An Innovative Course in Parallel Computing

Yi Pan

Georgia State University

Abstract:

An innovative course in parallel computing is described in this paper. Traditional parallel computing courses use either low-level message passing interfaces or high level language directives, but not both, due to limited time. In our course, we use both high-level and low-level language constructs. In this paper, we briefly introduce several language interface standards and discuss why we have chosen to use OpenMP (a high-level language interface) and MPI (a low-level message language interface) in our parallel computing class. Some of the drawbacks of using OpenMP in teaching are identified and we show how these drawbacks are being addressed in the course. Several programming projects and a research/survey project given in our class are also described in detail. Through careful design of the course, we show that students can learn many basic concepts through low-level parallel language interfaces and parallelize real (long) scientific codes in high-level parallel language directives within a short period. This would be impossible to accomplish if only one language had been taught in the course.

Index terms: MPI, OpenMP, parallel computing, parallel language, teaching.

Introduction

Parallel computing is information processing that emphasizes the concurrent manipulation of data elements belonging to one or more processor solving a single problem. A parallel computer is a multiprocessor computer capable of parallel processing. Parallel computing has emerged as a key enabling technology in modern computing. The past several years have witnessed an ever-increasing acceptance and adoption of parallel processing, both for high-performance scientific computing and for more "general-purpose" applications. The trend was a result of the demand for higher performance, lower cost, and sustained productivity. The acceptance has been facilitated by two major developments: massively parallel processors and the widespread use of clusters of workstations.

For those interested in high-speed computing, studying and using parallel computing techniques is a necessity now. In the last ten years, courses on parallel computing and programming have been developed and offered in many institutions as a recognition of the growing significance of this topic in computer science [1],[9],[10], [13]. Parallel computation curricula are still in their infancy, however, and there is a clear need for communication and cooperation among the faculty who teach such courses. Georgia State University (GSU), like many institutions in the world, has offered a parallel programming course at the graduate and Senior undergraduate level for several years. It is not a required course for computer science majors, but a course designated to accomplish computer science hours. It is also a course used to obtain a Yamacraw Certificate. Yamacraw Training at GSU was created in response to the Governor's initiative to establish Georgia as a world leader in highbandwidth communications design. High-tech industry is increasingly perceived as a critical component of tomorrow's economy.

As we all know, the message passing paradigm has several disadvantages: the cost of producing a message passing code may be very high, the length of the code grows significantly, and it is much less readable and less maintainable than the sequential version. For these reasons, it is widely agreed that a higher level programming paradigm is essential if parallel systems are to be widely adopted. Most schools teaching the course use low-level message passing standards such as MPI or PVM and have not yet adopted high-level parallel language directives such as OpenMP [1], [9], [10], [13]. To catch up with the industrial trend, we decided to teach the shared-memory parallel programming model (using a high-level parallel language directives) in addition to the message passing parallel programming model (using low-level message passing libraries). This paper describes our experience in using both high-level parallel language directives and low-level message passing libraries to teach a parallel programming course at Georgia State University. The paper is organized as follows. In Section 2, we give a brief introduction to several popular parallel programming standards in industry and academic environments. Section 3 will justify our choice of using both OpenMP and MPI in our teaching. In Section 4, we will discuss in detail how MPI and OpenMP are used in the projects and how they can complement each other in learning. We will conclude our paper in Section 5.

Parallel Programming Standards

Many parallel programming languages and standards have been proposed. The four major standards for programming parallel systems that were developed in open forums are: High Performance Fortran (HPF) [8], OpenMP [2], PVM [3] and MPI [5]. Among them, HPF and OpenMP are high-level parallel programming language standards, and PVM and MPI are low-level message-passing language interfaces.

HPF consists of Fortran 90 and a set of extensions to it. HPF extensions are defined by a coalition of industry, academic and laboratory representatives, and provide access to high-performance architecture features while maintaining portability across platforms. HPF relies on advanced compiler technology to ex-

pedite the development of data-parallel programs [8]. Thus, although it is based on Fortran, HPF is a new language, and hence requires the construction of new compilers. As a consequence, each implementation of HPF is, to a great extent, hardware specific, and until recently there were very few complete HPF implementations. Furthermore most of the current implementations are proprietary and quite expensive. HPF has been written for the express purpose of writing data-parallel programs, and, as a consequence, it is not well-suited for dealing with irregular data-structures or control-parallel programs.

