Parallel Computing for IS Majors

J. Rufinus
rufinus@cs.widener.edu

Y. Kortsarts
yanako@cs.widener.edu

Computer Science Department, Widener University
1 University Place, Chester, PA 19013, USA

Abstract

Most introductory parallel computing courses are designed for computer science students. With the increasing applications of parallel computing in many different areas including the Internet and World Wide Web, it is of great advantage to introduce the concept to Information Systems majors. In this paper we present some examples and suggestions of topics on developing and designing a parallel computing course for Information Systems majors.

Keywords: parallel computing for IS majors, parallel computers, teaching tips

1. INTRODUCTION AND MOTIVATION

Parallel computing has been around for quite a number of years (Gill 1958) (Holland 1959) and is getting more and more attention for several reasons. First, the amount of research in this area has increased tremendously. More researchers look to parallel computing techniques to speedup the calculations and to obtain the results faster. Second, parallel computing is being considered as a standard technique to solve problems in areas as diverse as weather prediction, aircraft design, drug design, and virtual reality. In fact, the Internet, the World Wide Web, and Grid computing are new areas in which parallel computers are utilized. Third, at the present time parallel computing is used not only in educational institutions but has spread to industries (banking, drug companies, manufacturing, etc.)

Decades ago a course on parallel computing was rarely offered. At present, however, parallel computing has been integrated into the computer science curriculum. Courses on parallel computing are currently offered in many computer science departments in many universities (see, e.g., Schaller 2001) and there are many new textbooks on parallel computing available in the market (Grama 2003) (Jordan 2003) (Quinn 2004) (Wilkinson 2005.) A course on parallel computing is usually designed for computer science and engineering majors. What about Information Systems (IS) students? Our own experience demonstrates that not many IS students choose a parallel computing course as a technical elective. Is the subject of parallel computing less attractive to IS students? More data will be needed to answer this question, but certainly a parallel computing class specially designed for IS students is an excellent idea. Can a specially designed one-semester parallel computing course be developed to attract more IS students? If we are selective in choosing topics of interest to IS majors for such a course, the answer to the above question is yes.

The IS 2002 Model Curriculum and Guidelines for Undergraduate Degree

Programs in Information Systems (Gorgone 2002) formulates several characteristics of the IS profession that are integrated into the curriculum. One of these characteristics is "IS professionals must have strong analytical and critical thinking skills." In our opinion, introducing parallel computing to IS students will strengthen the above required skills. It will also equip the students with some of the following capabilities suggested in the IS 2002 guidelines:

- Creativity
- Application development
- Database design
- Systems infrastructure (including design and development of multi-tiered architectures)

Furthermore, the importance of Grid computing in future computing infrastructure (Foster 2004) and the role of parallel algorithms in Grid computing should motivate the teaching of parallel computing techniques for IS majors.

Here we suggest some topics of interest to IS students for faculty interested in developing and designing a parallel computing course specifically for IS majors. These suggestions are based on our experience in teaching parallel computing courses for computer science and IS students in our department. Since our department covers both curricula (Computer Science and Information Systems), many courses are designed for both majors. This gives the opportunity for IS majors to attend a wide range of elective courses offered by our department.

The parallel computing course itself was designed for students with backgrounds in C programming language, data structures and algorithms. Since the parallel computing concepts given in full lecture format are difficult for students to grasp, we also give hands-on exercises during the class time to enhance the students' conceptual understanding. Laboratory sessions are also integrated into the course, as well as course projects. Laboratory sessions and course projects are meant for the students to work in teams (of two to three persons) and each team is allowed to evaluate other teams. In the past, students have worked together on a variety of class projects, from measuring the speedup of parallel algorithms to developing the parallel version of a serial algorithm.

2. EXAMPLES OF PROBLEMS USED IN THE CLASSROOM

Almost everything in nature is done in parallel. We could find a lot of examples around us that could illustrate parallel process. On the other hand, teaching parallel computing concepts to students, both majors and non-majors, is not an easy task because we are more accustomed to serial programming techniques. In most fundamental computer science courses, programming is usually taught as a sequential process, a line-by-line, serial method of programming. Thus we need to change the students' perspective to understand that many problems can also be done in parallel.

Here we provide some examples that illustrate the process of moving from serial to parallel algorithms.

