REPORTING COMPUTATIONAL EXPERIMENTS WITH PARALLEL ALGORITHMS: ISSUES, MEASURES, AND EXPERTS' OPINIONS

Richard S. Barr[1] and Betty L. Hickman[2]

October, 1992
(Revised)

Invited paper, *ORSA Journal on Computing*

[1]Department of Computer Science and Engineering,
Southern Methodist University, Dallas, Texas 75275
barr@seas.smu.edu

[2]Mathematics and Computer Science Department,
University of Nebraska at Omaha, Omaha, Nebraska 68182
hickman@unocss.unomaha.edu

ABSTRACT

Accompanying the increasing availability of parallel computing technology is a corresponding growth of research into the development, implementation, and testing of parallel algorithms. This paper examines issues involved in reporting on the empirical testing of parallel mathematical programming algorithms, both optimizing and heuristic. We examine the appropriateness of various performance metrics and explore the effects of testing variability, machine influences, testing biases, and the effects of tuning parameters. Some of these difficulties were explored further in a survey sent to leading computational mathematical programming researchers for their reactions and suggestions. A summary of the survey and proposals for conscientious reporting are presented.

KEYWORDS

Parallel processing; algorithm testing; performance evaluation; performance metrics; efficiency measures; mathematical programming.

Algorithm development is at the heart of mathematical programming research, wherein more efficient algorithms are prized. As an indicator of algorithmic performance, efficiency reflects the level of resources (central processor time, iterations, bytes of primary storage) required to obtain a solution of given quality (percent optimality, accuracy)[18,25]. A variety of measures and summary statistics has been devised to reflect efficiency and compare algorithms.

The efficiency of an algorithm relative to others has traditionally been determined by (1) theoretical, order analysis and (2) empirical testing of algorithmic implementations, or *codes*. While both approaches have merit, computational testing is increasingly an imperative for publication. This is due, in part, to the occasional failure of order analysis to predict accurately the behavior of an algorithm's implementation on problems of practical interest. For example, while the simplex method has daunting worst-case behavior, its efficiency as an optimizer for a wide variety of industrial applications is well-documented[43,46].

One technological advance that holds great promise for solving difficult problems is application-level parallel processing, whereby the power of multiple processing elements can be brought to bear on a single problem. If the work associated with an algorithm can be properly subdivided and scheduled on separate processors for simultaneous execution, opportunities for dramatic reductions in solution times arise. As with traditional single-processor (*serial*) machines, solution efficiencies are directly tied to how well the algorithmic steps match the architecture of the underlying machine. Therefore, with the evolution in computing machinery comes a corresponding evolution in algorithms and their implementations.

This paper addresses complications that arise when reporting on implementations of parallel algorithms. Several common metrics of the efficiency of parallel implementations result in measurement and comparison difficulties stemming from limitations imposed by machine designs, differences in machine architectures, the stochastic nature of some parallel algorithms, and inherent opportunities for the introduction of biases. Of particular concern is *speedup*, a widely used measure for describing the efficiency achieved over serial processing by the use of multiple processors. In addition to documenting problems of traditional reporting of parallel experimentation, we also present the results of a survey of leading mathematical programming researchers regarding their views on the topic, and close with proposed guidelines for conscientious reporting. We begin with an overview of parallel processing as it relates to mathematical programming research.

1. BACKGROUND

1.1. What Is Parallel Processing?

Parallel processing is the simultaneous manipulation of data by multiple computing elements working to complete a common body of work. While most computers have some degree of parallelism, as with overlapped input-output and calculation, only recently have systems become commercially available that allow an applications programmer to control several processing units. A parallel programmer might, for example, minimize a function $f(x)$ for $0 \le x \le 10$, by using one processor to examine the interval for $0 \le x < 5$, a second to simultaneously evaluate $5 \le x \le 10$, with the final result determined (in serial) upon completion of the previous tasks. The objective would be to complete the work in roughly half of the time required by a single processor.

Why use parallel processing to perform a task or solve a given problem? The most prevalent reasons are:

- *Absolute speed:* the reduction of real ("wall clock") time, considering all technological options. Although application-dependent, parallel processing is generally acknowledged to provide this capability, as evidenced by the multiprocessor design of all state-of-the-art supercomputers.
- *Relative speed and cost:* improvement in real time, subject to a practical constraint such as cost or limited computing options. The expectation is that an ensemble of relatively inexpensive slow processors can complete the work more quickly than a faster cost-equivalent serial machine, or in the same time at a lower cost. Such "cheap thrills" have been achieved by some applications, such as database transaction processing, but are not always attainable. While some writers believe that the range of applicability is a narrow one [7,19], most commercial parallel systems are designed with this result in mind.
- *Scalable computing:* having the ability to improve performance with additional processing elements. A flexible design permits a computing system to grow in terms of processing power, in the same manner that additional disk drives increase storage capacity. An application that exploits any number of parallel computing units can use incremental system upgrades to speed up processing.

If real-time reduction is not important, traditional serial processing is usually easier and more cost effective. Hence the motivation for parallel machines springs from the need to solve existing problems faster or to make tractable larger and more difficult ones.

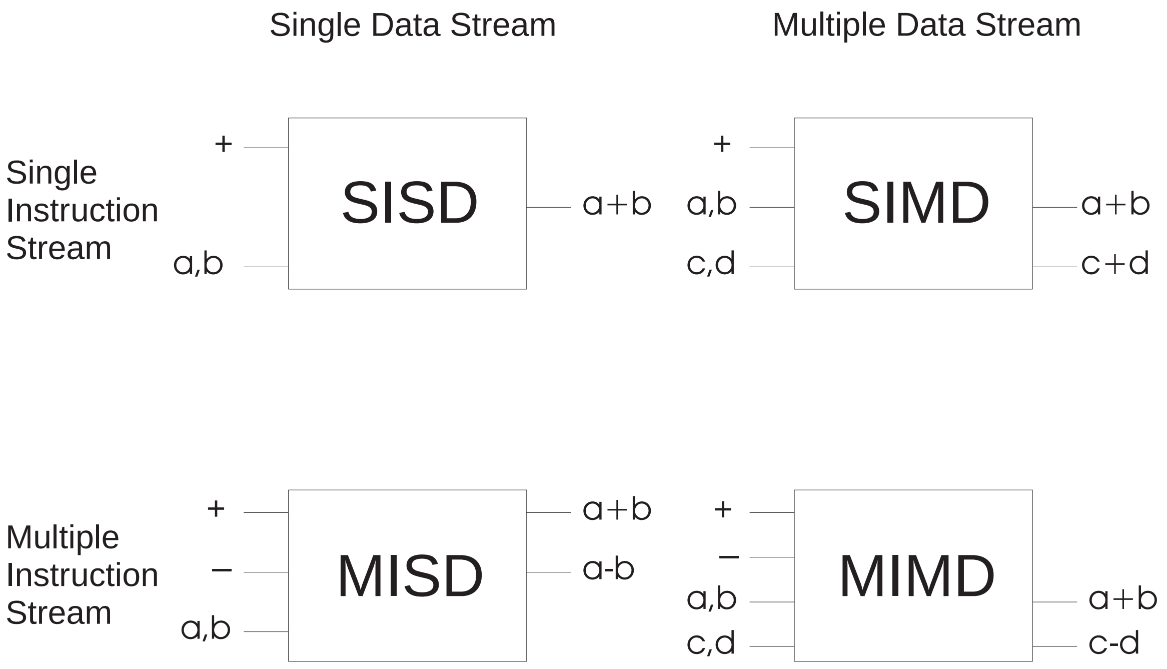

Figure 1. Flynn's Machine Classifications

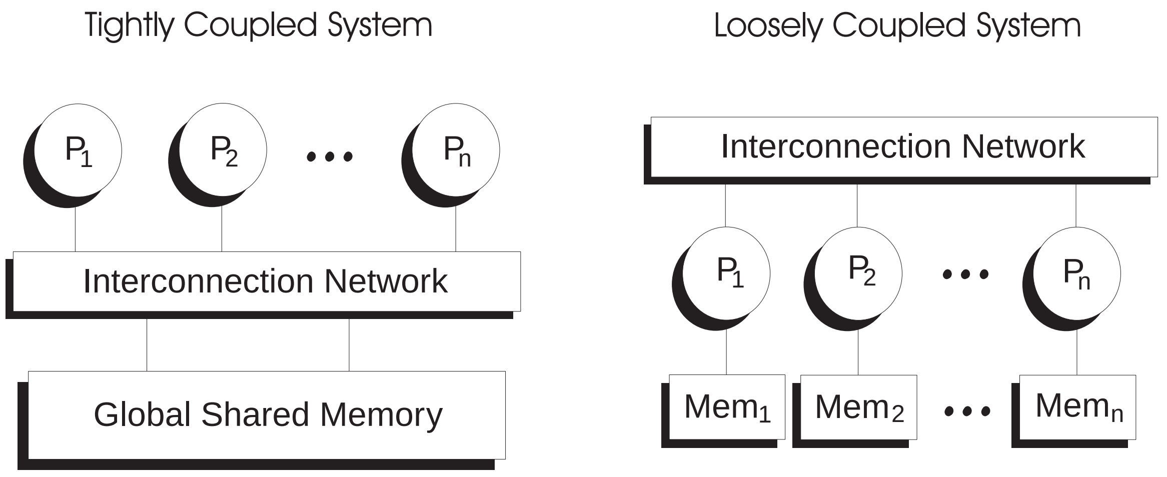

Figure 2. Tightly and Loosely Coupled Parallel Systems

1.2. Types of Parallel Computers

Parallel computers are as varied in design as they are many, hence classification schemes have proven useful in describing specific machines. One widely employed method of categorizing computer architectures was introduced by Flynn[16], and is based on the concepts of an instruction stream and a data stream. An instruction stream is a sequence of instructions carried out by a computer (that is, program steps), while a data stream is the set of data on which an instruction stream is executed. This taxonomy delineates machine types by the number and type of simultaneously executing streams, as follows (see also Figure 1 from [29]).

