ARCHITECTURE-SPECIFIC ALGORITHMS FOR A PARALLEL PROGRAMMING COURSE

James M. Conrad
Department of Computer Systems Engineering
University of Arkansas, Fayetteville, AR 72701-1201
jmc3@engr.engr.uark.edu

Abstract

A senior and graduate-level parallel programming course at the University of Arkansas is described. At the request of the students, the course utilizes four different computing environments rather than just one or two. The skills students learn and the programming exercises assigned are described. The specific algorithms and the architectures used are analyzed in detail.

Keywords: *Education, Multiprocessor, Multicomputer, Parallel Algorithm.*

Introduction and Background

A senior/graduate-level course titled *Parallel Programming,* has been offered at the University of Arkansas for three years. This course is cross listed between the Computer Science, Computer Systems Engineering, and Electrical Engineering departments.

The goals of the course are to teach good problem-solving techniques and analytical skills for developing software for advanced computer architectures. Students learn how a computer's architecture affects the algorithm development and coding of engineering problems. Practical experience is provided in the form of real-world engineering and scientific problems on several parallel machines.

Prerequisites include such computer science topics as: data structures, operating systems (including semaphores), computer organization, and the C programming language. Books used in the past are by Quinn [6] and Kumar, et.al. [3]. The author of this paper created this course and has served as the only instructor.

One difficulty of teaching this graduate-level parallel programming course is the diversity of student backgrounds. Students enroll from several disciplines, including Computer Science, Computer Systems Engineering, Electrical Engineering, and Civil Engineering. Because of this diversity, the instructor cannot be assured of the students' background in algorithm development, operating systems, and programming. Therefore, some of the course effort is devoted to ensuring all students have the basic skills needed to succeed.

The biggest difficulty is that no formal algorithm course exists in the University of Arkansas undergraduate and graduate curriculum. This fact further limits the pace of the course since students do not have the strong background in sequential algorithms analysis to extend to parallel algorithm development.

Important Parallel Programming Concepts

Programming a parallel computer or a distributed network of computers for the first time is a totally different, perhaps frightening, task. The coding skill-set for parallel programming is essentially the same as sequential programming, but the reasoning and problem solving skill-set differ. Some of the skills needed to effectively solve a problem using parallel computers includes the following:

Control vs. Data Parallelism

One of the major factors which decides the efficiency of a concurrent application is the decomposition of the problem. There are two broad classifications of problem decomposition: *control parallelism* and *data parallelism* [3].

Control parallelism divides the *tasks* rather than the *data.* Some algorithms can naturally be decomposed using control parallelism. Using tree search as an example, the branches of the tree can be thought of as tasks that need to be performed.

In data parallelism, on the other hand, the input data is partitioned and assigned to different processors. The same operation is performed on a different set of data.

Shared and Locked Variables

One concept that often befuddles students is the availability of variables to several processors in a parallel computer. Once students grasp that concept, they are more receptive to the differences between shared and private variables and the use of each. Through examples, they also understand the rationale for locking certain variables while they are being updated. The simple example of having two processors update a single memory location used as the total of an array of numbers easily demonstrates this concept.

Message Passing

Once students understand the concept of shared memory, they can extrapolate to the architectural difference of distributed memory. Once they ponder the difficulties of accessing this distributed memory, they can better appreciate message passing as the method to "share" variables and control. They need to understand the ability for messages to be synchronous (the programs wait for the messages) and asynchronous (the messages arrive and are acknowledged at any time).

Speedup

Performance measurements in the class are limited to one algorithmic parameter, the speedup. *Speedup* is the effective speed of a parallel computation in operations executed per time unit of a parallel computer with p processors [1]. This is expressed as:

$$speedup = S(p) = \frac{time(sequential\ algorithm)}{time(parallel\ algorithm)} \quad (1)$$

Students measure start-to-stop time, including data loading and parallel operating system overhead, of their programs. Although other measures could be examined (i.e., efficiency, parallelizability), this would detract from the most important metric.

