Scalable Parallel Computers for Real-Time Signal Processing

KAI HWANG and ZHIWEI XU

In this article, we assess the state-of-the-art technology in massively parallel processors (MPPs) and their variations in different architectural platforms. Architectural and programming issues are identified in using MPPs for time-critical applications such as adaptive radar signal processing.

First, we review the enabling technologies. These include high-performance CPU chips and system interconnects, distributed memory architectures, and various latency hiding mechanisms. We characterize the concept of scalability in three areas: resources, applications, and technology. Scalable performance attributes are analytically defined. Then we compare MPPs with symmetric multiprocessors (SMPs) and clusters of workstations (COWs). The purpose is to reveal their capabilities, limits, and effectiveness in signal processing.

In particular, we evaluate the IBM SP2 at MHPCC [33], the Intel Paragon at SDSC [38], the Cray T3D at Cray Eagan Center [1], and the Cray T3E and ASCI TeraFLOP system recently proposed by Intel [32]. On the software and programming side, we evaluate existing parallel programming environments, including the models, languages, compilers, software tools, and operating systems. Some guidelines for program parallelization are provided. We examine data-parallel, shared-variable, message-passing, and implicit programming models. Communication functions and their performance overhead are discussed. Available software tools andcommunication libraries are introduced.

Our experiences in porting the MIT/Lincoln Laboratory STAP (space-time adaptive processing) benchmark programs onto the SP2, T3D, and Paragon are reported. Benchmark performance results are presented along with some scalability analysis on machine and problem sizes. Finally, we comment on using these scalable computers for signal processing in the future.

Scalable Parallel Computers

A computer system, including hardware, system software, and applications software, is called *scalable* if it can *scale up* to accommodate ever increasing users demand, or scale down to improve cost-effectiveness. We are most interested in scaling up by improving hardware and software resources to expect proportional increase in performance. Scalability is a multi-dimentional concept, ranging from resource, application, to technology [12,27,37].

Resource scalability refers to gaining higher performance or functionality by increasing the *machine size* (i.e., the number of processors), investing in more storage (cache, main memory, disks), and improving the software. Commercial MPPs have limited resource scalability. For instance, the normal configuration of the IBM SP2 only allows for up to 128 processors. The largest SP2 system installed to date is the 512-node system at Cornell Theory Center [14], requiring a special configuration.

Technology scalability refers to a scalable system which can adapt to changes in technology. It should be *generation* scalable: When part of the system is upgraded to the next generation, the rest of the system should still work. For instance, the most rapidly changing component is the processor. When the processor is upgraded, the system should be able to provide increased performance, using existing components (memory, disk, network, OS, and application software, etc.) in the remaining system. A scalable system should enable integration of hardware and software components from different sources or vendors. This will reduce the cost and expand the system's usability. This *heterogeneity scalability* concept is called *portability* when used for software. It calls for using components with an open, standard architecture and interface. An ideal scalable system should also allow *space scalability*. It should allow scaling up from a desktop machine to a multi-rack machine to provide higher performance, or scaling down to a board or even a chip to be fit in an embedded signal processing system.

To fully exploit the power of scalable parallel computers, the application programs must also be scalable. *Scalability over machine size* measures how well the performance will improve with additional processors. *Scalability over problem size* indicates how well the system can handle large problems with large data size and workload. Most real parallel appli-

1053-5888/96/$5.00©1996IEEE

Table 1: Architectural Attributes of Five Parallel Computer Categories					
Attribute	PVP	SMP	DSM	MPP	COW
Example Systems	Cray C-90, Cray T-90	Cray CS6400, DEC 8000	DASH Cray T3D	Intel Paragon IBM SP2	Berkeley NOW, Alpha Farm
Processor Type	Custom Vector Processor	Commodity Microprocessor	Commodity Microprocessor	Commodity Microprocessor	Commodity Microprocessor
Memory model	Centralized Shared	Centralized Shared	Distributed Shared	Distributed Unshared	Distributed Unshared
Address Space	Single	Single	Single	Multiple	Multiple
Access Model	UMA	UMA	NUMA	NORMA or NUMA	
Interconnect	Custom Crossbar	Bus or Crossbar	Custom Network	Custom Network	Commodity Network

cations have limited scalability in both machine size and problem size. For instance, some coarse-grain parallel radar signal processing program may use at most 256 processors to handle at most 100 radar channels. These limitations can not be removed by simply increasing machine resources. The program has to be significantly modified to handle more processors or more radar channels.

Large-scale computer systems are generally classified into six architectural categories [25]: the *single-instruction-multiple-data* (SIMD) machines, the *parallel vector processors* (PVPs), the *symmetric multiprocessors* (SMPs), the *massively parallel processors* (MPPs), the clusters of workstations (COWs), and the *distributed shared memory multiprocessors* (DSMs). SIMD computers are mostly for special-purpose applications, which are beyond the scope of this paper. The remaining categories are all MIMD (*multiple-instruction-multiple-data*) machines.

Important common features in these parallel computer architectures are characterized below:

- *Commodity Components*: Most systems use commercially off-the-shelf, commodity components such as microprocessors, memory chips, disks, and key software.
- *MIMD*: Parallel machines are moving towards the MIMD architecture for general-purpose applications. A parallel program running on such a machine consists of multiple processes, each executing a possibly different code on a processor autonomously.
- *Asynchrony*: Each process executes at its own pace, independent of the speed of other processes. The processes can be forced to wait for one another through special synchronization operations, such as semaphores, barriers, blocking-mode communications, etc.
- *Distributed Memory*: Highly scalable computers are all using distributed memory, either shared or unshared. Most of the distributed memories are accessed by the *none-uniform memory access* (NUMA) model. Most of the NUMA machines support *no remote memory access* (NORMA). The conventional PVPs and SMPs use the centralized, *uniform memory access* (UMA) shared memory, which may limit scalability.

Parallel Vector Processors

The structure of a typical PVP is shown in Fig.1a. Examples of PVP include the Cray C-90 and T-90. Such a system contains a small number of powerful custom-designed *vector processors* (VPs), each capable of at least 1 Gflop/s performance. A custom-designed, high-bandwidth crossbar switch connects these vector processors to a number of *shared memory* (SM) modules. For instance, in the T-90, the shared memory can supply data to a processor at 14 GB/s. Such machines normally do not use caches, but they use a large number of vector registers and an instruction buffer.

Symmetric Multiprocessors

The SMP architecture is shown in Fig.1b. Examples include the Cray CS6400, the IBM R30, the SGI Power Challenge, and the DEC Alphaserver 8000. Unlike a PVP, an SMP system uses commodity microprocessors with on-chip and off-chip caches. These processors are connected to a shared memory though a high-speed bus. On some SMP, a crossbar switch is also used in addition to the bus. SMP systems are heavily used in commercial applications, such as database systems, on-line transaction systems, and data warehouses. It is important for the system to be *symmetric*, in that every processor has equal access to the shared memory, the I/O devices, and operating system. This way, a higher degree of parallelism can be released, which is not possible in an *asymmetric* (or *master-slave*) multiprocessor system.

Massively Parallel Processors

To take advantage of higher parallelism available in applications such as signal processing, we need to use more scalable computer platforms by exploiting the distributed memory architectures, such as MPPs, DSMs, and COWs. The term MPP generally refers to a large-scale computer system that has the following features:

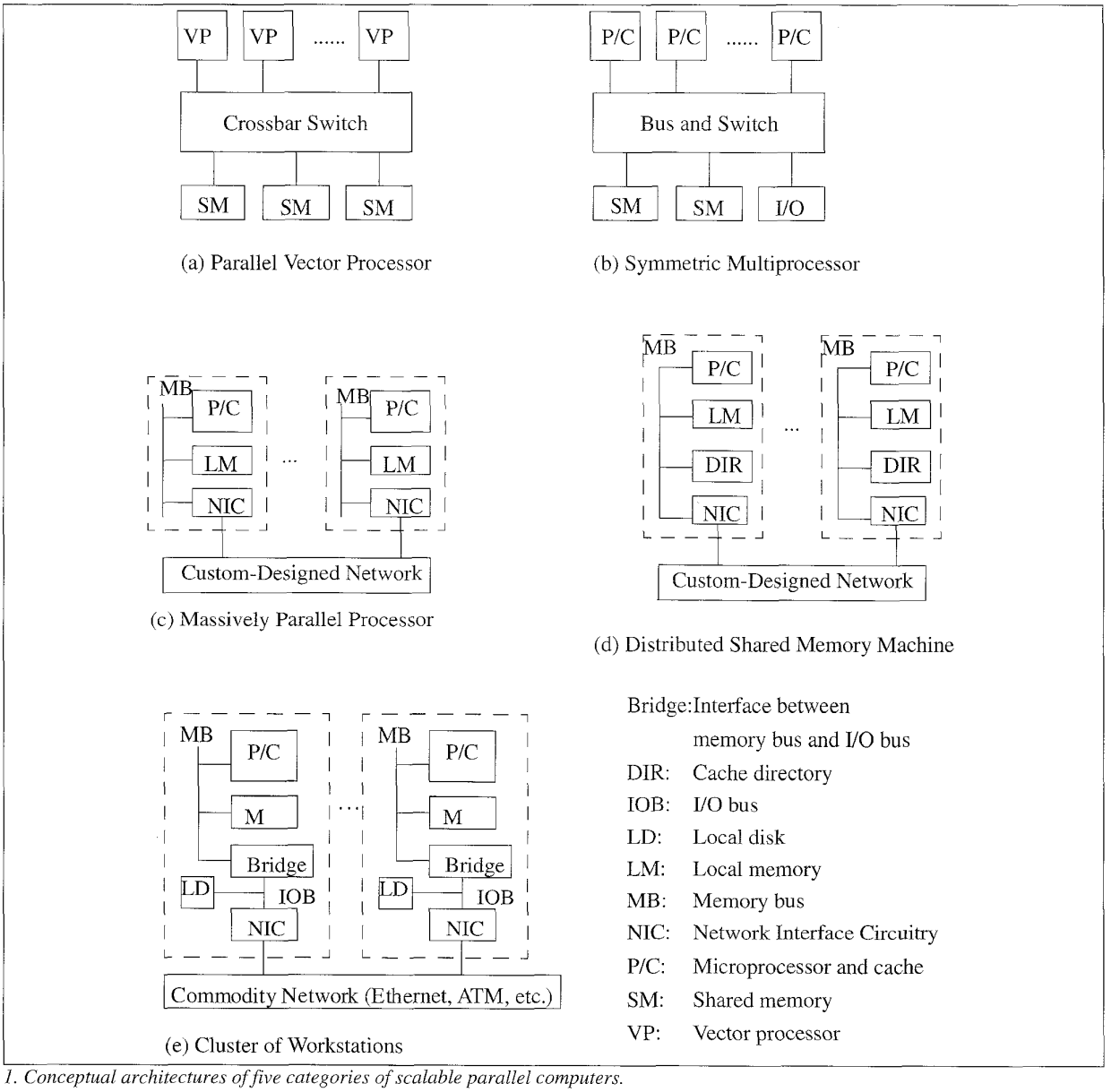
- It uses commodity microprocessors in processing nodes.
- It uses physically distributed memory over processing nodes.

