Benchmark Evaluation of the IBM SP2 for Parallel Signal Processing

Kai Hwang, Fellow, IEEE, Zhiwei Xu, Member, IEEE, and Masahiro Arakawa

Abstract—This paper evaluates the IBM SP2 architecture, the AIX parallel programming environment, and the IBM message-passing library (MPL) through STAP (Space-Time Adaptive Processing) benchmark experiments. Only coarse-grain parallelism was exploited on the SP2 due to its high communication overhead. A new parallelization scheme is developed for programming message passing multicomputers. Parallel STAP benchmark structures are illustrated with domain decomposition, efficient mapping of partitioned programs, and optimization of collective communication operations. We measure the SP2 performance in terms of execution time, Gflop/s rate, speedup over a single SP2 node, and overall system utilization. With 256 nodes, the Maui SP2 demonstrated the best performance of 23 Gflop/s in executing the High-Order Post-Doppler program, corresponding to a 94% system utilization. We have conducted a scalability analysis to reveal the performance growth rate as a function of machine size and STAP problem size. Important lessons learned from these parallel processing benchmark experiments are discussed in the context of real-time, adaptive, radar signal processing on massively parallel processors (MPP).

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

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). Our USC research team was among the first few user groups to test the performance of the SP2 using up to 256 nodes.

The STAP (space-time adaptive processing) benchmark [6], [20] consists of three programs: APT, HO-PD, and GEN, originally developed by MIT Lincoln Laboratory in sequential C code for adaptive radar signal processing on workstations. The benchmark requires performing billions of floating point operations (flop) over hundreds of Megabytes (Mbyte) 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 to 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, speed, speedup, and utilization?
4) Are the parallel STAP programs scalable with respect to increasing radar parameters and MPP size?

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 computing and communications capabilities of the SP2. Then we report the measured SP2 performance results, which are used to 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 the NAS benchmark experiments [3] and Dongarra’s Netlib site [7].

2 THE SP2 SYSTEM IN MAUI

Described below are the hardware platform, the software environment, and the testing conditions used in the STAP benchmark experiments conducted on the Maui SP2 system.

2.1 The SP2 Hardware Platform

As shown in Fig. 1, the Maui SP2 consists of 400 processing nodes [19]. During our dedicated use of the system, 256 nodes were allocated to us for exclusive execution of the STAP programs. The remaining nodes were still available to run other user 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 Mflop/s. The 400 nodes are interconnected by a 400-way multistage network called the High-Performance Switch (HPS) [23], along with some internal and external Ethernet connections. Although we were
given dedicated use of 256 nodes, we shared the HPS with
other users using the remaining nodes in the open re-
source pool.

$$\begin{array}{|c|c|}
\hline
\text{Dedicated for} & \text{Remaining nodes} \\
\text{STAP execution} & \text{for other users} \\
\hline
N_0 & N_1 \\
& \ldots \\
N_{255} & N_{256} \\
& \ldots \\
N_{399} & \hline
\end{array}$$

400-way High Performance Switch

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

The HPS is detailed in [23]. It is a packet switching, multi-
stage network with buffered wormhole routing. Messages are
divided into packets, which are in turn divided into 8-bit
flow control digits (flits). When there is no contention, it takes
only 125 ns for a flit to travel through one stage of the HPS.
Thus the contention-free hardware latency of the HPS is
small, only 875 ns or less for up to 512 nodes. The actual
latency seen by application processes is much higher: It takes
more than 40 µs for a process to send an empty message to
another process. A large portion of this message passing
latency is due to system software overhead.

2.2 AIX Parallel Programming Environment

Each node runs the IBM AIX operating system, the same
UNIX-based OS used in IBM RISC 6000 or PowerPC work-
stations. A parallel programming environment [18], [19]
was developed by IBM to support message-passing multi-
computing 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 [18] are supported in the AIX parallel programming
environment.

The AIX/OS supports dedicated single-user mode and
time-sharing mode for multiple users. Load balancing be-
 tween 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 devel-
ooped include the parallel operating environment (POE), the
message-passing library (MPL), ESSL scientific library, 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, and
FORGE.

2.3 Benchmark Conditions and Performance Metrics

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

- **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.

- **Use the best compiler options.** We compiled our pro-
grams 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. Without it, the compiler would generate code
for the generic RS/6000 architecture, common to all
Power, Power2, and PowerPC processors. Our test
shows that using the -qarch=pwr2 compiler option
can improve performance by as much as 15% on SP2.

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

- **Use single-precision floating-point numbers.** This condi-
tion 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.

- **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 we are more inter-
ested in wall-clock times and because it has a higher
  resolution of 1 microsecond. Each program was exec-
uted 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.

We define below some performance metrics used in our
work. The terminology is consistent with that proposed by
the Parkbench group [11], [12], [14], which is consistent
with the conventions used in other scientific fields, such as
physics.

**Workload** measures the amount of computational work
performed in an application. For scientific computing and
signal processing applications, where numerical calcula-
tions dominate, a natural metric is the number of floating
point operations that need to be executed. This metric
workload has a unit of one floating point operation (flop), a
Million flop (Mflop), or a Billion flop (Gflop). For instance,
an n-point FFT has a workload of 5nlog n flop. This should
be differentiated from the unit of computational speed, which
is Millions of flop per second or Billions of flop per second, de-
noted by Mflop/s and Gflop/s, respectively.