- Single-instruction, single-data (SISD), the traditional serial, uniprocessor computer, that executes one series of instructions on a single set of data (for example, the IBM Personal Computer, Cray-1, Vax 780, and Sun IPC);
- Single-instruction, multiple-data (SIMD), a parallel machine design wherein all processors execute the same instruction in lockstep, and apply it to different pieces of data (for example, the Thinking Machines' CM-2 and CM-5, and DAP);
- Multiple-instruction, single-data (MISD), which applies multiple operations simultaneously to a single data stream (no general-purpose MISD computers are available today); and
- Multiple-instruction, multiple-data (MIMD), the most widely employed parallel machine architecture; each such computing system contains multiple, independently executing processors which can operate on different datasets (including—but not limited to—the Cray Y-MP, IBM 3090 series, Vax 9000, Sequent Symmetry 2000, BBN Butterfly, Encore 90, Convex C-2, CM-5, Intel iPSC and Paragon, Teradata DBC/1012, NCR System 3600, Pyramid, Alliant FX, Myrias, nCUBE/2, MasPar MP-1, and Silicon Graphics).

Processors in SIMD and MIMD parallel computers communicate either via a common shared memory accessed through a central switch, or by messages passed through an interconnection network in a *distributed* system. Shared-memory multiprocessors are called *tightly coupled* if the time required to access a particular memory location is the same for all processors, as opposed to being proximity dependent or *loosely coupled* (see Figure 2). Computer systems are termed *massively parallel* if they contain 1,000 or more processors.

Each architecture has its own advantages and disadvantages. Shared memory systems are simpler to program than distributed ones, since processors can share code and data, and can communicate via the globally accessible memory. Since distributed systems have no central switch, they can accommodate a larger number of processors, but at the expense of slower communication and more elaborate programming.

Of the many varieties of parallel machines, the dominant category appears to be tightly coupled MIMD. We estimate the installed base (number of commercial machines installed and operating at customer sites) for distributed SIMD and MIMD systems to be around 1,000. This compares with the over 10,000 shared-memory multiprocessors installed by IBM and over 4,000 by Sequent Computer Systems alone—although the proportion of sites using the parallel capability is anyone's guess.

In addition to these advanced machine designs, software systems have emerged that permit parallel programming across independent computers connected by a local area network[10,40]. Because of the widespread use of networked workstations in the engineering and scientific communities, coupled with their heavy computational needs and limited budgets, this truly distributed approach to parallel processing may prove to be a popular one.

1.3. Parallel Algorithm Research and Computational Testing

If increased speed in solving a given problem is the goal of parallelism, what are the sources of speed? The first source is the machine used, its architecture, and the speed of each of its operations, including integer and floating-point arithmetic, memory moves, and communications. Second, speed comes from the solution algorithm and its effectiveness for the problem. Third, speed depends on the algorithm's implementation, or code. A code is a program which maps the algorithm's steps and data requirements onto a given machine, and its speed depends on the compatibility of the mapping and the efficiency of the coding itself.

What is sought then, for a problem, is not only a quick machine, but an efficient solution algorithm and an implementation whose steps and data needs match the machine's characteristics. Algorithm research is directed at the design and implementation of problem solution methods with these qualities. In a parallel setting, this means seeking efficient algorithms with steps that can be executed simultaneously. Hence, by designing algorithms to exploit the new machine architectures, researchers seek to answer the question: Can parallel computing lead to the solution of new problems and faster solutions of current problems?

Researchers need and want empirical evidence of an algorithm's efficiency, and performance of a code can be evidence of the effectiveness of the underlying algorithm. Order analysis is not always a good predictor of performance since it typically ignores the variety of machine operations involved in an implementation, and is often concerned with worst-case behavior that may not be representative of behavior on problems of practical interest.

Computational experience summarizes all operations, in a given setting on a specific problem set. While the appropriateness of a test set may be questioned, computational testing gets to the heart of the matter. Is not the reason for building algorithms the solution of problems?

2. FOCUS: PARALLEL MATHEMATICAL PROGRAMMING ALGORITHMS AND CODES

The focus of this paper is on the empirical testing of parallel optimization and mathematical programming algorithms. Traditional methods in this area tend to have different characteristics than those explored in the parallel processing literature. Examples of the

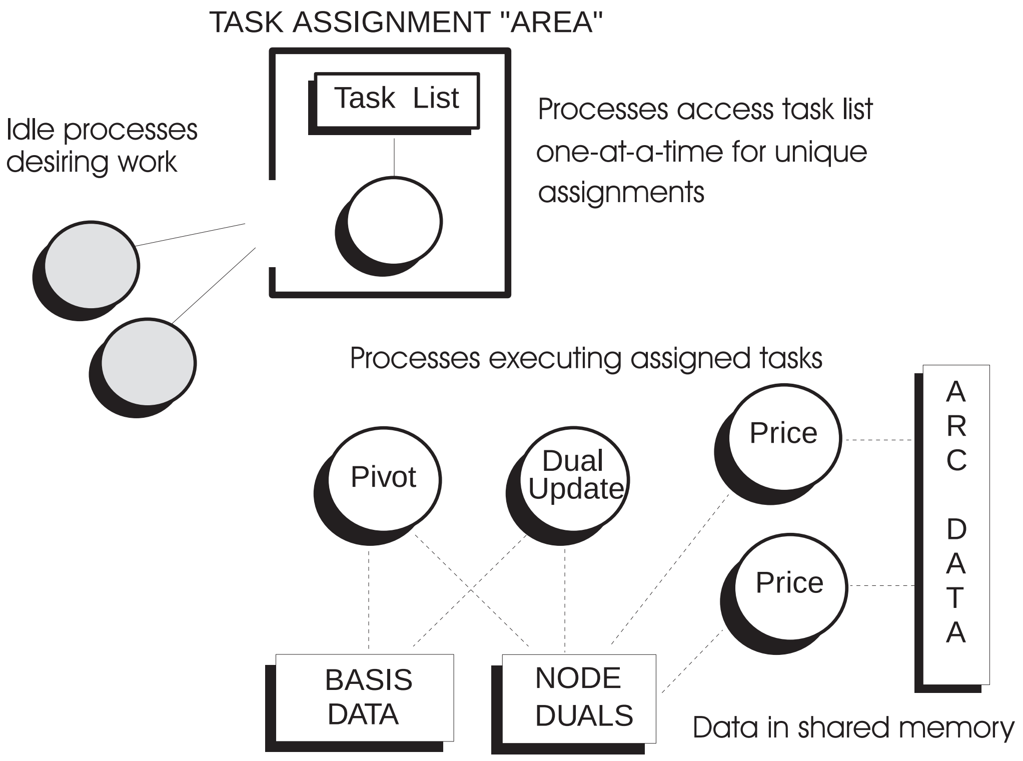
TASK ASSIGNMENT "AREA"
Task List
Idle processes desiring work
Processes access task list one-at-a-time for unique assignments
Processes executing assigned tasks
Pivot
Dual Update
Price
Price
ARC DATA
BASIS DATA
NODE DUALS
Data in shared memory

Figure 3. Example of a Parallel Network Simplex Code Design

“embarrassingly parallel” mathematical applications typically cited in this literature are: wave mechanics, fluid dynamics, and finite-element applications[21]; SIMD applications in the survey [24] included image processing, neural network training, and satellite orbit collision detection. In each case, the algorithm is highly parallelizeable (over 99.9% of the operations) and the problems easily decomposable, often linearly in terms of a single parameter.

In contrast, traditional mathematical programming optimization algorithms often have a strong serial component and the problems addressed are not decomposable in a simple manner. The problems attacked can have alternate optima, the ease or difficulty of their solution is not always predictable in terms of problem dimensions or other descriptive factors, and there are an enormous number of paths to an optimal solution. Hence much creativity is required in parallelizing traditional methods, and new algorithms must be developed and previously rejected ones reexamined in light of these new computing paradigms.

2.1. Examples of Parallel Mathematical Programming Applications

To provide a flavor of the research on parallel optimization algorithms, we next summarize research in two application areas (see [50] for an early survey). Compare the characteristics of these algorithms with the typical parallel applications described previously.

2.1.1. Parallel Network Simplex

Numerous authors have built parallel implementations of the network simplex algorithm on shared-memory MIMD machines[3,11,35,42]. Strategies employed in these codes included simultaneous pricing and pivoting, parallel pricing of different arc sets, and, in [3,11], decomposition of the pivot operation. Problems with as many as 50,000 nodes and one million arcs were solved in 12.2 minutes, and times were reduced to as little as one-sixteenth of serial using 19 processors[3].

In the implementation of Miller, Pekny, and Thompson[35], only the pricing operation is performed in parallel, so that the addition of processors results in a larger number of arcs considered for an incoming variable. Upon completion of the pricing step, all processors synchronize and perform the same pivot in local memory.

In contrast, the other codes assign simplex tasks to different processors. For example, in the authors' implementation[3], tasks are assigned on a self-scheduled, or "as available," basis. With all data in shared memory, some processors may be pricing variables, while others are performing different portions of the pivot operation (see Figure 3). When a processor completes its assignment, it selects another from a task list; when no tasks remain, the procedure terminates[32].

While the self-scheduled approach is highly efficient in terms of machine utilization, it results in stochastic code performance. Since key decisions—such as choice of incoming variable—are timing dependent, microsecond timing differences from one run to the next can yield divergent results (that is, number of pivots, solution time, alternate optimal solution). Hence, a program containing such timing-dependent logic—known as a *race condition*—may have good load balancing among the processors, but it is often accompanied by significant variability in run-time behavior.

2.1.2. Parallel Branch-and-Bound

A branch-and-bound algorithm for (mixed) integer programming lends itself well to parallelism. Since hundreds or thousands of linear programming relaxations are typically solved to identify the optimal integer solution, an obvious parallelization approach is to assign different parts of the search tree to separate processors. Tightly coupled MIMD implementations[6,9,34,49] have been highly successful, but reporting difficulties arise for the same reasons described above: the codes exhibit significant stochastic behavior, where race conditions and inherent system variability can result in differing numbers of sub-problems examined from run to run.

