Crossroads
The ACM Student Magazine

Introduction to Programming Shared-Memory and Distributed-Memory Parallel Computers

by ***Cory Quammen***

Introduction

In designing a multiple processor computer, an important question needs to be addressed: How do processors coordinate to solve a problem? Processors must have the ability to communicate with each other in order to cooperatively complete a task. This article discusses two methods of inter-processor communication, each suitable for different system architectures.

One parallel computing architecture uses a single address space. Systems based on this concept, otherwise known as **shared-memory multiprocessors**, allow processor communication through variables stored in a shared address space. Another major architecture for parallel computers employs a scheme by which each processor has its own memory module. Such a **distributed-memory multiprocessor** is constructed by connecting each component with a high-speed communications network. Processors communicate to each other over the network [3].

The architectural differences between shared-memory multiprocessors and distributed-memory multiprocessors have implications on how each is programmed. With a shared-memory multiprocessor, different processors can access the same variables. This makes referencing data stored in memory similar to traditional single-processor programs, but adds the complexity of shared data integrity. A distributed-memory system introduces a different problem: how to distribute a computational task to multiple processors with distinct memory spaces and reassemble the results from each processor into one solution.

This article introduces two existing standards for programming shared-memory and distributed-memory multiprocessors. The first, OpenMP, is a set of compiler directives, library functions, and environment variables for developing parallel programs on a shared-memory multiprocessor. The second, Message Passing Interface (MPI), is an interface for a set of library functions that processors in a distributed-memory multiprocessor can use to communicate with each other. I will present the basic backround of each standard and its corresponding paradigm, the issues that arise when working with each, and a simple example program written to illustrate basic features of each standard. Finally, some conclusions will be made with regards to the advantages and disadvantages of each standard.

OpenMP

OpenMP is an open standard for providing parallelization mechanisms on shared-memory multiprocessors. Specifications exist for C/C++ and FORTRAN, several of the most commonly used languages for writing parallel programs. The standard provides a specification of compiler directives, library routines, and environment variables that control the parallelization and runtime characteristics of a program. Since it is a standard which is enjoying increasing levels of implementation, code written with OpenMP is portable to other shared-memory multiprocessors [8]. The compiler directives defined by OpenMP tell a compiler which regions of code should be parallelized and define specific options for parallelization. In addition, some precompiler tools exist which can automatically convert serial programs into parallel programs by inserting compiler directives in appropriate places, making the parallelization of a program even easier. One example is the now discontinued product from Kuck and Associates (now owned by KAI Software), Visual KAP for OpenMP [12].

OpenMP is based on a thread paradigm. A running program, referred to as a process, is allocated its own memory space by the operating system when the program is loaded into memory. Within a process, multiple threads may exist. A thread is an active execution sequence of instructions within a process. Threads within a process share the same memory space and can access the same variables. They have the advantage of allowing a process to perform multiple tasks seemingly simultaneously. For example, a web browser may have a thread that requests and receives web pages, another thread to render web pages for display on the screen, and yet another thread to "listen" for user input and respond appropriately. Without threads, the web browser might be required to block while waiting for a web page to download, preventing a user from doing things such as accessing a pull-down menu [10].

The thread paradigm is a logical choice for a shared-memory multiprocessor. The concept is based on the **fork-join** model of parallel computation. A master thread runs serially until it encounters a directive to fork off new threads. These threads can then be distributed and executed on different processors, reducing execution time since more processor cycles are available per time unit. Results of each threads execution can then be combined. A user can set the number of threads created for a parallel region by setting the environment variable `OMP_NUM_THREADS`, or the programmer can set it using the library call `omp_set_num_threads`. Figure 1 shows the execution model of a simple OpenMP program [8].

