# Early Prediction of MPP Performance by Workload and Overhead Quantification: A Case Study of the IBM SP2 System

Zhiwei Xu and Kai Hwang

CENG Technical Report 95-15

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

# Early Prediction of MPP Performance by Workload and Overhead Quantification: A Case Study of the IBM SP2 System*

Zhiwei Xu and Kai Hwang
Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, CA 90089-2562
{kaihwang, zxu}@aloha.usc.edu

## Abstract

Software development for MPP applications is often very costly. The high cost is partially caused by lack of early prediction of MPP performance. The program development cycle may iterate many times before achieving the desired performance level. In this paper, we present an early prediction scheme to reduce the high cost of iterative software development. Using workload analysis and overhead estimation, one can optimize the parallel algorithm design, before entering the coding, debugging, and testing cycle for application code development.

We have tested the effectiveness of the early performance prediction scheme by running the MIT/STAP benchmark programs on a 400-node IBM SP2 system in Maui High-Performance Computing Center. Our prediction is shown rather accurate within 90% of the actual performance measured on the SP2. The scheme can be implemented on any scalable MPPs such as Cray T3D, Intel Paragon and IBM SP2. The scheme is applied at the user/programmer level. The main contribution of this work lies in providing a systematic procedure to estimate the computational workload, to determine the application attributes, and to reveal the parallelism/communication overhead based on MPP hardware/environment characteristics, which should be supplied by the MPP vendors or by major MPP user groups.

---

* Manuscript submitted to *Parallel Computing,* Special Issue on Multiprocessor Performance Evaluation, April 1996. 

Table of Contents

## 1. Introduction

In developing applications on MPPs, a user is most concerned with the performance of the parallel code. Two approaches to evaluating MPP performance are depicted in Fig. 1. The traditional approach relies on the actual evaluation of the parallel program after the code is developed, debugged, and tested, as depicted by dashed lines in Fig. 1. The *early prediction* approach, shown by solid arcs in Fig.1, has an added step to prevent excessive software development cost.

With the traditional method, the user needs to select a parallel algorithm and implement it as a parallel program through many iterative coding and debugging steps. Then the code is executed and performance data gathered. If the evaluation finds out that the code does not meet the performance goal, the algorithm needs to be redesigned, and the complete software cycle repeated. The performance evaluation is done at the very end of the software development cycle. Therefore, this approach often results in an expensive cost repeating the full software development cycle.

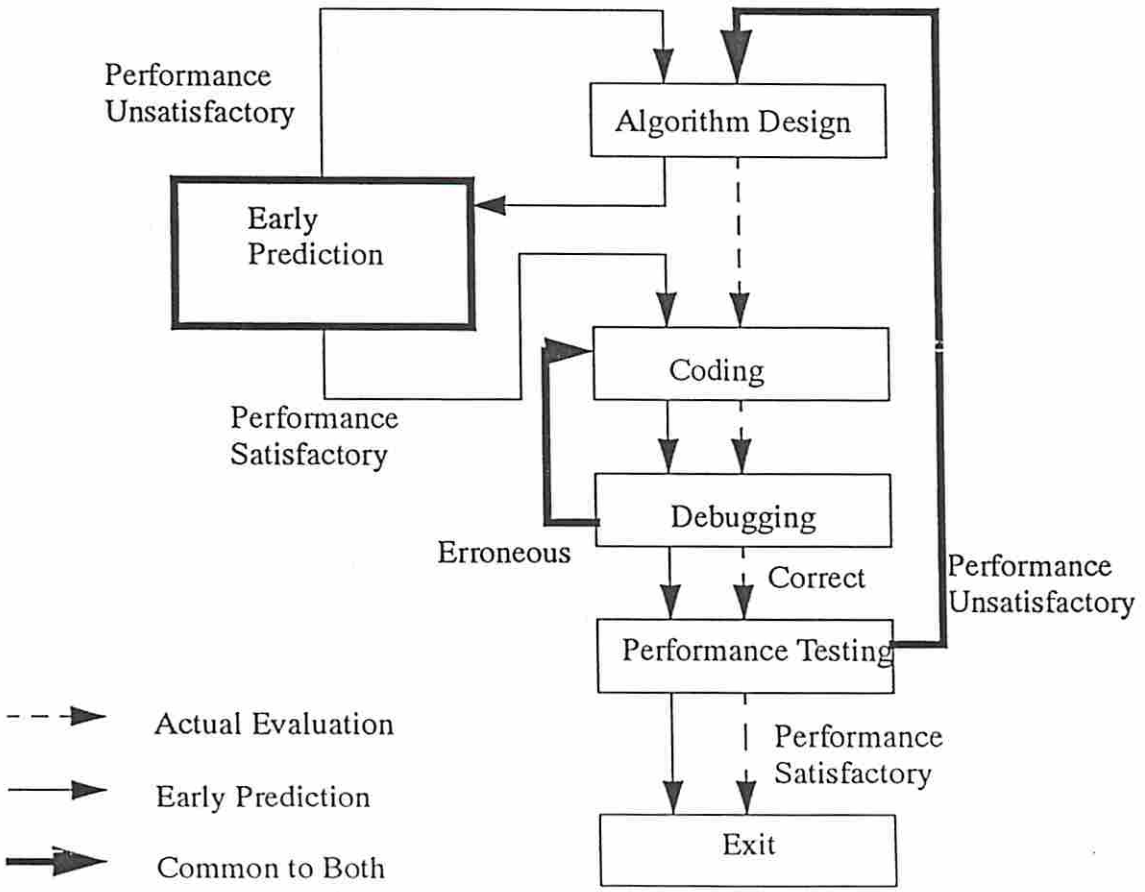


**Figure 1. Early prediction and actual evaluation of MPP performance**

Let us consider a common scenario in MPP application development: The user is given a parallel algorithm derived from a large sequential code with a computational workload $O(n^3)$, where $n$ is the problem size. The user is told that the algorithm has a time complexity of

$$T = O(n^3/N) + T_{comm} \quad (1)$$

if executed on an MPP with $n$ nodes, where $T_{comm}$ is the time delay caused by communication overhead. The user further learns from an MPP vendor that their machine supports communication efficiently. The user also sees some benchmark data showing that one node of the MPP is 3 times faster than his workstation. Encouraged as such, the user decides to implement the parallel algorithm code on the MPP. He wants to achieve a 500 folds speedup over his workstation code by using 256 MPP nodes, leaving some margin for slowdown caused by overhead.

After spending several months coding and debugging the parallel program and hundreds of thousands dollars in labor cost and parallel computer time, the user is faced with a disappointment: The application achieved a speedup of only 100 times! "Why is my code so slow?" has become a frequently asked question among MPP users. After further analysis aided by some performance profiling and monitoring tools, the user finally finds out that the original parallel algorithm was poorly designed for this MPP.

All the unnecessary software development costs could be avoided if the user had a method of *early performance prediction*. In this paper, we present a systematic method allowing the user to predict MPP performance at algorithm design time, before a single line of parallel code is written. The user can now quickly rule out inefficient algorithms, before costly coding and debugging begin. This method is based on workload analysis, done once per application, and overhead estimation, done once per MPP platform. It is specially designed for performance prediction of MPPs, where communication overhead can significantly degrade the performance.

