Programming Multiprocessor Machines

Greg Astfalk

Hewlett-Packard Company
Convex Division
PO Box 833851
Richardson, TX 75083–3851
astfalk@rsn.hp.com

Introduction

Reasonably achieving the solution to today's demanding computational problems often exceeds the capabilities of a single computer processor. The concurrent use of multiple processors on a single problem, parallel processing in today's vernacular, can often overcome the shortfall in performance of the single processor. The programming task of coordinating multiple processors to solve a computational problem is significantly more difficult then programming for a single processor. It is the parallel programming issue that we discuss here.

The complete endeavor of using a computer, parallel or sequential, to achieve a solution to a problem encompasses several steps. The major steps are;

- conceptual statement of the "physics,"
- formalized mathematical statement,
- algorithm(s),
- computer program (i.e., code),
- mapping code to computer architecture,
- tuning for better performance,
- debugging to achieve correctness,
- execution for solution.

In the above list we have placed physics in quotes since the problem need not be restricted to classic physics. As an example, the problem might be sorting a database. All but the first two items are affected when parallel computation is being sought. From the user's perspective all these steps are important, however it is beyond the scope of this paper to cover each of them. Our focus is on the computer program itself and how it specifies concurrency in order to involve multiple processors.

The machines of interest in HPC (high-performance computing) are all parallel processors. Parallel processors come in many flavors. The machine could be a parallel vector processor (PVP), symmetric multiprocessor (SMP), scalable parallel processor (SPP), massively parallel processor (MPP), cluster of SMPs, or a network of workstations (NOW). Each of these "architectures" has distinct features, some of which are advantageous for the programmer and the applications, and other features which work against the application's performance. To some extent the content of this paper is generic to, and spans, all of these diverse architecture types.

The overall effort of developing, tuning, and debugging a computer code is a human-intensive and time-consuming task. For most organizations, human time is more expensive (i.e., valuable) then machine time. This facet must enter into the considerations of how long it takes to complete all the items in the above list. This is also true in the parallel programming phase which is the topic of this paper.

Despite the parallel computing literature's propensity to cite speedups, parallel efficiency, CPU time, isoefficiency, and a litany of other measures, there is only one thing that matters; *less elapsed wall-clock time.* This is measured from the start of the application to the termination of the application. If a parallel program does not achieve this then it is simply not worth doing it in parallel.

Achieving Parallelism

The goal of parallel programming is to take an application code, or equivalently an algorithm, and cause portions of the computation to occur concurrently across a multiple number of processors. This working definition of parallelism applies to any architecture, any computation, and to any number of processors. Ideally we would like all of the computations to be independent so that we completely avoid any serial work. This is an ideal that is unreachable in practice.

There are several programming methodologies for achieving concurrency, independent of the programming language used. In no particular order, the methodologies are:

- explicit message-passing,
- compilation with vendor specific directives or pragmas,
- compilation invoking automatic parallelization,
- explicit parallelization via programming with threads,
- using a explicitly parallel language,
- explicit process-based parallelization.

All these methods can provably achieve parallelism. Which to choose is a nontrivial exercise. The ease of use of the method must be weighed against the amount of parallelism that can be achieved and against the amount of programming effort required. All of this must be done in the context of how often the resulting code will be used. There is little overall gain in spending weeks on parallelizing a code that only needs to be run a few times. Finally, there is the architecture and the application itself that must be considered in making the decision for a particular parallelization method to use. Not all architectures support all of the methods listed above. This might make the decision for you. The application affects the choice of method indirectly through its inherent parallelism. This is often characterized by its so-called "granularity."

Latency and Granularity

Latency is not directly related to parallel programming. Given this, then why a discussion of it in this paper? The reason may not be obvious but both latency and granularity affect the parallel programming method.

Latency is the time delay between the request for a data item and the actual receipt of the data. This is most often considered a hardware issue. Latency is so important to performance that it can not be ignored. Additionally, the actual latency incurred is a function of the programming model used.