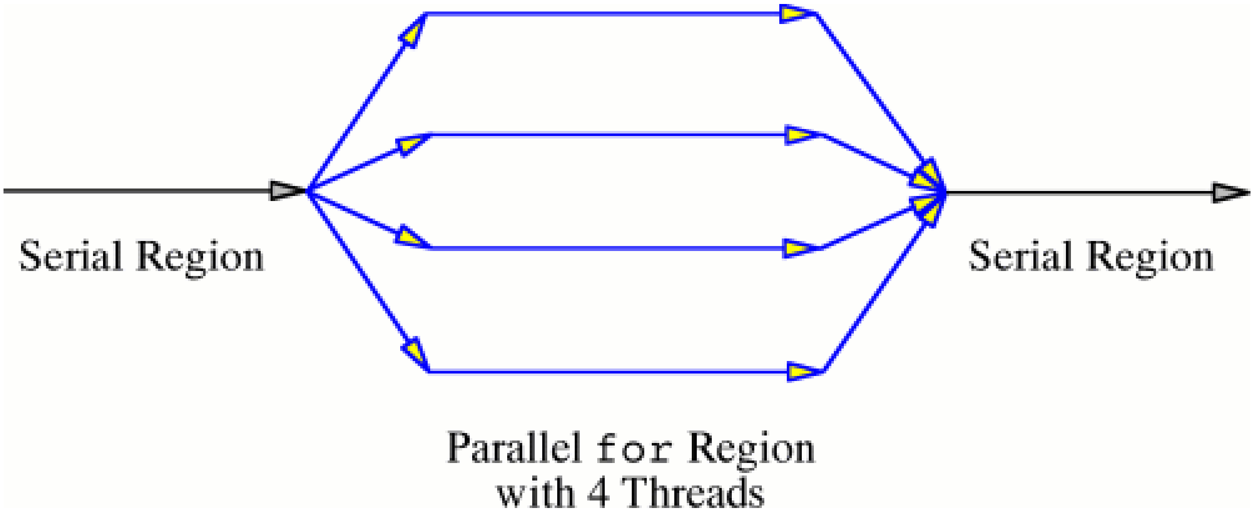

Figure 1. Program flow in an OpenMP execution model

Load Balancing in OpenMP

One of the issues that arises in any multiprocessing system is load balancing. **Load balancing** is the problem of distributing a task to a set of processors so that each processor has approximately the same amount of work to perform. As an analogy, consider a group of people bailing out a boat with buckets. There are two sizes of buckets, one twice as large as the other. Everyone removes the same number of buckets of water from the boat, but the larger buckets take twice as long to empty as the smaller buckets because they are heavier. The bailers with smaller buckets end up waiting a long time. In the time it takes the people with the larger buckets to empty one bucket-full, a person with a smaller bucket could have emptied two bucket-fulls. Clearly the water bailing could be more efficient.

In OpenMP, load balancing is often a problem of thread scheduling. By default, once a thread is finished with a region of code, it waits for the other threads to complete the same region, much as the water bailers in the example above. OpenMP has several options for thread scheduling to improve the inefficiency that can result in the default thread scheduling algorithm. Some options for thread scheduling in the context of a `for` loop, for example, include:

- static scheduling - all threads execute *n* iterations and then wait for all other threads to finish their *n* iterations
- dynamic scheduling - *n* iterations of the remaining iterations are dynamically assigned to threads that are idle
- guided scheduling - each time a thread finishes executing its assigned iterations, it is assigned approximately the number of remaining iterations divided by the number of threads [**8**]

It may seem that dynamic scheduling is the best scheduling algorithm available, but that is not always the case. A certain amount of overhead is involved in assigning additional iterations to a thread, slowing down the overall execution time of the OpenMP program. For this reason, finding an optimal *n* for dynamic scheduling can be difficult.

Issues in Shared-Memory Multiprocessor Programming

Programming in threaded environment brings up several issues that strictly serial programs never need to address. One problem that can arise when using threads is known as a **race condition**. A race condition occurs when more than one thread can modify the same variable or variables at the same time [10]. Consider two threads *T1* and *T2* in a finance program which share the task of compounding interest on customer loans in one database every month. Each thread compounds interest on half the loans in a database. Both must look up the balance and interest rate of the loan and then multiply the balance by the interest rate. The pseudo-code below shows the sequence of operations to be executed:

```
    balance = getBalance(loanID)
    rate = getRate(loanID)
    setBalance(loanID, balance * (1.0 + rate))
```

When both threads execute this code sequence, chaos ensues. The following is an example of the sequence of instructions the processor might execute:

```
T1: balance = getBalance(loan1)
T1: rate = getRate(loan1)
T2: balance = getBalance(loan2)
T1: setBalance(loan1, balance * (1.0 + rate))
```

Note that the balance for *loan1* will be set to the balance of *loan2* multiplied by *loan1*'s rate. Instructions in each thread may be executed on a processor or processors in an arbitrary order so that undesired execution results. Clearly, the program above would produce unhappy bank customers.

The problem in the example above is that both processes execute code in what is known as a **critical section**. Various methods of synchronization exist to prevent two threads from simultaneously executing critical sections. Typically, one thread will acquire a **lock** on a critical section, preventing other threads from executing code in that section. Threads that attempt to execute a locked critical section must wait until the lock is released by the thread that acquired it [10].

