Early Prediction of MPP Performance: The SP2, T3D, and Paragon Experiences[1]

Zhiwei Xu *Chinese Academy of Sciences, Beijing, China*
Kai Hwang *University of Hong Kong, Pokfulam, Hong Kong*

Abstract

The performance of *Massively Parallel Processors* (MPPs) is attributed to a large number of machine and program factors. 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 for reducing the cost of application software development. Using workload analysis and overhead estimation, our scheme optimizes the design of parallel algorithm before entering the tedious coding, debugging, and testing cycle of the applications. The scheme is applied at user/ programmer level, not tied to any particular machine platform or to any specific software environment.

We have tested the effectiveness of this early performance prediction scheme by running the MIT/STAP benchmark programs on a 400-node IBM SP2 system at the Maui High-Performance Computing Centre (MHPCC), on a 400-node Intel Paragon system at the San Diego Supercomputing Centre (SDSC), and on a 128-node Cray T3D at the Cray Research Eagan Centre in Wisconsin.

Our prediction is shown rather accurate compared with the actual performance measured on these machines. We use the SP2 data to illustrate the early prediction scheme. We provide a systematic procedure to estimate the computational workload, to determine the application attributes, and to reveal the communication overhead in using MPPs. These results can be applied to develop any MPP applications other than STAP radar signal processing, from which this prediction scheme was developed.

Keywords: Massively Parallel Processors, STAP Benchmarks, Performance Prediction, Communication Overhead, Workload Quantification, and Parallel Processing.

1 Introduction

A frequently asked question among MPP users is “why is my application so slow?” It is not uncommon that an MPP user spends many frustrating hours in designing, coding, and debugging a parallel application, only to find out that it runs at a speed far below the performance level expected. The disappointing users may even loose confidence in using MPPs. This sentiment is evident in a recent European workshop on “Crisis in High Performance Computing” [7]. What users need is a simple and accurate performance prediction scheme, that can assess the performance before the coding stage and reveal performance bottlenecks before the run time. MPP users can apply this scheme to quickly ascertain the achievable performance and to identify which part of the program that real improvement can be made meaningfully. Three approaches to performance evaluation: *measurement*, *prediction* (modelling), and *simulation*, were reviewed in [4]. The first two approaches are combined and depicted in Fig.1.

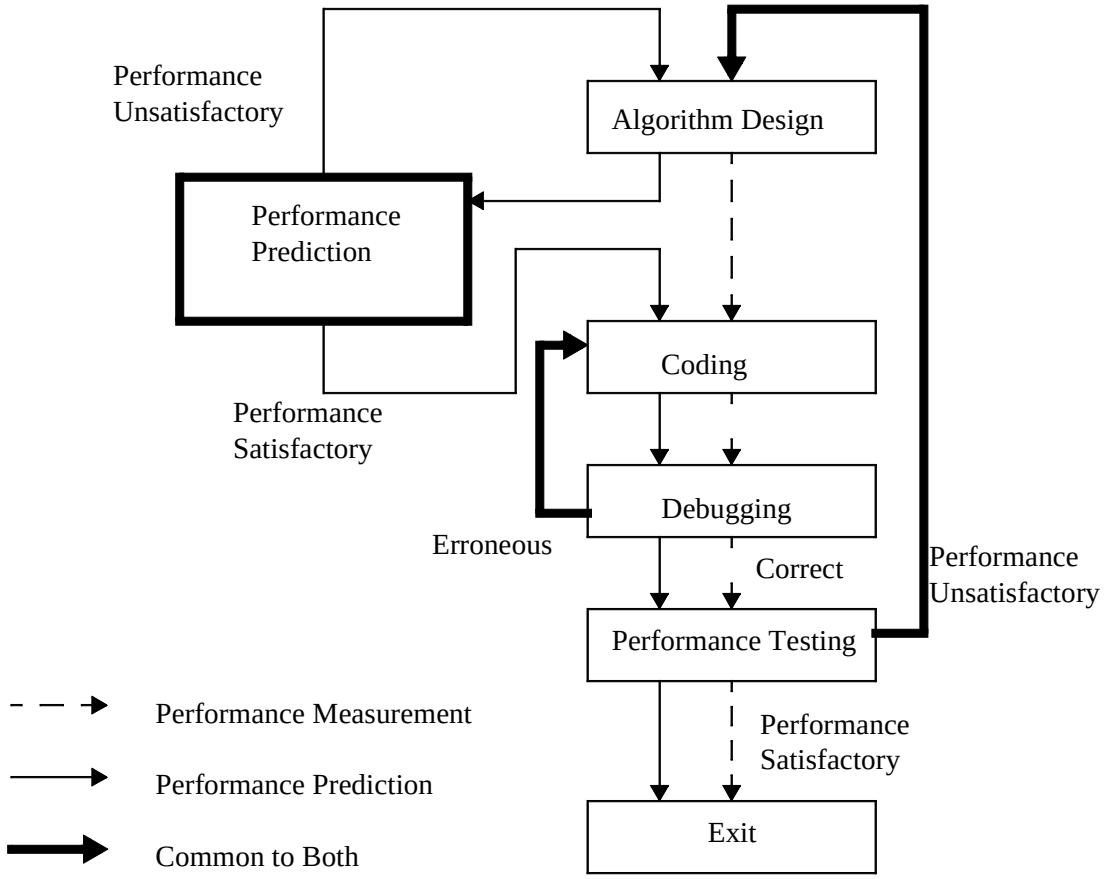


Figure 1: Comparison of prediction and measurement of MPP performance

The measurement approach is widely employed by MPP users, which relies on actually measuring the parallel program after the code is fully developed and debugged, as depicted by dashed lines in Fig. 1. The main advantage of the measurement approach is its accuracy. The *prediction* approach, shown by solid arcs in Fig.1, has an added step to rule out inefficient algorithms, before costly coding and debugging begin. This approach is not widely used, for the lack of good prediction schemes that can generate accurate performance results.

Currently, both measurement and prediction are done by MPP users manually. Although all MPP systems come up with some performance tools, virtually all of them are runtime support for performance evaluation [21]. Existing tools for performance prediction are still in the research stage, not yet available to general MPP users. For instance, Pease *et al* discussed an experimental system called PAWS (Parallel Assessment Window System) [22]. More recently, Fahringer described a tool called P^3T for performance estimation of Vienna Fortran programs on MPPs [9]. The recent joint issues of *IEEE Computer* [18] and *IEEE Parallel and Distributed Technology* [19] are dedicated to parallel performance tools.

In this paper, we address the performance prediction problem from a user's perspective. Our benchmark experiments were performed on the IBM SP2[20], Intel Paragon[24], and Cray T3D[1]. We parallelized the STAP (*Space-Time Adaptive Processor*) benchmark suite originally

written in sequential C language at MIT Lincoln Laboratory [5]. The contributions of this work is summarized below in three technical aspects:

- Users prefer a prediction scheme that can estimate the performance of a parallel application before a single line of parallel code is written. Our STAP experience shows that less than 10% time is spent on parallel algorithm design, and 90% time is spent on coding, debugging, and testing. As many as eight parallelizing strategies were considered in order to select the best one. Early prediction avoids the coding/debugging cost of the unselected Parallelization strategies.
- Most existing prediction schemes lack accuracy. This is mainly due to oversimplification in characterizing the workload and communication overheads. For instance, existing schemes usually assume every computational operation takes the same time, and estimate the overhead of a collective communication as the overhead sum of a sequence of point-to-point operations. We show quantitatively this does not work in real applications running on large number of processors.
- Our scheme is based on realistic quantification of workload and overhead, thus it is highly accurate. The scheme is validated by a sequence of STAP benchmark experiments on three MPPs. For up to 256 nodes, our method predict the MPP performance rather closely matching with the measured performance.

In summary, we attempt to answer the following five questions:

1. What performance can be achieved?
2. Is the parallelizing strategy a good one?
3. How many processors should be used?
4. How large a grain-size should be used?
5. Where is the performance bottleneck?

We start with what are required in a performance prediction scheme. We present the early prediction scheme in Section 3, using the STAP benchmark experiment on IBM SP2 as a running example. In section 4, we validate the method by comparing the predicted results with the measured results. In Section 5, we show how the prediction scheme can be used to increase system utilization. Finally, we comment on potential applications of the early performance prediction scheme

2 Performance Prediction Requirements

This section discusses what is required of a parallel performance *predictor*, either a manual method or an automatic tool, from an MPP user's perspective. We want to find out what should be the input to the predictor and what should be the output. We also comment on what is available to the user and what is lacking. These results are applied to in the next section to develop the early prediction scheme.

