Toward a better parallel performance metric *

Xian-He Sun and John L. Gustafson
Ames Laboratory, Iowa State University, Ames, IA 50011-3020, USA

Received April 1991
Revised June 1991
Errata corrected by second author, August 2013

Abstract

Sun, X.-H. and J.L. Gustafson, Toward a better parallel performance metric, Parallel Computing 17 (1991) 1093-1109.

The traditional definition of 'speedup' as the ratio of sequential execution time to parallel execution time has been widely accepted. One drawback to this metric is that it tends to reward slower processors and inefficient compilation with *higher speedup*. It seems unfair that the goals of high speed and high speedup are at odds with each other. In this paper, the 'fairness' of parallel performance metrics is studied. Theoretical and experimental results show that the most commonly used performance metric, *parallel speedup*, is 'unfair', in that it favors slow processors and poorly coded programs. Two new performance metrics are introduced. The first one, *sizeup*, provides a 'fair' performance measurement. The second one is a generalization of speedup – the *generalized speedup*, which recognizes that speedup is the ratio of speeds, not times. The relation between sizeup, speedup, and generalized speedup are studied. The various metrics have been tested using a real application that runs on an nCUBE 2 multicomputer. The experimental results closely match the analytical results.

Keywords. Parallel processing; performance measurement; parallel speedup; scaled speedup; sizeup

1. Introduction

Parallel processing has become a common approach for achieving high performance. Various parallel computer systems have been constructed and many parallel algorithms have been developed. However, effective techniques for evaluating the performance of these parallel machines and algorithms are lacking. There is no well-established metric to measure the performance gain of parallel processing. The most frequently used performance metric of parallel processing is *parallel speedup*. Almost twenty years ago, Ware [1] first summarized Amdahl's [2] arguments as a speedup formula, which today is widely known as *Amdahl's law*. Amdahl's law shows the limitation of parallel processing and was the only well-known performance criterion of parallel processing until scientists at Sandia National Laboratories questioned its underlying assumptions [3]. Based on their experimental results, Gustafson et al. revised Amdahl's law and proposed the *scaled speedup* principle [3, 4]. The argument of Gustafson et al. is that parallel processing gives the ability to tackle previously out-of-reach large-scale problems. So, as problem size is increased with computation power, the serial component as defined by Amdahl cannot be regarded as constant. Since then, intensive

* This research is supported by the Applied Mathematical Sciences Program of the Ames Laboratory, which is operated for the U.S. Department of Energy under contract No. W-7405-ENG-82.

research has been conducted to seek a better understanding of parallel speedup. In 1989, Van-Catledge [5] and Zhou [6] studied the relation between Amdahl's law and scaled speedup; Eager et al. [7] studied the speedup versus efficiency, Hockney [8] introduced the parameter $f_{1/2}$ to characterize the influence of memory references, and Barton and Withers [9] developed a speedup model which considers the manufacturing cost of processors as a performance factor. In 1990, the time constraint of scaled speedup was studied by Worley [10], the relation between different speedup models was studied by Sun and Ni [11], and a new metric was proposed to reveal aspects of the performance that are not easily discerned from other metrics [12].

Traditionally, speedup is defined as sequential execution time over parallel execution time. However, there are subtle differences in the way this definition has been interpreted. One definition emphasizes how much faster a problem can be solved with parallel processing based on partial ordering descriptions of the algorithm. Thus, the chosen sequential algorithm is the best sequential algorithm available. This definition is referred to as *absolute speedup*. Another definition of speedup, called *relative speedup*, deals with the inherent parallelism as the single processor execution time of the parallel algorithm, and defines the speedup as

$$S_N = \frac{\text{execution time using one processor}}{\text{execution time using } N \text{ processors}}. \tag{1}$$

The reason for using relative speedup is that the performance of parallel algorithms varies with the number of available processors. Comparing the algorithm itself with a different number of processors gives information on the variations and degradations of parallelism. While absolute speedup has been used to evaluate parallel algorithms, relative speedup is favored [13], especially in performance studies. The relative speedup (1) was used in all the above referenced studies. In this study we also focus on relative speedup, and we reserve the phrase *traditional speedup* for definition (1), unless we explicitly state otherwise.

In contrast with all the above referenced work, in our research the 'fairness' of performance metrics of parallel algorithms is studied. Experimental and analytical results are first presented to show that the traditional speedup favors slow processors and poorly-coded programs. Then, a new performance measurement metric, *sizeup*, is proposed. Our results show that the new performance metric provides a better 'fairness' than the traditional speedup. Another performance metric, which is a generalization of traditional speedup and which first appeared in [14], is also described and studied. Finally, the relation between sizeup, the *generalized speedup*, and the traditional speedup is presented. A real, scientific application is used to test the various metrics. The implementation results using the application provide confirmation of our theoretical results.

We assume the target machines are homogeneous, distributed-memory multiprocessors. We do not specifically consider vector architectures, because they can be considered a special form of parallel execution. The assumption of distributed memory is similarly unrestrictive. If we take the network and memory contention as the communication cost, then all the presented results can be extended to shared-memory parallel systems directly.

This paper is organized as follows: In section 2 we introduce some preliminary information and terminology. The need for better-defined performance metrics is discussed in section 3. A new performance metric, *sizeup*, is introduced in section 4. Analytical and empirical results suggest that sizeup provides a performance measure that is better than traditional speedup when the goal is to solve larger problems rather than reduce execution time. In section 5, another performance metric, which is a generalization of traditional speedup, is introduced. The relation between the new metrics and traditional speedup is also given in section 5. Conclusions and remarks are given in section 6.

2. Background and terminology

In our study we consider two main degradations of parallelism, *load imbalance* and *communication cost*. The former degradation is due to the uneven distribution of workload among processors, and is application-dependent. The latter degradation is due to the communication processing and latency. It depends on both the application and the parallel computer under consideration. To give an accurate performance measurement, both of the degradations need to be considered. Load imbalance is measured by *degree of parallelism*.

Definition 1. The *degree of parallelism* is an integer that indicates the maximum number of processors that can be busy computing at a particular instant in time, given an unbounded number of available processors.

