Benchmark Evalution of the IBM SP2 for Parallel Signal Processing

Kai Hwang, Zhiwei Xu and Masahiro Arakawa

CENG Technical Report 95-13

Department of Electrical Engineering - Systems
University of Southern
Los Angeles, California 90089-2562
(213)740-4470

Benchmark Evaluation of the IBM SP2 for Parallel Signal Processing[1]

Kai Hwang, Zhiwei Xu, and Masahiro Arakawa
Scalable Parallel Computing Laboratory
University of Southern California
Los Angeles, CA 90089-2562

Abstract:

We report the performance results of a benchmark evaluation of the SP2 system, a *massively parallel processor* (MPP) released by IBM in 1994. The STAP benchmark suite, consisting of three sequential C programs, was originally written by MIT Lincoln Laboratories for *space-time adaptive processing* (STAP) of radar sensor data in real-time. At USC, we parallelized the C code and ported them to the 400-node SP2 system at the Maui High-Performance Computing Center. With 256 nodes, the Maui SP2 demonstrated a performance of 23 GFLOPS, corresponding to a system efficiency of 34%.

This paper evaluates the SP2 hardware platform, the AIX parallel programming environment, the IBM message-passing library (MPL), and the STAP benchmark performance. Only coarse-grain parallelism was exploited due to the high communication overhead on the SP2. We propose a new parallelization scheme for programming MPPs. The scheme is based on early performance prediction before entering the tedious coding and debugging stages of the software cycle. This can result in a significant reduction in the software development cost by optimizing the application at algorithm design time.

Parallel STAP benchmark structures are illustrated with domain decomposition, efficient mapping of the distributed programs, and collective communication operations. We present the SP2 performance in terms of execution times, sustained GFLOPS rates, speedup over a single SP2 node, and overall system efficiency. Communication overheads are characterized for the SP2. Scalability analysis is provided to show the performance growth as the machine and problem sizes increase. Lessons learned from these benchmark experiments are discussed in the context of supporting dedicated, real-time applications on scalable MPPs.

Keywords: Message passing, data parallelism, massively parallel processors, adaptive sensor array processing, scalability, programmability, performance evaluation, STAP benchmarks, real-time applications.

---

[1]This work was supported by a research subcontract to USC from the MIT Lincoln Laboratories.

Table of Contents

1. Introduction

The IBM SP2 is a *massively parallel processor* (MPP) with distributed memories which are not shared. Interprocessor communication is achieved by explicit message passing. In October 1994, a 400-node SP2 configuration was installed at the Maui High-Performance Computing Center (MHPCC) [16]. Our USC research team was among the first few user groups to test the performance of the SP2 on up to 256 nodes.

The STAP (*space-time adaptive processing*) benchmark [4, 17] consists of three programs: APT, HO-PD, and GEN, originally developed by MIT/LL in sequential C code for adaptive radar signal processing on workstations. The benchmark requires performing billions of flops (*floating point operations*) over hundreds of megabytes of input data in less than half of a second. Clearly this calls for the use of an MPP system. Our benchmark experiments are aimed at answering the following four questions:

(1) What are the characteristics of STAP applications, such as *degree of parallelism, workload, grain size, communication patterns,* and *computation/communication ratio*?
(2) How do we parallelize the STAP programs on a message-passing multicomputer? What paradigm is most suitable for parallel signal processing?
(3) What are the achievable STAP *execution time, sustained performance, speedup*, and *system efficiency*?
(4) Are the parallel STAP programs scalable with respect to increasing radar and computer parameters?

This paper provides some answers to these questions, based on the benchmark results obtained. We first describe the IBM SP2 hardware platform and the software environment used in the benchmark runs. We provide a performance characterization of the parallel computing and communications capabilities of the SP2. Then we report the measured SP2 performance results. This paper presents a comprehensive report of the SP2 benchmark results and extends from our previous work reported in [13, 23, 24]. Only preliminary performance results were reported in the *Supercomputing Conference '95* paper [13].

These performance results reveal the SP2/STAP scalability requirements over increasing radar parameters and machine size. To our knowledge, our project is the first independent, comprehensive evaluation of the IBM SP2 for real-time applications. Other studies of the SP2 performance were reported in NAS benchmark experiments [2] and in Dongarra's report [6].

2. The SP2 System in Maui

In this section, we describe the hardware platform, the software environment, the benchmark conditions, and the execution procedures used in the STAP benchmark experiments on the Maui SP2.

2.1 The SP2 Hardware Platform

As shown in Figure 1, the Maui SP2 consists of 400 processing nodes [16]. During our dedicated time slots, 256 nodes were allocated to us for exclusive execution of the STAP programs. The remaining nodes were still available to run other applications. Each node is equipped with a 66 MHz POWER2 processor and 64-256 MB of local memory. Each POWER2 superscalar processor has a peak performance of 266 MFLOPS. The 400 nodes are interconnected by a 400-way multistage network called the *High-Performance Switch* (HPS), along with some internal and external Ethernet connections [21].

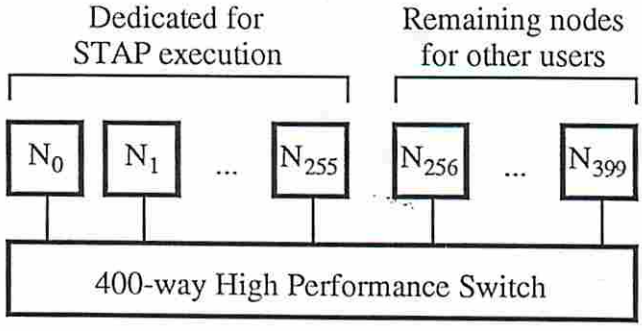


Figure 1 The SP2 configuration used for dedicated execution of STAP programs at the MHPCC

The HPS provides four distinct connecting paths between any pair of nodes in the system. Although we were given dedicated use of 256 nodes, we shared the HPS with other users using the remaining nodes in the open resource pool.

2.2 AIX Parallel Programming Environment

Each node runs the IBM AIX operating system, the same UNIX-based OS used on IBM RISC 6000 or PowerPC workstations. A *parallel programming environment* [14] was developed by IBM to support message-passing multicomputing on the SP2. Important features of the AIX-based software environment are summarized in Table 1. Both SPMD (*single program and multiple data streams*) and MPMD (*multiple programs and multiple data streams*) execution modes [12] are supported in the AIX parallel programming environment.

Table 1 AIX Parallel Programming Environment for the IBM SP2/MPP

Programming Model	SPMD or MPMD with message passing communication and static task creation
Operating System	Multi-user, multitasking UNIX-based AIX supporting: single user mode (dedicated use) batch mode (Loadleveler) time-sharing mode
Software Tools	Proprietary tools: [14] POE and MPL ESSL libraries VT for visualization Public-domain tools: PVM [8] and MPI [7, 9, 18, 23] Third party tools: Linda, Express [20], FORGE [12]
Languages	C, C++, Fortran, HPF
Databases	DB2, Oracle, Sybase

The AIX/OS supports a dedicated single-user mode and a time-sharing mode for multiple users. Load balancing between batch-mode users is handled by the Loadleveler. The system supports the C, C++, Fortran, and HPF languages and several database systems. IBM proprietary tools which were developed are the *parallel operating environment* (POE), the *message-passing library* (MPL), ESSL scientific libraries, and the VT visualization tool. Public-domain tools, such as PVM and MPI, are also implemented on the SP2, along with third party software tools such as Linda, Express [20], and FORGE [12].

All internode communication in the SP2 is handled by using MPL commands [14, 23]. Three types of communication operations are provided on the SP2: *point-to-point communication, collective communication, and aggregated computation*. In point-to-point communication, only two nodes, the sender and the receiver, are involved. The IBM SP2 uses the HPS to speed up all message-passing operations. All nodes are an equal distance from one another, and the concept of neighboring nodes or remote nodes does not exist in the SP2. A consequence of this fact is that the time for a point-to-point communication varies only with the message length, and not with the physical location of the nodes involved.

In an aggregated or global computation, tasks in a group synchronize with one another or aggregate partial results. The time for such an operation is a function of the group size, but not of the message length, as the message length is fixed, i.e., $t = f(n)$, where n is the number of nodes involved. For example, in a *barrier* operation, a group of tasks synchronize with one another; i.e., they wait until all tasks execute their respective barrier operation. In a *reduction* operation, a group of tasks aggregate partial results, one

from each task, into a final result. Typical examples of reductions are summing or finding the maximum of n values, one from each of the n tasks. In a *parallel prefix* operation, also known as a *scan*, a group of tasks aggregate n partial results, one from each of the n tasks, into n final results.

In a *collective communication* operation, tasks in a group send messages to one another, and the time is a function of both the message length, m, and the number of nodes, n, i.e., $t = f(n,m)$. In a *broadcast* operation, a single node sends an m-byte message to all n nodes. In a *gather* operation, a single node receives an m-byte message from each of the n nodes, so in the end mn bytes are received by that node. In a *scatter* operation, a single node sends a distinct m-byte message to each of the n nodes, so in the end mn bytes are sent by that node. In a *total exchange* operation, every one of the n nodes sends a distinct m-byte message to each of the n nodes, so in the end mn^2 bytes are communicated. In a *circular shift* operation, node i sends an m-byte message to node (i+1) modulo n.

2.3 Benchmark Conditions and Procedures

The following testing conditions were observed on the SP2 in order to generate consistent timing results in our benchmark experiments:

(1) *Use the best available resources.* Dedicated nodes were used to minimize interference from other user tasks. All communication was done through the HPS using the User-Space protocol.