For the first example, a worker is asked to sum all integer numbers from 1 to 100:

$$1 + 2 + 3 + \ldots + 100. \qquad (1)$$

The worker does not have too much choice (anyway, the worker is alone and let us assume the worker does not realize that there are 50 subtotals of the integer number 101, as Gauss did) other than to add 1 and 2 to get the first subtotal, and then to add the first subtotal and 3 to get the second subtotal, etc. Any way the worker does this summation (the worker can start to add 99 and 100 first, etc.) the worker still has to serially do the computation.

If the worker turns out to know how to write a computer program in C, then the for loop section may look like this:

```
sum = 0;
for (i = 1; i <= 100; i++)
        sum = sum + i;
```

Now, if the worker has one or more coworkers to work with, the task may be much faster to finish as the worker can slice the sum into several sub-sums. For example, if there are ten workers altogether,

the first worker can work on adding 1 to 10, the second worker on adding 11 to 20, etc. At the end these workers will communicate to add the subtotals. Thus the algorithm will simply look like this:

(1) Assign work for each worker
(2) Workers work independently to obtain the subtotal, at the same time
(3) Add the subtotals to get the total

Yes, the serial execution of this algorithm is still there (e.g. we cannot proceed to step (2) before finishing step (1) and we cannot proceed to step (3) before finishing step (2), etc.) but step (2) itself is executed in parallel! As a matter of fact, there is no single algorithm that fully (100%) consists of parallel structure.

In the above example, assume that a worker with the help of a calculator can get the sum (the final result) in 3 minutes. Should we obtain the same final result in 18 seconds if ten workers work concurrently? (assume again that the ten workers work with the same speed and the same efficiency). The answer is no. The speedup that could be achieved in a parallel program is not simply equal to the number of workers (or processors in the case of computers).

In fact, Amdahl's law (Amdahl 1967) teaches us that maximum speedup can be calculated using the following formula:

$$\text{Speedup} = \frac{1}{f + (1 - f) / p}, \qquad (2)$$

where f is the fraction of a program that must be performed serially, and p is the number of processors. If only 80% of a program can be parallelized, then as table 1 shows, the speedups for several different numbers of processors. It is easy to see that the speedup will saturate easily at a certain number of processors (for example, the increase of speedup from 2 to 4 processors is about 1.5 times, but going from 16 to 32 processors only increases the speedup by about 1.11 times)

p	Speedup
1	1
2	1.67
4	2.5
8	3.33
16	4
32	4.44

Table 1. The maximum speedup achievable by a parallel computer for different numbers of processors (p), in a program that contains 80% of parallel operations.

It should be noted, however, that Amdahl's law is only one of several formulas used to analyze the performance of a parallel program.

As the second example, we would like to find the minimum number in a sequence of N numbers. If we only have one processor to do the task we need to scan the entire sequence. Now assume that N = 16 and there is p = 8 number of processors. Let the input sequence to be {2, 3, 1, -4, 3, 5, 6, 7, 10, 0, 11, 8, -1, 12, -9, 4}. In the first step we will use all 8 processors (P1 through P8) so that each processor will search for the minimum among 16/8 = 2 data. For example, P1 finds the minimum of 2 and 3, P2 finds the minimum of 1 and –4, P3 finds the minimum of 3 and 5, P4 finds the minimum of 6 and 7, and so on. After the first step is executed we will have 8 results {2, -4, 3, 6, 0, 8, -1, -9.} In the second step we will combine these 8 results and split them among 4 processors. For example, P1 finds the minimum of 2 and –4, P3 finds the minimum of 3 and 6, and so on. The process will continue until we find the last minimum number (-9) that we need. This simple algorithm can be illustrated using a binary tree as shown in Figure 1.

More importantly, this second example illustrates the following concepts:

- Recursive decomposition method. This is a method of solving a problem by dividing it into a set of independent subproblems and recursively applying a similar division into each one of these subproblems. This is similar to the well-known divide-and-conquer strategy.

- Task-dependency graph. This is a directed graph in which the nodes represent tasks while the edges represent the dependencies amongst the tasks. Figure 1 is actually such a graph.
- Mapping and load balancing. Mapping is the process of assigning tasks to processors. Its goal is to maximize the processor utilization, which can be achieved when all processors begin and end execution at the same time. Load balancing is an "ideal" condition where each processor is kept busy for the duration of the overall computation. Our goal is to map the processes such that we can achieve the load balancing. In this example we see that there are 8 processors work in the first step, however only 4 processors (P1, P3, P5, P7) work in the second step, etc.