OpenMP has emerged as the standard for shared memory parallel programming. For the first time, it is possible to write parallel programs which are portable across the majority of shared memory parallel computers. OpenMP is a portable, scalable model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications for platforms ranging from the desktop to the supercomputer.

OpenMP has the following benefits for parallel programming compared with message passing models such as MPI: a) A user just needs to add some directives to the sequential code to instruct the compiler how to parallelize the code. Hence, it has unprecedented programming ease, making threading faster and more cost-effective than ever before. b) The directives are treated as comments when running a single processor. Hence, a single-source solution can be used for both serial and threaded applications, lowering code maintenance costs. c) Parallelism is portable across Windows NT and Unix platforms. d) The correctness of the results generated using OpenMP can be verified easily which dramatically lowers development and debugging costs.

The Parallel Virtual Machine (PVM) is a software package that permits a heterogeneous collection of Unix and/or Windows computers hooked together by a network to be used as a single large parallel computer. It uses the message-passing model to allow programmers to exploit distributed computing across a wide variety of computer types, including multiprocessor systems [3]. A key concept in PVM is that it makes a collection of computers appear as one large virtual machine, hence its name. The PVM computing model is simple yet very general, and accommodates a wide variety of application program structures. The programming interface is deliberately straightforward, thus permitting simple program structures to be implemented in an intuitive manner. The user writes his application as a collection of cooperating tasks. Tasks access PVM resources through a library of standard interface routines. These routines allow the initiation and termination of tasks across the network as well as communication and synchronization between tasks. The PVM message-passing primitives are oriented towards heterogeneous operation, involving strongly typed constructs for buffering and transmission. Communication constructs include those for sending and receiving data structures as well as high-level primitives such as broadcast, barrier synchronization, and global sum.

MPI is another message-passing standard for parallel programming. It has gained more popularity and acceptance in the parallel computing community recently. The goal of MPI is to develop a widely used standard for writing message-passing programs. As such the interface attempts to establish a practical, portable, efficient, and flexible standard for message passing. The MPI Forum seeks to make use of the most attractive features of a number of existing message passing systems, rather than selecting one of them and adopting it as the standard. The MPI standardization effort involves about 60 people from 40 organizations mainly from the United States and Europe. Most of the major vendors of concurrent computers are involved in MPI, along with researchers from universities, government laboratories, and industry. MPI specifies a library of extensions to C and Fortran that can be used to write message passing programs [5]. So an implementation of MPI can make use of existing compilers, and it is possible to develop more-or-less portable MPI libraries. Thus, unlike HPF, it is relatively easy to find an MPI library that will run on existing hardware. All of these implementations can be freely downloaded from the internet. Message passing is a completely general method for parallel programming. Indeed, the generality and ready availability of MPI have made it one of the most widely used systems for parallel programming. Compared with the PVM library, MPI has recently become more popular.

All the above standards have been used extensively in industry and academic environments. It is easy to see that both high-level parallel language standards and low-level message-passing interfaces have advantages over the others and their own limitations, and will co-exist in the future. However, we believe that the future of high performance computing heavily depends on high level parallel programming languages such as OpenMP due to the increasingly high labor costs and the scarcity of good parallel programmers. High level parallel programming languages are one way to make parallel computer systems popular and available to non-computer scientists and engineers. Hence, teaching students how to use high level parallel programming language interfaces is an important task for teaching parallel programming.

Why Teach Both OpenMP and MPI?

The OpenMP Application Program Interface (API) supports multi-platform shared-memory parallel programming in C/C++ and Fortran on all architectures, including Unix platforms and Windows NT platforms. Jointly defined by a group of major computer hardware and software vendors, OpenMP is a portable, scalable model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications for platforms ranging from the desktop to the supercomputer [2]. It consists of a set of compiler directives and library routines that extend FORTRAN, C, and C++ codes for shared-memory parallelism.

