IS--5049 DE91 007118

The Design of a Scalable, Fixed-Time Computer Benchmark

by

JOHN GUSTAFSON

DIANE ROVER

STEPHEN ELBERT

MICHAEL CARTER

Ames Laboratory^{*} Iowa State University Ames, IA 50011-3020

Date Transmitted: October, 1990

8

*Operated by Iowa State University for the U.S. Department of Energy under contract No. W-7405-ENG-82.

THE PROPERTY OF THIS COCUMENT IS UNLIN

MASTER

TABLE OF CONTENTS

Abstractiii
Chapter 1. Introduction1
Chapter 2. Performance Measurement Goals1
Chapter 3. Detailed Description5
3.1. Scalable Benchmarking5
3.2, Fixed-Time Benchmarking8
3.3. Language/Architecture Independence10
3.4. Precision Independence11
3.5. Figure of Merit
3.6. Complete Task Measurement14
3.7. Minimization of Human Effort Bias14
Chapter 4. Benchmark Results15
Chapter 5. Superlinear Speedup Effects
Definitions16
Superlinear Speedup from Changing Routine Profile17
Conclusions
Acknowledgements19
References
Distribution List

ABSTRACT

By using the principle of *fixed time* benchmarking, it is possible to compare a very wide range of computers, from a small personal computer to the most powerful parallel supercomputer, on a *single scale*. Fixed-time benchmarks promise far greater longevity than those based on a particular problem size, and are more appropriate for "grand challenge" capability comparison. We present the design of a benchmark, SLALOMTM, that scales automatically to the computing power available, and corrects several deficiencies in various existing benchmarks: it is highly scalable, it solves a real problem, it includes input and output times, and it can be run on parallel machines of all kinds, using any convenient language. The benchmark provides a reasonable estimate of the *size* of problem solvable on scientific computers. Results are presented that span six orders of magnitude for contemporary computers of various architectures. The benchmark also can be used to demonstrate a new source of superlinear speedup in parallel computers.

The Design of a Scalable, Fixed-Time Computer Benchmark*

JOHN GUSTAFSON, DIANE ROVER, STEPHEN ELBERT, AND MICHAEL CARTER

Ames Laboratory, Ames, IA 50011

1. INTRODUCTION

Computer power has increased over 70% per year for the last 50 years, or over 11 orders of magnitude. This increase makes it difficult to measure performance with a tool that does not *scale*. Furthermore, a given make of parallel processor can offer a performance range of over 8000 to 1, so the scaling problem exists even if applied to a computer of current vintage. Any benchmark of fixed size is soon obsoleted by hardware advances that render the time and space requirements of the benchmark unrepresentative of realistic use of the equipment. The common workaround of performing a fixed-size task repetitively is less than satisfactory.

A related issue is the difficulty of scientifically comparing computers with vastly different architectures or programming environments. A benchmark designed for one architecture or programming model puts a different architecture at a disadvantage, even when nominal performance is otherwise similar. Assumptions such as arithmetic precision, memory topology, and "legal" language constructs are invariably wedded to the job to be timed, in the interest of controlling as many variables as possible. This "ethnocentrism" in benchmark design has hampered comparison of novel parallel computers with traditional serial computers. Examples of popular benchmarks that have some or all of the foregoing drawbacks are LINPACK [3], the "PERFECTTM Club" [11], the Livermore Loops [9], SPEC [13], Whetstones [2], and Dhrystones [14].

Chapter 2 presents the design goals of a benchmark that attempts to solve these and other difficulties. Chapter 3 shows our techniques for achieving these goals. Chapter 4 gives experimental results for a wide range of parallel and serial computers. Chapter 5 discusses implications of the fixed-time method for superlinear speedup.

2. PERFORMANCE MEASUREMENT GOALS

Ideally, a benchmark should be scalable, broad in architectural scope, simple to apply and understand, representative of the way people actually use computers, and scientifically honest. A proper benchmark is both a task engineered to meet these goals and a set of *rules* governing the experimental procedure. It is more than just an application program or excerpt. We recognize that many of these goals are at odds with one another. As with any engineering design, a certain amount of compromise is necessary. In particular, a single benchmark with a single figure-of-merit cannot fully characterize performance for the entire range of computing tasks. However, it seems possible to restrict ourselves to large-scale scientific problems and capture salient features of that class of problems that are absent from other computer performance tests.

4

Goal: SCALABLE BENCHMARKING

It is a natural assumption that in *measuring* computer performance the problem being solved should be fixed as the computing power varies. Unfortunately, this is a dubious assumption since it does not reflect the way people actually *use* computers. Generally, problems scale to use the available resources, (both memory and speed), such that the execution time remains approximately constant [1, 6, 7, 15].

The problem of solving systems of linear equations is a scalable one, and one central to scientific computing. When the LINPACK software was first introduced, timings for solving a 100 by 100 system were gathered from a number of institutions and published. In 32-bit precision, the problem only required 40,000 bytes of storage and about 670,000 floating-point operations. A computer such as a VAX-11/780 took several seconds for the computation—a reasonable unit of time to wait for an answer. As computers have increased in size and speed, the 100 by 100 problem is increasingly inappropriate for measuring high-speed computers. The CRAY Y-MP8/832 performs that problem in less than 1/300 of a second, faster than almost any display can refresh to inform the user that the task is done. Even in 64-bit precision, the CRAY uses less than 1/30,000 of its main memory for such a problem. As a result, a new version of the benchmark was introduced for a 300 by 300 problem. When this also began to look small, a 1000 by 1000 version was added. Variations for precision and allowable optimizations have further multiplied the number of meanings of the phrase "LINPACK benchmark." The LINPACK benchmark has limited scalability even as a kernel, since its random number generator produces singular matrices for large matrix sizes. In fact, no major benchmark in use today has been designed for scalability.

Yet, most real scientific problems are inherently scalable. We use n here to indicate some measure of both problem size and difficulty. It need not be tied to instruction counts or floating point operations or bytes of storage; it should simply increase with the quality or complexity of the simulation. For example, scientific problems often involve n degrees of freedom, where n is variable over a wide range depending on the accuracy and realism desired. We seek such a problem as the basis of the benchmark. By varying n, the benchmark should be able to track changes in available performance.

We also wish to allow n to vary on a fine scale. That is, the problem should accommodate any integer n above some threshold and not, for example, restrict n to perfect squares or powers of 2. This will allow the exploration of the space of problem size versus number of processors in detail, for parallel systems with adjustable numbers of processors.

Goal: FIXED-TIME BENCHMARKING

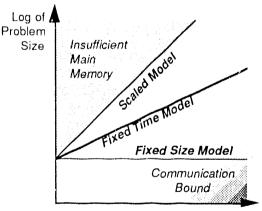
Rather than fix the size of the job to be run, we wish to fix the *time* and scale the work to be done to fit within that time. A time of one minute is used in our current effort, but any time range within the limits of human patience (about 0.1 second to 1 month) for a single computer task could be used as the constant. Shorter times do not fully exercise a system, and longer times are tedious and expensive to use as a benchmark. The benchmark should have logic to time itself and adjust automatically to find the problem size for the specified time, or allow the user to do the search manually. We wish to consider only elapsed time, not "CPU time" or other subsets of what the user perceives.

An important consequence of the fixed-time model is that "Amdahl's law" loses its relevance and its predictive powers in understanding the limits to vectorization, parallel processing, and other architectural ideas [6, 7]. We feel that this "fixed-time" approach should be used in benchmarking computers generally.

