

Teaching Parallel Programming Using Both High-Level and Low-Level Languages

Yi Pan

Georgia State University, Atlanta, GA 30303, USA
pan@cs.gsu.edu

Abstract. We discuss the use of both high-level and low-level languages in the teaching of senior undergraduate and junior graduate classes in parallel and distributed computing. We briefly introduce several language standards and discuss why we have chosen to use OpenMP and MPI in our parallel computing class. Major features of OpenMP are briefly introduced and advantages of using OpenMP over message passing methods are discussed. We also include a brief enumeration of some of the drawbacks of using OpenMP and how these drawbacks are being addressed by supplementing OpenMP with additional MPI codes and projects. Several projects given in our class are also described in this paper.

1 Introduction

Parallel computing, the method of having many small tasks solve one large problem, 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.

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],[7],[8],[10]. 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.

Our department offers a curriculum to prepare students for careers in Yamacraw target areas, and Parallel and Distributed Computing is one of the

courses in the curriculum. Graduate students from other departments may also take the course in order to use parallel computing in their research.

Low-level languages and tools that have been used at GSU for the course includes Parallel Virtual Machine (PVM) and the Message Passing Interface (MPI) on an SGI Origin 2000 shared memory multiprocessor system. As we all know, the message passing paradigm has several disadvantages: the cost of producing a message passing code may be between 5 and 10 times that of its serial counterpart, the length of the code grows significantly, and it is much less readable and less maintainable than the sequential version. Most importantly, the code produced using the message passing paradigm usually uses much more memory than the corresponding code produced using high level parallel languages since a lot of buffer space is needed in the message passing paradigm. 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 OpenMP [1], [7], [8], [10]. To catch up with the industrial trend, we decided to teach the shared-memory parallel programming model beside the message passing parallel programming model. This paper describes experience in using OpenMP as well as MPI to teach a parallel programming course at Georgia State University.

2 About OpenMP

The rapid and widespread acceptance of shared-memory multiprocessor architectures has created a pressing demand for an efficient way to program these systems. At the same time, developers of technical and scientific applications in industry and in government laboratories find they need to parallelize huge volumes of code in a portable fashion.

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 whole program at once. OpenMP is usually used to parallelize loops. A user finds his most time consuming loops in his code, and splits them up between threads. In the following, we give some simple examples to demonstrate the major features of OpenMP.

Below is a typical example of a big loop in a sequential C code:

```

void main()
{
    double A[100000];
    for (int i=0;i<100000;i++) {
        big_task(A[i]);
    }
}

```

In order to parallelize the above code in OpenMP, users just need to insert some OpenMP directives to tell the compiler how to parallelize the loop.

A short hand notation that combines the Parallel and work-sharing construct is shown below:

```

void main()
{
    double Res[100000];
#pragma omp parallel for
    for(int i=0;i<100000;i++)
    {
        big_task(Res[i]);
    }
}

```

The OpenMP work-sharing construct basically splits up loop iterations among the threads in a team to achieve parallel efficiency. By default, there is a barrier at the end of the “omp for”. We can use the “nowait” clause to turn off the barrier.

Of course, there are many different OpenMP constructs available for us to choose. The most difficult aspect of parallelizing a code using OpenMP is the choice of OpenMP constructs, and where these should be inserted in the sequential code. Smart choices will generate efficient parallel codes, while bad choices of OpenMP directives may even generate a parallel code with worse performance than its original sequential code due to communication overheads.

When parallelizing a loop in OpenMP, we may also use the schedule clause to perform different scheduling policies which effects how loop iterations are mapped onto threads. There are four scheduling policies available in OpenMP. The static scheduling method deals-out blocks of iterations of size “chunk” to each thread. In the dynamic scheduling method, each thread grabs “chunk” iterations off a queue until all iterations have been handled. In the guided scheduling policy, threads dynamically grab blocks of iterations. The size of the block starts large and shrinks down to size “chunk” as the calculation proceeds. Finally, in the runtime scheduling method, schedule and chunk size are taken from the OMP_SCHEDULE environment variable and hence are determined at runtime.

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). The following code shows that three tasks are parallelized using the OpenMP section work-sharing construct.

```

#pragma omp parallel
#pragma omp sections
{
    task1();
#pragma omp section
    task2();
#pragma omp section
    task3();
}

```

Another important clause is the reduction clause, which effects the way variables are shared. The format is `reduction (op : list)`, where `op` can be any general operation such as `+`, `max`, etc. The variables in each “list” must be shared in the enclosing parallel region. Local copies are reduced to a single global copy at the end of the construct. For example, here is an example for global sum and the final result is stored in the variable `res`.

```

#include <omp.h> #define NUM_THREADS 2 void main ()
{
    int i;
    double ZZ, func(), res=0.0;
    omp_set_num_threads(NUM_THREADS)
#pragma omp parallel for reduction(+:res) private(ZZ)
    for (i=0; i< 1000; i++)
    {
        ZZ = func(i);
        res = res + ZZ;
    }
}