Before considering its implication(s) on the programming model it is appropriate to quantify some hardware and software latencies. The approximate latencies for the memory hierarchy of contemporary machines are shown in Table 1. It is important to understand that the latency should be measured, and viewed, relative to the host processor's clock cycle, rather then time. The reasoning behind this is that the effect of latency is generally lost opportunity to perform useful work. Work is measured by the number of lost processor cycles. The clock frequency of the different processors on the market today, and in parallel machines, can vary by a factor of 4 or more. Normalizing latency to the processor cycle time is more meaningful.

Level	Processor clocks
register	1
primary (L1) cache	2–3
secondary (L2) cache	6–20
tertiary (L3) cache	14–25
local memory	20–200
remote shared memory	$\mathcal{O}(10^2)$
message-passing	$\mathcal{O}(10^3)$–$\mathcal{O}(10^4)$
secondary storage (disk)	$\mathcal{O}(10^4)$–$\mathcal{O}(10^7)$

Table 1: Approximate latencies for contemporary machines. Note that the latencies are given in terms of processor clocks rather than time.

If we consider the latency of local memory compared to message-passing we can see the effect of the choice of programming model on performance. In the case of message-passing the size of the parallel task's computation must be larger then the "equivalent" in shared-memory by a factor of 100 to 1000 to achieve comparable efficiency.

Even when working on the so-called NUMA (nonuniform memory access) machines the difference in latency between local memory accesses and remote memory accesses is large enough that it should be considered in coding for optimal performance. The choice of architecture and programming method will have an effect on the resulting performance.

The latency of the architecture and the latency in the programming method used to invoke parallelism must be considered in the context of the inherent, or at least achievable, parallelism in the application. It is counterproductive to attempt to use a high latency programming approach for an application that only offers small pieces of parallel work.

Granularity refers to the amount of computation that takes place between parallel overhead events. The terms fine-grained and coarse-grained are often used in talking about parallel algorithms and codes. We consider parallel overhead events to be communications and synchronizations between the parallel tasks and also the management of the tasks themselves (i.e., starting and stopping them). There is no precise definition for what is fine-grained vs. coarse-grained. The rule of thumb is that if the grain-size is a few tens of machine instructions or on the same order as the smallest parallel overhead it would be considered fine-grained. Conversely an algorithm that can do several thousands or millions of instructions between parallel overhead or has a grain-size that is some "large" multiple of the parallel overhead event time is considered coarse-grained. Note that this definition allows for an application to be fine-grained on one machine but coarse-grained on another machine.

In an absolute sense we *always* seek to develop algorithms that are coarse-grained. This simply better amortizes the overhead of the parallelism. Even an architecture that is capable of handling fine-grained code efficiently would perform better if it were given a coarse-grained application. Having just stated this we now temper it slightly by noting that coarse-grained applications can expose a negative aspect of parallel processing. Load-balancing can severely affect the overall performance of parallel applications. If the size of the coarse-grained threads are too disparate then it could result that all processors are awaiting a single long-running thread to complete. Naturally this wastes resources while the $(n-1)$ processors are idle. A "pool" of fine-grained threads does not have this type of degradation, at least not to the extent that coarse-grained applications do.

The actual development of coarse-grained algorithms is beyond the scope of this paper. There exists a large volume of published literature on this topic. A few suggested books for understanding parallel algorithms are [1, 3, 10, 11, 13, 14]. There are many others, not to even mention the vast number of published journal articles and conference proceedings.

Parallel Programming

As we had stated earlier the goal of parallel computers is to either do an existing problem with lower time-to-solution, or to permit larger and more complex problems than can be achieved on a sequential computer. In either case it is necessary to achieve concurrency in the volume of computations that make up the task. In the common vernacular, and in the remainder of this paper, we will refer to a stream of execution as a thread. This is an overloaded term but we take it to mean some set of computations that is proceeding on a processor at the same time that other threads are executing on other processors. There is no requirement that the threads be executing the same instructions, operating on the same data, or take the same amount of time to complete their work. The only overt requirement in the context of this paper is that some of these threads are executing concurrently.

We will see in a later section that there is a programming methodology called thread-based programming. It will be clear from the context which meaning of the word "thread" we are using.

Achieving the concurrency in the task's execution requires that the work, or data, of the task be divided among the participating threads. The approach taken in almost all cases of parallel programming can loosely be stated as, "divide, conquer and communicate."