Our early prediction method is validated by a sequence of benchmark experiments on the 400-node IBM SP2 supercomputer at Maui High-Performance Computing Center (MHPCC), using the STAP (*Space-Time Adaptive Processing*) benchmark suite developed by MIT Lincoln Laboratory for adaptive radar signal processing. For up to 256 nodes, our method generates performance predictions matching closely with the measured SP2 performance.

There has been much research in performance evaluation of parallel algorithms and parallel computers. Four common approaches are summarized below, from an MPP user's perspective:

(1). *Performance Metrics*: Many performance metrics have been proposed, such as execution time, Mflop/s speed, speedup, efficiency, critical path, and average parallelism [1,4,8,10]. However, on MPPs, an accurate estimation of these metrics depends on fundamental overhead metrics, which are much less developed. For instance, to decide whether to use a synchronous or asynchronous parallel algorithm, the user likes to know how much time a collective communication operation (e.g., a barrier) would take. Hockney's work [8,9] is very relevant in this regard.

(2) *Benchmarks*: Several benchmarks have been developed, such as the NAS[3], LFK[14], LINPACK[6], GENESIS[1], and STAP[5,16]. With a few exceptions, these benchmarks are designed to evaluate different computer system designs. These benchmarks are valuable for comparing various architectures and compilers to identify their relative strength and weakness. They are also useful for deciding which system is likely to satisfy a user's performance requirement. However, they offer little help to a user of a specific MPP on how to parallelize his application and how to make trade-off between communication and computation.

(3) *Performance Tools*: There are software tools for profiling and monitoring the performance of parallel programs. For instance, the *Visualization Tool* (VT) on IBM SP2 allows the user to watch his program progress graphically. This could be a great help to identify performance bottlenecks. However, these are *run-time* tools requiring that a parallel program be already developed. They are not very useful for early performance prediction. Our method complements such tools.

(4) *Algorithmic Complexity*. This traditional computer science approach does help predict performance at program design time. However, it is often too abstract, omitting many "details" that are important to an MPP user. Let us look at equation (1). The overhead term $T_{comm}$ is often not explicitly expressed, because it is architecture dependent, or it is expressed in a big-O notation, say, $T_{comm}=O(\log N)$. But the user needs to know the constant value, as $0.2\log N$ could be small but $20\log N$ could be large enough to demand an algorithm redesign.

The proposed *early prediction* scheme is illustrated in Figure 2. The process consists of 5 major steps. Details will be given in subsequent sections. Avoiding repeated coding and debugging, this scheme could reduce the program development time significantly. For instance, an APT program in the STAP benchmark was parallelized in two man-months without early prediction. Another HO program, even more complex than APT, was parallelized in 0.7 man-month with

early prediction. In predicting the STAP performance on the SP2, most prediction results differ by less than 10% from the measured performance results.

The proposed method relies on an accurate characterization of the computation workload and of the communication overhead. The workload is determined by user's applications and by the MPP platform. The overhead characterization depends on the MPP platform alone, and only needs to be done once per MPP platform. In section 2, we discuss how to characterize workload. The performance metrics are defined accordingly. In Section 3, we identify different sources of overhead and offer some suggestions to characterize them with a high degree of accuracy. In section 4, we characterize the overhead of SP2. In Section 5 we show how this method is applied to predict the performance of the STAP benchmark on the SP2. In section 6, we validate the method by comparing the predicted results with the measured results.

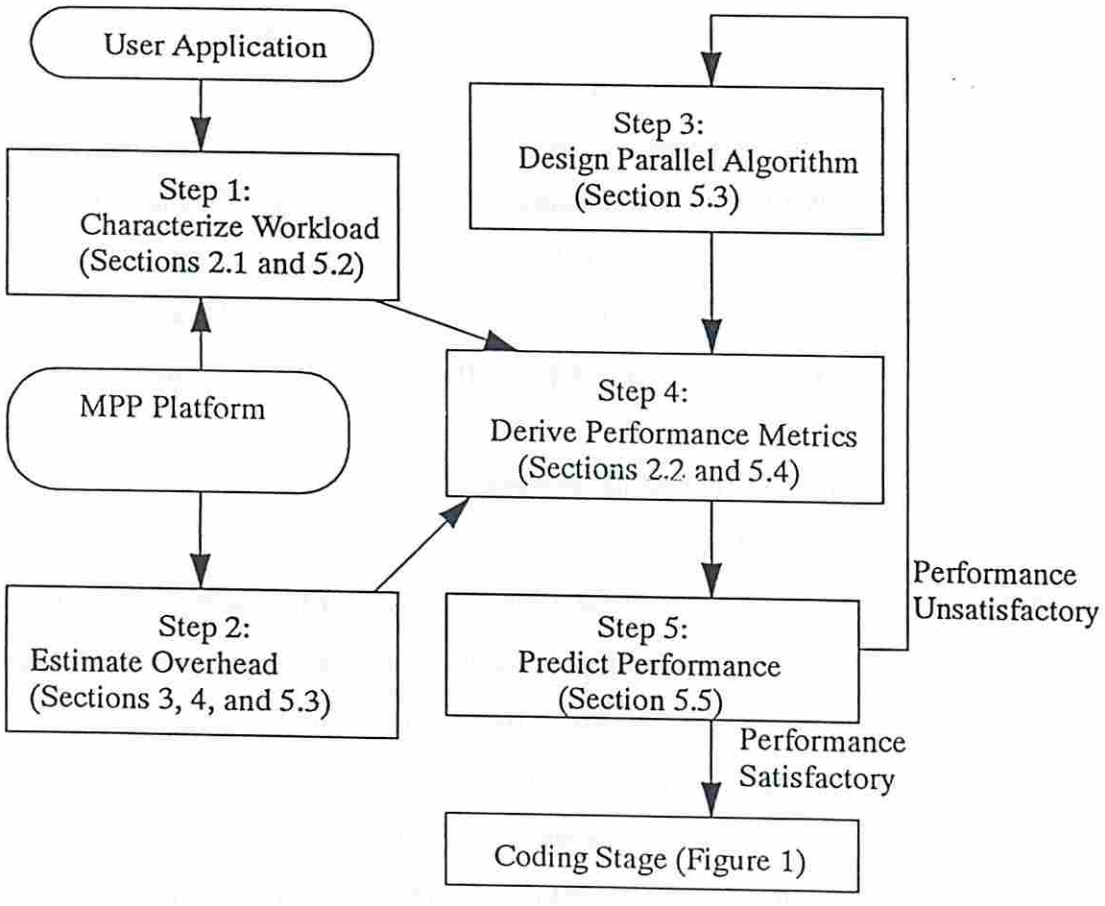


**Figure 2. The early performance prediction process to be specified in various sections**

## 2. Workload and Performance Metrics

For simplicity, we assume that the user starts with a sequential program $C$ consisting of a sequence of $k$ major component computations, $C_1, C_2, \ldots, C_k$. The purpose is to develop an efficient parallel program from $C$ by exploiting each individual component $C_i$ with a *degree of parallelism* $DOP_i$. We first discuss how to estimate the computational workload based on algorithmic and machine characteristics. Then we define the performance metrics to be used in the early prediction scheme.