- It uses an interconnect with high communication bandwidth and low latency.
- It can be scaled up to hundreds or even thousands of processors.

By this definition, MPPs, DSMs, and even some COWs in Table 1 are qualified to be called as MPPs. The MPP modeled in Fig.1c is more restricted, representing machines such as the Intel Paragon. Such a machine consists a number of *processing nodes*, each containing one or more microprocessors interconnected by a high-speed memory bus to a local memory and a *network interface circuitry* (NIC). The nodes are interconnected by a high-speed, proprietary, communication network.

Distributed Shared Memory Systems

DSM machines are modeled in Fig.1d, based on the Stanford DASH architecture. *Cache directory* (DIR) is used to support distributed coherent caches [30]. The Cray T3D is also a DSM machine. But it does not use the DIR to implement coherent caches. Instead, the T3D relies on special hardware and software extensions to achieve the DSM at arbitrary block-size level, ranging from words to large pages of shared data. The main difference of DSM machines from SMP is that the memory is physically distributed among different nodes. However, the system hardware and software create an illusion of a single address space to application users.



1. Conceptual architectures of five categories of scalable parallel computers.

Clusters of Workstations

The COW concept is shown in Fig.1e. Examples of COW include the Digital Alpha Farm [16] and the Berkeley NOW [5]. COWs are a low-cost variation of MPPs. Important distinctions are listed below [36]:

- Each node of a COW is a complete workstation, minus the peripherals.
- The nodes are connected through a low-cost (compared to the proprietary network of an MPP) commodity network, such as Ethernet, FDDI, Fiber-Channel, and ATM switch.
- The network interface is *loosely coupled* to the I/O bus. This is in contrast to the *tightly coupled* network interface which is connected to the memory bus of a processing node.
- There is always a local disk, which may be absent in an MPP node.
- A complete operating system resides on each node, as compared to some MPPs where only a microkernel exists. The OS of a COW is the same UNIX workstation, plus an add-on software layer to support parallelism, communication, and load balancing.

The boundary between MPPs and COWs are becoming fuzzy these days. The IBM SP2 is considered an MPP. But it has also a COW architecture, except that a proprietary *High-Performance Switch* is used as the communication network. COWs have many cost-performance advantages over the MPPs. Clustering of workstations, SMPs, and or PCs is becoming a trend in developing scalable parallel computers [36].

MPP Architectural Evaluation

Architectural features of five MPPs are summarized in Table 2. The configurations of SP2, T3D and Paragon are based on current systems our USC team has actually ported the STAP benchmarks. Both SP2 and Paragon are message-passing multicomputers with the NORMA memory access model [26]. Internode communication relies on explicit message passing in these NORMA machines. The ASCI TeraFLOP system is the successor of the Paragon. The T3D and its successor T3E are both MPPs based on the DSM model.

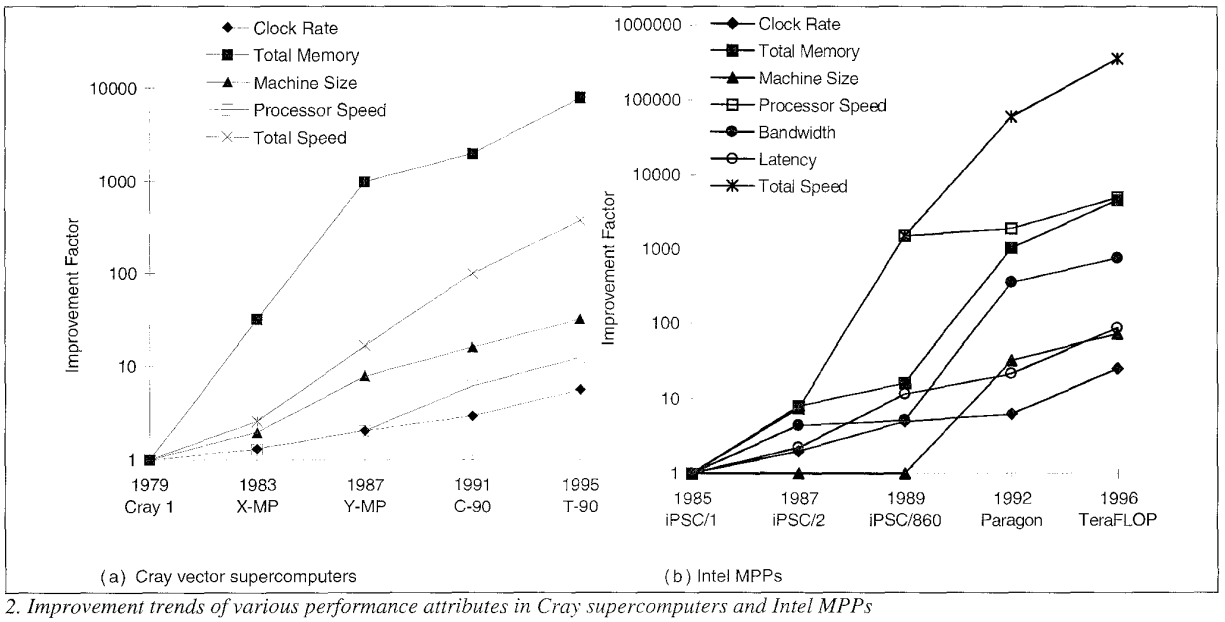
MPP Architectures

Among the three existing MPPs, the SP2 has the most powerful processors for floating-point operations. Each POWER2 processor has a peak speed of 267 Mflop/s, almost two to three times higher than each Alpha processor in the T3D and each i860 processor in the Paragon, respectively. The Pentium Pro processor in the ASCI TFLOPS machine has the potential to compete with the POWER2 processor in the future. The successor of T3D (the T3E) will use the new Alpha 21164 which has the potential to deliver 600 Mflop/s with a 300 MHz clock. T3E and TFLOPS are scheduled to appear in late 1996.

The Intel MPPs (Paragon and TFLOPS) continue using the 2-D mesh network, which is the most scalable interconnect among all existing MPP architectures. This is evidenced by the fact that the Paragon scales to 4536 nodes (9072

Table 2: Comparison of Current and Future Massively Parallel Processors

MPP Models	IBM SP2	Cray T3D	Cray T3E	Intel Paragon	Intel ASCI TeraFLOPS
A Large Sample Configuration	400-node 100 Gflop/s at MHPCC5	12-node 153 Gflop/s at NSA	Maximal 512-node, 1.2 Tflop/s	400-node 40 Gflop/s at SDSC	4536-node 1.8 Tflop/s at SNL
CPU Type	67 MHz 267 Mflop/s POWER2	150 MHz 150 Mflop/s Alpha 21064	300 MHz, 600 Mflop/s Alpha 21164	50 MHz 100 Mflop/s Intel i860	200 MHz 200 Mflop/s Pentium Pro
Node Architecture	1 processor, 64 MB-2 GB local memory, 1-4.5GB Local disk	2 processors, 64 MB memory 50 GB Shared disk	4-8 processors, 256MB-16GB DSM memory, Shared disk	1-2 processors, 16-128 MB local memory, 48 GB shared disk	2 processors 32-256 MB local memory shared disk
Interconnect and memory	Multistage Network, NORMA	3-D Torus DSM	3-D Torus DSM	2-D Mesh NORMA	Split 2-D Mesh NORMA
Operating System on Compute Node	Complete AIX (IBM Unix)	Microkernel	Microkernel based on Chorus	Microkernel	Light-Weighted Kernel (LWK)
Native Programming Mechanism	Message passing (MPL)	shared variable and message passing, PVM	shared variable and message passing, PVM	Message Passing (Nx)	Message Passing (MPI based on PUMA Portals)
Other Programming Models	MPI, PVM, HPF, Linda	MPI, HPF	MPI, HPF	SUNMOS, MPI, PVM	Nx, PVM
Point-to-point latency and bandwidth	40 μs 35 MB/s	2 μs 150 MB/s	480 MB/s	30 μs 175 MB/s	10 μs 380 MB/s



2. Improvement trends of various performance attributes in Cray supercomputers and Intel MPPs

Pentium Pro processors) in the TFLOPS. The Cray T3D/T3E use a 3-D torus network. The IBM SP2 uses a multistage Omega network. The latency and bandwidth numbers are for one-way, point-to-point communication between two node processes. The latency is the time to send an empty message. The bandwidth refers to the asymptotic bandwidth for sending large messages. While the bandwidth is mainly limited by the communication hardware, the latency is mainly limited by the software overhead. The distributed shared memory design of T3D allows it to achieve the lowest latency of only 2 μs.