Beating an Algorithm to Death

Since the field of parallel computing is vast (and growing every day), only a small part of the field can be addressed in a semester graduate course. This especially involves the type of algorithms solved in class.

Although there are many important problems that can be solved using parallel computers, seniors and beginning graduate students do not have the time to devote to complex problems. The course relies on a thorough examination of several small and simple algorithms. If the students can fully understand all facets of a single sequential algorithm, they will better understand the resulting parallel implementation.

One algorithm examined in class is the "Sieve of Eratosthenes." This algorithm is featured in the parallel computing book by Quinn [6, pp. 9-19]. The "Sieve" serves as an excellent example to analyze general algorithms, as well as introducing concepts of parallel algorithms and implementations.

The algorithm, shown in Figure 1, seeks to find all prime numbers between 1 and n. It is a very simple and short algorithm, and makes an excellent example to introduce the concepts and problem-solving skills listed above. This algorithm, and improvements to the algorithm, are analyzed for several weeks. Students are happy to leave it behind.

Sieve of Eratosthenes

```
integer $n$   /* input - size of problem*/
char $array(1..n)$
       /* used to mark prime/non-prime numbers */
fill $array(1..n)$ with ones
for $i = 2$ to trunc($\sqrt{n} + 1$) do
       if ($array(i) = 1$) do begin
              $j = i**2$
              while $j <= n$ do begin
                     $array(j) = 0$
                     $j = j + i$
              end while
              end if
```

Figure 1: The Sieve of Eratosthenes Algorithm

Programming Assignments

Students solve six programming assignments using a shared memory Multiple-Instruction, Multiple-Data (MIMD) computer, a distributed memory MIMD computer, a distributed network of Sun Workstations using Parallel Virtual Machine (PVM), and a Single-Instruction, Multiple-Data (SIMD) array processor. Students vote at the beginning of the semester whether to use two, three, or four machines, and nearly unanimously choose four.

Deliverables for the assignments include the code and a report describing their results and observations.

"Sieve" on a Shared Memory MIMD Computer

The first assignment is programmed for a shared memory MIMD multiprocessor. Students examine the use of atomic operations, sharing variables, and locking variables.

Students write a program which computes the Sieve of Eratosthenes for numbers up to 1,000,000 using data parallelism techniques, then control parallelism techniques. They use one, two, four and six processors and run the programs several times on these configurations. They also record the resulting number of primes (not the specific primes) for each run and measure the time each trial takes. The total number of primes needs to be determined in parallel from the shared data variable *array*.

Student reports indicate that they notice good speedup using data parallelism but little speedup using control parallelism. This is their first indication that some implementations of algorithms depend greatly on the parallel computer architecture. They also observe that some variables (like *array*) can be updated without locking and unlocking, but some (like the final count) need to be locked.

"Sieve" on a Distributed Memory MIMD Computer

The second assignment is programmed for a distributed memory MIMD multicomputer. Students learn the use of message passing and programming for distributed data storage.

Again, students write a program which computes the Sieve of Eratosthenes for numbers up to 1,000,000 using data parallelism techniques, then control parallelism techniques. They use one, two, four and eight processors, run their program several times, record the resulting number of primes and measure the time each trial takes.

Matrix Multiply (A[n,n],B[n,n],C[n,n])

```
for $i = 0$ to $n-1$ do
    for $j = 0$ to $n-1$ do begin
        $C[i,j] = 0$
        for $i = 0$ to $n-1$ do
            $C[i,j] = C[i,j] + A[i,k] \times B[k,j]$
    end for
```

Figure 2: The Sequential Matrix Multiplication Algorithm

Student reports indicate that they notice good speedup using data parallelism. Most are never able to implement control parallelism, and thus observe that some problems cannot be decomposed effectively on some architectures. They also observe that message passing takes quite a bit of time.

