Massively Parallel Computing: A Sandia Perspective

David E. Womble, Sudip S. Dosanjh, Bruce Hendrickson
Michael A. Heroux, Steve J. Plimpton, James L. Tomkins

Sandia National Laboratories, Albuquerque, NM 87185–1110 [1]

David S. Greenberg [2]

IDA/CCS, 17100 Science Drive, Bowie, MD 20715-4300

Abstract

The computing power available to scientists and engineers has increased dramatically in the past decade, due in part to progress in making massively parallel computing practical and available. The expectation for these machines has been great. The reality is that progress has been slower than expected. Nevertheless, massively parallel computing is beginning to realize its potential for enabling significant breakthroughs in science and engineering. This paper provides a perspective on the state of the field, colored by the authors' experiences using large scale parallel machines at Sandia National Laboratories. We address trends in hardware, system software and algorithms, and we also offer our view of the forces shaping the parallel computing industry.

---

[1] Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000.
[2] Contributions to this paper represent work done at Sandia National Laboratories.

1 Introduction

Nobody seems to agree on when parallel computing started, but we can agree that it has been around for a long time. Certainly, many of the concepts go back to the 19th century. However, from a practical standpoint, and for the purposes of this paper, we consider the beginning of parallel computing to be sometime around the middle 1980s. It was at this time that parallel computers first began to be programmed as true parallel machines and to compete with the established supercomputers, like the Cray vector machines.

There are several reasons why parallel computing became practical during the mid-1980s. First and foremost, were the hardware advances. Miniaturization of the electronics and the increasing power of the single–chip microprocessor allowed processors with sufficient computational power to be packaged together. Also, the ability to communicate between processors improved to match the computational power of these one-chip processors.

Two competing hardware approaches battled for dominance in the late '80s: single instruction, multiple data (SIMD) and multiple instruction, multiple data (MIMD). In SIMD machines, every processor executes the same instruction at each clock cycle but on different data. In MIMD machines, the processors operate independently, and synchronization is left to the user. In the evolutionary struggle between hardware paradigms, SIMD has fallen by the wayside. Although SIMD machines can be cost effective for some applications, they have proven to be less flexible and less general purpose than MIMD machines.

Hardware can also be classified as either shared memory or distributed memory. In a shared memory machine, any processor can access directly any part of memory. In a distributed memory machine, memory is assigned to a processor or node and data is shared through interprocessor communication. In recent years, the concepts of shared memory and distributed memory have essentially been merged. Individual computational nodes have shared memory, while "massive" parallelism is usually achieved by replicating these nodes and using distributed memory.

This classification of machines is also made somewhat difficult by the fact that some vendors have tried to provide a single memory image on a distributed memory machine. The goal is to remove the burden of message passing parallelism from the programmer by making the machine look more like a large workstation. This has been successful for small numbers of processors but less so as the number of processors increases. The failure has been due less to the hardware than to the fact that compilers are unable to recognize and exploit data locality. The Cray T3E, for example, has a very light-weight communication mode, but explicit message passing remains the most effective way to use more than a few processors. Shared memory programming on distributed memory machines may not be the holy grail, but it continues to provide a quest for many researchers.

It may be worthwhile here to note that shared memory parallelism involves implicit communication and explicit synchronization, while distributed memory parallelism involves explicit communication and implicit synchronization. The right approach depends on the hardware available and the amount of parallelism desired.

Table 1 shows a range of machines classified by instruction architecture and by memory type. This table is, of course, incomplete; it is meant to be illustrative of the types of machines in each category rather than exhaustive; however, we think we have listed the machines that can be considered commercial successes. The CRAY 1 and Cyber 205 shown in the table are vector machines, but this is a form of fine-grained parallelism and illustrates the shared-memory, SIMD approach. Also, the Origin 2000 implements the shared memory through a non-uniform memory architecture (NUMA).

Table 1
This table shows a classification of several machines as either SIMD or MIMD and as either shared memory or distributed memory.

	Shared memory	Distributed memory
SIMD	Cray 1	Connection Machine CM-2, CM-5
	CDC Cyber 205	MasPar
	Cray XMP, YMP	Intel iPSC-2, iPSC-860, Paragon, Tflops
MIMD	SGI Origin 2000	Cray T3D, T3E
		IBM SP1, SP2
		nCUBE 10, nCUBE 2

Hardware is only one member of a triad required for effective parallel computing. The other members are system software and algorithms. Research and development in these areas quickly followed the hardware improvement. Even though some “full-feature” operating systems were adapted for parallel computing, many of the successful machines used a more custom approach. “Stripped-down” operating systems that focused on speed, low memory requirements and fast communications (especially message latency) provided the best results. Compilers and tools have followed somewhat more slowly. Although automatically parallelizing compilers remain a very active area of research, most parallel machines have compilers that are based on directives or message passing library extensions. Similarly, most parallel debuggers have little more capability than low-end, single-processor workstation debuggers.

The third member of the triad is algorithms. The history of parallel numerical algorithms stretches back for many years and includes Jacobi iteration, von Neumann's weather forecasting, computing for the Manhattan project, asynchronous iterations, and many more. These algorithms focus on the applications themselves. Recent algorithmic developments have generally exploited the realization that the best serial algorithms are often the best parallel algorithms. In other words, high parallel efficiency usually cannot compensate for low algorithmic efficiency. A complementary effort has been devoted to enabling algorithms, those that enable or support the use of a parallel machine. These include load balancing and scheduling methodologies as well as collective communication algorithms.

Finally, it is worth mentioning data structures explicitly, although they are inseparably linked with algorithms. Effective use of parallel computers depends on an effective data layout. That is, data must be stored so that it is as "close" as possible to the processor that will be using it in computations. Algorithm design and data layout are often done with the idea of minimizing communications, although an alternate goal is to minimize transfers between levels in the memory hierarchy that now includes local main memory and distributed main memory. To date, the most effective parallelism has been done by hand. Also, until now, memory layout and data structures have been mostly static. However, this is changing, and the need for dynamic memory use is increasing. This trend will increase the need for tools for algorithm design, data layout, memory management and communication paradigms.

Most of the authors of this paper have spent a significant portion of their careers at Sandia National Laboratories where the focus has been on the use of distributed memory parallel computing for large-scale simulations in science and engineering. Because of this research focus, this paper will focus on the practical aspects of parallel computing and its use in applications. It will avoid historical presentations except where necessary. Section 2 will present an overview of hardware that has made an impact in practical, large-scale simulations. Section 3 will discuss operating systems and tools. Section 4 will discuss algorithms and enabling technologies. The focus in these three sections will be what is necessary for the practical use of parallel computers and what will be required for continued progress. Sections 5 and 6 will present an overview of industry and government impact on parallel computing. Finally, section 7 will summarize the paper with an eye toward the future.