The proposed early prediction scheme is illustrated in Fig. 2. The scheme consists of 5 major steps. Details of each step will be given in subsequent sections. The scheme relies on an accurate characterization of the computation workload and of the communication overhead. The workload is determined from user's application and the MPP platform (Step 1). The overhead characterization depends on the MPP platform alone, and only needs to be done once per MPP platform (Step 2). Once the workload and the overhead are quantified, the scheme iterates from Steps 3 to 5 to predict the performance of each Parallelization strategy.

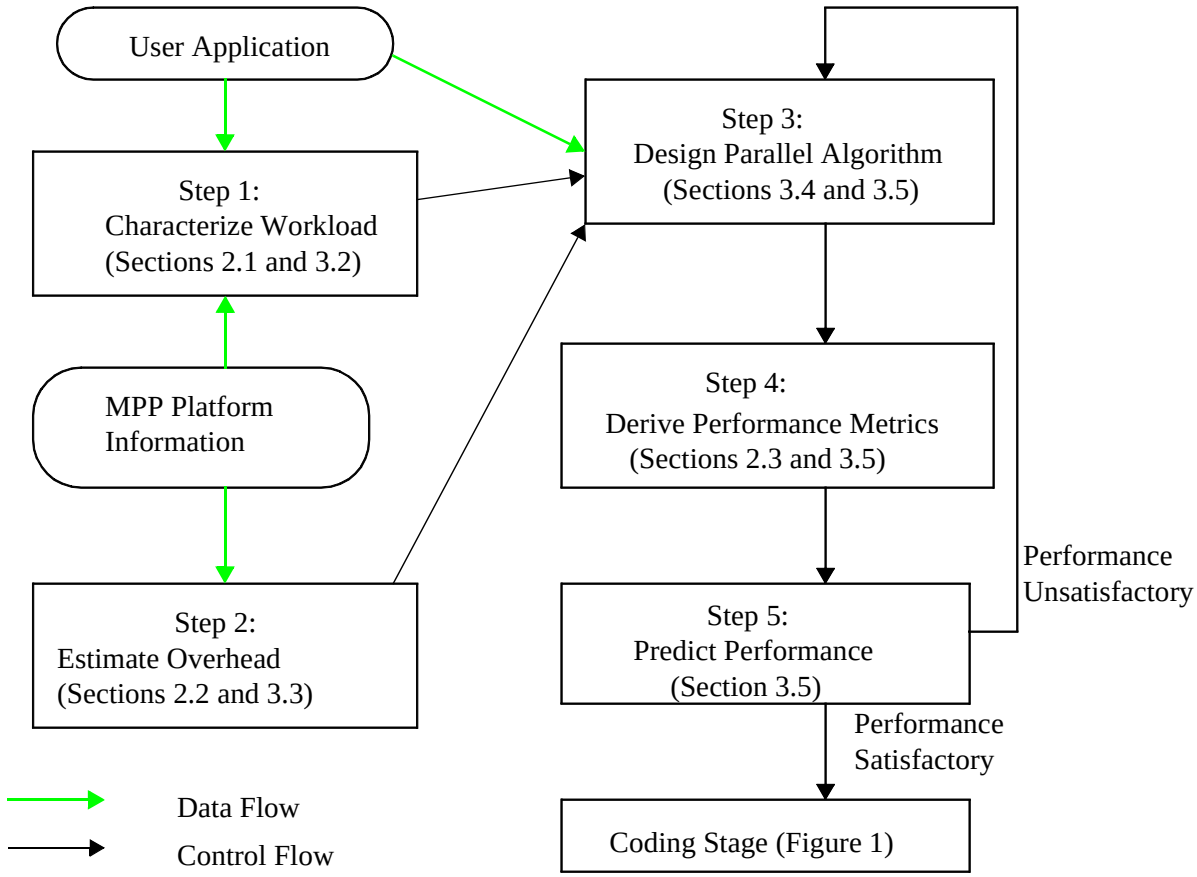


Figure 2: The early performance prediction process to be specified in various sections

2.1 User Applications

Any performance prediction is based on a specific user application and a specific MPP platform. But what should be the exact inputs from the application? The performance evaluation tool developed in [22] requires the application be specified as an ADA program or in an intermediate language. The P^3T evaluation reported in [9] requires a Vienna Fortran program. Some other predictors require a task graph, a dataflow graph, or a Petri net. All these schemes assume the existence of software tools, which are not available on current MPPs.

For most MPP users, either of he following may be initially available, when a parallel application is to be developed:

- A sequential program written in C or Fortran for a uniprocessor system. We will denote this sequential code by *C*.
- A number of parallel algorithms, in "paper-and-pencil" form but not yet implemented as parallel programs. These algorithms are most likely adopted from published articles which have not been tested in any computer platform.

A main objective of early performance prediction is to identify the best parallel algorithm. Some efficient and effective parallel algorithms will emerge after identifying the performance

bottlenecks. These algorithms can be generally structured by the following abstract model recently developed by the coauthors for programming MPPs or clusters of workstations [29].

2.1.1 The Phase Parallel Model

Consider a sequential program *C*. The program structure is divided into a sequence of *k phases*, $C_1, \quad , \quad , C_k$ as illustrated in Fig. 3. Each phase involves essentially a computational superstep consisting of coarse or medium grains of operations. Between two adjacent phases, an *interaction* (communication and synchronization) step is often needed. This could be collective communications involving multiple computing nodes.

Semantically, all operations in a phase followed by an interaction step must finish before the next phase can begin. Phase C_i has a computational *workload* of W_i million floating-point operations (Mflop), and takes $T_1(\ )$ seconds to execute on one processor. It has a *degree of parallelism* of DOP_i. In other words, when executing on *n* processors with $1 \leq \ \leq DOP_i$, the *parallel execution time* for phase C_i becomes $T_n(\ ) = T_1(\ )/n$. Associated with each phase, the user provides the predictor consisting of computations, workload, DOP, and interaction functions followed, etc.

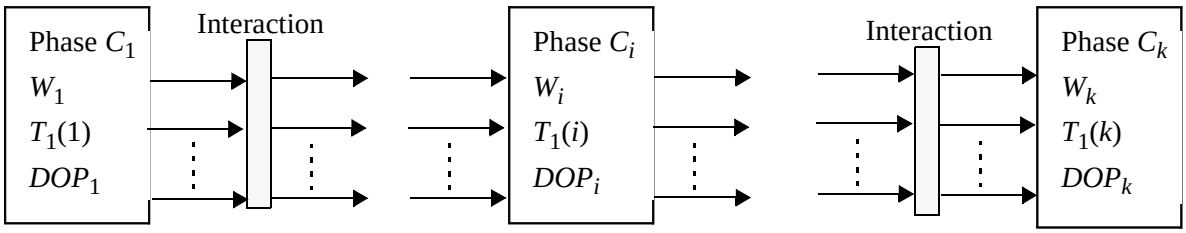


Figure 3: The Phase Parallel Model for MPP Application Code Development.

This model is especially efficient to implement SPMD (Single Program and Multiple Data streams) data-parallel programs. Depending on the interaction overhead encountered, the granularity of each phase can be adjusted to achieve a balance between computations and communications. This is crucial to apply the model for early performance prediction. The phase parallel model can cover several important parallel programming paradigms [6], such as the *synchronous iteration* model discussed in [3], the *loosely-synchronous parallelism* [10], and the *Bulk-Synchronous Parallel* (BSP) model [27]. We have applied this model to develop all the parallel STAP benchmark programs for the three MPPs being tested for performance evaluation.

This model is rather conservative in predicting the MPP performance, compared with those models allowing computations to be overlapped with interaction steps for latency hiding purposes. However, our model is easier to implement on MPPs. The model is also very effective to implement the pipelining operations required in [6].

2.1.2 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 PARKBENCH benchmark [13], and has become widely used.

When the application program C is simple and its workload is not input 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 measure the workload by running the sequential 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[23], 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 flop workload for computation step C_i and $W=W_1+...+W_k$ the total flop workload for the entire program C.

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 processor, we measured that the speed varies from 5 Mflop/s to 200 Mflop/s for different computation routines in the STAP benchmark suite, a difference of 40 times! Thus the sequential program is executed to generate the *execution time* rather the flop count. $T_1(i)$ denotes the sequential time for executing C_i. The single processor execution time T_1 is equal to the sum of $T_1(i)$ for all C_i. The sequential *speed* for each computation step C_i and for the entire sequential program C can then be computed by:

$$P_1(i) = W_i / T_1(i), \text{ and } P_1 = W / T_1.$$

Executing the sequential program on a single node may not be feasible due to two reasons: (1) The program needs more memory than what a single node can provide. Then the program will either not run at all, or execute at a very slow speed due to excessive paging. (2) Even when the program fits in a local memory, it may take a long time (e.g., days) to execute because of excessive computational complexity involved.