Message passing is supported as a native programming model in all three MPPs. The T3D is the most flexible machine in terms of programmability. Its native MPP programming language (called Cray Craft) supports three models: the data parallel Fortran 90, shared-variable extensions, and message-passing PVM [18]. All MPPs also support the standard *Message-Passing Interface* (MPI) library [20]. We have used MPI to code the parallel STAP benchmark programs. This approach makes them portable among all three MPPs.

Our MPI-based STAP benchmarks are readily portable to the next generation of MPPs, namely the T3E, the ASCI, and the successor to SP2. In 1996 and beyond, this implies that the portable STAP benchmark suite can be used to evaluate these new MPPs. Our experience with the STAP radar benchmarks can also be extended to convert SAR (synthetic aperture radar) and ATR (Automatic target recognition) programs for parallel execution on future MPPs.

Hot CPU Chips

Most current systems use commodity microprocessors. With wide-spread use of microprocessors, the chip companies can afford to invest huge resources into research and development on microprocessor-based hardware, software, and applications. Consequently, the low-cost commodity microprocessors are approaching the performance of custom-designed processors used in Cray supercomputers. The speed performance of commodity microprocessors has been increasing steadily, almost doubling every 18 months during the past decade.

From Table 3, Alpha 21164A is by far the fastest microprocessor announced in late 1995 [17]. All high-performance CPU chips are made from CMOS technology consisting of 5M to 20M transistors. With a low-voltage supply from 2.2 V to 3.3 V, the power consumption falls between 20 W and 30 W. All five CPUs are superscalar processors, issuing 3 or 4 instructions per cycle. The clock rate increases beyond 200 MHz and approaches 417 MHz for the 21164A. All processors use dynamic branch prediction along with out-of-order RISC execution core. The Alpha 21164A, UltraSPARC II, and R10000 have comparable floating-point speed approaching 600 SPECfp92.

Scalable Growth Trends

Table 4 and Fig.2 illustrate the evolution trends of the Cray supercomputer family and of the Intel MPP family. Commodity microprocessors have been improving at a much faster rate than custom-designed processors. The peak speed of Cray processors has improved 12.5 times in 16 years, half of which comes from faster clock rates. In 10 years, the peak speed of the Intel microprocessors has increased 5000 times, of which only 25 times come from faster clock rate, the remaining 200 come from advances in the processor architecture. At the same time period, the one-way, point-to-point communication bandwidth for the Intel MPPs has increased 740 times, and the latency has improved by 86.2 times. Cray supercomputers use fast SRAMs as the main memory. The custom-designed crossbar provide high bandwidth and low communication latency. As a consequence, applications run-

Table 3: High-Performance CPU Chips for Building MPPs					
Attribute	Pentium Pro	PowerPC 620	Alpha 21164A	UltraSPARC II	MIPS R10000
Technology	BiCMOS	CMOS	CMOS	CMOS	CMOS
Transistors	5.5 M/15.5 M	7 M	9.6 M	5.4 M	6.8 M
Clock Rate	150 MHz	133 MHz	417 MHz	200 MHz	200 MHz
Voltage	2.9 V	3.3 V	2.2 V	2.5 V	3.3 V
Power	20 W	30 W	20 W	28 W	30 W
Word Length	32 bits	64 bits	64 bits	64 bits	64 bits
I/D Cache	8 kB/8 kB	32 kB/32 kB	8 kB/8 kB	16 kB/16 kB	32 kB/32 kB
L2 Cache	256 kB on a multi-chip module	1-128 MB off-chip	96 kB on-chip	16 MB off-chip	16 MB off-chip
Execution Units	5 units	6 units	4 units	9 units	5 units
Superscalar	3 way	4 way	4 way	4 way	4 way
Pipeline depth	14 stages	4-8 stages	7-9 stages	9 stages	5-7 stages
SPECint92	366	225	500	350	300
SPECfp92	283	300	750	550	600
SPECint95	8.09	225	11	NA	7.4
SPECfp95	6.70	300	17	NA	15
Special Features	CISC/RISC hybrid, 2-level speculative execution	Short pipelines, large L1 caches	Highest clock rate and density with on-chip L2 cache	Multimedia and graphics instructions	MP cluster bus supports up to 4 CPUs

Table 4: Evolution of Cray Supercomputer and Intel MPP Families								
Company	Computer	Year	Clock (MHz)	Memory Capacity (MB)	Machine Size n	Peak Speed (Mflop/s)	Bandwidth (MB/s)	Latency (ms)
Cray	Cray 1	1979	80	1	1	160		
	X-MP	1983	105	2	2	210		
	Y-MP	1987	166	256	8	333		
	C90	1991	238	256	16	1000	9444	0.1
	T90	1995	454	512	32	2000		
Intel	iPSC/1	1985	8	0.5-4.5	32-128	0.04	0.5	862
	iPSC/2	1987	16	4-16	32-128	0.3	2.2	390
	iPSC/860	1989	40	8	32-128	60	2.6	75
	Paragon	1992	50	16-128	4-2048	75	175	40
	TFLOPS	1996	200	16-128	9216	1.8Tflop/s	380	10

ning on Cray supercomputers often have higher utilizations (15% to 45%) than those (1% to 30%) in MPPs.

Performance Metrics for Parallel Applications

We define below performance metrics used on scalable parallel computers. The terminology is consistent with that proposed by the Parkbench group [25], which is consistent with the conventions used in other scientific fields, such as physics. These metrics are summarized in Table 5.

Performance Metrics

The parallel computational steps in a typical scientific or signal processing application are illustrated in Fig. 3. The algorithm consisting of a sequence of k steps. Semantically, all operations in a step should finish before the next step can begin. Step i has a computational workload of W_i *million floating-point operations* (Mflop), and takes $T_1(i)$ seconds to execute on one processor. It has a *degree of parallelism* of DOP_i. In other words, when executing on n processors with

$1 \le n \le DOP_i$, the *parallel execution time* for step i becomes $T_n(i) = T_1(i)/n$. The execution time can not be further reduced by using more processors. We assume all *interactions* (communication and synchronization operations) happen between the consecutive steps. We denote the total interaction overhead as T_o.

Traditionally, four metrics have been used to measure the performance of a parallel program: *the parallel execution time*, the *speed* (or *sustained speed*), the speedup, and the efficiency, as shown in Table 5. We have found that several additional metrics are also very useful in performance analysis.

A shortcoming of the speedup and efficiency metrics is that they tend to act in favor of slow programs. In other words, a slower parallel program can have higher speedup and efficiency than a faster one. The *utilization* metric does not have this problem. It is defined as the ratio of the measured n-processor speed of a program to the peak speed of an n-processor system. In Table 5, P_{peak} is the peak speed of a single processor. The *critical path* and the *average parallelism* are two *extreme value* metrics, providing a lower bound for execution time and an upper bound for speedup, respectively.

Communication Overhead

Xu and Hwang [43] have shown that the time of a communication operation can be estimated by a general timing model:

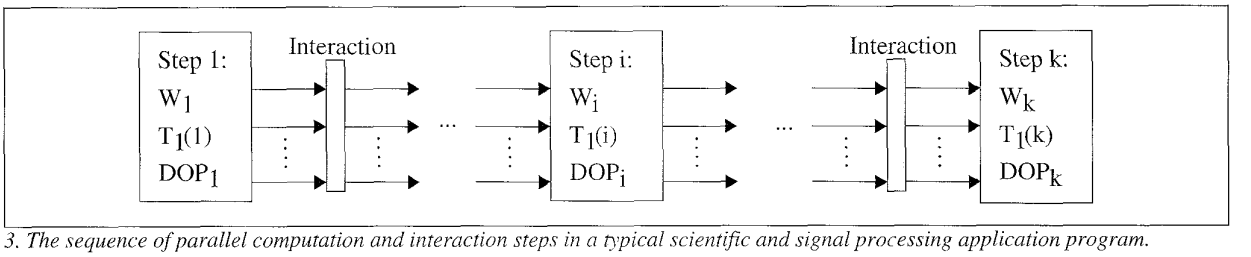
$$t = t_0(n) + \frac{m}{r_\infty(n)} \tag{1}$$

where m is the *message length* in bytes, the *latency* $t_0(n)$ and the *asymptotic bandwidth* $r_\infty(n)$ can be linear or nonlinear functions of n. For instance, timing expressions are obtained for some MPL message-passing operations on the SP2, as shown in Table 6. Details on how to derive these and other expressions are treated in [43], where the MPI performance on SP2 is also compared to the native IBM MPL operations. The total overhead T_o is the sum of the times of all interaction operations occurred in a parallel program.

Parallel Programming Models

Four models for parallel programming are widely used on parallel computers: *implicit, data parallel, message-passing*, and *shared variable*. Table 7 compares these four models from a user's perspective. A four-star (★★★★) entry indicates that the model is the most advantageous with respect to a particular issue, while a one-star (★) corresponds to the weakest model.

Parallelism issues are related to how to exploit and manage parallelism, such as process creation/termination, context switching, inquiring about number of processes.



3. The sequence of parallel computation and interaction steps in a typical scientific and signal processing application program.

Table 5: Performance Metrics used in Parallel Applications

Terminology	Definition	Unit
Total Workload	$W = \sum_{1 \le i \le k} W_i$	Mflop
Sequential Execution Time	$T_i = \sum_{1 \le i \le k} T_1(i)$	Seconds
Parallel Execution Time	$T_n = \sum_{1 \le i \le k} \frac{T_1(i)}{\min(DOP_i, n)} + T_o$	Seconds
Speed	$P_n = W/T_n$	Mflop/s
Speedup	$S_n = T_1/T_n$	Dimensionless
Efficiency	$E_n = S_n/n = T_1/(nT_n)$	Dimensionless
Utilization	$U_n = P_n/(nP_{peak})$	Dimensionless
Critical Path (or the length of the critical path)	$T_\infty = \sum_{1 \le i \le k} \frac{T_1(i)}{DOP_i}$	Seconds
Average Parallelism	T_1/T_∞	Dimensionless

Interaction issues address how to allocate workload and hot to distribute data to different processors and how to synchronize/communicate among the processors.