It is important to note that the fixed-time model is distinct from the "scaled speedup" model in which the problem size, as measured by the *storage of variables*, is scaled with the number of processors [7]. On ensemble computers, simply replicating the problem on every processor will usually make total execution time increase by more than just the cost of parallelism. Fixing work per processor instead of storage per processor keeps run time nearly constant. A simple example is that of matrix factoring. Consider the simple problem of solving *n* equations in *n* unknowns, with full coupling between equations (dense matrix representation). Arithmetic work varies as n^3 , with storage varying as n^2 . On a *P*-processor distributed memory system, simply replicating the storage structures on every processor will *not* generally lead to a fixed run time, since the arithmetic work to solve a matrix with Pn^2 elements is $P^{3/2}n^3$, whereas a fixed time model that assumes negligible parallel overhead on *P* processors would call for Pn^3 arithmetic work. This means that the scaled model execution time increases as $P^{1/2}$.

This situation appeared in the wave mechanics, fluid dynamics, and structural analysis problems run on the 1024-processor hypercube at Sandia [7], which similarly involved order $O(n^2)$ data storage and $O(n^3)$ arithmetic complexity. On the 1024-processor hypercube, to simulate a like amount of physical time (or convergence accuracy for the structural analysis problem) took about $1024^{1/2} = 32$ times as much ensemble computing time. It was then that we realized that the historical "Just make the problem larger!" argument for distributed memory might be simplistic to the point of being fallacious. The scaled model is still the best one to use if storage rather than time dictates the size of the problem that can be run, but the fixed-time model more realistically limits the extent to which problem scaling can be used to reduce communication cost for ensemble computers.

For these " $n^2 - n^{3}$ " problems, it is useful to think about increasing the ensemble size by powers of 64. With 64 times as much computing power, increasing *n* by a factor of 4 increases the work by a factor of $4^3 = 64$, which should keep execution time about constant if parallel overhead is low. However, the total data storage then only increases by a factor of $4^2 = 16$, not 64. Thus, each processor actually decreases in local storage requirements by a factor of 4. With a typical distributed memory approach of using subdomains on each processor, the subdomain dimensions shrink by 50% for every factor of 64 increase in the number of processors. *Fixed-time performance models must reduce the size of subdomains as the number of processors P increases, if work grows faster than storage*. For the $n^2 - n^3$ problems, the linear size *m* of an *m* by *m* subdomain will vary as $P^{-1/6}$ if we assume linear performance increases. On a log-log graph of problem size and ensemble size, the ideal fixed-time model appears as a line of slope 2/3, the ratio of the exponents for storage complexity and work complexity (see Fig. 1).



Log of Number of Processors

FIG. 1. Problem Size vs. Ensemble Size

Goal: LANGUAGE/ARCHITECTURE INDEPENDENCE

Rather than define the task with a particular program written in some language, the problem to be solved should be specified at a more abstract level. A benchmark should state *what* is to be computed for a given range of possible inputs, but not *how* to compute it. The range of possible inputs should be large enough that major deviations from running some form of the basic algorithm (such as looking up the answer in a large precomputed table) are not practical. This helps to control one experimental variable: the particular algorithms being compared. Any version of the benchmark should be permitted that arrives at correct answers, without artificial binding to language or architecture.

A benchmark should be able to exercise new architectural concepts as they arise, such as massive parallelism. We are most interested in the use of powerful computers to simulate physical systems. Since most physical systems have ample parallelism, use of a physics-based problem should provide a way for parallel computers to demonstrate their capabilities. An example of a trivially parallel problem is multiple runs with different starting assumptions. An example of an inherently sequential problem is the 3-body problem for a large number of timesteps. Both trivial parallelism and inherent sequentiality should be avoided as extreme cases of sequential/parallel ratios that are not representative of mainstream scientific computing.

Goal: PRECISION INDEPENDENCE

Rather than specify an arithmetic precision to be used, such as "64-bit IEEE floating-point arithmetic," self-consistency should be required in the result to a certain relative error. The user is then free to achieve a result within that tolerance using any calculation method or precision. The rules for precision should be determined by the desired precision in the result, not by dictating the method of calculation. Physical conservation laws are very helpful in testing self-consistency in scalable problems.

Goal: VALID FIGURE OF MERIT

Performance evaluation is inherently multidimensional. Yet, efforts to disseminate statistical information have not been very successful. The Livermore Loops present 24 speeds for 3 different vector lengths, with a variety of ways to sum and average the results, and yet one sees statements like, "Our computer runs the Livermore Loops at 10.8 MFLOPS." The SPEC benchmark also contains 10 components of widely varying nature (from matrix kernel operations to a complete circuit simulation), yet the "SPEC mark" is a scalar quantity derived from these components. Recognizing this, we seek to produce a single figure of merit number that is meaningful, to prevent misuse of multidimensional information.

Instead of using questionable performance measures such as MIPS (Millions of Instructions Per Second), or MFLOPS (Millions of Floating-Point Operations Per Second), the basis of comparison should be simply n, the problem size. Although other work measures can be provided by the benchmark as a guide to optimization, they are not the coin of the realm. A computer should be considered more powerful than another on this benchmark if and only if it runs a "larger" (bigger value of n) problem in the time allotted. It is not necessary that work be a simple function of n (and it seldom is), but the work should be a strictly increasing function of n.

Goal: COMPLETE TASK MEASUREMENT

With a fixed-time paradigm, it becomes practical to include costs such as disk input/output and the setting up of equations to be solved. Since computers tend to improve so as to balance speeds with fixed time rather than fixed-size jobs in mind, these previously excluded components of computer use can be fairly included in the measurement. We strongly feel it is incorrect to test only the compute-intensive part of a task. Even recent efforts such as the PERFECT and SPEC test suites excise the input and output functions in some or all of their component routines [11, 13].

Goal: MINIMIZATION OF HUMAN EFFORT BIAS

Since converting programs to different architectures imposes a burden that is reflected (at least temporarily) in reduced performance, the benchmark should be disseminated in as many representative forms as possible: traditional, vectorized, shared memory parallel, distributed memory parallel, etc. It should also be maintained in many languages such as C, Fortran 77, Pascal, and Fortran 90, to reduce language conversion effort. In the sense that computer benchmarks compare *programmers* as well as computers, a centralized and collective body of conversion tools makes the comparison fair and deemphasizes programming skill. For the same reason, great effort should be put into finding the "best serial algorithm," that is, the solution method with the smallest apparent complexity. Otherwise a problem thought to be some complexity like $O(n^3)$ might later prove to be $O(n^2 \lg n)$, which only some programmers would discover and exploit.

5

Goal: ACCOUNTABILITY

For some reason, virtually all published benchmark data deletes the source of the data. In contrast to scientific reporting, computer benchmark figures are seldom accompanied by the name of the person who ran the benchmark and the date the figures were submitted. To preserve the integrity and accountability of the comparison, the benchmark should include this data, along with the institutional affiliation of the person submitting the measurement.

3. DETAILED DESCRIPTION

The following sections amplify on the preceding ideas. The Scalable, Language-independent, Ames Laboratory One-minute Measurement (SLALOM^M) was created to meet the objectives described above.