The problem can be solved by scaling down the program size *along the parallelizing dimension*. Scaling down along other dimensions may distort the workload information. For *domain-decomposition* parallel programs (i.e., where parallelism is exploited by partitioning the data domain into multiple chunks), the parallelizing dimension is the dimension along which the data domain is partitioned. There may be more than one parallelizing dimension.

2.2 Platform Parameters

The following performance related information is generally provided by MPP vendors:

- *Architecture Parameters*: These include the machine size n (i.e., the number of processors) and the capacity of memory/cache per processor.
- *Peak Performance Numbers*: These include the peak speed of a processor (denoted by P_{peak}), the smallest *latency* and the peak *bandwidth* for point-to-point communication.
- *Benchmark Performance Numbers*: These include the measured performance of some public domain benchmarks. For MPPs, the most frequently cited are the NAS benchmark [23] and the LINPACK benchmark [8]. Some vendors also have performance data for some sequential kernel routines (e.g., mathematical libraries).

The above information is listed in Table 1 for three MPPs. The machine size, the cache, and the memory attributes show the ranges of possible configurations. The latency and the bandwidth are the best numbers provided by the vendors. The NAS FT benchmark performances a 3-D FFT

based algorithm to solve a partial differential equation. The timing results are based on highly optimized executions, which a user is unlikely to achieve.

Table 1: Performance Attributes of Three Massively Parallel Processors

Performance Attribute	IBM SP2	Cray T3D	Intel Paragon
Machine Size (n)	8-512	4-2048	4-4096
Data Cache per Processor	64-256 KB	8 KB	16 KB
Memory per Processor	64 MB-2 GB	32-64 MB	16-128 MB
Peak Processor Speed (P_{peak})	267 Mflop/s	150 Mflop/s	100 Mflop/s
Point-to-Point Communication Latency and Bandwidth	39 μs, 35 MB/s	2 μs, 150 MB/s	30 μs, 175 MB/s
NAS FT Benchmark Time (using 128 Processors)	14.52-15.68 Seconds	20.68 Seconds	22.76-42.07 Seconds

Now this is about all platform information that is available to a user, which is too limited. In particular, no information is provided for various *overheads*. It is very difficult to answer simple questions such as "How long does it take to create a process, to partition a process group, to do a barrier synchronization, or to sum *n* values from *n* processors? "

There are three types of operations in a parallel program: *Computation* operations include arithmetic/logic, data-transfer, and control flow operations that are found in sequential programs. *Parallelism* operations are needed to manage user processes, 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.

On current MPPs, these overheads could be quite large and 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 for the user to know the overhead values so as to avoid using expensive parallelism and interaction operations.

2.2.1 Overhead Quantification

Numerous benchmarks and metrics for MPPs have been proposed [4, 8, 9, 12, 13, 14, 15, 26]. 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 PARKBENCH benchmark [13], which measure point-to-point communication and barrier synchronization for distributed memory MPPs. The only overhead metrics are the parameters r_∞, $m_{1/2}$, t_0, and π_0, proposed by Hockney [12,13] 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.

MPP user groups should develop closed-form expressions to quantify the communication overheads in using communications libraries or standards such as PVM, MPL, or MPI. We have attempted to do so for the IBM SP2 as reported in [28]. Our prediction scheme (Fig.2) includes an overhead quantification step. General guidelines for this step are suggested below.

For each interaction (or parallelism) operation to be quantified, the user measures the wall-clock time for a number of combinations of machine size *n* and message length *m*. Our experience show that for current MPPS, the machine size should vary from 2 to at least 128, and the message length for a communication operation should vary from 4 bytes to at least 64 KB. Otherwise it is difficult to derive accurate overhead expressions. Overhead measurements should be done in a batch mode, because this is the mode adopted by most production runs on MPPs.The measured timing data is then curve-fitted to obtain the closed-form performance expressions.

(1) The overhead of a point-to-point or a collective communication operation is expressed by $t = t_0(\ ) + m/r_\infty(\ )$, where *m* is the message length and *n* is the number of processors involved in the operation. The exact forms of $t_0(n)$ and $r_\infty(n)$ depend on the specific MPP platform and the communication operation. The forms of $t_0(n)$ and $r_\infty(n)$ for SP2 are presented in Section 3.3. As we will see there, $t_0(n)$ and $r_\infty(n)$ correspond to latency and bandwidth, respectively.

(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 neighbouring 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 $t = c\log\ + d$. 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 above method requires quantifying each operation individually to achieve accuracy. However, the overhead expressions need to be measured and derived only once for a given MPP platform, and used by the entire user community many times. A simpler method for quantifying communication overheads is to use vendor-supplied latency and bandwidth numbers for point-to-point operations (i.e., those in Table 1), and treat a collective operation as a sequence of point-to-point operations. The main problem is that the accuracy could be off significantly.
Performance Metrics

We discuss the performance metrics that are important to MPP users. In section 3, we discuss how to use them to answer practical performance questions. The following metrics definitions refer to Fig. 3. The sequential execution time T_1 and speed P_1 have already been defined in Section 2.1. Note that they should be measured by executing the best sequential program, not a parallel program, on a single processor. The *n*-processor execution time T_n of a parallel program is estimated by:

$$T_n = \sum_{1 \le \ \le k} \frac{T_{\cdot}(\ )}{\min(\qquad)} + \qquad + T_{interact} \qquad (1)$$

where $T_{parallel}$ and $T_{interact}$ denote all parallelism and communication overheads, respectively. The *speed* using n processors 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 speed to peak speed is called the *utilization*, denoted by $U_n = P_n / (nP_{peak})$, where P_{peak} is the *peak speed* of one processor. The utilization indicates how much percentage of the full computing power is utilized in a specific application.

There are several metrics of extreme values which give lower and upper bounds for T_n, P_n, and S_n. Let T_∞ be the length of the *critical path*, which equals the time to execute an application using an unrestricted number of processors, excluding all overhead. From Eq. 1, T_∞ is:

$$T_\infty = \sum_{1 \le \ \le k} \frac{T_i(\ )}{DOP_i} \qquad (2)$$

The smallest n to achieve $T_n = T_\infty$ is called the *maximal parallelism*, denoted by N_{max}. This is the maximal number of processors that can be profitably used to reduce the execution time. This metric can be computed by $N_{max} = \max_{1 \le \ \le} (\qquad)$. The speed P_n is upper bounded by the *maximal speed* $P_\infty = W / T_\infty$. The quantity T_1/T_∞ is called The *average parallelism*.

2.2.2 Which Metrics to Use?

Obviously, execution time and speed are important metrics. Some users also care about speedup. However, speedup and efficiency could give misleading information. In fact, with a bad sequential program, a poor parallel program can have a speedup greater than the machine size n and an efficiency greater than 100%. In contrast, the utilization is always less than 100%, and a better program always has better utilization.

The critical path and the average parallelism are easy to compute, as they ignore overhead. They bound the achievable performance and are useful at the initial parallel algorithm design stage. The n-processor execution time T_n is lower bounded by T_1/ n and by T_∞. That is

$$T_n \ge max(\qquad) \qquad (3)$$

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

The execution time, speed, and utilization are the most important performance metrics. Special attention should be paid to the utilization metric, which is often overlooked, but more informative than execution time and speed. A low utilization always indicates a poor program or compiler. In contrast, a good program could have a long execution time due to a large workload, or a low speed, due to a slow machine.

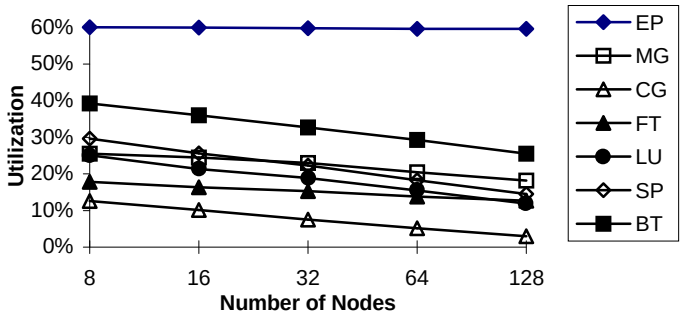
A sequential *application* executing on a single MPP processor has a utilization ranging from 5% to 40%, typically 8% to 25%. Some individual *subroutines* can be made faster to reach 75% or more. However, when such subroutines are incorporated into a real application, they do not

necessarily retain their high utilization. A *parallel application* executing on multiple processors has a utilization ranging from 1% to 35%, typically 4% to 20%.