Since program decisions are made in real time and the discovery of a strong bound can eliminate from consideration a large number of subproblems, the *order* of discovery of "good" bounds can greatly affect the proportion of the search tree explored. This can result in “anomalies” where a parallel algorithm using p_2 processors can take more time than one using $p_1 < p_2$ processors, or can achieve speed improvements greater than p_2/p_1[30]. Our experience with shared-memory MIMD branch-and-bound codes is that significant variation in parallel execution times can result from the minute timing differences present even in dedicated machine environments.

Also of interest is the difficulty of identifying a reasonable one-processor base case for comparison purposes. Simply executing one processor’s portion of a parallel code will yield a different order of tree traversal than the parallel case, potentially making the two results not comparable. On the other hand, simply directing a serial code to examine the tree in the same order as the parallel code does causes it to miss subproblem reoptimization opportunities, thereby increasing its run time. (See the related discussion in section 3.2.3.)

2.1.3. Distributed-Memory Network Algorithms

Some algorithms and problems are easily divisible, or *scalable*, and can more readily take advantage of distributed systems. On loosely coupled parallel systems, whether SIMD or MIMD, massive or non-massive, programmers must be concerned with the communication between processors and typically must contend with relatively small local processor memories (although the latter may change in future computer designs).

Different algorithmic approaches have been used to capitalize on this architecture. Notably, relaxation and row-action methods have been applied to nonlinear[24] and linear network models[31,38] and stochastic programming problems with network recourse[39,48]. In the SIMD codes developed for those applications and implemented on massively parallel machines, each network node is assigned a processor to manage supply and demand, and each arc is represented by two processors, one at the arc’s head and the other at its tail, to handle computation of the arc flows and duals. Efficient organization of data within the machine is crucial, since information must be passed between processors via the interconnection network, and communication speed between adjacent processors is much faster than nonadjacent message-passing and global broadcasting.

Unlike the previous applications, the synchronous nature of SIMD machines leads to uniform timing of events across different executions. Because of the extremely limited nature of individual processors and their memories in current systems, the impact of parallelism relative to a serial implementation has been difficult to determine. A linear problem with 50,000 nodes and one million arcs was solved in under 15 minutes on a 32,768-processor system[31].

2.2. Reporting Of Computational Testing

Much work has been accomplished in constructing a set of guidelines for reporting on computational experimentation, particularly in the area of mathematical programming. Following a series of early articles[17,26], the classic work by Crowder, Dembo, and Mulvey[13] provides reporting guidelines for computational experiments that have been adopted by a number of scholarly journals. Unfortunately, these recommendations were written before parallel processing systems became generally available and, therefore, did not address multi-processing issues. A recent follow-up report by Jackson, Boggs, Nash, and Powell[25] extended the topics covered in [13]. Relevant to this paper are its sections on choosing performance measures and reporting of computational tests on machines with advanced architectures.

In its discussion of performance measures for evaluating mathematical programming software, Jackson et al. state that an efficiency measure should reflect computational effort and solution quality. "(A)uthors [should] state clearly what is being tested, what performance criteria are being considered, and what performance measure is being used to draw inferences about these criteria. ... (R)eferees should bear in mind that performance measures are summary statistics and, as much as possible, should conform to all of the accepted rules regarding the use thereof."

In the sections that follow, we explore parallel performance measures and attempt to determine some acceptable rules for their use. Because of the differences in metrics used to describe shared-memory and distributed-memory applications, we devote a separate section to each category.

3. REPORTING EXPERIMENTS ON SHARED-MEMORY PARALLEL COMPUTERS

Of interest are measures of the improvement over traditional serial computing achieved by the use of multiple processors. The measures are influenced by the type of machine studied, since the architectures of parallel systems are quite varied. We will address metrics for the most prevalent class of commercial parallel design, tightly coupled MIMD.

3.1. What Is "Time?"

Since the rationale for parallelism is time-based, it is reasonable that a performance or efficiency measure be temporal as well. With single-processor systems, a common performance measure is the *CPU time* to solve a problem; this is the time the processor spends executing instructions in the algorithmic portion of the program being tested, and typically excludes the time for input of problem data, output of results, and system overhead activities such as virtual memory paging and job swapping. CPU time for a job is maintained by the operating system software, since many jobs may be sharing the same processor; the programmer uses this to compute that portion pertaining to the algorithm under study.

In the parallel case, time is not a sum of CPU times on each processor nor the largest across all. Since the objective of parallelism is real-time reduction, time must include any processor waiting resulting from an unbalanced workload and any overhead activity time. Hence the most prudent choice for measuring a parallel code's performance is the *real (wall clock) time* to solve a particular problem. Since this conservative measure includes all system paging, job swapping overhead, it is preferable that timings be made on a dedicated or lightly-loaded machine.

3.2. What Is "Speedup?"

The most common measure of the performance of an MIMD parallel implementation is speedup. Based on a definition of time, speedup is the ratio of serial to parallel times to solve a particular problem on a given machine. However, using different assumptions, researchers have employed several definitions for speedup in their reporting.

3.2.1. Speedup Definitions

Definition 1: Speedup. The *speedup*, *S*(*p*), achieved by a parallel algorithm running on *p* processors is defined as:

$$S(p) = \frac{\begin{array}{c}\textit{Time to solve a problem with the fastest serial code}\\ \textit{on a specific parallel computer}\end{array}}{\begin{array}{c}\textit{Time to solve the same problem with the parallel code}\\ \textit{using p processors on the same computer}\end{array}} .$$

For example, assume the fastest serial time is 100 seconds for a specific problem on a parallel machine, and a parallel algorithm solves the same problem in 20 seconds on the same machine using six processors. The speedup from this experiment would be $S(6) = 100/20 = 5$. *Linear speedup*, with $S(p)=p$, is considered an ideal application of parallelism, although *superlinear* results, with $S(p)>p$, are possible in some instances[2,3]. (See discussion in section 3.2.3.)

Definition 2: Relative Speedup. Some researchers use *relative speedup* in their reporting, defined as:

$$RS(p) = \frac{\textit{Time to solve a problem with the parallel code on one processor}}{\textit{Time to solve the same problem with the parallel code on p processors}} .$$

This should be used in cases where the uniprocessor version of the parallel code dominates all other serial implementations. Unfortunately, some papers interpret relative speedup as speedup when the serial case is not dominant, leading to erroneous claims of efficiency.

Definition 3: Absolute Speedup. The use of *absolute speedup* has also been proposed[44] to compare algorithms:

$$AS(p) = \frac{\textit{Fastest serial time on any serial computer}}{\textit{Time to execute the parallel code on p processors of a parallel computer}} .$$

The rationale for $AS(p)$ reflects the primary objective of parallel processing: real-time reduction. While this definition goes to the heart of the matter, it restricts research to those individuals with access to the fastest serial machine and cannot be determined until all relevant serial algorithms have been tested on all high-end systems, since “fastest” may be dependent on a particular combination of algorithm and machine for each problem tested.

3.2.2. Related Metrics

Another measure of the performance of a parallel implementation is *efficiency*, the fraction of linear speedup attained:

$$E(p) = \frac{S}{p}$$

where $S = S(p)$, $RS(p)$, or $AS(p)$. This is speedup normalized by the number of processors, and $E(p)$=1 for linear speedup. Note that, since $E(p)$’s value is a function of the definitions used for speedup and time, this normalized speedup is susceptible to all of the same reporting concerns and difficulties as the other performance metrics detailed in this report.

Incremental efficiency has also been used in reporting, defined as:

$$IE(p) = \frac{(p-1) \wp (\textit{Time for the parallel code on } p-1 \textit{ processors})}{(p) \wp (\textit{Time for the parallel code on p processors})}$$

where $p > 1$. This value shows the fraction of time improvement from adding another processor, and will be 1 for linear speedup. This variant of relative speedup has been used where one-processor times are unavailable[42].

Section 5 describes additional speedup and efficiency measures used as performance metrics for distributed-memory systems.

3.2.3. Is Superlinear Speedup Possible?

Although superlinear speedup—with $S(p)>p$—has been reported[3,34], its existence is still debated. Those that say that it is not possible note that if a particular problem can be solved in time t on p processors, then simulating the parallel code on one processor will yield a serial time of pt. Hence speedup is at most linear.

The other side of the argument is based on the belief that it is unfair to choose the "best" serial code, and tuning strategy, for each problem instance. That is, the best serial case should be chosen prior to particular problem instances. In this situation, superlinear speedup is possible. For example, with a parallel branch-and-bound code, one processor may find a good bound early in the solution process and communicate it to other processors for truncation of their search domain, possibly resulting in superlinear speedup.

It should also be noted that emulating a parallel environment on a serial machine is a difficult, and perhaps impossible, task. A naive approach would be to execute the parallel code as a set of interacting processes on a single processor, letting the operating system allocate time and other system resources. While such an arrangement would approximate a true parallel execution, it does not, for example, ensure the same ordering of events or replicate interprocess communication delays and resource contentions.

3.3. Issues In Reporting the Results of Parallel Testing

As with serial software, but even more so in parallel testing, performance measures abound, and a researcher must choose a small subset of measures to summarize succinctly the experimentation to draw conclusions about the underlying algorithm. The number of reported measures are limited by publication space restrictions, and the potentially large number of values for p.

Hence researchers must (1) select appropriate measures, and (2) use the measures in an objective way to accurately reflect the behavior of the algorithm. Although easily stated, these are surprisingly complicated tasks, especially when studying the behavior of parallel codes.

3.3.1. Choosing a Measure for Parallel Implementations

Jackson et al. recommend that "...when comparing a parallel algorithm with a scalar method, it is preferable to compare the parallel method not only with its scalar specialization, but also with the best scalar methods. In addition to reporting absolute speed-ups, times normalized by the number of processors are desirable." The authors clearly prefer $S(p)$ over $RS(p)$, and encourage the use of $E(p)$. (This would avoid instances such as [42] in which a parallel network code was compared with a slow serial code[28], yielding spectacular "results." Further analysis in [3]—this time using an efficient serial code—showed there was an average 3-processor speedup of only 1.4.)