OpenMP's programming model uses fork-join parallelism: the master thread spawns a team of threads as needed. Parallelism is added incrementally: i.e., the sequential program evolves into a parallel program. Hence, we do not have to parallelize the entire program at once. OpenMP is usually used to parallelize loops. A user identifies the most time consuming loops in the code, and splits them up between threads. When parallelizing a loop in OpenMP, we may also use the schedule clause to perform different

scheduling policies which effect how loop iterations are mapped onto threads. Hence, to achieve a limited degree of load balancing in OpenMP is quite easy. The section work-sharing construct gives a different structured block to each thread. This way, task parallelism can be implemented easily if each section has a task (procedure call).

Because OpenMP is a high level parallel language interface, many details are hidden from a programmer. The benefit is that students can learn quickly and start to program immediately after learning some techniques. The pitfall is that students cannot clearly see the communications involved in a parallel program. Our approach to overcome this problem is to supplement OpenMP projects with some simple MPI programs. Students first learn the basics of parallel programs in a distributed memory environment. They start to parallelize a sequential code using simple MPI communication constructs such as MPI_Bcast, MPI_Reduce, MPI_Send, and MPI_Recv. Through several small projects, they learn the concepts of one-to-one communication, multicast, broadcast, reduction, synchronization, and concurrency. Later, when they use OpenMP to parallelize a program, they already have a deep understanding of communication structure, communication overhead, scalability and performance issues.

Another shortcoming with the current OpenMP standard is that it does not provide memory allocation schemes for arrays and other data structures since OpenMP is designed for shared memory machines. Again, this relieves the students from complicated memory allocation decisions, allowing concentration on loop and task parallelism. This is good for the ease of programming, but students do not know the details of array allocation schemes such as BLOCK or CYCLIC schemes commonly used in distributed memory environments. Since the memory on the SGI Origin 2000, which is used in our teaching, is not physically shared, SGI provides data distribution directives to allow users to specify how data is placed on processors. If no data distribution directives are used, then data are automatically distributed via the "first touch" mechanism [4] which places the data on the processor where it is first used. Because different allocation schemes may affect the performance of a program greatly, SGI data distribution directives are required in the final project to show the performance improvement. Students are required to try several data distribution schemes, to observe the running times and to comment on the timing results as described below. In this way, the relationship between memory allocation schemes and performance is demonstrated and the problem with OpenMP in teaching is solved.

Due to the inherent limitations within OpenMP, students would not learn all the concepts and the whole picture in parallel programming using OpenMP alone. Our strategy is to supplement OpenMP with explanation on several typical MPI codes and small projects using the MPI standard. Then, students experiment with various scheduling policies and complicated parallelization methods in OpenMP. In this way, students experience various parallel schemes and techniques in a short period of time. This would be very hard to achieve if only the MPI or PVM programming model were used in teaching parallel programming because of the time demands for implementation and parallelization of large codes in MPI or PVM.

Hence, our strategy is to teach students the basic concepts in parallel programming such as scalability, broadcast, one-to-one communication, performance, communication overhead, speedup, etc, through a low-level parallel programming language interfaces (normally message passing), and teach other concepts such as various scheduling policies and task parallelism through a high-level parallel programming language constructs (normally directives). Since MPI and OpenMP are the most widely used languages in the two categories, these are selected to teach parallel programming. The following section details the strategy of using both OpenMP and MPI in the class.

Programming Projects

Programming projects are an important part of the learning experience The parallel programming class at GSU is a semester-long class for upper-level undergraduates and beginning graduate students. Several tutorials on MPI and OpenMP from the Ohio Supercomputing Center were used as supplements to a parallel algorithms textbook [12].