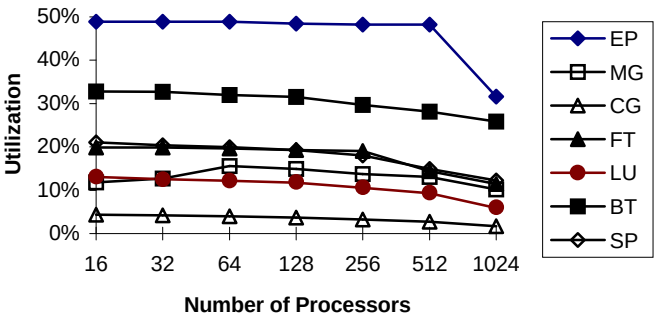
By processing the NAS parallel benchmark results [23] for SP2, T3D, and Paragon, we found that the utilization ranges from 1% to 60%, with a harmonic mean of 12%. Note that these results were generated from the vendors benchmark programs, which are often highly optimized. An ordinary user application of similar type is unlikely to attain the same performance.

Figure 4 shows the utilization of three MPPs in executing the NAS benchmarks (Class A). In general, the utilization of MPPs is lower than a traditional parallel vector supercomputer such as the Cray C90. The C90 achieved a utilization from 28% to 73%, with a harmonic mean of 53%. For everyday user applications, a C90 has a typical utilization of 21% to 27% [2].

(a) IBM SP2



(b) Cray T3D



(c) Intel Paragon

Data points not shown are unavailable

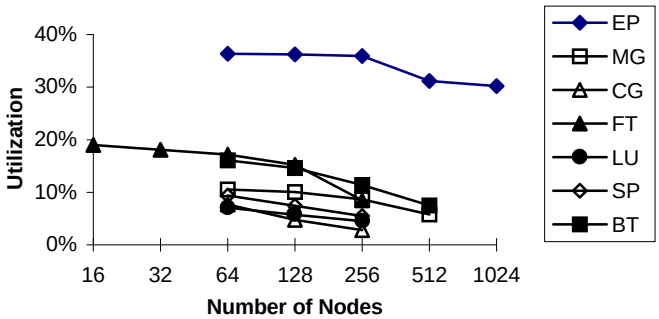


Figure 4: Utilization of three MPPs for NAS Parallel Benchmarks

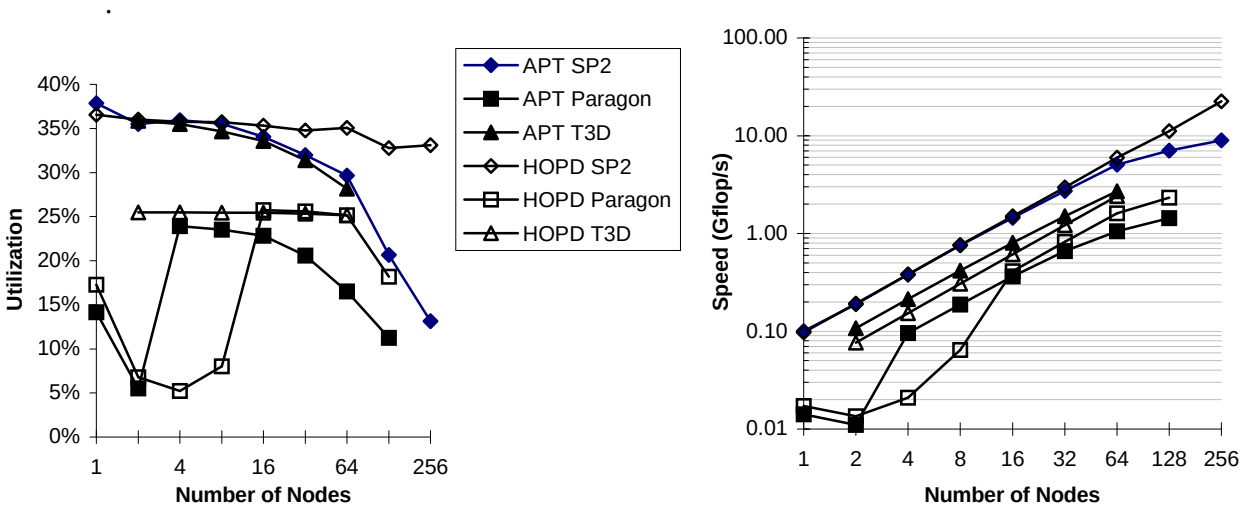


Figure 5: Measured utilization and Gflop/s speed of the parallel APT and HO-PD benchmark programs on three MPPs

In Fig. 5, we show the measured utilization and speed of two programs in the STAP benchmark suite, when executing on three MPPs. The utilization ranges from 5% to 38%.The utilization rates drop as more nodes are used in a given application. However, this may not be true in using a small number of nodes. This has happened in using the Paragon with less than 16 nodes. The main reason is that each Paragon node at the SDSC has only 16 MB of main memory, too small to fit the large data set we have. The dip and rise of the Paragon utilization curves was caused mainly by this memory problem.

3 STAP Benchmark Performance Results

The STAP benchmark suite was originally developed by MIT Lincoln Laboratory for adaptive radar signal processing. It contains three benchmarks: *Adaptive Processing Testbed* (APT), *High-Order Post-Doppler* (HO), and *General* (GEN). In this section, we use the performance prediction procedure outlined in Fig.2 to predict the performance of the parallel STAP programs on SP2. The prediction procedure is first summarize. Individual steps are explained in subsequent sections, using the APT benchmark as a working example.

3.1 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 processor. Recall that we assume the user starts with a sequential program *C* consisting of a sequence of *k* steps C_1, C_2, …, C_k, as shown in Fig.3.

***Step 1*:** Determine the workload W_i for each component C_i and the total workload $W = \sum_{1 \le \ \le k} W_i$. Time the sequential code C on one processor to determine the sequential times $T_1(i)$ for each C_i. The corresponding sequential speed and utilization are computed by: $P_1(\ ) = W_i / T_1(\ )$, and $U_1(\ ) = P_1(\ ) / P_{peak}$.

***Step 2*:** Quantify the overheads.

***Step 3*:** Derive a 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 program, where $N_{max} = \max_{1 \le \ \le}$ ( ).

***Step 4*:** Derive the following performance metrics:

- Parallel time: $T_n = \sum_{1 \le \ \le k} \frac{T_1(\ )}{\min(\ \ )} +$ [illegible] $+ T_{interact}$
- Critical path: $T_\infty = \sum_{1 \le \ \le k} \min(\ \ ) = \sum_{1 \le \ \le k} \frac{T_1(\ )}{DOP_i}$
- Average parallelism T_1 / T_∞ and maximal speed $P_\infty = W / T_\infty$
- Speedup $S_n = T_1 / T_n$ and speed $P_n = W / T_n$
- Efficiency $E_n = T_1 / (\ \ )$ and utilization: $U_n = P_n / (\ \ )$

***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.

3.2 Workload Quantification

Table 2 shows the workload and sequential performance of three STAP benchmark programs on a single SP2 processor. The entries of Table 2 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 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.

Table 2: Performance of Sequential STAP Programs on One SP2 Node

Benchmark Program	Program Component	Workload (Mflop)	Execution Time (Seconds)	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-PD	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

From Table 2, the utilization of FFT computations (DP in APT and HO-PD and FFT in GEN) are low, while the rest have good utilization. Slow FFT computation does not have much effect on the HO-PD program (DP takes less than 10% of total execution time), but has significant impact for the GEN program. The vector multiply (VEC) component also has poor utilization. Overall, the three sequential benchmarks APT, HO-PD, and GEN achieve a speed of 100 Mflop/s, 98 Mflop/s, and 38 Mflop/s, respectively. The corresponding utilization are 38%, 37%, and 14%. The utilization can be improved with better codes for FFT and VEC computations.

A simple way to improve performance is to use vendor supplied library routines. But the improvement may not be as high as expected. For instance, the FFT library routine for SP2 has a performance as high as 160 Mflop/s. But when used in the STAP benchmark, it only gives 30-45 Mflop/s. This is roughly about the same performance achieved by SP2 for the NAS FT benchmark, which is dominated by FFT computations.