2 The Development of MPP Hardware

Throughout the 1980's, supercomputing was dominated by vector processors such as those from Cray Research. By the end of that decade these machines were modestly parallel with as many as eight processors; however, they were mainly used in a mode in which applications ran on only a single processor. It was during this period, though, that the development of large scale parallel machines began. In the middle to late 1980's, several companies developed small scale parallel computer systems with a few tens of processors. Most of these machines had a shared memory architecture; however, there were a couple of large scale parallel computer systems that were developed at this time that would have a major impact on the progress of massively parallel processing systems (MPPs). These were the Connection Machines from Thinking Machines Corporation and the nCUBEs from nCUBE Corporation.

In 1987 Sandia National Laboratories began its transition from vector processing to massively parallel computing with its acquisition of a 1024 processor nCUBE 10 machine (see Table 1). The nCUBE 10 was a hypercube with a distributed memory and Multiple Instruction Multiple Data (MIMD) architecture in which each processor had 512 KBytes of memory and a performance equivalent to about that of a VAX-11/780. The full system was able to achieve about 100 MFlops on a real application. With this machine Sandia researchers John Gustafson, Gary Montry, and Bob Benner were able to demonstrate scaling of greater than a factor of 200 on a fixed size real scientific application and to win the Karp Challenge and the first Gordon Bell Award [10]. In fact, fixed–size speedups over 500 and scaled speedups over 1000 were achieved for a number of applications. Sandia was able to port several of its scientific and engineering application codes to this machine. On many of these application codes the nCUBE 10 was faster than a Cray YMP processor.

The nCUBE 2 machine was introduced in 1990. This machine had a theoretical maximum of 8,192 processors, however, none of that size were ever built. In 1990, Sandia acquired an nCUBE 2 with 1024 processors as a replacement for its nCUBE 10. This machine has 1024 processors each with 4 MBytes of memory and a peak performance of about 2.7 GFlops. Sandia ported a significant fraction of its high-end computing workload to this machine. For most of Sandia's applications codes this machine proved to be more capable than all eight processors of a Cray-YMP. The nCUBE 10 and the nCUBE 2 machines are well balanced architectures, in that their interconnect performance (latency and bandwidth) is sufficiently high that it does not limit the performance of typical applications. Also, these machines proved to be very reliable. Sandia's original nCUBE 2 and a second one acquired later are still in use today, and they have gone several years without a hardware failure.

The first computer system from Thinking Machines to have general scientific applicability was the CM2. This machine had up to 65,536 one bit processors in which 32 processors shared a Weitek floating point processor. In effect, a fully configured CM2 had 2,048 parallel floating point processors. The CM2 had a single instruction multiple data (SIMD) architecture with a hypercube interconnect. The CM2 used a front-end machine to provide an instruction stream and to process any serial code in the application. Each processor executed the same instruction in lock step with all other processors assigned to a particular job. The CM2 had a good balance between its interconnect and processor performance and it became a successful and widely-used MPP system. Sandia researchers found this machine to be a good match for a limited set of applications; however, they found the flexibility of the MIMD architecture, like that of the nCUBE machines led to wider applicability for scientific and engineering applications.

Thinking Machines Corporation's third generation computer system, the CM5, replaced the CM2 in about 1992. The architecture of this machine was such

Table 2
Characteristics of Some Significant MPP Computer System

Computer System	Year of first Installation	Architecture	Interconnect	Scalability and Balance
nCUBE 10	1987	distributed memory, MIMD	hypercube	good
nCUBE 2	1990	distributed memory, MIMD	hypercube	good
CM2	1989	distributed memory, SIMD	hypercube	good
CM5	1992	distributed memory, SIMD/MIMD	fat tree	fair
Paragon	1993	distributed memory, MIMD	2D-mesh	excellent
Tflops	1997	distributed memory, MIMD	2D-mesh	excellent
T3D	1994	distributed memory, MIMD	3-D torus	excellent
T3E	1997	distributed memory, MIMD	3-D torus	excellent

that it could be used in either the SIMD mode or MIMD mode. This machine used four vector processors together with a Sun Microsystems SPARC processor for each node. This machine was not as well balanced as the CM2, and it was not nearly as successful in the marketplace.

During the late 1980's and into the 1990's, Intel Corporation became a major player in MPP computing. Early Intel machines (IPSC 1, IPSC 2, and IPSC860) were built with a few tens to a hundred or so processors and had hypercube interconnects with distributed memory MIMD architectures. However, with the Delta machine and then the Paragon, Intel moved into large scale MPPs. The one-of-a-kind Delta and subsequent production Paragon systems are distributed memory MIMD architectures with 2D-mesh interconnects. Paragons were built with either two processor or three processor nodes.

In 1993 Intel delivered to Sandia a Paragon computer system with almost 3800 processors. Sandia's large Paragon had the two processor nodes. Sandia and Intel researchers were able to achieve 143 GFlops on the MP-LINPACK benchmark with this machine. This machine became Sandia's workhorse computer system for high-end computing a year or so after its installation at Sandia.

As part of a wider shakeout in the industry, Intel announced in 1996 that it had decided to get out of the supercomputing business. However, before this announcement Intel had signed a contract with Sandia as part of the Department of Energy (DOE) Accelerated Strategic Computing Initiative to build the world's first tera-scale computer system. As a result the Intel Tflops computer system that Intel delivered to Sandia in stages during the first half of 1997 is a one-of-a-kind machine.

The Tflops machine has more than 9,300 processors. The system is very well balanced between processor speed and interconnect performance. It has a distributed memory MIMD architecture with a very fast two layer mesh interconnect. The machine has several unique features which have proven to be very valuable. These include a completely separate communications network for system monitoring and management. It also has a high level of redundancy in the hardware. The machine is also unique in that it is split into three partitions with a classified end, an unclassified end, and a center section that can be switched between ends in approximately twenty minutes. While the split configuration is the usual configuration, the machine can easily be configured as a single system for either classified or unclassified computing on a single application.

The Tflops machine achieved a major milestone on December 4, 1996 when it became the first general purpose computer to achieve a sustained trillion floating point operations per second on the MP-LINPACK benchmark. (In June of 1997 the full machine achieved 1.338 Tflops on the MP-LINPACK benchmark.) Tflops has proven itself to be a phenomenal success. It exhibits unprecedented reliability; availability exceeds 90%, and over 80% of its theoretically available cycles are used by real applications. Researchers have been able to scale their application codes to the full machine capacity, and there is a continuous backlog of large parallel applications waiting to run on the machine. The Intel Tflops machine has recently been upgraded to a peak speed exceeding 3 Tflops with memory of 1.2 Tbytes.

In the early 1990's Cray Research moved into the MPP market with its T3D computer system. This machine used a three dimensional torus interconnect with a distributed memory SPMD (single program, multiple data) architecture. The machine is very well balanced between processor performance and interconnect performance and has proven to be a very scalable architecture. Cray's second generation of this architecture, the T3E, has proven to be very successful. With the exception of the Intel Tflops machine the Cray T3E is the only current high-end computer architecture to demonstrate scaling to very large numbers of processors on a wide variety of scientific and engineering applications.