The course begins with an overview of parallel computing and continues with a brief introduction to parallel computing models such as various PRAMs, shared memory models and distributed memory models. The concepts of data parallelism and pipelining are also introduced at that time. The next block of lectures forms a transition into a more or less standard parallel algorithms course. First serial and parallel versions for a very simple computation - e.g., prefix sums and prime finding, are discussed. In the course of analyzing the performance of these algorithms, the concepts of speedup, scalability and efficiency are developed. The deterioration of the performance of the parallel algorithm as the number of processes is increased leads naturally to a discussion of Amdahl's Law and scalability. Then, various interconnection networks are presented. Performance measures such as degree, diameter, average distance, connectivity, fault-tolerance, and routing are introduced. The performance and cost measures of different topologies are analyzed and compared. Students are shown that it is impossible to find a topology which is the best for interconnecting processors in a parallel computing system in terms of all the performance measures. The remainder of the lectures is devoted to the development and analysis of a variety of standard parallel algorithms from linear algebra, some algorithms for searching and sorting, and graph algorithms. When an algorithm is discussed, its implementation in MPI or OpenMP is also presented.

To effectively evaluate the students in both theory and practice in parallel computing, the course work consists of two tests, a final exam, five programming projects and a research paper. Since the course's emphasis is on parallel programming, projects are an important part of the course. The projects give students the opportunity to apply theory learned in the classroom to solving problems and provide hands-on experience to students on using parallel languages and compilers.

Project 1:

The purpose of the first project is simply to acquaint students with the system and programming environment of the Origin 2000. In this project, they

write a simple addition code, and measure the parallel times using different numbers of processors. Several MPI communication constructs are used in the project.

Project 2:

In the second project, the students implement an MPI code to calculate using Simpson's Rule instead of the rectangle rule discussed in class, where students are exposed to various MPI communication functions. For timing measurements and precision, they need to test the code using several different numbers of subintervals to see the effect on the precision of results and different number of processors on the execution times.

Project 3:

In the third project, students implement the parallel game of life. Through the assignment, students learn various domain decomposition strategies. Both 1D and 2D domain decompositions are discussed. Since 2D decomposition is quite involved in MPI, extra credits are given if they implement the code using 2D decomposition. Students are required to give performance curves (speedup and execution time) for their program with different number of processors.

Project 4:

After the first three projects, students now understand the communication mechanisms of parallel computing systems, and communication overhead within a parallel code. Hence, it is time to introduce OpenMP and to implement some more sophisticated codes. After briefly discussing the use of OpenMP and illustrating OpenMP through several examples, students are asked in the fourth project to initialize a huge array so that each element has its index as its value. A second real array which contains the running average of array is then created. The loops are parallelized with all four scheduling schemes available in OpenMP (static, dynamic, guided, and runtime) and the running times are measured with different scheduling policies and different chunk sizes. Students write up their observations on the timings using the four different scheduling policies and explain why the performance differs in these cases.

Project 5:

In the fifth project, students learn how to parallelize a real research Fortran code in OpenMP or HPF. Since only OpenMP is discussed in class, students need to learn HPF themselves if they choose to implement a code in HPF. Since HPF and OpenMP have many similarities, it seems that is is not a problem. The code is from our research project funded by NSF. The program calculates the power-spectral density of thin avalanche photodiodes, which are used in optical networks. The program extends the time-domain analysis of the dead-space multiplication model to compute the autocorrelation function of the APD impulse response [6]. In particular, the correlation subroutines are extremely memory and time intensive, since they involve large three dimensional arrays and four nested loops. Clearly, if we can parallelize these loops efficiently, then we can reduce the computation time drastically. In our research, we have parallelized the code using both MPI and OpenMP and the results show that both parallel codes are quite scalable up to 24 processors on an SGI Origin 2000 [11]. The results obtained in our research project are used as guidance for students to parallelize the code in project 5.

