Issues in Accurate and Reliable Use of Parallel Computing in Numerical Programs

William D. Gropp

August 27, 2004

1 Introduction

Parallel computing, as used in technical computing, is fundamentally about achieving higher performance than is possible with a uniprocessor. The unique issues for accurate and reliable computing are thus directly related to this desire for performance. This chapter briefly covers the major types of parallel computers, the programming models used with them, and issues in the choice and implementation of algorithms.

2 Special Features of Parallel Computers

The fundamental feature of parallel computation is multiple *threads of control*. That is, a parallel computer can execute more than one operation at a time. The most common parallel computers today can execute different instructions on different data; these are called multiple instruction multiple data (MIMD) computers. Another type executes the same operation on different data (same instruction multiple data, or SIMD); while some of the issues discussed in this chapter apply to SIMD computers, many are the result of the ability of different processors within MIMD systems to execute different instructions at the same time. The discussion focuses on MIMD parallel computers.

Parallel computers can also be distinguished by the way they provide access to data. A computer that provides hardware support for a common address space accessible by all processors is said to provide *shared memory*. The most common systems are *symmetric multiprocessors* (SMPs), where symmetric here means that all processors are equal; that is, no processor is designated as the "operating system" processor. The other major type of parallel computer does not provide support for a common address space and instead consists of essentially separate computers (often called nodes), each with its own memory space, and an interconnection network that allows data to be communicated

between the nodes. These machines are said to be *distributed memory* parallel computers. Most of the largest parallel computers are distributed-memory computers.

2.1 The Major Programming Models

The two most common programming models are distinguished by how they treat the memory of each processor being used by a single parallel program. In the *shared-memory* programming model, all of the memory is available to every thread of control. In the *distributed-memory* (or *shared-nothing*) programming model, only the memory of the process is available to that process (and a parallel program is made up of many processes). The most common instances of the shared-memory model are threads and compiler-directed parallelism through interfaces such as OpenMP [15]. The most common distributed-memory model is the message-passing model, specifically using the Message Passing Interface (MPI) [13, 14].

Another important programming model is the *bulk synchronous parallel* (BSP) model [12, 1]. This model emphasizes organizing a parallel computation around "supersteps" during which each processor can act on local data and initiate communications with other processors. At the end of the superstep a barrier synchronization is performed. One of the most common implementations of this model, the Oxford BSP library [10], provides remote memory operations (also known as put/get programming). Aspects of the BSP model were incorporated into version 2 of the MPI standard.

We note that the programming model and the hardware model are separate. For example, one can use the message-passing programming model on shared-memory parallel computers. In all cases, the programming models address two issues: *communication* of data between two (or more) threads and *coordination* of the exchange of data.

Hybrids of these programming models are also common; many parallel systems are built from a collection or cluster of SMPs, where shared memory is used within each SMP and message passing is used between SMPs. This more complex model shares the issues and features of both and will not be discussed further.

For a more comprehensive discussion of parallel computer hardware, see [5, Chapter 2] or [3].

2.2 Overview

This chapter is divided into two main sections. Section 3 discusses the impact of parallel computing on the choice of numerical algorithms. Because the cost of coordinating separate parallel threads of execution is not zero, just counting floating-point operations is often not an effective way to choose among numerically stable algorithms for a parallel computer. Section 3 describes a simple, yet effective time-complexity model for parallel computations and provides an

example of its use in understanding methods for orthogonalizing a collection of vectors that are distributed across a parallel computer.

Section 4 discusses the implementation issues, with emphasis on the hazards and pitfalls that are unique to parallel programs. This section provides only an introduction to some of the issues in writing correct parallel programs but it does cover the most important sources of problems. The chapter closes with recommendations for developing accurate and correct parallel programs.

3 Impact on the Choice of Algorithm

The costs of communicating data between processes or coordinating access to shared data strongly influence the choice of algorithms for parallel computers. This section illustrates some of the pitfalls in emphasizing performance over good numerical properties.

3.1 Consequences of Latency

The cost to communicate data between processes in the distributed-memory, or message-passing, model is often modeled as

$$T = s + rn, \tag{1}$$

where s is the latency (or startup cost), r the inverse of the bandwidth (or rate), and n the number of words. Typical numbers are $s = 10 - 50$ microseconds and $r = 10^{-7}$ seconds/word. The factors that contribute to the latency include the time to access main memory (itself relatively slow) and any software overheads. (A more detailed model, called logP, separates the overhead from the latency, [2].) For completeness in what follows, let f be the time for a floating-point operation. When conducting performance studies, it is often valuable to consider the quantities s/f and r/f so as to make the terms relative to the speed of floating-point computation.