Architectures for high-end systems are continuing to evolve rapidly. Intel has left the business as have Thinking Machines and nCUBE. Cray Research was acquired by Silicon Graphics Incorporated and has decided that there will be no follow-on to the T3E architecture. International Business Machines is, in its new products, building a hybrid distributed memory system with symmetric multiprocessor (SMP) nodes. Even though the distributed memory architecture has proven to provide scalability to nearly 10,000 processors (Tflops) the industry is moving towards an unproven architecture of a cluster of large SMPs with weak interconnects. We find this trend to be worrisome and fear that it will prove to be a major setback for high-end computing. To date, success in massively parallel computing has been achieved with computer architectures that are well balanced between processor performance and interconnect performance, so the slow connections between SMPs is likely to be problematic. Also, clusters of SMPs require more complex programming models (e.g., both threads and message passing) than pure distributed memory machines.

We are significantly more optimistic about another trend in high end architectures - the prevalence of build-your-own parallel machines comprised entirely of commodity components. Throughout the '90s there has been a trend towards building parallel machines out of commodity pieces, but each of the major vendors retains some proprietary components. But in the past few years there has been a groundswell of interest in constructing machines entirely out of off-the-shelf hardware and open source software. Several machines have been constructed this way with multiple hundreds of processors which exhibit outstanding price/performance. We are optimistic that the remaining challenges can be addressed to allow these systems to scale to thousands of processors.

3 System software

System software includes operating systems (OS's), compilers, debuggers and more. It is absolutely necessary, but in the case of high-performance computing, it has not met the basic expectations of many of its users. It has rarely provided a convenient user interface, portability, or highly efficient use of the underlying hardware. Highest performance often required reorganizing the structure of a code, adding system specific subroutine calls or directives, *and* weeks of tuning. For example, the CM-5 vector units, the T3D's block transfer engine, and the second and third processors of Paragon-class machines can only be accessed via unportable, mostly undocumented techniques.

But, users need not despair. Recent trends in hardware architecture such as the move toward increased (sometimes exclusive) use of commodity hardware and peripherals, the shift from 32-bit to 64-bit addressing, and the increased availability of multi-processor shared-memory systems each provide an opportunity for improved system software. It may be possible for high-performance modifications to be leveraged on the explosive development of open OS's such as Linux or to be piggy-backed on the necessary OS re-implementations of memory and file-system interfaces to take advantage of the larger address spaces and SMP architectures.

1999 and 2000 will be key years for the development of the system software. Ways must be found to add parallelism so that both shared-memory and distributed memory performance is improved. The remainder of this section reviews past system software approaches and recommends some new directions.

3.1 Partitioned system design

In order to take advantage of these architectural transitions it is desirable to amplify a trend in system software toward partitioning of services.

Traditionally, system software for large-scale systems has been built under one of two paradigms: full-custom or minor modifications to desktop systems. Neither approach has been completely successful. Full-custom systems can provide high performance and direct access to specialized new hardware features. However, they are expensive to produce, maintain and administer. More importantly, they are rarely portable to new systems. Conversely, desktop systems are designed to run on commodity processors and peripherals and inherit the mass-market cost structure. Yet the desktop systems rarely allow efficient access to new hardware features and often include overhead and restrictions which are necessary in their native environment but detrimental in high-end systems.

Both of these already flawed approaches are becoming even more difficult to apply. The full-custom OS's must support increasingly complex hardware architectures – a system which only supports one type of disk, one type of tape, and its own internal network cards is now considered unusable. On the other hand the high-end system requirement to run without rebooting for months at a time while supporting hundreds of users will not be met by desktop system software designed for single-user systems which are turned off at 5:00 pm each day. Increasingly, desktop systems concentrate on the need to protect inexperienced users and choose implementations which prevent experienced users from efficiently exploiting the hardware.

The partitioned system approach attempts to marry the best of custom and commodity systems. Partitioning makes the distributed nature of future architectures a virtue rather than a challenge. System software can be naturally partitioned along hardware boundaries; part of the machine can run a simple, custom kernel while another part can run a modified, full-functionality workstation OS. The kernel provides only the most basic functionality needed for highest performance applications while the workstation OS provides the user interface and support for peripheral devices. The result is a system with better maintainability, expandability, and robustness.

3.2 Puma and the ASCI/red machine

Sandia has successfully applied this partitioned approach to its large Intel systems[9]. We have been able to routinely run jobs using thousands of nodes and hundreds of GBs of memory over several days. On 32 MB Paragon nodes we are able to allocate 32 million byte data arrays. Message passing overhead is sufficiently low that parallel efficiencies exceeding 90% on thousands of processors are not uncommon.

Both the Paragon and Tflop machines use a variant of OSF1/AD Unix as their full-function OS. This OS runs on a service partition (about 15 nodes), and an IO partition (about 50 nodes.) The service partition handles user logins, parallel job launch, batch queues, debugger interfaces, and other user-oriented services. The IO partition supports all disk drives and all high-speed, off-machine networking. A custom kernel runs on the remaining nodes which compose the compute partition.

The custom kernels (SUNMOS on the Paragon and Puma/Cougar on the Tflop) run very close to the hardware – they attempt to provide as much access and as little overhead as possible. The memory footprint is less than 1MB, few if any daemons compete for processor time, cache and TLB use is minimized, and internode communication is an optimized (approximately 10ms latency) native feature. In comparison, standard OS's have footprint well over 8MB, depend on a wide variety of daemons to support normal operation, and provide external communication through slow, general-purpose mechanisms like the TCP/IP stack (over 100ms latency). The custom kernels succeed by being very efficient on the tasks they do (virtual addressing, numeric exceptions, and local processor scheduling (e.g. threads)) and by not providing many functions common to standard OS's (such as console support, http servers, etc.)

The functionality not provided by the custom kernel is spread across the machine in a natural way. Maintenance of physical peripherals such as disk and ATM interfaces is provided locally in the IO partition. Interactions with users, including windowing and job scheduling, are provided locally in the service partition. Functions which connect two types of interaction, e.g. a debugger must interact with the user and with the running application, are split between partitions at their functional boundaries. Thus the debugger's user interface and access to the source code resides on the service partition while its ability to read and modify the memory of running code resides on the compute partition.

Both the Paragon (until its processors became obsolete and it was decommissioned) and the Tflop machine run 24 hours/day, 7 days/week. Faulty hardware is replaced through hot-swapping of boards and on-the-fly reconfiguration of the system hardware. The partitioned software allows for robust, maintainable service which provides high efficiency to Sandia's aggressive code teams.

3.3 Partitioning to improve service

We further illustrate the use and advantages of partitioning in the next several subsections.

3.3.1 Memory Management

The use of virtual memory (VM) and demand paging is one of the hallmarks of today's computer systems. Unfortunately, although the VM/demand-paging combination is highly successful for monolithic systems, the combination can be deadly in the distributed realm. Sandia, and most of the early users of the Paragon, found that when running a single integrated OSF partition the Paragon yielded very poor performance. Since all nodes attempted to page in the executable from the same file on a single disk, the use of additional nodes often led to a slow-down rather than a speed-up. Synchronized applications tended to overwhelm the paging apparatus. Sandia's remedy was to install SUNMOS (at that time a research prototype) which allocated physical memory on each node, and never demand paged.