The degree of parallelism is a function of time. By drawing the degree of parallelism over the execution time of an algorithm, a graph can be obtained. We refer to this graph as the *parallelism profile.* Some software tools are available to determine the parallelism profile of large scientific and engineering applications [15]. *Figure* 1 is the parallelism profile of a hypothetical divide-and-conquer computation [16]. By accumulating the time spent at each degree of parallelism, the profile can be rearranged to form the *shape* of the algorithm [17] (*Fig. 2*). The different shadings depict the time period of different operation costs which are defined in Definition 2.

Let W be the amount of work (computation) of an algorithm. Let W_i be the amount of work executed with degree of parallelism i, and m be the maximum degree of parallelism. Thus, $W = \sum_{i=1}^{m} W_i$. Sequential (scalar) execution time can be written in terms of work:

$$\text{Sequential execution time} = \text{Amount of work} \times \frac{\text{Processor cycles per unit of work}}{\text{Machine clock rate}}. \quad (2)$$

Note that we do not break out a startup time and a time per element as would be appropriate for vector arithmetic. Vectorization can be treated as a special form of parallelism. An application may contain more than one work type. Different work types may require different numbers of processor cycles and, therefore, consume different execution times. Depending on how the work is defined (e.g., instructions, floating point operations), how the work is measured (e.g., the number of floating point operations required for finding a square root), and the architecture of the parallel system (e.g. the size of the cache), the difference of cycle requirements may be due to various reasons. One simple reason is the ratio of computation to memory reference. Hockney studied the influence of computation to memory reference ratio on the

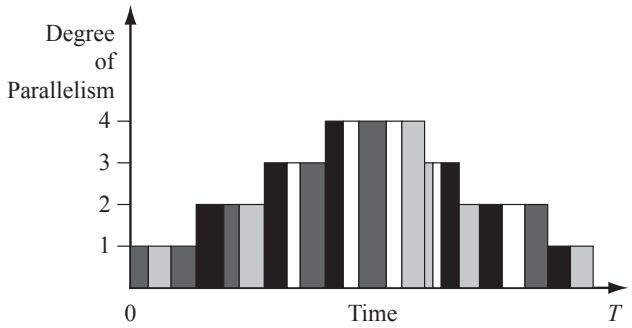

Fig. 1. Parallelism profile of an application.

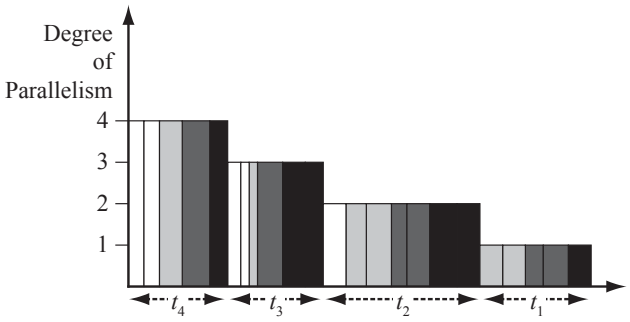

Fig. 2. Shape of the application.

```
      do 10 i=1, n
         s=s+a(i)*b(i)
10    continue
```

Fig. 3. Dot product.

performance of vector machines [8]. Here, we use two basic linear algebra operations, DOT and AXPY [18], to demonstrate how the ratio of computation and memory reference influences execution time. We count floating-point add and floating-point multiply each as one floating point operation, which takes one processor cycle to finish, and assume that each memory operation, including read and write, also takes one processor cycle to finish. In the DOT operation (see *Fig. 3*) the variable *s* can be kept in a register. The computation to memory reference ratio is 1. In the AXPY operation (see *Fig. 4*), the scalar *a* is kept in a register and two read memory references and one write memory reference are required for each loop. The computation to memory reference ratio of AXPY is $\frac{2}{3}$. The work presented in *Fig. 3* and the work presented in *Fig. 4* have the same number of floating point operations. However, they consume different execution times. Since the computation to memory reference ratio of AXPY is less than the ratio of DOT, AXPY consumes more execution time than DOT and requires more processor cycles to execute one floating point operation than DOT does. In general, uniprocessor work rate may vary throughout an application.

We define the *operation cost* of work type j, C_j, as follows:

Definition 2. The *operation cost* of work type j is the ratio of the number of processor cycles required to perform one unit operation of work type j to the total number of processor cycles in one unit of time.

The execution time for computing W_i with a single processor can be represented in terms of operation costs:

$$t_i(1) = \sum_{j=1}^{k} C_j W_{ij},$$

where W_{ij} is the total work of type j executed with degree of parallelism I, and we assume that

```
      do 20 j=1, n
           s(j)=s(j)+a*b(j)
20    continue
```

Fig. 4. Vector update.

there are k different types of work. If there are i processors available and the algorithm is homogeneous in work type at any time, the execution time will be

$$t_i(i) = \frac{\sum_{j=1}^{k} C_j W_{ij}}{i}.$$

With an infinite number of processors available, the execution time will be

$$t_i(i) = t_i(\infty) = \frac{\sum_{j=1}^{k} C_j W_{ij}}{i} \text{ for } 1 \leq i \leq m.$$

Therefore, without considering communication latency and assuming the algorithm is homogeneous in work type at any given time, the total time on a single processor and on an infinite number of processors will be

$$T(1) = \sum_{i=1}^{m} \sum_{j=1}^{k} C_j W_{ij} \tag{3}$$

$$T(\infty) = \sum_{i=1}^{m} \frac{\sum_{j=1}^{k} C_j W_{ij}}{i} \tag{4}$$

The traditional speedup will be

$$S_\infty = \frac{T(1)}{T(\infty)} = \frac{\sum_{i=1}^{m} \sum_{j=1}^{k} C_j W_{ij}}{\sum_{i=1}^{m} \frac{\sum_{j=1}^{k} C_j W_{ij}}{i}} \tag{5}$$

When $k = 1$, all the work is done at the same operation cost, and eq. (5) becomes

$$S_\infty = \frac{\sum_{i=1}^{m} W_i}{\sum_{i=1}^{m} \frac{W_i}{i}} \tag{6}$$

which is the eq. (3) given in [11].