Matrix Multiplication on a Shared Memory MIMD Computer

The third assignment is programmed on the multiprocessor. Students write a matrix multiplication program to compute $A \times B = C$. The sizes of A and B vary, so students must make sure their program is flexible enough read in the data, balance the work, and compute the correct sum in an efficient manner. An example of the sequential matrix multiply algorithm is shown in Figure 2.

The students must ensure the program uses dynamic load balancing. For example, the row of the C matrix should be computed by first getting the row number to work on from a common shared variable.

Matrix multiplication is a fairly simple algorithm, yet there are many ways to implement it on parallel machines which yield little or no speedup. Students learn:

- the benefits and tradeoffs of making some variables private and non-shared,
- it is best to compute local, private values and later write the computed results to the shared variable area in order to take advantage of an architecture's cache memory,
- how to malloc shared variables,
- the difference between static and dynamic assignment of work, and
- good speedup can be obtained for larger, 100 by 100 number matrix multiplication problems.

Linear Equations on a Distributed Memory MIMD Computer

The fourth problem requires the students to implement Gaussian Elimination with Partial Pivoting and back propagation using the multicomputer. They can implement either the row and column distributed versions. Again they take measurements of time and compute speedup for 4, 8, 16, and 32 processors, but this time they compete against their classmates for the best times. Completed programs are submitted electronically, and the instructor compiles and executes each submission to rank the timings.

Several algorithms to solve a system of linear equations, and specifically Gaussian Elimination, have been reported in textbooks, including parallel programming books by Quinn [6, pp. 229-237] and Kumar, et. al. [3, pp. 178-197]. These algorithms will not be detailed here.

Students often have difficulty understanding the full sequential Gaussian Elimination algorithm, so they often fail to implement the algorithm on a parallel computer successfully. They usually find their implementation exhibits a *slow-down,* or speedup values less than one.

```
PROCEDURE BT(k)
FOR Z[k] = 1 TO m[k] DO
        BEGIN
        Consistent = True;
        FOR p = 1 TO k-1 WHILE consistent DO consistent = check(p, z[p], k, z[k]);
        IF consistent THEN IF k = n THEN output(z) ELSE BT(k+1)
        END
END;
```

Figure 3: The Sequential Backtracking Algorithm

n-Queens on a Distributed Computer Network (PVM)

The fifth programming assignment is the solve the n-Queens problem for a 1 to 20 square chessboard using from 1 to 10 Sun workstations running PVM. PVM is a software package developed by Oak Ridge National Lab, the University of Tennessee at Knoxville, Emory University, and Carnegie Mellon University [2]. PVM allows a distributed network of workstations or parallel computers to appear as a single parallel computer resource. PVM uses a message-passing paradigm, and is available free on the Internet.

The n-Queens problem is to place n chess queens on an $n \times n$ chess board, such that each queen is safe from capture. In chess, queens attack in straight lines along the X, Y, and diagonal directions.

The algorithms described in class to solve the problem are the backtracking and backjumping tree search [5]. These algorithms are shown in Figures 3 and 4. Students implement one of the sequential algorithms on a single workstation, then use PVM to solve larger instances of the n-Queens problem.

Student reports indicate a solid understanding of the mechanics of distributed computing, although they note that PVM seems to be fragile. The do learn the nature of algorithms in which the larger the "parallel computer," the larger the problem can be scaled. They also notice good speedup for large problems.

Image Processing on an SIMD Computer

This sixth assignment is to take an pictorial image file, filter the image data, and write the image back out to a file using a SIMD array processor. Their program must read a binary image file 512 pixels wide by 480

```
FUNCTION BJ(k);
returndepth = 0;
FOR Z[k] = 1 TO m[k] DO
    BEGIN
    Consistent = True;
    FOR p = 1 TO k - 1 WHILE Consistent DO Consistent = check(p, z[p], k, z[k]);
    IF not Consistent THEN faildepth = p-1;
    IF Consistent THEN IF k = n THEN BEGIN output(z); faildepth = n-1 END
                                ELSE BEGIN
                                faildepth = BJ(k+1);
                                IF faildepth < k THEN  Return(faildepth)
                                END;
    returndepth = max(returndepth, faildepth)
    END;
Return(returndepth)
```