But "gray areas" in reporting may result from ambiguity in the definitions of time and speedup. What portion of the system overhead time (for example, for process creation and termination) should be included in the serial and parallel results? If real time is used for the parallel algorithm, should this also be used for the serial, even though we have more accurate serial data available regarding the processor time spent executing algorithmic steps?

A more slippery question turns out to be: What constitutes the base, serial case? Since most implementations have "tuning" parameters, such as multipricing options, reinversion frequency, and tolerances—each of which influences a problem's solution time—how should these be set? It is widely known that, in many cases, experimentation to determine good values for the parameters can result in a significant reduction in execution times. If a researcher wishes to use the best possible serial time, how much testing with different *strategies* (parameter sets) should be performed? Further, does each code and strategy combination create a new algorithm to be considered separately?

Determining the parallel time involves the same question, further complicated by the fact that the value of individual strategies varies with p, the number of processors. A strategy that works well for a given p does not necessarily work well for a different number of processors. Should testing of numerous strategies be performed for each instance of p to be reported, and how extensive should such testing be? Or should a fixed strategy be used for all values of p? Should prescribed strategy formulas that vary with p be used instead?

Complicating matters further is the stochastic nature of some parallel algorithms. Many iterative procedures not only have multiple paths to a problem's solution, but the path chosen may be non-deterministic, due to timing-dependent decisions (race conditions) in the algorithm design. For example, multiple executions of our parallel network simplex codes[3,5,22] typically yield different solution times and number of pivots when applied to the same problem with the same strategy, under virtually identical operating conditions. In some cases, differences of thousands of pivots and 15% time variations were observed. This is due not only to alternate optima, but to slight differences in timings of events, resulting in different incoming variables, tie-breaking choices, and, therefore, a different sequence of extreme points traversed.

So how is this to be tested and reported? Should multiple runs be performed for each combination of code, problem, strategy, and number-of-processors? Should all resultant timings be reported or summarized in statistics? If a researcher wishes to determine speedup, should the best, worst, or average times be used? (Averages would require a new definition of speedup). Some researchers always use the best times, arguing that these show the actual capability of the code; is this reasonable? We attempt to answer these questions in the sections that follow.

3.3.2. Potential Sources of Bias in Reporting Parallel Results

As is evident from the previous discussion, many choices must be made in the design and reporting of experiments with parallel codes. This being a relatively new area of research, there are few generally accepted answers to the questions posed. Analysis of data variation is central to statistically designed experimentation[1,27,33], but in dealing with speedup there is variation in the components needed to compute this statistic, which is a different issue.

Also, since speedup is a ratio of serial to parallel time, we have the following observation:

Observation: The longer the serial time, the greater the parallel speedup, and vice versa.
Evidence: From inspection of the $S(p)$, $RS(p)$, and $AS(p)$ definitions. ∎

So while fast single-processor times highlight the strength of the serial code, they can produce unimpressive parallel speedups. Conversely, a slow serial time can yield seemingly spectacular parallel results. Hence it is a simple matter to influence (inadvertently or deliberately) the outcome of an experiment employing speedup as a performance measure through the choice of serial and parallel strategies. An advantageous set of strategies can, therefore, positively skew the research findings (of course a disadvantageous set would have the opposite effect).

This has motivational implications for the level of effort expended in exploring alternate strategies. Nominal serial testing may be rewarded with strong parallel results, while a more thorough search for the best one-processor strategy could only downgrade the parallel findings.

3.3.3. An Example of Difficulties in Interpreting Reported Speedup

As noted previously, [35] reports on computational experimentation with a parallel network simplex code. The machine employed was a BBN Butterfly, a loosely coupled MIMD system, in which each of the 14 processors has its own “local” memory and can also access other processors’ “remote” memories via a switching network.

In the code, only the pricing step was parallelized: each processor priced the arcs in its local memory and communicated the best candidate found to the others. All processors then executed the same pivot. To determine the serial times, one processor was used for computation, and the arc data was distributed across all processor memories. Hence most data was accessed remotely in serial testing.

On the Butterfly, remote data access takes 20 times longer than local. Hence, on average, pricing an arc in the serial tests was much slower than in the parallel runs—since local memory was used only a fraction of the time.

The reporting difficulty is: What portion of the (relative) speedup comes from the application of parallelism, and what portion is due to the use of slower memory accesses in the serial case? While [35] did not address this specific issue and simply reported speedups, the implementation was described in sufficient detail to permit identification of the ambiguity. Empirical testing would be required for a definitive answer.

3.4. How Should We Approach These Issues?

The preceding sections illustrate some of the difficulties that are encountered when attempting to objectively summarize experimentation with parallel codes. While we made what we considered reasonable decisions for our research papers, we also sought the insight of others in the research community in hopes of finding clear-cut answers or, at least, a consensus on some of the issues.

4. A SURVEY OF EXPERTS

To get a "sense of the community," we invited a group of computationally oriented mathematical programming researchers to participate in a survey. The design objectives for the survey were: (1) to address definitional issues regarding the speedup of parallel algorithm implementations, (2) to help identify a consensus regarding the usefulness of speedup as a measure for reporting parallel performance, and (3) to elicit a high response rate.

To meet these objectives, we constructed a series of simple examples, accompanied by six short-answer questions. Each multiple-choice question included a user-definable response. Comments were welcomed on each question and on the survey as a whole, and respondents could remain anonymous. Commonly used speedup definitions were included for terminological consistency.

The survey was sent to 41 researchers, and 23 completed forms were returned (see acknowledgments), a strong 56% response rate. Our selections are not included in the totals, but in the accompanying discussion. The following sections explore each question, and summarize and comment on the participants' responses. Also included are selected, unattributed comments from consenting respondents.

4.1. Question 1: Effects of Tuning Parameters

The first survey question is shown in Figure 4. The issue involved is: How should times from different strategies be used in computing speedup? Code B represents a competing algorithm that is, for the most part, dominated by Code A.

Question 1. Two optimization codes, A and B, are used to solve the same problem on the same parallel machine and identify the same optimal solution. Each code has a “tuning” parameter which is determined for each run using a “strategy” that fixes the parameter based on problem size and/or number of processors. Runs are made using four different strategies, giving the following results.

Code	A	A	A	B
Strategy	W	X	Y	Z
Serial Solution Time	100	90	200	150
Parallel Solution Time (with two processors)	60	70	50	100

The authors of code A wish to report speedups. What value for two-processor speedup should be reported? (Please mark your choice.)

☐ (a) $\frac{100}{60} = 1.67$ Strategy W (Both cases “good”)

☐ (b) $\frac{90}{50} = 1.8$ Use the best individual times.

☐ (c) $\frac{90}{70} = 1.29$ Strategy X (Fixed strategy with best serial)

☐ (d) $\frac{200}{50} = 4.0$ Strategy Y (Best two-processor time)

☐ (e) $\frac{\frac{100}{60} + \frac{90}{70} + \frac{200}{50}}{3} = 2.32$ Average speedup across all A strategies tested.

☐ (f) Other: Rationale:

Figure 4. Question 1: The Effects of Tuning Parameters

Implicit in the choice to be made are several fundamental questions: Does a change in strategy form a different algorithm? Should the same strategy be used for both serial and parallel times? Should only the best times across all tested strategies be used? Should we average the speedups, or something else?

4.1.1. Survey Results

The distribution of answers is depicted in Figure 5. Of the “Other” responses, 58% wanted to include the entire table of times, 34% computed speedup for each code and strategy combination and included all values or the range, and 6% proposed a different calculation.

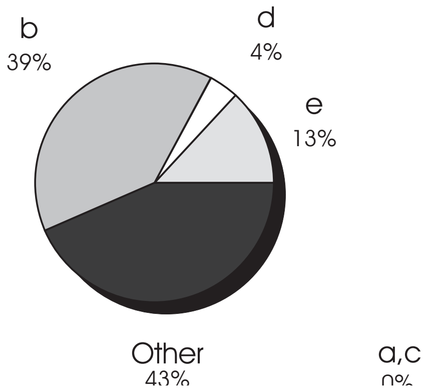

Figure 5. Question 1 Responses

Selected comments: (1) “The single number cannot capture all of the relevant information.” (2) “The strategies perform differently enough to suggest three codes: AW, AX, and AY.” (3) “There really isn’t an appropriate summary of this data.” (4) “[Use (e) and] report the standard deviation also.” (5) “Reporting raw data as well as speedup is important.”

4.1.2. Commentary

The leading selection was “Other,” with a majority of those respondents wanting to include all data in reports. $S(2)$ was the dominant summary measure [choice (b)], comparing the best individual parallel and serial times across all codes and strategies, and the remaining responses varied widely.

While we sympathize with the desire for all of the raw data, the sheer volume of such data generated by conscientious experimentation can become overwhelming. For example, a small test bed might consist of 50 representative test problems, to be examined on 1 to 20 processors, with, say, 20 reasonable strategies in each case. This results in 20,000 combinations to test for a single code, ignoring the fact that multiple instances of each combination may be required due to variations in timings or demands of a rigorous experimental design[1,27]. Even with only 4 values for p, and an exploration of 10 strategies, 2,000 problems must be run. And if the problems are substantial enough to demand the application of parallel processing, the total processing time (especially the search for the best serial case) makes the testing, much less the reporting, impractical.

From our viewpoint, an algorithm definition includes the strategy; the two notions should not be separated in reported results. Hence, we concur with comment (2), and believe that the times for a given code and strategy should be kept together. The strategy may be dynamic with the number of processors or problem characteristics, but must be rule-based and documented in the experimentation reports.

The difficulty then becomes: How to identify good code and strategy combinations? Conscientious researchers will work diligently to find a combination that has both strong serial and robust parallel performance across a wide range of problems. For all of these

Question 2. Here we have the same scenario, but different results. In this instance, B is a serial code.