S_∞ gives the best possible speedup based on the inherent parallelism of an algorithm. No machine-dependent factors are considered. With only a limited number of processors and with the communication cost considered, the speedup will be less than the ideal speedup S_∞. If there are N processors available and $N < i$, then some processors have to do $W_i/i \lceil i/N \rceil$ work and the rest of the processors will do $W_i/i \lfloor i/N \rfloor$ work. In this case, assuming W_i and W_j cannot be solved simultaneously for $i \neq j$ and assuming the algorithm is homogeneous in work type at any given time, the elapsed time will be

$$t_i(N) = \frac{\sum_{j=1}^{k} C_j W_{ij}}{i} \left\lceil \frac{i}{N} \right\rceil$$

and

$$T(N) = \sum_{i=1}^{m} \frac{\sum_{j=1}^{k} C_j W_{ij}}{i} \left\lceil \frac{i}{N} \right\rceil. \tag{7}$$

The traditional speedup is

$$S_N = \frac{T(1)}{T(N)} = \frac{\sum_{i=1}^{m} \sum_{j=1}^{k} C_j W_{ij}}{\sum_{i=1}^{m} \frac{\sum_{j=1}^{k} C_j W_{ij}}{i} \left\lceil \frac{i}{N} \right\rceil}. \tag{8}$$

Communication cost is an important factor contributing to the complexity of a parallel algorithm. Unlike degree of parallelism, communication cost is machine-dependent. It depends on the communication network, the routing scheme, and the adopted switching technique. Let $Q_N(W)$ be the communication overhead when N processors are used and the problem size is W; the speedup becomes

$$S_N = \frac{T(1)}{T(N)} = \frac{\sum_{i=1}^{m} \sum_{j=1}^{k} C_j W_{ij}}{\sum_{i=1}^{m} \frac{\sum_{j=1}^{k} C_j W_{ij}}{i} \left\lceil \frac{i}{N} \right\rceil + Q_N(W)}. \tag{9}$$

Three models of speedup were studied in [11]. They are *fixed-size speedup*, *fixed-time speedup*, and *memory-bounded speedup*. Fixed-size speedup fixes the problem size. With more and more computation power available, the problem can be solved in less and less time. For the fixed-time speedup, when more computation power is available, we increase the problem size, do more operations, get a more accurate solution, and keep the execution time unchanged. The memory-bounded speedup also scales the problem size with the number of processors available. The difference between memory-bounded speedup and fixed-time speedup is that in memory-bounded speedup the memory capacity is the dominant limiting factor of the scaling. The execution time can vary in the memory-bounded speedup model. Speedup formulation (9) is the extended fixed-size speedup given in [11] when the operation cost is considered as an influential factor. When the problem size is scaled up, the work at different degrees of parallelism may vary differently. Let W'_{ij} be the amount of scaled work of type j executed with degree of parallelism i in the fixed-time model, and let m' be the maximum degree of parallelism of the scaled problem in the fixed-time model, $W' = \sum_{i=1}^{m} \sum_{j=1}^{k} W'_{ij}$. Define W^*_{ij}, W^*, and m^* similarly for the memory-bounded model. Following the arguments similar to those used in [11] for fixed-time speedup and memory-bounded speedup respectively, when N processors are available, we have

$$S'_N = \frac{\sum_{i=1}^{m'} \sum_{j=1}^{k} C_j W'_{ij}}{\sum_{i=1}^{m'} \frac{\sum_{j=1}^{k} C_j W'_{ij}}{i} \left\lceil \frac{i}{N} \right\rceil + Q_N(W')} = \frac{\sum_{i=1}^{m'} \sum_{j=1}^{k} C_j W'_{ij}}{\sum_{i=1}^{m} \sum_{j=1}^{k} C_j W_{ij}}. \tag{10}$$

$$S_N^* = \frac{\Sigma_{i=1}^{m^*} \Sigma_{j=1}^{k} C_j W_{ij}^*}{\Sigma_{i=1}^{m^*} \frac{\Sigma_{j=1}^{k} C_j W_{ij}^*}{i} \left\lceil \frac{i}{N} \right\rceil + Q_N(W^*)}. \quad (11)$$

Equation (10) is the fixed-time speedup when operation costs are considered, and eq. (11) is the memory-bounded speedup when operation costs are considered. When $k = 1$ they are equal to speedup (9) and speedup (10) proposed in [11], respectively.

3. The need for a better speedup definition

In recent years, many parallel algorithms have been developed for different parallel computing systems. These parallel algorithms are machine-dependent and many have been fine-tuned for a given computer architecture to achieve high performance. The tuning process is painful and elusive. Application codes are difficult to optimize uniformly across different architectures. Parallel algorithms are commonly compared with different programming efforts and compared indirectly by using results from different machines [19]. These practical approaches raise some questions: Does the same algorithm achieve the same performance on different computer systems? Does the optimized code, which gives the shortest execution time, provide the same or better performance than the unoptimized code? For a performance metric to be 'fair', we want the same algorithm to achieve the same or better performance than the unoptimized code. Two new terminologies are needed for the 'fairness' study.

Definition 3. A performance metric is *machine-independent* if the performance is the same on any computing system for a given algorithm when the communication cost is negligible.

Definition 4. A performance metric is *programming-independent* if the performance is independent of the programming effort when the communication cost is negligible.

It is easy to see that speedup (1) is machine-independent if and only if it is programming-independent. However, speedup (1), the most commonly used performance metric, is machine- and programming-dependent. Based on his study on vector processing and fixed-size speedup, in 1983 Hockney noticed that using speedup as a merit for parallel algorithms can be misleading [20]. He introduced a new metric T_p^{-1} in 1987 [21] and presented a more detailed study on *what's wrong with speedup* in his latest work [13]. The tendency of traditional speedup to favor slow processors has also been noticed. Barton and Withers [9] studied the influence of computation capacity of the processing elements on speedup. They used four different kinds of processing elements, namely 386 (i80386), 387, (i80387), SX (Weitek 1167), and VX (Analog Devices) to run an algorithm on iPSC/2 systems. Their implementation results show that the slowest processor, 386, is virtually linear in speedup, but provides low speed (in the sense of MFLOPS). The fastest processor, the VX board, achieves a much lower speedup while it provides the maximum speed and the shortest elapsed time. Theorem 1 shows that, even without considering the communication cost, the traditional speedup favors slower processing elements and is machine-dependent. In our proof we do not consider the communication cost, and we assume that the degree of parallelism only has two cases, a sequential part and a perfectly parallel part. The restriction on degree of parallelism is only for clarity; this result can be extended to algorithms with a general degree of parallelism.