Students have to apply all the knowledge learned so far in the course to parallelizing the code. The project contains several parts. First, they need to select the best scheduling policy and chunk size. In order to do so, they can pick up one typical procedure in the code for parallelization. They need to test the code using static, dynamic, and guided scheduling policies with different chunk sizes and different number of processors. After measuring the time and speedup for each case, list the results in a table. Second, they pick up the best combination of scheduling policy and chunk size based on the results obtained, and use them to parallelize the other similar subroutines which dominate the running time. Other subroutines take much less time and are not required for parallelization. Now, they have parallelized all the subroutines using loop parallelism, it is time to parallelize the code using both loop and task parallelism in the third step. Several subroutines in the code are independent, and can be executed concurrently. So far, the best scheduling policy, best chunk size for the policy, and both loop and task parallelism are obtained in the above three steps, while array mapping is done automatically by the OpenMP compiler. For the final step, the arrays are distributed manually using SGI array distribution directives because array distribution directives are not available in OpenMP. The purpose is for students to understand the effect of array distribution on the runtime performance. Students are also required to write a short report to summarize the results obtained. Through these steps, students learn how to parallelize a real code in a step-by-step fashion.

Final Paper:

Besides programming projects, students are also required to write a research paper or a survey paper on a chosen topic in parallel processing. The purpose is for the students to apply the knowledge learned in the course to an application through a research paper or to acquire a deep knowledge in a specific topic through a survey.

A research paper should include some new discovery and discussion in the area of parallel language/system design, parallel architecture, or parallel algorithms. Design and analysis of a new parallel algorithm and a comprehensive comparison with other existing parallel algorithms could be a good topic. Some references are needed to give an overview of the area and to give some pointers to existing algorithms and schemes. Some students have implemented algorithms using MPI and/or OpenMP using various strategies and compared the performance of their implementations with the results published in the literature. We believe that some of the research findings are publishable after some revisions. For example, one student implemented a parallel program for Cholesky factorization using both MPI and OpenMP, and did a

lot of testing using various scheduling and partition strategies. He also did a comprehensive comparison among the different implementations, and wrote an excellent research paper at the end of the course [7]. The paper is being revised and potentially could be published in a conference. At the end of the term, students need to present their findings and submit a research report.

A survey paper shall be a review and discussion (e.g., compare, contrast, add opinions of your own) of some recent topic in parallel language/system design, parallel architecture, or parallel algorithms. It should include reference to at least two papers published within the last 5 years from reputable technical journals or proceedings from related conferences. Some students selected a topic not covered in the course, while others surveyed the most recent research results in a topic which has been covered in the class, but not so deeply. It is required that student come up with their own opinions about the current research issues in the particular area. At the end of the term, students need to present a talk overviewing the topic they choose and submit a survey paper. Other students also benefit from their survey talks, since of the most time survey talks are about new topics or recent developments in an area.

It is our intention for the students to learn theoretical concepts and parallel programming practices in the course through lectures, tests, programming projects and a research/survey paper. The outcome of the course is very good. Based on student evaluations and comments on the course, most students feel that they learned a lot in the course. For example, in the teaching evaluation of the course in the Spring term of 2002, I received 4.52 out of 5 for Question 13 (Stimulated thinking and gave subject insights) and 4.68 out of 5 for Question 15 (Motivated students to learn). My overall teaching effectiveness (Question 17) in the course is 4.83 out of 5 (20 students gave me the highest mark A and 4 students gave me the second highest mark B). This shows that our teaching methodology and approaches adopted in the course are very successful. Some of the students have already applied the knowledge learned in the course to research projects supported by the NSF and Air Force. This would have been impossible if only MPI had been taught in the course due to limited time.

Conclusion

Similar to assembly languages and high-level languages such as C or Java in the sequential domain, both low-level and high-level parallel language interfaces have their shortcomings and advantages. Hence, students need to learn both low-level message passing interfaces and high-level parallel language directives. This paper provides an innovative course design which teaches both low-level message passing interfaces and high-level parallel language directives. As OpenMP becomes more popular for parallel programming because of its many advantages over message passing programming models, it is important to introduce OpenMP in a parallel programming course. However, OpenMP also has some shortcomings for teaching parallel programming concepts. Our strategy is to use MPI to convey the basic concepts of parallel programming and to use OpenMP to tackle more complicated problems, such as various scheduling policies and combined loop and task parallelism. It appears that the strategy is well received by the students. In the course, students learned the basic concepts, and also could parallelize real scientific codes within a short period using OpenMP. The practical experience gained in parallelizing a real scientific code is essential for students to apply parallel computing in their future career.

