HPCN 99

Memory Efficiency of Parallel Programs and Memory Bounded Speedup

M.A. Kartawidjaja[1] and A.G. Hoekstra[2]

1 Faculty of Computer Science, University of Indonesia, Kampus UI, Depok, Depok Indonesia, email: mawiran@ibm.net

2 Parallel Scientific Computing and Simulation group, Faculty of Mathematics and Computer Science, University of Amsterdam, Kruislaan 403, 1098 SJ Amsterdam, the Netherlands, email: alfons@wins.uva.nl

Abstract. We introduce the concept of memory efficiency of a parallel program. Memory efficiency is a measure of the amount of data replication in a parallel program. It describes how well a parallel program is equipped to exploit the available memory in (distributed memory) parallel computers, and whether a parallel program is scalable in its memory usage. We apply memory efficiency in the memory bounded speedup model as introduced by Sun and Ni. We show how the memory efficiency concept simplifies the analysis of the memory bounded speedup model.

1 Introduction

The concept of scalability of parallel computers and parallel programs has played - and still plays - a key role in the advances of parallel computing. Although a strict and formal definition of scalability is hard to produce [1, 2] it is generally agreed upon that scalability expresses if and how a parallel system is able to preserve its performance, measured in some metric, if its size is increased. A large number of scalability metrics have been proposed and studied in the context of many applications and parallel systems. This work has been reviewed by Kumar and Gupta [2].

The best-known and most-used scalability metric is speedup. In its original definition speedup S_p measures the quotient of the execution time T_p of a parallel program running on p processors and T_{seq}, the execution time of the fastest sequential program which solves the same problem. Usually one measures the relative speedup, where T_{seq} is replaced by T_1, the execution time of the parallel program running on 1 processor.

Ware [3] has formulated a speedup law, based on earlier work by Amdahl [4], which we now know as *Amdahl's Law*:

$$S_p = \frac{p}{1+(p-1)\alpha} \qquad (1)$$

where α is the inherently sequential fraction of a program. In Amdahl's law all communication latencies are ignored. If we take the limit of $p \to \infty$ we find $S_\infty = 1/\alpha$, i.e. the speedup is bounded by the sequential part of the program. Amdahl's law raised much skepticism about the potential of massively parallel computing, since a relative small α of say 0.01 would limit the speedup to 100.

In Amdahl's law the amount of computational work is kept constant as the number of processors is increased. This fixed-load leads to Amdahl's sequential bottleneck and prevents to reach very high speedups. Although the fixed-load constraint is essential in some application areas (e.g. real time control applications) the situation in many engineering and research applications is very different.

54

Gustafson [5] has solved Amdahl's sequential bottleneck by formulating a fixed-time concept, which results in scaled speedup models. If the available computational power is increased, one usually increases the computational load too. This is done for instance by performing simulations on finer and/or larger grids, with more particles, or in more dimensions. Instead of trying to solve the same problem faster, one tries to solve a larger problem in approximately the same amount of time. Under the assumption of a constant sequential workload Gustafson's fixed time speedup model results in

$$S_p = p - \alpha(p-1) \, . \qquad (2)$$

Proportional scaling of the parallel workload with the number of processors facilitates large speedups. Gustafson [5] reported measurements of scaled speedups of 1016 to 1021 on three applications actually running on a 1024 node hypercube architecture.

Gustafson has introduced a very important principle which has been applied in all subsequent scalability studies. Scalability is a property which investigates how a performance metric of a parallel system behaves if both the system size *and* the workload of the program running on the system are allowed to increase. The exact formulations of many scalability studies are very different, but in all cases varying the workload is an essential feature of the analysis.

Scaling workloads raises the question if one is allowed to increase the workload indefinitely. Under the constraint of finite memory resources (per processor) this is certainly not possible. Furthermore, especially on distributed memory parallel computers data is frequently replicated in memory of each processor to prevent communication between processors. In this paper we introduce the concept of memory efficiency and apply it to a generalized scaled speedup model, the memory-bounded speedup of Sun and Ni [6]. We will show that memory efficiency allows for a straightforward inclusion of the effect of data replication into the model.