For each benchmark program B, we apply the following
performance metrics:

- **Sequential Execution Time, T(1),** which is the total ex-
  ecution time (in seconds) of the best sequential
  program for B running on one node.

- **Parallel Execution Time, T(p),** which is the total execution
time (in seconds) of a parallel program for B running
on p nodes, including all communication overhead.
TABLE 1
AIX PARALLEL PROGRAMMING ENVIRONMENT FOR THE IBM SP2

<table>
<thead>
<tr>
<th>Programming Model</th>
<th>SPMD or MPMD with message passing communication and static task creation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating System</td>
<td>Multi-user, multitasking UNIX-based AIX supporting: single user mode (dedicated use) batch mode (Loadleveler) time-sharing mode</td>
</tr>
<tr>
<td>Languages</td>
<td>C, C++, Fortran, HPF</td>
</tr>
<tr>
<td>Databases</td>
<td>DB2, Oracle, Sybase</td>
</tr>
</tbody>
</table>

- Speedup, which is defined by the ratio \( S = \frac{T(1)}{T(p)} \).
- Measured Computation Speed (or measured performance) \( C = \frac{W}{T(p)} \), where the workload \( W \) indicates how many MFlop in the original benchmark \( B \). It will not change even if a parallel program has to perform more flops. The unit of \( C \) is MFlop/s. This parameter is called benchmark performance in Parkbench, meaning the performance defined strictly for a particular benchmark \( B \).
- Utilization, \( U = \frac{C}{266p} \) is the ratio of the measured performance to the peak performance of a \( p \)-node SP2 system. Recall that each SP2 node is capable of 266 MFlop/s peak performance. This metric is sometimes called efficiency, which is also defined by the ratio \( S/p \). We will not call it efficiency in order not to cause confusion.

3 INTERNODE COMMUNICATIONS PERFORMANCE

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

3.1 Message-Passing Performance Metrics

Three types of communication operations are provided on the SP2: point-to-point communication, collective communication, and aggregated computation. In a 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. Because the HPS is a multi-stage network, conceptually all nodes are of equal distance from one another. Consequently, the time for a point-to-point communication may vary only with the message length, not with the physical location of the nodes involved. This is not completely true on SP2. Point-to-point communication time does vary with the physical locations of the sender and the receiver. But the variation is small.

In an aggregated 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. 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 \( p \) values, one from each of the \( p \) tasks. In a parallel prefix operation, also known as a scan, a group of tasks aggregate \( p \) partial results, one from each of the \( p \) tasks, into \( p \) 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, \( p \). In a broadcast operation, a single node sends an \( m \)-byte message to all \( p \) nodes. In a gather operation, a single node receives an \( m \)-byte message from each of the \( p \) nodes, so in the end \( mp \) bytes are received by that node. A scatter operation is just the opposite of a gather: a single node sends a distinct \( m \)-byte message to each of the \( p \) nodes, so in the end \( mp \) bytes are sent from that node. In a total exchange operation, every one of the \( p \) nodes sends a distinct \( m \)-byte message to each of the \( p \) nodes, so at the end \( mp^2 \) bytes are communicated. In a circular shift operation, node \( i \) sends an \( m \)-byte message to node \( (i+1) \mod p \).

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

\[
t = t_0 + \frac{m}{r_m}
\]  

where \( m \) denotes the message length in bytes. The parameter \( r_m \) is called the asymptotic bandwidth in Mbyte/s, which is the maximal bandwidth achievable when the message length approaches infinity. The parameter \( t_0 \) is called the latency (or start-up time), which is the time (in \( \mu s \)) needed to send a 0-byte message. Two additional parameters were derived. The half-peak length, denoted by \( m_{1/2} \), is the message length required to achieve half of the asymptotic bandwidth. The specific performance, denoted by \( \pi \), Mbyte/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:

\[
t_0 = \frac{m}{r_m} = \frac{1}{\pi_0}
\]  

From (1) and (2), the point-to-point communication time can be rewritten as
The $m_0/2$ metric is a handy one to indicate how fast the asymptotic bandwidth can be approached when the message size increases. An MPP with a large $m_0/2$ does not efficiently support short message communications, even if it has excellent asymptotic bandwidth. This metric was originally proposed by Hockney to capture how well SIMD and vector machines support short vector length parallel computation [15].

Hockney's model applies only to point-to-point communication operations. It needs to be generalized for collective communication and aggregated computation operations. Xu and Hwang [25] have obtained a generalized timing model:

$$t = t_0(p) + \frac{m}{r_0(p)}$$  \hspace{1cm} (4)

where the latency $t_0(p)$ and the asymptotic bandwidth $r_0(p)$ can be linear or nonlinear functions of $p$. 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 all follow the general form given in (4).

Timing expressions are obtained for some MPL collective message-passing operations on the SP2, as shown in Table 2. Details on how to derive these and other expressions are treated in [25], where the MPI performance on SP2 is also compared to the native IBM MPL operations.

The expression for point-to-point communication translates to a $t_0 = 46 \mu s$ latency and a $r_0 = 28.6$ Mbyte/s asymptotic bandwidth. IBM [23] reported better values: $t_0 = 40$ and $r_0 = 35.5$. Each expression in Table 2 is obtained by curve-fitting all measured timing data with the number of nodes varying from two to 256 and the message length varying from 4-byte to 16-Mbyte. The best bandwidth we observed in our measurement is 35.5 Mbyte/s, the same as what IBM has reported [23].