The "division" is either done on the data that makes up the application or on the tasks within the application. While the data itself is not visible to the compiler, it is implicitly present in the form of the language constructs that declare, allocate, and operate on the data.

The "conquering" portion is the assignment of the threads to the processors of the parallel system in order that they can concurrently operate to reduce the time to solution. The sometimes difficult issue of load balancing is part of the conquer phase. If the threads are doing different amounts of work then at some point many threads will need to wait for the longest running thread to complete. Load-balance can have a significant effect on the time-to-solution.

The "communicate" phase involves two major parts. First, is just what the name implies, the communication that takes place between threads in order to exchange or update data that each either has, or needs. Additionally, the threads need to synchronize their execution. This often occurs in the form of reduction to single threaded execution or the rendezvous of all threads before any, or all, can proceed.

In the following sections we make more explicit the differences in the various means by which parallelism can be achieved.

Automatic parallelization

Automatic parallelism is achieved by no work on the part of the user other than compilation with a particular compiler flag (i.e., option) enabled. Users deservedly want this approach to parallelism to be the best choice. Obviously, it is only the best choice if it offers acceptable parallel performance.

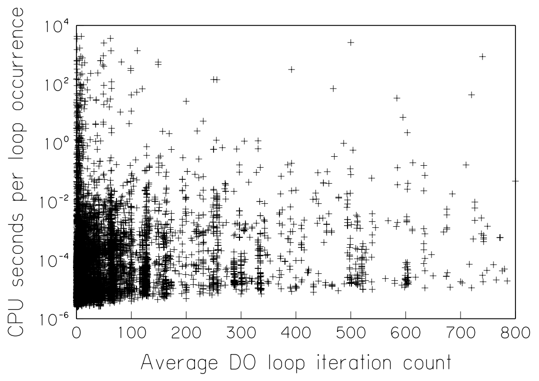
Compiler produced automatic parallelization is generally confined to the loops and loop-nests within the application. Often this has insufficiently coarse granularity to be profitable across more than a few processors. Automatic parallelization was better suited to PVP machines since they offer lower parallel overhead and better synchronization facilities. With this advantage automatic parallelization at the loop level was often profitable. Owing to market and economic pressures RISC-based systems are replacing the PVPs. With RISC-based systems comes larger parallel overhead thereby increasing the difficulty of achieving profitable automatic parallelization based solely on loops and loop-nests.

In order for a compiler to generate parallel code it must emit the requisite instructions to spawn the multiple, concurrent threads of execution and it must also produce code to join the threads at the end of the parallel work. In virtually any real application this will be repeated many times during the course of the application's execution. Compilers have no trouble with this aspect. What is difficult is that in order to get correct results the compiler must determine *all* the dependencies among the data in the various threads that it creates and generate correct parallel code. The dependency analysis is the most difficult part.

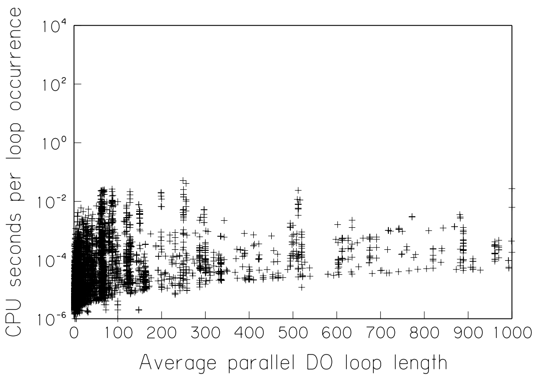
Today, generally speaking, the available technology permits loops and loop-nests to be analyzed and parallelized quite well. This does not necessarily translate directly into improved performance, as we will see momentarily. The ability to perform the requisite dependency analysis across procedures is available but it is not nearly as developed or robust as that for loops. Automatic parallelism across or involving procedures offers the possibility of much coarser grained parallelism than just loops and loop nests. Even given the inter-procedural dependency analysis the code transformations required to translate this into correct parallel code are not well developed. This is an area that is under study by a number of groups. Industrial strength inter-procedural optimizing compilers that can, with high probability and minimal user interaction, deliver good parallel performance are simply not available at this time.

