add a group of ten numbers to another group of ten numbers, a loop would be used to add each number to the next number then add that number to the next and so on until all the numbers have been added. Instead of taking in only two numbers at a time, a vector processor grabs the first set of numbers and the second set of numbers and adds them all in a fraction of the time it takes a scalar processor. Vector processors were used for supercomputers in the past but due to high cost, they are used less today. The choice of a parallel system is important as well as the architecture of the processor and memory.

2.1 Flynn's Taxonomy

Flynn's Taxonomy provides four possible architectures for parallel computers in terms of processors and memory [4]. A standard non-parallel computer is referred to as a single instruction, single data machine because during any one clock cycle there is one instruction stream being executed using one data stream as input. Single instruction, multiple data refers to a machine that for one clock cycle, every processor of the machine executes the same code with separate inputs.

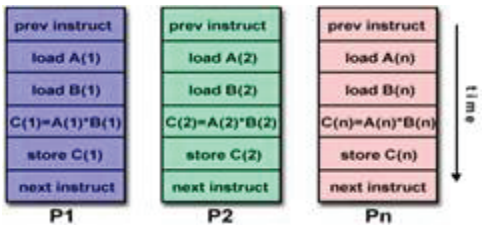

Figure 3: Single Instruction, Multiple Data Execution [4]

For example if during the first clock cycle processor 1 is executing a store instruction, then every other processor is at the same time executing a store instruction but each with different data to store. As you can see in the figure above, every processor is executing the same set of instructions however, P1 is using "1" as input, P2 is using "2" as input and P3 is using "n" as input. In an ideal situation, the processors will compute the inputs simultaneously such that they end execution at the same time and can synchronize the output. The remaining two machines are the multiple instruction, single data machines and the multiple instruction, multiple data machines which use separate instructions for each processor given a single set of data or multiple sets of data. Most modern parallel computers fall into the multiple instruction, multiple data category.

3. Programming Models

Currently there are many programming models being used, but only four of the more commonly used and less difficult models are discussed here [4]. These models are simply abstract ideas to be implemented on different hardware architectures. A shared memory model requires semaphores are used to control when a processor has access to the shared memory. This simplifies the execution in that data does not have to be sent between processors because it is all stored in a common space. Next is the threads model which allows code to have multiple execution paths by using threads to act like subroutines in a serial program. Third is the message passing model, where each process in the code has its own memory that can send and receive data from other processes. Finally, the last process is the data parallel model where processes all work on the same set of data but different partitions within that set of data.

4. Automatic Parallelism

Automatic parallelism, or implicit parallelism, involves parallelizing a program at compile time instead of implemented as in manual parallelism. An example of serial code converted to parallel code can be found in Figures 4 and 5 below.

```
npoints = 10000
circle_count = 0
do j = 1,npoints
  generate 2 random numbers
between 0 and 1
  xcoordinate = random1 ;
ycoordinate = random2
  if (xcoordinate,
ycoordinate) inside circle
  then circle_count =
circle_count + 1
end do
PI =
4.0*circle_count/npoints
```

Figure 4: Pseudo Serial Code for Computing PI [13]

```
npoints = 10000
circle_count = 0
p = number of processors
num = npoints/p
find out if I am MASTER or WORKER
do j = 1,num
  generate 2 random numbers
between 0 and 1
  xcoordinate = random1 ;
ycoordinate = random2
  if (xcoordinate, ycoordinate)
inside circle
  then circle_count =
circle_count + 1
end do
if I am MASTER
  receive from WORKER their
circle_counts
  compute PI (use MASTER and
WORKER calculations)
else if I am WORKER
send to MASTER circle_count
endif
```

Figure 5: Pseudo Parallel Code for Computing PI[13]

The set of pseudo-code above shows an algorithm for approximating PI through serial and parallel code. In figure 2, after the program inscribes a circle in a square, the serial algorithm finds the number of points in the square that are also in the circle then it divides the number of points in the circle by the total points in the square, multiplies by 4 and gets an approximation of PI. The parallel code is marked in red where changes have been made to make it parallel. In the execution of the parallel code, a master processor is setup that takes in all the information of the other processors. The worker processors each take a different section of the loop that needs computing, they each work on their share simultaneously and then send the information back to the master processor. This takes less time because more computations are being processed per clock cycle.