This solution did limit the applications' ability to create memory structures larger than the physical memory size. This restriction was acceptable because applications had access to huge amounts of physical memory (on the Tflop there is over 512GB available). Problems which ran for days or weeks could easily be fit within the physical memory.

The restriction on demand paging matched the Sandia philosophy of applying big machines to big problems which could not be solved otherwise. Even so, without partitioning, the no-demand-paging solution might not have succeeded. Many user services were designed to work within a multi-user, dynamically paged environment. It would have been difficult to get these programs to abide by the memory restrictions. Fortunately these programs could be kept in the service partition where the full-OS still used demand paging.

The partitioning allowed further memory management tuning. Significant improvements were obtained by modifying its custom kernel to use more efficient, larger virtual pages. However, the use of such large pages would have been impossible if small tasks such as shell commands each had to be allocated a large page. Since small interactive tasks were all segregated to the service and IO partitions the large page approach was a success.

3.3.2 File systems and disks

A second pillar of OS design is the abstraction of disk (or other storage device) details into a file. Unfortunately, as with demand paging, the standard file system model is not appropriate for large distributed systems. The use of a file by the many components of a single, integrated, high-speed application is radically different from the use of a file by many independent users. A common high-performance operation is the coordinated reading or writing of a large data set between many processors and many disjoint portions of file. Interfaces such as MPI-IO[14] and Sandia's PXI/PDS/PIO have been designed to help applications programmers easily express the required coordination. Yet these interfaces are forced to shoe-horn the resulting, naturally-parallel data transfers into an explicitly-serial, often high-overhead file system. Some parallelism can be regained by having the interface program maintain thousands of small files corresponding to each parallel data stream but this makes changes in parallel shape difficult and requires users to commit to only accessing the data through the particular interface used to create the data.

Partitioning provides a possible alternative. Interfaces such as MPI-IO can run on the compute partitions and be provided with a suitable virtual file system (initial development seems likely to occur within Linux' VFS but most vendor OS's also have some form of virtual file system). This local portion of the parallel file system, LPFS, would be responsible for maintaining blocks of files used by the local application component. Advanced techniques in IO traffic analysis could be used to tune the prefetching, caching, etc. Mappings from data blocks to disk locations would be the responsibility of the OS within the IO partition. A range of file system types could be offered to the compute partition.

A very fast file system might optimize the mapping from file and block number to host IO node by using a simple function requiring only a list of the IO nodes. Compute nodes could cache this function once during a collective open call and then perform all subsequent operations directly with the proper IO node. An MPI-IO implementation which queries the local file-system for parallel shape could then coordinate its accesses to maintain as full parallelism as possible.

A well-defined interface between the LPFS and the IO partition OS would allow the IO partition hardware and software to develop independently of the compute partition. Thus, new compute engines would not be forced to await the development of a suitable IO subsystem but would instead plug into existing ones.

There is a great potential for improvement in this area. However, no approach, including the one described above will be successful without the emergence of and adherence to a set of common rules and conventions.

3.3.3 Debuggers and programming environments

System software is not limited to operating system maintenance of hardware. It also includes the growing array of tools to which programmers and users of applications have become accustomed. For a programmer the most basic of these tools are the compiler/runtime/debug suite used to create the programs. The debuggers for high-performance machines lag the furthest behind the tools for serial machines.

Debugger development faces many difficulties, but we focus on just two: the necessity of extending debuggers across a network and the challenges in representing data from hundreds or thousands of threads-of-control.

Traditionally the debugger runs on the same machine as the code being debugged. In fact, the debugger takes control of the process – all activity filters through the debugger. The debugger also has direct access to program source. This model is hard to apply to a distributed application. The debugger can be given control of every process but it is no longer clear which debugger process is in charge. In short the debugger must become a distributed program.

The partition model provides a simple strategy for the debugger. The control portion of the debugger runs within the service partition using standard techniques. (Using the workstation version the debugger in the service partition also keeps it up-to-date and compatible with the latest compilers.) The portion of the debugger which controls code (e.g. sets breakpoints, read memory locations, or writes to memory locations) is distributed throughout the compute partition. These functions are typically highly OS and hardware dependent. Yet they are also relatively simple and of short duration. Thus OS extensions or modules may be unable to implement these functions and make them efficient. The efficient implementation of debug functions on a parallel machine is particularly important since many bugs involve the timing between various nodes and slow debugging can alter the timing. In fact, it is a challenge to write a parallel debugger that does not introduce false timings, change race conditions and create out-of-order execution dilemmas.

The above model of debugger design still suffers from information explosion. The debugger must be able to filter responses to produce responses such as "the common value is x but processor p has value y". Graphical trace trees or other multi-media representations may be necessary. However, no amount of filtering and presentation will change the fact that debugging of 100-way and higher parallel codes is extremely difficult.

We should not spend too much effort trying to make parallel debuggers look just like serial debuggers only bigger. Instead fundamental research will have to be done on ways to allow the computer to assist in debugging. For example, we may not be able to find a race condition on the update of a particular variable by stepping through the code. Instead we may need utilities which first apply strict protection semantics to all variables and then progressively weaken the protection on subsets of variables. When a set which results in a different result is found a search can be made to find out which variable is the culprit.

Space does not allow a discussion of all the issues facing system software. Continued work will be required in the development of parallel job management systems including batch queues. Heterogeneous systems will present many new problems not the least of which will be the maintenance of a reliable, robust, and secure system. New interconnection network hardware will continue to make fast, safe data movement a difficult research topic.

4 Algorithms and Applications: The Software Challenge

4.1 Parallel Programming

The principle objection to parallel computers is that they are difficult to program. There is a significant component of truth in this claim, particularly for large-scale parallel machines. However, it has been our experience that for most scientific calculations, the complexity of this task has been overrated.

There are three reasons why parallel programming is more challenging than serial programming. First, parallel programs must include the mechanics of exchanging data between processors or handling mutual exclusion regions. This adds complexity to both the semantics and the syntax of a program. Second, in an efficient parallel program the work must be evenly divided among processors. This is an algorithmic challenge with no serial counterpart. Third, the data structures must be divided among processors to preserve data locality. This is obviously true for distributed memory machines in which data movement is costly. It is also true for shared memory machines since locality reduces the cost of maintaining cache coherence. This issue has a counterpart in serial programming since data locality is also essential for good cache performance on serial machines. However, the performance implication is much greater in parallel. In general, the task of reducing parallel overhead, managing processor synchronization and balancing the work load makes it difficult to write efficient and scalable parallel code.