With only loops and loop-nests as the engines of parallelism we should explore what opportunity they present for parallelism. The answer to this is not all that encouraging. In Figure 1(a) we show the CPU time used by each of 21,000 loops contained in 127 real applications [2]. In Figure 1(b) we show the *same* CPU times[1] but only for those loops that are automatically parallelizable by a competent, contemporary automatically

[1]What we are saying is that the times in Figure 1(b) are the sequential times, not parallel times.



(a)



(b)

Figure 1: (a) The processor time used in all 21,000 of the loops in 127 real application vs. the loop span. (b) The same data for only those loops that can be automatically parallelized by a contemporary production parallelizing compiler.

parallelizing Fortran compiler. The key fact revealed in these plots is that the CPU time for automatically parallelizable loops is, in an absolute sense, quite low. This is especially true when it is understood that the CPU time shown is for all iterations of the loop. When a loop is done in parallel, the range of the induction variable is divided among the participating processors. Naturally, this further reduces the CPU time per processor. Given that the overhead to "go parallel" is non-zero it becomes difficult to expect automatic parallelism of loops to provide a big win for the end-user.

It is this author's opinion that users should not expect too much parallel performance from automatic parallelization. If the number of processors is restricted to 2–4 and the memory is shared then in some cases this can be of benefit. In a general setting however the granularity of loops, especially those for which a compiler can perform dependency analysis (i.e., without procedure calls) is too small to achieve any reasonable parallel efficiency. Note that any loop containing a procedure call is generally not a candidate for automatic parallelization. This would require interprocedural dependency analysis.

Shared-memory directives

Each vendor of a shared-memory, or global-shared-memory (GSM), parallel system offers some vendor specific directives (in Fortran) and pragmas (in C) to explicitly control the compilation process. These directives can be used to give relatively simple directions to the compiler, or they can perform rather complex tasks such as parallelization.

As a specific example, consider that every vendor offers a directive to force a loop to be parallelized. When using such a directive the onus to insure correctness is on the user. The directive is, in almost all cases, a specific command that tells the compiler, "I *know* what I want, you just produce it." The compiler will then do exactly that.

The user must be aware of any data dependencies or required synchronizations within the scope of the loop. In practice this directive often shows up on loops that contain procedure calls. Since the compiler can not generally do full dependency analysis across all the invoked routines it will not be able to automatically parallelize the loop, hence the explicit use of a directive.

Using directives is a mechanism to produce rather sophisticated parallel code. However, there is a disadvantage. At this time there is no standard, ad-hoc or sanctioned, for shared-memory, parallel directives. Each vendor offers their own set of directives. There is also no standard for the syntax of the directives or pragmas. Admittedly there is a large degree of commonality and the mapping between vendor A's and vendor B's directives is not too difficult.

As a specific example of pragma induced parallelism consider:

```
#pragma _CNX loop_parallel( ivar=i )
   for ( i=0 ; i<n ; i++ )
      a[i] = b[i] * c[i];
```

This pragma does significant work for the user despite its simple appearance. The code that the compiler generates will spawn the requisite number of threads, it will determine which subset of the loop iterations get handled by each thread, and it will place a barrier after the loop to insure that only a single thread continues. This preserves the sequential semantics of the code following the parallel loop, or more generally, any directive defined parallel construct.

There is a significant fraction of users that use directives to accomplish their parallel needs. From the end-user perspective directives are quite a productive tool since the burden placed on the user's time is quite small and the compiler, which is good at such things, does the tedious work of producing the parallel code. Remember that the correctness of the parallelism when pragmas are used is now the user's responsibility.

Other types of tasks that can be accomplished via directives are to define regions of parallel code, to select the type of scheduling or partitioning of the loop induction variable, defining critical sections, privatizing variables and arrays, and quite a few other tasks related to parallelism.

In contrast to automatic parallelization the use of directives requires some involvement on the users part. Pragmas and directives do offer significant leverage in that they cause a fairly significant code expansion that the user need not be concerned with. Directives tend to bring the tedium of parallel programming more to the level at which the user thinks. Naturally this is a good thing.