3.1. Scalable Benchmarking

In September 1989, we began a search for a complete, practical scientific problem that demands the solution of a set of *n* fully-coupled equations similar to the traditional LINPACK test. Conventional methods for such problems require $O(n^3)$ operations for solution, and $O(n^2)$ operations for setup. Storing the answer, a list of *n* numbers, takes O(n) operations. Reading a description of the geometry and other physical parameters of the problem takes O(1) operations. The memory required for the problem varies as n^2 . These scaling characteristics capture the salient features of a wide spectrum of scientific computing tasks. With careful design of the problem discretization, *n* can be chosen as any positive number, to permit fine adjustment of the work and storage needed.

We have been unable to find a genuine scientific problem for which the best known algorithm requires the direct solution of a nonsymmetric, dense matrix with partial pivoting. However, a diagonally-dominant dense matrix problem was found in the pioneering paper by Greenberg, Goral, *et al.* on "radiosity," [5] which is the equilibrium radiation given off by a coupled set of diffuse surfaces that emit and absorb radiation. The problem is easily described and understood: A room is painted with a separate color for each wall, plus floor and ceiling, and one or more of the six surfaces also emits light. Emissivity and reflectivity are described as red-green-blue components for each wall of the room. The problem is to find the color variation over each wall. Goral's paper uses an example test case as shown in Fig. 2, with unit face sizes.

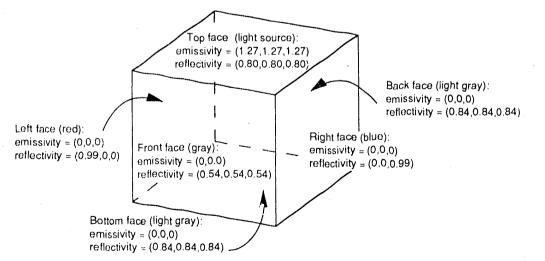


FIG. 2. Radiosity in a Box

There is a white, light-emitting ceiling, shades of gray on the floor, front, and back walls, and saturated red and saturated blue side walls. (We will usually use the term "face" instead of "wall," and "box" instead of "room," in this paper.) With diffuse surfaces, there is a "bleeding" of color to nearby surfaces.

Goral's paper offers limited scaling, breaking each face into 3 by 3, 5 by 5, and 7 by 7 "patches," with $6m^2$ equations to solve for an *m* by *m* patch decomposition. The coupling between patches is the "fraction of the sky" each patch "sees" occupied by another patch, for which [5] uses an approximate quadrature.

We coded the radiosity problem in a scalable fashion, to allow any number of patches *n*, from six on up. The challenge is to write an automatic decomposition algorithm that is both concise and amenable to parallel processing, so the process will be treated in some detail here. The initial approach was to assume the box is a unit cube, find the largest *m* such that $6m^2$ is less than *n*, then halve patches until *n* was reached. For example, for n = 27 patches, one would start with $\lfloor \sqrt{(27/6)} \rfloor = 2$ for *m*, and then split three patches in two (see Fig. 3).

ſ	2	34	8	9	12	13	16	17	20	21	24	25	
5	6	7	10	11	14	15	18	19	22	23	26	27	

FIG. 3. Initial Attempt at Scalable Decomposition

This simple approach worked, but with drawbacks. It created patches of very different areas, implying uneven accuracy in the numerical solution. A practical program would more likely seek to reduce the maximum error by keeping patches as similar in area as possible. Furthermore, the regularity of such a decomposition encourages a clever programmer to shortcut coupling calculations by noticing that many pairs of patches have the same spatial relationship (see Fig. 4).

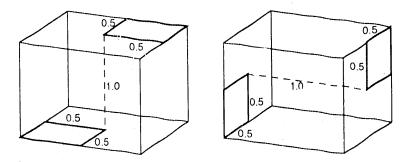


FIG. 4. Exploitable Symmetries

Also, the solution for a perfect cube is too special to resemble a practical radiosity calculation. Hence, we allow variable box dimensions, restricted to the range 1-100 length units, and decompose the surface of the box into patches that are as nearly square and as nearly equal in area as possible. Exploitation of repeated geometric relationships becomes much more difficult, accuracy for a given number of patches is improved, and the problem more closely resembles a real problem for which a scientific computer might be used.

Let A_i be the total area of face i, and A be the total area. Then we want

Number of patches on face $i \approx n \times A_i / A_i$.

Actually, we mark "start-end" patch numbers for each face. Face 1 starts with patch 1; Face 6 ends with patch *n*. In between, face *i* starts with the patch that face *i*-1 ended with, plus one. Face *i* ends with patch $\lfloor \sum A_j/A + 0.5 \rfloor$, where the summation is j = 1 to *i*. The " $\lfloor \ldots + 0.5 \rfloor$ " technique explicitly rounds to the nearest integer, breaking ties by rounding up. This explicitness was discovered to be necessary when we tested language independence between Fortran and Pascal, since implicit rounding functions in Pascal use round-to-nearest-even rules, whereas Fortran uses round-toward-zero rules.

Within a face, we desire patches that are as nearly square as possible, to reduce discretization error. This is accomplished by dividing each face first into columns, with the number of columns given by $\lfloor \sqrt{(\text{patches/eccentricity}) + 0.5} \rfloor$. The eccentricity is the ratio of the face dimensions. For example, to put 7 patches on a 2 by 3 face, use $\lfloor \sqrt{(7/3)} + 0.5 \rfloor = 3$ columns. We can slightly increase robustness by using *one* column when this formula gives *zero* columns for a face. However, there must still be an error trap for the case of no *patches* on a face. For example, a 1 by 1 by 50 box with only six patches will decompose to having no patches on the 1 by 1 faces (each being only 1/200 of the surface area) and the benchmark must signal a need for more patches for such a problem.

Let n_{patch} be the number of patches on a face, and *i* be the local number of a patch on that face, $1 \le i \le n_{patch}$. Let n_{col} be the number of columns on the face, as determined by the preceding discussion. Then patch *i* resides in the column given by

$$i_{col} = \lfloor (i-1) \times n_{col} / n_{patch} \rfloor + 1 \tag{1}$$

for arrays with index origin 1. Note that $1 \le i_{col} \le n_{cok}$ This assignment of patches to columns distributes "remainder" patches (that is, those in excess of an exact integer division of n_{patch} by n_{col}) evenly across the face rather than clumping them at one extreme. A geometrical interpretation of the subdivision for $n_{patch} = 7$ and $n_{col} = 3$ is shown in Fig. 5.

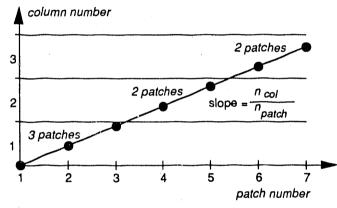


FIG. 5. Column number versus local patch number

We can invert (1) to find the range of *i* for a given value of i_{col} :

$$i_{col} - 1 \le (i - 1) \times n_{col} / n_{patch} < i_{col}$$
$$\Leftrightarrow (i_{col} - 1) \times n_{patch} / n_{col} + 1 \le i < n_{col} \times n_{patch} / n_{col} + 1$$

Since the left and right bounds are noninteger in general, use floor and ceiling functions to sharpen the range:

$$\left\lceil (i_{col} - 1) \times n_{patch} / n_{col} \right\rceil + 1 \le i \le \lfloor i_{col} \times n_{patch} / n_{col} + 1 \rfloor$$
(2)

where the ceiling function $\lceil n / m \rceil$ is calculable from $\lfloor (n + m - 1) / m \rfloor$, a more language-independent construct. It now follows that the number of *rows* in a given column is

$$n_{row} = \left[i_{col} \times n_{patch} / n_{col} \right] - \left[(i_{col} - 1) \times n_{patch} / n_{col} \right]$$
(3)

which gives $n_{row} = \{3, 2, 2\}$ for $i_{col} = \{1, 2, 3\}$ for the example shown in Fig. 5.