The aggregated asymptotic bandwidth, denoted by $R_0$ Mbyte/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 MPS. For a point-to-point communication, $R_0 = r_0$. For a total exchange, $R_0 = p^2 r_0$. For other collective communications (broadcast, gather, scatter, and shift), $R_0 = p r_0$. From Table 2, the greatest aggregated bandwidth is achieved with the circular shift operation, reaching 4 Gbyte/s on 256 nodes.

In choosing communication operations, we should always apply the most powerful, high-level operations, 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 will be up to 50 times slower on the SP2 than using just one high-level MPL function.

3.3 MPL Communication Times

From the above analysis, we observe that the MPL communication time depends not only on the message size $m$, but also on the number of nodes $p$ involved in the collective communications or aggregated computations. In Table 3, we list the measured times on the SP2 for a short message of $m = 4$ bytes and a long message of $m = 64$ KB, assuming that $p = 128$ nodes are involved in the collective or aggregated operations.

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.

4.4 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. The sequential performance will be compared with parallel performance in later sections.

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 [20], [21]. The APT and HO-PD are both written to test the STAP algorithms for adaptive radar signal processing. Both programs start with Doppler processing (DP), in which the program performs a large number of one-dimensional FFT computations. Both end with target detection. The difference between the two programs is that APT performs a Householder transform (HT) to generated jammer-nullled 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. 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. Details of these substeps can be found in [20], [21].

Component algorithms and the workloads of the three STAP benchmarks are identified in the leftmost three columns of Table 4. The STAP workload is related to the size of the input data set, which in turn depends on the values of radar parameters. Table 4 shows the workload for a nominal 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 Teraflop for a maximal data set [6].
### Table 2
**Communication Time and Aggregated Bandwidth on the IBM SP2**

<table>
<thead>
<tr>
<th>MPL Command</th>
<th>Communication Time in μs</th>
<th>Aggregated Bandwidth in Mbyte/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Point-to-Point</td>
<td>46 + 0.035n</td>
<td>28.6</td>
</tr>
<tr>
<td>Broadcast</td>
<td>186log p + (0.025log p/m)</td>
<td>40log p</td>
</tr>
<tr>
<td>Gather/Scatter</td>
<td>170log p + 15 + (0.025p – 0.02)m</td>
<td>(0.003p – 0.02)</td>
</tr>
<tr>
<td>Total Exchange</td>
<td>80log p + (0.03p^1.25)m</td>
<td>33.36</td>
</tr>
<tr>
<td>Circular Shift</td>
<td>80log p + 60 + (0.003log p + 0.04)m</td>
<td>(0.003log p + 0.04)</td>
</tr>
<tr>
<td>Barrier</td>
<td>94log p + 10</td>
<td>N/A</td>
</tr>
<tr>
<td>Reduction</td>
<td>20log p + 23</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Table 3
**Measured Communication Time on a 128-Node SP2 for Short (4 Bytes) and Long (64 Kbytes) Messages**

<table>
<thead>
<tr>
<th>Message Passing Library</th>
<th>Operation Type</th>
<th>Message Length</th>
<th>Communication Time in μs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Point-to-Point</td>
<td>One Way</td>
<td>4 B</td>
<td>49</td>
</tr>
<tr>
<td>Collective Communication</td>
<td>Broadcast</td>
<td>4 B</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td>Gather/Scatter</td>
<td>4 B</td>
<td>130</td>
</tr>
<tr>
<td></td>
<td>Total Exchange</td>
<td>4 B</td>
<td>949</td>
</tr>
<tr>
<td></td>
<td>Circular Shift</td>
<td>4 B</td>
<td>105</td>
</tr>
<tr>
<td>Aggregated Computation</td>
<td>Barrier</td>
<td>N/A</td>
<td>678</td>
</tr>
<tr>
<td></td>
<td>Reduction</td>
<td>4 B</td>
<td>153</td>
</tr>
<tr>
<td></td>
<td>Parallel Prefix (Scan)</td>
<td>4 B</td>
<td>577</td>
</tr>
</tbody>
</table>

### Table 4
**Sequential STAP Benchmark Performance**

<table>
<thead>
<tr>
<th>Program</th>
<th>Component Algorithms</th>
<th>Workload (Mflop)</th>
<th>Execution Time (Seconds)</th>
<th>Measured Speed (Mflop/s)</th>
<th>Utilization (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>APT</td>
<td>DP</td>
<td>1447</td>
<td>14.27</td>
<td>100</td>
<td>100.00</td>
</tr>
<tr>
<td></td>
<td>HT</td>
<td>3</td>
<td>0.04</td>
<td>72</td>
<td>27.07</td>
</tr>
<tr>
<td></td>
<td>BF</td>
<td>1314</td>
<td>9.64</td>
<td>136</td>
<td>51.22</td>
</tr>
<tr>
<td></td>
<td>TD</td>
<td>46</td>
<td>0.57</td>
<td>75</td>
<td>28.01</td>
</tr>
<tr>
<td>HO-PD</td>
<td>DP</td>
<td>12853</td>
<td>130.61</td>
<td>98</td>
<td>37.00</td>
</tr>
<tr>
<td></td>
<td>BF</td>
<td>220</td>
<td>11.62</td>
<td>19</td>
<td>7.02</td>
</tr>
<tr>
<td></td>
<td>TD</td>
<td>14</td>
<td>0.17</td>
<td>32</td>
<td>30.96</td>
</tr>
<tr>
<td>GEN</td>
<td>SORT</td>
<td>5326</td>
<td>121.05</td>
<td>44</td>
<td>16.54</td>
</tr>
<tr>
<td></td>
<td>FFT</td>
<td>1183</td>
<td>22.80</td>
<td>52</td>
<td>19.51</td>
</tr>
<tr>
<td></td>
<td>VEC</td>
<td>1909</td>
<td>79.14</td>
<td>24</td>
<td>9.06</td>
</tr>
<tr>
<td></td>
<td>LIN</td>
<td>604</td>
<td>19.11</td>
<td>32</td>
<td>11.88</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1650</td>
<td>20.23</td>
<td>32</td>
<td>31.00</td>
</tr>
</tbody>
</table>