The cost to communicate and coordinate data in a shared-memory model is more complex to model because of the interaction with the memory system hardware. However, the cost of memory-atomic operations such as locks, which are often necessary to coordinate the activities of two threads, is also on the order of a microsecond. With modern processors running at speeds of over 2 GHz (2×10^9 cycles/second), a microsecond corresponds to roughly 2,000 floating-point operations. Because of this relatively high cost, algorithms for parallel processors often trade more local computation for fewer communication or coordination steps.

Orthogonalization of Vectors. A common operation in many numerical algorithms is adding a vector $\mathbf{u}$ to a set of orthonormal vectors $\mathbf{v}_i$, $i = 1, \ldots, n$ to form a new vector, $\mathbf{v}_{n+1}$, that is orthonormal to the original set. In exact

arithmetic, the new vector $\mathbf{v}_{n+1}$ is given by

$$\begin{aligned} \mathbf{v}' &= \mathbf{u} - \sum_{i=1}^{n} \mathbf{v}_i(\mathbf{u} \circ \mathbf{v}_i) \\ \mathbf{v}_{n+1} &= \frac{\mathbf{v}'}{\|\mathbf{v}'\|_2}. \end{aligned}$$

This is the Gramm-Schmidt process. Note that the individual inner products are independent; thus they can be computed with a single parallel reduction operation (over a vector with n entries). Because reduction operations, particularly on systems with many processors, are relatively expensive, this is an attractive formulation. Using the performance model in Equation 1, and assuming that the vectors are of length (dimension) m, one can compute the cost of this approach on a parallel distributed memory computer with p processors as roughly

$$\begin{aligned} T_{gs} &= (s + rn)\log p + (s + r)\log p + nmf/p \\ &= (2s + r(n+1))\log p + nmf/p. \end{aligned}$$

The cost of this algorithm, for large m, scales as m/p, with a term, due to the inner products, that grows as $\log p$.

Unfortunately, the classical Gramm-Schmidt algorithm is well known to be unstable in floating point arithmetic (see, e.g., [7]). Numerical analysis texts often recommend the *modified Gramm-Schmidt* method:[1]

$$\begin{aligned} &\mathbf{v}'_0 \leftarrow u \\ &\text{for } i = 1, \ldots, n \ \{ \\ &\quad \mathbf{v}'_{i+1} \leftarrow \mathbf{v}'_i - (\mathbf{v}'_i \circ \mathbf{v}_i) \\ &\quad \mathbf{v}'_{i+1} = \frac{\mathbf{v}'_{i+1}}{\|\mathbf{v}'_{i+1}\|_2} \\ &\} \\ &\mathbf{v}_{n+1} = \mathbf{v}'_{n+1}. \end{aligned}$$

While this is numerically superior to the unmodified form, the cost on a parallel computer is much greater because of the need to compute each inner product separately, since the ith step relies on the results of step $i-1$. The cost of the modified Gramm-Schmidt method on a parallel computer can be modeled as

$$T_{mgs} = (n+1)(s+r)\log p + nmf/p.$$

On message-passing platforms, s is of the order of 10 μsec, or roughly $20,000$ floating-point operations. (Even on shared-memory platforms, s is on the order

[1] Other methods could be used that have even better stability properties; the same analysis used here may be used to evaluate the performance of those methods on parallel computers.

of 1 μsec if remote memory is involved, or roughly the same time as for $2,000$ floating-point operations.)

An important lesson from this discussion is that the performance goals of parallel computing can conflict with good numerical behavior. Further, for a programmer unskilled in numerical analysis, the transformation from the poorly performing modified Gramm-Schmidt to the much faster unmodified Gramm-Schmidt will be an obvious one. In addition, parallel performance tools are likely to draw the attention of the programmer to this part of the computation.

The situation isn't lost, however. Versions of Gramm-Schmidt that iterate on the unmodified version can often be used. These have time complexity

$$T_I = k((2s + r(n+1)) \log p) + knmf/p, \tag{2}$$

where k iterations are taken. This algorithm would be chosen only on a parallel computer because the floating-point cost is larger than that of either original Gramm-Schmidt algorithm. In the parallel case, however, it combines adequate numerical properties with good parallel performance. In the final analysis, one must balance the numerical properties of an algorithm with the performance properties, just as partial rather than complete pivoting is usually considered adequate for the solution of linear systems of equations on uniprocessors. For example, in the GMRES method, classical Gramm-Schmidt is often considered sufficient, particularly when only modest accuracy is required.