To assist the programmer in meeting these challenges, several different parallel programming methodologies have been seriously pursued; each has its shortcomings. The SIMD approach proved to be too inflexible and limiting, so it is now only used in niche applications. HPF, with it's SIMD-like design, has foundered. Automatic parallelization of sequential code can be very useful for relatively simple programs on small numbers of processors, but is currently of little value on larger machines. Language extensions like Split-C [6] or Titanium [18] have generated significant academic interest but minimal usage in the wider community. A key reason for this is the chicken-and-egg problem that programmers are understandably reluctant to use a language which lacks vendor support, and likewise vendors will not support a new language no one is using. By the metrics of vendor support and number of users, the two most successful parallel programming methodologies are explicit message passing as standardized by MPI, and shared memory emulation as instantiated in OpenMP. Of these, only MPI can currently be described as an unqualified success – OpenMP is sufficiently new that the jury is still out.

New architectures combining a large number of SMP nodes present a new set of challenges. Message passing (e.g., MPI) emulation on the SMP node is often absent or inefficient, so that threads-based programming is preferred within a node. At the same time, threads-based programming techniques generally do not scale to large numbers of nodes, so that message passing is preferred between nodes. Mixing programming models can make the already complex task of algorithm design even more difficult.

Although explicit message passing can be tedious and error prone, it has a key advantage over other approaches. It focuses the programmer's attention on the performance-critical issue of the parallel hardware – data locality. Message passing requires a processor to own (or acquire) in its local memory all the data it needs for its computations. An implementation that achieves substantial data locality is rewarded with high performance and scalability on distributed or shared memory platforms. The price for this is in the complexity of algorithmic design and implementation; as yet no compiler or automatic tools can do this adequately.

Thus, the real challenge of programming large parallel machines is not in the expression of the parallelism, but rather in designing and implementing scalable, data-local algorithms. This is particularly true on large-scale parallel machines, since as the number of processors increases, latent unscalability is exposed. Scalability analysis is an important tool in devising parallel algorithms, specifically the quantification of the communication requirements of an algorithm. Unfortunately, many real applications are too complex to be easily modeled in this way. Calculations that involve multiple phases, multiple synchronization points or adaptivity are not easily amenable to analysis. In the end, parallel algorithm development for complicated problems often involves as much practical engineering as it does theoretical science. It is worth noting that the most studied academic model of parallelism, the PRAM, does not even account for communication cost. And even the more realistic models like LogP, BSP, and their many descendants are not widely used by practitioners. Having said that, it is worth noting that, although we do not use any of the formal models explicitly, the parallel programming style which we have found most effective is broadly consistent with the BSP methodology [3]. Specifically, we like the programming discipline imposed by BSP's *superstep* concept in which phases of local computation are interleaved which phases of data exchange.

Sandia has acquired a series of large-scale parallel machines over the past decade that have been conveniently homogeneous in architecture. They have all been true distributed memory machines with a thousand or more relatively low-end processors and fast proprietary communication networks: an nCUBE 1 (1,024 custom procs), nCUBE 2 (1,024 custom procs), Intel Paragon (3,800

[3] More information about BSP can be found at http://www.bsp-worldwide.org/.

i860 procs), and currently the Intel Teraflop machine (9,300 Pentium procs). These machines have required us (a luxury or burden, depending on your outlook) to (1) code all our algorithms and applications in the lowest-common denominator style of message-passing and to (2) pay careful attention to scalability in order to run efficiently on thousands of processors. These requirements have taught us several useful rules-of-thumb about parallel algorithm design.

(1) The parallelism is in the problem, not in the code. Thus it is important that the programmer understand the problem being solved.

(2) To paraphrase the real estate mantra, there are 3 important issues in good parallel algorithm design: locality, locality, locality. This means creating distributed data structures and choosing decompositions (assignments of data to processors) that minimize inter-processor communication. Though these choices can be viewed as a programming burden, they are at the heart of designing an efficient parallel algorithm. The real challenge is devising algorithms and decompositions in which locality is enforced and, simultaneously, the work load is balanced.

Distributed memory machines or programming models like OpenMP which provide the illusion of shared memory (i.e. through a global address space) are convenient for the programmer and can simplify the code development process. This ease of expression can lull the developer into the illusion that data locality is not important, but on large numbers of processors it always is. Even on machines that provide global address spaces, a programmer must eventually confront the same set of algorithmic challenges posed by explicit message passing.

(3) For scientific computing problems, it is generally the case that the fastest parallel algorithm is an implementation of the best serial algorithm. This may change in the future as even more complex applications are tackled, but it has been the case to date. In the early and mid 80s, it was widely assumed that the arrival of parallel computers would lead to a rash of new algorithms motivated by parallelism. Within scientific computing, this has generally not happened. For example, the initial excitement about asynchronous iterative solvers which looked well suited to parallel architectures has largely abated. Instead, the past decade has seen a focus on the efficient parallelization of existing serial algorithms, such as preconditioned conjugate gradient and multilevel algorithms. One reason for this is that large parallel machines allow for the solution of large problem instances. Inefficient serial algorithms become prohibitively expensive, even if they parallelize well. To put it another way, parallel scalability is generally less important than algorithmic scalability.

Fundamentally new algorithms have been required principally where parallelism generates new issues which lack serial counterparts. Specifically, novel algorithms have been required for problem decompositions, load balancing and collective communication operations.

(4) Attention should be paid to load-balance at the very beginning of the

design process. A load-imbalance of a few percent on a few processors will typically amplify to kill scalability on hundreds or thousands of processors.

(5) At a particular stage of a complex application, there is often a decision to be made about which processor will perform what work (choosing a decomposition). Typically this decision involves a trade-off between load-balance and communication cost. That is, to load-balance the work, more communication must be done to get data on the right processor, or vice versa. In accordance with the previous point, we have found load-balance is usually the more important factor in this trade-off. Current machines typically have fast enough communication networks to justify the extra data movement, though this may be less true on the increasingly popular build-your-own Beowulf-class clusters.

In aggregate, these issues pose an initial barrier to developing a new algorithm or porting an existing application to a parallel machine. However, our experience has been that once a good algorithm is devised, its implementation is not much more difficult than serial programming. The reward is high performance and scalability in the final product.

4.2 Parallel Tools and Applications

In the past decade the study of parallel algorithms for scientific computing problems has matured significantly. Ten years ago the best parallelization strategies for most scientific problems were still uncertain. Today, in many domains the parallel algorithms are well understood and the principal efforts are devoted to the development of good tools and libraries to encapsulate the results of the algorithmic research.

This maturation has proceeded at a different pace for different applications. Dense linear algebra calculations and regular grid finite difference codes were among the first to mature. More complex calculations followed later like particle simulations, unstructured grid finite element methods and iterative solvers. A number of highly challenging computations are still in the algorithmic-research stage. Among these are sparse direct methods, radiation transport, good parallel preconditioners and adaptive calculations.