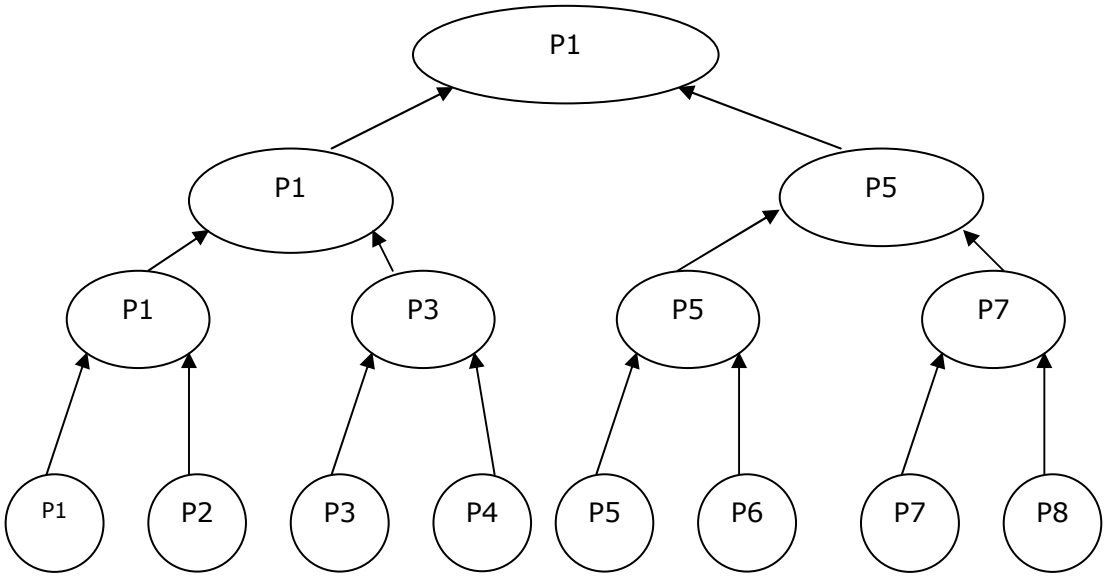

Figure 1. A task-dependency graph to find the minimum in a sequence of N numbers. P1 denotes the processor number 1, P2 denotes the processor number 2, etc.

As the third and last example, we will illustrate the use of Monte Carlo method to estimate the value of π. Consider a unit square and the quadrant of a circle inside it. The radius of the circle is 1 unit and the side of the square is also 1 unit. Next, we choose a point randomly inside this unit square. The probability that one point will randomly hit the circle area is:

$$\frac{\text{Area of circle}}{\text{Area of square}} = \frac{\pi\, 1^2/4}{1^2} = \pi/4 \,. \qquad (3)$$

By repeating this experiment with many points or samples one can get the estimated value of π. This is the famous Monte Carlo method to find π. To convert the above sequential algorithm into parallel one just need to split the number of samples among different processors and to collect the results at the end of computations.

The parallel algorithm will be as follows: If N is the number of samples and p is the number of processors, then:

(1) Each processor performs N/p samples that include:
  (a) Choose the point randomly inside the unit square.
  (b) Check if the point hit the circle.
  (c) Count the number of times that the random point hit the circle.
(2) Collect the data from p processors.

(3) Calculate the estimation of π by using the following formula:

$$\pi = \frac{4\ \text{NumCircle}}{N}, \qquad (4)$$

where NumCircle is number of times that randomly chosen points hit the circle over N samples.

In general, parallel Monte Carlo methods have a negligible amount of interprocessor communications – commonly known as "embarassingly" parallel computation. Thus, using p processors can either (1) Find an estimate about p times faster, as we do in the above example, or (2) Reduce the error of the estimate by a factor of square root of p (it is because the variance of the answer is reduced by a factor of p.)

3. CHOICE OF HARDWARE

To support all the course activities (homework, laboratory, class project, etc.) requires a specific hardware problem. There are several choices available, e.g., to use the supercomputers at a supercomputer center, to build a special-purpose multi-processor computer laboratory, or to use a cluster of computers.