Preventing a race condition can lead to another problem known as **deadlock**. Deadlock can occur between two threads when each waits for a lock that the other holds. Since neither thread releases a lock until it acquire the other, both threads wait forever, essentially dead [10].

Thread Synchronization in OpenMP

OpenMP provides synchronization capabilities for the programmer to make avoiding the potential pitfalls of a threaded paradigm easier. One can specify a critical section using `#pragma critical`. The critical section can only be executed by one thread at a time, preventing any race conditions in that region.

One can also specify that certain variables be localized to individual threads using the directive `#pragma local(`*`list`*`)` or `#pragma threadprivate(`*`list`*`)` where *list* is a list of the variables to be privatized. By default, each thread can read and write every variable in a parallelized section of code. Declaring a variable local or private essentially creates a copy of the variable for each thread to use privately.

Of course, the notion of parallel computation is based on combining the computational efforts of multiple processors. Another directive provided by OpenMP is `#pragma reduction(`*`op`*`:` *`list`*`)`. Essentially, every variable in *list* is made private to each thread. When the threads finish executing the region of code in which the reduction variables are defined, a shared copy receives the value of each local copy combined by the operator. In this way, every thread receives the final computation of the parallel region [8].

Additionally, the library routines can be used to specify parallel execution parameters in certain program regions. Some parameters can be changed during the execution of the program, i.e., the number of threads forked in a parallel region. They can also be used for data synchronization schemes developed by the programmer. The environment variables of the specification are used to set characteristics of the parallel execution not defined by the compiler directives or library routines. For example, one can set the number of threads for parallel regions of code by changing the environment variable `OMP_NUM_THREADS`.

An Example Program in OpenMP

While OpenMP offers task-parallelism, it is often used to distribute work in `for` loops. Take Code Listing 1 for example. It is a loop written in C that sums a 100 element integer array, and we want to divide the work among four threads on a shared-memory multiprocessor. A serial processor executes each iteration through the loop, doing all the work. On a shared-memory multiprocessor, however, we could have one thread sum array elements 0 through 24, another processor to sum elements 25 through 49, another to sum elements 50 through

74, and yet another one to sum elements 75 through 99.

Code Listing 1

```
int main(int argc, int *argv[]) {
  int i, intArray[100];
  int sum = 0;

  /* Assume initArray initializes intArray to the numbers 1..100. */
  initArray(intArray, 100);

  /* Sum the array elements. */
   for (i=0;i<100;i++) {
     sum = sum + intArray[i];
   }
}
```

With OpenMP, all that needs to occur to parallelize this region is to add three `#pragma` directives to the source code instructing the compiler to parallelize the `for` loop. The modification is shown in Code Listing 2. Then, before we execute the program, we set the `OMP_NUM_THREADS` environment variable to 4, indicating we want this loop and any other loop parallelized with OpenMP to run on four threads.

Code Listing 2

```
int main(int argc, int *argv[]) {
   int i, intArray[100];
   int sum = 0;

   /* Store some values in intArray. */
   initArray(intArray, 100);

   #pragma parallel for      /* Make the for loop a parallel region */
   #pragma threadprivate(i)
   #pragma reduction(+: sum)
   for (i=0;i<100;i++) {
      sum = sum + intArray[i];
   }
}
```

The first `pragma` tells the compiler to parallelize the `for` loop while the second declares the variable `i` as a private variable for each processor. The localization of `i` is necessary for each processor to keep track of which iteration of the loop it is on. Finally, the reduction variable `sum` combines the work of each thread into the variable `sum` using the + operator.

Message Passing Interface

MPI is a standard for inter-process communication on distributed-memory multiprocessor. The standard has been developed by a committee of vendors, government labs, and universities [6]. Implementation of the standard is usually left up to the designers of the systems on which MPI runs, but a public domain implementation, MPICH, is available [2]. MPI is a set of library routines for C/C++ and FORTRAN. Like OpenMP, MPI is a standard interface, so code written for one system can easily be ported to another system with those libraries.

The execution model of a program written with MPI is quite different from one written with OpenMP. When an MPI program starts, the program spawns into the number of processes as specified by the user. Each process runs and communicates with other instances of the program, possibly running on the same processor or different processors. The greatest computational speedup will occur when processes are distributed among processors. Basic communication consists of sending and receiving data from one process to another, unlike OpenMP's thread communication via shared variables. This communication takes place over a high-speed network which connects the processors in the distributed-memory system.