Lemma 1. *If computing system one has operation costs* C_1, C_2 *for sequential work* $W_1 > 0$ *and parallel work* W_N *respectively; computing system two has operation cost* C_1 *for sequential work and has operation cost* C_2' *for parallel work where* $C_2' < C_2$*, then system two provides a smaller traditional speedup than system one. That is, when there are* $N > 1$ *processors available and* $W_1 > 0, W_N > 0$,

$$\frac{C_1W_1 + C_2W_N}{C_1W_1 + \frac{C_2W_N}{N}} > \frac{C_1W_1 + C_2'W_N}{C_1W_1 + \frac{C_2'W_N}{N}}.$$

Proof. Since $C_2 > C_2'$, we have $C_2 - C_2' > 0$. Therefore,

$$C_2 - C_2' > \frac{1}{N}(C_2 - C_2')$$

$$C_2W_N - C_2'W_N > \frac{C_2W_N}{N} - \frac{C_2'W_N}{N}$$

$$C_2W_N + \frac{C_2'W_N}{N} > C_2'W_N + \frac{C_2W_N}{N}.$$

Since $C_1W_1 > 0$, we have

$$C_1W_1C_2W_N + C_1W_1\frac{C_2'W_N}{N} > C_1W_1C_2'W_N + C_1W_1\frac{C_2W_N}{N},$$

add terms $C_1W_1C_1W_1 + C_2W_NC_2'W_N/N$ to both sides and factor:

$$(C_1W_1 + C_2W_N)\left(C_1W_1 + \frac{C_2'W_N}{N}\right) > (C_1W_1 + C_2'W_N)\left(C_1W_1 + \frac{C_2W_N}{N}\right),$$

and

$$\frac{C_1W_1 + C_2W_N}{C_1W_1 + \frac{C_2W_N}{N}} > \frac{C_1W_1 + C_2'W_N}{C_1W_1 + \frac{C_2'W_N}{N}}. \quad \square \tag{12}$$

In order to extend our result to more general cases, we introduce the following definition.

Definition 5. The *average operation cost* of degree of parallelism *i* is the average operation cost divided by the work of degree of parallelism *i* for a given algorithm.

$$\text{Average operation cost of degree of parallelism } i = c_i = \frac{\sum_{j=1}^{k} C_jW_{ij}}{\sum_{j=1}^{k} W_{ij}} \tag{13}$$

We now show that traditional speedup favors slower computing systems.

Theorem 1. *If computing system one has operation costs* $C_j, j = 1, \ldots, k$*, and computing system two has operation costs* $C_j', j = 1, \ldots, k$*, where* $C_j = C_j'$*, for* $j = 1, \ldots, t, C_j > C_j'$*, for* $j = t + 1, \ldots, k$*, the computation of sequential work only involves work of type j,* $1 \le j \le t$ *and the*

computation of parallel work W_N involves some work of type j where $j > t$, then system one provides a greater traditional speedup than system two. That is

$$\frac{\sum_{j=1}^{t} C_j W_{1j} + \sum_{j=t+1}^{k} C_j W_{Nj}}{\sum_{j=1}^{t} C_j W_{1j} + \sum_{j=t+1}^{k} \frac{C_j W_{Nj}}{N} \left\lceil \frac{j}{N} \right\rceil} > \frac{\sum_{j=1}^{t} C_j' W_{1j} + \sum_{j=t+1}^{k} C_j' W_{Nj}}{\sum_{j=1}^{t} C_j' W_{1j} + \sum_{j=t+1}^{k} \frac{C_j' W_{Nj}}{N} \left\lceil \frac{j}{N} \right\rceil}.$$

Proof. Since $C_j = C_j'$, for $j \le t$, the average operation cost for sequential work of computing system one, c_1, is the same as the average operation cost for sequential work of computing system two, $c_1 = c_1'$. Since $C_j > C_j'$ for $j > t$, the average operation cost for parallel work of computing system one, c_2, is greater than the average operation cost for parallel work of computing system two, c_2'.

$$c_2 = \frac{\sum_{j=1}^{k} C_j W_{Nj}}{\sum_{j=1}^{k} W_{Nj}} > \frac{\sum_{j=1}^{k} C_j' W_{Nj}}{\sum_{j=1}^{k} W_{Nj}} = c_2'.$$

Therefore, by Lemma 1,

$$\frac{c_1 W_1 + c_2 W_N}{c_1 W_1 + \frac{c_2 W_N}{N}} > \frac{c_1 W_1 + c_2' W_N}{c_1 W_1 + \frac{c_2' W_N}{N}}.$$

That is

$$\frac{\sum_{j=1}^{t} C_j W_{1j} + \sum_{j=1}^{k} C_j W_{Nj}}{\sum_{j=1}^{t} C_j W_{1j} + \sum_{j=1}^{k} \frac{C_j W_{Nj}}{N} \left\lceil \frac{j}{N} \right\rceil} > \frac{\sum_{j=1}^{t} C_j' W_{1j} + \sum_{j=1}^{k} C_j' W_{Nj}}{\sum_{j=1}^{t} C_j' W_{1j} + \sum_{j=1}^{k} \frac{C_j' W_{Nj}}{N} \left\lceil \frac{j}{N} \right\rceil}. \quad \square \tag{14}$$

Operation costs not only depend on the parallel system and the algorithm, but also on the programming effort on the node. Operation cost can be reduced by careful coding. For instance, the Vector update (see *Fig. 4*) can be rewritten in assembly language to reduce the operation cost. Theorem 1 can be presented differently to show that the traditional speedup is programming-dependent and favors poorly-coded programs.