### 2.1. Workload Characterization

We need to measure the amount of work performed in an application (referred to as *workload W*). For scientific computing and signal processing applications where numerical calculation dominates, 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 the computational speed, which is *Millions of flops per second*, denoted by Mflop/s. This notation is from the Genesis benchmark [1].

To use this metric meaningfully, we must follow some rules in counting the flops. Table 1 lists some of the rules used in our work, which resemble those in popular benchmarks [1,3,5,6,14].

**Table 1. Rules for Counting Floating-Point Operations**

| Operation | Flop Count | Comments on Rules |
|---|---|---|
| A[2*i] = B[j-1] +1.5 * C - 2 ; | 3 | Add, subtract, or multiply each count as one<br>Index arithmetic not counted<br>Assignment not separately counted |
| X = Y; | 1 | An isolated assignment is counted as one |
| if (X>Y) Max=2.0*X; | 2 | A comparison is counted as one |
| X = (float) i + 3.0 ; | 2 | A type conversion is counted as one |
| X = Y / 3.0 + sqrt(Z) ; | 9 | Each division or square root is counted as four |
| X = sin(Y) - exp(Z) ; | 17 | Each exponential, sine, etc. is counted as eight |

When the application program $C$ is simple and its workload is not data-dependent, the workload of $C$ can be determined by code inspection. When the application code is complex, or when the workload varies with different input data (e.g., sorting and searching programs), one can measures the workload by running the application on a specific machine, which will generates a report of the number of flops actually executed. This approach has be used in the NAS benchmark[3], where the flop count is determined by an execution run on a Cray Y-MP. Throughout the rest of this paper, we denote by $W_i$ the workload for program component $C_i$ and $W=W_1+...+W_k$ the total workload for the entire program $C$.

It is important that a single workload should be used consistently in predicting the performance of an application, even though some parallel algorithms need to perform extra flops to solve the same problem.

In theoretical performance analysis, it is often assumed that every flop takes the same amount of time. This *uniform speed* assumption does not hold in real MPPs. For instance, on a single SP2 node, we observed that the speed varies from 5 Mflop/s to 200 Mflop/s. Thus sequential execution time is also used to complement flop count. $T_1(i)$ denotes the sequential time for executing $C_i$. The single node execution time $T_1$ is equal to the sum of $T_1(i)$ for all $C_i$. The sequential *sustained speed* for each component and for the entire program $C$ can then be computed by: $P_1(i) = W_i/T_1(i)$, and $P_1 = W/T_1$.

### 2.2. Performance Metrics Used

There are three types of operations in a parallel program: *Computation* operations include arithmetic/logic, data-transfer, and control flow operations that can be found in a traditional sequential program. *Parallelism* operations are needed to manage user tasks, such as creation and termination, context-switching, and grouping. *Interaction* operations are needed to communicate and to synchronize among processes. The parallelism and the interaction operations are the sources of *overhead*, in that they need extra time to carry out besides the pure computational operations.

The $n$-node execution time $T_n$ is estimated by:

$$T_n = \sum_{1 \le i \le k} \frac{T_1(i)}{\min(DOP_i, n)} + T_{par} + T_{comm} \qquad (2)$$

where $T_{par}$ and $T_{comm}$ denote all parallelism and communication overheads, respectively. These overheads will be treated in Sections 3 and 4. The *sustained speed* using $n$ nodes is defined by $P_n=W/T_n$. The *speedup* is defined by $S_n = T_1 / T_n$ . The *efficiency* is $E_n = S_n / n = T_1 / (nT_n)$. The ratio of sustained speed to peak speed is called the *utilization*, denoted by $U_n = P_n / (nP_{peak})$, where $P_{peak}$ is the *peak performance* of one computing node. For instance, each node of an IBM SP2 has a peak speed of 266 Mflop/s. The utilization indicates how much percentage of the full computing power is utilized in a specific application. The utilization is always less than 100%.

There are several metrics of extreme values which give lower and upper bounds for $T_n$, $P_n$, and $S_n$. Let $T_\infty$ be the length of the *critical path*, which equals the time to execute an application using an unrestricted number of nodes, excluding all overhead [4]. From Eq. (2), $T_\infty$ is:

$$T_\infty = \sum_{1 \le i \le k} \frac{T_1(i)}{DOP_i} \tag{3}$$

The smallest $n$ to achieve $T_n = T_\infty$ is called the *maximal parallelism*, denoted by $N_{max}$. This is the maximal number of nodes that can be profitably used to reduce the execution time. This metric can be computed by $N_{max} = \max_{1 \le i \le k}(DOP_i)$ . The sustained speedup $P_n$ is upper bounded by the *maximal performance* $P_\infty = W/T_\infty$. The $n$-node execution time $T_n$ is lower bounded by $T_1/n$ and by $T_\infty$ . That is,

$$T_n \ge max\,(T_1/n, T_\infty) \ . \tag{4}$$

The *average parallelism* $T_1/T_\infty$, provides an upper bound on the speedup. That is, $S_n \le T_1/T_\infty$ .

## 3. Parallelism and Interaction Overheads

We use three metrics to measure the overhead incurred. The basic metric is *execution time* (in μs). Sometimes, the user may want to use the number of clock cycles, or the number of flops that can be executed by a single node in the same amount of time. For instance, the SP2 uses 66 MHz, 266 Mflop/s POWER2 nodes. A *send/receive* in SP2 takes at least 39 μs, which is equivalent to 2601 clocks, or 10374 flops.

### 3.1. Sources of Overhead

There are three sources of *parallelism overhead*, as summarized below:

- *Process Management*: This type includes process creation, termination, suspension, wakeup, context switching, etc.
- *Process Inquiry*: Also known as *process information*, this type of operations ask for information related to a process, such as process identification, the rank of the process in a group, the number of processes in a group, the group identification, etc.
- *Grouping*: A *process group* is a set of processes in a parallel program which are involved in *collective communications*. Different groups can be created at run time. A process can belong to different groups. A grouping operation is used to create or destruct a process group.

There are three sources of *interaction overhead*, as summarized below:

- *Synchronization*: This type of operations force a process to wait. Examples include *barrier* synchronization, *atomicity* and *mutual-exclusion* operations (e.g., locks, semaphores, and critical regions), and conditional synchronizations (e.g., events).
- *Aggregation*: This type of operations combine or aggregate partial data computed by individual processes into one or more results. Examples include reduction and parallel prefix (also know as scan).
- *Communication*: This type of operations pass data values among processes. Examples include *point-to-point communication*, *collective communication*, and *reading/writing of shared variables*.

MPP companies provide computational performance data in various forms, such as:

- The peak performance (MIPS or Mflop/s) per node, or per $n$-node system.
- The performance for frequently used computation kernels and benchmarks, e.g., FFT, LINPACK, and NAS.