Figure 4: The Sequential Backjumping Algorithm

```
Image Filtering
int $image[1..480, 1..512], new\_image[1..480, 1..512]$
int $convol[-1..1, -1..1] =$
    $-1, -1, -1, -1, 9, -1, -1, -1, -1$

 Read in $image$ data
 Copy edges of $image$ to $new\_image$
 for $i = 2$ to 479 do
     for $j = 2$ to 511 do
         for $k = -1$ to 1 do
             for $l = -1$ to 1 do
                 $new_image[i, j] += image[k + i, l + j]$
                     $\times convol[i, j]$;
 Ensure $new\_image$ data is between 0 and 255
 Write out $new\_image$ data
```

Figure 5: The Image Filtering Algorithm

pixels deep, where each byte in the file represents one pixel, perform edge enhancement filtering, and output the new image. In order for the raw image to be viewed on a Sun workstation, a header describing the color mapping must be appended to the initial and final images. Students first implement the sequential algorithm on a Sun workstation, then implement the parallel algorithm.

The algorithm (Figure 5), described in a book by Lindley [4, pp. 367-377], involves taking each pixel from the input image and creating a new value for the pixel based on the surrounding pixel values. The high pass spatial filter HP1 serves as the 3 pixel by 3 pixel convolution kernel for the pixel neighborhood. An example of the the original image and the filtered image are shown in Figures 6 and 7.

Since the University of Arkansas has several Image Processing and Advanced Graphics courses, this program is popular. Students often spend more time than they should examining the different convolution matrices, parallelization tools, and programming environments. They find the algorithm well suited to an SIMD computer architecture.

Other Algorithms

Some other algorithms examined and programmed in the past include the Traveling Salesman Problem, sorting, and Fast Fourier Transforms. Past experi-

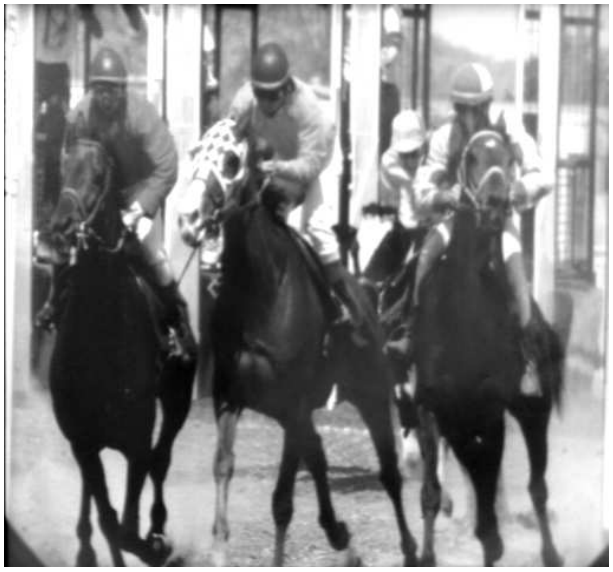

Figure 6: Original Image for Filtering

ences have shown that students can only grasp a limited number of algorithms, therefore these algorithms were set aside in favor of simpler algorithms.

Projects

The final project is a group effort, with two or three people in a group. The participants choose a commercially available computer not discussed in class and report on its hardware and software attributes. They are required to analyze its architecture as it relates to programming, its operating system, compilers, and means or lack of variable sharing. An alternative assignment is to solve an approved problem using a parallel computer.

Each group writes a paper and makes a formal presentation of their work to the class.