3.2 Consequences of Blocking

Another algorithmic technique that is used in parallel computing is blocking. Blocking is a problem decomposition technique that divides a problem into smaller blocks, which may be better able to take advantage of the computer. Examples of blocking include algorithms for dense matrix-matrix multiply that reduce the problem into one of multiplying subblocks of the matrix. Just as blocking can make better use of memory systems on uniprocessors (e.g., BLAS3 [4]), blocking in parallel computers can make better use of the communication interconnect by reducing the number of separate communication events. In the case of a shared-memory system, it may also reduce the number of locks or other techniques used to avoid race conditions (described in Section 4.1). However, blocked algorithms, because they perform operations in a different order from that of unblocked algorithms, have different numerical properties. This effect of ordering operations is most easily seen in the simple operation of a parallel dot product.

Dot Products. When discussing the orthogonalization example, we assumed the existence of a fast parallel dot product routine. Such routines exist, but they rely on associating the arithmetic so that the values are added together in a tree-like fashion. This is not the same order of operations that would normally be used on a single processor. Figure 1 shows two possible orderings for adding the results from four processors. Note that a careful numerical routine may choose an order for the summation that depends on the magnitude of the values.

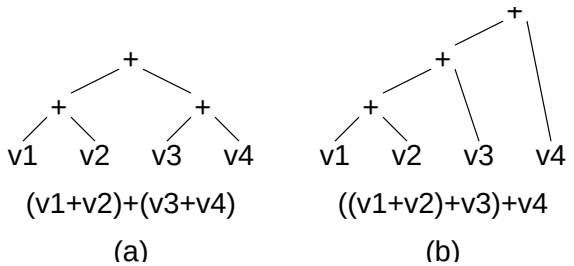

Figure 1: Two orderings for summing 4 values. Shown in (a) is the ordering typically used by parallel algorithms. Shown in (b) is the natural "do loop" ordering.

One result is that many parallel programs lack *bitwise reproducibility*. That is, because efficient parallel execution may require re-associating floating-point operations, the exact values computed may depend on the number of threads of execution. Hence, validating a parallel program is more difficult because one of the most common validation approaches is to require bitwise identical results whenever a program is modified. Addressing the issue of validating a parallel computation against a nonparallel (uniprocessor) computation is difficult, and it is further complicated by implementation issues specific to parallel computers.

4 Implementation Issues

Parallel computing provides new ways to write incorrect programs. These are a result of the tension between performance and correctness. That is, most programming models for parallel computers have made a deliberate choice to present opportunities for performance at the cost of a greater chance for programmer error. This section details a few of the most common programming errors for both shared-memory and message-passing programming models.

4.1 Races

One of the most dangerous and common errors in shared-memory parallel programs is a *race condition*. This occurs in parts of the code where a race between two or more threads of execution determines the behavior of the program. If the "wrong" thread wins the race, then the program behaves erroneously. To see how easy it is to introduce a race condition, consider the following OpenMP program:

```
integer n
n = 0
!$omp parallel shared(n)
n = n + 1
!$omp end parallel
```

The intent of this program is to count the number of threads used in the parallel region.[2] Variations on this program are commonly used in programming texts to introduce parallel programming using threads. Unfortunately, this program has a race condition. The statement `n = n + 1` is not executed atomically (that is, without possibility of interruption) on any modern processor. Instead, it is split into a sequence of instructions, such as the following, presented here in a generic assembly language:

```
load n, r1
add  #1, r1
store r1, n
```

In words, this loads the value `n` into a register, adds one to that value, and stores the result back into the memory location `n`. A possible execution sequence with two threads is as follows (comments on the execution are to the right of the exclamation point, and time proceeds down the page):

Thread 0	Thread 1
load n, r1 ! r1 has value 0	
	load n, r1 ! r1 also has value 0
add #1, r1 ! r1 had value 1	
	add #1, r1 ! r1 has value 1
store r1, n ! n now has value 1	
	store r1, n ! n gets value 1 again

Each thread is executing on its own processor with its own set of registers. After this sequence, the program will have counted one rather than two threads.