In contrast, they rarely provide overhead data, except latency and bandwidth in point-to-point communications. It is very difficult to answer the questions: "How long does it take to create a process, to partition a process group, to do a barrier synchronization, or to perform a collective reduction?" on any given MPP. Sometimes, even the vendors themselves do not have the answers. The users often have to measure the overhead themselves, as we did on the SP2, which is a rather costly exercise.

### 3.2. Quantification of Overhead

On current MPPs, the overhead could be quite large and have significant impact on the performance of user applications. Table 2 shows the overhead of process management and communication on some Unix systems. The Process Creation column shows the time needed to fork an empty child process which does nothing but immediately exits. The Context Switching column shows the time needed to switch between two processes. The Pipe Latency column shows the round-trip time for two Unix processes to pass a small token back and forth through a pipe. The Pipe Bandwidth column shows the bandwidth achieved when a large message (50 MB) is passed between two processes through a pipe. The communication overhead is measured for two processes on the same node. The values could be much worse for processes on two separate processing nodes.

**Table 2. Overhead of Some Unix Operations (Source: McVoy [13])**

| Host Processor | Operating System | Process Creation (μs) | Context Switching (μs) | Pipe Latency (μs) | Pipe Bandwidth (MB/s) |
|---|---|---|---|---|---|
| POWER2 | AIX 3 | 1.4K | 21 | 138 | N/A |
| POWER | AIX 2 | 2.0K | 20 | 143 | 34 |
| Snake | HP-UX A.09 | 2.6K | 47 | 296 | 19 |
| IP22 | IRIX 5.3 | 3.1K | 40 | 131 | 34 |
| Pentium | Linux 1.1 | 3.3K | 66 | 157 | 13 |
| Alpha | OSF1 V2.1 | 4.8K | 25 | 185 | 32 |
| SS20 | SunOS 5.4 | 8.0K | 37 | 150 | 24 |

Often a large portion of the overhead is caused by OS kernel or system software cost. Consequently, the overhead could be quite different from one system to another, even when all the systems used the same processor architecture. For instance, the hardware latency of the *High-Performance Switch* in SP2 s less than 1 μs. But the latency jumps to several hundred μs if the processes communicate by using the kernel-level IP protocol. IBM has implemented a User-Space protocol to bypass the kernel, which drastically reduces the latency to as small as 39 μs.

Table 3 shows the overhead for process creation and synchronization in some current systems. Note that all three systems are based on SPARC processors. The Sun Microsystem Solaris operating system supports three types of processes: the heavy-weighted *Unix process*, the *light-weighted processes* (LWP), and user-level *threads*. The overhead of Unix processes are one to two orders of magnitude higher than those of user-level threads. Cilk is a user-level thread library developed at MIT[2]. The thread creation overhead is very small, only 50 clock cycles. The MIT Alewife is a research multiprocessor supporting multithreading, shared memory, and fine-grain synchronization. Its overhead is also small.

**Table 3. Overhead of Process Creation and Synchronization in MPP Systems**

| MPP System | Operation | Overhead |
|---|---|---|
| SMP Running Sun Solaris 2 [18] | Fork a Unix Process | 3,057 μs |
| | Create a Light-Weighted Process | 422 μs |
| | Create a Thread | 101 μs |
| | Process-Level Lock | 105 μs |
| | Thread Level Lock | 1.8 μs |
| The MIT Cilk Thread Library [4] | Create a Thread | 50 clocks |
| The MIT Alewife Architecture [2] | Unload Thread | 120-130 clocks |
| | Reload Thread | 90-100 clocks |
| | Read J-Structure (Success) | 2-6 clocks |
| | Read J-Structure (Failure) | 6-21 clocks |
| | Write J-Structure | 3-10 clocks |

These data are not meant to compare systems. Rather, we use them to demonstrate that the parallelism and interaction overheads could be very large, and the values vary greatly from one system to another. Users can not just extrapolate their past experience from a "similar" system to guess what the overhead will be. It is important to for the user to know the overhead values so as to avoid using expensive parallelism and interaction operations.

Numerous benchmarks and metrics for MPPs have been proposed [1,3,5,6,8]. But few provide estimation of overhead. To our knowledge, the only benchmarks that measure overhead in MPPs are the COMMS1, COMS2, and SYNCH1 in the GENESIS benchmark [1], which measure point-to-point communication and barrier synchronization for distributed memory MPPs. The only overhead metrics are the parameters $r_\infty$, $m_{1/2}$, $t_0$, and $\pi_0$, proposed by Hockney [8,9] for measuring point-to-point communication. Hockney also proposed two other metrics $f_{1/2}$ and $s_{1/2}$ to identify memory bottleneck and to estimate synchronization cost.

Parallelism and interaction overheads may degrade the performance of a user applications. There is an urgent need for the research community, together with the parallel computing industry, to develop a complete set of metrics to characterize these two types of overhead. It would be a great help to the MPP user community if parallel computer companies could provide values for these metrics and characterize overhead of their systems using close-form expressions.

It is important to decide on the forms of these expressions. General guidelines are suggested below. In the next section, we show how these guidelines are used to estimate the overhead on the IBM SP2.

(1) The overhead of a point-to-point or a collective communication operation can be expressed by an expression $t = t_0(n) + m/r_\infty(n)$, where $m$ is the message length and $n$ is the number of nodes involved in the operation. The exact forms of $t_0(n)$ and $r_\infty(n)$ depend on the specific MPP platform. The forms of $t_0(n)$ and $r_\infty(n)$ for SP2 are presented in Section 4. Detailed discussions of this overhead expression can be found in [20].

(2) On MPPs such as the Cray T3D, communication can be done either explicitly by message passing, or implicitly through shared memory. The shared memory communication overhead can be estimated as the time to load or store $m$ bytes to the memory hierarchy. The same linear expression of the form $t = t_0 + m/r_\infty$ can be used with different coefficients for the four cases: when the $m$-byte data is in the cache, in the local memory (i.e., on the same node), in a neighboring memory (on a neighbor node), and in a remote memory.

(3) The overhead for creating a process (task, thread) can be estimated by a linear function $t = cn + d$ or a log-linear expression $t = c\log n + d$. The constant $d$ represents a fixed cost incurred no matter how many (or few) processes are created. The constant $c$ represents a per-process additional cost. The type of processes should also be noted, e.g., heavy-weight or light-weight, kernel level or user level, local or remote. The group creation overhead can be similarly

estimated.

The parallel computer companies are best qualified to provide these expressions, because they are in the position to perform controlled experiments to measure and derive the fundamental metrics (i.e., the coefficients in the overhead expressions). However, if such information is not available, the user can measure various parallelism and interaction operations on the target MPP to collect raw overhead data, and then use least-square curve-fitting to derive the needed overhead expressions. These metrics and expressions need to be measured and derived only once for a given MPP platform, and used by the entire user community many times. In the next section, we show how to develop close-form overhead expressions for IBM SP2.