The workload values in Table 4 are obtained through code inspection, counting only dominant terms. For instance, the dominant computations in the DP step of the HO-PD benchmark are RNG × EL FFTs, where each FFT is of PRI points. Given that PRI = 128, RNG = 1024, EL = 48, and each n-point FFT has 5log n flop, the workload for the DP step is about 220 Mflop. The formulae for computing the workload based on radar parameters are detailed in [6].

#### 4.2 Sequential STAP Benchmark Results

We executed the sequential version of the STAP benchmark programs with a nominal data set on a single SP2 node with 256 MB of main memory and 256 KB of cache. The results are summarized in Table 4. The component algorithms show a large variation in the Mflop/s rate. The BF2 step in APT achieved a 136 Mflop/s performance. The FFT in the GEN program showed only a 24 Mflop/s performance out of 266 Mflop/s, the peak speed of a single POWER2 node. The following observations are based on sequential benchmark results on a single node in 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 achieved 6 Mflop/s on a single SP2 node:

```c
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];
```
The above code is called the inner-product method [15]. With the loop interchange shown below (called the middle-product method in [15]), a 17-fold 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) The choice of data structure can greatly impact the performance. For instance, arrays of complex numbers are frequently used in signal processing applications. Two popular methods to represent a complex array follow:

```c
/* 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];
```

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%.

3) The FFT performance is low (about 19 to 24 Mflop/s, or 7% to 9% utilization). The original STAP codes had an even lower FFT performance, only 14 to 18 Mflop/s. We have tried techniques such as loop unrolling without much improvement. The performance gain in the new FFT routine is based on a FFT code from George Gusciora of MHPCC, which differs from the original FFT routine in representing complex numbers by an array of structures (i.e., Method 1 above). Improving FFT performance is still one of our on-going efforts in optimizing the STAP benchmarks.

IBM has reported much higher FFT performance (up to 140 Mflop/s) using their ESSL FFT routines [22]. Since the STAP suite is meant to be portable, we did not use any special library routines in the parallel STAP codes. However, we did test the ESSL FFT routines with the STAP codes on SP2. The result is only slightly better. Benchmark results from vendors are often obtained under ideal conditions, which do not necessarily exist in real applications. For instance, the ESSL FFT routine needs to be called at least twice, the first time to initialize and the second to perform the actual FFT. The 140 Mflop/s FFT performance was obtained by repeating the same FFT 16 times after initializing, and with the empty-cache assumption. We performed the unrealistic repeated ESSL FFT, but without the empty-cache assumption, to achieve a 110 Mflop/s performance. However, the same ESSL FFT routine, when used in the STAP codes, only sustained no more than 30 Mflop/s.

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 a 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 Fig. 2. 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 [26]. 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 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 [25] to find methods for modeling communication overheads, and to [26] for procedures to carry out the early performance prediction. Details of these methods are not repeated in this paper.

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 computational operations (flops) performed by one task between two communication operations. In general, the granularity is inversely proportional to the degree of parallelism (DOP) exploited [16]. Granularity should be chosen based on three factors:

1) the application performance requirement,
2) the input data size, and
3) the communication overhead of the MPP.

For the STAP benchmarks, the performance requirement was set to 10 to 25 Gflop/s [6], [20]. This calls for a DOP of 128 or more, according to Fig. 2. If only one Gflop/s 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 Gflop/s.

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 [1], which has a smaller MFlop/s rate per node but faster communication support than the SP2.

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 p tasks are executed in parallel, each executes (RNG \times EL)/p FFTs without any communication. Similarly, the BF and the TD steps should be parallelized along the PRI dimension [6]. This strategy requires a total exchange after the DP step.

In Fig. 3, 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/p and p is the machine size. After the total exchange, parallelism is exploited along the PRI dimension, as shown on the right. When 256 nodes are used, we need to exploit additional parallelism along the range segment dimension (not shown), as there is not enough parallelism in the PRI dimension.

5.3 Selection of Parallelization Paradigm
For all of the STAP programs, it is sufficient to use a simple compute-interact paradigm, which is similar to Valiant’s Bulk Synchronous Parallel (BSP) model [24]. 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 Fig. 5.

In the first phase, the nodes perform the Doppler Processing step, which is followed by an interact phase, where the nodes perform a total exchange and three circular shifts. The second compute phase consists of the BF step followed by the TD step, as no interaction is needed between these two steps. Finally, another interact phase performs a global reduction to merge the partial target reports generated by the nodes into a complete target report.