The maturation of the field has taken longer than many predicted for several reasons. One is that most observers underestimated the difficulty of devising good parallel algorithms for even comparatively simple calculations. As an example, the early dense linear algebra codes all used one-dimensional decompositions and it wasn't until the early 90s that the superiority of two dimensional decompositions was widely accepted. A second reason is that, as discussed above, parallel software is inherently more complex than serial software. The Fortran 77 coding style that predominated in the 80s was an impediment to rapid progress. In recent years, the scientific computing community has eagerly adopted more modern software development techniques, particularly object orientation. In the short run this change consumed time as the community learned new languages and paradigms. But it will undoubtedly lead to better tools and libraries in the long run.

While research issues remain in all areas of parallel computing, mature libraries and tools have emerged for many kernel computations. For developers working on new application codes, these tools significantly simplify the writing of parallel software. They also allow developers to work at a higher level of abstraction and avoid low-level coding. For example, SCALAPACK [4] is now a standardized dense linear algebra library for parallel machines, similar to LAPACK in spirit. Parallel sparse matrix libraries such as PETSc [3] and AZTEC [12] provide a rich set of iterative solver and preconditioner options to the user. Partitioning tools such as Chaco [11] and METIS [13] are widely used to create near-optimal decompositions of grids or other computational loads across processors.

In many areas where libraries are difficult to develop, the basic algorithmic issues are well understood. This understanding has led to the emergence of powerful frameworks for applications development like PETSc [3] and POOMA [1]. As with standard libraries, these tools simplify software development and raise the level of abstraction.

Another important metric for measuring the maturity of scientific parallel software is by the complexity of applications that have moved to parallel platforms. In the early days of parallel computing, the majority of successful applications were user-written research-level codes designed to model a particular narrow problem in science or engineering. The early application conferences in this field were rich with papers where individual researchers, many of them students, had written their own new parallel code from scratch to simulate a particular phenomenon. Such codes were often small – a few thousand lines. If the physical phenomena were innately parallel, such codes often performed very scalably on hundreds or even thousands of processors on early machines, though it was only on very large parallel machines that they could compete in terms of raw performance with the multi-head vector supercomputers of the day (e.g. the Cray Y-MP).

In the last few years, we have begun to see full-scale engineering and science applications run scalably on parallel machines. Often these packages encompass a rich feature set, tens to hundreds of thousands of lines of code, and a large user base. We cite a few examples:

- finite element fluid flow: SALSA [7]
- molecular modeling: NWCHEM suite[15], QUEST [16], CHARMM [5]
- transient dynamics: PRONTO [2]
- particle in cell electrodynamics: 3DPIC [8]

These codes are all scalable and achieve very high performance, as evidenced by the recent entry of many of them in the Gordon Bell competition. This IEEE-sponsored contest gives awards each year to applications which perform at the very high end of parallel computing. The first finalists ten years ago were small research and proof-of-concept codes that ran at less than 1 Gigaflop/sec. Ten years later, large production-scale codes dominate the competition and the performance of the 1998 winner reached the 1 Teraflop/sec (sustained) performance milestone [17]. This is a remarkable thousand-fold increase in only ten years! Much of the increase is due to faster hardware and larger machines, but it is also impressive that code complexity has kept pace as application developers have become more accustomed to thinking in parallel and more sophisticated in their algorithm development.

A key feature all of these applications and libraries have in common is that they were written essentially from scratch with parallelism in mind. In some cases this meant starting with an empty file. In others, legacy code was kept, but a fundamental redesign of data structures, code structure, and solution methods was necessary for a distributed memory implementation. This is clearly costly in terms of development time. But once it is finished, the new version replaces the old one, and it runs portably on either serial machines or any kind of parallel platform.

By contrast, several of these applications have competitors that have not yet made the transition to massively parallel, at least in the commercial or most widely supported versions. For example, the DYNA package for transient dynamics and GAUSSIAN code for molecular modeling have resisted fully scalable parallel implementation. This is primarily due to the sheer volume of legacy serial code and man-years of development invested in these very popular applications. The inability to create parallel versions of such codes for their broad user communities has been the Achilles heel of parallel scientific computing. It is an open question whether such popular legacy codes will ever be implemented in parallel, or will be supplanted by their competitors, or whether their lack of existence will prove to be the eventual downfall of high-end parallel computing.

5 Parallel Computing in Industry

Private industry plays the role of both producer and consumer of parallel computing. On the producer side, computer hardware and software companies build the components that are in turn used to build large-scale parallel computers. In fact, large-scale parallel computers are economically feasible to build only because the cost of component development is spread across a variety of other end-products. On the consumer side, there are many industrial applications that are voracious consumers of parallel computing, many of these applications, e.g., parallel web servers, being far afield from the original parallel computing application base.

One of the most dramatic influences on parallel computing over the last decade has the been the tremendous growth in the use of computers and networks in businesses and industries outside of traditional computational science and engineering. As a result of this growth, and the relatively slower growth in the demand for computers for science and engineering applications, we find that large-scale parallel computing customers have minimal influence today on the design of new computer components. At the same time, because of the increased demand for more powerful computers and networks in other fields, large-scale parallel computing has benefited greatly from having much lower cost components, even if the components are not optimally designed for science and engineering applications. Two specific results of the growth in computing outside of science and engineering applications are the decline of massively parallel processing (MPP) computers and the corresponding rise in shared memory parallel (SMP) systems and commodity-off-the-shelf (COTS) systems. We discuss these in more detail below.

The Decline of MPP Systems

In the mid 1980s, massively parallel processing (MPP) computer systems were predicted to be the next big advance in high performance computing. Many new companies, e.g., nCUBE, Thinking Machines, Kendall Square Research, etc. were formed to build MPP systems. In addition, several existing companies such as Intel, IBM and Cray Research also decided to design and build these systems.

In those early days, a few universities, national laboratories, petroleum companies and large weather forecasting facilities were among some of the organizations that embraced MPP systems and made them work. These organizations had compelling need for the performance MPP systems promised and also had the resources to build an infrastructure to support the vastly different computing model that MPP systems required.

Ultimately however the industrial marketplace, as a whole, has not adopted MPP systems. There are many opinions as to why MPP systems has not succeeded in industry, but we believe it is primarily because many of the key applications have not been successfully ported and integrated into the existing computing environment. There have been many notable efforts to get industrial applications working well on MPP systems and some succeeded, but many did not. Even those efforts that have been ultimately successful came too late to save the industrial MPP marketplace. A second contributing factor to the decline of MPP systems in industry is the steady increase in capabilities of workstations and PCs. These low cost have systems provided a doubling of single processor performance every 18-24 months with little or no change required in applications, and furthermore have started coming in multiple processor configurations. These developments have had an impact on the entire high-end computer system market, and have made it especially hard to justify large-scale efforts to get MPP systems working.

Without a large industrial customer base, MPP computer companies could not stay in business. As a result, most computer systems that were designed to be commercially available MPP systems are either gone or on their last generation. The exceptions are Compaq, which will introduce a SMP-node MPP, and IBM, which shows no signs of quitting their MPP development. Most major MPP systems development efforts today are focused at national laboratories or are being developed to address other very specific customer needs. However, even though commercially available MPP systems have declined, large-scale parallel computing in industry has gained new momentum, and the efforts in parallel application development have found new types of computer systems that can deliver impressive performance improvements.