Because manual parallelism has many time and budget restraints, this paper explores automatic parallelism more deeply. The automation of the compiler provides a less error-prone, less time consuming, and less complex way to parallelize code.

4.1 Compilers

To understand how a parallel compiler works we will first look at a standard serial compiler which commonly consists of 7 parts [5].

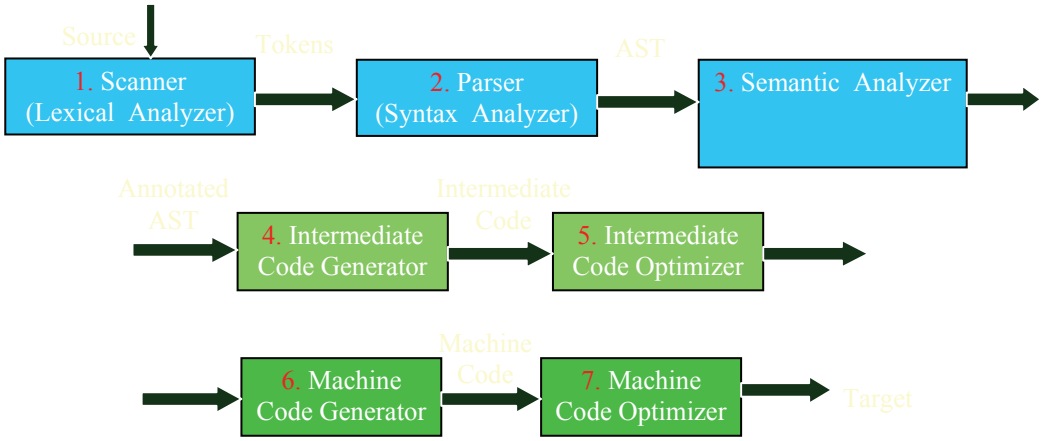

Figure 6: Compiler Diagram [5]

The first part of the compiler is the Scanner which is also known as the Lexical Analyzer. Here is where the source code is tokenized. White space is removed as well as comments and lexical errors are handled at this step. Next is the Parser. The Parser makes sure the syntax of the language is followed according to the language's context free grammar. This is where the code is tested for parallelism and separated in a parallel compiler. Then, the Semantic Analyzer handles type checking. From there an abstract syntax tree is used to generate and optimize a lower level code which is then generated and optimized into machine code.

4.2 Problems to Consider

When designing any kind of parallel compiler a few problems must be considered [6]. Latency is the time it takes to send a zero length message from one processor to another. High latency will require more planning in sending messages. Because of latency, smaller sized messages should be clustered together before sending in order to be more efficient. Bandwidth refers to the speed at which data is transferred between processors during communication. Lower bandwidth would call for as little communication as possible to be optimal. Determining which tasks have to communicate to each other is a problem called scope. All communication has to be synchronized as well which means synchronization points need to be created to optimize all the processing power and not have any processes waiting for others to finish. One major problem to consider when creating parallel code is dependencies.

Dependencies within code are the largest problems for parallelism and therefore to look at dependencies within code we use a data dependency graph.

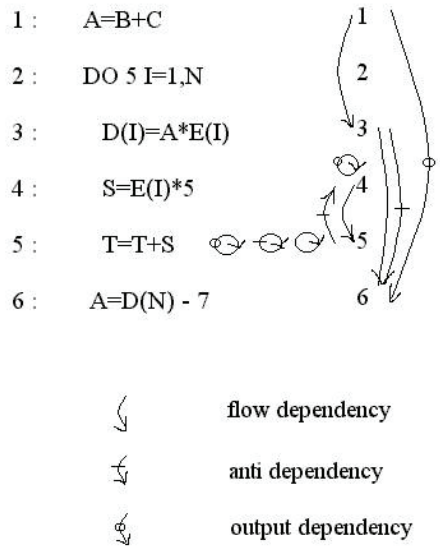

Figure 7: Dependency Graph [7]

Flow dependencies are ones where there is an execution path between two statements. In a flow dependency, a value is written and then read. An output dependency is where the same memory location is being accessed and written to more than once. Anti dependency is when a value of a variable is read and then written [8]. From a dependency graph we could optimize the dependencies so that the code can be as parallel as possible. With the remaining dependencies it is necessary to use synchronization points to optimize parallelism.