Using computers at a supercomputer center is usually done through a remote access. Sometimes there are many users who are simultaneously and continuously doing work on these supercomputers, thus the performance of a parallel program to be tested may be affected tremendously.

The special-purpose multi-processor computer laboratory is the second choice. This laboratory may be equipped with a certain tools that capable of handling parallelization in the operating system and compiler levels. This option, however, is costly. It needs budget and personnel to maintain this laboratory. Big universities usually take this option.

In our case, we have built our own cluster of computers, sometimes called a Beowulf cluster (Sterling 1999), from various grants (university and the NSF). This is a good choice for small universities like ours since it is not that expensive to build such a cluster. Our cluster is installed with Linux RedHat (Other Linux distributions can also be used) and consists of 16 Intel Xeon 2.4 GHz processors. All machines are connected through a Gigabit Ethernet. An interface is needed to use these interconnected machines as a cluster. There have been several interfaces available, including Message Passing Interface (MPI), Parallel Virtual Machine (PVM) and Linda. Further, there are two kinds of MPI implementation: (1) Local Area Multicomputer (LAM) MPI from the University of Notre Dame (www.lam-mpi.org) and (2) MPICH (www-unix.mcs.anl.gov/mpi/mpich). Textbooks and reference books on MPI are also available (Pacheco 1996) (Gropp 1999.)

Both implementations, LAM-MPI and MPICH, are freely available and can be downloaded through the Internet. TCP Linda is proprietary software (Lindaspaces 2004.) We installed LAM-MPI in our cluster and used it in our course.

4. SUGGESTED TOPICS

In any introductory parallel computing course, all the fundamentals of parallel programming should be discussed thoroughly and probably would take the longest time to cover. These include:

(I) The basic terminologies. For example, speedup is defined as

$$\text{speedup} = \frac{\text{sequential runtime}}{\text{parallel runtime}}, \qquad (5)$$

while the efficiency is defined as:

$$\text{efficiency} = \frac{\text{speedup}}{\text{processors used}}. \qquad (6)$$

Thus, efficiency is a measure of processor utilization. It tells us how the processors are being used on the actual computation. For example, if the efficiency = 50%, the processors are utilized half the time. The maximum efficiency (100%) occurs when the speedup is equal to the number of processors.

(II) Parallelization method, which involves the following steps:

1. Identify parallelism in a problem. In this step we study the problem carefully, and identify whether there are portions of the problem that could be performed concurrently. Identifying parallelism in a problem is crucial, since it will determine whether parallelization will accelerate solution of the problem.

2. Partitioning. This is a process of dividing/decomposing the computation and/or the data into smaller parts. Drawing a task dependency graph may be helpful in this process. In this step we also have to recognize the communication pattern between the tasks and how to organize them into larger tasks, with the goal to lower communication overhead.

Thus, in the above two steps (identifying parallelism in a problem and partitioning the problem into smaller pieces) the IS students would really learn is how to organize data. First, they would have to recognize which data could be executed concurrently. Second, these data should then be grouped together in the decomposition step. This is an excellent way to learn to organize data. "Always look in parallel universes", is a lesson for organizational learning (Hipple 2003.) The study of parallel computing techniques would enhance the organizational problem solving capability of IS students.

3. Mapping the tasks into processors is the software implementation step. We have to consider this process because we are executing our program in a cluster of computer, which operating system does not automatically map the tasks into processors. The ultimate goals of mapping process are to (a) utilize the processors as much as possible, (b) minimize the communication between processors.

(III) Performance analysis. The objective of this analysis is to predict the performance of a parallel program by using some well-known formulas (e.g., Amdahl's law.)

Timing various parallel algorithms and measuring the speedup of those algorithms with respect to the serial version are integrated into the whole course content. In the laboratory sessions, for example, students are given varieties of programs to test. The students then plot the speedups with respect to the number of processors and compare the obtained results with theory.

The next topic to be taught relates to the question of how the parallel algorithms would be implemented. The mastery of C language is enough to code a parallel algorithm. In addition, we have to learn another tool that will handle the interprocessor communications. The most popular tool to use at the present time is MPI, an interface to administer/manage communications among processors by passing/sending messages. MPI consists of many functions that can be called in a program, even though practically speaking not all of these functions need to be used in writing a parallel program.

It usually takes one to two weeks to teach the concepts of MPI to students. Simple examples should be given in class, plus laboratory and homework, which consist of programming assignments.