Semantic issues consider termination, determinacy, and correctness properties. Parallel programs are much more complex than sequential codes. In addition to infinite looping, parallel programs can deadlock or livelock. They can also be indeterminate: the same input could produce different results. Parallel programs are also more difficult to test, to debug, or to prove for correctness.

Programmability issues refer to whether a programming model facilitates the development of portable and efficient application codes.

The Implicit Model

With this approach, programmers write codes using a familiar sequential programming language (e.g., C or Fortran). The compiler and its run-time support system are responsible to resolve all the programming issues in Table 7. Examples of such compilers include KAP from Kuck and Associates [29] and FORGE from Advanced Parallel Research [7]. These are platform-independent tools, which automatically convert a standard sequential Fortran program into a parallel code.

Table 6: Communication Overhead Expressions for the SP2 MPL Operations

MPL Command	Communication Time in μs
Point-to-Point	$46+0.035m$
Broadcast	$(52\log n) + (0.029\log n)m$
Gather/Scatter	$(17\log n + 15) + (0.025n-0.02)m$
Total Exchange	$80\log n + (0.03n^{1.29})m$
Circular Shift	$6(\log n +60) + (0.003 \log n + 0.04)\, m$
Barrier	$94\log n + 10$
Reduction	$20\log n + 23$

```
parameter (MaxTargets = 10)
complex A(N,M)
integer temp1(N,M), temp2(N,M)
integer direction(MaxTargets), distance(MaxTargets)
integer i, j
!HPF$ PROCESSOR Nodes(NUMBER_OF_PROCESSORS( ))
!HPF$ ALIGN WITH A(i,j):: temp1(i,j), temp2(i,j)
!HPF$ DISTRIBUTE A(BLOCK, *) ONTO Nodes
... ...
L1:   forall (i=1:N, j=1:M) temp1(i,j) = IsTarget(A(i,j))
L2:   temp2 = SUM_PREFIX (temp1, MASK=(temp1>0))
L3:   forall (i=1:N, j=1:M;
            temp2(i,j)>0 .and. temp2(i,j)<=MaxTargets)
                distance(temp2(i,j)) = i
                direction(temp2(i,j)) = j
      end forall
```

4. A data-parallel HPF code for target detection

Some companies also provide their own tools, such as the SGI Power C Analyzer [35,39] for their Power Challenge SMPs.

Compared to explicit parallel programs, sequential programs have simpler semantics: (1) They do not deadlock or livelock. (2) They are always *determinate*: the same input always produces the same result. (3) The single-thread of control of a sequential program makes testing, debugging, and correctness verification easier than parallel programs. Sequential programs have better portability, if coded using standard C or Fortran. All we need is to recompile them when porting to a new machine. However, it is extremely difficult to develop a compiler that can transform a wide range of sequential applications into efficient parallel codes, and it is awkward to specify parallel algorithms in a sequential language. Therefore, the implicit approach suffers in performance. For instance, the NAS benchmark [11], when parallelized by the FORGE compiler, runs 2 to 40 times slower on MPPs than some hand-coded parallel programs [7].

The Data Parallel Model

The data parallel programming model is used in standard languages such as Fortran 90 and *High-Performance Fortran* (HPF) [24] and proprietary languages such as CM-5 C*. This model is characterized by the following features:

- *Single thread*: From the programmer's viewpoint, a data parallel program is executed by exactly one process with a single thread of control. In other words, as far as control flow is concerned, a data parallel program is just like a sequential program. There is no control parallelism.
- *Parallel operations on aggregate data structure*: A single step (statement) of a data parallel program can specify multiple operations which are simultaneously applied to different elements of an array or other aggregate data structure.
- *Loosely synchronous*: There is an implicit or explicit synchronization after every statement. This statement-level synchrony is *loose*, compared with the *tight* synchrony in an SIMD system which synchronizes after every instruction directly by hardware.
- *Global naming space*: All variables reside in a single address space. All statements can access any variable, subject to the usual scoping rules. This is in contrast to the message passing approach, where variables may reside in different address spaces.
- *Explicit data allocation*: Some data parallel languages, such as *High-Performance Fortran* (HPF), allows the user to explicitly specify how data should be allocated, to take advantage of data locality and to reduce communication overhead.
- *Implicit communication*: The user does not have to specify explicit communication operations, thanks to the global naming space.

The Shared Variable Model

The shared-variable programming is the native model for PVP, SMP, and DSM machines. There is an ANSI standard

Table 7: Comparison of Four Parallel Programming Models					
Issues		Implicit	Data Parallel	Message Passing	Shared Variable
Platform Independent Examples		Kap, Forge	Fortran 90, HPF	PVM, MPI	X3H5
Platform Dependent Examples		Convex Exemplar	CM C*	IBM SP2, Intel Paragon	Cray MPP, SGI Power C
Parallelism Issues		★★★★	★★★	★★	★★
Interaction Issues	Data Allocation	★★★★	★★	★	★★★
	Computation Allocation	★★★★	★★★★	★★	★★★
	Communication	★★★★	★★★	★	★★★
	Synchronization	★★★★	★★★★	★★	★
Semantic Issues	Termination	★★★★	★★★★	★	★
	Determinacy	★★★★	★★★★	★★	★
	Correctness	★★★★	★★★	★	★
Programmability Issues	Efficiency	★	★★	★★★★	★★★★
	Potability	★★★★	★★★	★★★	★

for shared-memory parallel programming (X3H5) which is language and platform independent [6]. Unfortunately, the X3H5 standard is not strictly followed by the computer industry. Therefore, a shared-variable program developed on one parallel computer is not generally potable to another machine. This model is characterized by the following features:

- *Multiple threads*: A shared-variable program uses either *multiple-program-multiple-data* (MPMD), where different codes are executed by different processes, or *single-program-multiple-data* (SPMD), where all processes execute the same code on different data domains. In either case, each process has a separate thread of control.
- *Single address space*: All variables reside in a single address space. All statements can access any variable, subject to the usual scoping rules.
- *Implicit distribution of data and computation*: Because of the single address space, data can be considered to reside in the shared memory. There is no need for the user to explicitly distribute data and computation.
- *Implicit communication*: Communication is done implicitly through reading/writing of shared variables.
- *Asynchronous*: Each process execute at its on pace. Special synchronization operations (e.g., barriers, locks, critical regions, or events) are used to explicitly synchronize processes.

The Message Passing Model

The message passing programming model is the native model for MPPs and COWs. The portability of message-passing programs is enhanced greatly by the wide adoption of the public-domain MPI and PVM libraries. This model has the following characteris tics:

- *Multiple threads* (SPMD or MPMD) of control in different nodes.
- *Asynchronous* operations at different nodes.
- *Separate Address Spaces*: The processes of a parallel program reside in different address spaces. Data variables in one process are not visible to other processes. Thus, a process can not read from or write to another process's variables. The processes interact by executing message-passing operations.
- *Explicit Interactions*: The programmer must resolve all the interaction issues, including data mapping, communication and synchronization. The workload allocation is usually done through the *owner-compute* rule, i.e., the process which owns a piece of data performs the computations associated with it.

Both shared-variable and message-passing approaches can achieve high performance. However, they require greater efforts from the user in program development. The implicit and the data parallel models shift many burdens to the compiler, thus reducing the labor cost and the program development time. This tradeoff should be based on each specific application. For signal processing, we often require the highest performance. Furthermore, a parallel signal processing application, once developed, is likely to be used for a long time. This suggests the use of message-passing model for its high efficiency and better portability.

Realization Approaches

The parallel programming models just described are realized in real systems by extending Fortran or C in three approaches: *library subroutines, new language con structs, and compiler directives*. More than one of them can be used in realizing a parallel programming model. We show in Fig.4 an example HPF code for target detection in radar applications, to illustrate the three realization methods (the algorithm in this code is credited to Michael Kumbera of the Maui High-Performance Computing Center).

- *New Constructs*: The programming language is extended with some new constructs to support parallelism and interaction. An example is the **forall** construct in Fig.4. This approach has several advantages: It facilitates portability, and it allows the compiler to check and detect possible errors associated with the new constructs. However, it requires the development of a new compiler. This approach has been used in Fortran 90, HPF, and Cray MPP Fortran.
- *Library Subroutines*: In addition to the standard libraries available to the sequential language, a new library of functions are added to support parallelism and interaction. The NUMBER_OF_PROCESSORS() function in Fig.4 is such an example. Due to its ease of implementation, this library approach is widely used, with the best known example being MPI and PVM. However, this approach leaves error checking to the user.
- *Compiler Directives*: These are formatted comments, called *compiler directives* or *pragmas*, to help the compiler to do a better job in optimization and parallelization. The three !HPF$ lines in Fig.4 are examples of compiler directives. This approach is a trade-off between the previous two approaches.