A data packet sent with MPI requires several pieces of information: the sending process, the receiving process, the starting address in memory of the data to be sent, the number of data items being sent, a message identifier, and the group of processes that can receive the message. All of these items are able to be set by the programmer. For example, one can define a group of processes, then send a message only to that group.

Some collective communication routines do not require all of items. For example, a routine which allows one process to communicate with all other processes in a group when called by each of those processes would not require the specification of a receiving process since every process in the group should be a receiver.

In the simplest MPI programs, a master process sends off work to worker processes. Those processes receive the data, perform tasks on it, and send the results back to the master process which combines the results. More complex coordination schemes are possible with MPI, but they introduce challenges which will be discussed shortly. Note that the other processes run continuously from the launch of the program, a difference from the OpenMP fork-join model. Figure 2 shows the execution model of a basic MPI program.

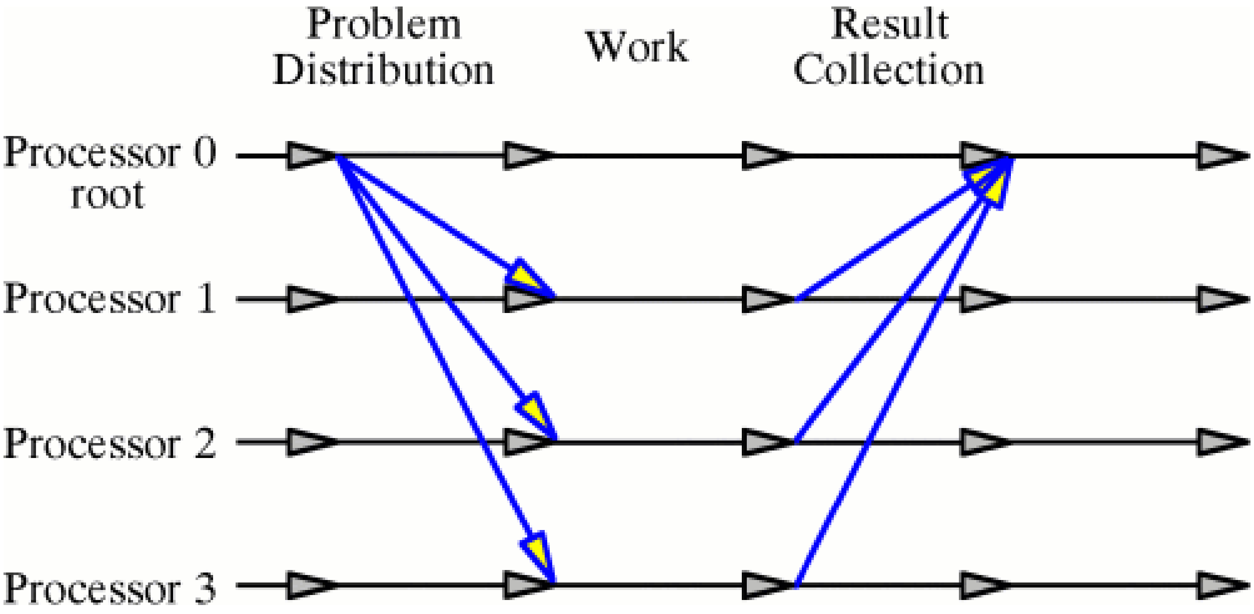

Figure 2. MPI execution model.

Load Balancing in MPI

Like programs on shared-memory multiprocessors, programs on distributed-memory multiprocessors must solve the problem of load balancing. The goal is to keep every process busy computing useful results while minimizing costly communication overhead. Let us briefly revisit the water bailer analogy. This time, assume that each bailer can bail the same amount of water in the same amount of time (using the same sized buckets) and that each bailer is in a different compartment of the ship completely sealed off from other compartments. We would want the water to be distributed evenly among the compartments in order to get the work done the most quickly.

For some problems, such as multiplying a large dense matrix by a vector, determining a good load balancing scheme is relatively straightforward: send *r*/*p* rows of the matrix and the entire vector to each processor where *r* is the number of rows and *p* is the number of processes, have each process compute a sub-matrix, and collect the results back into the final matrix. Distributing other problems, however, is not as easy. It can often be difficult or impossible to determine the amount of computation required by a subdomain of a problem. For example, distributing an irregular tetrahedral mesh in a fluid dynamics simulation is a challenging problem, one which has been investigated by [**11**, **5**].