3.3 Overhead Estimation

We briefly discuss below how to derive overhead expressions for SP2. The reader is referred to [28] for detailed discussions. Most MPP applications 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 processor. 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 the number of processors used in that run. The number of processors (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.

3.3.1 Parallelism Overhead

In a real signal processing application on MPP, the same parallel program is executed many times to process a stream of incoming radar sensor data. There is a one-time overhead for creating all processes and groups, which can be used by subsequent computations. In other words, the parallelism overhead is amortized. Therefore, in predicting STAP performance on SP2, we assumed that the parallelism overhead $T_{parallel} = 0$.

We must point out, though, creating a process or a group is expensive on MPPs. Our measurement shows that creating a process on SP2 takes about 10000 μs or more, equivalent to millions of flop. A group creation takes about 1000 μs. Therefore, a program that needs to frequently create processes or groups must have very large computation grains.

In what follows, we concentrate on communication overhead. In message passing MPPs, all interactions 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., $t = f(n, m)$).

3.3.2 Point-to-Point Communication

Using the *High-Performance Switch* (HPS), all processors on an SP2 can be considered equal distance away. The concept of neighboring processors or remote processors 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 processors. We measured the blocking send and blocking receive operations of the IBM MPL, using the *pingpong* 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. The total time for this pingpong operation is divided by 2 to get the point-to-point communication time. Some of the timing results are shown in Table 3, which confirms that there are only small differences for sending messages to different processors.

Table 3: One Way Point-to-Point Communication Times (μs) on SP2

n / *m*	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 3 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 [12]: The overhead is a linear function of the message length *m* (in bytes):

$$t = t_0 + m/r_\infty \quad (4)$$

where t_0 (called *latency*, or *start-up time*, in μs) is the time needed to send a 0-byte message, and r_∞ (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: $t = 46 + 0.035m$. In other words, the latency is t_0=46 μs, and the asymptotic bandwidth is r_∞=1/0.035=28.57 MB/s.

3.3.3 Collective Communications

In a *broadcast* operation, processor 0 sends an *m*-byte message to all *n* processors. In a *gather* operation, processor 0 receives an *m*-byte message from each of the *n* processors, so in the end *mn* bytes are received by processor 0. In a *scatter* operation, processor 0 sends a distinct *m*-byte message to each of the *n* processors, so in the end *mn* bytes are sent by processor 0. In a *total exchange* operation, every processor sends a distinct *m*-byte message to each of the *n* processors, so in the end mn^2 bytes are communicated. In a *circular-shift* operation, processor *i* sends an *m*-byte message to processor *i*+1, and processor *n*-1 sends *m* bytes back to processor 0. Note that in current message-passing systems, a collective communication always requires a process to send a message to itself. That is why mn^2 bytes are communicated in a total exchange, not $mn(n-1)$ bytes.

We have extended Hockney's expression (Eq. 4) 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 processors *n*. In other words,

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

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 4.

Table 4: Collective Communication Overhead Expressions for SP2

Operation	**Timing Formula**
Broadcast	$(52\log n) + (0.029\log n)m$
Gather/Scatter	$(17\log n + 15) + (0.025n - 0.02)m$
Total Exchange	$80\log n + (0.03n^{1.29})m$
Circular Shift	$(6\log n + 60) + (0.003\log n + 0.04)\, m$

3.3.4 Collective Computations

We measured three representative collective computation operations: *barrier*, *reduction* and *parallel prefix* (also known as *scan*). The curve-fitted communication overhead expressions are

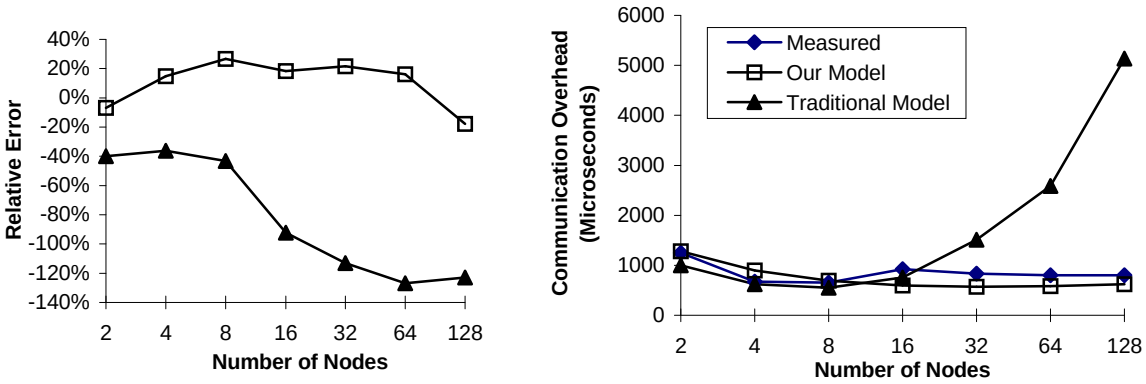
shown in Table 5. Note that over 256 processors, the barrier overhead is 768 μs, equivalent to the time to execute as many as 768×266=202,692 flops. This answers the question: "Should I use a synchronous algorithm?" The answer is only when the grain size is large. That is, hundreds of thousands flops are executed before a barrier.

Table 5: Collective Computation Overhead Expressions for SP2

Operation	Time Expression
Barrier	94 log$n$+10
Reduction	20log$n$ + 23
Parallel Prefix	60 log$n$ -25

There is a simpler method to estimate the overhead of a collective communication or computation, which is widely employed by MPP users, as it does not require any measurement by the users. This traditional method treats a collective operation as a sequence of point-to-point operations. The overhead of the collective operation is computed by summing the overheads of these point-to-point operations, using vendor provided latency and bandwidth values.

Figure 6 compares our method to the traditional method for predicting the overhead of a total exchange for both short and long messages. In part (a), we show the relative errors of the two methods when mn^2 =16 MB (e.g., m=1024 bytes when n=128). The traditional method underestimates the overhead with large errors, especially for large numbers of processors. In part (b), we compare the measured overhead with those projected by the two methods, when mn^2 =64 KB (e.g., m=4 bytes when n=128). While our method is close to the measured result, the traditional method overestimates the overhead significantly for large numbers of processors.



(a) Relative error for a total exchange of 16 MB messages

(b) Communication overhead for a total exchange of 64 KB messages

Figure 6: Comparison of two methods for predicting the overhead for total exchange

3.4 Parallel Algorithm Design

How does one design the initial parallel algorithm? The decision is affected by many factors: How many processors should be used? What should be the grain size? How to partition data and

computation? and so on. The following heuristics are very useful: Before making any other decisions, first determine the grain size of each individual step using speed information in Table 2. The grain size is defined to be the number of flop executed before a communication. Referring to Fig. 3, the grain size for step C_i should be no less than $t_0 \times P_1(i)$. Then analyze the data/control dependence of the sequential code to find out how much degree of parallelism (DOP) can be exploited for each step, subject to the grain size requirement just obtained.

It is often helpful to perform a *zero-overhead prediction* on the initially designed parallel algorithm. Table 6 shows the zero-overhead performance prediction of three parallel STAP benchmarks. This prediction is useful to provide some upper and lower bounds on the projected MPP performance and to answer the question "Will the parallel algorithm just designed achieve the desired performance level?" The answer depends on the performance requirement. Three requirements can be posed by a user:

Table 6: Extreme Value Metrics of the STAP Programs

Program	Workload W (Mflop/s)	Sequential Time T_1 (Seconds)	Maximal Performance P_∞ (Gflop/s)	Average Parallelism T_1/T_∞	Critical Path T_∞ (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

- *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. Note that the critical path lower-bounds the parallel execution time. From Table 6, the APT and the HO-PD 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 processors 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 speed, say 10 Gflop/s. From the Maximal Speed column of Table 6, Both APT and HO-PD 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.

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.

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

Even when the preliminary prediction shows promising result (e.g., the HO-PD program passes all three requirement tests), we should not rush to the coding stage, because we have not considered overhead yet. We next incorporate all communication overheads and consider various parallelization strategies. The next section shows how performance prediction is used to select the best strategy. After several prediction and algorithm redesign steps, we obtain a parallel program which is likely to perform well.

For instance, the structure of the final parallel APT algorithm is shown in Fig. 7. The DP step is distributed to up to 256 processors. The *total exchange* step is collectively executed by all processors., with an aggregated message length of 17 MB. The HT step is sequentially executed on a single processor (DOP=1). The *broadcast* operation must send a 80KB message to all processors. The BF step performs beamforming operations using up to 256 processors. The TD step needs to track the targets on up to 256 processors locally. Then all the local target reports are merged through the final collective *reduction* operation.