(2) *Use the best compiler options.* We compiled our programs using

```
mpcc -O3 -qarch=pwr2 code.c     for parallel code and
cc -O3 -qarch=pwr2 code.c       for sequential code.
```

The `-O3` option provides the highest optimization level in the IBM C compiler. The `-qarch=pwr2` option instructs the compiler to generate POWER2-efficient code.

(3) *Use high-level, portable C code.* Except for the standard I/O and timer libraries, and MPL, no other libraries or assembly code were used.

(4) *Use single-precision floating-point numbers.* This condition was set so that the parallel benchmark programs could be run on a single node as a baseline. If we were to use double-precision, we would need 400 MB of memory per node, exceeding what is available on any one SP2 node.

(5) *Measure both CPU and wall-clock times.* We measured the execution time of the benchmark programs using both `times()` and `gettimeofday()`. This guards against potential time code errors. We used the wall clock time generated by `gettimeofday` in analyzing the benchmark timing results, because it has a higher resolution of 1 microsecond. Each program was executed at least ten times, and the best time result was used in our analysis, because it corresponds to the run experiencing the least interference from the operating system and other users.

3. Internode Communications Performance

The SP2 message-passing capability and communications overheads are evaluated in this section. First, we explain how to characterize the performance of the MPL. Then, we report the measured communication overheads using the MPL commands for short and long messages.

3.1 Message-Passing Performance Metrics

Hockney [11] characterized the *communication time* (in μs) of a point-to-point communication operation as follows:

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

where m denotes the message length in bytes. The parameter r_∞ is called the *asymptotic bandwidth* in MB/s, and is the maximal bandwidth achievable when the message length approaches infinity. The parameter t_0 is called the *latency* (or *startup time*), and is the time needed to send a 0-byte message. Two additional parameters were derived. The *half-peak length*, denoted by $m_{1/2}$ bytes, is the message length required to achieve half of the asymptotic bandwidth. The *specific performance*, denoted by π_0 MB/s, indicates the bandwidth for short messages. Only two of these four parameters are independent. The other two can be derived using the following relations:

$$t_0 = \frac{m_{1/2}}{r_\infty} = \frac{1}{\pi_0} \tag{2}$$

Hockney's model applies only for point-to-point communication operations. It needs to be generalized for collective communication and aggregated computation operations. Xu and Hwang [23] have obtained the following generalized timing model:

$$t = t_0(n) + \frac{m}{r_\infty(n)} \tag{3}$$

where $t_0(n)$ and $r_\infty(n)$ can be linear or non-linear functions of n. For aggregated computation operations, we only need to focus on the latency. The asymptotic bandwidth can be viewed as a small constant. These overhead expressions are different for different MPPs. However, they should follow the form given in Equation 3.

Timing expressions for $t_0(n)$ and $r_\infty(n)$ were obtained for some MPL message-passing operations on the SP2. Details on how to derive these expressions are treated in [23], where the MPI performance is also compared to the native IBM MPL operations.

Table 2 Collective Communication Time and Aggregated Bandwidth on the IBM SP2

MPL Command	Communication Time in μs (Equation 3)	Aggregated Bandwidth in MB/s (R_∞)
Broadcast	$52\log n+(0.029\log n)m$	$40n/\log n$
Gather/Scatter	$(45\log n+10)+(0.04n-0.04)m$	$n/(0.025n+0.03)$
Total Exchange	$80\log n+(0.03n^{1.29})m$	$33.3n^{0.71}$
Reduction	$20\log n+23$	N/A
Circular Shift	$6\log n+60+(0.003\log n+0.04)m$	$n/(0.003\log n+0.04)$

The *aggregated asymptotic bandwidth*, denoted by R_∞ MB/s, is defined as the ratio of the total number of bytes transmitted in a communication operation to the time needed to execute the operation, as the message size m approaches infinity. This metric reflects the aggregated communication capability of the HPS. For a point-to-point communication, $R_\infty = r_\infty$. For a *total exchange*, $R_\infty = n^2 r_\infty$. For other collective communications (*broadcast, gather, scatter,* and *shift*), $R_\infty = nr_\infty$. From Table 2, the greatest aggregated bandwidth is achieved with the *circular shift* operation, reaching 4 GB/s for 256 nodes.

In choosing communication operations, we should always apply the most powerful, high-level operations first, instead of using a sequence of low-level primitives. For instance, to implement the *total exchange* operation, we should use the MPL operation `mpc_index`. We could use a sequence of *scatters* or even *send/receives* to implement the *total exchange*. Based on Table 2, a collective communication using a sequence of low-level primitives is 2 to 50 times slower on the SP2 than using just one high-level MPL function.

3.2 MPL Communication Overheads

From the above analysis, we can see that the MPL communication overhead depends not only on the message size m, but also on the number of nodes n involved in

the collective communications or aggregated computations. In Table 3, we list the measured overheads on the SP2 for a short message of $m = 4$ bytes and a long message of $m = 64$ KB, assuming that $n = 128$ nodes are involved in the collective or aggregated operations.

Table 3 Measured Communication Overheads on a 128-node SP2 for Short (4 B) and Long (64 KB) Messages

Message Passing Library	Operation Type	Message Length	Communication Overhead in μs
Point-to-Point	One Way	4 B	49
		64 KB	2,235
Collective Communication	Broadcast	4 B	120
		64 KB	14,965
	Gather/Scatter	4 B	133
		64 KB	252,199
	Total Exchange	4 B	949
		64 KB	966,173
	Circular Shift	4 B	105
		64 KB	5,642
Aggregated Computation	Barrier	N/A	678
	Reduction	4 B	153
	Parallel Prefix (Scan)	4 B	577

The above results show that the overhead for a single communication operation on SP2 can be long enough to perform thousands or more floating-point operations. Therefore, only coarse-grain parallelism should be exploited on the SP2. For collective communication or aggregated computation operations, the derived overhead formulae can identify where the bottleneck lies. Modeling communication delays, overhead, and bandwidth becomes critically important for designers and programmers of MPPs.

4. Sequential STAP Performance

In this section, we introduce the STAP benchmark suite and characterize its computational workload. Then we present the sequential STAP benchmark performance measured on a single SP2 node.

4.1 The STAP Benchmark Suite

The STAP benchmark consists of three radar signal processing programs: *Adaptive Processing Testbed* (APT), *High-Order Post-Doppler* (HO-PD), and *General* (GEN), totaling more than 4,000 lines of C code. The APT and HO-PD are both written to test the STAP algorithms [4, 5, 22] for adaptive radar signal processing. Both programs start with *Doppler processing* (DP), in which the program performs a large number of

FFT computations. Both end with *target detection*. The difference between the two programs is that APT performs a *Householder transform* (HT) to generated jammer-nulled beams and then performs beamforming to null the clutters, whereas, in the HO-PD program, the two adaptive beamforming steps are combined into one step.

Component algorithms of all three STAP benchmarks are identified in Table 4. The GEN program consists of 4 subroutines to perform sorting, FFT, vector multiply, and linear algebra. These are the kernel routines often used in signal processing applications. Details of these substeps can be found in [4, 17].

Table 4 Computational Workload of the STAP Benchmark Programs Based on a Nominal Radar Signal Data Set

Benchmark Program	Component Algorithms	Flop Count in Mflop	Total Flop Count
APT	Doppler Processing (DP)	83.89	1.45 Gflop
	Householder Transform (HT)	2.88	
	Beamforming (BF)	1,313.67	
	Target Detection (TD)	46.45	
HO-PD	Doppler Processing (DP)	220.00	12.85 Gflop
	Adaptive Beamforming (BF)	12,618.00	
	Target Detection (TD)	14.00	
GEN	Sorting (SORT)	1,183.00	5.30 Gflop
	FFT	1,909.00	
	Vector Multiply (VEC)	603.00	
	Linear Algebra (LA)	1,603.00	

4.2 STAP Workload Characterization

Workload measures the amount of computational work performed in an application. For scientific computing and signal processing applications, where numerical calculations dominate, a natural metric is the number of *floating point operations* that need to be executed. This metric of workload has a unit of *Millions of flops* (Mflop) or *Billions of flops* (Gflop). For instance, an N-point FFT has a workload of $w = 5N \log N$ flop. This should be differentiated from the unit of computational speed, which is *Millions of flops per second*, denoted by MFLOPS.

The STAP workload is summarized in Table 4. The total workload for each program is the sum of the flop counts of its component algorithms. The workload calculation is directly related to the application specifications and the underlying algorithms to be implemented.

The STAP computational workload is related to the size of the input data set. Depending on the radar parameters used, the problem size could vary dramatically. For

example, the HO-PD program has a workload of 12.85 Gflop for a nominal data set. The workload may grow to 33.4 Tflop for a maximal data set [4].

4.3 Sequential STAP Benchmark Results

We executed the sequential version of the STAP benchmark programs with nominal data sets on a single SP2 node with 256 MB of main memory and 256 KB of cache. The results are shown in Figure 2.