There are a few methods that have been implemented to get around the problem of dependencies [9]. Some values of a program are not known until run-time which makes splitting code difficult. For small network setups, multiple path allocation is a good method to use. It involves scheduling every possible path that the data could take at compile-time and then run the appropriate one at run-time. This obviously becomes very inefficient for larger numbers of processors because as processors are added, path allocation greatly increases. Another technique is to use a single data stream instead of multiple streams between processors. This is called dynamic destination selection. The single stream carries all of the data which is marked by the destination that the data is going to. This way the run-time systems are able to quickly send data that is dependent to the same place.

5. Related Topics

There are many compilers, systems, and tools already available that use parallel computing such as the popular systems and tool, BOINC and TPL.

5.1 BOINC

According to their website, “the [Berkeley Open Infrastructure for Network Computing] is a software platform for volunteer computing and desktop grid computing” [10]. This meaning that users volunteer their processor’s computing power over an anonymous connection or a small private network. Projects have been developed like SETI@home where anyone can sign up and use their home computer’s down time to do large amounts of parallel processing helping the search for extraterrestrial intelligence.

5.2 TPL

There are many tools available to programmers such as Microsoft’s Task Parallel Library [11] which provides methods for parallelism. Microsoft shows the efficiency of the TPL Parallel.For method for loops versus the .NET ThreadPool class where as the number of processors increase, the efficiency of the TPL increases.

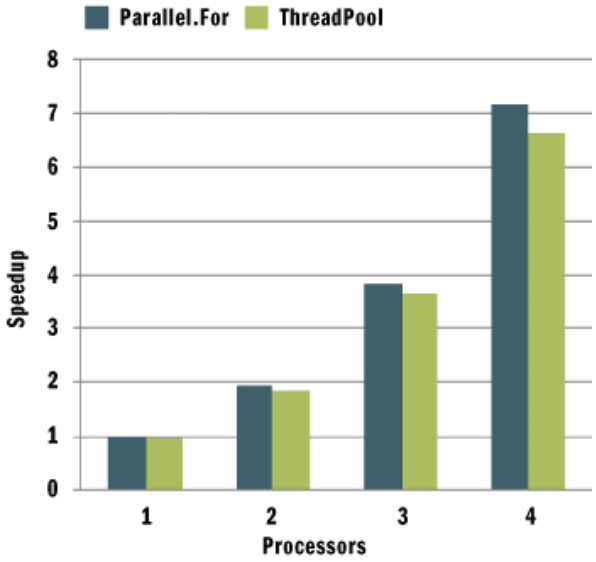

Figure 8: Efficiency Graph [11]

One example that Microsoft gives for their TPL uses ray tracing. Ray tracing is a way to produce photorealistic renderings which requires much computation [11]. In ray tracing, a path is traced from what an imaginary eye would look at and the lighting is affected accordingly. Each path of pixels is called a ray.

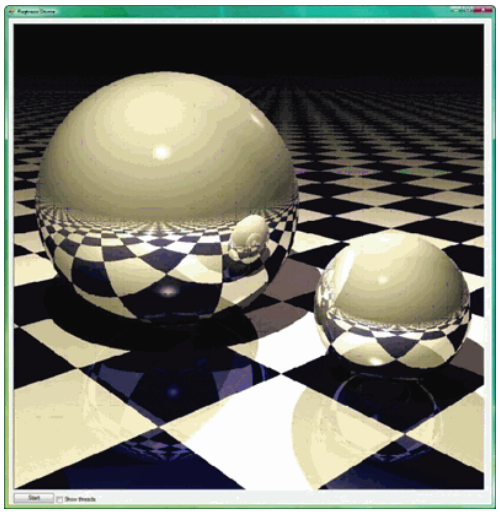

Figure 9: Rendered Image [11]

Figure 5 is an image from a rendering using the ray tracing method and the Microsoft TPL. The result was a seven times speed up on an eight processor machine from 1.7 frames per second to 12 frames per second in its animations.

6. Proposed Work

Since compilers are written to optimize code for the type of processor, it is evident that despite much of parallel optimization being done at a high level, low level optimization is more effective. Optimization of parallel compilers has reached the point where the number of processors does not matter to the compiler rather it is important at run time [15]. If there existed a compiler that could parallelize code at a lower level, then optimization across different processor types could be increased dramatically.

Dependency graphs are generated in the parser stage of a compiler. Intermediate code then optimizes the newly parallelized code and generates machine code. However, if an intermediate code generation could be used between different processor types that produces parallel code, the final machine code could be more efficiently generated parallel code because the generation and optimization is taking place at a lower level.