```

Programming in a shared memory environment is generally easier than in a distributed memory environment and thus saves labor costs. However, programming using message passing in a distributed memory environment usually produces more efficient parallel code. This is much like the relationship between assembly languages and high level languages. Assembly codes usually run faster and are more compact than codes produced by high-level programming languages and are often used in real-time or embedded systems where both time and memory space are limited, and labor costs are not the primary consideration. Besides producing efficient codes, assembly languages are also useful when students learn basic concepts about computer organization, arithmetic operation, machine languages, addressing, instruction cycles, etc. When we need to implement a large complicated program, high-level languages such as C, C++, or Java are more frequently used. Similarly, students can learn a lot of concepts such as scalability, broadcast, one-to-one communication, performance, communication overhead, speedup, etc, through low-level languages such as MPI or PVM. These concepts are hard to obtain through high-level parallel programming languages due to the fact that many details are hidden in the language constructs. However, students can implement a relatively large parallel program

using a high-level parallel language such as OpenMP or HPF easily within a short period of time.

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 languages as well as the low level message passing paradigm is an important task for teaching parallel programming.

3 Why OpenMP and MPI

There are currently four major standards for programming parallel systems that were developed in open forums: High Performance Fortran (HPF) [6], OpenMP [2], PVM [3] and MPI [5].

HPF relies on advanced compiler technology to expedite the development of data-parallel programs [6]. 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.

The Parallel Virtual Machine (PVM) system 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 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.

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.

The most important reason for our adoption of OpenMP in a parallel programming class is that students can parallelize some realistic code (not toy problems) within a short period of time due to the ease of programming that it offers. Students can also experiment with different scheduling schemes such as static or dynamic loop scheduling policies within a short period of time, which would be impossible otherwise using MPI. Using OpenMP also has the advantage that task parallelism can be easily implemented by just inserting several OpenMP directives. By combining loop parallelism and task parallelism, better performance and higher scalability can be achieved. On the other hand, task parallelism is difficult to implement using MPI or HPF.

Another reason for our selection of OpenMP in our class is that we have an SGI Origin 2000 shared memory multiprocessor system in our department. A shared memory programming model fits in well.

Besides the above reasons, 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.

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, and teach other concepts such as various scheduling policies and task parallelism through a high-level parallel programming language. Since MPI and OpenMP are the most widely used languages in the two categories, these are selected to teach parallel programming.

4 Some Pitfalls with OpenMP

Because OpenMP is a high level parallel language, many details are hidden from a programmer. The good thing 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 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.

The second shortcoming with OpenMP 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 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.

For example, the following data distribution directive distributes the 4D array `H` on dimension 2:

```
!$sgi distribute_reshape H(*,BLOCK,*,*)
```

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.

Due to these pitfalls with 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. The following section details the strategy of using both OpenMP and MPI in the class.

5 Using MPI and OpenMP in Projects

The parallel programming class at GSU is a semester-long class for upper-level undergraduates and beginning graduate students. In it tutorials on MPI and OpenMP from the Ohio Supercomputing Center were used as supplements to a parallel algorithms textbook [9]. The code presented in the lectures uses both C and Fortran.

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.

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 purpose of the first project is simply to acquaint students with the system and programming environment of the Origin 2000. In this they write a simple addition code, and measure the parallel times using different numbers of processors.

In the second project, the students implement an MPI code to calculate π 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.

In the third project, students implement the parallel game of life. Through the assignment, students learn various domain decomposition strategies. All the above projects are implemented in MPI.

Once students understand the communication mechanisms of parallel computing systems, and communication overhead within a parallel code, it is time to introduce OpenMP. After briefly discussing the use of OpenMP and illustration of OpenMP through several examples, students are asked in the fourth project to initialize a huge array A so that each element has its index as its value. A second real array B which contains the running average of array A 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.

In the fifth project, students learn how to parallelize a real research Fortran code in OpenMP. The project contains several parts which includes the parallelization of the major loops in the code in OpenMP using both the loop and task parallelism of OpenMP. Thus the best scheduling policy, best chunk size for the policy, and both loop and task parallelism are obtained in the above two 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.

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. 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. At the end of the term, students need to present their findings and submit a paper.

The outcome of the course is very good. Based on student evaluations and comments on the course, most students feel that they learn a lot in the course. Some of the students have already applied the knowledge learned in the course to research projects supported by the NSF and Air Force. 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. The paper is being revised and potentially could be published in a conference. This would have been impossible if only MPI had been taught in the course.

6 Conclusion

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 seems that the strategy is well received by the students.

References

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, 300 pages, (October 2000)
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. C. H. Koelbel. The High performance Fortran handbook, MIT Press, Cambridge, Mass., (1994)
7. R. Miller. The status of parallel processing education, Computer, vol. 27, no. 8, pp. 40-43, (Aug. 1994)
8. C. H. Nevison. Parallel computing in the undergraduate curriculum, Computer, vol. 28, no. 12, pp. 51-53, (Dec. 1995)
9. M. J. Quinn. Parallel Computing - Theory and Practice, McGraw-Hill, INC., (1994)
10. B. Wilkinson and M. Allen. A state-wide senior parallel programming course, IEEE Trans. Educations, vol. 42, no. 3, pp. 167-173, (1999)