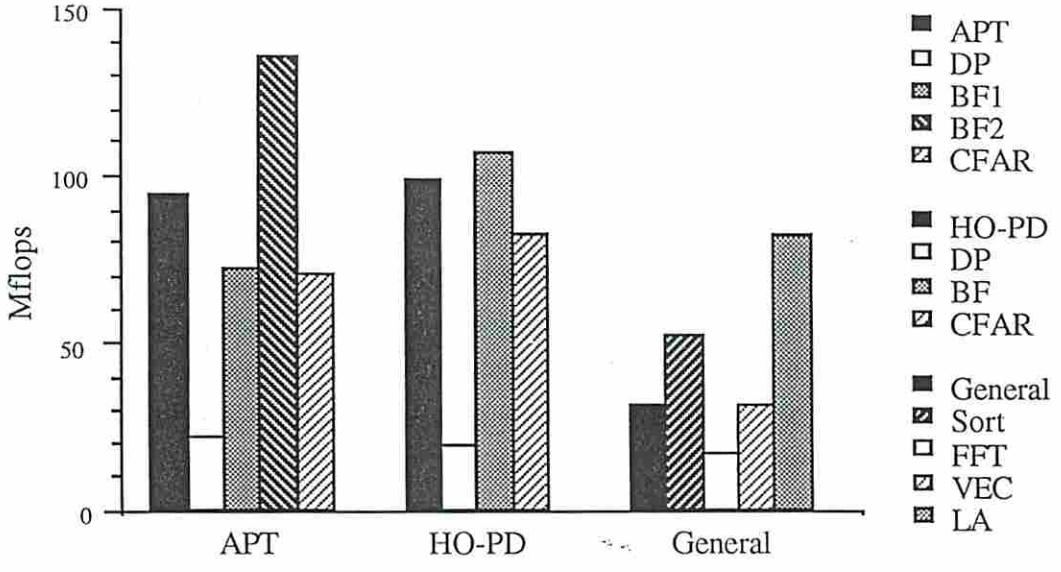


Figure 2 Sequential STAP performance on a single SP2 node

The component algorithms show a large variation in the MFLOPS rate. The BF2 step in APT showed a 136 MFLOPS performance. The FFT in the GEN program showed only a 24 MFLOPS performance out of 266 MFLOPS, the peak speed of a single POWER2 node. The following observations are based on our experience in the sequential benchmark experiments on a single node on the SP2:

(1) It is important to exploit data locality to take advantage of the data cache in the local processor. For instance, a loop interchange designed to exploit greater data locality results in a faster matrix multiplication. The following code segment achieved a 6 MFLOPS performance on a single SP2 node:

```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    for (k = 0; k < N; k++)
      c[i][j] += a[i][k] * b[k][j];
```

With the loop interchange shown below, a 17-fold speed improvement was observed:

```
for (i = 0; i < N; i++)
  for (k = 0; k < N; k++)
    for (j = 0; j < N; j++)
     c[i][j] += a[i][k] * b[k][j];
```

(2) We have found that the choice of data structure can also greatly impact performance. For instance, arrays of complex numbers are frequently used in signal processing applications. Two popular methods to represent a complex array follow:

```
/* Method 1: Array of structures */
typedef struct { double real, imag; } COMPLEX;
COMPLEX data [N1][N2][N3];

/* Method 2: Separate arrays */
double data_real [N1][N2][N3],
       data_imag [N1][N2][N3];
```

Our experience has shown that Method 1 should always be used, because it has a much smaller cache miss ratio. In our application, Method 2 results in a cache miss ratio as high as 33%. Changing to Method 1 reduces the cache miss ratio to as low as 3%.

5. Parallelization of STAP Programs

During the benchmark study, we developed a methodology for parallelizing sequential programs. The method not only helps the development of parallel programs, but also increases software development productivity. For instance, the APT benchmark was initially parallelized in an ad hoc way, taking about two man-months. In contrast, the HO-PD program was parallelized using the new approach, reducing the development time to 0.7 man-months. Although this method was specially developed for the STAP benchmark suite with respect to the IBM SP2 architecture, it is applicable to other MPP architectures and applications as well.

5.1 Application Software Development

This performance-driven approach to developing application software is illustrated in Figure 3. Based on the application specifications, an initial algorithm design and workload characterization can be worked out. Based on the MPP hardware platform and software environment, the communication overhead can be estimated. An *early prediction scheme* was proposed in a companion paper [24]. The idea is to use quantified workload and overhead information to predict the performance of the MPP as early as possible. If the predicted performance is unsatisfactory, the algorithm must be redesigned. Otherwise, we proceed to the coding and debugging stages. The corrected parallel code is

then ported onto the MPP and its performance measured. If the performance does not meet the goals, the entire software development cycle must be repeated.

The early prediction scheme can result in a significant savings in the software development cost by optimizing the parallelization strategy at algorithm design time, before entering the tedious coding and debugging stages of the software cycle. The accuracy of the early MPP performance prediction depends on how accurately the workload and overhead are characterized. Two sources of overhead, namely the *parallelism overhead* and the *interaction overhead*, are identified. The parallelism overhead covers the extra time needed for concurrent process creation, context switching, process inquiry, and grouping for collective communications. The interaction overhead includes the extra time needed to perform synchronization, aggregation, and communications among the processing nodes. Workload characterization is essentially based on the expected sequential performance. Readers are referred to [23] to find methods for modeling communication overheads, and to [24] for procedures to carry out the early performance prediction. Details of these methods are not repeated here.

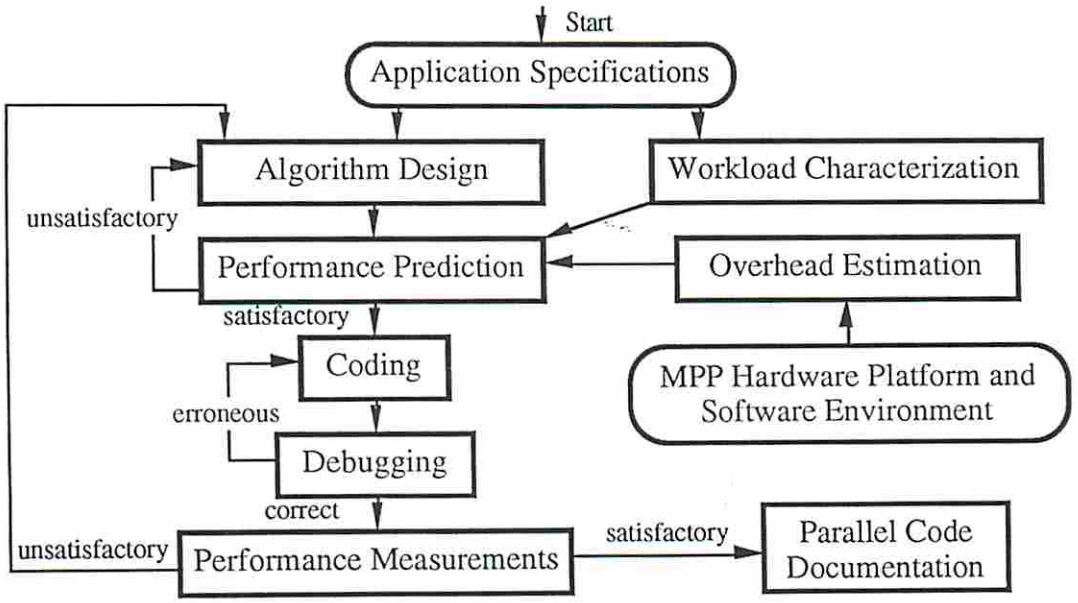


Figure 3 A performance-driven approach to parallel program development for an MPP system

To select a suitable strategy for parallelizing a sequential program, decisions should be made on the following issues: *granularity*, *homogeneity*, *data distribution*, and *algorithmic paradigm*. We define *granularity* (or *grain size*) as the number of computation operations (flops) performed by one task between two communication operations. In general, the granularity is inversely proportional to the *degree of parallelism* (DOP) exploited. Granularity should be chosen based on three factors : (1) the application performance requirement, (2) the input data size, and (3) the computation/communication ratio for the MPP.

For the STAP benchmarks, the sustained performance requirement was set to 10 to 25 GFLOPS [17]. This calls for a DOP of 128 or more, according to Figure 2. If only one GFLOPS were required, we need only to exploit a DOP of 16, and thus a coarser granularity. As the speed requirement increases, finer granularity must be exploited. The input data parameters also affect the granularity. For instance, the *pulse repetition interval* PRI = 256 in APT nominally. If the PRI parameter is reduced to 64, a coarse grain approach will not provide enough parallelism to achieve 10 GFLOPS.

A single SP2 communication operation has a time delay equivalent to the execution time of thousands of flops. Thus only coarse-grain parallelism should be exploited on the SP2. That is, thousands or more flops should be performed by a task on a node before a communication operation is performed. Finer grain parallelism can be exploited on machines like the Cray T3D, which has a smaller per-node flops rate but faster communication support.

5.2 Exploiting SPMD Data Parallelism

Should we exploit data parallelism (SPMD) or control parallelism (MPMD) on the SP2? The SP2 supports both execution modes. The programmer must make a decision on which mode to use, based on the application characteristics. It is easier to develop and execute an SPMD program than an MPMD one. For the STAP benchmark, the SPMD mode was found to be sufficient, where multiple nodes execute the same program over different chunks of the data domain.

How should the data domain be partitioned and distributed among multiple tasks? The answer comes from dependence analysis, and the minimization of the communication overhead. In the HO-PD benchmark, dependence analysis reveals that the DP step should be parallelized along the *range gate* (RNG) and the *channel element* (EL) dimensions. When n tasks are executed in parallel, each executes (RNG×EL)/n FFTs without any communication. Similarly, the BF and the TD steps should be parallelized along the PRI dimension [4]. This strategy requires a *total exchange* after the DP step.

In Figure 4, we show the data distribution before and after the *total exchange* operation in the HO-PD program. Before the *total exchange,* as shown on the left, each node is allocated with a data subcube over all the 128 PRIs, all 48 ELs, and a chunk of x RNGs, where $x = 1024/n$ and n is the machine size. After the *total exchange*, parallelism is exploited along the PRI dimension, as shown on the right.

5.3 Selection of Parallelization Paradigm

For all of the STAP programs, a simple *compute-interact* paradigm is sufficient. Under this paradigm, a parallel program is divided into a sequence of alternate *computation* and *interaction* phases. During a computation phase, multiple node tasks

perform independent computations in parallel. These tasks then interact in the following phase. An interaction could be one or more communication (e.g., *total exchange*), synchronization (i.e., *barrier*), or aggregated computation (i.e., *reduction* or *parallel prefix*) operations. For instance, the HO-PD program can be executed at each node with the same sequence of four phases as illustrated in Figure 5.