Fig. 6 shows a solution for the benchmark problem for a 512-patch decomposition.

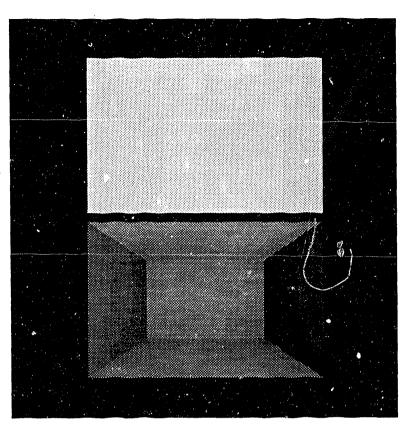


FIG. 6. Example of Radiosity Solution

This completes the solution to the scalability problem with respect to domain decomposition. For any problem of size six or greater, the preceding method decomposes the benchmark task in a reasonable, portable, concise, numerically sound manner. For a parallel ensemble, the geometry of any subset of the patches can be computed directly from the number of the patch, removing a potential serial bottleneck.

3.2. Fixed-Time Benchmarking

It is possible to make any scalable benchmark into a fixed-time benchmark simply by putting an upper time bound in the ground rules of the benchmark. If a user-written program can time its own execution, the program can *scale itself* to run in a specified time. Just as a recursive program operates on its own output, the fixed-time driver creates a benchmark that operates on its own performance.

The number to adjust is an integer, n, that describes the "size" of the problem in some sense. Here, n is the number of patches in a radiosity problem, but the technique is general.

The user is asked by the program to supply a desired time interval, which we call *goal*. (We have found, by experiment, that 60 seconds is a good compromise between realistically long run times and easy-to-benchmark short times, but the *goal* time is arbitrary.) The user is then asked to supply a value of *n* such that the program will take less than *goal* time to execute.

The program tests that n is within limits imposed by the problem and the computer. For example, the radiosity problem requires $n \ge 6$. (If the box is highly eccentric, the minimum n could be larger.) If n passes as a valid lower bound, the timer is started and the benchmark is run. If the benchmark fails to run in less time than *goal*, the driver repeats its request until a satisfactory lower-bound n is supplied. If it succeeds, the n is saved and the driver proceeds to the next step.

8

The next stage is to find an *n* such that the run time is greater than or equal to *goal*. The reason is to disallow equality with *goal*, which rewards low-resolution timers. For example, a computer capable of self-timing only to 1 second resolution might run 60.999 seconds, report it as 60 seconds, and thus be able to run a larger *n* than a computer with a more precise clock.

If goal is large, n might exceed the value allowed by the computer memory allocated by the program being benchmarked. The user is responsible for altering the benchmark to allow sufficiently large n, even if it means explicit management of mass storage. (Running out of memory to achieve a one-minute SLALOM run might be interpreted as a symptom of unbalanced or special-purpose computer design.) If the n supplied as an upper bound fails to equal or exceed the *goal* time, the driver repeats its request until a satisfactory n is supplied.

Note that a given computer might not be powerful enough to run even the minimum n permitted by the benchmark in *goal* time. We have chosen the problem and *goal* such that virtually every programmable machine currently marketed is sufficiently powerful to qualify, although computers from a few years ago might not.

With an upper bound and a lower bound, the problem of finding the n that is as large as possible without requiring time greater than or equal to *goal* is a classic root-finding problem. The time is not necessarily an increasing function of n, nor is it particularly "smooth" for most computers. Pipeline lengths, cache sizes, and memory organization can complicate performance enough to destroy monotonicity. Methods such as Newton-Raphson iteration were tried and found nonconvergent in general, for the preceding reason.

There might also be random timing variation for any single value of n. If the variation is greater than the difference in timing for values of n differing by unity, then the n determined by the driver will be a random variable. Intuitively, the distribution of n is zero above some integer, since the hardware has inherent limits (Fig. 7). Hence, we look at the record largest n achievable over any desired number of tests to again reduce the measurement to a single integer value.

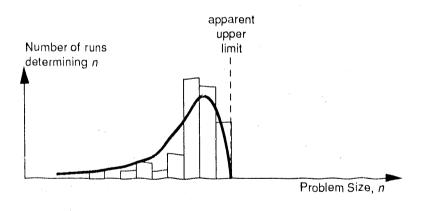


FIG.7 Distribution of *n*

A convergent method that is used in the current version of the benchmark driver is *recursive* bisection: While $n_{upper} - n_{lower} > 1$, find $n_{mean} = (n_{upper} + n_{lower})/2$. Time the benchmark for n_{mean} ; if less than *goal*, replace n_{lower} by n_{mean} and repeat. Otherwise, replace n_{upper} by n_{mean} and repeat.

Once $n_{upper} - n_{lower} = 1$, the desired *n* is n_{lower} . A problem with this method is that random fluctuations in the timing might assign a particular *n* as below *goal* on one run, but above it on the next. Currently, our workaround is to refer to the instance where the execution time was below *goal* and use that run as the result. We ignore the "final" report of an *n* value if it equals or exceeds *goal*.

The fixed-time driver has been developed for several computers and written in several languages. It works satisfactorily in most cases. On the MasPar and Cray versions, the machines have a preference for multiples of 64 in the problem size, and manual selection is used to find the largest problem that runs in less than one minute.

3.3. Language/Architecture Independence

Two approaches are used to remove ties to a particular language or architecture: a high-level problem description, and a multiplicity of working examples covering a wide spectrum of environments.

A high-level problem description is practical if guidance is supplied as to what appears to be a "good" solution method, and if the input-output is specified so as to rule out unrealistic use of precomputed answers. Supplying guidance is like an athletic competition; although certain techniques are known to be effective, competitors may choose what works best for them individually. Advances in technique that appear general are made publicly known as quickly as possible to eliminate an advantage based on disparate knowledge.

If only a single input and output are specified, a benchmark with such liberal rules quickly degenerates into the trivial recall of precomputed answers. But if input is not specified, run times will vary with input (in general), introducing an uncontrolled variable. The solution is this: the program must work for a specified *range* of inputs, and must time an input (supplied as standard) using the same method used for arbitrary input. Stated another way, the program cannot contain any information specific to the standard case.

For the radiosity problem, the standard case is much like the example in Goral's paper [5] shown in Fig. 2. The main changes are to make the faces rectangular (13.5 by 9 by 8) rather than square, and to derive the coupling with exact analytic expressions instead of approximate quadrature. The matrix formulation and solution are similar, except that we divide the matrix row entries by the area of the patch to which they pertain, which renders the matrix symmetric. The discovery that the radiosity problem could be made symmetric, cutting solution time almost by half for large problems, was a surprise to us. It reduces the resemblance of SLALOM to the LINPACK benchmark, but one could argue that symmetric systems of equations are the rule rather than the exception in physical simulations.