Explicit parallelism via threads

A thread is an entity that the operating system manages which is comprised of a set of instructions that are sequentially executed by a processor. In addition the thread has associated with it various resources such as private storage, attributes that may be specified by the programmer, register context, stack, and file descriptor information (file descriptors are owned by the process but shared by the threads). The actions required by the application's threads are achieved through procedure calls to thread library routines. Thread capabilities allow the programmer to exert very specific and fine control over the management of the threads. The most pervasive thread library today is the POSIX threads library. In practice POSIX threads is only an API definition [8]. Each vendor or developer of a POSIX thread implementation builds the library itself. POSIX threads are most commonly referred to as pthreads.

Doing explicit thread programming is definitely a lower-level approach than the use of compiler directives. For the price of more programming and attention to detail threads offers the user more explicit control of the parallelism and the ability to perform tasks that may not be achievable via the compiler directives.

To severely generalize the spectrum of what thread programming can offer you can consider that the tasks to perform are to create a thread, to (possibly) synchronize threads with each other, to define regions of mutual exclusion (i.e., critical section), and to terminate threads. Clearly there is much more to threads then this short list. A very readable account of programming with pthreads is [12].

To create a thread requires a single procedure call. Neglecting the requisite declarations;

```
ret_val = pthread_create( &thread_id ,
                          (pthread_attr_t *) NULL ,
                          thread_procedure ,
                          (void *) NULL );
```

In this example the `pthread_create` procedure does just that; creates a thread which begins execution at the procedure `thread_procedure`. `thread_procedure` is a regular C procedure. Each of the created threads will begin execution at the entry point to this procedure. The different execution paths or data to be operated on is generally controlled via the procedure's arguments. The user is given a thread ID in the first argument so that this particular thread can be identified.

In order to terminate a thread it is generally true that you don't need to do anything. When the threaded procedure returns the thread itself is terminated by an implied call to the `pthread_exit` procedure.

What is relatively common is the need to synchronize threads. The mechanism for doing this is an object called a mutex. Mutex stands for "mutual exclusion." A mutex can be in one of two states; either locked or unlocked. Its mutual exclusion property permits only one thread to have it in a locked state. If any other thread attempts to lock an already locked mutex it will not succeed. Only when the holding thread releases, that is unlocks, its mutex can another thread then lock it.

The following procedures will initialize a mutex, lock the mutex, and unlock the mutex.

```
int pthread_mutex_init( pthread_mutex_t     *mutex ,
                        pthread_mutexattr_t *attribute );

int pthread_mutex_lock( pthread_mutex_t *mutex );

int pthread_mutex_unlock( pthread_mutex_t *mutex );
```

Naturally there is much more to the pthreads API that allows for very sophisticated programming. Handling of signals, scheduling the threads, waiting on conditions, semaphores, cancellation of threads, and much more is available.

When doing thread-based programming the user needs to explicitly attend to every detail. The compiler is only producing sequential code and the calls to the thread library are the driver of parallelism. The most difficult aspect of thread-based parallelism is getting it right (no surprise). The point is that threads live in a shared address space. Thus any given thread can step on a variable without the expressed permission of any other thread. Often the shared address space is a distinct advantage. However, it can lead to data inconsistencies and also to race or deadlock conditions.

To illustrate the dark side of thread programming we illustrate a case where the application can get into a dead-lock. Dead-lock occurs when each of two threads are awaiting resources, such as mutexes, that are held by the other thread. This generalizes to either one or many threads. Specifically, consider the case where thread 1 locks mutex `mutxA` and then thread 2 locks `mutxB`. Now, without releasing `mutxA` thread 1 attempts to lock `mutxB` and thread 2 attempts to lock `mutxA`, without releasing `mutxB`. At this point both threads are dead-locked awaiting a resource, the mutexes, that is not releasable by the holder of the resource. While this simple example seems easy to detect it is not the case in a code of many thousands of lines of source with many threads interacting in complex ways.

Message-passing

Message-passing code can be considered (at the time of this paper) as concurrently executing processes that exchange information via “messages.” The mechanism for the message-passing is through message-passing libraries. At this time there are two dominant libraries; MPI (message passing interface) [6] and PVM (parallel virtual machine) [5]. Both of these libraries are available on essentially every parallel computer system to which you can gain access.

Message passing is similar to thread programming. The user explicitly codes all of the parallel actions that are required. In most cases this is the sending and receiving of messages containing the application's data that must be shared among the participating processes. Since the processes are separate they do not share an address space so that anything that needs to be known to any other process must be passed via a message. Each message is coded by the programmer.