Code	A	A	B
Strategy	W	Y	Z
Serial Solution Time	100	90	80
Parallel Solution Time (with two processors)	60	70	N.A.

The authors of code A wish to report speedups. What value for two-processor speedup should be reported? (Please mark your choice.)

☐ (a) $\frac{100}{60} = 1.67$

☐ (b) $\frac{90}{70} = 1.29$

☐ (c) $\frac{90}{60} = 1.5$

☐ (d) $\frac{80}{60} = 1.33$

☐ (e) $\frac{80}{70} = 1.14$

☐ (f) Other: Rationale:

Figure 6. Question 2: Basic Definition of Speedup

reasons, we would have picked responses (a) or (c) or, in the absence of other test problems, would have devised a hybrid strategy from X and Y that would have yielded the experts' leading choice, (b).

4.2. Question 2: Definition of Speedup

Figure 6 depicts the survey's second question which concerns the proper definition and calculation of speedup. Serial code B, which only has a single strategy Z, has the fastest serial time on the problem from question 1. Code A's parallel times vary with strategy. What is the speedup of code A?

4.2.1. Survey Results

The response percentages are given in Figure 7. Most of the "Other" responses had the same answer as on question 1, and for the same reasons.

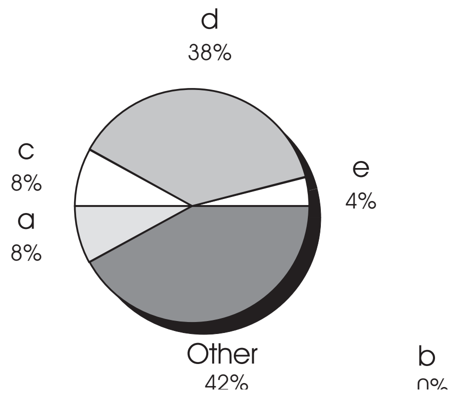

Figure 7. Question 2 Responses

Selected comments: (1) “Present the table. The single number, speedup, cannot capture all of the relevant information. In this case it would be better to report the details. In the text of the paper, one could mention a range 90-200 of serial times and 50-70 of parallel times.” (2) “More problems should be tested here.”

4.2.2. Commentary

This question highlights the main objective of parallel processing, namely, reduction of the real time required to solve a given problem. Here, a serial code exists which runs faster than the one-processor parallel code, hence better speedups could be reported by ignoring the existence of B. However, that would certainly be misleading, since code B should be used for the serial time.

The majority of respondents selecting a speedup metric chose one based on serial code B [(d) or (e)], with which we concur. We also feel that, if at all possible, results from a well-known code, such as MINOS[37] or NETFLO[28], should be included in reports. Such reference codes give the reader an awareness of the overall efficiency of all software being tested, and is of particular value on new, relatively unfamiliar technology.

4.3. Question 3: Effect of Timing Variations on Speedup

The third question, shown in Figure 8, focuses on a single problem, code, and strategy combination, illustrating the stochastic nature of both parallel and serial testing. Repeated executions of this combination under identical system conditions show variability in both the serial and parallel timings. Variation from the timing mechanism affects all values, and the larger parallel variability is due to time dependencies in the algorithm. How is speedup to be computed in this more realistic setting? (The italicized annotations in Figure 8 were not on the survey itself.)

Question 3. The times for code A using strategy Z dominate all other code and strategy combinations. However, multiple executions of the same algorithm in the same circumstances gave different times. Differences in serial times are due to variability within the computer's timing mechanism, and the (larger) differences in parallel times are due to timing-dependent choices (race conditions) in the algorithm. The following results come from 14 runs of code A with strategy Z.

	Serial Runs	2-Processor Runs
Solution times	98,99,100,100,	50,50,60,60,
(7 observations each)	100,101,102	60,70,70
Mean time	100	60
Min, Max time	98, 102	50, 70

What value for two-processor speedup should be reported for code A? (Please mark your choice.)

☐ (a) $\frac{100}{60} = 1.67$ *Ratio of means*

☐ (b) $\frac{98}{60} = 1.63$ *Best individual serial over mean parallel*

☐ (c) $\frac{100}{50} = 2.0$ *Mean serial over best parallel*

☐ (d) $\frac{98}{50} = 1.96$ *Best serial over best parallel*

☐ (e) Mean of $\left(\frac{100}{50}, \frac{100}{50}, \frac{100}{60}, \frac{100}{60}, \frac{100}{60}, \frac{100}{70}, \frac{100}{70}\right) = 1.69$

Mean of speedups with mean serial base case

☐ (f) Mean of $\left(\frac{98}{50}, \frac{98}{50}, \frac{98}{60}, \frac{98}{60}, \frac{98}{70}, \frac{98}{70}\right) = 1.66$

Mean of speedups with best serial base case

☐ (g) Other: Rationale:

Figure 8. Question 3: Effect of Timing Variation

4.3.1. Survey Results

Figure 9 shows the response percentages. Of the "Other" responses, 63% wanted some measures that reflected variability, 25% suggested the ratio of means or medians, and 12% wished to report the raw data.

Selected comments: (1) "The greater variance in the two-processor times is an important part of the data." (2) "Also, the standard deviation should be used [along with (e)]." (3) "For

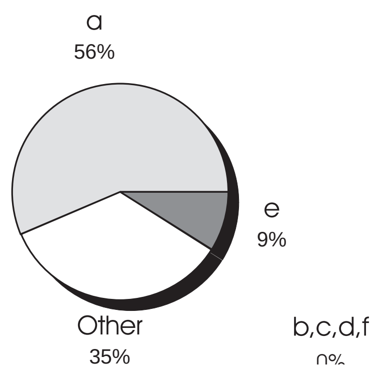

Figure 9. Question 3 Responses

serial time, use the average—errors are due to timing anyway.” (4) “Should provide variance measures since that is the purpose of the study.”

4.3.2. Comments

This question yielded the clearest consensus thus far, with 57% of the respondents choosing (a), the ratio of mean times, and many indicated a desire for a supplementary indicator of variability. On this question we differ with the respondents. We feel that (e) is a similar, but slightly more correct, choice for the following reasons. The serial variation is due to random measurement error, and hence should be averaged to form the base case, per responses (a), (c), and (e). With this base, speedup can be computed for each two-processor time and averaged to give the mean speedup ratio, instead of the ratio of mean times which does not follow any of the standard speedup definitions. This would also permit reporting speedup variability measures such as the standard deviation and range. We note that the harmonic—not the arithmetic—mean of the ratios should be taken (see [27], p. 188, for a full discussion), and is:

$$\ddot{x} = \frac{7(100)}{50+50+60+60+60+70+70} = 1.667.$$

The situation underscores the difficulty in reporting all of the raw data, and leads to the question: How many instances of each combination of problem, code, strategy, and number of processors should be run? The testing of five instances of one code on 50 problems, with 10 strategies, and four processor settings, involves 10,000 runs; practicalities will likely force compromises on such an experimental design.

4.4. Question 4: Degree of Effort on Serial Case

Because of the crucial role of the serial time in computing speedup, the level of effort expended to identify a “best” value is a significant factor in reporting. Respondents expressed the importance of such experimentation on a scale of 1 to 10 (see Figure 10).

Question 4. How much effort should be spent identifying the “best” serial time for a given problem? (For example, with a simplex-based algorithm, how much testing should be performed to identify the best pivot strategy?) Please express your answer on a scale from 1 to 10 where 1 = minimal testing and 10 = as exhaustive as possible.
Your answer: ________________

Figure 10. Question 4: Effort for Serial Case

4.4.1. Survey Results

Of the 83% that gave a numerical answer, the “effort” statistics are: mean, 5.8; median and mode, 5; standard deviation, 2.9; this distribution is shown in Figure 11. The 17% nonrespondents said that the answer depended on the purpose of the experimentation.

Selected comments: (1) “[Expend] just as much effort as would be done for the parallel code.” (2) “Use standard setting of parameters.” (3) “Be reasonable. Give arguments as to why you made the choice. Realize that performance is highly problem-dependent and there may not be a `best’ serial version.” (4) “In many cases it may be preferable to compare with a `standard’ algorithm (e.g., MINOS for simplex).” (5) “Difficult to answer, depends on reason for study.”

4.4.2. Comments

With a full range of values, and slightly left-skewed distribution, the responses indicate that a reasonable amount of testing should be performed. We feel that our testing has been in the exhausting, but not exhaustive, 8 to 9 range. Comment (1) seemed appropriate, but

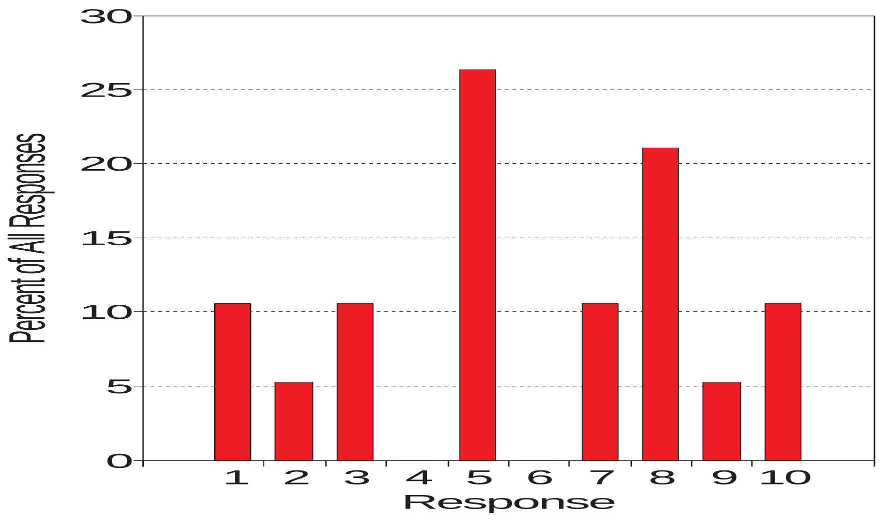

Figure 11. Distribution of Responses to Question 4

Question 5. The solution strategy used in a code can strongly affect execution times. When reporting results of testing a code on a given problem with different numbers of processors, the reported results should be based on a strategy which is:

- ☐ (a) Fixed, invariant of the number of processors
- ☐ (b) Rule-based, and can vary with the number of processors and problem parameters
- ☐ (c) The fastest of all tested for each number of processors
- ☐ (d) Other:

Figure 12. Question 5: Setting Solution Strategy

we are reminded that the best code and strategy combination tends to vary from problem to problem and with number of processors used (that is, the fastest for two processors is not necessarily fastest with six).