To study this more, we would have to study high level and low level code and look at intermediate code generated by different compilers that are optimized for different processor types. From this we could determine the areas where parallel code can be generated easily. Then developing a compiler that generates the lower level parallel code, we can test the efficiency of the parallel code produced to that produced of current parallel compilers.

7. Conclusion

In starting this research, it quickly became apparent that parallel computing is a very broad field and narrowing down on a single problem would not be very simple. The majority of information that could be found was very abstract and mostly served as an introduction into the topic. In order to understand the problem that was being researched, a very extensive background knowledge had to have been developed. It was not until starting to search for information on implicit parallelism and compilers did there appear more technical sources of information. Eventually, the topic of dependencies was brought up. This is a large problem in parallelism and ways around it must be found in order to use parallelism to its full potential. In the studying of dependencies we can understand a better way to parallelize code at lower levels thus resulting in more efficient parallel code. Automatic parallelism is a shot at better future standards of programming. If there could be one standard compiler that worked over a multi-platform system and was efficient due to low level parallel code generation, manual parallelism would not be needed and the ease of understanding parallel code would be increased. With slowing hardware development, parallel software will become more popular as well as parallel systems. With a better development of automatic parallelism, new software would be simple and inexpensive to write.

References:

[1] M. Katz, G. Bruno, "Guide to Parallel Computing," CTBP & SDSC Education Module. [Online]. Available: http://ctbp.ucsd.edu/pc/html/intro1.html. [Accessed: Nov. 4, 2007].

[2] http://www.buyya.com/microkernel/chap1.pdf [Accessed: Nov. 2, 2007].

[3] "Vector Processing," Wikipedia. [Online]. Available: http://en.wikipedia.org/wiki/Vector_processor. [Accessed: Nov. 4, 2007].

[4] B. Barney, "Introduction to Parallel Computing," *Introduction to Parallel Computing*, Livermore Computing, June 2007. [Online]. Available: http://www.llnl.gov/computing/tutorials/parallel_comp. [Accessed: Sept. 11, 2007].

[5] V. Gehlot, "CSC8310 Linguistics of Programming Languages," Villanova University. [Online]. Available: http://www.csc.villanova.edu/~gehlot/8310/lec/Week2.ppt. [Accessed: Oct. 28, 2007].

[6] S. Chang, "Parallelization of Codes on the SGI Origins," NASA Ames Research Center. [Online]. Available: http://people.nas.nasa.gov/~schang/origin_parallel.html#fraction [Accessed: Sept 26, 2007].

[7] "Parallel Compiler Tutorial," Tutorial-Reports.com. [Online]. Available: http://www.tutorial-reports.com/computer-science/parallel-compiler/tutorial.php. [Accessed Oct. 28, 2007].

[8] "Data Dependency," The Everything Development Company. [Online]. Available: http://everything2.com/index.pl?node=data%20dependency. [Accessed: Oct 28, 2007].

[9] C. Metcalf, "Approaches to Data Dependency,". [Online]. Available: http://home.comcast.net/~cdmetcalf/papers/cop/node67.html. [Accessed: Nov 2, 2007].

[10] "BOINC," Berkeley Open Infrastructure for Network Computing. [Online]. Available: http://boinc.berkeley.edu/. [Accessed: Sept. 28, 2007].

[11] D Leijen and J Hall, "Optimize Managed Code For Multi-Core Machines," The Microsoft Journal for Developers, 2007. [Online]. Available: http://msdn.microsoft.com/msdnmag/issues/07/10/Futures/default.aspx [Accessed: Sept. 28, 2007].

[12] "Ray Tracing," Wikipedia. [Online]. Available: http://en.wikipedia.org/wiki/Ray_tracing . [Accessed: Nov. 4, 2007].

[13] "Introduction to Parallel Programming," Maui High Performance Computing Center. [Online]. Available: http://www.mhpcc.edu/training/workshop/parallel_intro/MAIN.html. [Accessed: Nov 6, 2007].

[14] "Parallel Computing." CFD Online. [Online]. Available: http://www.cfd-online.com/Wiki/Parallel_computing. [Accessed: Nov 19, 2007].

[15] M Lam, "Locality Optimizations for Parallel Machines," The Stanford SUIF Compiler Group, 1994. [Online]. Available: http://suif.stanford.edu/papers/lam94.ps