There are many other parallel programming paradigms, such as compute-aggregate-broadcast, synchronous iteration, asynchronous iteration, worker-pool, divide and conquer, and pipelining [3], [10], [24]. 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 (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. A target is detected if the IsTarget function evaluates to True. 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 256 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 [5], 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.

**Paradigm 1. Synchronous Iteration.** $p$ 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 $94 \log p + 10 \mu s$, which is about $762 \mu s$ per barrier over 256 nodes. For the HO-PD benchmark, RNG = 1024, which gives a total barrier time of about $762 \times 1024 = 0.78$ seconds, far exceeding the 0.1 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 $\mu s$ per message (cf. Table 3), an atomic operation will need at least 12,700 $\mu s$ when 256 tasks are used. Thus, if more than seven 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 $p$ tasks in parallel, a gather operation is performed by these $p$ 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 0.1 seconds on an SP2 with 256 nodes.

**Paradigm 4. Compute and Reduce.** Each of the $p$ tasks computes a local report of at most 25 closest-ranged targets. Then a user-defined reduction operation is performed to sort the $p$ 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 the sum of the time to compute local reports and the time to do the global reduction.

On a 256-node system, the time to compute local report is negligible (only about 0.17/256 seconds). As indicated in Table 2, a reduction takes $20 \log p + 23 = 183 \mu s$. 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 183 by 100, we estimate the target reduction time to be about 18,300 $\mu s$, or 0.018 seconds, which is five 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 programs.

### 5.4 Performance Prediction

After a parallel algorithm is designed, its performance should be predicted, taking into consideration all communication overhead. As shown in Fig. 2, 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.

From Table 4, the sequential execution time of the HO-PD is $T(1) = 130.61$ seconds. Since each of the DP, BE, and TD steps can be fully parallelized, the execution time for parallel computing in these steps is $130.61/p$ when $p$ nodes are used. There are four communication steps: a total exchange (where the message length is $m = 50$ Mbyte), two shifts ($m = 393$ Kbyte for each shift), and a reduction. From Table 2, the total communication time is estimated by:

$$T(\text{comm}) = T(\text{index}) + 2T(\text{shift}) + T(\text{reduce})$$

$$= 1.5p^{-0.71} + 0.0044 \log p + 0.0054$$

(5)

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

$$T(p) = 130.61/p + T(\text{comm})$$

$$= 130.61/p + 1.5p^{-0.71} + 0.0044 \log p + 0.0054$$

(6)

With 256 nodes, (5) and (6) lead to a predicted HO-PD performance of 21 Gflop/s, corresponding to a speedup of 214 and a utilization of 31% on the SP2. In Section 6, we shall show that this predicted SP2 performance is very close to the measured performance. This demonstrates that the early-prediction scheme is indeed effective. Interested readers can find details of this prediction scheme in [26].

After the parallelization paradigm is selected and verified, we proceed to the actual writing of the parallel pro-
gram. 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 Fig. 4.

![Diagram](image-url)

**Fig. 4.** Mapping of the parallel HO-PD program on a 256-node SP2 system, using the compute-and-reduce paradigm.

A technique we found useful is to first develop a 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 are 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 measured speed in Gflop/s. These are measured performance results subject to the benchmark conditions specified in Section 2.

6.1 HO-PD Benchmark Results

Fig. 5 shows the measured parallel execution time and speed as a function of machine size. With 256 nodes, we achieved a total execution time of 0.56 seconds for the entire HO-PD program, corresponding to a 23 Gflop/s speed and a 34% utilization.

The Doppler Processing involves the mapping of RNG × EL = 49152 FFTs onto p nodes. The BF and TD steps exhibit a maximum DOP of 128 along the PRI dimension. Our early prediction scheme shows that 128 nodes can achieve 12.8 Gflop/s, which is only slightly higher than the measured value of 11.3 Gflop/s. On 256 nodes, we need to exploit additional parallelism along another radar parameter, the range segment. The parallel code becomes more complex, but the computation took only 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.

An interesting phenomenon is observed in testing the parallel HO-PD program: In theoretical study of parallel computing, a common belief is that communication overhead increases with increasing machine size. This is not true for the parallel HO-PD program. Referring to (6) and (7), we see that although the shifts and the reduction times increases with respect to p, the total exchange time actually decreases as p increases. Since the total exchange is the dominates communication, the total communication time is also a decreasing function of p.

Fig. 6 plots the total communication time of the parallel HO-PD benchmark for different machine sizes. There is no communication for one node. Afterwards, the communication time decreases as p increases. The time spent on the total exchange operation decreased from a high of 0.72 seconds on two nodes to 0.04 seconds on 256 nodes. This is attributed to the decreasing message size (m is about 50/p Mbyte as the machine size p increases. This phenomenon is observed for almost all STAP benchmark programs.

![Graph](image-url)

**Fig. 5.** Parallel HO-PD performance on the SP2 system.

**Fig. 6.** Total communication time of the parallel HO-PD on the SP2 system.

6.2 APT Benchmark Results

The structure of the parallel APT program is very similar to that shown in Fig. 4. In Doppler Processing (DP), all independent FFTs are distributed evenly to the p 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.
Fig. 7 shows the measured parallel execution time and speed 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. Including all communication overheads, the parallel APT program achieved a sustained performance of 9 Gflop/s over 256 nodes. The corresponding total execution time was reduced to 0.16 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 \( p \) is attributed to the decreasing message size, which is about \( m = 16.78/p \) Mbyte.