The Rise of SMP and COTS Systems

The decline of MPP systems in industry did not mean the decline of parallel computing in industry. In fact, the growing availability of shared memory parallel (SMP) computer systems and, even more interestingly, the growth of commodity-off-the-shelf (COTS) systems have given parallel computing new momentum in the industrial marketplace.

SMP systems SMP systems are available from many computer system vendors. The most important reason for the success of SMPs is that they provide a flexible platform for a variety of uses. They can simultaneously be viewed as (i) a better throughput engine for many independent computer jobs, (ii) a shared memory parallel system to provide incremental performance improvements for a few compute-intensive segments of an existing serial application and (iii) a distributed memory system with fast message passing.

Item (i) above is by far the most important reason for the success of SMP systems. It provides a cost-effective means of increasing the computing capacity for many different types of computer users. This type of parallel computing is at the job level and generally not of interest to the parallel computing community. However, it is important because it has made SMP systems a commercially successful product.

Item (ii) is increasingly becoming a reason for growth in SMP systems sales. This is driven primarily by the soon-to-be ubiquitous presence of SMP PCs along with OpenMP, an emerging standard for SMP programming. However, the effectiveness of this type of parallelism is usually limited to a few processors because only small sections of the code are rewritten and the rest runs in serial mode. In fact, successful parallelism, even for a small number of processing elements, is seldom achieved automatically. Significant code modifications, directives or other programmer interventions are often required for efficient parallelism.

Item (iii) is of most interest to us. It would seem that running an SMP system and pretending it has distributed memory would be a waste of the SMP hardware, but it is in fact often the best way to use an SMP system that has a lot of processors. This is because almost all SMP systems have a non-uniform memory access (NUMA) architecture, primarily in the form of a large secondary cache. Treating memory as distributed increases the locality of memory reference. It also reduces false–sharing cache conflicts, memory–bank conflicts and other problems that make UMA SMPs non-scalable. Furthermore, if the message passing library is customized for the SMP hardware, it can be implemented with memory copies and achieve efficient and scalable parallel performance.

COTS Systems In addition to SMP systems, COTS systems are becoming a very important parallel computing platform. Low cost, high performance PCs, workstations and networking systems, along with increasing interest in Linux, gives us the ability to create inexpensive and powerful parallel distributed memory computers from components that are available at your local shopping mall, or from your favorite online dealer. As with SMP systems, COTS systems are made possible because the components that go into building one are useful to the general computing community. The growing availability of COTS systems also increases the attractiveness of distributed memory applications. No longer do you need a high priced computer to benefit from parallelism. Unlike SMP systems, which become very expensive beyond a small number of processors. COTS systems can currently inexpensively deliver scalable parallel performance on tens of processors for the right types of applications. Certainly large-scale COTS systems require some modification of system hardware and software to get good performance on a broad set of applications, as is exemplified by the CPlant project at Sandia, but the investment is far less than that required to develop a custom designed computer system.

Shortcomings SMP and COTS systems certainly have shortcomings that can make them less attractive to parallel computing users than MPP systems. In particular, SMP systems are typically not set up to dedicate a set of processors to an entire application. Instead, SMP systems are usually set up to share resources dynamically, meaning processors will come and go from a particular job during the life of that job. Although this appears to be a trivial issue, it is in fact something that is only slowly being addressed by SMP systems vendors. A further complication of SMP systems comes from cache coherency. Distributed memory applications do not need, and do not want cache coherency across processors. Unfortunately, this feature is part of the hardware support and cannot be shut off. As a result, false cache line sharing can seriously degrade performance of parallel applications.

COTS systems are currently not widely used in many industries because the COTS system is not really a single system but a collection of systems and there is a build-it-yourself requirement. Currently there are a few universities, research laboratories and small companies that are addressing this issue and having good success. They are taking these individual components and integrating them both physically in terms of packaging and also logically in terms of a layer of administrative software to make the cluster look as much like a single system as possible. We see this trend continuing, and think that eventually larger companies will also be providing these types of systems and support.

The Status of Parallel Computing in Industry

Parallel computing in industry must objectively be considered a secondary issue, except in enterprises who care a great deal about large scale scientific and engineering applications, e.g., oil exploration and automotive companies. Other factors like cost (especially cost of ownership), accuracy and relevance of results, and integration of computers into daily business practices are much more important to most industrial computer users. At the same time, the emergence of SMP systems, especially multiprocessor PCs, and the growing ease of setting up inexpensive COTS systems promise to make access to parallel computing easier and more cost effective than ever before.

Parallel computing has an established foothold in many industrial markets. Most notably, some parts of the aerospace industry have used parallel cluster computing for more than a decade to do large computational fluid dynamics (CFD) calculations. High-end workstations, used during the day by CAD engineers, are transformed into parallel computers for external airflow calculations during the off hours. Fault tolerance, batch processing and load balancing are built into the application since there is minimal OS support. By taking this approach, the aerospace industry has been able to utilize a latent computing resource.

A relatively small number of industries have driven the demand for large–scale parallel machines. These include the oil and gas industry and the automotive industry. The increased availability of parallel commercial codes has increased the use of parallel machines. For example, parallel CFD codes are becoming commonplace in a variety of engineering markets as a result of the successful introduction of parallel version of many of the most important general purpose CFD packages, e.g., FLUENT, STAR-CD, CFX, etc. Similarly, parallel versions of industry-standard applications are being slowly introduced in many other markets, including automotive, chemical/pharmaceutical, oil and gas, environmental and electronics. However, with few exceptions the process of introducing parallel computing into the production-computing environment is clearly in the early stages of development.

Interestingly, it is the growing availability of SMP and COTS systems, systems that were not custom-designed for parallel applications, that is finally spurring the growth in industrial parallel computing. The wide availability of these systems is what has broadly attracted the attention of application software vendors to parallel computing, something that custom-designed MPP systems could not accomplish. The tremendous growth of computing in industry over the past decade, and corresponding decrease in costs, is transforming parallel computing from being something highly specialized and available to only a few specialists into something that is accessible to anyone with a few PCs, network cards and a hub. This is a welcome change because it increases the base of parallel computing platforms, the number of parallel computing users and the number of parallel application developers. However, we hold no illusions that the increase in number of low-end parallel systems makes high-end systems easy to build or use. A great deal of effort is still required to utilize effectively hundreds or thousands of processors within a single application.

6 Government involvement in MPP Computing

The government and its national laboratories have had a major impact on high performance computing from its inception, both through funding and technology development. In the 60's and early 70's, CDC's successes with the CDC 1604, CDC 6400, CDC 6600 and CDC 7600 computers were driven by the needs of government laboratories and government-funded universities. From the mid-seventies Cray Research depended on government laboratories as the first adopters of its Cray-1S, Cray X-MP, Cray Y-MP and C-90 parallel vector processors. These two families dominated high performance computing until the advent of Massively Parallel Processing (MPP).