Theorem 2. *If a computing system has operation costs $C_j, j = 1, \ldots, k$, for a program and has operation costs $C_j', j = 1, \ldots, k$ for the optimized version of the program, where $C_j = C_j'$, for $j = 1, \ldots, t, C_j > C_j'$, for $j = t + 1, \ldots, k$, the computation of sequential work only involves work of type j, $1 \le j \le t$ and the computation of parallel work W_N involves some work of type j where $j > t$, then the optimized version provides a smaller speedup than the original program.*

A real application, the *radiosity application* [22], is chosen to provide experimental results. Radiosity is the equilibrium radiation given off by a coupled set of diffuse surfaces that emit and absorb radiation. The sample radiosity program is a diagonally-dominant dense matrix problem and is easily 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

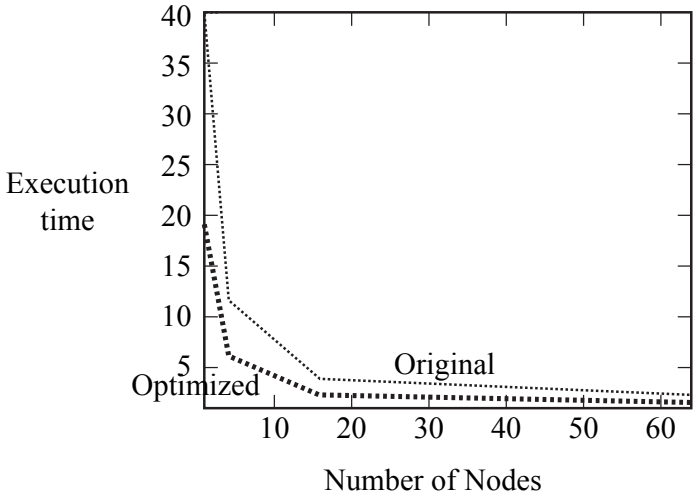

Fig. 5. Execution time of two implementations.

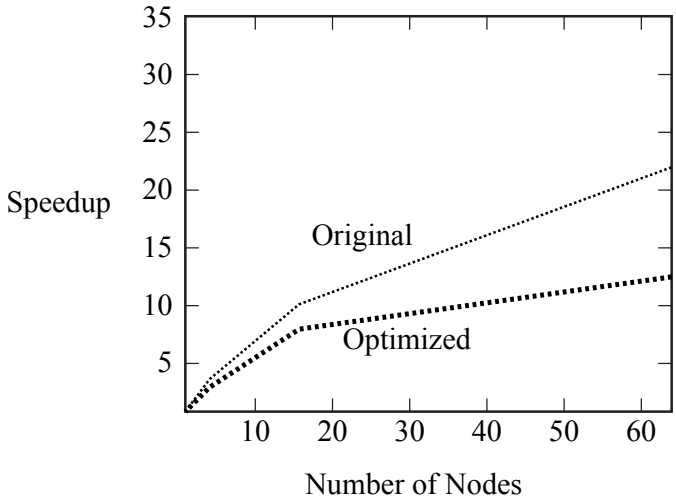

Fig. 6. The traditional speedup of two implementations

are described as red-green-blue components for each wall of the room. The radiosity problem is to find the color variation over each wall.

We implemented the radiosity algorithm on a 64-node nCUBE 2 multicomputer with two different versions of the program. One was written in FORTRAN and another version optimized certain subroutines of the FORTRAN code with assembly language. The implemented results of the two codes are depicted in *Fig. 5* and *Fig. 6*. From the implementation results we can see that, while the optimized version provides a shorter elapsed time, the unoptimized version provides a higher traditional speedup. A researcher attempting to achieve high speedup on a parallel system is therefore at odds with the more basic goal of achieving high net performance.

4. A new performance metric

Theorem 1 and Theorem 2 show that the traditional speedup favors slow processors and poorly-coded programs even when the communication cost is not considered. Thus, performance measurements given in terms of speedup must be interpreted with particular care. New, better-defined performance metrics are needed to provide a ‘fair’ performance measurement. The scaled speedup concept [3,4] lets the problem size increase with computation power and emphasizes how much work is done in a fixed time. A natural step in the search for a better

metric is to follow the scaled concept and define a new metric, called *sizeup*, as parallel work over sequential work:

$$\text{Sizeup} = \frac{\text{parallel work}}{\text{sequential work}}. \tag{15}$$

In a sizeup measurement, the problem size is scaled so as to keep the parallel execution time a constant, as the number of processors increase. Since more than one process is used in parallel processing, in general more work will be done in parallel processing. Sizeup indicates the ratio of the work increase. The term *sizeup* was coined by G. Montry in 1987 during the experiments done at Sandia [3].

Theorem 3 shows that sizeup is machine-independent.

Lemma 2. *If computing system one has operation costs C_1 and C_2 for sequential work W_1 and parallel work W_N respectively; computing system two has operation cost C_1 for sequential work and has operation cost C_2' for parallel work where $C_2' < C_2$, then, system one and system two provide the same sizeup.*

Proof. In sizeup, the workload is scaled with the number of processors available. Let the original workload be $W, W = W_1 + W_N$, and let the scaled workload for system one be $W' = W_1 + W_N'$ and the scaled workload for system two be $W^* = W_1 + W_N^*$. Since, by the definition of sizeup, the parallel execution time is fixed with the sequential execution time, we have

$$C_1 W_1 + C_2 W_N = C_1 W_1 + \frac{C_2 W_N'}{N}$$
$$W_N' = N W_N,$$

and

$$C_1 W_1 + C_2' W_N = C_1 W_1 + \frac{C_2 W_N^*}{N}$$
$$W_N^* = N W_N,$$

Therefore,

$$W_N' = W_N^*$$
$$\frac{W'}{W} = \frac{W^*}{W}. \quad \square \tag{16}$$

Theorem 3. *If computing system one has operation costs $C_j, j = 1, \ldots, k$ and computing system two has operation costs $C_j', j = 1, \ldots, k$, where $C_j = C_j'$ for $j = 1, \ldots, t$; the computation of sequential work is independent of the number of processors available and only involves work of type $j, 1 \le j < t$, then system one and system two provide the same sizeup.*

Proof. Use Lemma 2 and follow arguments similar to those used in the proof of Theorem 1. □

If we assume program tuning only improves the parallel portion of the algorithm, which is the common case in practice, then Theorem 3 also shows that sizeup is programming-independent. This can be shown as follows.

Theorem 4. *If a computing system one has operation costs $C_j, j = 1, \ldots, k$ for a given program and has operation costs $C_j', j = 1, \ldots, k$, for the optimized version of the given program, where*

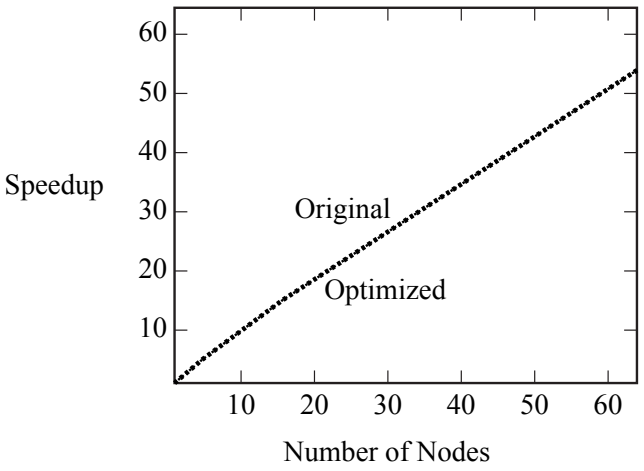

Fig. 7. Sizeup of two implementations.