## 4. Overhead Estimation in the IBM SP2

The IBM SP2 allows the applications to use *static processes*: All the processes are created only once at the program load time. They stay alive until the entire application is completed. In all of our testing runs, we always assign one process to a node. The processes of a program can form different *groups*. We always use only one unique group: the group of all processes. Thus the group size is always equal to the number of processes in a program run, which in turn equals to the number of nodes used in that run. The number of nodes (processes) used, denoted by $n$, range from 1 to a maximum of 256. The message length is denoted by $m$ (bytes), ranging from 4B to 16 MB.

Since process creation and group creation are done only once, they do not contribute significantly to the total execution time. Therefore in performance prediction for SP2, we assumed that the parallelism overhead $T_{par} = 0$. Our measurement shows that for parallel STAP programs, $T_{par}$ ranges from 1000 μs to 10000 μs, which is negligible compared with the total execution time of the STAP programs of 0.5 seconds. In what follows, we concentrate on communication overhead. In message passing MPPs, all synchronization, communication, and aggregation are called *communication* operations.

In a *point-to-point* communication, one process sends a message to another process. Thus only two processes, one sender and one receiver, are involved. In a *collective computation*, a group of processes are involved to synchronize with one another or to aggregate partial results. The time for such an operation is a function of the group size, but not of message length, as the message length is usually fixed (i.e., $t = f(n)$). In a *collective communication*, a group of processes send message to one another, and the time is a function of both the message length and the group

size (i.e., $t = f(n, m)$).

### 4.1. Point-to-Point Communication

Using HPS, all nodes on an SP2 are equal distance away. The concept of neighboring nodes or remote nodes does not exist in SP2. Thus, the time for a point-to-point communication is a function of message length, not of the number of nodes. We measured the blocking send and blocking receive operations of the IBM MPL, using the *pingpong* [9] scheme in an $n$-process run: Process 0 executes a blocking send to send a message of $m$ bytes to process $n$-1, which executes a corresponding blocking receive. Then process $n$-1 immediately sends the same message back to process 0. All other processes do nothing. The total time for this pingpong operation is divided by 2 to get the point-to-point communication time. Some of the results are shown in Table 4, which confirms that there are only small differences for sending messages to different nodes.

**Table 4. Measured SP2 Point-to-Point Communication Overhead in Microseconds**

| $m$ \ $n$ | 2 | 8 | 32 | 128 |
|---|---|---|---|---|
| 4B | 46 | 47 | 48 | 48 |
| 1KB | 101 | 120 | 120 | 133 |
| 64KB | 1969 | 1948 | 1978 | 2215 |
| 4 MB | 1.2M | 1.2M | 1.2M | 1.2M |

The overhead as presented in Table 4 is not very convenient for the user. Ideally, the user would prefer to have a simple, close-form expression which can be used to compute the overhead for various message lengths. Such an expression has been suggested by Hockney for point-to-point communications [9]: The overhead is a linear function of the message length $m$ (in bytes):

$$t = t_0 + m/r_\infty \qquad (5)$$

where $t_0$ (called the *latency*, in μs) is the time needed to send a 0-byte message, and $r_\infty$ (called *asymptotic bandwidth*, in MB/s) is the bandwidth achieved when the message length $m$ approaches infinity.

Using least-square fitting of the measured timing data, we can express the point-to-point

communication overhead as a linear function: $t = 46 + 0.035m$. The best bandwidth achieved in our experiments is 35.54 MB/s, which is exactly what IBM reported. An optimistic user may want to use 1/35.54=0.028 to replace 0.035 and the latency 39 to replace the coefficient 46.

### 4.2. Collective Communication

In a *broadcast* operation, node 0 sends an $m$-byte message to all $n$ nodes. In a *gather* operation, node 0 receives an $m$-byte message from each of the $n$ nodes, so in the end $mn$ bytes are received by node 0. In a *scatter* operation, node 0 sends a distinct $m$-byte message to each of the $n$ nodes, so in the end $mn$ bytes are sent by node 0. In a *total exchange* (or *index*) operation, every 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, and node $n$-1 sends $m$ bytes back to node 0.

We have extended Hockney's expression (Eq. 5) as follows: The communication overhead $t$ is still a linear function of the message length $m$. However, the latency and the asymptotic bandwidth are now simple functions of the number of nodes $n$. In other words,

$$t = t_0(n) + m/r_\infty(n) \tag{6}$$

After fitting the measured timing data to different forms of $t_0(n)$ and $r_\infty(n)$, we derived the formulae for the five collective operations as shown in Table 5.

**Table 5. Collective Communication Overhead**

| Operation | Timing Formula |
|---|---|
| Broadcast | $(52\log n) + (0.029\log n)m$ |
| Gather/Scatter | $(45\log n + 10) + (0.04n - 0.04)m$ |
| Total Exchange | $80\log n + (0.03n^{1.29})m$ |
| Circular Shift | $(6\log n + 60) + (0.003\log n + 0.04)\,m$ |

### 4.3. Collective Computation

We measured three representative collective computation operations: *barrier*, *reduction* and *parallel prefix* (also known as *scan*). The curve-fitted communication overhead expressions are shown in Table 6. Note that over 256 nodes, the barrier overhead is 768 μs, equivalent to the time

to execute as many as 768×266=202,692 flops. This answers the question: "Should I use a synchronous algorithm?" The answer is only when the grain size is large. That is, hundreds of thousands flops are executed before a barrier.

**Table 6. Collective Computation Overhead Expressions**

| Operation | Time Expression |
|---|---|
| Barrier | 94 log$n$+10 |
| Reduction | 50log$n$ + 16 |
| Parallel Prefix | 60 log$n$ -25 |

## 5. Benchmark Performance of the SP2

In this section, we discuss how to use the metrics developed in Sections 2 and 4 to predict the performance of the parallel STAP programs on SP2. The early predication procedure outlined in Fig.2 is specified with details. Then each prediction step is explained in subsequent sections.

### 5.1. The Early Prediction Procedure

The five early prediction steps in Fig. 2 are specified below for any MPP system. Applying these steps on the SP2 is discussed in subsequent subsections. The procedure is applied by the user at the parallel algorithm design time, without involving the expensive coding and debugging stages. The only programming effort involved is to time the sequential code on a single node. Recall that we assume the user starts with a sequential program $C$ consisting of a sequence of $k$ components $C_1, C_2, \ldots, C_k$.

*Step 1*: Determine the workload $W_i$ for each component $C_i$ and the total workload $W = \sum_{1 \le i \le k} W_i$. Time the sequential code $C$ on one node to determine the sequential times $T_1(i)$ for each $C_i$ and the total sequential time $T_1$. The sequential performance is computed by: $P_1(i) = W_i/T_1(i)$, and $P_1 = W/T_1$

*Step 2*: Characterize the overheads, as is done in Section 4 for SP2.

*Step 3*: Derive a preliminary parallel algorithm. Then analyze it to reveal the degree of parallelism $DOP_i$ for each $C_i$ and the maximal parallelism $N_{max}$ for the entire pro-

gram, where $N_{max} = \max_{1 \le i \le k}(DOP_i)$ .