In the 80s and early 90s, DARPA funded computer architecture research at Thinking Machines Corporation and Intel Corporation. This funding directly impacted the development of the Connection Machine and the Paragon. A third major vendor at the time, nCUBE, entered into a partnership with Sandia National Laboratories. Although none of these three vendors currently build supercomputers, they played a key role, along with Cray Research, in developing and demonstrating massively parallel computing technology.

A number of federal laboratories have developed important high performance technologies. Sandia was a leader in demonstrating the practicality of solving real engineering and science problems on massively parallel computers – its load balancing software (Chaco) and parallel iterative solver library (Aztec) have been licensed to hundreds of users. Oak Ridge National Laboratory developed the widely used PVM software that enables applications to execute across distributed computers. Los Alamos National Laboratory successfully implemented a widely used ocean circulation model on the CM-5. Argonne National Laboratory and Sandia have pioneered the development of immersive visualization technology so that users could more easily interpret and interact with the huge data sets that are generated on massively parallel computers.

Universities have also made key contributions to MPP technology. Much of this work has been funded by DARPA, NSF and DOE. Caltech helped develop much of the early scalable parallel hardware technology. The Pittsburgh Supercomputer Center made MPP computing widely available to university researchers with its T3D and T3E systems.

More recently, the U.S. Department of Energy's Accelerated Strategic Computing Initiative (ASCI) has funded the development of high-end supercomputing technology. ASCI has supported research partnerships between Sandia and Intel, Lawrence Livermore National Laboratory and IBM, and Los Alamos National Laboratory and SGI/Cray. All three of these partnerships are aimed at Tera scale computing.

7 Summary

In this paper, we have discussed the major aspects of massively parallel computing, including hardware, system software and algorithms. Progress in parallel computing has been slower than expected, but both hardware and software have continued to advance. The current generation of massively parallel computers achieves peak computational rates in excess of two teraflops, far exceeding the performance of the best single processor computers. And massively parallel computing is now widely used in production in government and some industrial applications, e.g., oil and gas applications. The speed of MPP computers has depended on advances in both microprocessors and interprocessor communication networks. We expect further significant increases in hardware capabilities, and we expect that the trend towards the use of commodity components will continue.

System software and development environments have considerably lagged hardware and algorithm development, and have often been adaptations of workstation software. Yet, despite this and the fundamental challenges of fast changing hardware, commoditization, 64-bit addressing, and shared memories, system software is likely to improve. The use of partitioning within a machine to reduce the concurrent requirements on the software allows for simpler solutions. The advent of open operating systems (such as OpenBSD, Linux, and Solaris) allows the creation of prototypes and the performance of fundamental research. If the high-performance community makes a commitment to provide access to hardware for system software research and development, there should be a significant payoff.

Finally, we are optimistic about the future of high-end parallel scientific computing algorithms and software. The reasons for this optimism include

(1) the emerging availability of parallel libraries and frameworks,
(2) the maturation of parallel algorithms for common application areas such as finite element codes,
(3) the growing cadre of programmers who are comfortable with the effort involved in writing message-passing codes, and
(4) the ubiquitous low-end hardware that is making moderate and even large-scale parallel computers available to the masses, which will increase the supply of and demand for parallel codes.

During the past decade, parallel computing has provided a rapid increase in modeling and simulation capabilities for science and engineering, a capability of which we are only beginning to take advantage.

References

[1] S. Atlas, S. Banerjee, J. C. Cummings, P. J. Hinker, M. Srikant, J. V. W. Reynders, and M. Tholburn. POOMA: A high performance distributed simulation environment for scientific applications. In *Proc. Supercomputing '95*, San Diego, CA, December 1995. ACM and IEEE.

[2] S. Attaway, T. Barragy, K. Brown, D. Gardner, B. Hendrickson, S. Plimpton, and C. Vaughan. Transient solid dynamics simulations on the Sandia/Intel Teraflop computer. In *Proc. SC'97*. ACM and IEEE, November 1997.

[3] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. PETSc 2.0 users manual. Technical Report ANL-95/11 - Revision 2.0.22, Argonne National Laboratory, 1998.

[4] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide.* SIAM, Philadelphia, PA, 1997.

[5] B. R. Brooks and M. Hodoščeк. Parallelization of CHARMM for MIMD machines. *Chemical Design Automation News*, 7:16–22, 1992.

[6] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, , and K. Yelick. Parallel programming in Split-C. In *Proc. Supercomputing '93*, Portland, OR, November 1993. ACM and IEEE.

[7] K. D. Devine, G. L. Hennigan, S. A. Hutchinson, A. G. Salinger, J. N. Shadid, and R. S.Tuminaro. High performance MP unstructured finite element simulation of chemically reacting flows. In *Proc. SC'97*. ACM and IEEE, November 1997.

[8] J. W. Eastwood, W. Arter, N. J. Brealey, and R. W. Hockney. Body-fitted electromagnetic PIC software for use on parallel computers. *Comp Phys Comm*, 87:455–178, 1995.

[9] David S. Greenberg, Ron Brightwell, Lee Ann Fisk, Arthur Maccabe, and Rolf Riesen. A system software architecture for high-end computing. In *Proceedings of SC'97*, 1997. Available at http://www.supercomp.org/sc97/proceedings.

[10] J. L. Gustafson, G. R. Montry, and R. E. Benner. Development of parallel methods for a 1024–processor hypercube. *SIAM J. Sci. Stat. Comput.*, 9:609–638, 1988.

[11] B. Hendrickson and R. Leland. The Chaco user's guide: Version 2.0. Technical Report SAND94–2692, Sandia National Laboratories, Albuquerque, NM, October 1994.

[12] S. A. Hutchinson, L. V. Prevost, J. N. Shadid, C. Tong, and R. S. Tuminaro. Aztec user's guide: Version 2.0. Technical Report ANL-95/11 - Revision 2.0.22, Sandia National Laboratories, 1998.

[13] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. Technical Report CORR 95–035, University of Minnesota, Dept. Computer Science, Minneapolis, MN, June 1995.

[14] MPI-IO: a parallel file I/O interface for MPI. The MPI-IO Committee, April 1996. Version 0.5. See WWW http://lovelace.nas.nasa.gov/MPI-IO/mpi-io-report.0.5.ps.

[15] http://www.emsl.pnl.gov:2080/docs/nwchem/nwchem.html. Technical report, 1996.

[16] M. P. Sears, K. Stanley, and G. Henry. QUEST: Gflop performer. In *Proc. SC'97*. ACM and IEEE, November 1997.

[17] B. Ujfalussy, X. Wang, X. Zhang, D. M. C. Nicholson, W. A. Shelton, G. M. Stocks, A. Canning, Yang Wang, and B. L. Gyorffy. High performance first principles method for non-equilibrium states in magnets. In *Proc. SC'98*. ACM and IEEE, November 1998.

[18] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance Java dialect. In *Proc. Workshop on Java for High-Performance Network Computing*, Stanford, CA, February 1998. ACM.