Past projects titles include:

- Direct Solvers for Sparse Systems of Linear Equations
- Neural Networks on SIMD Array Processors
- Processing Global Information Systems Data using PVM
- Scheduling Tasks on Distributed Computer Systems
- A Comparative Study of the n-Queens Problem Using Multiple Parallel Architectures

Observations

The students of this parallel programming class enjoy the chance to have access to several different parallel computing platforms. Small schools like the

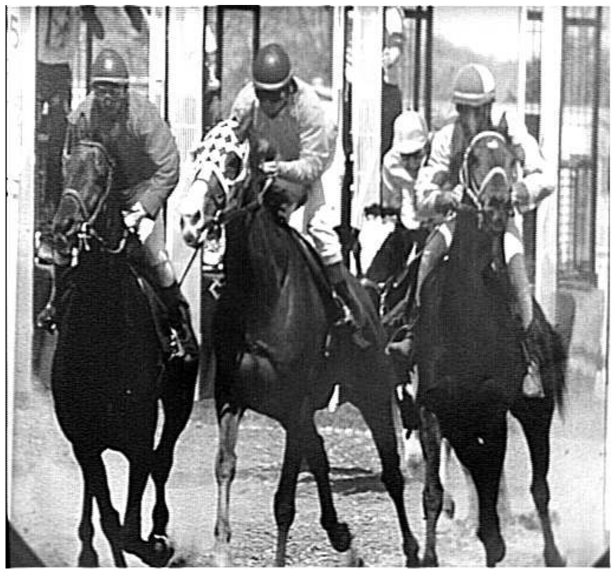

Figure 7: Filtered Image

University of Arkansas must rely on larger schools or supercomputing centers for their parallel computing needs. However, access to these platforms has become more difficult in the last few years since many older machines are being retired, and network bandwidth is squeezed by World Wide Web traffic.

An alternative might be to organize regional consortia and house older machines on campuses distributed throughout the United States, with the machines to be restricted to educational use. If parallel computers were readily available, more schools could offer parallel programming courses.

One can say that explicitly programming parallel algorithms is not useful since there are now parallelization tools and slick programming environments. Teaching a parallel programming course could be similar to the rationale of teaching assembly language programming. Teaching assembly language is still useful to help students understand the inner working of computer architecture, while most of the useful programs are written in C/C++. Programming parallel algorithms explicitly is useful to help students understand the inner workings of parallel architectures and parallelization tools.

Conclusions

This paper presented an example of a parallel programming course at the University of Arkansas. This course is well received by students, especially since students are able to use several different parallel computers. Students learn about sequential algorithm development, parallel algorithm development, shared variables, message passing, and performance. Concepts are demonstrated through several programming assignments. A few algorithms are analyzed in great detail rather that many algorithms shallowly. With more resources available to educational uses, more schools could introduce more students to the wonderful world of parallel programming.

Acknowledgements and Further Information

The author would like to thank the following graduate students for access to their assignment reports and final project reports: Binada Ramachandran, Steven Lovelady, Kris Ramalingam, Arthur Hennessey, Xuyang Li, and Ven Chitturi. The author would like to thank Laxmi Bhuyan at Texas A&M for use of his school's parallel computers.

Information about PVM can be found by anonymous ftp to **netlib2.cs.utk.edu.** Subdirectory **pvm3/book** holds the PVM book referenced above.

More information on this parallel computing course can be found by reading the author's WWW home page at: **http://www.engr.uark.edu/~jmc3**

References

[1] D.P. Agrawal, *Advanced Computer Architecture (Tutorial Text)*, IEEE Computer Society Press, Washington, DC, 1986.

[2] Geist, A., A Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM: Parallel Virtual Machine,* MIT Press: Cambridge, MA, 1994.

[3] Kumar, V., A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing,* Benjamin/Cummings: Redwood City, CA, 1994.

[4] Lindley, C.A., *Practical Image Processing in C,* Wiley: New York, NY, 1991.

[5] Nadel, B.A., "Constraint Satisfaction Algorithms," *Computational Intelligence,* **5**(4), 188-224, Nov. 1989.

[6] Quinn, M.J., *Parallel Computing: Theory and Practice,* McGraw-Hill: New York, NY, 1994.