2 Memory Efficiency

Define $m(p)$ as the memory requirement per processor of a parallel program running on p processors.[1] Equivalent to the definition of the efficiency of a parallel program we can define the memory efficiency ε_m of a parallel program as

$$\varepsilon_m = \frac{m(1)}{p\,m(p)} \, . \qquad (3)$$

The memory efficiency is a measure of the scalability of a parallel program in terms of usage of the memory of a parallel computer. In shared memory computers $\varepsilon_m = 1$. However, in distributed memory computers the memory efficiency can take any value between 1/p and 1, due to data replication. In the sequel of the paper we will assume that we are dealing with distributed memory systems.

Let M be the total memory requirement of a (parallel) program on 1 processor, and m_K the size of the memory per node. We assume homogenous nodes, i.e. all nodes have equal amounts of memory. If this program is to be executed on p processors, the following expressions must hold:

1 We assume that m is the memory requirement for the data segment of the parallel program. The code segment and the memory needed by the run time kernel is assumed to be neglegible

$$\frac{M}{m_R} \le \varepsilon_m p \qquad (4)$$

If $\varepsilon_m < 1$ the parallel program in not able to fully use the available distributed memory, and it effectively needs more memory than the sequential program. In the most extreme case ε_m is inversely proportional with p. In that case we find $M/m_R \le k$, with k a constant number. In this situation an increase in the number of processors, and thus the amount of available memory, will *not* allow us to increase M beyond $k\ m_R$. This means that the total workload can also not be increased further, and scaling of the workload with the number of processors is no longer possible, thus putting us back to the fixed-load situation of Amdahl.

We will examine two example programs, a parallel matrix-matrix product and a parallel matrix-vector product. In order to find expressions for the memory efficiency, we have to specify how the parallel workload is carried out. Fig (1) shows how the example problems can be executed in parallel.

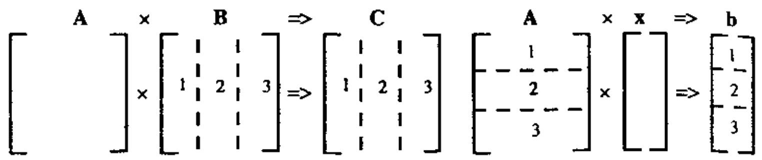

Figure 1: The parallel matrix-matrix product shown left, and the parallel matrix-vector product shown right.

In the matrix-matrix product we assume square, nxn matrices. Matrix **A** in the matrix-matrix product is replicated in every processor, matrix **B** and **C** are column-block decomposed, such that each processor has n/p columns of **B** and **C** in memory. Obviously this is not an optimal way to perform a matrix-matrix product in parallel (for this see e.g. [7]) but it results in a good illustration of the memory efficiency concept.

The parallel matrix-vector product can be carried out by a row-block decomposition of the matrix. The argument vector **x** has to be replicated in memory of each processor. We will investigate two variants of this parallel matrix-vector product. In one case the matrix is kept in main memory, and in the other case the matrix elements are not kept in memory, but are calculated as they are needed. In many applications the matrix **A** is too large to store in memory. In this case the matrix has to be stored on disk, and the elements are retrieved when needed. Obviously, this disk I/O is a very time consuming operation, especially on current parallel systems. Therefore, it is usually preferable to recalculate the matrix elements (assuming that this is possible). As will become clear in the sequel, this approach will result in an enormous degradation of the memory-bounded speedup.

From Fig. (1) we can derive that for the parallel matrix-matrix product the memory requirement per node is[2]

$$m(p) = n^2 + 2n^2 / p. \qquad (5)$$

and therefore the memory efficiency equals

2 Actually, we should write $m(p) = n^2 + 2n\lceil n/p \rceil$, with $\lceil\ \rceil$ the ceiling function. For reasons of clarity of the discussion, we will not include this level of detail into the equations.

$$\varepsilon_m = \frac{3}{2+p} \ . \qquad (6)$$

For large p, the memory efficiency of the matrix-matrix product approaches $3/p$. This means that on a distributed memory parallel system with an infinite number of processors and therefore infinite memory resources, we can only story a matrix-matrix product with 3 times as much memory requirements as the largest product fitting on one node. Obviously, this is caused by the replication of matrix **A** in memory of each processor. Usually, the reason for such data replication is to avoid the need for much (time consuming) communication during execution, which will result in a better efficiency of the program. A better efficiency means a better utilization of processor resources, however at the expense of a very poor utilization of the (much more expensive) memory resources.

The memory requirement of the matrix-vector product, with the matrix stored in memory, is

$$m(p) = n^2 / p + n + n / p. \qquad (7)$$

The resulting memory efficiency is

$$\varepsilon_m = \left[1 + \frac{p-1}{n+2}\right]^{-1} \ . \qquad (8)$$

If we assume that $n >> p$, we find $\varepsilon_m = 1$, i.e. the parallel matrix-vector product can fully utilize the available distributed memory.