The consequences of races are made more serious by the fact that in many codes, the races are almost always "won" as the programmer expects. Thus, codes with races will often work and pass tests, even though the possibility of losing the race (and thus returning an incorrect result) remains. Validating such codes is very difficult; races are in fact a common source of error in complex programs such as operating systems.

Avoiding such race conditions is difficult. One approach was suggested by Lamport [11]. In his *sequential consistency* model, the processor must execute the program as if the lines of the program are executed by the different processors in some interleaved (but not interchanged) order. With the interpretation of "line" as "line of the program in the programming language," sequential consistency will avoid the race illustrated above. Unfortunately, the performance consequences of sequential consistency are severe, leading computer science researchers to define various weaker consistency models and for most programming languages to provide no guarantees at all. Most processors provide a way to guarantee atomic execution of some operations. However, this always comes with a performance cost. As a result, programming models such as OpenMP do not provide atomic execution (and hence sequential consistency) by default.

[2] There are easier ways to perform this particular operation in OpenMP; this example was picked because it is easy to explain and similar code is common in real applications.

In fact, no major programming model provides sequential consistency; all major parallel programming models *allow* the programmer to explicitly enforce sequential consistency, but none ensures sequential consistency.

4.2 Out-of-Order Execution

In writing algorithms with the shared-memory model, one often has to ensure that data on one thread is not accessed by any other thread until some condition is satisfied. For this purpose *locks* are often used. The following shows a simple example of two threads controlling access to shared data with a lock:

Thread 0	Thread 1
lock	(wait for data to be available)
update array	
unlock	
	lock
	access array
	unlock

Unfortunately, locks are often quite expensive to execute. Programmers are often tempted to use a simple flag variable to mediate access to the data, as shown in the following:

Thread 0	Thread 1
flag=0	
update array	do while(flag .eq. 0)
flag=1	
	access array
	flag = 0

However, this code may not execute correctly. The reason is there is no guarantee that either the compiler or the processor will preserve the order of the operations to what appears to be (within a single thread) independent statements. Either the compiler or the CPU may thus move the assignment `flag=1` before the array update is complete (a very aggressive optimizing compiler may move the assignment `flag=1` *before* the update to avoid what the compiler sees as an unnecessary store of zero to the storage location `flag`). In general, even within a single thread, the order of operations is not guaranteed. Special assembly language instructions can be used to force the processor to complete memory operations before proceeding, but these must usually be inserted explicitly by the programmer.[3]

If the array update is part of a time integration, then this bug will introduce a Δt error into the calculation. This is one of the worst possible bugs, because it reduces the accuracy of the computation rather than providing a clear indication of trouble.

[3] C programmers may use `volatile` to avoid some but not all of these problems.

In addition to these correctness issues, there exist many issues related to performance that are specific to shared-memory parallel programs. One important case is called *false sharing*. This occurs when a parallel program has two separate memory locations on the same cache line. Even though each item is accessed only by a single thread, the fact that the two items share a cache line can cause serious performance problems. For the programmer, the situation is made more difficult by the fact that most programming languages try to insulate the programmer from details of the hardware, thus making it more likely that performance problems will occur.

The issues described so far have pertained to a shared-memory programming model. The message-passing model has its own share of issues.

4.3 Message Buffering

Many message-passing programs are immune to race conditions because message passing combines data transfer and notification into a single routine and because there is no direct access to the memory of another process. However, message-passing programs are susceptible to resource limits. Consider the following simple program in which each process sends n words to the other:

```
Process 0                    Process 1
dest = 1;                    dest = 0;
Send( buf, n, dest );        Send( buf, n, dest );
Recv( rbuf, n, dest );       Recv( rbuf, n, dest );
```

This is an example of an *unsafe* program. It is unsafe because it depends on the underlying system to *buffer* the message data (the array `buf` on both processes in this example) so that the `Send` operations can complete. At some size `n`, there cannot be enough space available, and the processes will wait forever (*deadlock*). In some ways, the limit on the size of `n` under which a program will complete is the analogue of machine precision—it is a number that reflects a reality of computing, a number that we wish was infinite but is all too finite.

The real risk in message passing comes from using buffered send operations in unsafe ways. The program may operate correctly for some inputs but deadlock for others. There are several ways to avoid this problem.