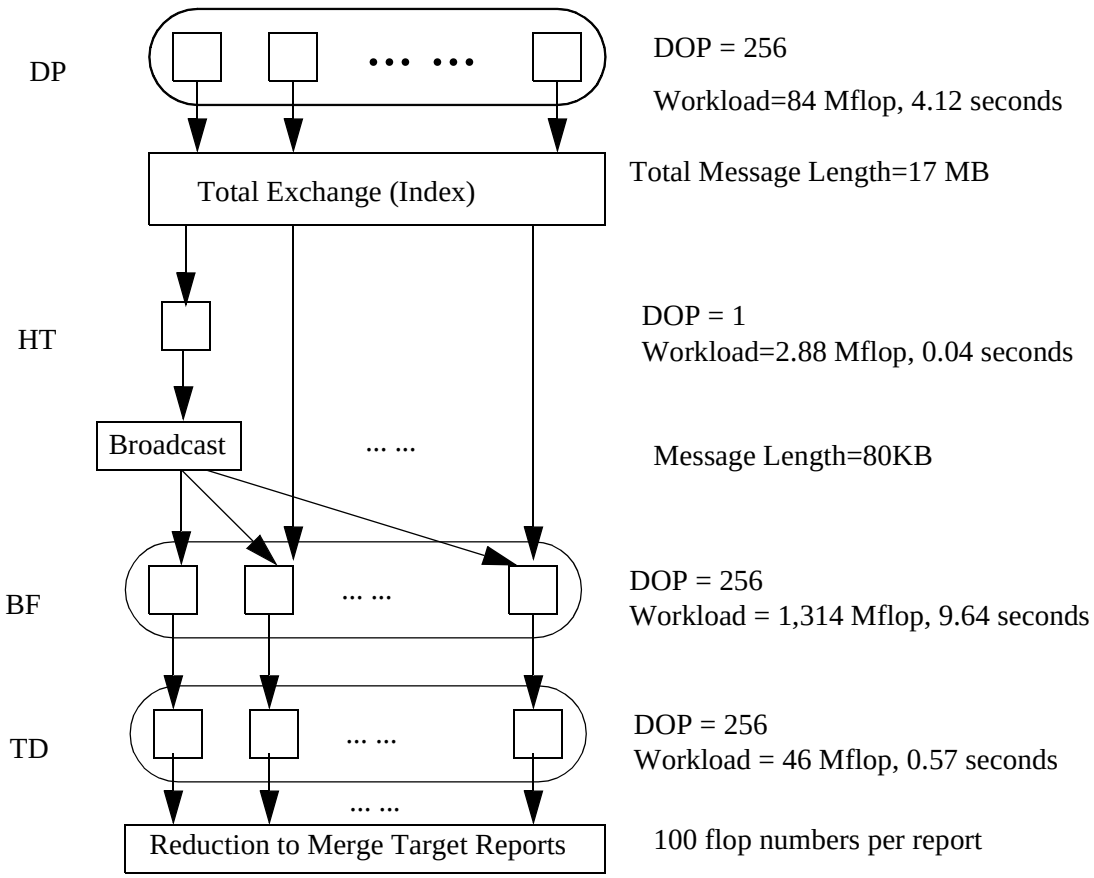


Figure 7: Structure of A Coarse Grain Parallel APT Algorithm

3.5 Performance Prediction

Table 7 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 3.3. From these expressions, it is straightforward to calculate the other performance metrics using the formulae in Section 3.1 to perform a full-fledged prediction including all overhead. The entries of Table 8 are obtained from Table 7 for a 256-processor 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 7: Predicted Execution Time of Parallel STAP Programs on SP2

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

Let us look at the parallel APT program more closely. From Table 7, the execution time has four terms. The first term is a constant, due to sequential execution of the HT step. The second term is due to parallel execution of the DP, BF, and TD steps. The third term is due to the total exchange communication. The fourth term is equally due to the broadcast and the reduction communication steps. For large numbers of processors, the HT step becomes a bottleneck. To further improve the APT performance, an obvious choice is to parallelize the HT step. But this does not work on SP2, as shown below.

We show how the prediction model can help select a parallelizing strategy by considering the Householder transform routine. This routine is important because it has been used heavily in all five STAP benchmarks. So we must answer the question: “Should the Householder transform be parallelized?” As it turned out, the answer is no for SP2 and Paragon, but yes for T3D. We will focus on the householder transform used in the HT step of the APT benchmark.

The Householder transform in the STAP benchmark suite is unusual in that (1) the matrices to be transformed are small (the largest one is 32×320); and (2) the matrix is not even square, thus not symmetric. Many existing parallel Householder algorithms assume symmetry. The time

Table 8: Predicted Performance of Parallel STAP Programs on 256 SP2 Node, Including All Overheads

Program	Parallel Time T_{256}	Speed P_{256}	Speedup S_{256}	Utilization U_{256}
APT	0.137 Seconds	10.5 Gflop/s	104	15%
HO	0.606 Seconds	21.2 Gflop/s	215	31%
GEN	1.400 Seconds	3.8 Gflop/s	101	6%

consuming component in the Householder transform is to triangularize a complex $p \times q$ matrix. The sequential time needed for such a triangulization is approximately:

$$T_1 = 8 \times p^2 \times q \times T_{flop} \tag{6}$$

where T_{flop} is the time (in μs) needed to perform one floating point operation. A natural way to parallelize the triangulation algorithm is to partition the matrix into *n* chunks along the *q* dimension. In other words, each of the *n* processors will process q/n columns. The parallel execution time will be approximately:

$$T_n = \frac{16 \times p^2 \times q \times T_{flop}}{n} + p^2 \times R(n) \tag{7}$$

where the second term is the communication overhead. More specifically, $R(n)$ is the time (in μs) needed to perform a reduction operation over *n* processors. The speedup of using the parallel algorithm is:

$$S_n = \frac{T_1}{T_n} = \frac{n}{2 + \frac{nR(n)}{8qT_{flop}}} \tag{8}$$

If we ignore the communication overhead, the speedup would be *n*/2. Now let us apply the speedup formula to APT and the SP2 specifically. The HT step of the APT benchmark in Table 2 has a sustained speed of 72 Mflop/s, which translates to a $T_{flop} = 1/72 = 0.014$ μs. From Table 5, we have $R(n) = 20\log n + 23$. The parameter *q* is equal to 320 in the APT. Using these parameter values, we obtain the speedup equation:

$$S_n = \frac{}{2 + \frac{}{8 \times 320 \times 0.014}} = \frac{}{2 + \qquad + 0.64n} \tag{9}$$

For the speedup to be greater than 1, the following inequality must hold:

$$n > 2 + \qquad + 0.64n \Rightarrow n > 1.56n\log \quad + 5.56 \tag{10}$$

This inequality clearly cannot be satisfied. Thus the parallel algorithm is always slower than the sequential one for any machine size n. We also know the reason why it is slow: the reduction overhead is too high. In fact, even with the optimistic assumption that $R(\ ) = \qquad = 43$ for any k processors, the speedup is still always less than 1.

4 Validation of Early Prediction

Suppose the user changed the performance requirement to 100 times of speedup.Table 8 suggests that all three benchmarks satisfy this new requirement by using 256 processors, as the predicted speedup are 104, 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 processors. The sustained speed performance is compared with the prediction. The results are shown in Fig. 8. For the majority of cases, the error is less than 10%, and the maximal error is 22%. We also see that sometimes the predictions underestimate 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 are available as the machine size increases.