As of this writing, we have converted the high-level description of the radiosity problem, as supplied by Goral's paper, into the following forms:

- Fortran 77 for Sun 4, VAX, IBM PC, etc.
- Vectorized Fortran for single processor Iris and NCUBE computers (the NCUBE is scalar, but benefits from vector library calls).
- Pascal for IBM PC compatibles.
- BASIC for Macintosh (both interpreted and compiled).
- C for SUN4 and other UNIX-based workstations.
- C (extended with plural variables) for MasPar, 1024 to 16384 processors.
- Fortran with parallel loop compiler directives for CRAY-2 and Iris (shared memory, 1-8 processors).
- Fortran with message-passing constructs for NCUBE (up to 4096 processors)
- Fortran with "PARDO" constructs for Myrias (supplied by Jim Roche, Myrias Computer Corp.).

We are in the process of putting all of the versions under the SCCS revision control system.

3.4. Precision Independence

We feel the goal should be to compute an answer within a specified tolerance of the correct answer, and not specify the word size or anything else about how to get to that level of precision. The benchmark has two self-consistency checks. One is inside the timed part of the benchmark, since the check is also used to improve the accuracy of the answer if within tolerance limits. The other is a pass/fail verification after the computation is done, not ti ned. It is very unlikely that an incorrect program will pass both tests, and experience has confirmed this. (We also use comparison of output files and examination of graphic displays of the output as a convenient way to check program correctness for small problems).

The first self-consistency check involves matrix setup. Let f_{ij} = the fraction of the hemisphere taken up by patch *j*, as seen from patch *i*. For example, $f_{ij} \approx 1$ for close, parallel planes, about 0.2 for unit squares a unit apart or perpendicular and touching, and near 0 for patches that are small relative to their separation (Fig. 8).

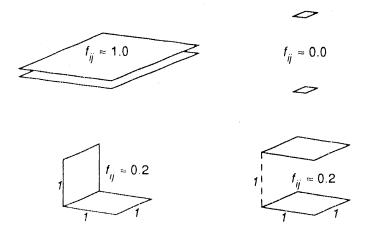


FIG. 8. "Form Factor" Examples

These f_{ij} are variously called "form factors" or "coupling factors" or "shape factors" in the radiation transfer literature. Analytic means exist to compute them for special geometric shapes, based on evaluation of 4-dimensional integrals.

Initially, we attempted to use approx_nations to the form factors that would be easy to comptom. like those in the Goral paper [5]. However, we found the accuracy to be poor for small numbers of parches, unrealistic for a scientific program. We evaluated the integrals in closed form for parallel and perpendicular patches with edges parallel to the xyz coordinate axes, eventually creating a one-page program for the f_{ij} computation that is considerably more compact than any appearing in the literature on form factors. Also, a cyclic ordering of faces eliminated the need for extensive "case" statements. Fig. 9 illustrat is this:

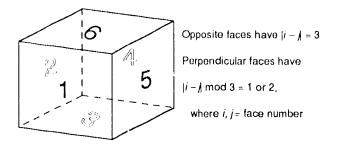


FIG. 9. Cyclic Face Numbering Advantages

It is important for a benchmark program to be concise and manageable, to minimize conversion effort and maintenance costs, yet represent the demands of a real computer application. These terse setup portions of the benchmark only take about 200 lines of a high-level computer language.

By using closed form expressions, the f_{ij} factors inherit the property that $\sum f_{ij} \approx 1$, for all *i*, when correctly evaluated. Since each f_{ij} requires hundreds of operations to evaluate (including square roots, logarithms, and arctangents), this summation provides a sensitive independent test of the matrix setup. We choose a tolerance of 0.5×10^{-8} for the $\sum f_{ij}$ to deviate from unity, that is, an accuracy of 7 decimals. This requires somewhat more than "single-precision" arithmetic on most computers (7.4 decimals ideally, but fewer because of cumulative errors) but is comfortably within the "double-precision" range. This provides a level playing field for the various arithmetic formats. It is usually advantageous to use the smallest number of bits or digits that satisfies the tolerance. This number will vary with problem size, but the user is free to meet the tolerance by adjusting precision as needed, throughout the task.

For $\sum f_{ij}$ values within the tolerance limits but not numerically equal to unity, the f_{ij} values are normalized by the sum to force the sum to unity. This helps control minor cumulative rounding errors. Instead of normalizing the entire row of the matrix, we simply scale the right-hand side scalar and diagonal elements, trading *n* multiplications for two.

The area of patch *i* can be denoted a_i . Because the a_i are not all the same, $f_{ij} \neq f_{ji}$ in general (see Fig. 10.)



FIG. 10. Asymmetric Coupling

This means the radiosity matrix is nonsymmetric. However, in the process of trying to remove minor undiscovered shortcuts from the algorithm, we discovered a major one. If the matrix rows are divided by a_i , the matrix becomes symmetric, as mentioned in Section 3.3. Symmetry reduces solution cost by roughly a factor of two. Again, the scaling by a_i is applied to the diagonal and right-hand side, saving n^2 multiplications by $1/a_i$ of the other matrix elements. Cholesky factorization can be used for the matrix solution, for which there are well-tuned routines in many software libraries.

The second self-consistency test involves "residual" checks. For the linear system Ax = b, where A is an n by n matrix and x and b are vectors of n elements, the residual is defined as ||Ax - b||, where we choose a computationally easy norm, the maximum of the absolute values of the elements. To specify a tolerance, the residual is normalized by the norms of A and x, a quantity sometimes called the *relative residual*. We require that $||Ax - b|| / ||A|| ||x|| < 0.5 \times 10^{-8}$ for each of the x values computed by the benchmark (one x for each component of the radiation: red, green, and blue). Thus, the residual check is really three tests, all of which must pass. The residual check is performed after timing, since application software would generally eliminate such tests once program errors appeared to have been removed.

The user is encouraged to use whatever means work best to reduce the residual to the required tolerance. The problem is well-posed. Partial pivoting would add $O(n^2)$ floating-point comparison operations and introduce a serial bottleneck into the factoring algorithm. When partial pivoting was tried, the pivot was always on the diagonal, and so pivoting was eliminated from the benchmark. Diagonal dominance can be easily proved from the fact that reflectivity is less than unity and the sum of off-diagonal elements in a row is unity.

The second self-consistency check greatly improves the "rules" under which the benchmark is run. Some parallel computers might favor iterative methods, or solution methods of very different internal composition from the one supplied. The alternative method merely has to satisfy the 0.5×10^{-8} tolerance for the full range of possible inputs, and it is then deemed a fair method to use.

For this reason, the range of possible inputs has been carefully bounded. The faces can range in dimension from 1 to 100 on an edge, and from 0.001 to 0.999 in reflectivity. Some cases in these ranges will be difficult to solve by iterative methods. For example, consider the box shown in Fig. 11.

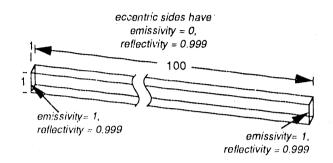


FIG. 11. Difficult Iterative Case

Iterative methods must numerically accumulate enough terms of a slowly-converging infinite series to account for the multiple low-loss reflections of radiation from the left face traveling down the box to the right. Just as

$1/(1+x)\approx 1-x$

favors the right-hand side for ease of computation when x is near 0,

$$1/(1+x) \approx 1 - x + x^2 - x^3 + \dots + x^{11}$$

will favor the "direct method" on the left if x is slightly larger than -1. In this manner, we constrain competing machines to use methods that are similar (that is, direct solvers), but not by artificial rules. The rules are instead driven by requirements for the output delivered to the user.