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.

![Fig. 7. Parallel APT performance on the SP2 system.](image)

6.3 GEN Benchmark Results

The GEN benchmark consists of four component programs: the sorting (SORT) program, the FFT program, the vector multiply (VM) program, and the linear algebra (LA) program. They are completely independent. Three parameters (DIM1, DIM2, DIM3) determine the data set size, the workload, and the parallelism. For nominal data set, we have \( \text{DIM1} = 64, \text{DIM2} = 128, \text{and DIM3} = 1536. \)

For SORT, parallelism is exploited along DIM3 dimension. The maximum DOP is DIM3/3 = 512. The dominant communication is a reduction whose time is small compared to the computation time. 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. Additional parallelism can be exploited along the DIM1 dimension. Between FFT2 and FFT3, a total exchange needs to be performed, which has significant overhead, because the message size \( m \) about 100/p MB, is much larger than those in APT and HO-PD. The VM program is similar to the FFT program. The LA program does not need communication, but has a limited DOP of only DIM2/4 = 32 along the DIM2 dimension, as specified in [6].

We plot the GEN performance results in Fig. 8 as functions of the SP2 machine size. With 256 nodes, we achieved 10.2 Gflop/s, 4.5 Gflop/s, 2.45 Gflop/s, and 2.75 Gflop/s for the SORT, FFT, VEC, and LA programs, respectively. These include all communication overheads.

Almost linearly scalable performances were observed in the total execution time and Gflop/s rate in all four GEN programs. In particular, we observed a superlinear speedup when two 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.

![Fig. 8. Parallel General benchmark performance on the SP2 system.](image)

7 Scalable Performance Analysis

The speedup and utilization 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. Fig. 9 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.
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 program’s 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 Mbyte 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).

The APT program’s speedup saturates at 64 nodes, because the HFT 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 better 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 Gflop/s, compared to the 9 Gflop/s for the APT. To understand why this is so, we need to look at the system efficiency values, shown in Fig. 10.

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 Fig. 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.

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? The early prediction scheme can be used to predict the scalability, as exemplified in Section 5. However, it requires defining many radar and program parameters. Bond [6] 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 be applied to other MPPs as well.

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 [6], [22]. The maximum parallelism is computed by finding the largest DOP of the individual steps. The critical path (or more precisely, the length of 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. Each step’s contribution to the critical path is its workload (Table 4) divided by its DOP. The average parallelism [16] 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 49152, 256, and 256, respectively. The maximum parallelism is max(49152, 256, 256) = 49152. The critical path of the parallel HO-PD algorithm is 220/49152 + (12618 + 14)/256 = 49.35 Mflop. The average parallelism is 12852/49.35 = 261.
<table>
<thead>
<tr>
<th>Program</th>
<th>Workload (Gflop)</th>
<th>Aggregated Message Length in Total Exchange (Mbyte)</th>
<th>Computation-to-Communication Ratio (flop/byte)</th>
</tr>
</thead>
<tbody>
<tr>
<td>APT</td>
<td>1.447</td>
<td>16.78</td>
<td>86</td>
</tr>
<tr>
<td>HO-PD</td>
<td>12.653</td>
<td>50.33</td>
<td>254</td>
</tr>
<tr>
<td>GEN: FFT</td>
<td>1.900</td>
<td>100.66</td>
<td>18</td>
</tr>
<tr>
<td>GEN:VEC</td>
<td>0.614</td>
<td>100.66</td>
<td>6</td>
</tr>
</tbody>
</table>

**TABLE 6**

**Summary of Parallel STAP Performance on a 256-Node SP2**

<table>
<thead>
<tr>
<th>Benchmark Program</th>
<th>Execution Time (seconds)</th>
<th>Measured Speedup (Gflop/s)</th>
<th>Speedup Over a Single SP2 Node</th>
<th>Utilization (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>APT</td>
<td>0.16</td>
<td>9.0</td>
<td>87</td>
<td>13</td>
</tr>
<tr>
<td>HO-PD</td>
<td>0.58</td>
<td>2.0</td>
<td>231</td>
<td>34</td>
</tr>
<tr>
<td>GEN: Sort</td>
<td>0.12</td>
<td>10.2</td>
<td>196</td>
<td>15</td>
</tr>
<tr>
<td>GEN: FFT</td>
<td>0.42</td>
<td>4.3</td>
<td>188</td>
<td>7</td>
</tr>
<tr>
<td>GEN:VEC</td>
<td>0.25</td>
<td>2.3</td>
<td>78</td>
<td>4</td>
</tr>
<tr>
<td>GEN:LA</td>
<td>0.57</td>
<td>2.4</td>
<td>34</td>
<td>32</td>
</tr>
</tbody>
</table>