- Use a synchronous send instead of a regular send. In MPI terms, use `MPI_Ssend` instead of `MPI_Send`. This ensures that the program is independent of buffering, that is, that unsafe programs will deadlock for all input, not just some. This approach simplifies validation of the program.
- Use explicit buffer spaces, and manage them carefully. In MPI terms, use `MPI_Buffer_attach` and the `MPI_Bsend` routine.
- Avoid point-to-point operations entirely, and use collective operations such as broadcast and reduce.
- Use nonblocking send operations; in MPI terms, these are the family of routines that include `MPI_Isend` and `MPI_Send_init`.

All of these approaches have drawbacks. Their major advantage is that they make the code's dependence on buffering explicit rather than relying on the implicit buffering within the message-passing system.

4.4 Nonblocking and Asynchronous Operations

A common technique for working around high-latency operations is to split the operation into separate initiation and completion operations. This is commonly used with I/O operations; many programmers have used nonblocking I/O calls to hide the relatively slow performance of file read and write operations. The same approach can be used to hide the latency of communication between processes. For example, in MPI, a message can be sent by using a two-step process:

```
MPI_Request request;
MPI_Isend( buf, n, MPI_INT, dest, tag, comm, &request );
... other operations and program steps
MPI_Wait( &request, MPI_STATUS_IGNORE );
```

This sequence sends `n` integers from the calling process to the process `dest` and allows the code to perform other operations without waiting for the data to be transfered until the call to `MPI_Wait`. Because MPI (and most other approaches to message-passing) use libraries rather than introducing a new parallel language, there is no way to enforce this requirement or to confirm that the user has not violated it. As a result, this approach is not without its dangers [9]. In particular, the user must not change (or even access) the elements of `buf` until after the `MPI_Wait` completes.

Unfortunately, programming languages provide little direct support for nonblocking operations, and hence the programmer must exercise care when using these operations. Programming languages such as Fortran can make such approaches (for asynchronous I/O as well as for parallel computing) quite hazardous, because a Fortran compiler may not preserve the data array after the call that initiates the operation (`MPI_Isend` above) returns. A particular example in Fortran is array sections; the compiler may not preserve these, even in simple cases, after the call returns. See [14, Section 10.2.2] for a discussion of this problem in the context of MPI and Fortran. Another example is variable scope. Particularly in C and C++, many variables are local to the routine in which they are used; their space is reused once the routine exits (the variables are usually allocated on a stack that is shared by all routines within the same thread). A nonblocking operation should ensure that a variable remains allocated (also called "in scope") until the nonblocking operation completes.

4.5 Hardware Errors

Because parallel computers are often used for the most challenging problems, another source of problems is the low, but not zero, probability of an error in the computer hardware. This has sometimes led to problems with long-running simulations, particularly with high-performance interconnect networks.

For example, an interconnect that has an error rate of 1 in 10^{12} bits sent and a data rate of 100 MB/sec will have an error roughly every 20 minutes (per link!). In the past, the role of error rates in the interconnect has not always been recognized, leading to unexpected errors in the computation. Since most parallel programming models, both libraries and languages, specify error-free behavior of the hardware, hardware and system software implementors work together to provide an error-free system. Unfortunately, not all parallel systems have properly addressed these issues, and often the application developer must check that internode communications are reliable.

4.6 Heterogeneous Parallel Systems

A *heterogeneous parallel system* is one in which the processing elements are not all the same. They may have different data representations or ranges. For example, some processors may have 32-bit integers stored with the most significant byte first, whereas others may have 64-bit integers with the least significant byte first. Such systems introduce additional complexity because many algorithms assume that given the exact same input data, the same floating-point operations will produce, bit for bit, the same output data. For example, in some cases, a parallel algorithm will distribute the same (small) problem and then have each processor compute a result rather than try to parallelize a small problem; this approach is sometimes used in multigrid and domain decomposition methods for solving the coarse-grid problem.

On a heterogeneous system, such an approach may not be valid, depending on how the results are used. Additional issues concern the different handling of the less-specified parts of the IEEE 754 floating-point specification, such as the exact meaning of the bits in a NaN (not a number). These issues must be considered when contemplating the use of heterogeneous systems.

5 Conclusions and Recommendations

Algorithms for parallel computers must often trade additional floating-point operations against communication. Moreover, providing adequate parallelism may require using different algorithms. These algorithms must be chosen carefully because it is all too easy to use numerically unstable algorithms for common operations.

Writing correct parallel programs is also difficult. There is really no substitute for disciplined coding practices, particularly for shared-memory programs. One approach is to carefully annotate all accesses to shared or nonlocal data; these annotations may be processed by program development tools to identify potential race conditions. Some tools along these lines have been developed (e.g., [6]), but much more needs to be done. In particular, an approach is needed that allows an algorithm with no race conditions to be expressed in a way that provably cannot introduce a race condition.