To strengthen the students' capability in the field of Information Technology Hardware and Systems Software (IS 2002.4), a one-week topic on different parallel programming platforms should also be added to the list of topics of interest. Here the students will learn about the concepts of multiprocessors, multicomputers, and interconnection networks.

Since most IS students have taken courses on data structures and algorithms by their sophomore years it is suggested that the following topics also be discussed:

(*) A variety of parallel sorting algorithms
(*) A variety of parallel graphs algorithms

Additional topics to be covered are usually optional and will depend on the students' technical background (e.g., discrete mathematics, algebra, and statistics, but not differential equations, or higher algebra.) The following topics are of interest to our IS students:
(I) Solving the linear systems. This includes topics such as:

- Parallel matrix-vector and parallel matrix-matrix multiplications. These are excellent topics to illustrate many of the concepts of MPI.

- Back substitution, Gaussian elimination, and some other methods.

(II) Monte Carlo methods. Interestingly, even though parallel Monte Carlo technique is an example of "embarrassingly" parallel computation, its applications are ample and can be found not only in science and engineering but also in finance (Brandimarte 2002), economics, etc.

(III) Parallel database. Database sizes have been increasing steadily in the past decade. It is now common to find very large databases that can hold data of more than 1 terabytes. Queries are run to access data warehouses with the purpose to gather important information, decision making, etc. Such complex queries, of course, require a lot of time to execute. Parallel database could help reducing the execution time. Applications of parallel database include the World Wide Web search engine, etc.

5. CONCLUSION

Equipping IS students with knowledge of parallel computing will enhance their capabilities in several areas, including technology, organizational problem solving, creativity, as well as algorithmic design.

6. ACKNOWLEDGEMENT

The work of JR and the cluster of computers used in this project are partially funded by the National Science Foundation grant number 0304429 and Widener University Faculty Development Grants.

7. REFERENCES

Amdahl, G., 1967, "Validity of the Single-Processor Approach to Achieving Large-Scale Computing Capabilities." Proceedings of 1967 AFIPS Conference Vol. 30, p. 483.

Brandimarte, Paolo, 2002, "Numerical Methods in Finance." John Wiley.

Foster, Ian, and Carl Kesselman, Editors, 2004, "The Grid 2." Morgan-Kauffman.

Gill, S, 1958, "Parallel Programming." Computer Journal Vol. 1, pp. 2-10.

Gorgone, John T., Gordon B. Davis, Joseph S. Valacich, Heikki Topi, David L. Feinstein, and Herbert E. Longenecker, Jr., 2002, "IS 2002 Model Curriculum and Guidelines for Undergraduate Degree Programs in Information Systems." Association for Information Systems.

Grama, Ananth, Anshul Gupta, George Karypis, and Vipin Kumar, 2003, "Introduction to Parallel Computing." Second Edition, Addison-Wesley.

Gropp, William, and Marc Snir, 1999, "MPI: The Complete Reference." MIT Press.

Hipple, Jack, 2003, "The Future of Organizational Problem Solving." http: //www.innovation-triz.com/papers/ futureorganize.ppt.

Holland, J., 1959, "A Universal Computer Capable of Executing an Arbitrary Number of Subprograms Simultaneously." Proceedings of East Joint Computer Conference Vol. 16, pp. 108-113.

Jaeckel, Peter, 2002, "Monte Carlo Methods In Finance." John Wiley.

Jordan, Harry F. , and Gita Alaghband, 2003, "Fundamentals of Parallel Processing." Prentice-Hall.

Lindaspaces, 2004, http://www.lindaspaces.com/ about/index.html

Pacheco, Peter, 1996, "Parallel Programming with MPI." Morgan-Kauffman.

Queen, Michael J., 2004, "Parallel Programming in C with MPI and OpenMP." McGraw-Hill.

Schaller, N., 2001, "Nan's Parallel Computing Page." http://www.cs.rit.edu/~ncs/ parallel.htm.

Sterling, Thomas L., John Salmon, Donald J. Becker, and Daniel F. Savarese, 1999, "How to Build a Guide to the Implementation and Application of PC Clusters." MIT Press.

Wilkinson, Barry, and Michael Allen, 2005,
"Parallel Programming. Techniques and
Applications Using Networked
Workstations and Parallel Computers."
Second Edition, Prentice-Hall.