3.5. Figure of Merit

The notion of using operation counts or other "work" measures for computer performance evaluation has several drawbacks. It tends to reward inefficient methods that exercise the hardware, even if they get the result more slowly. The notion of what to consider an "operation" has not stood the test of time. In the 1950's and 1960's, multiplications dominated overall run time for compute-intensive problems, so complexity analysis considered only multiply and divide counts. By the 1970's additions and multiplications had comparable cost and were often weighted equally. Now, memory references often take longer than the arithmetic, but are much harder to assess analytically for an abstract computer.

To date, the generally accepted practice has been to use execution time as the figure or merit, fixing the problem to be timed. This has disadvantages already described, but at least execution time is a physically measurable quantity.

Here, we make *problem size* the figure of merit (the larger the better), another measurable quantity not subject to dispute. The use of problem size can lead to slightly unconventional ranking of machines, as shown in Table I:

	TABLE I Differences in Figure of Merit							
Computer A	Computer B							
1392 patches	1400 patches							
2.70 billion operations	2.74 billion operations							
58 seconds	59 seconds							
46.6 MFLOPS	46.4 MFLOPS							

By conventional measures, Computer A is ranked higher since it performed more MFLOPS. By our metric, Computer B is ranked higher because it ran a larger problem (more patches). Possibly Computer 1 has difficulty applying its speed to a slightly larger run because it runs out of fast memory, exceeds a hardware vector length, etc. The effect will generally be only a slight difference from the MFLOPS-based ranking, except when the MFLOPS for a computer is a jagged function of the problem size.

Since supercomputer purchases are generally motivated by a desire to run larger problems (not achieve higher MFLOPS rates), the problem size makes a better figure of merit. This is the "grand challenge" esthetic. It contrasts, say, with the esthetic of maximizing conventional data processing throughput. The achievement of a 40,000-patch SLALOM run might be more significant than the achievement of a "teraflop" of nominal speed, since there would be at least a little assurance that the speed might be applicable to real problems.

3.6. Complete Task Measurement

The idea of a fixed-time benchmark solves the decades-old difficulty of including such parts of the benchmark execution as program loading, input, output, and other tasks with rather invariant time cost. With a fixed-sized problem, these components eventually dominate total execution time as vector or parallel methods are applied to the compute-intensive portions of the job (Amdahl's 1967 argument against parallel architectures). Hence, previous benchmarks have solved the problem by including only the kernel in the timing, with an enormous loss of realism.

With a fixed time of about one minute, the non-kernel part of the work should take just a few seconds, and can be included in the timing without distortion effects. For the radiosity problem described here, time should grow as

- O(1) for program loading,
- O(1) for reading problem geometry,
- $O(n^2)$ for setting up the matrix,
- $O(n^3)$ for solving the matrix, and
- O(n) for storing the solution.

Traditional benchmarks only time the $O(n^3)$ part, or possibly both $O(n^2)$ and $O(n^3)$ parts. Here we time everything essential to the run other than the original writing and compiling of the program (which is presumably amortized over many runs and hence legitimate to neglect). Interestingly, the lower-exponent parts of the problem are the hardest to make run in parallel, so massively-parallel architectures will reveal the same input/output challenges for SLALOM that they face in general applications.

3.7. Minimization of Human Effort Bias

To reduce the effect of variable human analytical skill in adapting a given program to a particular computer, we apply the same technique already mentioned in Section 3.3: a variety of best-effort versions are maintained in the library of possible starting points, for as many different architectures and languages as possible. New versions, motivated by the desire of a vendor to show high performance, are added to the library rather than kept proprietary. In this way, contributors must provide not just performance data but also their *method* for achieving that performance in software, so that others may build on their accomplishment.

4. BENCHMARK RESULTS

Table II gives the results of using the SLALOM benchmark on a wide range of machines. All runs took between 58 and 60 seconds, so times are not given. The computers are listed in order of decreasing problem size that they were able to solve.

				1 A A A A A A A A A A A A A A A A A A A		
Machine, environment	Processors	Patches	Operations	MFLOPS	Measurer	Date Measured
Cray Y/MP-8, 167 MHz Fortran+tuned LAPACK solver (Stras	8 ssen)	5120	126. G	(2130.)	J. Brooks (v) Cray Research	9/21/90
Cray Y/MP-4, 167 MHz Fortran+tuned LAPACK solver (Stras	4 ssen)	4096	65.2 G	(1190.)	J. Brooks (v) Cray Research	9/21/90
Cray Y/MP-2, 167 MHz Fortran+tuned LAPACK solver (Strat	2 ssen)	3200	31.6 G	(557.)	J. Brooks (v) Cray Research	9/21/90
Cray Y/MP-1, 167 MHz Fortran+tuned LAPACK solver (Stra	1 ssen)	2560	16.4 G	(283.)	J. Brooks (v) Cray Research	9/21/90
Cray-2S/8-128, 244 MHz Fortran+directives, FPP 3.00Z25	8	2443	14.4 G	240.	S. Elbert Ames Lab	9/8/90
nCUBE 2, 20 MHz Fortran+tuned subroutines	256	2071	8.90 G	153.	J. Gustafson Ames Lab	10/1/90
MasPar MP-1, 12.5 MHz C with plural variables (mpl)	16384	2044	8.57 G	144.	B. Wheelock (v) MasPar	10/5/90
MasPar MP-1, 12.5 MHz C with plural variables (mpl)	8192	1663	4.73 G	80.9	K. Jacobsen (v) MasPar	10/5/90
nCUBE 2, 20 MHz Fortran+tuned subroutines	64	1438	2.83 G	47.2	J. Gustalson Ames Lab	9/17/90
MasPar MP-1, 12.5 MHz C with plural variables (mpl)	4096	1343	2.57 G	43.7	K. Jacobsen (v) MasPar	10/5/90
MasPar MP-1, 12.5 MHz C with plural variables (mpl)	2048	1055	1.30 G	22.1	J. Brown (v) MasPar	10/5/90
Silicon Graphics 4D/380S, 33 MHz Fortran (-O2 -mp -lparalin)	8	1010	1.15 G	19.2	S. Elbert Ames Lab	6/15/90
MasPar MP-1, 12.5 MHz C with plural variables (mpl)	1024	927	903. M	15.9	J. Brown (v) MasPar	10/5/90
Silicon Graphics 4D/380S, 33 MHz Fortran (-O2 -mp -lparalin)	4	853	716. M	11.96	S. Elbert Ames Lab	6/15/90
nCUBE 2, 20 MHz Fortran+tuned BLAS	16	840	687. M	11.5	J. Gustafson Ames Lab	10/8/90
Silicon Graphics 4D/380S, 33 MHz Fortran (-O2 -mp -lparalin)	2	676	378. M	6.39	S. Elbert Ames Lab	6/15/90
IBM RS/6000 POWERstation 320, 2 Fortran (xlf -O -Q)	20 MHz 1	642	328. M	5.6	S. Elbert Ames Lab	5/14/90