Issues in Distributed-Memory Multiprocessor Programming

One of the biggest challenges in programming a distributed-memory multiprocessor is implementing efficient inter-process communication. Communication is not limited to the simple master-worker relationship shown in Figure 2. It may very well be the case that a process requires data or computed results from any other process during execution. It may also be the case that each process requires the same data sent from a single process, or that all processes require data from all the other processes. Ensuring process synchronization in these cases adds a level of complexity to programs developed on distributed-memory multiprocessors. Making communication efficient, that is, minimizing the overhead involved in message passing, adds further complexity.

MPI provides many communication routines to aid the programmer in developing inter-process communication. These routines include:

- barriers - points within a program where each process waits until all other processes get there. When this occurs, each process resumes execution.
- blocking sends and receives - message passing routines which cause a process to wait until a message is sent or received before continuing execution
- non-blocking sends and receives - message passing routines which do not cause a process to wait until a message is sent or received before continuing execution
- collective communications - messages sent in one of three ways: one process sends other processes in a process group a message, all processes in a group send messages to all processes in the same group, and all processes in a group send a message to one process

Table 1 lists some routines defined in MPI and describes their use [4].

MPI Routine	Use
MPI_Barrier	Cause all processes in a group to block until all other processes reach this routine
MPI_Send	Send a process data; block until received
MPI_Recv	Receive data from another process; block until sent
MPI_Isend	Send a message; do not block
MPI_Irecv	Receive a message; do not block
MPI_Probe	See if a message is waiting; block until message is detected
MPI_Iprobe	See if a messge is waiting; does not block
MPI_Bcast	Send data to all processors in a group
MPI_Reduce	Collect a variable from all processors in a group with a combination operation
MPI_Allreduce	All processes receive the reduction variable after it has been combined

MPI_Gather	Collects data from all processes in a group into an array. Data is of same size
MPI_Allgather	All processes receive collected data from all other processes in a group. Data is of same size
MPI_Scatter	Distribute data to all processes in a group. Data is of same size
MPI_Gatherv	Collects data from all processes in a group into an array. Data may have different sizes
MPI_Scatterv	Distribute data to all processes in a group. Data may have different sizes
MPI_Alltoall	Distribute data to all processors in a group from all processes in the group. Data is of same size
MPI_Alltoallv	Distribute data to all processors in a group from all processes in the group. Data may have different sizes

For a more thorough overview of MPI, see either [9] or [7].

An Example Program in MPI

As an example of how we can use MPI, we can convert the code in Code Listing 1 to one that will run on a distributed-memory multiprocessor with an MPI library. We will assume that the user has decided to have MPI run four copies of the program. The MPI version is listed in Code Listing 3.

Code Listing 3

```
int main(int argc, int *argv[]) {
   int        i, numberProcessors,
              myProcessorNumber, sum, result;
   int        intArray[100];
   int        myChunk[25];
   int        err; /* We will ignore errors in this code. */
   MPI_Status status;

   /* Initialize MPI. */
   err = MPI_Init(&argc, &argv);
   err = MPI_Comm_size(MPI_COMM_WORLD, &numberProcessors);
   err = MPI_Comm_rank(MPI_COMM_WORLD, &myProcessorNumber);

   /* Take two different actions depending on
      whether this is the main processor or not.*/
```

```
if (myProcessorNumber == 0) {

   /* Assume initArray performs the same function as in Code
      Listings 1 and 2.
   initArray(intArray, 100);

   /* I am the main processor, so I distribute
      the problem to processors. */
   for (i=1; i<numberProcessors; i++) {

      /* Send chunks of array out to each processor.  Arguments
         are: pointer to starting address, number of items sent,
         type of items sent, destination processor, message id,
         group of processors which are eligible to receive the
         message (in the case of MPI_COMM_WORLD, all of them) */
      err = MPI_Send(&intArray[i*25], 25, MPI_INT, i, 100,
               MPI_COMM_WORLD);

      /* Copy main processor's data into its chunk array. Assume
         function copyArray copies n integers from one integer
         array to another integer array. */
      copyArray(intArray, myChunk, 25);
   }
} else {

   /* I am not the main processor, so I receive a chunk to work
      on. Arguments are: buffer in which to receive data, number of
      items sent, type of items sent, destination processor,
      message id, group of processors which are eligible to receive
      the message, pointer to a status variable (contains
      information about status of transmission) */
   err = MPI_Recv(&myChunk[0], 25, MPI_INT, 0, 100, MPI_COMM_WORLD,
      &status); }

/* Now that the problem is distributed, solve it. Each processor will
   have its share of the work in the myChunk array. */
sum = 0;

for (i=0;i<25;i++) {
   sum = sum + myChunk[i];
}

/* Print out the sums each processor calculates. */
```

```
   printf("%d summed %d\n", myProcessorNumber, sum);

   /* Send the results back to the main processor. */
   if (myProcessorNumber == 0) {

      /* I receive the partial results and compute the total result. */
      for (i=1;i<numberProcessors;i++) {
         err = MPI_Recv(&result, 1, MPI_INT, i, 200, MPI_COMM_WORLD, &status);
         sum = sum + result;
      }

   } else {
      /* I am not the main processor, so I send off my results. */
      err = MPI_Send(&sum, 1, MPI_INT, 0, 200, MPI_COMM_WORLD);
   }

   /* Do something with the final result. */


   /* Have MPI perform its shut-down. */
   err = MPI_Finalize();
}
```