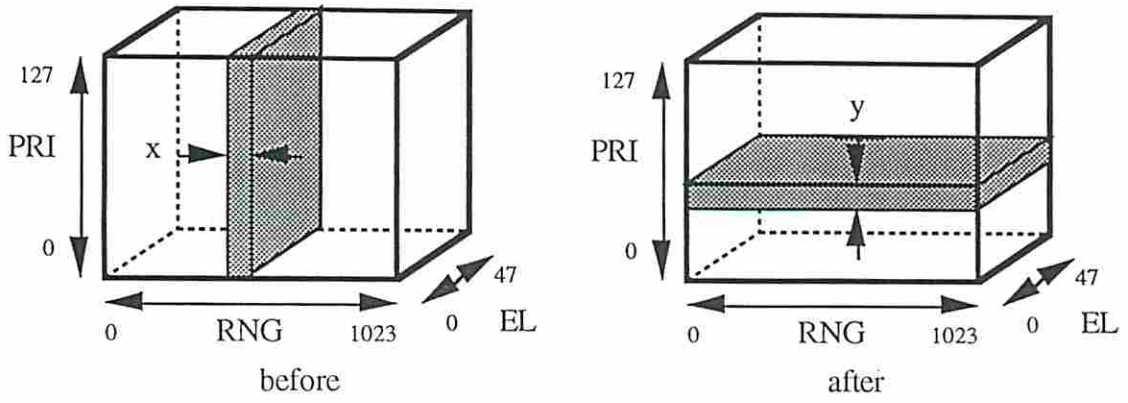


Figure 4 Data distribution before and after the total exchange operations in the HO-PD program, where x = 1024/*n*, y = 128/*n*, and *n* is the SP2 machine size

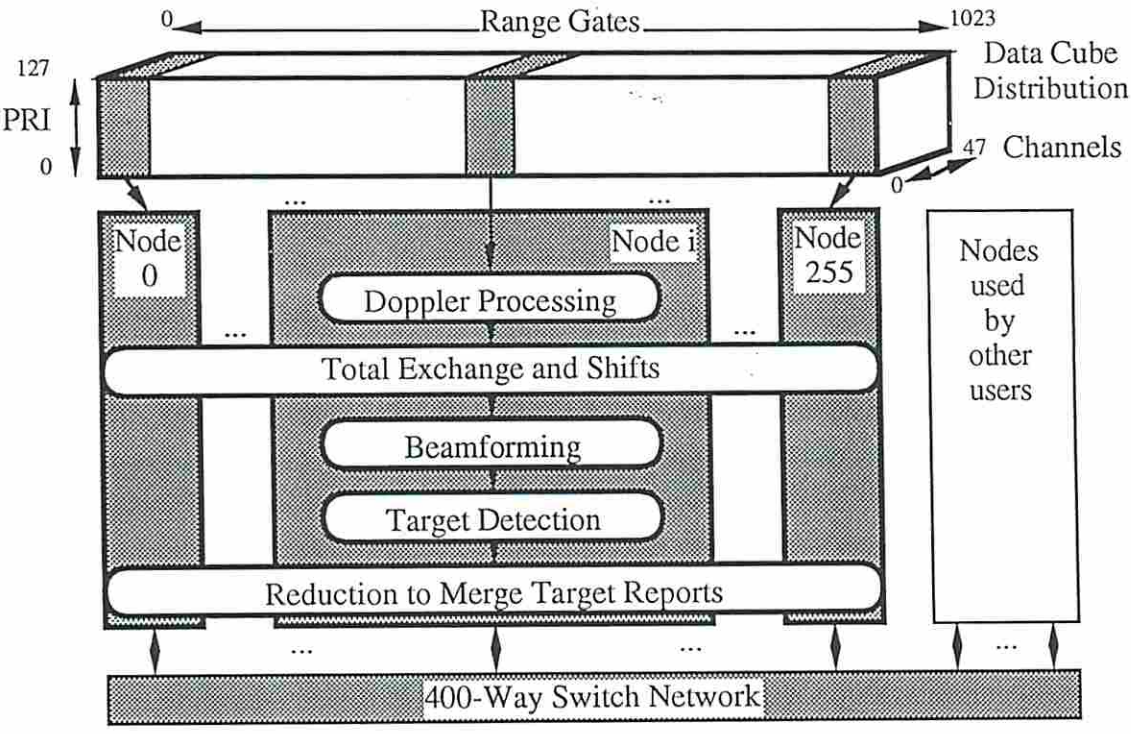


Figure 5 Mapping of the parallel HO-PD program on a 256-node SP2 system, using the compute-and-reduce paradigm

Compute	Doppler Processing (DP)
Interact	Total Exchange and Shifts
Compute	Beamforming (BF) followed by Target Detection (TD)
Interact	Reduction to find the complete target report

There are many other parallel programming paradigms : *compute-aggregate-broadcast*, *synchronous iteration*, *asynchronous iteration*, *worker-pool*, *agenda parallelism*, and *result parallelism* [3, 10]. It is not trivial to decide which paradigm should be used, especially when complex interactions are involved.

We illustrate below how to select the best paradigm by showing how to parallelize the following target report program, which must be used in the APT and HO-PD benchmarks.

```
q = 0;
for (k = 0; k < RNG; k++)
  for (i = 0; i < PRI; i++)
    for (j = 0; j < BEAM; j++)
      if (IsTarget (in detect_cube[i][j][k]))
        {
          target_report[q].pri = i;
          target_report[q].beam = j;
          target_report[q].rng = k;
          target_report[q].power
            = detect_cube[i][j][k].real;
          q = q + 1;
          if (q > 25) goto finished;
        }
finished:
```

`RNG`, `PRI`, and `BEAM` are program constants determined by radar parameters, where `BEAM` stands for the number of beam directions generated. This code tests each element of the three-dimensional data array `detect_cube`. The `power` represents the intensity of the radar signal reflected from the target. When `power` exceeds a specified threshold, a target is detected. The detected target is put into a 25-element `target_report` array. The program terminates when 25 targets are found. This sequential code is difficult to parallelize for three reasons :

(1) Close-range targets should be reported first, which apparently forces the outermost loop to be executed sequentially.

(2) At most 25 targets should be reported. Thus, if each of the 128 node tasks finds a target, we must ensure that only the 25 closest ranged targets are reported.

(3) The data structure `target_report` must be accessed atomically or mutually exclusively.

In [3], both *synchronous iteration* and *asynchronous iteration* paradigms are proposed to design parallel and distributed algorithms. We use the early-prediction scheme to show that these two paradigms are not suitable for the target reporting problem on the SP2. Instead, we propose a *gather-and-sequential-compute* paradigm and a *compute-and-reduce* paradigm. We show below that the latter is a better choice, based on predicted performance results. With static tasking on each node, we use the term *task* to refer to the node program being executed on the node at a given time.

Paradigm 1. Synchronous Iteration : n tasks test the *detect-cube* in parallel. When a target is found, it is put into the target report atomically. To ensure that close range targets are reported first, all tasks perform a barrier for each range gate. This paradigm is not suitable for the SP2, because the *barrier* is a very expensive operation, each taking $94\log n + 10$ μs, which is about 700 μs per *barrier* over 128 nodes. For the HO-PD benchmark, RNG = 1024, which gives a total barrier time of about $700 \times 1024 = 0.7$ seconds, far exceeding the 0.08 seconds time in Paradigm 3. However, this paradigm may be viable on MPP systems such as the CM-5 or the Cray T3D [1], where a barrier takes much less time.

Paradigm 2. Asynchronous Iteration : To eliminate the barrier overhead, we have developed an asynchronous iteration solution based on a priority queue scheme. When a node task discover a new target, it puts it into a priority queue of length 25. When the queue is full, if the newly found target has a closer range than a target already in the queue, the last target will be pushed off the queue.

This solution is not good for the SP2 either, because many atomic operations need to be performed, which are expensive on the SP2. Unlike Express, MPL does not support atomic operations directly. Assuming 50 μs per message (cf. Table 3), an atomic operation will need at least 12,700 μs when 128 tasks are used. This figure does not even include the time needed to process the messages, and the extra overhead due to network congestion. Thus, if more than 7 atomic operations are needed, this solution is worse than Paradigm 3, specified below.

Paradigm 3. Gather and Sequential Compute : After computing targets in the TD step by n tasks in parallel, a *gather* operation is performed by these n tasks to put the entire `detect_cube` in task 0, which then performs target report sequentially. The total execution time of the HO-PD is estimated to be 0.08 seconds on an SP2 with 128 nodes.

Paradigm 4. Compute and Reduce : This paradigm is the best one for target report and was used in the STAP benchmarks. Each of the n tasks computes a local report of at most 25 closest-ranged targets. Then a user-defined *reduction* operation is performed to sort the n target reports and to put the final target report in task 0. The operator in this *reduction* sorts two local reports into one report. The total execution time is estimated by:

$$T = 6 \times (RNG \times PRI \times BEAM / n) \times T_{flop} + T_{reduce} \quad (4)$$

On a 128-node system, the computation time is negligible (only about 0.03/128 seconds). As indicated in Table 2, a reduction takes $20 \log n + 23 = 163 \mu s$. However, this is for reducing one floating-point number. A target report has 25 elements, each consisting of four numbers (*pri, rng, beam,* and *power*), a total of 100 floating-point numbers. Multiplying 163 by 100, we estimate T_{reduce} to be about 16,300 μs, or 0.016 seconds, which is four times faster than Paradigm 3. This paradigm is the best one for target reporting and was chosen to implement the target search routine in the APT and HO-PD benchmarks.