Finally consider the matrix-vector product without storing the matrix. In this case the memory requirement per node is

$$m(p) = n + n / p, \qquad (9)$$

and the memory efficiency is

$$\varepsilon_m = \frac{2}{1+p} \ . \qquad (10)$$

Again, due to the replication of data, in this case the argument vector **x**, the memory efficiency will be inversely proportional to p for large p, resulting in a bad utilization of the available distributed memory.

3 Memory Bounded Speedup

3.1 The Model of Sun and Ni

Recently Sun and Ni generalized the scaled speedup laws, by introducing the memory-bounded speedup model [6]. Both Amdahl's fixed-load speedup, Eq. (1), and Gustafson's fixed-time speedup, Eq. (2), are contained as a special case in the memory-bounded speedup model. First we will introduce the memory-bounded speedup model of Sun and Ni, and show their approach to include data replication into the model. Next we will apply the idea of memory efficiency in the model, and reproduce some results of Sun and Ni for the parallel matrix-matrix product. Finally, we apply the model to the matrix-vector product.

Define Δ as the computing capacity of a processor (expressed in e.g. Mflop/s), and W_i the amount of work in a parallel program with a degree of parallelism i. The total amount of work in the parallel program is

$$W = \sum_{i=1}^{m} W_i \ . \tag{11}$$

with m the maximum degree of parallelism in the program. If the workload W_i is executed on p processors, the execution time of the workload W_i is

$$t_i(p) = \frac{W_i}{i\Delta}\left\lceil \frac{i}{p} \right\rceil \ , \tag{12}$$

with $\lceil x \rceil$ the ceiling function of x. The workloads with $i < p$ model the load imbalance in the parallel program.

The execution time of the total workload W on p processors equals

$$T(p) = \sum_{i=1}^{m} t_i(p) = \sum_{i=1}^{m} \frac{W_i}{i\Delta}\left\lceil \frac{i}{p} \right\rceil \ . \tag{13}$$

We can now formulate the generalization of Amdahl's law, the fixed-load speedup factor, as

$$S_p = \frac{T(1)}{T(p)} = \frac{\sum_{i=1}^{m} W_i}{\sum_{i=1}^{m} \frac{W_i}{i}\left\lceil \frac{i}{p} \right\rceil} \ . \tag{14}$$

Here we have ignored communication latencies and other overheads. If $Q_p(W)$ is the total elapsed time which is due to overheads, and put $Q_1(W) = 0$, the fixed-load speedup becomes

$$S_p = \frac{T(1)}{T(p) + Q_p(W)} = \frac{\sum_{i=1}^{m} W_i}{\sum_{i=1}^{m} \frac{W_i}{i}\left\lceil \frac{i}{p} \right\rceil + Q_p(W)} \ . \tag{15}$$

Here it is assumed that the degree of parallelism is not affected by communication latencies.

We will now generalize the speedup models, by considering the only constraint in scaling the workload: available memory. The memory of each node of a real parallel computer is limited. Therefore, scaled speedup models have to consider memory limitations. Sun and Ni [6] proposed a memory-bounded speedup model. Their idea is to scale the problem to its maximum amount, thus fully utilizing both memory capacities and computational power of a parallel computer. We will first assume that the available memory is fully utilized.

If W is the workload of the unscaled problem and M the memory requirement connected to this workload, when define a function g such that

$$W = g(M) \ . \tag{16}$$

If $M = m_R$ (the memory per node) we can formally write for the maximum scaled workload W^*,

$$W^* = g(pm_R) = g(pg^{-1}(W)), \tag{17}$$

Sun and Ni's general speedup formula for memory-bounded speedup is

$$S_p^* = \frac{\sum_{i=1}^{m^*} W_i^*}{\sum_{i=1}^{m^*} \frac{W_i^*}{i}\left\lceil \frac{i}{p} \right\rceil + Q_p(W^*)} \ . \tag{18}$$

In the sequel we will restrict ourselves to the situation where $W_i = 0$ if $i \neq 1$ and $i \neq p$. Furthermore, we assume $Q_p(W) = 0$. This means that the assumption of a full utilization of the available memory is correct, since data replication is not necessary if the communication latencies are zero. If we insert this in Eq. (15) we reproduce Amdahl's law, Eq. (1):

$$S_p = \frac{W_1 + W_p}{W_1 + W_p/p} \ ,$$

if we realize that $\alpha = W_1 / (W_1 + W_p)$.