Conclusions

Both shared-memory multiprocessors and distributed-memory processors have advantages and disadvantages in terms of ease of programming. Porting a serial program to a shared-memory system can often be a simple matter by adding loop-level parallelism with OpenMP, but one must be aware of race conditions, deadlocks, and other problems associated with the paradigm that may arise. For programmers used to a thread paradigm, moving to OpenMP is relatively straightforward. Writing an MPI program, on the other hand, involves the additional problem solving of how to divide the domain of a task among processes with separate memory spaces. Coordinating processes with communication routines can be quite a challenge. There is no concern over thread issues, but data synchronization is still a consideration.

Where OpenMP has an advantage in ease of programming and ease of porting serial programs, shared-memory systems in general have poor scalability. Adding additional processors to a shared-memory multiprocessor increases the bus traffic on the sytem, slowing down memory access time and delaying program execution. Distributed-memory multiprocessors, however, have the advantage that each processor has a separate bus with access to its own memory. Because of this, they are much more scalable. In addition, it is possible to build large, inexpensive **cluster computers** by using commodity systems connected via a network. The Beowulf Project is developing clusters from Linux systems using MPI [1]. However, latency of the network connecting the individual processors is an issue, so efficient communication schemes must be devised.

Acknowledgements

I would like to thank the Army High Performance Computing Research Center in Minneapolis, MN, for selecting me to participate in their 2001 Summer Institute for High Performance Computing and introducing me to topics in parallel computing. Thanks also to the helpful comments of the anonymous reviewers.

References

1

The Beowulf Project. **http://www.beowulf.org** (October, 2001).

2

Gropp, W., Lusk, E., Doss, N., Skjellum, A. A High-Performance, Portable Implementation of the Message Passing Interface Standard. Parallel Computing. Vol. 22. No. 6. pp. 789-828. Sept., 1996.

3

Hennessy, J. and Patterson, D. *Computer Organization & Design*. Morgan Kaufmann Publishers. San Francisco, 1998.

4

Johnson, A. Introduction to MPI. Army High Performance Computing Research Center Summer Institute Course Notes. 2001.

5

Karypis, G., Kumar, V. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. Technical Report 95-035, University of Minnesota, 1995.

6

MPI - The Message Passing Interface Standard. **http://www-unix.mcs.anl.gov/mpi/** (October 1, 2001).

7

MPI Forum. MPI: A Message Passing Interface. In Proceedings of 1993 Supercomputing Conference, Portland, Washington. November, 1993.

8

OpenMP C and C++ Application Program Interface. **http://www.openmp.org/specs/**. OpenMP Architecture Review Board (October, 1998).

9

Pacheco, P. *Parallel Programming with MPI*. Morgan Kaufmann Publishers. San Francisco, 1996.

10

Silberschatz, A., Galvin, P., Gagne, G. *Applied Operating System Concepts*. John Wiley and Sons. New York, 2000.

11

Sohn, A. Simon, H. A Scalable Parallel Dynamic Paritioner for Adaptive Mesh-based Computations. Proceedings of Supercomputing 1998, Orlando, Florida.

12

Visual KAP for OpenMP. **http://www.kai.com/vkomp/_index.html** (Oct. 28, 2001).

Biography

Cory Quammen (**cquammen@acm.org**) is a senior completing an undergraduate degree in computer science at Gustavus Adolphus College in St. Peter, MN. After completing his Ph.D., he plans to pursue a research and development position in computer science or teach at a university.