4.5. Question 5: Setting Strategies

The question posed in Figure 12 addresses the means by which strategies should be determined for reporting purposes. Implicit is the issue of whether researchers should be able to determine a unique strategy for each problem and processor combination and report the results of the best that was found empirically.

4.5.1. Survey Results

The response percentages are given in Figure 13. Of the "Other" respondents, 43% said "any," 29% "(a) and (b)," 14% "all", and 14% said that it depends on the research objective.

Selected comments: (1) "Report (a) and (b). The more the merrier." (2) "[Rule-based,] but part of the program." (3) "More important is to explicitly state which of the above was used."

4.5.2. Comments

Respondents are closer to a consensus on this issue, with a strong majority preferring a rule-based strategy, rather than individually tuned ones, and we agree. We note that results can be easily biased if choice (a) is employed—simply choose a good multi-processor strategy. In our experiments with the network simplex, a strategy that performed well in parallel typically worked poorly serially; hence using such a fixed strategy could result in dramatic speedups. Choice (b) seems the most fair, but leaves open the question of how to devise the rule.

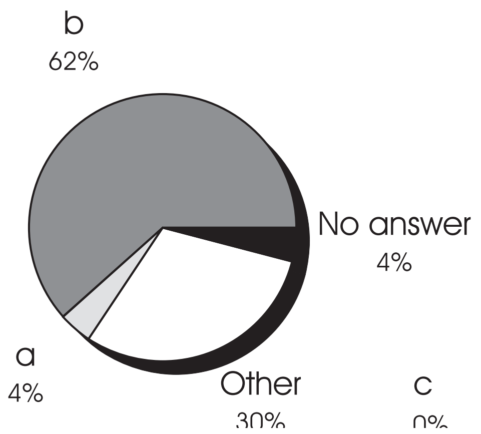

Figure 13. Question 5 Responses

4.6. Question 6: Validity of Speedup as a Metric

The last question, shown in Figure 14, allows participants to summarize their feelings about speedup's current role as the leading performance measure for reporting parallel testing.

4.6.1. Survey Results

The response percentages are shown in Figure 15. Suggested alternatives or additions included measures of cost-effectiveness, efficiency, robustness, quality of solution, and chance of catastrophic error.

Selected comments: (1) "I tend to be skeptical about one number measuring the goodness of an algorithm." (2) "No, but it is attractive to boil performance down to a single number, so it will likely continue as the dominant measure." (3) "[Use] a number of performance measures, just as we learned when dealing with serial algorithms." (4) "There must be a better method. But I do not know it."

4.6.2. Comments

A slight majority begrudgingly accepted speedup as the primary parallel performance measure but, as revealed in the comments, would prefer a better one. Several indicated the need for variation and cost-effectiveness to be represented in reportings.

Question 6. Should speedup be used as the primary measure of performance of a parallel algorithm?

- ☐ (a) Yes
- ☐ (b) No, we should use:

Figure 14. Question 6: Efficacy of Speedup as Metric

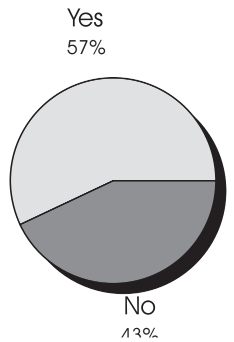

Figure 15. Question 6 Responses

4.7. Overall Comments

The following quotations were selected from the general comments of consenting respondents regarding the questionnaire or the reporting of parallel experimentation.

“Rating the effectiveness of a parallel algorithm by a single measure, speedup, seems analagous to describing a probability distribution by reporting only its mean.”

“As a rule, authors of a code will present the data so as to make their code appear best. That is human nature. More important is to explicitly state how they are reporting and how testing was performed, i.e., acknowledge their biases.”

“The value of parallelism is an economic issue or real time speedup issue. ...Without the cost of the parallel system the benefits of speedup are meaningless.”

“Also we should report the actual times—not only speedups—and on problem sizes where speedup is of the essence.”

“After the initial excitement of actual implementation of conventional or new algorithms on parallel machines, speedup factors are going to lose their allure. ... However, if we show that what took an hour on a $10 million superframe now takes 15 minutes on a $500K multicomputer, it will have a significant impact whatever the speedup factor is.”

“Rules to follow: 1) Avoid point samples, i.e., solve each problem instance several times and solve many problem instances. 2) Summarize data in more than one way. Be willing to report negative results.”

“More people should think of these important details.”

4.8. Survey Conclusions

The survey of leading researchers in computational mathematical programming in most cases did not yield a clear consensus. The large number of “Other” responses may be due in part to independent-thinking participants, ambiguities in the questions and answers, or simply the lack of obvious or appealing solutions to the situations posed. Even so, a general level of agreement was reached on the following.

- The use of speedup as a parallel performance measure is tolerable, but more than one measure of parallelism effect is desirable in reporting computational results.
- Measures of variation and cost-effectiveness are important also.
- Report as much of the raw data as possible.
- A rule-based strategy should be used when reporting results.

We also feel from the survey and our parallel testing experiences that:

- No strong consensus exists regarding the method for summarizing a large body of data.
- While many survey participants wanted all raw data reported, this is clearly not feasible when there are many combinations of problem, code, strategy, numbers of processors, and numbers of repetitions. We must continue to work towards better numerical and graphical methods for summarizing the data.
- Reference values from well-known codes are needed, particularly in the serial case.
- Experimental design procedures should be used in reporting to accommodate the variability in timings and to add rigor to the process.
- Even more complications will arise when using speedup as the response variable in a statistical experimental design.
- We should focus on the real time and cost required to solve difficult problems.

5. REPORTING ON DISTRIBUTED SYSTEMS TESTING

SIMD machines and distributed-memory MIMD machines are distinguished from shared memory machines in that there is no main memory which is globally accessible by all processors. To date, massively parallel systems have tended to have a large number of low-power processors; for example, the SIMD CM-2 uses thousands of 1-bit processors, with small local memories[23]. These characteristics complicate the computation of speedup in a distributed memory environment.

Since speedup is typically computed as the ratio of the single-processor to the multiprocessor times to solve the same problem, to compute speedup on such a distributed memory system the size of the problem must be restricted to one which will fit into the memory associated with a single processor. This clearly places severe limitations on the magnitude of problems on which speedup (in the traditional sense) may be reported. This restriction has led researchers to revert to earlier simpler metrics, to derive new ones, and to formulate alternative speedup models.

5.1. Performance Metrics Used with Distributed Optimization

One metric reported is the *MFLOPS* (*million floating-point operations per second*, or *megaflops*) rate. Typically the number of megaflops sustained by a program is given along with the peak MFLOPS possible on that machine. This measure is an indicator of how well a particular implementation of an algorithm exploits the architecture of the machine, but, as Bertsekas, Castañon, Eckstein, and Zenios[8] note, it "...does not necessarily indicate whether this is an efficient algorithm for solving problems. It is conceivable that an alternative algorithm can solve the same problem faster, even if it executes at a lower MFLOP rate." (Note that, as technology progresses, the measure is shifting to *gigaflops* and *teraflops*, refering to billions and trillions of operations per second, respectively.)

Another common metric is the *real time* required to solve a given problem. It is valuable in that it provides a fairly unambiguous account of the efficiency of an implementation of an algorithm on a particular machine (assuming variability in timings is not an issue). Often only one set of times is reported; that is, each problem is solved for only one value of p, the number of processors. For example, [38] reports times required to solve network problems on a 16,384-processor CM-2 and [31] reports times required to solve similar problems on a 32,768-processor CM-2. This information is useful for comparative purposes to researchers who are using the same machine, and to those interested in absolute speed. However, with times for only one value of p, researchers would be unable to extrapolate regarding the value of additional processors.

To date, these have been the only performance measures used in reporting on SIMD implementations of parallel optimization algorithms, besides algorithm-specific statistics such as the number of major and minor iterations, and portion of time spent performing certain steps. Some loosely coupled MIMD applications report relative speedup[45,49].

In the next section, we describe a series of performance metrics which have been proposed for parallel testing. Some of these have been used for the comparison of computer systems, and have been adopted for the comparison and testing of algorithms.

5.2. Additional Performance Measures for Distributed Algorithms

In some instances, times may be available for several different values of p, but not for p=1. Consider, for example, a 1,024-processor distributed-memory machine. Although a problem may be too large to run on one processor, it may be solvable on, say, 64 processors. It could then perhaps also be solved using 128 processors, then 256, 512, and 1,024. In [39], times are reported for solving linear stochastic programs on a 32,768-processor CM-2 using both 16,384 processors and 32,768 processors. In such cases traditional speedup cannot be computed.

We propose an alternate model to quantify the value of additional processors, termed *generalized incremental efficiency*, computed as follows:

$$GIE(p,q) = \frac{(p) \cdot (Time\ to\ execute\ parallel\ code\ on\ p\ processors)}{(q) \cdot (Time\ to\ execute\ parallel\ code\ on\ q\ processors)} \ .$$

This value shows the fraction of linear speedup which was attained by increasing the number of processors from *p* to *q*. A value of 1 indicates linear speedup. For example, [39] reports that "...doubling the number of processors from 16K to 32K decreases the solution time by a factor of almost 2." In this case *GIE(16K,32K)* would be close to 1, indicating near linear speedup. (The concept of incremental efficiency was reported by Peters[42], but was limited in that it only measured the value of adding one processor at a time. In terms of generalized incremental efficiency, this may be expressed as: *IE(p) = GIE(p-1,p)*.)