The computational work of the problem on a single node is $W_1 + W_p$, and for the scaled problem $W_1^* + W_p^*$. The memory bounded speedup now becomes

$$S_p^* = \frac{W_1^* + W_p^*}{W_1^* + W_p^*/p} \ . \tag{19}$$

The workload of the sequential part is assumed to be independent of both problem size and system size:

$$W_1 = W_1^* \ .$$

The scaled parallel workload needs some more consideration. The workload W and the memory requirement m for this workload are related by $W = g(m)$. If we assume that $g(x)$ is a semihomomorphism[3] and that the total memory capacity M of one processor is available for the workload W_p, we find

$$W_p^* = g(pm_R) = \bar{g}(p)g(m_R) = \bar{g}(p)W_p \ . \tag{20}$$

The function $\bar{g}(p)$ describes the increase in parallel workload after increasing the total amount of memory in the system with a factor p. The resulting memory-bounded scaled speedup is

$$S_p^* = \frac{W_1 + \bar{g}(p)W_p}{W_1 + \bar{g}(p)W_p/p} \ , \tag{21}$$

which is Eq. (16) of Sun and Ni [6].

Let us investigate three special cases.

- •1 $\bar{g}(p) = 1$. This corresponds to the fixed-problem size and equation 21 reduces to Amdahl's law.
- •2 $\bar{g}(p) = p$. The workload increases linearly with the available memory, keeping the total execution time fixed. This corresponds to Gustafson's law.
- •3 $\bar{g}(p) > p$. Here, the workload increases faster than the memory requirements of the parallel program, and the resulting speedup is larger than the fixed-time scaled speedup.

3 A function $g(x)$ is a semihomomorphism if $g(cx) = \bar{g}(c)g(x)$. For instance, the function $g(x) = ax^b$ is a semihomomorphism with $\bar{g}(x) = x^b$.

In order to find expressions for $\bar{g}(p)$ we usually have to perform an order of magnitude analysis, where we only keep the highest order terms. We will investigate the example programs of the previous section.

The matrix-matrix product requires to store three $n{\times}n$ matrices, therefore the memory requirement is $M = 3n^2$. The total work (assuming that it can be done in parallel) is $W_p = n^2(2n - 1) \sim 2n^3$ for large n. Therefore $W_p = 3^{-3/2} \times 2\ M^{3/2}$, and we immediately find $\bar{g}(p) = p^{3/2}$ and

$$W_p^* = p^{3/2} W_p \ . \tag{22}$$

This result can also be derived by putting $M^* = pM$. From this we find for n^*, which is the size of the scaled matrix, $n^* = p^{1/2}n$. Therefore,

$$W_p^* = 2(n^*)^3 = 2p^{3/2}n^3 = p^{3/2}W_p \ .$$

This is an example where $\bar{g}(p) > p$, and the memory-bounded speedup is even better than for fixed-time speedup.

Secondly consider the parallel matrix-vector product with the $n{\times}n$ matrix stored in main memory. The memory requirement is $M = n^2 + 2n$. The first term is the memory of the matrix, the second term is for the argument and result vector. The work is $W_p = n(2n - 1)$. If we assume that n is very large, $W_p = 2M$, and $\bar{g}(p) = p$. Therefore

$$W_p^* = pW_p \ ; \tag{23}$$

memory bounded speedup and fixed time speedup are equivalent in this case. Again, this result is easily derived by putting $M^* = pM$.

Finally, consider the case of the parallel matrix-vector product, where the matrix is not kept in memory. Assume that we can calculate the matrix elements, and that the amount of work to calculate one element equals e. In that case, the total amount of parallel work in the matrix vector product equals $W_p = n(2n - 1) + en^2$. Now we only have to store the argument and result vector, and therefore $M = 2n$. Assuming large n we find

$$W_p = \frac{2 + e}{4} M^2 \ ,$$

and $\bar{g}(p) = p^2$. In this special case the memory bounded work increases as the square of the number of processors,

$$W_p^* = p^2 W_p \tag{24}$$

We will postpone numerical calculations of the resulting scaled speedups until the next section.