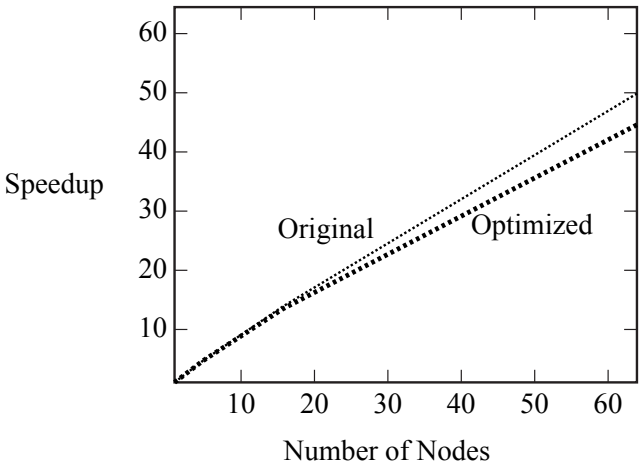

Fig. 8. Sizeup when data collection phase is added.

$C_j = C_j'$ *for* $j = 1, \ldots, t$*; the computation of sequential work is independent of the number of processors available and only involves work of type* $j, 1 \leq j < t$*, then the given program and the optimized version achieve the same sizeup.*

Theorems 3 and 4 have been confirmed by empirical results from the radiosity algorithm. As shown in *Fig. 7*, under assumptions of Theorem 3 and Theorem 4 respectively, sizeup is

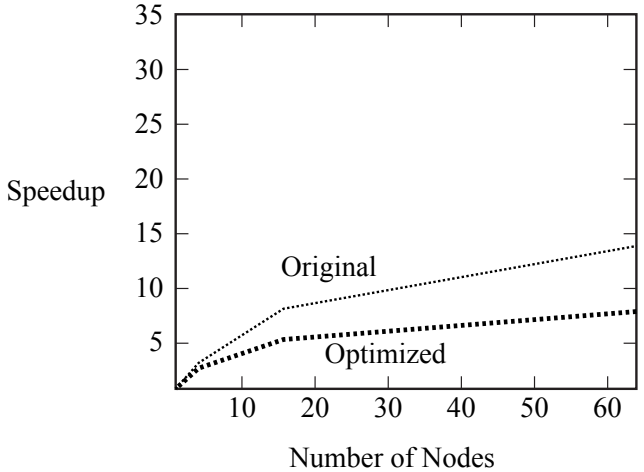

Fig. 9. The traditional speedup when data collection is added.

machine- and programming-independent. However, if the sequential work increases with the problem size, then sizeup is no longer independent of either. To test this case, we modified the radiosity algorithm to contain a data collection phase: All computed results are collected to a single mass storage device. The data collection phase is a sequential process and increases with problem size. The modified radiosity algorithm is also implemented on an nCUBE multicomputer. *Figure 8* shows that, when the sequential portion of a program increases with the problem size, the sizeup also favors slower processors or poorly coded programs. However, the dependence is much less than with traditional speedup; compare *Fig. 8* and *Fig. 9.*

5. The generalized parallel speedup

Computation capacity of processors is generally measured by MIPS (Millions of Instructions Per Second) or MFLOPS (Millions of FLoating point Operations Per Second). The performance of processors is given in speed: work divided by time. (The particular choice of unit for work is irrelevant to the arguments that follow.) Since the computing capacity of processors is given in speed, a natural way to measure parallel processing gain would be the ratio of parallel execution speed to sequential execution speed [14]. We introduce a new metric, which we call *generalized speedup.*

$$\text{Generalized Speedup} = \frac{\text{parallel execution speed}}{\text{sequential execution speed}}. \tag{17}$$

We have introduced two new performance metrics. In the rest of this section we study the relation between the newly-proposed performance metrics and traditional speedup. The results show that traditional speedup is the restriction of (17) to fixed work or to fixed operation cost, and sizeup is the restriction of (17) to fixed time. In the following we use S_N to represent the generalized speedup (17), and use S_N to represent the traditional speedup (1).

By definition

$$S_N = \frac{\text{parallel execution speed}}{\text{sequential execution speed}} \tag{18}$$

$$= \frac{\left(\frac{\text{parallel work}}{\text{parallel execution time}}\right)}{\left(\frac{\text{sequential work}}{\text{sequential execution time}}\right)} \tag{19}$$

$$= \frac{\left(\frac{\boldsymbol{W}'}{\sum_{i=1}^{m} \frac{C_1 \boldsymbol{W}_i'}{i\lceil \frac{i}{N} \rceil} + Q_N \boldsymbol{W}'}\right)}{\left(\frac{W}{\sum_{i=1}^{m}\sum_{j=1}^{k} C_j W_{ij}}\right)} \tag{20}$$

The workload $\boldsymbol{W}'$ is the parallel workload. It may be scaled up with system size if the fixed-time or memory-bounded speedup is used. If the fixed-size model of speedup is used, then $\boldsymbol{W}' = W$ and $\boldsymbol{W}_i' = W_i$.