In Fig.4, we want to find the ten closest targets in an array A. The **forall** statement L1 simultaneously evaluates every element of array A, and assign temp1(i,j)=1 if A(i,j) is a target. Suppose there are four targets A(1,3), A(1,4), A(2,1), and A(4,4). Then temp1 has the value as shown below (assuming N=3 and M=4):

$$temp1 = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

$$temp2 = sum\_prefix(temp1) = \begin{bmatrix} 0 & 0 & 2 & 3 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 \end{bmatrix}$$

L2 assigns to array temp2 the prefix sum of all positive elements of temp1. The second **forall** construct updates the target list (represented by two arrays *distance and direction).*

STAP Benchmark Performance

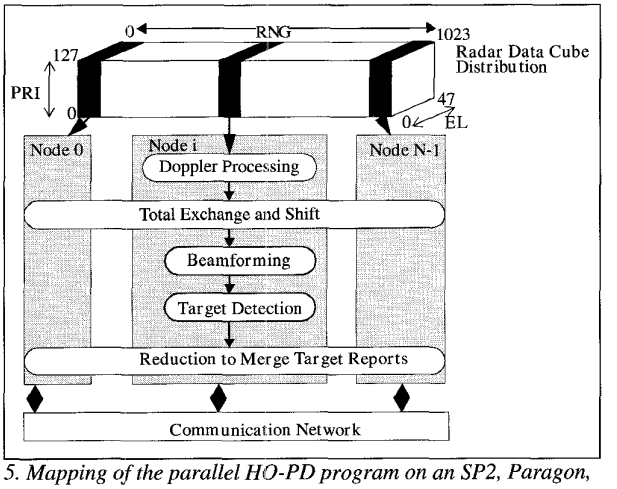
To demonstrate the performance of MPPs for signal processing, we choose to port the *space-time adaptive processing* (STAP) benchmark programs, originally developed by MIT Lincoln Laboratory for real time radar signal processing on UNIX workstations in sequential C code [34]. We have to parallelize these C codes on all three target MPPs. The STAP benchmark consists of five radar signal processing programs: *Adaptive Processing Testbed* (APT), *High-Order Post-Doppler* (HO-PD), *Element-Space PRI-Staggered Post-Doppler* (EL-Stag), *Beam-Space PRI-Staggered Post-Doppler* (BM-Stag), and *General* (GEN).

These benchmarks were written to test the STAP algorithms for adaptive radar signal processing. These programs start with *Doppler processing* (DP), in which a large number of one-dimensional FFT computations are performed. All end with target detection. The APT performs a *Householder transform* to generate a triangular learning matrix, which is used in a beamforming step to null the jammers and the clutter; whereas, in the HO-PD program, the two adaptive beamforming steps are combined into one step. The GEN program consists of four component algorithms to perform sorting, FFT, vector multiply, and linear algebra. These are the kernel routines often used in signal processing applications. The EL-Stag and the BM-Stag programs are similar to HO-PD, but use a staggered interference training algorithms.

Parallelization of STAP Programs

We have used three MPPs (IBM SP2, Intel Paragon, and Cray T3D) to execute the STAP benchmarks. Performance results on SP2 were reported in [28]. Performance results on the Paragon and T3D are yet to be released. In what follows, we show how to parallelize the HO-PD program and compare the performance of all three MPPs. We then show three ways to scale the APT application over different machine sizes of the SP2. We then analyze the scalability of three STAP programs over problem size, which is decided by the radar parameters and sensor data size.

The sequential HO-PD program was parallelized to run on the IBM SP2, the Intel Paragon, and the Cray T3D. The parallel HO-PD application program is shown in Fig.5 for all three MPP machines. The collection of radar signals forms a 3-dimensional data cube, coordinated by the numbers of *antenna elements* (EL), *pulse repetition interval* (PRI), and *range gates* (RNG). This is an SPMD program, where all nodes execute the same code consisting of three computation steps (*Doppler Processing*, *Beamforming*, and *Target Detection*) and three communication steps (*Total Exchange*, *Circular Shift*, and *Target Reduction*). The program was run in batch mode to have dedicated use of the nodes. But the communication network was shared with other users. The parallel program uses the MPI message-passing library (the MPICH portable implementation) for inter-node communication on Paragon and T3D, and the native MPL library on



5. Mapping of the parallel HO-PD program on an SP2, Paragon, or T3D.

Table 8: Breakdown of Communication Overhead and Computation Time in HO-PD						
Number of Nodes	Communication Overhead (Seconds)			Computation Time (Seconds)		
	SP2	Paragon	T3D	SP2	Paragon	T3D
1	0.0000	0.0000	0.0000	129.3637	728.7813	327.57
2	1.1203	524.0000	0.8559	64.5537	404.2969	163.786
4	0.7006	432.6406	0.4590	32.3737	169.1406	81.894
8	0.4109	105.7188	0.2587	16.1507	90.0938	40.9546
16	0.2712	0.5313	0.1304	8.1027	30.0156	20.4598
32	0.1815	0.3750	0.1300	4.0687	14.9856	10.2189
64	0.1518	0.2969	0.1038	1.9557	7.5156	5.1076
128	0.1422	0.3125	0.0853	0.9847	5.0938	2.5548
256	0.1144	0.3625	0.0989	0.4437	3.8203	1.3446

SP2. We also used the best compiler optimization options appropriate for each machine, after experimenting with all possible combinations of options.

Measured Benchmark Results

Figure 6 shows the measured parallel execution time, speed, and utilization as a function of machine size. Only the HO-PD performance is shown here. The SP2 demonstrates the best overall performance among the three MPPs. With 256 nodes, we achieved a total execution time of 0.56 seconds on the IBM SP2, corresponding to a 23 Gflop/s speed. This is partly due to SP2's fast processor, with a peak 266 Mflop/s compared to Paragon's 100 Mflop/s and T3D's 150 Mflop/s (Table 2). The degradation of Paragon performance when the number of nodes is less than 16 is due to the use of small local memory (16 MB/node in the SDSC Paragon, of which only 8 MB is available to the user applications). This results in excessive paging when a few nodes are used.

The SP2's high performance is further explained by Fig.6c, which shows the utilization of the three machines. The SP2 has the highest utilization. In particular, the sequential performance is very good, with an utilization of 36%. The relatively high utilization is due to a good compiler, a large data cache (64-256 kB per processor versus 16 kB in Paragon and 8 kB in T3D), and a large processor-memory bandwidth (as high as 654 MB/s compared to T3D's 384 MB/s according to the STREAM benchmark results, http://perelandra.cms.udel.edu:80/~mccalpin/hpc/stream/).

Execution Timing Analysis

In Table 8, we show the breakdown of the communication overhead and the computation time of the HO-PD program in all three MPPs. The parallel HO-PD program is a computation-intensive application. The communication time is less than 6% of the total time, not counting the exceptional cases of 2-8 nodes Paragon runs. There, excessive paging drastically increases both the computational and communication times. There is no communication for one node. Afterwards, the communication time decreases as n increases. This is attributed to the decreasing message size (m is about $50/n$ Mbyte) as the machine size n increases. This phenomenon was observed in almost all STAP programs executed on all three MPP machines.

Scalability over Machine Size

In an MPP, the total memory capacity increases with the number of nodes available. Assume every node has the same memory capacity of M bytes. On an n-node MPP, the total memory capacity is nM. Assume an application uses all the memory capacity M on one node and executes in W seconds (e.g., W is the sequential workload). This total workload has a sequential portion, x, and a parallelizable portion $1 - \alpha$. That is: $W = \alpha W + (1 - \alpha)W$. Three approaches have been used to get better performance as the machine size increases, which are formulated as three scalable performance laws.

Sun and Ni's Law

When n nodes are used, a larger problem can be solved due to the increased memory capacity nM. Let us assume that the parallel portion of the workload can be scaled up $G(n)$ times. That is, the scaled workload is $T^* = \mathrm{x}T + (1\text{-x})\,G(n)\,T$. Sun and Ni [41] defined the *memory bound speedup* as follows:

$$S_n = \frac{\text{sequential time for scaled workload}}{\text{parallel time for scaled workload}} = \frac{\alpha W + (1-\alpha)G(n)W}{\alpha W + (1-\alpha)G(n)W/n + T_o} = \frac{\alpha + (1-\alpha)G(n)}{\alpha + (1-\alpha)G(n)/n + T_o/W} \quad (2)$$

Amdahl's Law

When $G(n) = 1$, the problem size is fixed. Then Eq. 2 is called Amdahl's law [4] (for *fixed-workload speedup*) and has the

following form:

$$S_n = \frac{1}{\alpha + (1-\alpha)/n + T_o / W} \tag{3}$$

Gustafson's Law:

$$S_n = \frac{\alpha + (1-\alpha)n}{1 + T_o / W} \tag{4}$$

When $G(n)>n$, the computational workload increases faster than the memory requirement. Thus, the memory-bound model (Eq. 2) gives a higher speedup than the fixed-time speedup (Gustafson's law) and the fixed-workload speedup (Amdahl's law). These three speedup models are comparatively analyzed in [26].

These speedup models are plotted in Fig.7 for the parallel APT program running on the IBM SP2. We have calculated that $G(n) = 1.4n+0.37\sqrt{n} > n$, thus the fixed-memory speedup is better than the fixed-time and the fixed-workload speedups. The parallel APT program with the nominal data set has a sequential fraction a = 0.00278. This seemingly small sequential bottleneck, together with the communication overhead, limits the potential speed up to only 100 on a 256-node SP2 (the fixed-load curve). However, by increasing the problem size thus the workload, the speedup can increase to 206 using the fixed-time model, or 252 using the memory-bound model.

This example demonstrate that increasing the problem size can amortize the sequential bottleneck and communication overhead, thus improve performance. However, the problem size should not exceed the memory bound. Otherwise excessive paging will drastically degrade the performance, as illustrated in Fig.6. Furthermore, increasing the problem size is profitable only when the workload increases at a faster rate than communication overhead.

Scalability Over Problem Size

We are interested in determining how well the parallel STAP programs scale over different problem sizes. The STAP benchmark is designed to cover a wide range of radar configurations. We show the metrics for the minimal, maximal, and nominal data sets in Table 9. The input data size and the workload are given by the STAP benchmark specification [8,9,10,13]. The maximum parallelism is computed by finding the largest *degree of parallelism* (DOP) of the individual steps. The *critical path* (or more precisely, the length of the critical path) is the execution time when a potentially infinite number of nodes is used, excluding all communication overhead. For simplicity, we assume that every flop takes the same amount of time to execute. Each step's contribution to the critical path is its workload divided by its DOP.