While generalized incremental efficiency does indicate the value of additional processing elements, it does not allow one to fully utilize all memory available. (All processors would be required to solve a problem which required all available memory. Therefore no comparisons could be made for different values of *p*.) One model which addresses this issue is *scaled speedup*[21,36,47]. In this model, problem size increases with *p*. If we assume that the largest problem which may be stored in the memory associated with one processor is of size *n* and that storage is proportional to *n*, then *p* processors should be able to solve a problem of size *np*. Therefore the scaled speedup is defined as:

$$SS(p) = \frac{Estimated\ time\ to\ solve\ problem\ of\ size\ np\ on\ 1\ processor}{Actual\ time\ to\ solve\ problem\ of\ size\ np\ on\ p\ processors} \ .$$

This metric is attractive in that it allows for full utilization of the machine resourses while quantifying the value of additional processors. Its major disadvantage is the uncertainty inherent in the numerator. Accurate estimation of the solution time based on one (or more) scaling factor(s) is impractical for many mathematical programming problems. For example, given only the time required to solve a linear program with *m* constraints, it is difficult to determine a precise estimate of the time to solve a linear program with 2*m* constraints, since there are many influencing factors and *2m* may be outside the observed experience range. Furthermore, many mathematical programming problems are not easily scalable.

Closely related to scaled speedup is *fixed time speedup*[20]. The concept of fixed time speedup is more closely associated with measuring the performance of computers than algorithms, but deserves notice in the present context. As with scaled speedup, the computation of fixed time speedup requires that the one-processor time be estimated. The difference between the two models lies in the size of the problem being solved. Scaled speedup is computed based on a problem size which is largest for the number of processors used. Fixed time speedup is computed based on a problem size which is the largest that can be solved in a particular amount of time. If we assume that the largest problem which may be stored in the memory associated with one processor is of size *n*, fixed time speedup is computed as follows:

$$FTS(p) = \frac{\textit{Estimated time to solve problem of size k on one processor}}{\textit{Actual time to solve problem of size k on p processors}}$$

where *k* is the size of the largest problem which can be solved on *p* processors in no more time than one processor can solve a problem of size *n*. As with scaled speedup, fixed time speedup shares the disadvantage of estimating the one-processor solution time. Furthermore, the process of determining *k* is imprecise at best.

Also closely related to scaled speedup is an efficiency measure termed *scaleup*. According to [14], "(s)caleup is defined as the ability of an *n*-times larger system to perform an *n*-times larger job in the same elapsed time as the original system." Therefore, this measure allows for full utilization of system resources but, unlike scaled speedup, is not based on an estimated one-processor time. It is computed as follows:

$$Scaleup(p,n) = \frac{\textit{Time to solve size m problem on p processors}}{\textit{Time to solve size nm problem on np processors}} \ .$$

Linear scaleup occurs when *scaleup(p,n)* = 1. A form of scaleup is used in [39], which reports that "...the time to solve a 1024 scenario problem on a 16K machine is almost the same as that for solving a 2048 scenario problem on a 32K machine." In this instance *scaleup(16K,2)* is close to 1, hence nearly linear.

To help explain their results, authors may wish to include statistics such as *fraction of time spent in communication* and *synchronization-related idle time*. While indispensable for correct code operation, both communication and synchronization time are considered overhead items, since no work, per se, is accomplished during that time. The inclusion of this type of measure can provide insight into the sources of inefficiency in the code and underlying algorithm.

5.3. Conclusions

Many of the newer performance metrics require an accurate model for estimating solution times for problems of a given "size" and number of processors. We feel that such a requirement is unrealistic within the domain of mathematical programming, where a large number of factors with undiscovered interrelationships determine problem solution difficulty and computational effort for a given problem.

Researchers testing massively parallel and distributed algorithms are struggling to find useful measures of the value of parallelism and the efficiency of their methods. For these settings, only the basic metrics have been reported in the optimization literature. This is likely due to the embryonic state of research in this area, the small number of standard test problems for comparative testing, and the lack of a generally accepted summary measure.

We encourage researchers in this domain to:

- Compare their results with standard codes, on other platforms if necessary, to provide a frame of reference for the timing data;
- Include system prices and address cost-effectiveness issues in solving problems of practical interest; and
- Explore the newer performance metrics, and hopefully devise better ones.

6. REPORTING ON PARALLEL EXPERIMENTS WITH HEURISTIC CODES

The objective of heuristic methods is the identification of "high quality" feasible solutions to problems more quickly than can be done with provably optimizing approaches. This permits users to obtain solutions to problems that are known to be difficult (for example, are NP-hard) or not easily described by traditional mathematical formulations (as with some routing and scheduling problems). Key application areas for heuristics have been integer programming, combinatorial, and graph theoretic models.

Many generalized approaches, or *meta-heuristics*, have emerged, including: *k*-exchange, simulated annealing, tabu search, genetic, neural network, GRASP, target analysis, ejection chain, and beam search algorithms. All of these methods are highly amenable to parallel exploitation.

Reporting on parallel implementations of heuristics has the same set of problems as with optimizing codes: stochastic results from race conditions, determination of the proper base case for speedup calculations, difficulty in summarizing a large body of data, and so on. In addition, heuristics have special reporting problems of their own:

- Because of the nature of some heuristic algorithms, including—but not limited to—those with a randomization component, serial and parallel executions are likely to return different solutions.
- Bounds on the optimal solution value can be weak or nonexistent, hence the closeness to optimality cannot be determined accurately.
- Many procedures have no standard termination rule, other than: stop when you run out of money or patience.

- Such arbitrariness permits the reporting of results using stopping rules based on preprocessing of the problems. For example, if a heuristic code is run for 10,000 iterations, but the last improving solution was found at repetition 890, then one could simply report timings based on the rule: stop after 900 iterations.
- As with optimizing codes, heuristics often have a large number of control parameters to set. A given solution strategy may depend on a dozen or more decisions, such as candidate list lengths, descent rate, number of training repetitions, tolerances, and thresholds, plus a stopping rule. Since execution times are often heavily influenced by tuning such values, researchers should document not only the parameter settings used in the testing, but the robustness of those values.

Of primary interest in reporting on computational testing of heuristic-based codes are descriptive measures of the tradeoff between time and solution quality. Does the code find a "good" solution quickly? Given enough time, does the code find a high-quality solution? For a benchmark problem, was the best known solution identified? Was a new best solution found?

Perhaps the best descriptor of heuristic performance on a problem is a graph of the best solution value found versus time. When variability results from parallelism, average and interval values can be graphed as well. Related statistics include: time to first feasible solution, time to best solution, and time to *n* iterations.

Example metrics that have been used in parallel heuristic testing[15,41] are: relative speedup; percent of cases examined where best known solution was found; minimum, average, and maximum times for *n* iterations and for various numbers of processors, *p*; and a graph of the cumulative number of problems in the test set achieving a stated quality level, versus time, for various values of *p*.

7. SUMMARY AND GUIDELINES

In recent years, great strides have been made in harnessing the power of parallel computers to solve mathematical programming problems. To assist authors, referees, and editors, we offer the following guidelines for the reporting of computational experiments with parallel codes, based on our experiences and a survey of experts in the field. They are meant to augment previous reporting standards[13,26], and be considered as work-in-progress, to be refined and enhanced as experience, understanding, and insight grow.

Parallel Reporting Guidelines

I. Thoroughly document the process

A. Describe the code being tested. This includes the algorithm on which it is based, including any modifications; the overall design; the data structures used; and the available tuning parameters.

B. Document the computing environment for the experimentation. Report all pertinent characteristics of the machine used, including the manufacturer, model, types and number of processors, inter-processor communication schemes, size of memories, and configuration.

C. Describe the testing environment and methodology.

1. State how times were measured.
2. When reporting speedup, state how it was computed. In particular, indicate what was used for the base (serial) case.
3. Report all values of tuning parameters.

II. Use a well-considered experimental design

A. Focus on the real time and cost required to solve difficult problems.

B. Try to identify those factors that contribute to the results presented, and their effects. This includes the impacts of problem characteristics, tuning-parameter strategy, and parallelism.

B. Provide points of reference. If possible, use well-known codes and problems to determine reference values, particularly in the serial case, even if testing must be performed on different machines (as required with some distributed systems).

C. Perform final, reported testing on a dedicated or lightly loaded system.

D. Employ statistical experimental design techniques. This powerful, often neglected, methodology can highlight those factors that contributed to the results, as well as those that did not.

III. Provide a comprehensive report of the results

A. For summary measures, use measures of central tendency, variability, and cost-effectiveness.

B. Use graphics where possible and when informative.

C. Provide as much detail as possible. If a journal will not publish all pertinent data—perhaps due to space limitations—make them available in a research report.

D. Describe the sensitivity of the code to changes in the tuning strategy.

E. Be courageous and include your "failures," since they provide insight also.

Parallelism holds the promise of permitting operations researchers and computer scientists to reach the previously unattainable: the routine solution of problems and the support of models that were too large or complex for previous generations of computers and algorithms. By so doing we bring the benefits of the “OR-approach”[12] to a wider audience, hopefully affecting and improving the lives of an increasingly larger portion of the world’s populace. The accurate reporting of research is central to progress and the growing understanding of how to capitalize on these new opportunities so that we might realize the fruits of the promise.

ACKNOWLEDGMENTS

We wish to thank all of the survey participants for their time and thoughtful contributions. We are also grateful to the Editor of the *ORSA Journal on Computing* for the invitation to prepare this paper and participate in this new feature of the journal. Finally, we note that some of the materials in this article appeared previously in [4].

REFERENCES

[1] M.M. Amini and R.S. Barr, 1990. Network Reoptimization Algorithms: A Statistically Designed Comparison. Technical Report 90-CSE-4, Department of Computer Science and Engineering, Southern Methodist University, Dallas, Texas (to appear in *ORSA Journal on Computing*).