5.4 Performance Prediction Metrics

After a parallel algorithm is designed, its performance should be predicted, taking into consideration all communication overhead. As shown in Figure 3, if the performance is not satisfactory, the parallel algorithm should be analyzed to reveal the potential performance bottlenecks, and a revised algorithm derived to overcome the difficulties. The following metrics are suggested to predict the benchmark performance before actually running the parallel code:

(1) *Sequential Time*, T(1), is the total execution time (in seconds) of a sequential program running on one node.

(2) *Parallel Time*, T(n), is the total execution time (in seconds) of a parallel program running on n nodes, including all communication overhead.

(3) *Speedup Factor* is defined by the speed ratio S =T(1) / T(n).

(4) *Sustained Computation Rate*, C = total workload / T(n). The unit of C is MFLOPS. This parameter is often called the *sustained performance*.

(5) *Efficiency*, E = C / 266n, is the ratio of the sustained performance to the peak performance of an n-node system. Recall that each SP2 node is capable of 266 MFLOPS peak performance.

From Figure 2, the total execution time of the HO-PD is *T(1)* = 130.61 seconds. Since each of the DP, BF, and TD steps can be fully parallelized, the execution time for parallel computing in these steps is 130.16/n when n nodes are used. There are four communication steps: a *total exchange* (where the message length is $m = 50$ MB/ n^2), two *shifts* (m = 393 KB for each shift), and a *reduction*. From Table 2, the total *communication time* is estimated by:

$$\begin{aligned} T(comm) &= T(index) + 2T(shift) + T(reduce) \\ &= 1.5n^{-0.71} + 0.005358 \log n + 0.003144 \end{aligned} \quad (5)$$

The total *execution time* for the parallel HO-PD program is predicted by:

$$T(n) = 130.16/n + T(comm) \qquad (6)$$
$$= 130.16/n + 1.5n^{-0.71} + 0.0044\log n + 0.0314$$

With 256 nodes, Equations 5 and 6 lead to a predicted HO-PD performance of 21.12 GFLOPS, corresponding to a speedup of 214 and an efficiency of 31.01% on the SP2. In Section 6, we shall show that these performance predictions on the SP2 are indeed very close to the measured performance. This demonstrates that the early-prediction scheme is indeed effective for use in the parallelization of the STAP programs.

After the parallelization paradigm is selected and verified, we proceed to the actual writing of the parallel program. It could help reduce errors by first developing a parallel program skeleton. Such a skeleton for the parallel HO-PD benchmark is given in the Appendix. This parallel HO-PD skeleton highlights the distributed node program as depicted in Figure 5.

A technique we found useful is to first develop *synchronous* parallel code. That is, add barriers between stages in the code. This makes semantic and performance debugging easier. After the code is fully debugged, the barriers can be removed to eliminate unnecessary overhead.

6. Parallel STAP Benchmark Performance

The STAP benchmark results are presented in this section. In each benchmark program, we show the *total execution time* in seconds, and the *sustained execution rate* in GFLOPS. These are measured performance results subject to the benchmark conditions specified in Section 2. These results are further analyzed in Section 7.

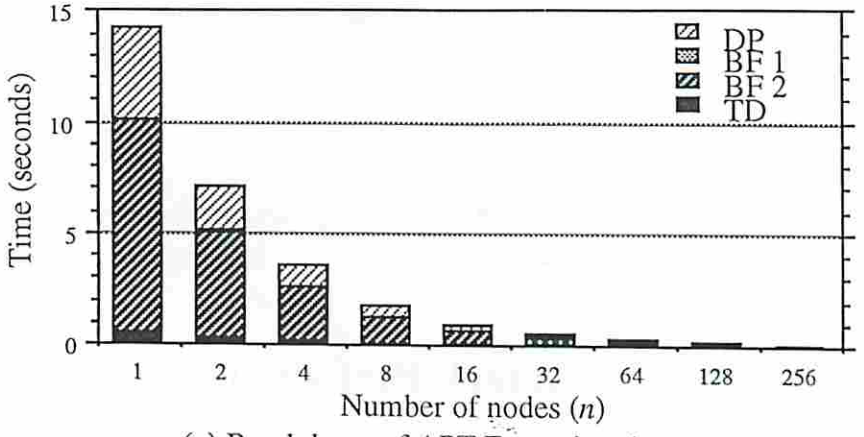
6.1 APT Benchmark Results

The structure of the parallel APT program is very similar to that shown in Figure 5. In Doppler Processing (DP), all independent FFTs are distributed evenly to the n nodes for parallel execution. After a *total exchange* operation, the Householder transform (HT) is performed on a single node. Then a global *broadcast* operation is performed over all 256 nodes, followed by the *beamforming* (BF) and *target detection* (TD) steps over 256 nodes. Finally, the *reduction* operation is performed to merge target reports from all 256 nodes.

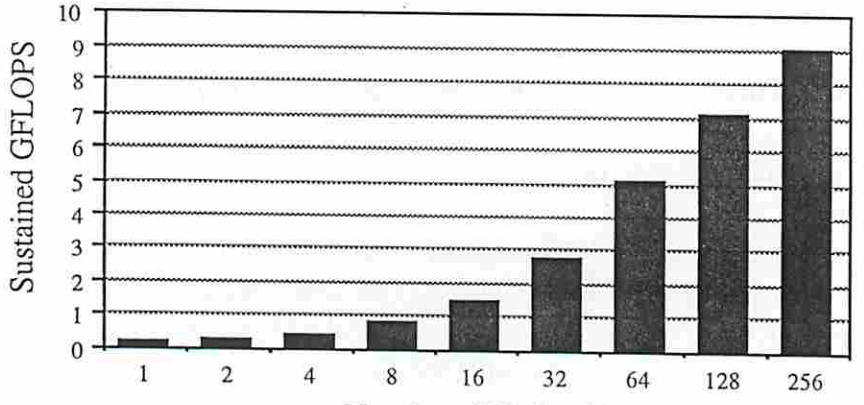
We achieved a total execution time of 0.16 seconds for the entire APT program on an SP2 configuration of 256 nodes. Figure 6a shows the execution time as a function of machine size. Figure 6b shows the sustained GFLOPS rate as a function of machine size. Up to 256 nodes, doubling the number of nodes halves the computation time as predicted. On 256 nodes, the computation took 0.09 seconds. The communication overhead is as

low as 0.056 seconds on an SP2 with 64 nodes. The time spent on the *total exchange* operations decreased from a high of 0.44 seconds on 2 nodes to 0.031 seconds on 64 nodes. The decrease in the *total exchange* time with increasing machine size is attributed to the decreasing message size.

Our results for the APT benchmark are encouraging. Time spent on computation decreases with increasing number of nodes used. Furthermore, the communication overhead remains under control for up to 256 nodes. Including all system and communication overheads, the parallel APT program achieved a sustained performance of 8.9 GFLOPS over 256 nodes. The corresponding total execution time was reduced to 0.16 seconds.



(a) Breakdown of APT Execution time



(b) Parallel APT Execution Rate

Figure 6 Parallel APT performance results on the SP2 System

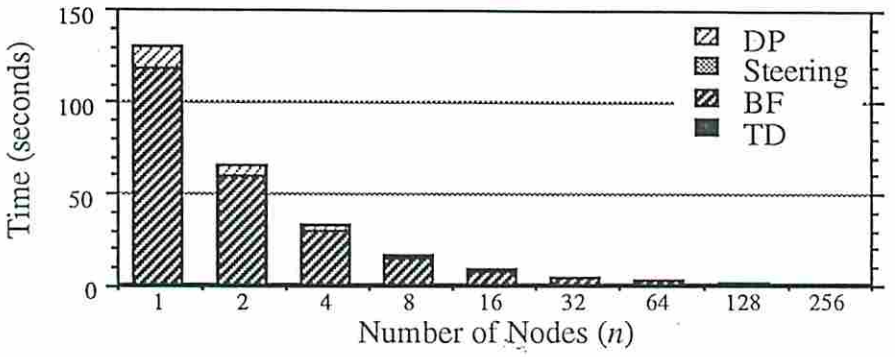
6.2 HO-PD Benchmark Results

With 256 nodes, we achieved a total execution time of 0.56 seconds for the entire HO-PD program. Figure 7a shows a breakdown of the computation components. Figure

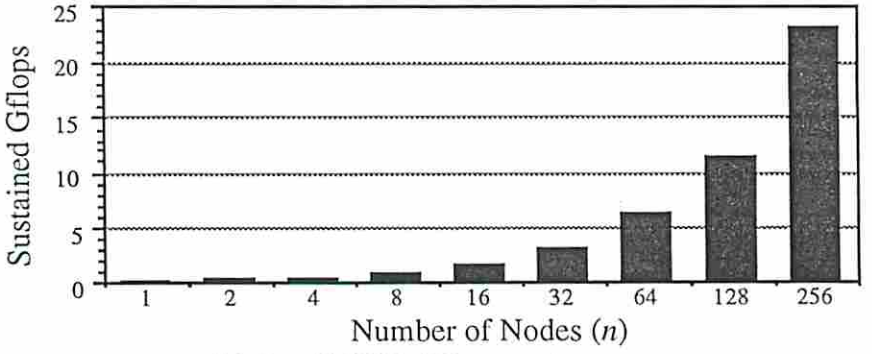
7b shows the sustained GFLOPS rate. The best performance is 23 GFLOPS over 256 nodes.

The Doppler Processing involves the mapping of 1024 FFTs onto n nodes. The BF and TD steps exhibit a maximum DOP of 128 along the PRI dimension. Using Amdahl's law, we predicted that 128 nodes can achieve 12.8 GFLOPS, which is only slightly higher than the measured value of 11.3 GFLOPS.