Average Parallelism

The *average parallelism* is defined as the ratio of the total workload to the critical path. The average parallelism sets a hard upper bound on the achievable speedup. For instance, suppose we want to speed up the sequential APT program by a factor of 100. This is impossible to achieve using a minimal data set with an average parallelism of 10, but it is possible using the nominal or larger problem sizes.

When the data set increases, the available parallelism also increases. But how many nodes can be used profitably in the parallel STAP programs? A heuristic is to choose the number of nodes to be higher than the average parallelism. When the number of nodes is more than twice the average parallelism, at least 50% of the time the nodes will be idle. Using this heuristic, the parallel STAP programs with a large data set can take advantage of thousands of nodes in current and future generations of MPPs.

For sequential programs, the memory required is twice of the data set size. But for parallel programs, the memory required is six times that of the input data set, or three times of the sequential memory required. The additional memory is needed for communication buffers. We have seen (Fig.6)

Table 9: Problem Scalability of the STAP Benchmark Programs

Program	Input Data Size (MB)	Sequential Mem (MB)	Parallel Mem (MB)	Workload (Mflop)	Average Parallelism	Critical Path (Mflop)
APT (min)	0.12	0.24	0.72	5	10	0.51
APT (norm)	8.39	16.77	50	1447	177	8.19
APT (max)	1638	3276	9828	12100000	1005	12036.05
HO (min)	0.08	0.16	0.48	21	17	1.24
HO (norm)	25	50	150	12852	261	49.35
HO (max)	1638	3276	9828	33263288	65839	505.22
GEN (min)	0.13	0.26	0.78	6	103	0.05
GEN (norm)	50	100	300	5326	108	49.27
GEN (max)	16978	33956	101868	4604011	4332	1062.81

that lack of large local memory in the Paragon could significantly degrade the MPP performance.

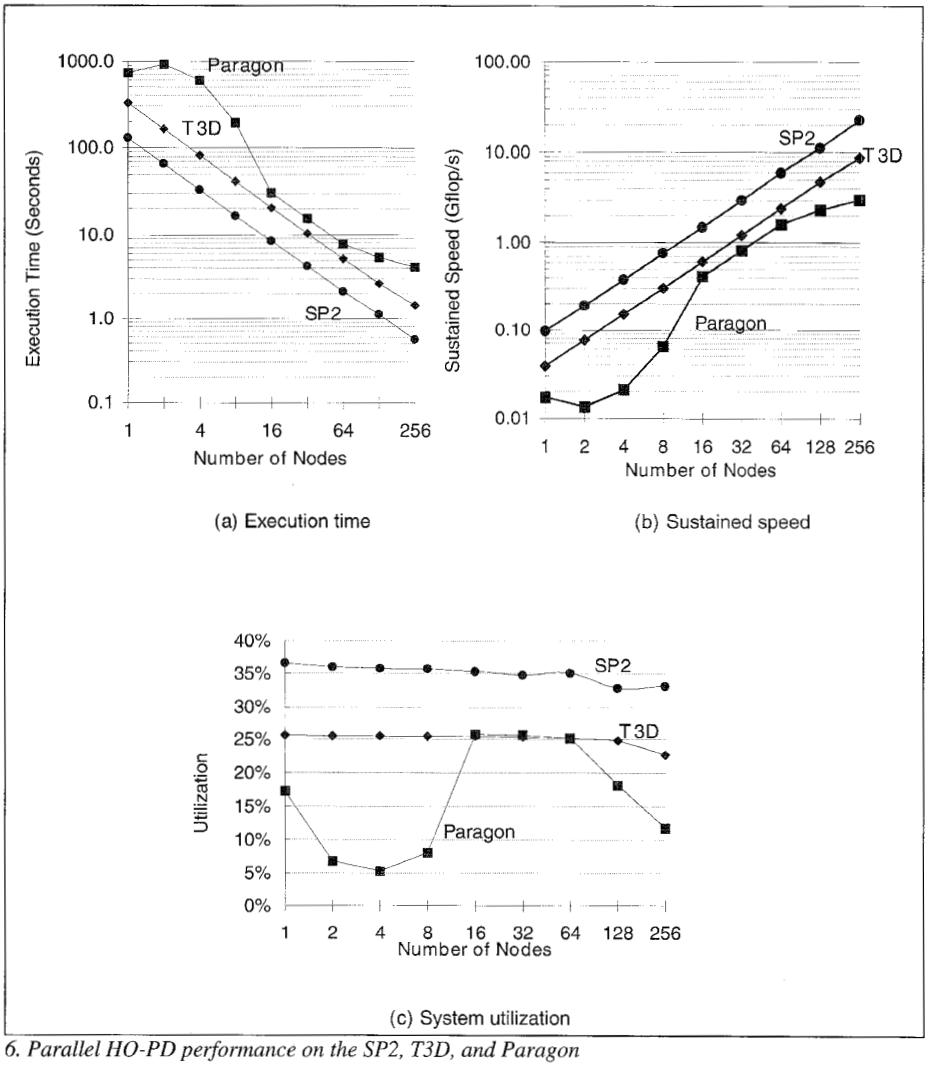
STAP Memory Requirements

Table 9 implies that for large data sets, the STAP programs must use multiple nodes, as no current MPPs have large enough memory (3 to 34 GB) on a single node. It further tells us that existing MPPs has enough memory to handle parallel STAP programs with the maximal data sets. For instance, from Table 9, an n-processor MPP should have a $102/n$ GB memory capacity per processor, excluding that used by the OS and other system software. Note that the corresponding average parallelism is 4332, larger than the maximal machine sizes of 512 for SP2 and of 2048 for Paragon and T3D. On a 512-node SP2, the per-processor memory requirement is 102GB/512 = 200 MB, and each SP2 node can have up to 2 GB memory. On a 2048-node T3D, the per-processor memory requirement is 102GB/2048 = 50 MB, and each T3D processor can have up to 64 MB memory.

Signal processing applications often have a response time requirement. For instance, we may want to compute an APT in one second. From Table 9, this is possible for the norminal data set on current MPPs, as there are only about 8 Mflop on the critical path. All the three MPPs can sustain 8 Mflop/s per processor for APT. To execute HO-PD in a second, we need each MPP node to sustain 50 Mflop/s. On the other hand, it is impossible to compute APT or HO-PD in one second for the maximal data sets, no matter how many processors are used. The reason is that it would require a processor to sustain 500 Mflop/s to 12 Gflop/s, which is impossible in any current or next generation MPPs.

Lessons Learned and Conclusions

We summarize below important lessons learned from our MPP/STAP benchmark experiments. Then we make a number of suggestions towards general-purpose signal processing on scalable parallel computer platforms including MPPs, DSMs, and COWs.



6. Parallel HO-PD performance on the SP2, T3D, and Paragon

STAP Benchmark Experience on MPPs

Among the three current MPPs: the SP2, T3D, and Paragon, we found that SP2 has the highest floating-point speed (23 Gflop/s on 256 SP2 nodes). The next is T3D and Paragon shows the lowest speed performance. The Paragon architecture is the most size scalable, the next is T3D, and the SP2 is difficult to scale beyond the current largest configuration of 512 nodes at Cornell University [15]. It is technically interesting to verify the Teraflop/s performance being projected for Intel ASCI TFLOPS system and by the Cray T3E/T3X systems in the next few years.

None of these systems is supported by a real-time operating system. A main problem is that due to interferences from the OS, execution time of a program could vary by an order of magnitude under the same testing condition, even in dedicated mode. The Cray T3D has the best communication performance, small

execution time variance, and little warm-up effect, which are desirable properties for real-time signal processing applications.

We feel that the reported timing results could be even better, if these MPPs are exclusively used for dedicated, real-time signal processing. We expect the system utilization to increase beyond 40%, if a real-time execution environment could be fully developed on these MPPs.

Developing an MPP application is a time-consuming task. Therefore, performance, portability, and scalability must be considered during program development. An application, once developed, should be able to execute efficiently on different machine sizes over different platforms, with little modification. Our experiences suggest four general guidelines to achieve these goals:

- *Coarse Granularity*: Large-scale signal processing applications should exploit coarse-grain parallelism. As shown in Fig.2, the communication latency of MPPs has been improving at a much slower rate than the processing speed. This trend is likely to continue. A coarse-grain parallel program has better scalability over current and future generations of MPPs.
- *Message Passing*: The message passing programming model has a performance advantage over the implicit and the data parallel models. It enables a program to run on MPPs, DSMs, SMPs, and COWs. In contrast, the shared-variable model is not well supported by MPPs and COWs. The single address space in a shared-variable model has the advantage of allowing global pointer operations, which is not required in most signal processing applications.
- *Communications Standard*: The applications should be coded using standard Fortran or C, plus a standard message passing library such as MPI or PVM. The MPI standard is especially advantageous as it has been adopted by almost all existing scalable parallel computers. It provides all the main message passing functionalities required in signal processing applications.
- *Topology Independent*: For portability reasons, the code should be independent of any specific topology. A few years ago, many parallel algorithms were developed specifically for the hypercube topology, which has all but disappeared in current parallel systems.

Major Performance Attributes

Communication is expensive on all existing MPPs. As a matter of fact, a higher *computation-to-communication* ratio implies a higher speedup in an application program. For example, this ratio is 86 flop/byte in our APT benchmark and 254 flop/byte in our HO-PD benchmark. This leads to a measured 23 Gflop/s speed on the SP2 for the HO-PD code versus 9 Gflop/s speed for the APT code. This ratio can be increased by minimizing communication operations or by hiding communication latencies within computations via compiler optimization, data prefetching, or active message operations. Various latency avoidance, and reduction, and hiding techniques can be found in [1,26,27,30]. These techniques may demand algorithm redesign, scalability analysis, and special hardware/software support.