*Step 4*: Derive the following performance metrics:

- Parallel time: $T_n = \sum_{1 \le i \le k} \frac{T_1(i)}{\min(DOP_i, n)} + T_{parallelism} + T_{interact}$
- Critical path: $T_\infty = \sum_{1 \le i \le k} \frac{T_1(i)}{\min(DOP_i, \infty)} = \sum_{1 \le i \le k} \frac{T_1(i)}{DOP_i}$
- Average parallelism: $T_1/T_\infty$ and maximal performance $P_\infty = W/T_\infty$
- Speedup: $S_n = T_1/T_n$ and sustained speed $P_n = W/T_n$
- Efficiency: $E_n = T_1/(nT_n)$ and utilization: $U_n = P_n/(nP_{peak})$

*Step 5*: Use these metrics to predict the performance. If the prediction shows promising performance, continue to coding and debugging. Otherwise, analyze the predicted performance results to reveal deficiencies in the parallel algorithm. Modify the algorithm and go to Step 3.

### 5.2. Sequential Performance

Table 7 shows the sequential performance of the STAP benchmark on a single SP2 node. The entries of Table 7 are obtained by applying those workload and performance formulae in Step 1. The workload values are obtained by inspecting the source STAP programs. The execution time values are from actual measurement of each of the component algorithms. For example, the APT program is divided into four component algorithms: *Doppler Processing* (DP), *Householder Transform* (HT), *Beamforming* (BF), and *Target Detection* (TD). Each component algorithm performs different amount of workload, thus resulting in different execution time and sustained speed. These differences reveal the whereabouts of the bottleneck computations in each benchmark program. These information items are useful to restructure the parallel algorithm in Step 3.

### 5.3. Overhead Estimation and Algorithm Design

These correspond to what needs to be done in Step 2 and Step 3 of the early-prediction process. The communication overhead of SP2 was already characterized in Section 4. This overhead may reveal the real bottleneck in an application. Therefore, the algorithm may require modification in order to minimize the communication overhead. When static processes are used to run the

Table 7. Sequential STAP Benchmark Performance

| Program Component | Workload (Mflop) | Execution Time (Seconds) | Sustained Speed (Mflop/s) | Utilization (%) |
|---|---|---|---|---|
| APT: DP | 84 | 4.12 | 20 | 7.65 |
| HT | 2.88 | 0.04 | 72 | 27.07 |
| BF | 1,314 | 9.64 | 136 | 51.22 |
| TD | 46 | 0.57 | 75 | 28.01 |
| HO: DP | 220 | 11.62 | 19 | 7.12 |
| BF | 12,618 | 118.82 | 106 | 39.92 |
| TD | 14 | 0.17 | 82 | 30.96 |
| GEN: SORT | 1,183 | 22.80 | 52 | 19.51 |
| FFT | 1,909 | 79.14 | 24 | 9.06 |
| VEC | 604 | 19.11 | 32 | 11.88 |
| LIN | 1,630 | 20.23 | 82 | 31.00 |

STAP programs on SP2, the parallelism overhead is much smaller than the message passing overhead encountered. Therefore, we can simply concentrate on the latter in restructuring the STAP algorithms.

The initial parallel STAP algorithm was converted from the sequential code using the following heuristics: We first use the peak performance metrics provided by IBM to find out how well the system supports fine-grained parallelism. The sequential code is then analyzed for data and control dependences to expose available parallelism (DOP) at the desired grain level.

To determine the suitable grain size, we define another metric, called the *system communication-computation ratio* (SCCR). This is the product of the per-node peak performance $P_{peak}$ and the latency $t_0$ for communicating a 0-byte message between two nodes. Each SP2 node has a peak computing rate of 266 Mflop/s, while the smallest application-level communication latency is 39 μs. Thus the SCCR is 266×39=10,374. In other words, a messaging communication is equivalent to performing 10,374 flops. Thus we are restricted to choosing a coarse-grain approach.

The STAP benchmarks were parallelized on the SP2. The STAP benchmark suite contains three programs: *Adaptive Processing Testbed* (APT), *High-Order Post-Doppler* (HO), and *General* (GEN), totaling more than 4,000 lines of C code. Using a coarse-grain approach, we parallelized only the main algorithm, not individual subroutines. This makes dependence analysis much easier, and a parallel algorithm can be quickly designed for each of the three benchmark programs. For instance, the structure of a parallel APT algorithm is shown in Fig. 3.

The DP step is distributed to up to 256 nodes. The *total exchange* step must be globally executed by all nodes., with an aggregated message length of 17 MB. The HT step is sequentially executed on a single node (DOP=1). The *broadcast* operation must send a 80KB message to all nodes. The BF step performs beamforming operations using up to 256 nodes. The TD step needs to track the targets on up to 256 nodes locally. Then all the local target reports are merged through the final collective *reduction* operation.