When we have a unique operation cost, that is, $C_i = C_1$, for $j = 2, \ldots, k$, then

$$S_N = \frac{\frac{W'}{\sum_{i=1}^{m} \frac{C_1 W_i'}{i} \left\lceil \frac{i}{N} \right\rceil + Q_N\left(W'\right)}}{\left(\frac{W}{\sum_{i=1}^{m} C_1 W_i}\right)} = \frac{\frac{W'}{\sum_{i=1}^{m} \frac{W_i'}{i} \left\lceil \frac{i}{N} \right\rceil + Q_N'\left(W'\right)}}{\left(\frac{W}{\sum_{i=1}^{m} W_i}\right)}$$

Where $Q_N'(W') = C_1 Q_N(W')$, and (see eq. (9))

$$S_N = \frac{W'}{\sum_{i=1}^{m} \frac{W_i'}{i} \left\lceil \frac{i}{N} \right\rceil + Q_N'\left(W'\right)} S_N \tag{21}$$

Equation (210 shows that, when we have a unique operation cost, the new metric is the same as the traditional speedup. If the fixed-size speedup is used, eq. (21) is the same as eq. (9). If the fixed-time model is adopted, the workload is scaled up with the execution time; eq. (21) is the same as eq. (10). When the workload is scaled up with memory capacity, then $W' = W^*$ and the speedup S_N is equal to S_N^* given in eq. (11). In the fixed-time case, we also have

$$S_N = \frac{W'}{\sum_{i=1}^{m} \frac{W_i'}{i} \left\lceil \frac{i}{N} \right\rceil + Q_N'\left(W'\right)} = \frac{W'}{W} = \frac{\text{parallel work}}{\text{sequential work}} = \text{sizeup.}$$

Notice, by eq. (19), that when the problem size is fixed,

$$S_N = \frac{\text{sequential execution time}}{\text{parallel execution time using } N \text{ processors}}. \tag{22}$$

which is the same as the definition of the traditional speedup. Equations (21) and (22) show, when the operation cost is unique or when the problem size is fixed, the generalized speedup is the same as traditional speedup. Since, historically, performance issues have been studied under the assumption that the computation capacity is unique [11], or that the problem size is fixed [2], the results of previous studies remain true under the new definition, and the newly-proposed metric can be seen as a generalization of the traditional speedup.

By eq. (19), when the execution time is fixed, we also have

$$S_N = \frac{\text{parallel work}}{\text{sequential work}} = \text{sizeup.} \tag{23}$$

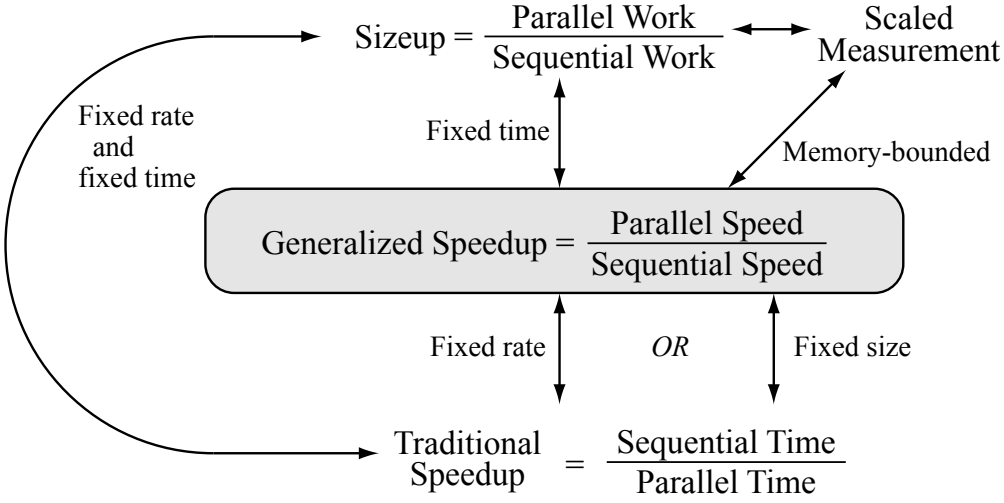

Fig. 10. The generalized speedup.

Sizeup is the fixed-time model of speedup (17). The relation between the generalized speedup (17), sizeup, and the traditional speedup (1) are summarized in *Fig. 10*, in which *Fixed Rate* indicates that the problem has a unique operation cost.

We would like to compare generalized speedup and traditional speedup. Notice in Theorem 3 and Theorem 4 that sizeup, the fixed-time case of the generalized speedup, sets the parallel execution time equal to the sequential execution time. But, with two different computing systems, the sequential time could be different. If system two has lower operation costs than system one, system two may provide a shorter sequential execution time than system one. With more than one parallel system available we have more options for comparison. In Theorem 3 we fixed the workload on system one and system two when on processor is used, and let system one and system two have different sequential execution time, then the traditional speedup will be machine- and programming-independent, no matter whether the fixed-size model of speedup or the fixed-time model of speedup is used. We only give the proof for the fixed-size model of speedup. the result for fixed-time speedup can be obtained similarly.

Proposition 1. *Under the conditions of Lemma 1, if we let system one and system two have the same sequential execution time, then system one and system two provide the same traditional speedup.*

Proof. Since system one and system two have different computing capacity for parallel computation, with fixed time, the work they execute will be different. Let $W = W_1 + W_N$ be the work executed by system one. Let $W' = W_1' + W_N'$ be the work executed by system two. Then,

$$C_1 W_1 + C_2 W_N = C_1 W_1 + C_2' W_N' \tag{24}$$

$$W_N' = \frac{C_1}{C_2'} W_N. \tag{25}$$

and the speedup of system two is equal to the speedup of system one.

$$\frac{C_1 W_1 + C_2' W_N'}{C_1 W_1 + C_2' \frac{W_N'}{N}} = \frac{C_1 W_1 + C_2' \left(\frac{C_2}{C_2'} W_N \right)}{C_1 W_1 + C_2' \frac{\left(\frac{C_2}{C_2'} W_N \right)}{N}} = \frac{C_1 W_1 + C_2 W_N}{C_1 W_1 + C_2 \frac{W_N}{N}}. \quad \square \tag{26}$$

Figure 11 depicts the machine- and programming dependence of traditional speedup and generalized speedup. Some of the results listed in the tables have been proven in this section, and the rest can be proven by following similar arguments. For instance, replacing W_N by

	Size	Time
Size	Dependent	Independent
Time	Independent	Independent

Generalized Speedup

	Size	Time
Size	Dependent	Dependent
Time	Independent	Independent

TraditionalSpeedup

Fig. 11. Tables of dependence.

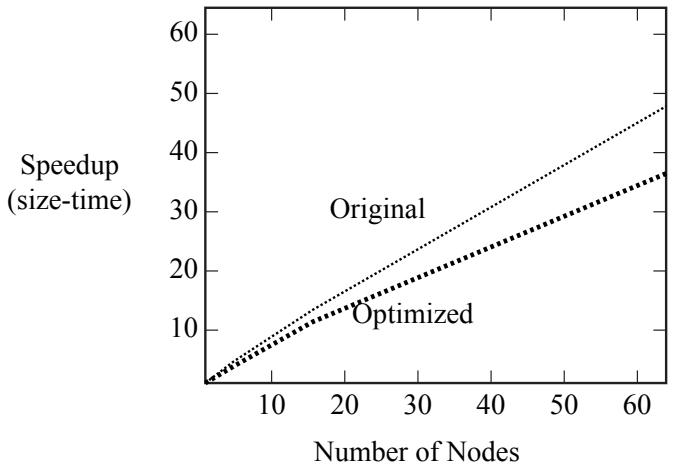

Fig. 12. The size-time model of traditional speedup.