The primary reason that SP2 outperforms the others is attributed to the use of POWER2 processors and a good compiler. Among the high-end microprocessors we have surveyed in Table 3, we feel that the Alpha 21164A (or the future 21264), UltraSPARC II, and MIPS R10000 have the highest potential to deliver a floating-point speed exceeding 500 Mflop/s in the next few years. With a clock rate approaching 500 MHz and continuing advances in compiler technology, a superscalar microprocessor with multiple floating-point units has the potential to achieve 1 Gflop/ s speed by the turn of the century. Exceeding 1000 SPECint92 integer speed is also possible by then, based on the projections made by Digital, Sun Microsystems, and SGI/MIPS.

Future MPP Architecture

In Fig.8, we suggest a common architecture for future MPPs, DSM, and COWs. Such a computer consists of a number of *nodes*, which are interconnected by up to three *communica-*

Table 10: Comparison of MPPs and COWs for Adaptive Signal Processing

Application Attributes	Massively Parallel Processors (MPPs)	Clusters of Workstations (COWs)
Number of Nodes	Hundreds to thousands	Tens to hundreds
Reported Performance (Gflop/s)	Tens to hundreds	Less than ten
Task Granularity	Dedicated single-tasking per node	Multitasking or multiprocessing per node
Internode communication and security	Proprietary network and enclosed security	Often standard
Node Operating System	Homogeneous microkernel	Could be heterogeneous, often homogeneous; complete Unix
Strength and Potential	High throughput with higher memory and I/O bandwidth	Higher availability with easy access of large-scale database managers
Application Software	Signal processing libraries exist and portable	Untested for signal processing applications
Shortcomings and Open Problems	Expensive and lack of real-time OS support	Heavy communication overhead and lack of single system image.

tion networks. The node usually follows a *shell architecture* [40], where a custom-designed shell circuitry interfaces a commodity microprocessor to the rest of the node. In Cray terminology [1], the overall structure of a computer system as shown in Fig.8 is called the *macro-architecture*, while the shell and the processor is called the *micro-architecture*. A main advantage of this shell architecture is that when the processor is upgraded to the next generation or changed to a different architecture, only the shell (the micro-architecture) needs to be changed.

There is always a local memory module and a network interface circuitry (NIC) in each node. There is always cache memory available in each node. However, the cache is normally organized as a hierarchy. The *level-1 cache*, being the fastest and smallest, is on-chip with the microprocessor. A slower but much larger *level-2 cache* can be on-chip or off the chip, as seen in Table 3.

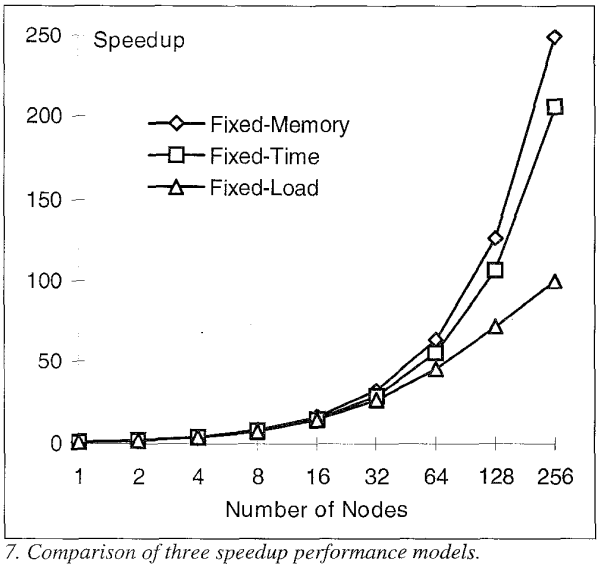
Unlike some existing MPPs, each node in Fig.8 has its own local disk and a complete multi-tasking Unix operating system, instead of just a microkernel. Having local disks facilitates local swapping, parallel I/O, and checkpointing. Using a full-fledged workstation Unix on each node allow multiple OS services to be performed simultaneously at local nodes. On some current MPPs, functions involving accessing disks or OS are routed to a server node or the host to be performed sequentially.

The native programming model for this architecture is Fortran or C plus message passing using MPI. This will yield high performance, portability and scalability. It is also desirable to provide some VLSI accelerators into the future MPPs for specific signal/image processing applications. For example, one can mix a programmable MPP with an embedded accelerator board for speeding up the computation of the adaptive weights in STAP radar signal processing.

The Low-Cost Network

Up to three communication networks are used in scalable parallel computer. An inexpensive commodity network, such as the Ethernet, can be quickly installed, using existing, well-debugged TCP/IP communication protocols. This low-cost network, although only supporting low speed communications, has several important benefits:

- It is very reliable and can serve as a backup when the other networks fail. The system can still run user applications, albeit at reduced communication capability.
- It is useful for system administration and maintenance, without disrupting user communications through the other networks.
- It can reduce system development time by taking advantage of *concurrent engineering*: While the other two networks and their communication interface/protocols are under development, we can use the Ethernet to design, debug, and test the rest of the system.
- It also provide an alternative means for user applications development: While the high-speed networks are used for production runs of applications, a user can test and debug the correctness of his code using the Ethernet.



7. Comparison of three speedup performance models.

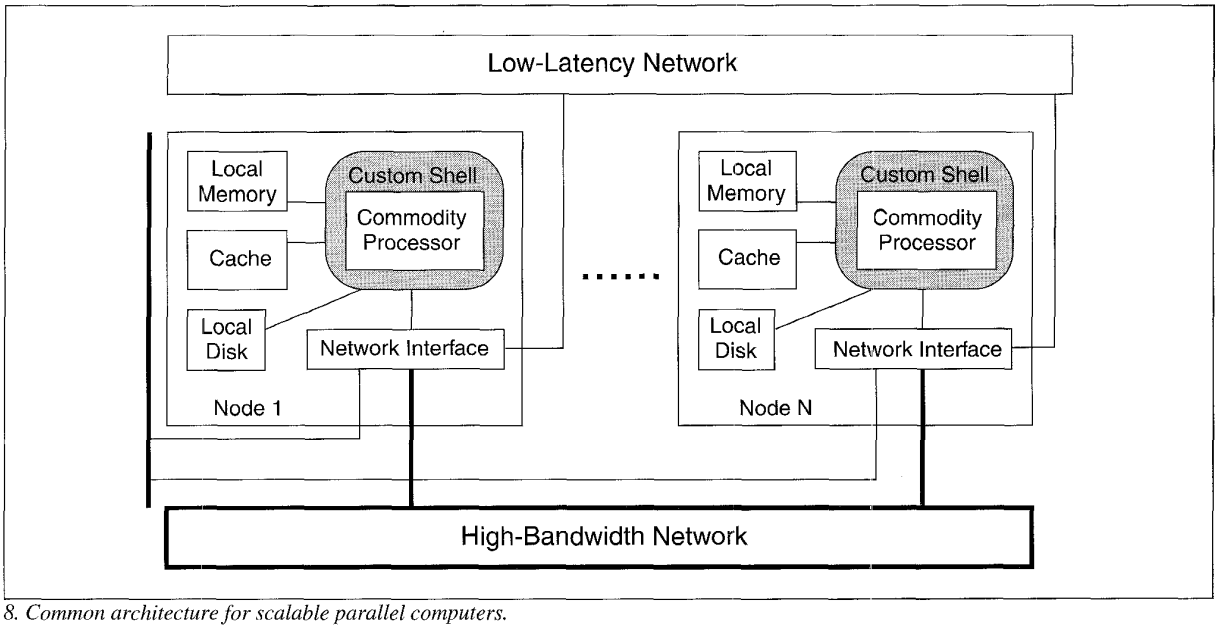
The High-Bandwidth Network

The high-bandwidth network is the backbone of a scalable computer, where most user communications take place. Examples include the 2-D mesh network of Paragon, the 3-D torus network of Cray T3D, the multi-stage *High-Performance Switch* (HPS) network of IBM SP2, and the fat-tree data network of CM-5. It is important for this network to have a high bandwidth, as well as short latency.

The Low-Latency Network

Some systems provide a third network to provide even lower latency to speed up communications of short messages. The *control network* of Thinking Machine CM-5 and the barrier/eureka hardware of Cray T3D are examples of low-latency networks. There are many operations important to signal processing applications which need to have small delay but not a lot of bandwidth, because the messages being transmitted are short. Three such operations are listed below:

- *Barrier*: This operation forces a process to wait until all processes reach a certain execution point. It may be needed in a parallel algorithm for radar target detection, where the processes must first detect all targets at a range gate before proceeding to the next farther range gate. The message length for such an operation is essentially zero.
- *Reduction*: This operation aggregate a value (e.g., a floating-point word) from each process and generate a global sum, maximum, etc. This is useful, e.g., in a parallel Gauss elimination or Householder transform program with pivoting, where one needs to find the maximal element of a matrix row or column. The message length could vary, but is normally one or two words.
- *Broadcasting of a short message*: Again, in a parallel Householder transform program, once the pivot element is found, it needs to be broadcast to all processes. The message length is the size of the pivot element, one or two words.



8. Common architecture for scalable parallel computers.

Comparison of MPPs and COWs

We feel that future MPPs and COWs are converging, once commodity Gigabit/s networks and distributed memory support become widely used. In Table 10, we provide a comparison of these two categories of scalable computers, based on today's technology. By 1996, the largest MPP will have 9000 processors approaching 1 Tflop/s performance; while any of the experimental COW system is still limited to less than 200 nodes with a potential 10 Gflop/s speed collectively.

The MPPs are pushing for finer-grain computations, while COWs are used to satisfy large-grain interactive or multitasking user applications. The COWs demand special security protection, since they are often exposed to the public communication networks; while the MPPs use non-standard, proprietary communication network with implicit security.

The MPPs emphasize high-throughput and higher I/O and memory bandwidth. The COW offers higher availability with easy access to large-scale database system. So far, some signal processing software libraries have been ported to most MPPs, while untested on COWs. Finally, we point out that MPPs are more expensive and lack of sound OS support for real-time signal processing, while most COWs can not support DSM or lack of single system image. This will limit the programmability and make it difficult to achieve a global efficiency in cluster resource utilization.