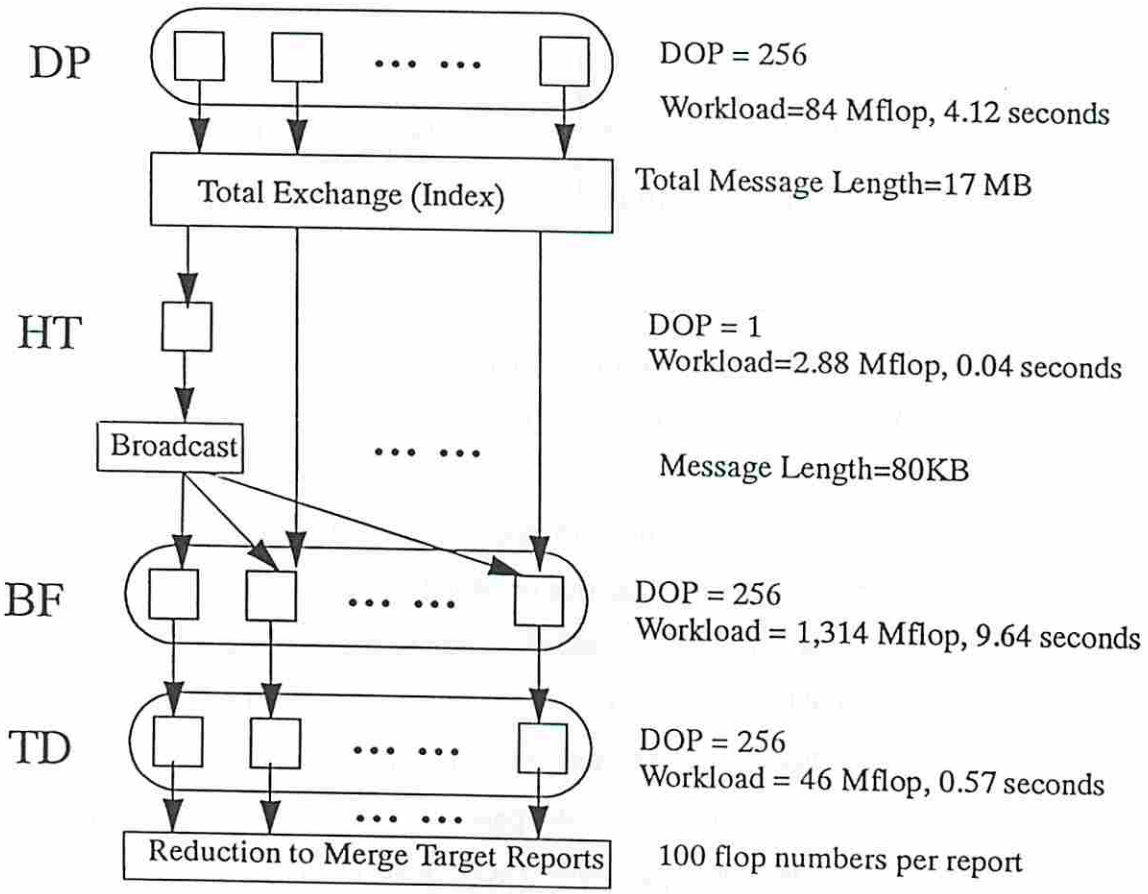


**Figure 3. Structure of A Coarse Grain Parallel APT Algorithm**

### 5.4. Performance Prediction

Table 8 shows the parallel execution times $T_n$ for the three STAP benchmarks, when all communication overheads are included. The communications components in these parallel time expressions are obtained by using the overhead formulae given in Section 4. From these expressions, it is straightforward to calculate the other performance metrics using the formulae in Section 5.1.

**Table 8. Execution Times of the STAP Benchmark over $n$ Nodes**

| Program | $n$-Node Execution Time $T_n$ in seconds |
|---|---|
| APT | $0.04 + \frac{14.33}{n} + 0.51n^{-071} + 0.004\log n$ |
| HO | $\frac{130.61}{n} + 1.5n^{-071} + 0.0044\log n + 0.0314$ |
| GEN | $\frac{121.05}{n} + 0.00188\log n + 6n^{-071} + \frac{20.23}{max\,(n, 32)} + 0.0016$ |

It is often helpful to perform a *zero-overhead prediction* first. Such an initial prediction is obtained from Table 8 by excluding all overhead. This prediction is useful to provide some upper and lower bounds on the projected MPP performance. Table 9 shows the zero-overhead performance prediction of all three parallel STAP benchmarks.

**Table 9. Early Prediction of STAP Performance on the SP2 Excluding All overhead**

| Program | Workload $W$ (Mflop/s) | Sequential Time $T_1$ (Seconds) | Maximal Performance $P_\infty$ (Gflop/s) | Average Parallelism $T_1/T_\infty$ | Critical Path $T_\infty$ (Seconds) |
|---|---|---|---|---|---|
| APT | 1,446 | 14.37 | 18 | 180 | 0.08 |
| HO | 12,852 | 130.61 | 28 | 281 | 0.46 |
| GEN | 5,326 | 141.28 | 8 | 220 | 0.64 |

The average parallelism metric provides a heuristic for answering the question: "How many

nodes should be used in my program?" A rule of thumb is to use no more than twice the average parallelism. When the number of nodes $n = 2T_1/T_\infty$, the efficiency is no more than 50%. Note that the *utilization* of the sequential STAP code is as low as 7.12% (Table 7). A 50% efficiency would drag the parallel utilization down to only 3.56%. On a 256-node SP2 with a maximum of 68 Gflop/s peak speed, only 2.4 Gflop/s can be achieved with such a low utilization.

Will the parallel algorithm just designed achieve a desired performance level? The answer depends on the performance requirement. Three requirements can be posed by a user:

- *Time*: For real-time, embedded applications, the user often does not care how much Mflop/s performance or how much speedup is achievable. All that matters is that the job is guaranteed to finish within a time limit. For instance, in airborne radar target tracking, the user may want to detect all targets within half of a second. From Table 9, the APT and the HO parallel programs have a chance to satisfy this requirement, as their critical paths are less than 0.5 seconds. However, the GEN program fails, no matter how many nodes are used.
- *Speed*: The STAP programs are *benchmarks*, in that they are not the real production programs. Rather, they characterize the real codes. What the user really wants is a STAP system that can deliver a sustained performance, say 10 sustained Gflop/s. From Table 9, Both APT and HO may meet the requirement but not the GEN program.
- *Speedup*: The user may want to see how much parallelism can be exploited in his application code to achieve, say 200 speedup minimum. By this standard, the APT program fails, because its average parallelism is only 180 (Table 9).

This preliminary performance evaluation generated some useful prediction: The parallel APT algorithm failed to achieve the 200 times speedup requirement, and the parallel GEN algorithm fails the other two requirements. There is no point to further develop the full-fledged GEN parallel code. Instead, we need to change the algorithm or the problem size.

Even when the preliminary prediction shows promising result (e.g., the HO program passes all three requirement tests), we should not rush to the coding stage, because we have not considered overhead yet. Using Table 8, one can perform a full-fledged prediction including all overhead. The entries of Table 10 are obtained from Table 8 for a 256-node SP2 system. These are early prediction results, yet to be validated against results obtained from real benchmark runs as

discussed in the next section.

**Table 10. Early Prediction of the STAP Performance on 256 SP2 Nodes Including All Overheads**

| Program | Parallel Time $T_{256}$ (Seconds) | Performance $P_{256}$ (Gflop/s) | Speedup $S_{256}$ | Efficiency $E_{256}(\%)$ | Utilization $U_{256}(\%)$ |
|---|---|---|---|---|---|
| APT | 0.125 | 11.5 | 114 | 45 | 17 |
| HO | 0.606 | 21.2 | 215 | 87 | 31 |
| GEN | 1.400 | 3.8 | 101 | 39 | 6 |

## 6. Validation of Early Prediction

Suppose the user changed the performance requirement to 100 times speedup. By Table 10, all three benchmarks can satisfy this new requirement by using 256 nodes, as the predicted speedups are 114, 215, and 101, respectively. We can then go ahead to code and debug the parallel programs, with high confidence that the final programs will meet the performance goal.

How accurate is the proposed performance prediction method? To answer this question, we have tested the actual parallel STAP codes on SP2 over up to 256 nodes. The sustained performance is compared with the prediction. The results are shown in Fig. 4. The prediction is very accurate. For the majority of cases, the error is less than 10%, and the maximal error is 22%. We also see that sometimes the predictions under-estimate the performance, i.e., the measured performance is *better* than the predicted. This is especially true for the GEN program. This is due to the fact that more cache and memory available as the machine size increases.

We also compared three prediction schemes: The *theoretical prediction* assumes a uniform sequential computation rate, i.e., every flop takes the same amount of time. The *zero-overhead* and the *full-overhead* predictions use the real speed values in Table 7. Both the zero-overhead and the theoretical predictions ignore the overhead. The projected speedup performance of the STAP benchmark is compared with the measured speedup. The results are shown in Fig.5. Overall the full-overhead prediction is more accurate than the other two prediction schemes, which exclude the overhead. Look at the case when the APT program is executed on 256 SP2 nodes. Fig.5 shows

that, when compared with the measured performance, the theoretical prediction has a 48% error. The 0-overhead prediction has a 41% error. But the full-overhead prediction brings the error down to only 15%.