[2] R.S. Barr and M. Christiansen, 1989. Parallel Auction Algorithm: A Case Study in the Use of Parallel Object-Oriented Programming. In R. Sharda, B. L. Golden, E. Wasil, O. Balci, and W. Stewart, eds., *The Impact of Recent Computer Advances on Operations Research*, North-Holland, Amsterdam, 23-32.

[3] R.S. Barr and B.L. Hickman, 1989. A New Parallel Network Simplex Algorithm and Implementation for Large Time-Critical Problems. Technical Report 89-CSE-37, Department of Computer Science and Engineering, Southern Methodist University, Dallas, Texas.

[4] R.S. Barr and B.L. Hickman, 1992. On Reporting the Speedup of Parallel Algorithms: A Survey of Issues and Experts. In O. Balci, R. Sharda, and S.A. Zenios, eds. *Computer Science and Operations Research: New Developments in their Interfaces.* Pergamon Press, Oxford, U.K., 279-294.

[5] R.S. Barr and B.L. Hickman, 1992. A Parallel Approach to Large-Scale Network Models with Side Conditions. Technical Report 92-CSE-18, Department of Computer Science and Engineering, Southern Methodist University, Dallas, Texas.

[6] R. S. Barr and W. Stripling, 1992. A Parallel Mixed-Strategy Branch-and-Bound Approach to the Fixed Charge Transportation Problem. Technical Report 92-CSE-19, Department of Computer Science and Engineering, Southern Methodist University, Dallas, Texas. (Presented at *Symposium on Parallel Optimization 2*, Madison, Wisconsin, 1990.)

[7] M.L. Barton and G.R. Withers, 1989. Computing Performance as a Function of the Speed, Quantity, and Cost of the Processors. *Proceedings of Supercomputing '89*, 759-764.

[8] D. Bertsekas, D. Castañon, J. Eckstein, S.A. Zenios, 1991. Parallel Computing in Network Optimization. Research report 91-09-02, Department of Decision Sciences, The Wharton School, University of Pennsylvania, Philadelphia (to appear in *Handbook of Operations Research*).

[9] R.L. Boehning, R.M. Butler, B.E. Gillett, 1988. A Parallel Integer Linear Programming Algorithm. *European Journal of Operational Research 34*, 393-398.

[10] N. Carriero and D. Gelernter, 1989. How to Write Parallel Programs: A Guide to the Perplexed. *ACM Computing Surveys 21*, 323-357.

[11] R.H. Clark, J.L. Kennington, R.R. Meyer, and M. Ramamurti, 1992. Generalized Networks: Parallel Algorithms and an Empirical Analysis. *ORSA Journal on Computing 4*, 132-145.

[12] T. Cook, 1991. The Challenge of the 90's. Plenary speech at the TIMS/ORSA Joint National Meeting, Nashville.

[13] H.P. Crowder, R.S. Dembo, and J.M. Mulvey, 1980. On Reporting Computational Experiments with Mathematical Software. *ACM Transactions on Mathematical Software 5*, 193-203.

[14] D. DeWitt and J. Gray, 1992. Parallel Database Systems: The Future of High Performance Database Systems. *Communications of the ACM*, 35:6, 85-98.

[15] T.A. Feo, M.G.C. Resende, and S.H. Smith, 1990. A Greedy Randomized Adaptive Search Procedure for Maximum Independent Set. Technical Report, Operations Research Program, Department of Mechanical Engineering, University of Texas at Austin, Austin, Texas.

[16] M.J. Flynn, 1966. Very High-Speed Computing Systems. *Proceedings of the IEEE 54*, 1901-1909.

[17] J. Gilsinn, K. Hoffman, R.H.F. Jackson, E. Leyendecker, P. Saunders, and D. Shier (1977). Methodology and Analysis for Comparing Discrete Linear L1 Approximation Codes. *Communications in Statistics 136*, 399-413.

[18] H.J. Greenberg, 1990. Computational Testing: Why, How and How Much. *ORSA Journal on Computing 2*, 7-11.

[19] P. Gregory, 1992. Will MPP Always Be Specialized? *Supercomputing Review, 5:3*.

[20] J.L. Gustafson, 1992. The Consequences of Fixed Time Performance Measurement. *Proceedings of the 25th Hawaii International Conference on Systems Sciences,* IEEE Computer Society, 113-124.

[21] J.L. Gustafson, G.R. Montry, and R.E. Benner, 1988. Development of Parallel Methods for a 1024-Processor Hypercube. *SIAM Journal of Scientific and Statistical Computing 9,* 609-638.

[22] B.L. Hickman, 1991. *Parallel Algorithms for Pure Network Problems and Related Applications*. Doctoral dissertation, Southern Methodist University, Dallas, Texas.

[23] W.D. Hillis, 1985. *The Connection Machine*. The MIT Press, Cambridge, Massachusetts.

[24] R.M. Hord, 1990. *Parallel Supercomputing in SIMD Architectures*. CRC Press, Boca Raton, Florida.

[25] R.H.F. Jackson, P.T. Boggs, S.G. Nash, and S. Powell, 1990. Report of the Ad Hoc Committee to Revise the Guidelines for Reporting Computational Experiments in Mathematical Programming. *Mathematical Programming 49,* 413-425.

[26] R.H.F Jackson and J.M. Mulvey, 1978. A Critical Review of Comparisons of Mathematical Programming Algorithms and Software (1953-1977). *J. Research of the National Bureau of Standards 83*, 563-584.

[27] R. Jain, 1991. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, New York.

[28] J.L. Kennington and R.V. Helgason, 1980. *Algorithms for Network Programming*. John Wiley & Sons, New York.

[29] G.A.P Kindervater and J.K. Lenstra, 1989. Parallel Computing in Combinatorial Optimization, *Annals of Operations Research 14:1*, 245-289.

[30] T.H. Lai and S. Sahni, 1984. Anomalies in Parallel Branch-and-Bound Algorithms. *Communications of the ACM 27,* 594-602.

[31] X. Li and S.A. Zenios, 1991. Data-level Parallel Solution of Min-cost Network Flow Problems Using ε-Relaxations. Report 91-05-04, Department of Decision Sciences, The Wharton School, University of Pennsylvania, Philadelphia, Pennsylvania.

[32] E.L. Lusk and R.A. Overbeek, 1985. Use of Monitors in FORTRAN: a Tutorial on the Barrier, Self-Scheduling Do-Loop, and Askfor Monitors. In Kowalik, J.S., ed.,

Parallel MIMD Computation: The HEP Supercomputer and its Applications, The MIT Press.

[33] R.L. Mason, R.F. Gunst, and J.L. Hess, 1989. *Statistical Design and Analysis of Experiments*. John Wiley & Sons, New York.

[34] D.L. Miller and J.F. Pekny, 1989. Results from a Parallel Branch and Bound Algorithm for the Asymmetric Travelling Salesman Problem. *Operations Research Letters 8:3*, 129-135.

[35] D.L. Miller, J.F. Pekny, and G.L. Thompson, 1990. Solution of Large Dense Transportation Problems Using a Parallel Primal Algorithm. *Operations Research Letters 9:5*, 319-324.

[36] C. Moler, 1986. Matrix Computation on Distributed Memory Multiprocessors. In M. Heath (ed.), *Hypercube Multiprocessors*, SIAM, Philadelphia, pp. 181-195.

[37] B.A. Murtagh and M.A. Saunders, 1987. MINOS 5.1 Users Guide. Report SOL 83-20R, Stanford University.

[38] S.S. Nielsen and S. A. Zenios, 1991. Proximal Minimizations with D-Functions and the Massively Parallel Solution of Linear Network Programs. Report 91-06-05, Decision Sciences Department, The Wharton School, University of Pennsylvania, Philadelphia, Pennsylvania.

[39] S.S. Nielsen and S.A. Zenios, 1992. Solving Linear Stochastic Programs using Massively Parallel Proximal Algorithms. Report 92-01-05, Decision Sciences Department, The Wharton School, University of Pennsylvania, Philadelphia, Pennsylvania.

[40] Parasoft, 1988. *EXPRESS: A Communication Environment for Parallel Computers*. Parasoft, 27415 Trabuco Circle, Mission Viejo, California 92692.

[41] P.M. Pardalos, A.A. Murthy, and Y. Li, 1992. Computational Experience with Parallel Algorithms for Solving the Quadratic Assignment Problem. In O. Balci, R. Sharda, and S.A. Zenios, eds. *Computer Science and Operations Research: New Advances in the Interfaces*. Pergamon Press, Oxford, U.K., 267-278.

[42] J. Peters, 1990. The Network Simplex Method on a Multiprocessor. *Networks 20*, 845-859.

[43] P. W. Purdom, Jr. and C. A. Brown, 1985. *The Analysis of Algorithms*. Holt, Rinehart and Winston, New York.

[44] M.J. Quinn, 1987. *Designing Efficient Algorithms for Parallel Computers*. McGraw-Hill, New York.

[45] C. Roucairol, 1989. Parallel Branch and Bound Algorithms—An Overview. Rapports de Recherche N° 962, Unitè de Recherche Intia-Rocquencourt, Domaine de Voluceau, Rocquencourt, B.P. 105, 78153 Le Chesnay Cedex, France.

[46] A. Schrijver, 1986. *Theory of Linear and Integer Programming*. John Wiley & Sons, Chichester.

[47] C. Seitz, 1985. The Cosmic Cube. *Communications of the ACM 28,* 22-23.

[48] H. Vladimirou and J. Mulvey, 1990. Parallel and Distributed Computing for Stochastic Network Programming. Research Report SOR-90-11, Department of Civil Engineering and Operations Research, Princeton University, Princeton, NJ.

[49] M.K.Yang and C.R. Das, 1991. A Parallel Branch-and-Bound Algorithm for MIN-Based Multiprocessors. *Performance Evaluation Review 14:1*, 222-223.

[50] S. A. Zenios, 1989. Parallel Numerical Optimization: Current Status and an Annotated Bibliography. *ORSA Journal on Computing 1:1*, 20-39.