On 256 nodes, the computation took 0.44 seconds out of a total execution time of 0.56 seconds. Total overhead was as low as 0.12 seconds on 256 nodes. The time spent on the *total exchange* operation decreased from a high of 0.72 seconds on 2 nodes to 0.069 seconds on 128 nodes. This is attributed to the decreasing message size as the machine size increases.



(a) Breakdown of parallel HO-PD execution time



(b) Parallel HO-PD execution rate

Figure 7 Parallel HO-PD performance on the SP2 system

6.3 GEN Benchmark Results

The structure of the parallel GEN programs is illustrated in Figure 8. We execute the sort program, the FFT program, the vector multiply (VM) program, and the linear algebra (LA) program in a sequence. Parallelism is exploited along the dimension DIM3 in the sorting code with a maximum DOP of 512. The first two FFT steps (FFT1 and

FFT2) exploit a maximum DOP of 1536 along the DIM3 dimension. The FFT3 step exploits a maximum DOP of 128 along the DIM2 dimension. Similar selections are made for the three vector multiply steps. The linear algebra program has a DOP of only 32 along the DIM2 dimension specified in [4].

Almost linearly scalable performances were observed in the total execution time and sustained GFLOPS rate in all four GEN programs. In particular, we observed a superlinear speedup when 2 to 32 nodes are used simultaneously. This is due to the fact that the sequential execution time includes a much larger memory-access overhead resulting from all 100 MB of data being fetched from the same node memory. When the program is run on 32 nodes, only 3.1 MB of data are accessed from each of the node memories. With distributed data, much fewer page faults and cache misses occur, and thus the memory overhead is significantly reduced.

We plot the performance results in Figure 9 as functions of the SP2 machine size. Figure 9a shows the total execution time, with four bars per each machine size. Communication overheads exist for FFT and VEC programs. SORT and LA programs do not have overheads. Figure 9b shows the sustained GFLOPS including all overheads.

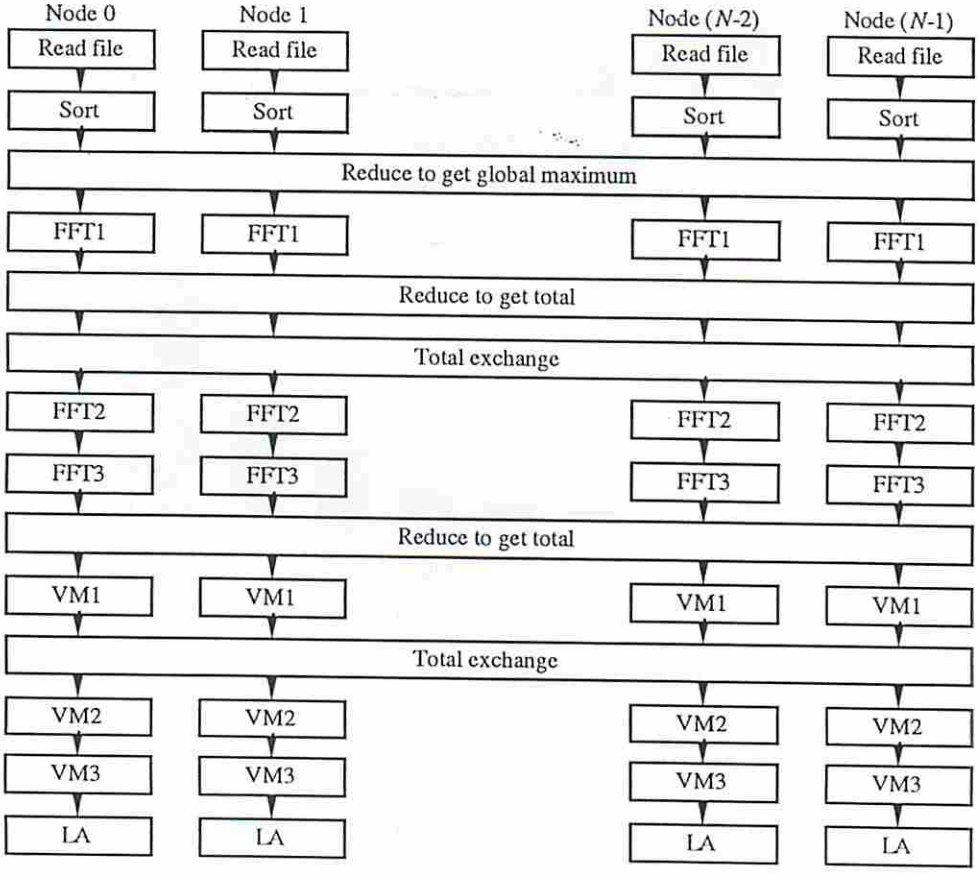
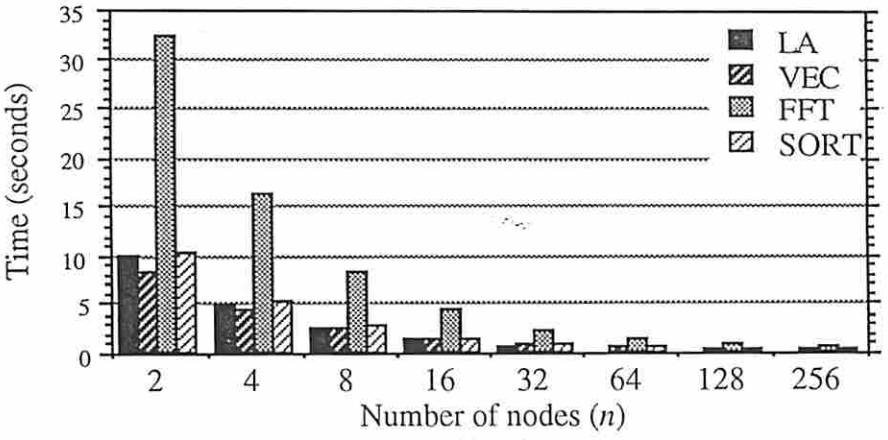


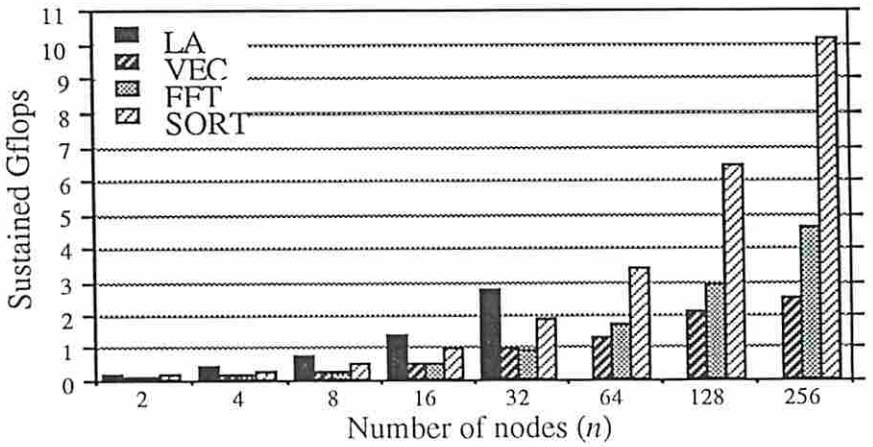
Figure 8 Structure of the Parallel GEN benchmark program on the SP2

With 256 nodes, we achieved 10.2 GFLOPS, 4.5 GFLOPS, 2.45 GFLOPS, and 2.75 GFLOPS for the SORT, FFT, VEC, and LA programs, respectively. These include all forms of overheads, such as memory access delay, message-passing latency, and system interfaces in a time-sharing SP2 environment. The performance of the STAP benchmark programs could be further improved if the following conditions could be created on the SP2 :

(1) Use a dedicated switching network (the HPS) without sharing with other user groups.
(2) Use double-precision format for floating-point operations in the sequential subroutines in the suite.
(3) Further optimize the FFT code by reducing the *total exchange* overhead.
(4) Increase the local memory size and enhance the data locality in the local caches.
(5) Use a real-time OS to manage the dedicated use of all the SP2 resources for STAP applications.



(a) Execution time



(b) Processing rate

Figure 9 Parallel General benchmark performance results on the SP2

7. Scalable Performance Analysis

The speedup and system efficiency performance are presented below. Then we discuss the scalability of STAP benchmark over SP2 machine size and radar parameters. This scalability analysis is useful to extend the STAP benchmarks for future generations of MPPs and adaptive radar systems.

7.1 Scalability Over Machine Size

For a fixed problem size, scalability refers to how well the speedup grows with increasing machine size. Figure 10 shows the speedup performance of all six STAP benchmark programs, when the problem sizes are fixed to the nominal data set. All programs scale well up to 256 nodes with the exception of LA, which scales only up to 32 nodes due to a DOP of only 32.