Consider the following (nonsensical) fragment of code:

```
MPI_Init( &argc , &argv );
MPI_Comm_size( MPI_COMM_WORLD , &num_procs );
MPI_Comm_rank( MPI_COMM_WORLD , &my_id );

if ( my_id == root_node ) {
   for ( tag=1 ; tag<num_procs ; tag++ )
      ret = MPI_Recv( &a[chunk*tag] , chunk , MPI_DOUBLE ,
                      MPI_ANY_SOURCE , tag , MPI_COMM_WORLD , &stat );
   }

if ( my_id != root_node )
   ret = MPI_Send( &a[chunk*my_id] , chunk , MPI_DOUBLE ,
                   root_node , my_id , MPI_COMM_WORLD );

MPI_Finalize();
```

In the case of MPI the processes to be involved in the parallel computation are created one-time only at the beginning of the parent process' execution. With our goal for parallel processing—reduced time to solution—this overhead should be taken into account. Message-passing has relatively high latency for communication. In order to achieve efficient parallel execution with message-passing requires that there be coarse granularity. This is either inherent in the application or algorithm or the programmer must get clever to avoid excessive message traffic. A positive side-effect of message-passing is that it forces you to "do the right thing" regarding the decomposition of the applications data or functionality.

Explicitly Parallel Language

There are a great many explicitly parallel languages. Perhaps the most often mentioned one is High Performance Fortran (aka HPF) [7, 10]. These explicitly parallel languages have parallelism inherent in the semantics of the language. This includes the management of the parallel threads of execution and the distribution of data in the systems processor and memory topology.

To make this more explicit consider the following fragment of HPF code.

```
!HPF$ PROCESSORS procs(128)
      real*8 my_data(4096)
      integer pntrs(4096)
!HPF$ DISTRIBUTE my_data(BLOCK) onto procs
!HPF$ DISTRIBUTE pntrs(CYCLIC)
```

The HPF constructs begin with the "`!HPF$`" directive. The `PROCESSORS` directive specifies a topology of processors. This topology is abstract in that it need not match the architectural topology of the host system. (However, an HPF compiler is only *required* to accept processor arrangements that specify the number of physical processors available or 1 processor.) In this specific example we "define" a linear set of processors. The `DISTRIBUTE` directive is an instruction to partition an array onto the abstract topology of processors. Note that in the `DISTRIBUTE` directives the arrays are followed by an attribute specifying how to distribute the data. In the case of a `BLOCK` distribution the array is partitioned among the processors in blocks of size N/P where N is the size of the array and P is the number of processors. The `CYCLIC` directive causes the partitioning to be done cyclically across all the processors, every P-th element of the array will be mapped to the same processor

It is generally accepted, at least in this author's experience, that HPF is most appropriate for codes that are being written from scratch. Attempting to retrofit existing Fortran-77 codes to HPF is a difficult and time-consuming task. HPF also offers no support for irregular problems or task management. These issues are being addressed in the HPF-2 forum that is currently working.

With the limited space we have available we can not possibly do justice to this topic. We mention that (in this author's experience) the next most often mentioned explicit parallel language is Split-C [4].

Process Parallelism

There are a few notable and highly used codes that achieve their parallelism by way of Unix processes. The parallel threads in this case are Unix processes that are `fork`-ed or `exec`-ed by the sequential parent process. Communication is accomplished by one of two methods. The "shared-memory" mechanism that Unix provides allows separate Unix processes to share a common piece of virtual address space. It is important to note that this "shared-memory" is not the architectural shared-memory that we more

Methodology	Pro	Con
Automatic	• very easy to do • avoids subtle bug introduction	• limited applicability • low performance potential
Pragmas	• easy to use • adequate coverage • ignored comments to other systems	• not standard • shared-memory only • not all possible required parallel tasks covered
Threads	• low overhead • complete functionality • standardized	• amount of programming • shared-memory only • care to avoid nondeterminism
Message-passing	• most standard method • well developed • "requires" a reasonably correct decomposition	• high overhead • (possibly) lots of programming
Parallel language	• ease of use • expressiveness	• longevity of language • multi-platform support • efficiency of code
Process parallelism	• only uses standard Unix • can be heterogeneous	• huge overhead • some system level programming

Table 2: A brief summary of the points for and against the common parallel programming methodologies.

commonly consider. It is often referred to as System-V shared memory or `mmap`. The individual Unix processes can also communicate with each other via the standard Unix socket mechanism. The key points here is that heavy-weight Unix processes are involved and Unix process-based communication mechanisms are used. In each case the latency is rather large so the parallelism that is exploited by this methodology must be very coarse-grained. If this is not the case then the parallel performance will be quite poor.