Ċ

TABLE II SLALOM PERFORMANCE FOR VARIOUS MACHINES

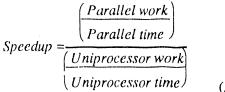
Silicon Graphics 4D/380S, 33 MHz Fortran (-O2 -mp -Iparalin)	1	530	195. M	3.27	S. Elbert Ames Lab	6/15/90	
nCUBE 2, 20 MHz Fortran+tuned BLAS	4	500	167. M	2.83	J. Gustalson Ames Lab	10/9/90	
SUN 4/370, 25 MHz, C (ucc -O4 -dalign etc.)	1	419	105. M	1.75	M. Carter Ames Lab	10/8/90	
iPSC/860, 40 MHz Fortran (-OLM -i860)	. 1	419	105. M	1.75	J. Gustafson Ames Lab	5/17/90	
Myrias SPS-2 (mc68020, 16.7 MHz) Fortran (mpfc -Ofr)	64	399	92.2 M	1.56	J. Roche (v) Myrias	6/21/90	
nCUBE 2, 20 MHz Fortran + assembler subroutines (-O2)	1	354	67.5 M	1.13	J. Gustafson Ames Lab	8/13/90	
Silicon Graphics 4D/20, 12.5 MHz, Fortran (177 -O2)	1	290	40.5 M	0.679	S. Elbert Ames Lab	5/15/90	
DECStation 2100, 12.5 MHz, Fortran (f77 -O2)	1	285	38.8 M	0.649	J. Gustalson Ames Lab	5/4/90	
Cogent XTM (T800 Transputer) Fortran 77 (-O -u)	1	149	7.89 M	0.133	C. Vollum (v) Cogent	6/11/90	
IBM PC-AT, 8 MHz CFG Fortran 1.7	1	67	1.26 M	0.0216	J. Gustalson Ames Lab	4/30/90	
 Mac IIcx, 68030, Interpreted QuickBASIC	1	24	0.142 M	0.00239	J. Gustafson Ames Lab	5/1/90	

NOTE: a "(v)" after a name means the benchmark was run by the vendor. Vendors often have access to special tools, early compiler releases, and proprietary libraries, so remember the source. We quote MFLOPS for continuity with earlier benchmarks, but the number of *patches* determines rank. MFLOPS assume $O(n^3)$ cost for matrix factoring, and are likely to be inaccurate (too large) for problems that use block methods with $O(n^{2.8})$ Strassen multiplication or better.

5. SUPERLINEAR SPEEDUP EFFECTS

5.1. Definitions

Almost every paper on parallel speedup makes the following definition: "Speedup is the ratio of the uniprocessor execution time to the execution time on the parallel processor." *Speedup* is the ratio of speeds, not times. *Speed* is work divided by time. Work can be defined as essential floating point operations, instructions, memory references, or whatever seems a reasonable currency on a given system. The choice of definition for *work* does not affect the arguments presented here.



16

(5)

The fixed-size model assumes work is constant, resulting in simplification to the ratio of times. Since problems generally scale to fit the time that a user will tolerate, we avoid this simplification. If anything is constant in practical computer use, it is the *time*. Hence, one might simplify speedup as the ratio of parallel work to uniprocessor work done in a given amount of time. Definition (5) is the one used here.

Efficiency is traditionally defined as speedup divided by the number of processors. The definition assumes the impossibility of superlinear speedup, guaranteeing that efficiency cannot exceed unity. Since the arguments presented below show that the quantity often exceeds unity, the term "Efficiency" is a misnomer in this context.

5.2. Superlinear Speedup from Changing Routine Profile

Superlinear speedup results when problem scaling causes more time to be spent in faster routines. Consider the matrix setup and matrix factoring parts of SLALOM. The setup will take order n^2 work and the factoring will take order n^3 work. For small problems, setup might dominate the work, depending on the cost per matrix entry. The factoring approaches 100% of the work as n increases. Both steps can readily be done in parallel. In the fixed-time model, the fraction of the time spent on factoring increases with the number of processors. If the factoring proceeds at a higher speed than the setup (often the case) then each processor will run faster (more work per second) as the result of using more processors.

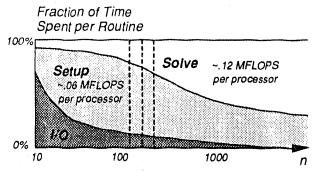
This reasoning is the theory of superlinear speedup by shifting algorithm profile. To test it experimentally, we used a version of SLALOM for the first-generation NCUBE computer. The speed in MFLOPS, as a function of P, was measured as follows:

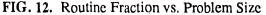
TABLE III Speedup on SLALOM

P	Problem Size, n	MFLOPS	Speedup
1	112	0.067	1.00
2	150	0.138	2.06
4	200	0.279	4.16

Even after extensive use of assembly language tuning, the problem setup ran at only 0.06 MFLOPS per processor, because of calls to intrinsic functions and irregular sequences of operations. The matrix solution, however, ran at 0.12 MFLOPS for large n. For the single-processor run, problem setup took 60% of the time, so the speed was close to 0.06 MFLOPS. On four processors, the larger n possible in a oneminute run causes factorization to take more of the time, so the speed *per processor* increased to about 0.07 MFLOPS. The effect would have been more dramatic except for the lack of parallelism in the input, output, and backsolving tasks. With further work, these will also run in parallel and the superlinearity should approach about sixfold speedup on four processors. Fig. 12 illustrates the effect described, with vertical dashed lines representing the cases in Table III:

C





It is more accurate to note that the MFLOPS rates within each shaded region are not constant with n. Fig. 13 shows this third dimension, using polynomial fits for experimental measurements on a SUN 4/370:

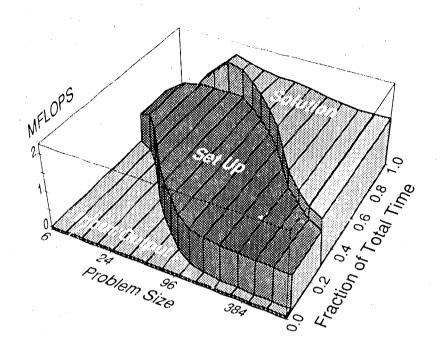
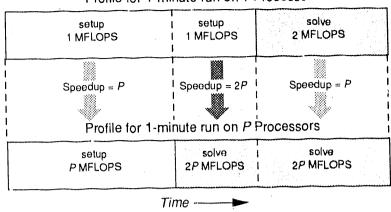


FIG. 13. Profile vs. *n* vs. MFLOPS Rate

5.3. A Fixed-Time Paradox

A curious fact emerges when one examines the speedups of *individual routines*. For example, in going from one to four processors, the setup speedup was 3.9, the solve speedup was 3.7, and the speedup for all remaining routines was 0.7. (That is, the remaining routines slowed down because of parallel overhead and unparallelized tasks.) Yet, the overall speedup is greater than 4.

This counterintuitive result shows that subtask component speedups do not average, even with appropriate weights. A resolution of this paradox was suggested by D. Krumme of Tufts University, and is illustrated in Fig. 14.



Profile for 1-minute run on 1 Processor



The superlinear effect is caused by part of the time being *replaced* by a higher-speed task. For the part of the benchmark shown in the middle section in Fig. 14, the speedur increases by the product of the relative MFLOPS rates of the tasks and the number of processors, *P*. With timing broken down in this manner, one can again think about overall speedup as some average of the component speedups. It is fallacious, however, to average components based on subtasks for a fixed-time performance model.

6. CONCLUSIONS

We view SLALOM as a significant step toward providing a level playing field for advanced architectures. We are committed to maintaining the scientific integrity of this benchmark, and look forward to measuring and publishing even more wide-ranging SLALOM numbers in the future.

SLALOM illustrates a new source of non-spurious superlinear speedup. Specifically, speed per processor is *not* constant as problems scale; it changes with fraction of time spent in routines of different algorithmic complexity. Speedup is the ratio of speeds, not times. Together, these observations give reason to expect that an ensemble computer will often be "more than the sum of its parts" in performance on problems of practical interest.