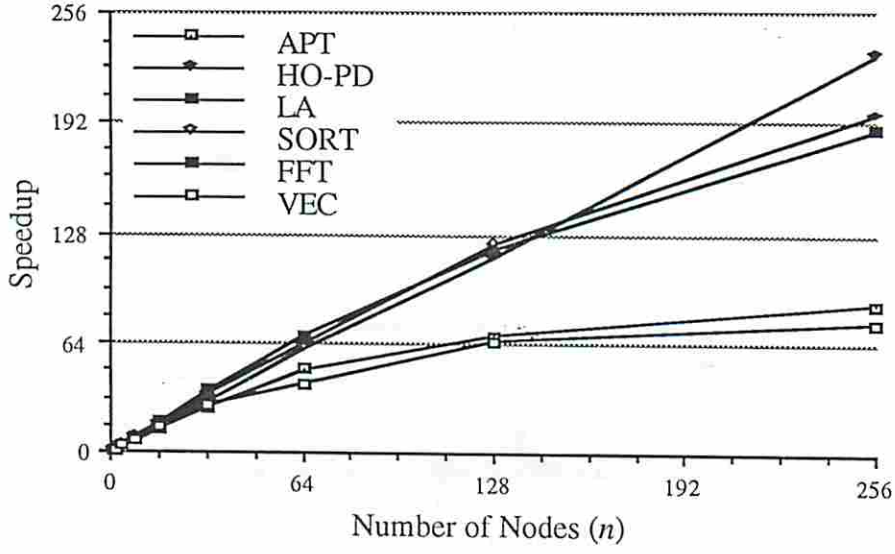


Figure 10 Speedup performance of six STAP benchmark programs on the SP2

In most of the STAP programs, the *total exchange* operation is the dominant communication step. Other communication overheads are relatively small. We define the *computation-to-communication ratio* of a STAP program as the ratio of the programs' total workload to the total number of bytes in the *total exchange* step. These values are shown in Table 5. For instance, in the APT benchmark, the total computational workload is 1.442 Gflop, and 16.78 MB of information is communicated in the *total exchange* step. This gives a computation-to-communication ratio of 86 flop per byte.

The HO-PD parallel program shows the best scalable performance of all the STAP programs, with an almost linear speedup up to 256 nodes. This is due to two facts: (1) All three computational steps of the HO-PD program can be fully parallelized to run

on up to 256 nodes, and (2) the computation-to-communication ratio is large (254 flop for every byte communicated).

Table 5 Key Characteristics of the Parallel STAP Programs

Program	Computation Workload (Gflop)	Aggregated Message Length in Total Exchange (MB)	Computation-to-Communication Ratio (Mflop/MB)
APT	1.442	16.78	86
HO-PD	12.850	50.33	254
GEN: FFT	1.909	100.66	19
GEN: VEC	0.612	100.66	6

The APT program's speedup saturates at 64 nodes, because the HT step has to be executed sequentially, creating a bottleneck when a large number of nodes are used (Amdahl's Law).

The LA program of the GEN benchmark shows a linear speedup up to 32 nodes, because it has no communication overhead and no sequential components.

The other three programs (FFT, VEC, and SORT) of the General benchmark all have a DOP of more than 256. However, they have smaller computation-to-communication ratios. Thus, their speedups are not as good as that of the HO-PD program.

Higher scalability does not necessarily imply good performance. The performance results of the parallel STAP programs executed on 256 nodes are summarized in Table 6. Although the FFT program has a better speedup than the APT program it has a worse performance of 4.5 GFLOPS, compared to the 8.9 GFLOPS for the APT. To understand why this is so, we need to look at the system efficiency values, shown in Figure 11.

Table 6 Summary of Parallel STAP Performance on a 256-Node SP2

Benchmark Program	Execution Time (seconds)	Sustained Speed (GFLOPS)	Speedup Over a Single SP2 Node	System Efficiency (%)
APT	0.16	9.0	87	13
HO-PD	0.56	23.0	231	34
GEN: Sort	0.12	10.2	196	15
GEN: FFT	0.42	4.5	188	7
GEN: VEC	0.25	2.5	78	4
GEN: LA	0.61	2.8	34	32

On a single node, the APT program has a very impressive efficiency of 38%, compared to 22%, 14%, and 11% for SORT, VEC, and FFT, respectively. The good sequential performance of the APT is the main reason why it has better performance than SORT, VEC, and FFT for up to 128 nodes. The efficiency of VEC drops more rapidly than that for FFT, because it has a smaller computation-to-communication ratio than FFT (6 versus 19 in Table 5); thus the communication overhead in VEC grows faster. The relatively low performance of FFT as shown in Figure 7 is mainly due to its slow sequential speed. The poor performance of VEC is due to its slow sequential speed and excessive communication overhead.

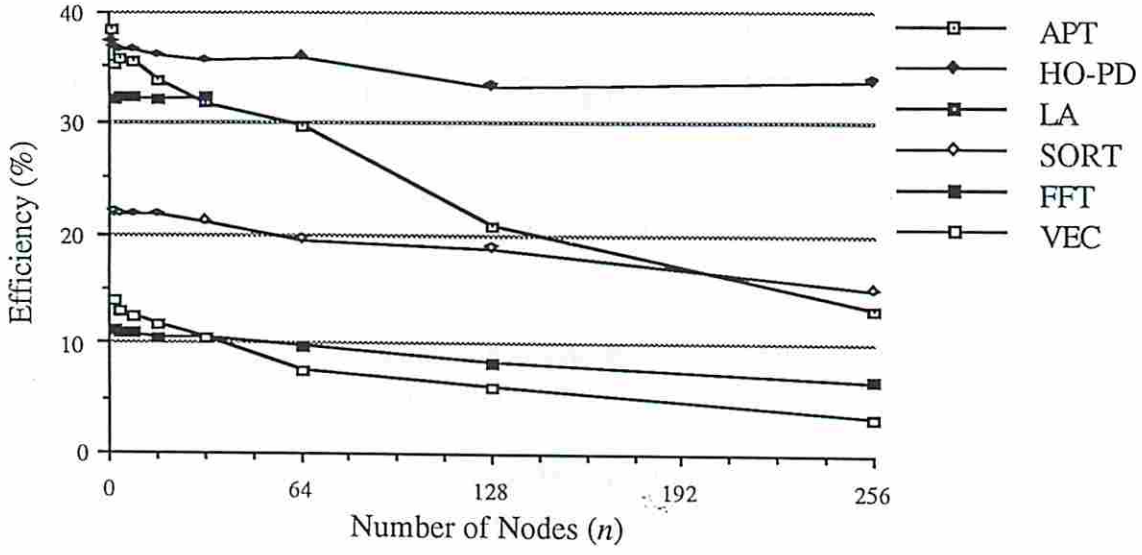


Figure 11 SP2 system efficiency of six STAP benchmark programs

7.2 Scalability Over Problem Size

Are the parallel STAP programs scalable over different problem sizes, which are governed by the choice of radar parameters? Our benchmarking results give a concrete answer. An accurate prediction was done using the early prediction scheme exemplified in Section 5. However, it requires defining many radar and program parameters. Bond [4] estimated that, to fully study the scalability of the APT benchmark, more than 10 million parameter sets need to be considered. For simplicity, we present below a theoretical analysis, which can also be applied to other MPPs.

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 7. The input data size and the workload are given by the STAP benchmark specification [17]. The *maximum parallelism* is computed by finding the largest DOP of the individual steps. The *critical path* is the execution time when an infinite number of nodes is used. For simplicity, we assume that every flop takes the same amount of time to execute. Then,

each step's contribution to the critical path is its workload (Table 4) divided by its DOP. The *average parallelism* is defined as the ratio of the workload to the critical path.

For instance, the HO-PD program has three steps: DP, BF, and TD. For a nominal data set, these steps have a DOP of 49,152, 256, and 256, respectively. The maximum parallelism is max(49,152, 256, 256) = 49,152. The critical path of the parallel HO-PD algorithm is 220/49,152 + (12618+14)/256 = 49.35 Mflop. The average parallelism is 12,852/49.35 = 261.

Table 7 Scalability of STAP Programs Over Problem Size

Program	Input Data Size (MB)	Workload *W* (Mflop)	Maximum Parallelism	Average Parallelism	Critical Path (Mflop)
APT (min)	0.12	5	1,800	10	0.51
APT (nom)	18.35	1,446	8,192	177	8.19
APT (max)	3,276.80	12,100,000	400,000	1,005	12,036.05
HO-PD (min)	0.15	21	1,176	17	1.24
HO-PD (nom)	50.33	12,852	49,152	261	49.35
HO-PD (max)	3,276.80	33,263,288	200,000	65,839	505.22
GEN (min)	0.26	6	2,048	103	0.05
GEN (nom)	100.00	5,326	196,608	108	49.27
GEN (max)	33,957.00	4,604,011	16,580,608	4,332	1,062.81

The average parallelism sets the upper bound on the achievable speedup [12]. For instance, suppose we want to speed up the sequential HO-PD program by a factor of 100. This is impossible to achieve using a minimal data set with an average parallelism of 17, but it is possible to achieve 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 no more than twice the average parallelism. When a number of nodes more than twice the average parallelism is used, at least 50% of the time, these nodes will be idle. By this heuristic, the parallel STAP programs with a large data set can take advantage of MPPs with thousands of nodes in current and future generations.

8. Conclusions and Suggestions

We summarize below the main results obtained and make a number of suggestions regarding the selection of processors, communication cost, degree of parallelism, grain size, programming paradigm, benchmark performance, computation-to-

communication ratio, atomic operations, Eureka, Exhandle, dynamic tasking, portability, and suitability of using the SP2 for real-time applications.

(1) **POWER2** : The POWER2 processor is excellent for floating-point computation intensive applications. The POWER2 shows good performance for most test programs, giving sustained performance ranging 52 to 199 MFLOPS per node. We expect the successor to the POWER2 processor to perform even better by increasing the superscalar degree, the cache size, speculative computations, and by using multiple functional units. Candidate CPU chips could be selected from the PowerPC 620/630, Alpha 21164, MIPS R10000, UltraSPARC, P6/ P7, etc.

(2) **Communication Cost** : Communication is expensive on the SP2. For instance, the time needed for one node to send a 4B message to another node is equivalent to the time to perform 12,000 floating-point operations. A *reduction* over 256 nodes is equivalent to 48,000 floating-point operations. A *total exchange* with 4B messages is equivalent to 170,000 floating-point operations.