Bibliography

1. F.C. Berry. An undergraduate parallel processing laboratory, IEEE Trans. Educations, vol. 38, pp. 306-311, Nov. 1995
2. Rohit Chandra, Ramesh Menon, Leo Dagum, David Kohr, Dror Maydan, and Jeff McDonald, Parallel Programming in OpenMP, Morgan Kaufmann Publishers, October 2000, 300 pages
3. J. Dongarra, P. Kacsuk, N. Podhorszki (Editors): Recent Advances in Parallel Virtual Machine and Message Passing Interface: 7th European PVM/MPI Users' Group Meeting, Balatonfuered, Hungary, September 2000
4. J. Fier. Performance Tuning Optimization for Origin 2000 and Onyx 2. Silicon Graphics, 1996.http://techpubs.sgi.com
5. W. Gropp, E. Lusk, A. Skjellum. Using MPI : portable parallel programming with the message- passing interface, MIT Press, Cambridge, Mass., 1994.
6. M. M. Hayat, O-H. Kwon, Y. Pan, P. Sotirelis, J. C. Campbell, B. E. A. Saleh, and M. C. Teich, Gain-bandwidth characteristics of thin avalanche photodiodes, IEEE Transactions on Electron Devices, Vol. 49, No. 5, May 2002, pp. 770-781.
7. Z. Liu. Parallelization of Cholesky Factorization Algorithm with MPI and OpenMP Implementations, CSC6310 Final Project Report, Department of Computer Science, Georgia State University, May 2001.
8. C. H. Koelbel. The High performance Fortran handbook, MIT Press, Cambridge, Mass., 1994
9. R. Miller. The status of parallel processing education, Computer, vol. 27, no. 8, pp. 40-43, Aug. 1994
10. C. H. Nevison. Parallel computing in the undergraduate curriculum, Computer, vol. 28, no. 12, pp. 51-53, Dec. 1995
11. Yi Pan, C.S. Ierotheou, and M.M. Hayat, Parallel Implementation of the Recurrence Method for Computing the Power-Spectral Density of Thin Avalanche Photodiodes, Proc. of the 4th Workshop on High-Performance Scientific and Engineering Computing with Applications, IEEE CS Press, August 18-21, 2002, Vancouver, British Columbia, Canada.
12. M. J. Quinn. Parallel Computing - Theory and Practice, McGraw-Hill, INC., 1994
13. B. Wilkinson and M. Allen. A statewide senior parallel programming course, IEEE Trans. Educations, vol. 42, no. 3, pp. 167-173, 1999

Dr. Yi Pan received his B.Eng. and M.Eng. degrees in computer engineering from Tsinghua University, China, in 1982 and 1984, respectively, and his Ph.D. degree in computer science from the University of Pittsburgh, USA, in 1991. Currently, he is an associate professor in the Department of Computer Science at Georgia State University.

His research interests include parallel and distributed computing, optical networks and wireless networks. His pioneer work on computing using reconfigurable optical buses has inspired extensive subsequent work by many researchers, and his research results have been cited by more than 100 researchers worldwide in books, theses, journal and conference papers. He is a co-inventor of three U.S. patents (pending) and 5 provisional patents. He has co-edited 9 books/proceedings, and published more than 120 research papers including over 50 journal papers (more than 20 of which have been published in various IEEE journals) and received many awards from agencies such as NSF, AFOSR, JSPS, IISF and Mellon Foundation. His recent research has been supported by NSF, AFOSR, AFRL, JSPS, and the states of Georgia and Ohio. Dr. Pan is currently serving as an editor-in-chief or editorial board member for 7 journals including 3 IEEE Transactions and a guest editor for 7 special issues. He has also served as a program or general chair for several international conferences and workshops.

Dr. Pan has delivered over 40 invited talks, including keynote speeches and colloquium talks, at conferences and universities worldwide. Dr. Pan is an IEEE Distinguished Speaker (2000-2002), a Yamacraw Distinguished Speaker (2002), a Shell Oil Colloquium Speaker (2002), and a senior member of IEEE. He is listed in Men of Achievement, Who's Who in Midwest and Who's Who in America.