We hope that the benchmark will last several decades without a fundamental change. It may be the first benchmark with such longevity, and will permit the tracking of technology trends over a wide baseline.

ACKNOWLEDGEMENTS

We thank everyone who has participated in this effort. In particular, analysts at Myrias, MasPar, Cogent, and Cray have contributed suggestions, ideas, and versions of the SLALOM program. Much of the work was performed at the Scalable Computing Facility at Ames Laboratory/Center for Physical and Computational Mathematics.

REFERENCES

- 1. R. E. Benner, G. R. Montry, and J. L. Gustafson, "A Structural Analysis Algorithm for Massively Parallel Computers," *Parallel Supercomputing: Methods, Algorithms, and Applications*, edited by G. F. Carey, Wiley Series in Parallel Computing, 1989.
- 2 Curnow and Wichmann, "A Synthetic Benchmark," Computer Journal, February, 1976.
- 3. J. J. Dongarra, "Performance of Various Computers Using Standard Linear Equations Software in a Fortran Environment," Argonne National Laboratory, *Technical Memorandum No.* 23, Feb. 2, 1988.
- 4. V. Faber, O. Lubeck, and A. White, "Superlinear Speedup of an Efficient Sequential Algorithm is Not Possible," *Parallel Computing*, 3 (1986), pp. 259-260.
- 5. C. M. Goral, K. E. Torrance, D. P. Greenberg, and B. Battaile, "Modeling the Interaction of Light Between Diffuse Surfaces," *Computer Graphics*, Volume 18, Number 3, July 1984.
- 6. J. L. Gustafson, "Reevaluating Amdahl's Law," Communications of the ACM, Volume 31, Number 5, May 1988.
- 7. J. L. Gustafson, G. R. Montry, and R. E. Benner, "Development of Parallel Methods for a 1024-Processor Hypercube," SIAM Journal on Scientific and Statistical Computing, Vol. 9, No. 4, July, 1988.
- 8. D. P. Helmbold. and C. E. McDowell, "Modeling Speedup(n) greater than n," 1989 International Conference on Parallel Processing Proceedings, (1989), Volume III, pp. 219-225.
- 9. F. M. McMahon, "The Livermore Fortran Kernels: A Computer Test of Numerical Performance Range," *Tech. Rep. UCRL-55745*, Lawrence Livermore National Laboratory, Univ. of California, October 1986.
- 10. D. Parkinson, "Parallel Efficiency can be Greater than Unity," Parallel Computing, 3 (1986), pp. 261-262.
- 11. L. Pointer, "PERFECT: Performance Evaluation for Cost-Effective Transformations, Report 2," CSRD Report No. 964, March, 1990.
- 12. C. L. Seitz, "The Cosmic Cube," Communications of the ACM, Volume 28 (1985), pp. 22-33.
- 13. SPEC, "SPEC Benchmark Suite Release 1.0," October, 1989.
- 14. R. P. Weicker, "Dhrystone: A Synthetic Systems Programming Benchmark," Communications of the ACM, 27:10, October, 1984.
- 15. P. H. Worley, "The Effect of Time Constraints on Scaled Speedup," *Report ORNL/TM-11031*, Oak Ridge National Laboratory, January, 1989.

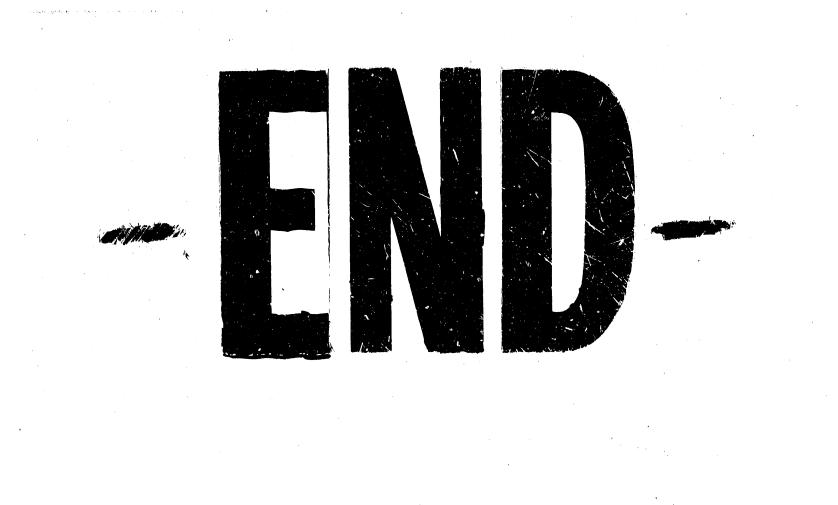
J

JOHN GUSTAFSON received the B.S. degree in applied mathematics from Caltech (1977), the M.S. and the Ph.D. at Iowa State University in 1981 and 1982, respectively. He was Product Development Manager and Senior Staff Scientist at Floating Point Systems from 1982 to 1986, Staff Scientist at NCUBE from 1986 to 1987, and a Member of the Technical Staff at Sandia National Laboratories from 1987 to 1989. His work on the 1024-processor hypercube at Sandia, with colleagues Gary Montry and Robert Benner, won the inaugural Gordon Bell award in 1988. Since 1989, he has led research efforts in massively-parallel computing at the Ames Laboratory. Dr. Gustafson is a Subject Area Editor for Performance Evaluation for the *Journal of Parallel and Distributed Computing*. His interests include computational physics and chemistry, novel performance metrics, and parallel algorithms. He is a member of SIAM.

DIANE ROVER received the B.S. degree in computer science in 1984, the M.S. degree in computer engineering in 1986, and the Ph.D. degree in computer engineering in 1989, all from Iowa State University. From 1985 to 1988, she was awarded an IBM Graduate Fellowship. In 1986, Dr. Rover was an intern with McDonnell Douglas Corporation, and in 1987, with the IBM Thomas J. Watson Research Center. Since 1983, she has been a Technical Education Consultant for IBM. She is currently a postdoctoral researcher in the Scalable Computing Facility at the Ames Laboratory. Her research interests include parallel processing, computer architecture, performance evaluation, instrumentation, and performance visualization. Dr. Rover is a member of the IEEE Computer Society, the Association for Computing Machinery, Sigma Xi, and the Society of Women Engineers.

STEPHEN ELBERT received the B.S. degree in chemistry from Iowa State University in 1968 and the Ph.D. in theoretical chemistry from the University of Washington in 1973. He was a postdoctoral fellow at the University of Bonn from 1973-1975 and at Iowa State from 1975 to 1977. Since 1977 he has been a research scientist on the staff of the Ames Laboratory. His research interests include large scale *ab initio* quantum chemistry calculations to determine the reaction surfaces of small molecules, with particular emphasis on the efficiency of the algorithms involved. He is a member of Sigma Xi.

MICHAEL CARTER received the B.S and M.S. degrees in electrical engineering from Oklahoma State University in 1987 and 1989, respectively, and is a Ph.D. candidate in the Department of Electrical Engineering and Computer Engineering at Iowa State University. His interests include image synthesis, parallel algorithms, and computer architecture. Mr. Carter is a member of the ACM, IEEE Computer Society, Phi Kappa Phi, and Tau Beta Pi. He is currently a research assistant at the Scalable Computing Facility at the Ames Laboratory.



DATE FILMED 02/13/91

1 f n i ана на селото на село Селото на с Селото на с

а. 19

the second se