For message-passing codes, no code should rely on system buffering for correct operation. In MPI terms, any program should still work if all uses of `MPI_Send` are replaced with `MPI_Ssend` (the synchronous send). Fortunately, it is easy to test this by using the MPI profiling interface.

Moreover, the recommendations must depend on the scale of the parallelism, that is, the number of processing elements. If the number is relatively small (on the order of four to eight), then parallelism should be considered cautiously. The reason is that with the rapid increase in computing power (typically doubling every 18 months), improving performance by a factor of four is like waiting three years for a faster computer. This does not mean that parallelism at this scale should not be used, only that it should be used with care and without going to extreme lengths to get better performance. For example, such applications should use robust synchronization mechanisms in a shared-memory code rather than relying on write-ordering and flags, as described in Section 4.2.

At much greater scale, particularly in the tens of thousands of processors, distributed-memory computers programmed with message-passing dominate. Applications at this scale both are more sensitive to performance scaling and are unlikely to be replaced by a nonparallel application in the forseeable future. These codes should exploit modern software engineering practices to isolate the parallel implementation into a small number of well-tested library routines. Their algorithms should be carefully chosen for the scale at which they will operate, and attention must be paid to the effects of ordering of operations and decompositions based on the layout of the parallel computer's memory.

With proper care, parallel computers can be used effectively for numeric computation. One approach is to isolate within a numerical library most of the issues described in this chapter. Additional challenges in creating numerical libraries for high-performance computing systems are discussed in [8].

Acknowledgment

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract W-31-109-ENG-38.

References

[1] R. H. Bisseling, *Parallel Scientific Computation: A Structured Approach Using BSP and MPI*, Oxford University Press, Oxford, UK, March 2004.

[2] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, *LogP: Towards a realistic model of parallel computation*, ACM SIGPLAN Notices, 28 (1993), pp. 1–12.

[3] D. E. Culler, J. P. Singh, and A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaufmann, San Francisco, 1999.

[4] J. Dongarra, J. D. Croz, I. Duff, and S. Hammarling, *A set of level 3 Basic Linear Algebra Subprograms*, ACM Trans. Math. Soft., 16 (1990), pp. 1–17.

[5] J. Dongarra, I. Foster, G. Fox, W. Gropp, K. Kennedy, and L. Torczon, eds., *The Sourcebook of Parallel Computing*, Morgan Kaufmann, San Francisco, 2002.

[6] D. Engler, B. Chelf, A. Chou, and S. Hallem, *Checking system rules using system-specific, programmer-written compiler extensions*, in Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI 2000), San Diego, 2000.

[7] G. H. Golub and C. F. V. Loan, *Matrix Computations*, Johns Hopkins Studies in the Mathematical Sciences, The Johns Hopkins University Press, Baltimore, third ed., 1996.

[8] W. Gropp, *Exploiting existing software in libraries: Successes, failures, and reasons why*, in Object Oriented Methods for Interoperable Scientific and Engineering Computing, M. Henderson, C. Anderson, and S. L. Lyons, eds., Philadelphia, 1999, SIAM, pp. 21–29.

[9] P. B. Hansen, *An evaluation of the Message-Passing Interface*, ACM SIGPLAN Notices, 33 (1998), pp. 65–72.

[10] J. M. D. Hill, B. McColl, S. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. H. Bisseling, *BSPlib: The BSP programming library*, Parallel Computing, 24 (1998), pp. 1947–1980.

[11] L. Lamport, *How to make a multiprocessor computer that correctly executes multiprocess programs*, IEEE Transactions on Computers, C-28 (1979), pp. 690–691.

[12] W. F. McColl, *BSP programming*, in Proc. DIMACS Workshop on Specification of Parallel Algorithms, G. Blelloch, M. Chandy, and S. Jagannathan, eds., Princeton, May 1994, American Mathematical Society.

[13] Message Passing Interface Forum, *MPI: A message passing interface standard*, International Journal of Supercomputer Applications, 8 (1994), pp. 159–416.

[14] ——, *MPI2: A message passing interface standard*, High Performance Computing Applications, 12 (1998), pp. 1–299.

[15] *OpenMP Fortran Application Program Interface, Version 2.0.* `http://www.openmp.org`, November 2000.