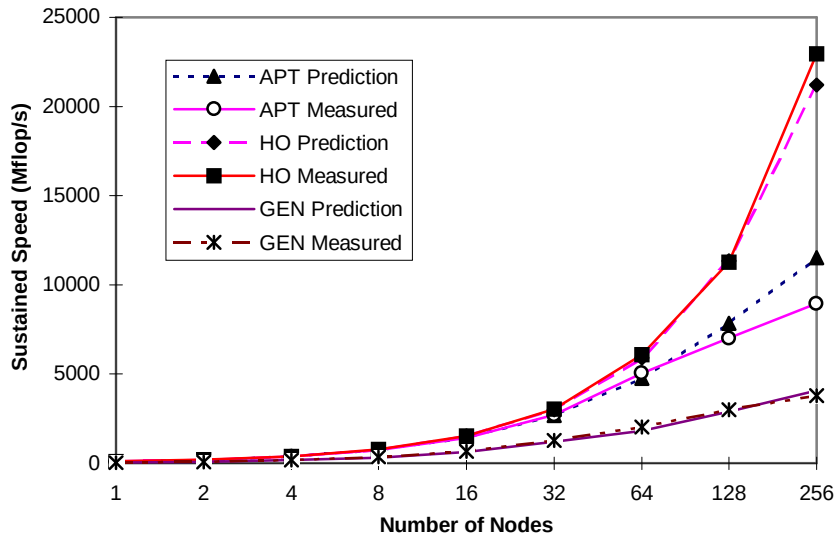


Figure 8: Comparison of predicted and measured performance of the STAP programs

We also compared three prediction schemes: The *uniform-speed 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 2. Both the zero-overhead and the uniform predictions ignore communication overhead. The relative errors of projected execution times with respect to the measured times are shown in Fig. 9 for the parallel APT benchmark. 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 processors. Fig.9 shows that, when compared with the measured performance, the uniform-speed prediction has a 90% error. The zero-overhead prediction has a 69% error. But the full-overhead prediction brings the error down to only 17%.

5 In Search of higher System Utilization

The parallel APT program was projected to achieve an utilization of only 15%, which is typical of many real applications. It is difficult to reduce the execution time further. All but the HT step are fully parallelized, and we saw in Section 3 that parallelizing the HT step will not pay. The communication overheads seem to be unavoidable. Is there any way to increase the utilization without reducing the execution time? The answer is yes. We mention three popular techniques below. The early prediction scheme can help decide which technique to use and how to use them.

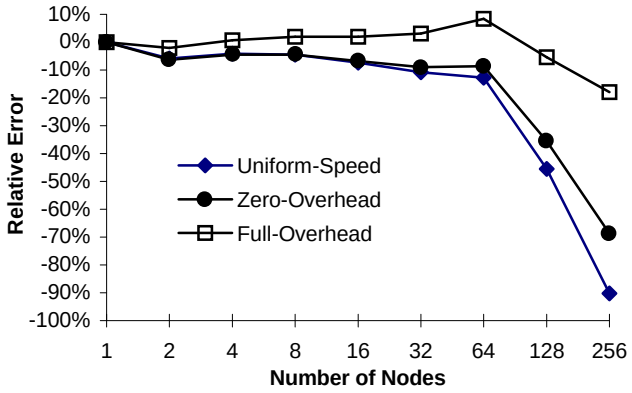


Figure 9: Comparison of three performance prediction schemes with uniform-speed, zero-overhead, and full-overhead respectively.

5.1 Scaling Workload

The first method scales up the problem size, thus the workload. For many programs, this would amortize the communication overhead and the sequential bottleneck, generating higher utilization. There are two approaches to scaling workload:

- *Fixed-Time*. The total workload of the parallel program is increased so that the parallel program will have the same execution time as the original sequential program [11].
- *Memory-Bound*. The total workload of the parallel program is increased so that the parallel program will use all available memory in the MPP [26].

In general, memory-bound scaling should be used. The early prediction scheme can be used to estimate the potential improvement of these workload scaling approaches. In Fig.10, the projected utilization curves of these approaches are compared to the unscaled utilization for the parallel APT program running on the IBM SP2. The utilization of the parallel APT without workload scaling (the Fixed-Load curve) drops to 15% as the machine size increases. However, using memory-bound scaling, a parallel APT can maintain an almost constant utilization of 36%.

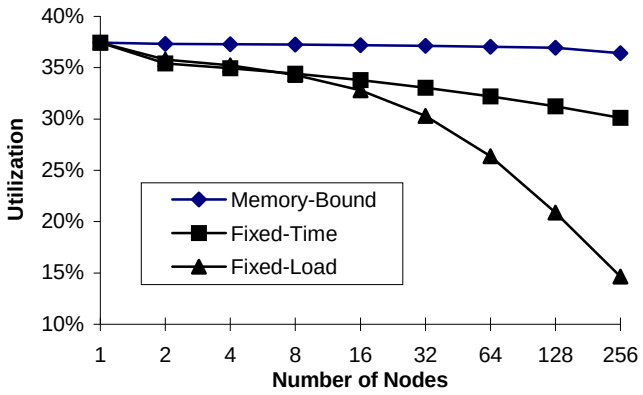


Figure 10: Improving utilization by workload scaling: parallel APT programs on the IBM SP2

The other two methods aim at improving the *throughput*, defined as the number of jobs processed in unit time. If only one job is executed at a time, throughput is just the reciprocal of the execution time. For instance, according to our prediction (Table 2), the throughput for the parallel APT program on a 256-node SP2 is one APT per 0.137 seconds, or 1/0.137=7.3 APTs per second.

However, the throughput metric is usually used when multiple jobs are executed. In many cases, the system throughput can be increased at the expense of longer execution time for each individual job. Both methods are suitable for parallel signal processing, as a continuous stream of radar sensor data needs to be processed.

5.2 Pipelining

Another way to increase the throughput of a parallel system is by *pipelining*, where successive jobs overlap their executions among several pipeline stages. The throughput becomes the reciprocal of the execution time at the longest pipeline stage, instead of the execution time of the entire pipeline.

Table 9 shows the predicted timing of various steps in the parallel APT algorithm of Fig.7. We can construct a two-stage pipeline for the APT program on a 256-node SP2, by using 128 nodes per pipeline stage. The first stage consists of the DP, HT, broadcast, and the total exchange steps, while the second pipeline stage contains the remaining steps. The time to forward data between stages can be ignored. The total execution time for each APT increases from 0.137 to 0.196 seconds. However, the longer of the two pipeline stages has an execution time of only 0.102 seconds. The throughput increases from 1/0.137 to 1/0.102=9.8 APTs per second, or equivalently, 14 Gflop/s. The corresponding utilization increases to 20%. Note that we can not further improve throughput with more stages, because the BF step along would need 0.15 seconds on 64 nodes, more than the original 0.137 seconds.

Table 9: Time Breakdown in Parallel APT Program Components

No. of Nodes	DP	HT	BF	TD	Index	Reduce	Broadcast
64	0.064	0.04	0.150	0.009	0.026	0.012	0.012
128	0.032	0.04	0.075	0.004	0.016	0.014	0.014
256	0.016	0.04	0.037	0.002	0.010	0.016	0.016

5.3 Throughput Enhancement

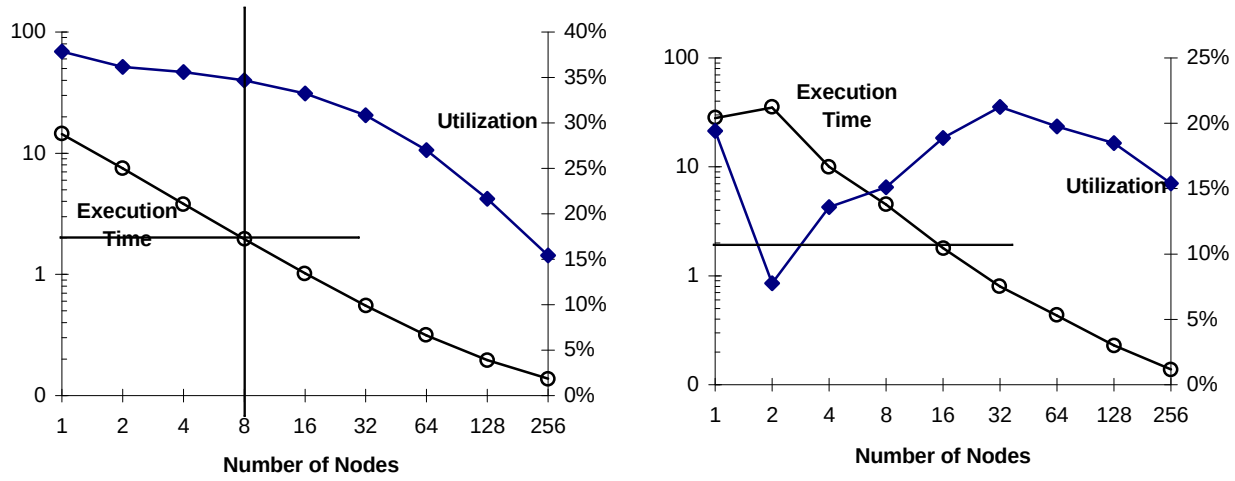
This technique is more powerful than pipelining. The idea is to increase throughput by assigning a separate sequential job to each node. At the extreme, up to *n* jobs are processed simultaneously on an *n*-node system. The throughput is *n* divided by the execution time of the longest job. For example, we can assign a sequence of APTs to each of the 256 nodes. The execution time per APT increases to 14.37 seconds, but the throughput becomes 256/14.37=17.81 APTs per second, or 25.6 Gflop/s.