Extended Signal Processing Applications

So far, our MPP signal processing has been concentrated on STAP sensor data. The work can be extended to process SAR (synthetic aperture radar) sensor data. The same set of software tools, programming and runtime environments, and real-time OS kernel can be used for either STAP or SAR signal processing on the MPPs. The ultimate goal is to achieve automatic target recognition (ATR) or scene analysis in real time. To summarize, we list below the processing requirements for STAP/SAR/ATR applications on MPPs:

- The STAP/SAR/ATR source codes must be parallelized and made portable on commercial MPPs with a higher degree of interoperability.
- Parallel programming tools for efficient STAP/SAR program partitioning, communication optimization, and performance tuning need to be improved using visualization packages.
- Light-weighted OS kernel for real-time application on the target MPPs, DSMs, and COWs must be fully developed. Run-time software support for load balancing and insulating OS interferences are needed.
- Portable STAP/SAR/ATR benchmarks need to be developed for speedy multi-dimensional convolution, fast Fourier transforms, discrete cosine transform, wavelet transform, matrix-vector product, and matrix inversion operations.

Acknowledgment

This work was carried out by the Spark research team led by Professor Hwang at the University of Southern California. The Project was supported by a research subcontract from MIT Lincoln Laboratory to USC. The revision of the paper was done at the University of Hong Kong, subsequently. We appreciate all the research facilities provided by the HKU, USC, and MIT Lincoln Laboratory. In particular, we want to thank David Martinez, Robert Bond and Masahiro Arakawa of MIT Lincoln Laboratories for their help in this project. The assistance from Choming Wang and Mincheng Jin of USC, the User-Support Group at MHPCC, Richard Frost of SDSC, and the User-Support team of Cray Research made it possible for the team to develop the fully portable STAP benchmark suites on three different hardware platforms in a short time period.

Kai Hwang is Chair Professor of Computer Engineering at the University of Hong Kong, on leave from the University of Southern California. He can be contacted at e-mail: kaihwang@cs.hku.hk.

Zhiwei Xu is a Professor at the National Center for Intelligent Computing Systems, Chinese Academy of Sciences, Beijing, China. He can be contacted at e-mail: zxu@diana.usc.edu

References

1. D. Adams, *Cray T3D System Architecture Overview Manual*, Cray Research, Inc., September 1993. See also http//www.cray.com/PUBLIC/product-info/mpp/CRAY-T3D.html

2. R.C. Agarwal *et al.*, "High-Performance Implementations of the NAS Kernel benchmarks on the IBM SP2," *IBM System Journal, Vol. 34, No. 2, 1995, pp. 263-272.*

3. T. Agerwala, J. L. Martin, J. H. Mirza, D. C. Sadler, D. M. Dias, and M. Snir, "SP2 System Architecture," *IBM System Journal, Vol. 34, No. 2, 1995, pp. 152-184.*

4. G.M. Amdahl, "Validity of Single-Processor Approach to Achieving Large-Scale Computing capability," *Proc. AFIPS Conf.*, Reston, VA., 1967, 483-485.

5. T.E. Anderson, D.E. Culler, D.A. Patterson, et al, "A Case for NOW (Networks of Workstations)," *IEEE Micro,* February 1995, pp. 54-64.

6. ANSI Technical Committee X3H5, *Parallel Processing Model for High Level Programming Languages,* 1993, ftp://ftp.cs.orst.edu/standards/ANSI-X3H5/

7. Applied Parallel Research, "APR Product Information," 1995. http://www.infomall.org/ apri/prodinfo.html

8. M. Arakawa, Z. Xu, and K. Hwang, "User's Guide and Documentation of the Parallel HO-PD Benchmark on the IBM SP2," *CENG Technical Report 95-10,* University of Southern California, June 1995.

9. M. Arakawa, Z. Xu, and K. Hwang, "User's Guide and Documentation of the Parallel APT Benchmark on the IBM SP2," *CENG Technical Report 95-11,* University of Southern California, June 1995.

10. M. Arakawa, Z. Xu, and K. Hwang, "User's Guide and Documentation of the Parallel General Benchmark on the IBM SP2," *CENG Technical Report 95-12,* University of Southern California, June 1995.

11. D.H. Bailey *et al.*, "The NAS Parallel Benchmarks" and related performance results can be found at http://www.nas.nasa.gov/NAS/NPB/

12. G. Bell, "Why There Won't Be Apps: The Problem with MPPs," *IEEE Parallel and Distributed Technology,* Fall 1994, pp. 5-6.

13. R. Bond, "Measuring Performance and Scalability Using Extended Versions of the STAP Processor Benchmarks," *Technical Report,* MIT Lincoln Laboratories, December 1994.

14. Convex, *CONVEX Exemplar Programming Guide,* Order No. DSW-067, CONVEX Computer Corp., 1994. See also http://www.usc.edu/UCS/high_performance/sppdocs.html

15. Cornell Theory Center, IBM RS/6000 Scalable POWERparallel System (SP), 1995. http:// www.tc.cornell.edu/UserDoc/Hardware/SP/

16. DEC, *AdvantageCluster: Digital's UNIX Cluster,* September 1994.

17. J. H. Edmondson and P. Rubinfeld and R. Preston and V. Rajagopalan, "Superscalar Instruction Execution in the 21164 Alpha Microprocessor," *IEEE Micro,* April, 1995, pp. 33-43.

18. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Mancheck, V. Sunderam, *PVM: Parallel Virtual Machine - A User's Guide and Tutorial for Networked Parallel Computing,* MIT Press, Cambridge, MA, 1994. Also see http://www.epm.ornl.gov/pvm/pvm_home.html.

19. D. Greenley et. al. "UltraSPARC: The Next Generation Superscalar 64-bit SPARC," *Digest of Papers, Compcon,* Spring 1995, pp. 442—451.

20. W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message Passing Interface,* MIT Press, Cambridge, MA, 1994.

21. J.L. Gustafson, "Reevaluating Amdahl's Law," *Comm. ACM,* 31(5)(1988)532-533.

22. L. Gwennap, "Intel's P6 Uses Decoupled Superscalar Design," *Microprocessor Report,* February 1995, pp. 5-15.

23. L. Gwennap, "MIPS R10000 Uses Decoupled Architecture," *Microprocessor Report,* October 1994, pp. 18-22.

24. High Performance Fortran Forum, *High Performance Fortran Language Specification,* Version 1.1, November 10, 1994, http://www.erc.msstate.edu/hpff/hpf-report/hpf-report/hpf-re port.html.

25. R. W. Hockney, "The Communication Challenge for MPP: Intel Paragon and Meiko CS-2," *Parallel Computing,* Vol. 20, 1994, pp. 389-398.

26. K. Hwang, *Advanced Computer Architecture: Parallelism, Scalability, and Programmability,* McGraw-Hill, New York, 1993.

27. K. Hwang and Z. Xu, *Scalable Parallel Computers: Architecture and Programming,* McGraw-Hill, New York, to appear 1997.

28. K. Hwang, Z. Xu, and M. Arakawa, "Benchmark Evaluation of the IBM SP2 for Parallel Signal Processing," *IEEE Transactions on Parallel and Distributed Systems,* May 1996.

29. Kuck and Associates, *The KAP Preprocessor,* http://www.kai.com/kap/kap_what_is.html.

30. D.E. Lenoski and W.-D. Weber, *Scalable Shared-Memory Multiprocessing,* Morgan Kaufmann, San Francisco, CA, 1995.

31. D. Levitan and T. Thomas and P. Tu, "The PowerPC 620 Microprocessor: A High Performance Superscalar RISC Microprocessor," *Digest of Papers, Compcon95,* Spring 1995, pp. 285-291.

32. T.G. Mattson, D. Scott, and S. Wheat, "A TeraFLOP Supercomputer in 1996: The ASCI TFLOPS System," *Proc. of the 6th Int'l Parallel Processing Symp.,* 1996.

33. MHPCC, *MHPCC 400-Node SP2 Environment,* Maui High-Performance Computing Center, Maui, HI, October 1994.

34. MIT/LL, "STAP Processor Benchmarks," MIT Lincoln Laboratories, Lexington, MA, February 28, 1994.

35. NCSA, "Programming on the Power Challenge," National Center for Supercomputing Applications, http://www.ncsa.uiuc.edu/Pubs/UserGuides/Power/Power5Prog_1. html

36. G.F. Pfister, *In Search of Clusters,* Prentice Hall PTR, Upper Saddle River, NJ, 1995.

37. J. Rattner, "Desktops and TeraFLOP: A New Mainstream for Scalable Computing," *IEEE Parallel and Distributed Technology,* August 1993, pp. 5-6.

38. SDSC, SDSC's Intel Paragon, San Diego Supercomputer Center, http://www.sdsc.edu/Services/Consult/Paragon/paragon.html

39. SGI, IRIS Power C User's Guide, Silicon Graphics, Inc., 1989.

40. J. Smith, "Using Standard Microprocessors in MPPs," presentation at *Int'l. Symp. on Computer Architecture,* 1992.

41. X.H. Sun, and L. Ni, "Scalable Problems and Memory-Bounded Speedup," *Journal of Parallel and Distributed Computing,* Vol. 19, pp.27-37, Sept. 1993.

42. H.C. Torng and S. Vassiliadis, *Instruction-Level Parallel Processors,* IEEE Computer Society Press, 1995.

43. Z. Xu and K. Hwang, "Modeling Communication Overhead: MPI and MPL Performance on the IBM SP2 Multicomputer," *IEEE Parallel and Distributed Technology,* March 1996.

44. Z. Xu and K. Hwang, "Early Prediction of MPP Performance: SP2, T3D and Paragon Experiences," *Parallel Computing,* accepted to appear in 1996.