**TABLE 7**

**Scalability of STAP Programs Over Problem Size**

<table>
<thead>
<tr>
<th>Program</th>
<th>Input Data Size (MB)</th>
<th>Workload W (Mflop)</th>
<th>Maximal Parallelism</th>
<th>Average Parallelism</th>
<th>Critical Path (Mflop)</th>
</tr>
</thead>
<tbody>
<tr>
<td>APT (min)</td>
<td>0.23</td>
<td>5</td>
<td>1800</td>
<td>10</td>
<td>0.51</td>
</tr>
<tr>
<td>APT (norm)</td>
<td>16.77</td>
<td>1447</td>
<td>6152</td>
<td>177</td>
<td>8.19</td>
</tr>
<tr>
<td>APT (max)</td>
<td>32.68</td>
<td>121000000</td>
<td>4000000</td>
<td>1005</td>
<td>12036.05</td>
</tr>
<tr>
<td>HO (min)</td>
<td>0.15</td>
<td>21</td>
<td>1176</td>
<td>17</td>
<td>1.24</td>
</tr>
<tr>
<td>HO (norm)</td>
<td>50.33</td>
<td>12852</td>
<td>49152</td>
<td>261</td>
<td>49.35</td>
</tr>
<tr>
<td>HO (max)</td>
<td>32.68</td>
<td>33263288</td>
<td>200000</td>
<td>65839</td>
<td>505.22</td>
</tr>
<tr>
<td>GEN (min)</td>
<td>0.26</td>
<td>6</td>
<td>2048</td>
<td>103</td>
<td>0.05</td>
</tr>
<tr>
<td>GEN (norm)</td>
<td>100</td>
<td>526</td>
<td>196608</td>
<td>108</td>
<td>49.27</td>
</tr>
<tr>
<td>GEN (max)</td>
<td>33957</td>
<td>4604011</td>
<td>16560008</td>
<td>4352</td>
<td>1062.81</td>
</tr>
</tbody>
</table>

The average parallelism sets the upper bound on the achievable speedup [16]. 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 thousands of nodes in current and future generations of MPPs.

8 LESSONS LEARNED AND CONCLUSIONS

We summarize below the main results obtained and make a number of suggestions regarding the suitability of using the SP2 for signal processing applications, and the properties of the STAP benchmark suite that are relevant to parallelization.

1) **Sequential Performance:** It is very important to improve the performance of the sequential program. Otherwise, one may develop a parallel program with a good speedup, but still poor in raw performance. Almost all of today's processors in MPPs have the following features:

   - Multiple-functional units: On the POWER2 processor in SP2, there are a branch unit, two integer units, and two floating-point units.
   - Superscalar: Several instructions can be executed simultaneously on multiple functional units. On a POWER2, up to five instructions can be issued per clock cycle. For instance, one branch, two loads, and two floating-point instructions can all be executed in one clock.
   - Split Caches: On POWER2, there is a 32-KB instruction cache and a 64-KB to 256-KB, 4-way set associative, data cache.

   Many sequential programs, including STAP, were not specifically written to take advantage of the above features in modern processors. However, the powerful optimizations of the IBM C compiler enabled the POWER2 to achieve a STAP performance up to 100 Mflop/s per node. The highest performance (199 Mflop/s, double precision) is achieved for the Beamforming step in APT. Aggarwal et al. have shown [2] that performance can be substantially improved if the above features are consciously exploited in developing sequential programs. They discussed several concrete techniques to improve the ESSL routines [22] to reach 100 to 250 Mflop/s on a POWER2.

2) **Grain Size and DOP:** 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 re-
duction 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. Such high communication cost implies that only coarse-grain parallelism should be exploited on SP2. For nominal data sets, the coarse-grain approach is sufficient to achieve the desired 10 Gflop/s performance in most STAP benchmarks. 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 × EL = 8,192 in the APT program, RNG × EL = 49,156 in the HO-FD program, and DIM1 × DIM2 = 8,192 in the GEN program.

3) **Programmability of the SP2:** Programming distributed-memory, message-passing MPPs is a nontrivial task. Three features in the SP2 make programming easier: Firstly, the multistage HPS enables one to virtually treat SP2 as a fully-connected network. One does not need to worry about the physical locations of the nodes. Secondly, the MPL provides both point-to-point and a rich set of collective communication routines, which offers great help in developing the parallel STAP programs. Our STAP experiments used the following MPL communications routines, in decreasing order of importance: barrier, total exchange, reduction, shift, broadcast, blocking send/receive, and gather. We did find some important functionalities missing in MPL, which could have further improved the parallel STAP codes. The most important two are the support for atomicity and the support for eureka. Both features could help improve the Target Detection program in APT and HO-FD programs. Eureka refers to the capability for one task to asynchronously notify other tasks that something happened. This is available in Cray T3D architecture [1].

But the third and the most beneficial feature for program development is the network clustering concept, which manifests in SP2 in the following ways:

- Each node is a full-fledged workstation, with its own disk and a complete, multi-user, multi-tasking OS. Multiple tasks, either from the same application or from different users, can be executed on the same node at the same time.
- An application can be invoked from a host to run on multiple hosts, where a host can be either an SP2 node or a RS/6000 workstation anywhere on the internet.
- The hosts are connected, either through Ethernet or the HPS.
- During an execution, all I/O and error messages can be automatically routed to the invoking host, with a tag showing the originating host.

This network clustering concept facilitates parallel program development. One can first develop a parallel code on a local PowerPC 601 workstation. Because the limited memory, the parallel code might be a scaled-down one. Then one debugs and tests the exactly same code on the SP2 in interactive mode and extend it to a full-scale code (by simply changing a few constant values in a header file) to test on larger configurations. The resource is almost always available, due to the multi-user feature. After the code is fully debugged, the production code can run in the batch-mode through the Loadleveler, or even in a dedicated mode.