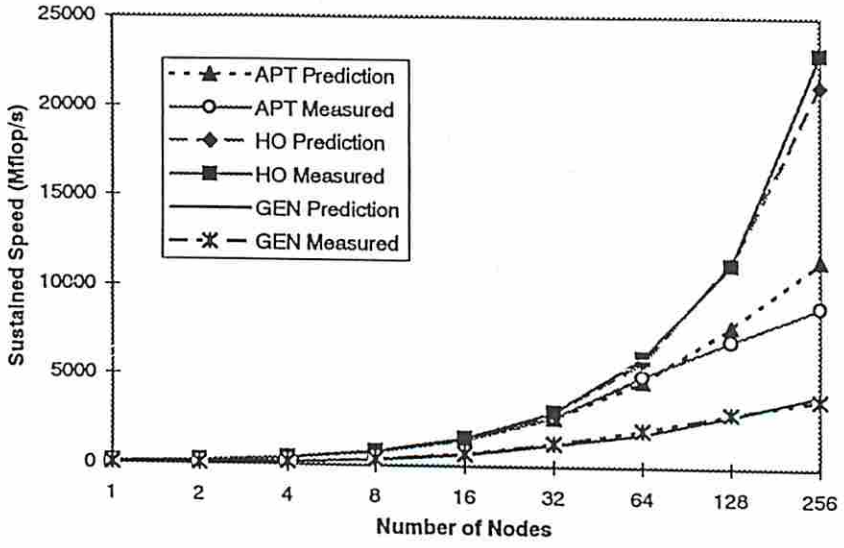


**Figure 4. Comparison of predicted and measured performance of the STAP programs**

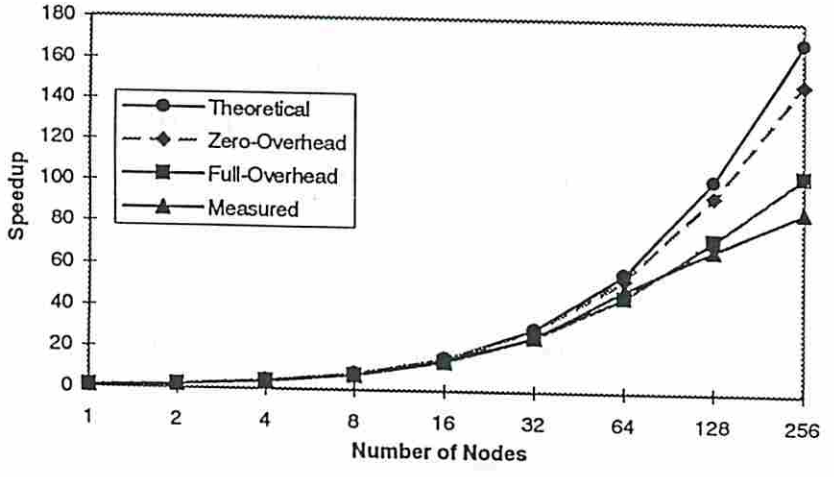


**Figure 5. Comparison of theoretical, zero-overhead, and full-overhead prediction schemes**

## 7. Conclusions

We have demonstrated the advantage of using early prediction to avoid unnecessary software development costs, by porting the STAP benchmark on the IBM SP2 system. The early-prediction procedure outlined in Fig.2 and detailed in Sections 2 to 5 is not difficult to implement on existing

MPPs, including Intel Paragon, IBM SP2, Cray T3D, and Convex SPP1. We have only tested the effectiveness of the prediction scheme on the SP2. We encourage other research groups to test it on other MPPs. That will further validate the usefulness of the idea of early performance prediction.

Continued research effort can be directed to study the scalability of MPPs by changing either the problem size, or the machine size, or both. Our STAP/SP2 benchmark performance report [11] shows good scalability of the STAP algorithm on various SP2 configurations. Another area of interest is to verify the isoefficiency of applications [7] using our early prediction scheme. In other words, the early-prediction will not only benefit algorithm designers, but also help develop production code for present or future MPPs.

**Acknowledgments**

We would like to thank David Martinez and Robert Bond at MIT Lincoln Laboratory for their support in this work. We are especially grateful to Peggy Williams, Rosanne Arnowitz, Blaise Barney, Tim Fahey, George Gusciora, and Lon Waters at Maui High-Performance Computing Center for their help. We thank Lionel Ni of Michigan State University, Craig Stunkel and Hubertus Franke of IBM for their helpful discussion on measuring the SP2 communication overhead. We thank Larry McVoy of SGI for providing the LMBENCH data on Unix platforms.

**References**

[1] C. Addison, *et al*, "The GENESIS Distributed Memory Benchmarks. Part 1: Methodology and General Relativity Benchmarks with Results for the SUPRENUM Computer", *Concurrency: Practice and Experience,* 5(1) (1993) 1-22.

[2] A. Agarwal, *et al*, "The MIT Alewife Machine: Architecture and Performance", MIT Laboratory for Computer Science, December 1994.

[3] D.H. Bailey, *et al*, "NAS Parallel Benchmark Results 3-94", NASA Ames Research Center Technical Report, March 1994

[4] R.D. Blumofe *et al*, "Cilk: An Efficient Multithreaded Runtime System", MIT Laboratory for Computer Science, December 1994.

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

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

[7] A.Y. Grama, A. Gupta, and V. Kumar, "Isoefficiency: Measuring the Scalability of Parallel Algorithms and Architectures", *IEEE Parallel & Distributed Technology*, 1(3)(1993)12-21.

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

[9] R. W. Hockney, "The Communication Challenge for MPP: Intel Paragon and Meiko CS-2", *Parallel Computing*, 20 (1994) 389-398.

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

[11] K. Hwang, Z. Xu, M. Arakawa, "STAP Benchmark Performance on the IBM SP2 Massively Parallel Processor", Technical Report, University of Southern California, Department of EE-Systems, Los Angeles, January 1995.

[12] IBM Corp., AIX Parallel Programming Environment, Pub. No. GH26-7229, GH26-7225, SH26-7230, SH26-7228, GA22-7137, 1994.

[13] L. McVoy, "LMBENCH: Portable Tools for Performance Analysis", SGI, Nov. 1994.

[14] F.H. McMahon, "The Livermore Fortran Kernels: A Computer Test of the Numerical Performance Range", Lawrence Livermore Laboratory Report UCRL-53745, December 1986.

[15] MHPCC, MHPCC 400-Node SP2 Environment, Maui High-Performance Computing Center, October 1994.

[16] MIT/LL, STAP Processor Benchmarks, MIT Lincoln Laboratories, February 28, 1994.

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

[18] Su9n Microsystem, Solaris Threads Home Page, http://www.sun.com/sunsoft/Products/Developer-products/sig/threads/faq.html.

[19] C. B. Stunkel, *et al*, "The SP2 Communication Subsystem", IBM T.J. Watson Research Center Report, August 1994.

[20] Z. Xu, K. Hwang, "Modeling Communication Overhead: MPI and MPL Performance on the IBM SP2 System", Technical Report, University of Southern California, Department of EE-Systems, Los Angeles, January 1995.