(3) **Degree of Parallelism** : For coarse-grain parallel STAP programs, the overall maximal DOP is 256, except the linear algebra step in the General benchmark, which has a maximal DOP of 32. Other individual steps can have higher DOPs. For instance, the FFT step has a maximal DOP of RNG $\times$ EL = 8192 in the APT program, RNG $\times$ EL = 49156 in the HO-PD program, and DIM1 $\times$ DIM2 = 8192 in the GEN program.

(4) **Grain Size** : Only coarse-grain parallelism should be exploited on SP2. This choice is necessary because the time for a communication operation on the SP2 is equivalent to that of thousands or more floating-point operations. For nominal data sets, the coarse-grain approach is sufficient to achieve the desired 10 GFLOPS performance in most STAP benchmarks in selecting the programming paradigm.

(5) **Programming Paradigm** : Use Phased SPMD. Although other modes and paradigms can be used on the SP2, using the SPMD mode and the *compute-interact* paradigm not only is sufficient for the STAP, but also simplifies program development and performance analysis. The collective communication subroutines available in MPL are of great help.

(6) **Benchmark Performance** : The STAP programs achieved good performance on the SP2 as summarized in Table 6. With 256 nodes, we have achieved 9 GFLOPS for the APT benchmark and 23 GFLOPS for the HO-PD benchmark. For the General benchmark, we have achieved 10.2, 4.5, 2.5, and 2.8 GFLOPS for the Sorting, the FFT, the Vector Multiply, and the Linear Algebra steps, respectively. These measured results closely match the predicted performance.

(7) **Computation-to-Communication Ratio** : Higher computation-to-communication ratio implies higher speedup. The HO-PD program has the best performance among the three. A primary reason is that this program has a large computation-to-communication ratio, as shown in Table 5. In most of the programs, the total exchange is the dominant communication. Other communication overheads are small. In the APT benchmark, the total computational workload is 1.442 Gflop, and a total of 16.78 MB information is communicated in the total exchange step. This gives a computation to communication ratio of 86 flops per byte. For HO-PD, the ratio is 254.

(8) **Atomicity and Mutual Exclusion**: These are desired to support atomic transactions and critical sections. Express from Parasoft [20] supports these features. We could benefit from using atomic operations in the STAP benchmarking experiments if they were available on the SP2. We recommend that MPP vendors implement these useful features in their future models.

(9) **Eureka and Exhandle**: The *Eureka* feature supported on the Cray T3D [1] or the *exhandle* featured supported in Express [20] would be helpful in speeding up the multiple radar target searching process. It would be nice if an MPP has these features to support massively parallel searches in database and radar signal processing applications.

(10) **Dynamic tasking**: We assumed static tasking on the SP2 nodes. It would further improve the node efficiency if *dynamic tasking* were supported on the SP2. This will enable the dynamic creation, termination, and migration of tasks at run time. Furthermore, dynamic tasking is useful in enhancing fault tolerance on any MPP system. One can use PVM to implement dynamic tasking. It would be nice to include this feature in MPI or MPL functions.

(11) **Portability** : The parallel STAP programs can be ported to other message-passing systems without much difficulty. It is straightforward to port the parallel codes to Intel Paragon and any system that supports MPI, since the NX of Paragon and the public domain MPI have all the functionalities of the IBM MPL used in our codes. To port to Cray T3D [1], we need to convert the message-passing operations into shared memory equivalents to achieve maximum performance.

(12) **Suitability:** The commercial IBM/SP2 was not originally designed for real-time applications. Our benchmark results show that the Maui SP2 configuration is indeed very powerful to support real-time STAP applications, delivering up to 23 GFLOPS performance. However, the system achieved only a maximum of 34% efficiency.

Acknowledgments

This work was supported in part by a research subcontract from MIT Lincoln Laboratories to the University of Southern California. We would like to thank David

Martinez and Robert Bond at MIT Lincoln Laboratories for their support of this project. Craig Stunkel of IBM provided the IBM latency and bandwidth data for point-to-point communication. We are grateful to Peggy Williams, Rosanne Arnowitz, Blaise Barney, Tim Fahey, and George Gusciora at Maui High-Performance Computing Center for their timely answers to our technical questions. Special thanks are due to Lon Waters of MHPCC for his enthusiastic efforts in creating the dedicated testing environment for our experiments. The User Service Group at MHPCC is the best. We highly praise their contributions to the entire supercomputing community.

Appendix: Parallel HO-PD Program Skeleton

```
COMPLEX data_cube[PRI][RNG][EL], detect_cube[PRI][BEAM][RNG];
/* Doppler Processing (DP) */
parfor (j = 0; j < RNG; j++)  /* There are RNG tasks */
 {
  COMPLEX twiddle[PRI];  /* Local variable */
  compute_twiddle_factor (out twiddle, in PRI);
   /* Each task computes its own twiddle factor */
  for (k = 0; k < EL; k++)
   /* Each task computes EL FFTs */
   fft (inout data_cube[.][j][k], in twiddle, in PRI);
 }
barrier
total_exchange data_cube[.][j][.] to data_cube[i][.][.]
shift data_cube[i][.][.] to data_cube[i-1 % PRI][.][.]
shift data_cube[i][.][.] to data_cube[i+1 % PRI][.][.]
barrier
parfor (i = 0; i < PRI; i++)
 { /* Beamforming (BF) */
  beam_form (in data_cube[i][.][.],
          in data_cube[i-1 % PRI][.][.],
          in data_cube[i+1 % PRI][.][.],
          out detect_cube[i][.][.]);
  barrier
  /* Target Detection (CFAR) */
  for (j = 0; j < BEAM; j++)
   compute_target (inout detect_cube[i][j][.]);
  /* Target Report */
  m = 0;
  for (k = 0; k < RNG; k++)
   for (j = 0; j < BEAM; j++)
    if (IsTarget (in detect_cube[i][j][k]))
      {
       local_report[m].pri = i;
       local_report[m].beam = j;
       local_report[m].rng = k;
       local_report[m].power = detect_cube[i][j][k].real;
       m = m + 1;
       if (m > 25) goto finished;
      }
  finished:
 }
reduce local_reports to target_report
```

References

[1] D. Adams, *Cray T3D System Architecture Overview Manual*, http://www.cray.com, Cray Research, Inc., September 1993

[2] D. H. Bailey, et al, *NAS Parallel Benchmark Results 3-94*, RNR Tech. Report, March 1994

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

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

[5] J. Day, *Tutorial on Digital Adaptive Beamforming*, IEEE, 1993, National Radar Conference, Martin Marietta Corp., Utica, N. Y., April 22, 1993

[6] J. J. Dongarra, "Performance of Various Computers Using Standard Linear Equations Software in a Fortran Environment", Argonne National Laboratory Report, April 1987

[7] H. Franke, P. Hochschild, P. Pattnail, J. P. Prost, M. Snir, "MPI on IBM SP1/SP2 : Current Status and Future Directions", (contact frankeh@watson.ibm.com), 1993

[8] 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

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

[10] P. B. Hansen, *Studies in Computational Science: Parallel Programming Paradigms*, Prentice-Hall, N.J., 1995

[11] R. W. Hockney, "Performance Parameters and Benchmarking of Supercomputers", *Parallel Computing Journal*, 17 (1991) 1111-1130.

[12] K. Hwang, *Advanced Computer Architecture : Parallelism, Scalability, and Programmability*, McGraw-Hill Inc., New York, 1993

[13] K. Hwang, Z. Xu, and M. Arakawa, "STAP Benchmark Performance on the IBM SP2 Massively Parallel Processor", submitted to *Supercomputing*, San Diego, CA Dec. 4-8, 1995

[14] IBM Corp., *AIX Parallel Environment : Programming Primer*, Release 2.0, Pub. No. SH26-7223, IBM Corp., June 1994

[15] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing : Design and Analysis of Algorithms*, The Benjamin/Cummings Pub. Co., Redwood City, CA., 1994

[16] MHPCC, *MHPCC 400-Node SP2 Environment*, Maui High-Performance Computing Center, Maui, HI, October 1994

[17] MIT/LL, "STAP Processor Benchmarks", MIT Lincoln Laboratories, Lexington, MA, February 28, 1994

[18] MPI Forum, "MPI : A Message-Passing Interface Standard", *International Journal of Supercomputer Applications*, 1994, 8 (3/4)

[19] N. Nupairoj and L.M. Ni, "Benchmarking of Multicast Communication Services", Technical Report MSU-CPS-ACS-103, Michigan State University, April 1995

[20] Parasoft Corp., *Express C: User's Guide Version 3.0*, 2500 E. Foothill Blvd., Pasadena, CA 91107, 1990

[21] C. B. Stunkel, et al., "The SP2 Communication Subsystem", Technical Report, IBM T. J. Watson Research Center, Yorktown Heights, NY, 1994

[22] K. Teitelbaum. "Parallel Computation Techniques for Matrix Transformation", *Joint AEW Space-Time Adaptive Processing Requirements Study*, Meeting record held at Naval Research Lab., Washington, D.C., January 19-20, 1994

[23] Z. Xu, K. Hwang, "Modeling Communication Overhead: MPI and MPL Performance on the IBM SP2 Multicomputer", submitted to *Parallel and Distributed Technology*, IEEE Computer Society Press, January 10, 1995.

[24] Z. Xu and K. Hwang, "Early Prediction of MPP Performance by Workload and Overhead Quantification: A Case Study of the IBM SP2 System", submitted to the *Parallel Computing Journal*, April 8, 1995

Benchmark Evalution of the IBM SP2 for Parallel Signal Processing

Kai Hwang, Zhiwei Xu and Masahiro Arakawa

CENG Technical Report 95-13

Department of Electrical Engineering - Systems
University of Southern
Los Angeles, California 90089-2562
(213)740-4470