4) **STAP Code Complexity:** The sequential STAP benchmark suite is not a big code, containing a little over 4000 lines. The control flows of the programs are relatively clear. There are a few conditional statements (if, while, etc.) and gotos, but most branches are for loops. Still, the cyclomatic complexities of most modules are more than 10, with the highest being 29. But the biggest problem we have encountered is the existence of a large number of related variables and constants. This is due to the desire to model a wide range of radar configurations. In fact, 84% of APT modules, 94% of HO-FD modules, and all of GEN modules exceed the control or the data complexity thresholds set by MIT Lincoln Laboratory [21]. The high cyclomatic complexity and the large number of variables makes dependence analysis and parallelism extraction difficult. While the current parallel STAP suite can not lead to any general conclusions, our experience indicates that sequential programs with lower control and data complexity tend to be easier to analyze, and the data complexity seems to be the more important software complexity metric.

5) **Computation-to-Communication Ratio:** Higher computation-to-communication ratio implies higher speedup. The HO-FD 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 all STAP programs except the Linear Algebra code, 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-FD, the ratio is 254.

6) **Computational Paradigm:** A Phased SPMD paradigm similar to Vassiliad's BSP [24] is sufficient for parallelizing the STAP benchmarks on SP2. Although other modes and paradigms can be used on the SP2, using the SPMD mode and the phased paradigm simplifies program development and performance analysis. The STAP programs achieved good performance on the SP2 as summarized in Table 6. With 256 nodes, we have achieved 9 Gflop/s for the APT benchmark and 23 Gflop/s for the HO-FD benchmark. For the General benchmark, we have achieved 10.2, 4.5, 2.5, and 2.8 Gflop/s for the Sorting, the FFT, the Vector Multiply, and the Linear Algebra steps, respectively. These measured results closely match with the performance predicted before the execution runs.
APPENDIX

PARALLEL HO-PD PROGRAM SKELETON

```c
COMPLEX data_cube[PRI][PRI];
COMPLEX twiddle[PRI];
COMPLEX compute_twiddle_factor_OUT[];
COMPLEX compute_twiddle_factor_IN[];
COMPLEX tasks[];

for (i = 0; i < PRI; i++)
  /* Each task computes its own twiddle factor */
  for (k = 0; k < EL; k++)
    /* Each task computes EL PPTs */
    fft(inout, data_cube[i][j][k], twiddle, in PRI);

barrier

forall (i = 0; i < PRI; i++)
  /* Beamforming (BF) */
  beam_form(in_data_cube[i][j][k], in_data_cube[i-1 % PRI][j][k], in_data_cube[i+1 % PRI][j][k], out_data_cube[i][j][k]);

barrier

/* Target Detection (CFAR) */
forall (i = 0; i < PRI; i++)
  /* Target Report */
  m = 0;
  for (k = 0; k < RNG; k++)
    for (j = 0; j < BEAM; j++)
      if (isTarget(in data_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_report to target_report
```

ACKNOWLEDGMENTS

This work was supported in part by a research subcontract from MIT Lincoln Laboratory to the University of Southern California. We would like to thank David Martinez and Robert Bond at MIT Lincoln Laboratory for their support of this project. 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 Don 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.

We would like to thank the anonymous referees whose detailed comments helped improve the presentation of this article.

REFERENCES


Kai Hwang received the PhD degree in electrical and electronic engineering from the University of California at Berkeley in 1972. He is a chair professor of computer engineering at the University of Hong Kong, on leave from the University of Southern California. He has received several awards in recognition of his academic achievements and scientific contributions in the computing field.

An IEEE Fellow, Dr. Hwang specializes in computer architecture, digital arithmetic, and parallel processing. He has published more than 140 scientific papers and five books in these areas. He has been a Distinguished Visitor of the IEEE Computer Society, a member of the ACM SigArch Board of Directors, and a coeditor-in-chief of the Journal of Parallel and Distributed Computing since 1983.

While at USC, Dr. Hwang led a parallel processing research group performing the STAP benchmark evaluation of three massively parallel processors, namely the IBM SP2, Cray T3D, and Intel Paragon. He currently heads a research center program at HKU which is developing a cluster of SMP servers and workstations and corresponding network interfaces and software support for scalable supercomputing and distributed multimedia applications.

Zhiwei Xu is received the PhD degree from the University of Southern California in 1987. He is a professor and chief architect at the National Center for Intelligent and Computing Systems, Chinese Academy of Sciences. Prior to joining the Chinese Academy of Sciences, he was engaged in the MIT/STAP benchmark experiments on the IBM SP2, Cray T3D, and Intel Paragon, while visiting Dr. Hwang's research group at USC from 1994 to 1996. He taught at Rutgers University and New York Polytechnic University from 1987 to 1994.

Dr. Xu's current research interests include parallel computer architectures and their programming requirements.

Masahiro Arakawa received the SB degree in electrical engineering and science from the Massachusetts Institute of Technology in 1993 and the MS degree in computer engineering from the University of Southern California in 1995. He is currently a staff member at MIT Lincoln Laboratory in Lexington, Massachusetts. His current research interests include massively parallel architectures and algorithms.