NW_N and following arguments similar to those used in the proof of Lemma 1, we can show that the size-time model of traditional speedup also favors slow processors. The *size* beside the tables indicates that the workload of single processor execution on different systems is fixed. The *time* beside the tables indicates that the single processor execution time of different systems is fixed. The *size* and *time* above the tables represent that fixed-size or fixed-time model of speedup is used, respectively. From *Fig. 11* we can see that speedup (17) is not fully machine- and programming-independent, but it provides less penalty than the traditional speedup. The implementation results of the size-size model of traditional speedup and the size-time model of the generalized model are shown in *Fig. 6* and *Fig. 7* respectively. The implementation results of the size-time model of traditional speedup are shown in *Fig. 12*. From *Fig. 12*, we can see that, while the size-time model of the generalized speedup is machine- and programming-independent, the size-time model of the traditional speedup is not. Generalized speedup provides a little more 'fairness' than traditional speedup.

Conclusion and remarks

We have studied the 'fairness' of parallel performance metrics. We have shown that the most frequently used performance metric, traditional speedup, favors slow processors and poorly-programmed codes. The reason for the 'unfairness' of this model has been identified and quantified. New terminologies and concepts such as operation cost, machine dependence, and programming dependence, which can be used in future studies, have been introduced and carefully defined. Two new performance metrics have been proposed. The first new metric, *sizeup*, provides a 'fair' performance measurement. The second metric, *generalized speedup*, contains both sizeup and the traditional speedup as special cases. Our study focused on the relative speedup (1). The study results can be used to evaluate absolute speedup and other parallel performance metrics. For instance, since relative speedup favors slow processors when the operation cost is fixed (or, in a weaker sense, the faster processors reduce execution time in the same ratio for both the best sequential algorithm and the parallel algorithm on one processor), then the absolute speedup also favors slow processors. Absolute speedup is also machine-dependent.

Due to the diversity of existing architectures, algorithm codes are difficult to optimize uniformly across architectures. Parallel algorithms are commonly compared with different programming efforts and compared indirectly by using implementation results from different machines. This situation is unlikely to change in the near future. A better-defined, 'fair'

performance metric is urgently needed. Sizeup is 'fairer' than traditional speedup. However, sizeup is based on the scaled speedup principle and only provides a 'fair' measurement under certain conditions. Can we find a better, fixed-size performance metric? Can we find a metric that provides an unconditional 'fairness'? Many questions remain.

References

[1] W. Ware, The ultimate computer, *IEEE Spectrum* 9 (1972) 84–91.

[2] G. Amdahl, Validity of the single-processor approach to achieving large scale computing capabilities, in *Proc. AFIPS Conf.* (1967) 483–485.

[3] J. Gustafson, G. Montry and R. Benner, Development of parallel methods for a 1024-processor hypercube, *SIAM J. SSTC* 9 (Jul. 1988).

[4] J. Gustafson, Reevaluating Amdahl's law, *CACM* 31 (May 1988) 523–533.

[5] F.A. Van-Catledge, Toward a general model for evaluating the relative performance of computer systems, *International J. Supercomputer Applications* 3 (2) (1989) 100–108.

[6] X. Zhou, Bridging the gap between Amdahl's law and Sandia Laboratory's result, *Commun. ACM* 32 (8) (1989) 1014–1015.

[7] D. Eager, J. Zahorjan and E. Lazowska, Speedup versus efficiency in parallel systems, *IEEE Trans. Comput.* (Mar 1989) 403–423.

[8] R.W. Hockney and I.J. Curington, $f_{1/2}$: A parameter to characterize memory and communication bottlenecks, *Parallel Computing* 10(3) (1989) 277–286.

[9] M. Barton and G. Withers. Computing performance as a function of the speed, quantity, and cost of the processors, in *Proc. Supercomputing '89* (1989) 759–764.

[10] P. T. Worley, The effect of time constraints on scaled speedup, *SIAM J. SSC.* 11 (Sep. 1990) 838–858.

[11] X.-H. Sun and L. Ni, Another view on parallel speedup, in *Proc. Supercomputing'90* (1990) 324–333.

[12] A.H. Karp and H.P. Flatt, Measuring parallel processor performance, *CACM* 33 (May 1990) 539–543.

[13] R.W. Hockney, Performance parameters and benchmarking of supercomputers, *Parallel Comp.*, this issue.

[14] J. Gustafson, Fixed time, tiered memory, and superlinear speedup, in *Proc. Fifth Conf. on Distributed Memory Computers* (1990)

[15] M. Kumar, Measuring parallelism in computation intensive scientific/engineering applications, *IEEE-TC* 37 (Sep. 1988) 1088–1098.

[16] X.-H. Sun, Parallel computation models: Representation, analysis and applications, Ph.D. Dissertation, Computer Science Department, Michigan State University, 1990

[17] K. Sevcik, Characterizations of parallelism in applications and their use in scheduling, in *Proc. ACM SIGMETRICS and Performance '89* (May 1989).

[18] C. Lawson, R. Hanson, D. Kincaid and F. Krogh, Basic linear algebra subprograms for FORTRAN usage, *ACM Trans. Math. Soft.* 5 (1979) 308–323.

[19] T. Li, H. Zhang and X.-H. Sun, Parallel homotopy algorithm for symmetric tridiagonal eigenvalue problem, *SIAM J. Sci. Stat. Comput.* 12 (May 1991).

[20] R.W. Hockney, Characterizing computers and optimizing the FACR(1) Poisson-solver on parallel unicomputers, *IEEE Trans. Comput.* c.32 (10) (1983) 933–941.

[21] R.W. Hockney, Parameterization of computer performance, *Parallel Comput.* 5 (1&2) (1987) 97–103.

[22] C. Goral, K. Torrance, D. Greenberg and B. Battaile, Modeling the interaction of light between diffuse surfaces, *Comput. Graphics* 18 (Jul. 1984).