Fourteen seconds may be too long an elapsed time for real-time radar signal processing. In many production environments, it is common practice to maximize the throughput within an execution time limit. Suppose we use the parallel APT program to process radar signals on a 256-processor SP2, with the constraint that each APT should be executed in no more than 2 seconds. How many processors should we use for one APT? We discuss below how to find the optimal number of processors to maximize throughput, which also answers the question: “How do I best utilize my 5000 CPU-hours allocation on an MPP?”

Draw on the same chart the utilization and the execution time as a function of machine size. Then find all machine sizes that have no more than 2 seconds execution time, and from among those, find the machine size with the best utilization. This is the machine size that will gives the best throughput. This process is shown in Fig.11a for the parallel APT program. Since the execution time monotonically decreases, using 8 or more nodes will satisfy the 2-second time requirement. It happens that the 8-node machine size also has the best utilization. Thus, we should assign 8 nodes to an APT, and 32 APTs are simultaneously processed by the 256 nodes. The throughput will be 16 APTs per second, or 23 Gflop/s.

A widespread misconception is that single-node, or sequential computation always has the highest utilization, as parallel computing has extra communication and idling overheads. This is often but not always true. Figure 11b shows a scenario which happens in many real applications with a large data set. The sequential program requires more memory than what is available in a single node. Excessive paging increases the execution time considerably. The parallel program needs more memory for message buffers. In fact, all parallel STAP programs need three times as much memory as their sequential counterparts. So when the program is executed on two nodes, paging becomes even more severe. This, together with added communication and idling overheads, degrades performance significantly. The situation gets better when more nodes are used. The utilization tappers off after 32 nodes, due to overhead and sequential bottleneck.

In this example, using 16 or more nodes satisfies the 2-second time requirement, and the 32-node machine size gives the best utilization, with an execution time of about one second. So 256/32=8 APTs are processed simultaneously, and the throughput is 8 APTs per second, or 11.6 Gflop/s.



(a) Execution time and utilization without excessive paging overhead

(b) Execution time and utilization with excessive paging overhead

Figure 11: Finding the optimal machine size subject to a 2-second execution time limit

6 Concluding Remarks

We have demonstrated the advantages of using early prediction to avoid unnecessary cost in repeated software development on MPPs. Our STAP benchmarking experiences on the SP2, T3D, and Paragon have led to the development of the early performance prediction scheme being presented. The scheme is easy to apply and provides pretty accurate prediction of MPP performances by realistically quantifying the workload and communication overhead.

Most user applications running on current MPPs resulted in rather low a system utilization. We show how to apply the early prediction scheme to assess achievable performance level, before embarking on costly encoding and debugging efforts. The scheme produces performance Metrics, often in simple mathematical expressions. Our scheme helped increase the system utilization and achieve higher throughput in real-time applications reported in several companion articles [16,17,28,29]. We encourage MPP users to apply the phase parallel model and use the early scheme to predict performance, especially in coarse-grain, SPMD, data-parallel applications.

Acknowledgments

This research was carried out while both authors worked at the University of Southern California. We would like to thank David Martinez and Robert Bond at MIT Lincoln Laboratory for their support in this work. We are grateful to the User-Support group at Maui High-Performance Computing Centre for their help. We thank Lionel Ni of Michigan State University, Howard Ho, Craig Stunkel and Hubertus Franke of IBM for helpful discussions. We want also to thank Cho-Ming Wang of USC, Richard Frost of SDSC, David Scott and Victor Jackson of Intel SSD, and Richard Foster of Cray Research for their technical assistance.

References

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

2 R.J. Bergeron, “The Performance of the NAS HSPs in 1st Half of 1994", Report NAS-95-008, NASA Ames Research Centre, February 1995.

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

4 L.N. Bhuyan and X. Zhang, *Tutorial on Multiprocessor Performance Measurement and Evaluation*, IEEE Computer Society Press, Los Alamitos,Ca., 1995.

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

6 P. Brinch Hansen, *Studies in Computational Science: Parallel Programming Paradigms*, Prentice-Hall, New Jersey, 1995.

7 *Crisis in High Performance Computing*, University College London, Sept. 1995. http://www.hensa.ac.uk/parallel/groups/selhpc/crisis

8 J. J. Dongarra, “The Performance Database Server (PDS): Reports: Linpack Benchmark - Parallel", http://performance.netlib.org/performance/html/linpack-parallel.data.co10.html

9 T. Fahringer, “Estimating and Optimizing Performance for Parallel Programs”, *IEEE Computer*, 28(11), pp.47-56, Nov. 1995.

10 G.C. Fox, R.D. Williams, P.C. Messina, *Parallel Computing Works!*, Morgan Kaufmann Publishers, Inc., San Francisco, Ca., 1994.

11 J.L. Gustafson, “Re-evaluating Amdahl’s Law”, *Comm. ACM*, 31(5),pp.532-533, May 1988.

12 R. W. Hockney, “Performance Parameters and Benchmarking of Supercomputers”, *Parallel Computing*, 17, pp. 1111-1130, 1991.

13 R. W. Hockney and M. Berry, “Public International Benchmarks for Parallel Computers: PARKBENCH Committee Report No. 1", *Scientific Computing*,3(2),pp.101-146, Feb.1994.

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

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

16 K. Hwang, Z. Xu, M. Arakawa, "STAP Benchmark Performance on the IBM SP2 Massively Parallel Processor”, *IEEE Transactions on Parallel and Distributed Systems*, pp. 522-536, May 1996.

17 K. Hwang and Z. Xu, “Scalable Parallel Computers for Real-Time Signal Processing”, *IEEE Signal Processing*, July 1996. pp.50-66.

18 *IEEE Computer* Special Issue on Performance Evaluation Tools, Nov. 1995.

19 *IEEE Parallel and Distributed Technology* Special Issue on Performance Evaluation Tools, Winter 1995.

20 MHPCC, MHPCC 400-Node SP2 Environment, Maui High-Performance Computing Centre, October 1994.

21 C.M. Pancake, M.L. Simmns, and J.C. Yan, “Performance Evaluation Tools for Parallel and Distributed Systems”, *IEEE Computer*, 28(11), pp.16-19, Nov. 1995.

22 D. Pease, et al, “PAWS: A Performance Evaluation Tool for Parallel Computing Systems”, *IEEE Computer*, 24(1), pp.18-29, Jan. 1991.

23 S. Saini and D.H. Bailey, “NAS Parallel Benchmark Results 12-95”, NASA Ames Research Centre *Technical Report* NAS-95-021, Dec. 1995.

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

25 C. B. Stunkel, *et al*, “The SP2 Communication Subsystem”, IBM T.J. Watson Research Centre Report, August 1994.

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

27 L.G. Valiant, “A Bridging Model for Parallel Computation”, *Comm. ACM*, 33(8), pp.103-111, Aug. 1990.

28 Z. Xu, K. Hwang, “Modelling Communication Overhead: MPI and MPL Performance on the IBM SP2 System”, *IEEE Parallel and Distributed Technology*, pp. 9-23, March 1996. pp.9-23.

29 Z. Xu and K. Hwang, “MPP versus Clusters for Scalable Computing”, *International Symposium on Parallel Architectures, Algorithms and Networks*, Beijing, China, June12-14, 1996.

Biographical Sketch

Zhiwei Xu is a Professor at the National Centre for Intelligent Computing Systems (NCIC), Chinese Academy of Sciences, Beijing, China. He received the Ph.D. degree from the University of Southern California in August 1987. He has taught at Rutgers University and Polytechnic University in New York. This research was carried out at the University of Southern California, while he was working in the STAP Benchmark evaluation of the IBM SP2, Cray T3D, and Intel Paragon, a project supported by MIT/Lincoln Laboratory. His current research interests include parallel computer architecture and their programming.

Kai Hwang is a Chair Professor of Computer Engineering at the University of Hong Kong, on leave from the University of Southern California. He received the Ph.D. degree from the University of California at Berkeley. An IEEE Fellow, he has served as a Distinguished Visitor of the Computer Society, the ACM SIGARCH Board of Directors, and is the founding Editor-in-Chief of the *Journal of Parallel and Distributed Computing*. He has published over 140 scientific papers and five advanced computer books, most of which are in computer architecture and parallel processing. His current research interest lies in scalable multiprocessors, clustered multicomputers; parallel software tools, communication libraries, and programming environments for scalable supercomputing and distributed multimedia applications.