3.2 Data Replication and Application of Memory Efficiency

We will now again focus our attention to Distributed Memory architectures. Here, as was pointed out by Sun and Ni [6], data in parallel programs usually has to be replicated. This is due to the fact that in many parallel calculations some data items are needed in all processors. Replication of this data in memory of all processors is more efficient than to keep it stored in memory of one processor and communicate it to other processors. However, due to this replication of data, the relation between the scaled and original workload in the memory bounded speedup model, as expressed in Eq. (20), no longer holds.

Sun and Ni circumvented this problem by defining the function

$$G(p) = \frac{W_p^*}{W_p} \ . \tag{25}$$

With this definition the memory bounded speedup becomes

$$S_p^* = \frac{W_1 + G(p)W_p}{W_1 + G(p)W_p/p + Q_p(W^*)} \ , \tag{26}$$

where we also included the overhead function Q.

We will derive an expression for $G(p)$ using the memory efficiency. From Eq. (4) and Eq. (16) we find that the total workload of a parallel program is limited by

$$W < g(M) = \bar{g}(\varepsilon_m p)\ w_{max}, \tag{27}$$

with w_{max} the maximum attainable workload of a program running on 1 processor. This limit on the workload of a program gives an upper bound to the scaling of workloads in scaled speedup models. In fact, we will show that the function $G(p)$, as defined by Eq. (25), equals $\bar{g}(\varepsilon_m p)$.

The memory requirement per node of the scaled workload in the memory bounded speedup model equals the memory requirement of the original workload:

$$m^*(p) = M .$$

Furthermore, the total memory requirement of the scaled workload equals the memory requirement per node of the scaled workload for $p = 1$:

$$m^*(1) = M^* .$$

If we substitute these two relations in the definition of the memory efficiency, Eq. (3), we find

$$M^* = \varepsilon_m^* pM \ . \tag{28}$$

Using Eq. (25) and Eq. (28) we can now derive an expression for $G(p)$:

$$G(p) = \frac{g(M^*)}{g(M)} = \bar{g}(\varepsilon_m^* p) \ . \tag{29}$$

and a final resulting expression for the memory bounded speedup

$$S_p^* = \frac{W_1 + \bar{g}(\varepsilon_m^* p)W_p}{W_1 + \bar{g}(\varepsilon_m^* p)W_p/p + Q_p(W^*)} \ . \tag{30}$$

Note that this memory bounded speedup is not necessarily the optimal scaled speedup, because the overhead function Q depends strongly on the details of the parallel program and the underlying parallel hardware. However, if we neglect the overhead Q, but still include the effects of data replication via ε_m, Eq. (30) results in a realistic upper bound of speedup. As will become clear, in many situations this upper bound is between the fixed-load upper bound of Amdahl and the fixed-time upper bound of Gustafson.

Let us now consider once more the examples of the matrix-matrix product and the matrix-vector product. As was shown in the previous section, for the matrix-matrix product we have $\bar{g}(p) = p^{3/2}$. Using Eq. (29) and Eq. (6) we immediately find the expression for $G(p)$:

$$G(p) = \left(\frac{3p}{2 + p}\right)^{3/2} \ . \tag{31}$$

which is equal to the result of Sun and Ni [6]. For large p, $G(p) = 3^{3/2}$, which is larger than the fixed-load speedup ($G = 1$), but much smaller than the fixed-time speedup ($G = p$). Due to data replication the memory capacity requirements increase much faster than the computational requirements.

For the matrix-vector product, with the matrix stored in memory, we have $\bar{g}(p) = p$ and $\varepsilon_m = 1$, resulting in

$$G(p) = p, \tag{32}$$

which is exactly the fixed time case.

Finally consider the matrix-vector product without storing the matrix. Remembering that in this case $\bar{g}(p) = p^2$, and using Eq. (10), the memory bounded scaling function for the out-of-core matrix-vector product is

$$G(p) = \left(\frac{2p}{1+p}\right)^2 . \tag{33}$$

Again, due to the replication of data, the memory capacity requirements grow faster than the computational requirements, and $G(p) \sim 4$ for large p. Memory bounded speedup will be slightly better than fixed-load speedup, but will not come close to the fixed-time speedup of Gustafson.

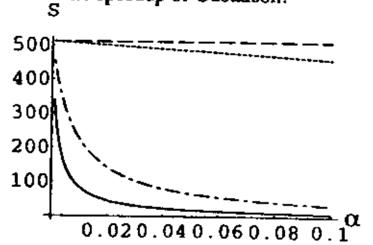

Figure 2: Scaled speedup S of the out-of-core matrix vector product as a function of the sequential portion of the program α, for 512 processors; the solid line is the fixed-load case (Amdahl), the dotted line is the fixed-time case (Gustafson), the dashed line is the memory bounded speedup without data replication, and the dashed dotted line is the memory bounded speedup with data replication.