Epilogue

There are a few safe, that is to say non-controversial, statements that we can make to attempt a summary of this paper. For those machines that have shared-memory message-passing is often used within the shared-memory domain. Automatic parallelism and directive/pragma assisted parallelism seem to be used slightly more often on this type of architecture. Thread programming is, like C++ and Fortran-90, increasing its presence in high-performance programming.

For the distributed-memory machines message-passing is (obviously) the programming method of choice. While there is no true standard it is clear that both PVM and MPI are the most widely used. It is this author's expectation that MPI will become the more prevalent library in the future.

With full awareness of the possible misinterpretation of a succinct table summarizing the major points made earlier in the paper, we do exactly that in Table . Obviously brevity is required in the table but the significant points regarding the parallel methods are listed.

Message-passing is a method that, in spite of its detractors, will be heavily used for some time to come. It does offer the highest degree of portability of any parallel programming method available today. Almost any machine that is available today supports both the MPI and PVM libraries. The choice between the two is largely dictated by the needs of the application or the machine environment that will be utilized for the calculation. This benefit comes at a steep price however, this being the high communication latency and (relatively) low communication bandwidth.

Sage advice to anyone beginning a software project is to think through the consequences, constraints, benefits, and long-term side effects of the choice of parallel programming method. The programming language itself is less of an issue and more a matter of picking the right language for the particular task and also one of personal taste. The parallel programming method has larger consequences so think of the long-term ramifications and choose wisely.

Acknowledgments

I am glad to have the opportunity to thank my colleague, Joel Williamson. My many discussions with him over the years on programming and models for parallel programming have been insightful and illuminating.

References

[1] G.S. Almasi and A. Gottlieb, *Highly Parallel Computing*, Benjamin/Cummings, Redwood City, 1994.

[2] G. Astfalk, Hewlett-Packard Company, Richardson TX, unpublished work, 1996.

[3] D.P. Bertsekas and J.N. Tsitsiklis, *Parallel and Distributed Computing*, Prentice Hall, Englewood Cliffs, 1989.

[4] D.E. Culler, A. Dusseau, S.C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick, *Parallel programming in Split-C*, Proceedings of Supercomputing '93, Portland, OR, November 15–19, 1993, pp: 262–273.

[5] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM: Parallel Virtual Machine—A Users' Guide and Tutorial for Networked Parallel Computing*, MIT Press, 1994.

[6] W. Gropp, W. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, MIT Press, 1994.

[7] High Performance Fortran Forum, *High Performance Fortran language specification, version 1.0*, Technical Report CRPC-TR92225, Center for Research on Parallel Computation, Rice University, Houston, TX, 1993.

[8] *IEEE Standard for Threads Interface to POSIX*, IEEE Draft Standard P1003.1c/D10, IEEE, 445 Hoes Lane, PO Box 1331, Piscataway, NJ 08855-1331.

[9] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, and M. Zosel, *The High Performance Fortran Handbook*, MIT Press, 1994.

[10] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing*, Benjamin/Cummings, Redwood City, 1994.

[11] S. Lakshmivarahan and S.K. Dhall, *Analysis and Design of Parallel Algorithms*, McGraw-Hill, New York, 1990.

[12] S.J. Norton and M.D. DiPasquale, *Thread Time: The Multithreaded Programming Guide*, PTR Prentice Hall, Englewood Cliffs, to appear, 1996.

[13] J.M. Ortega *Introduction to Parallel and Vector Solution of Linear Systems*, Plenum Press, New York, 1988.

[14] E.F. Van de Velde, *Concurrent Scientific Computing*, Springer-Verlag, New York, 1994.