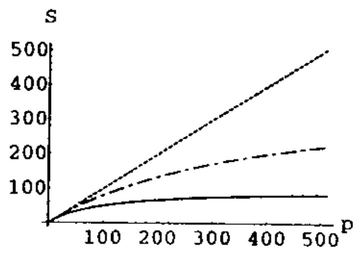

Figure 3: Scaled speedup S of the out-of-core matrix vector product as a function of the number of processors, for $\alpha = 0.01$; the solid line is the fixed-load case (Amdahl), the dotted line is the fixed-time case (Gustafson), and the dashed dotted line is the memory bounded speedup with data replication. The memory bounded speedup without data replication is indistinguishable from the fixed time case, and therefore not drawn.

Fig. (2) and (3) show the resulting scaled speedup for the out-of-core matrix-vector product as a function of α (the sequential fraction of the program) for $p = 512$, and as a function of p for $\alpha = 0.01$ respectively. The speedup was calculated for the fixed-load case, the fixed-time case and the memory-bounded case, with - and without data replication. Note that in Fig. (3) the memory bounded speedup without data replication is indistinguishable from the fixed time case, and therefore not drawn.

4 Discussion and Conclusions

We introduced the concept of memory efficiency, which is a measure of the amount of data replication in a parallel program. Memory efficiency expresses how well distributed memory is utilized if the number of nodes is increased in a parallel computer. If one realizes that memory usually is the most expensive part of a parallel computer, we can assume that memory efficiency of a parallel program should play a major role to assess the cost-effectiveness of parallel computing.

The fraction of replicated data plays the same role as the fraction of sequential work. However, in contrast with the sequential workload, increasing the total workload can lead to an increase in the amount of replicated data. This results in increasingly bad utilization of available memory, and a degradation of the scalability of the parallel program.

From Fig. (2) and (3) we can draw two important conclusions. Scaled speedup models are *the* solution to Amdahl's sequential bottle-neck, and memory bounded speedup models with $\varepsilon_m = 1$ will give almost ideal upper bounds to speedup. However, if the implementation of the parallel program is such that the memory efficiency is inversely proportional to p, e.g. due to the large global data buffers in the out-of-core matrix vector product, memory bounded speedup will only result in a modest improvement compared to Amdahl's law.

We have to be cautious if we interpret our theoretical results of memory-bounded speedup in terms of the daily practice of parallel computing. First, in the computation of the memory-bounded speedup in Fig. (2) and (3) it is assumed that the original, unscaled problem is the largest problem fitting in memory of one node. If we would start with a much smaller unscaled workload, we can increase the workload beyond the bounds given by the function $G(p)$, resulting in larger speedups. Memory-bounded speedup and memory efficiency become important when at some point the total available memory is full, and one wants to analyze how the parallel program behaves if the number of nodes is increased.

In conclusion, memory efficiency is a theoretical construction which is useful to assess the utilization of a parallel program of the available memory in distributed memory computers. Furthermore, it can be integrated with scalability theories, as was shown in the case of the memory-bounded speedup model, to account for the fact that data replication can result in bounds in the scaling of workloads.

5 References

1] X.H. Sun and D.T. Rover, "Scalability of Parallel Algorithm-Machine Combinations," IEEE Trans. Parallel Distrib. Systems. 5, 599-613 (1994).

2] V. Kumar and A. Gupta, "Analyzing Scalability of Parallel Algoritms and Architectures," J. Parallel Distrib. Computing 22, 379-391 (1994).

3] W. Ware, "The ultimate computer," IEEE Spectrum 9, 84-91 (1972).

4] G. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proc. AFIPS Conference*, 1967, pp. 483 - 485.

5] J.L. Gustafson, "Reevaluating Amdahl's Law," Communications of the ACM 31, 532 - 533 (1988).

6] X.H. Sun and L.M. Ni, "Scalable Problems and Memory-Bounded Speedup," J. Parallel Distrib. Comput. 19, 27-37 (1993).

7] G.C. Fox, S.W. Otto, and A.J.G. Hey, "Matrix algorithms on a hypercube I: Matrix multiplication," Parallel Computing 4, 17 - 31 (1987).