Parallel Computing: Performance Metrics and Models *

Sartaj Sahni and Venkat Thanvantri

Computer & Information Sciences Department

University of Florida

Gainesville, FL 32611, USA

sahni@cis.ufl.edu

Abstract

We review the many performance metrics that have been proposed for parallel systems (i.e., program – architecture combinations). These include the many variants of speedup, efficiency, and isoefficiency. We give reasons why none of these metrics should be used independent of the run time of the parallel system. The run time remains the dominant metric and the remaining metrics are important only to the extent they favor systems with better run time. We also lay out the minimum requirements that a model for parallel computers should meet before it can be considered acceptable. While many models have been proposed, none meets all of these requirements. The BSP and LogP models are considered and the importance of the specifics of the interconnect topology in developing good parallel algorithms pointed out.

*This work was supported in part by the Army Research Office under grant DAA H04-95-1-0111.

1 Introduction

For over three decades, researchers in the parallel processing area have procalimed that parallel computing and computers (throughout this paper, we use the term parallel computer to refer to a computer comprised of $P > 1$ homogeneous processors) are the wave of the future. The reason most commonly cited for the imminent arrival of this wave is: *We are quickly approaching the speed limit of serial computing. This limit needs to be overcome so that we can solve, in an acceptable amount of time, the many compute intensive applications that exist today as well as those that might crop up in the future. The only way to overcome the speed limit of serial computers is to harness the power of several serial computers to solve a single problem. I.e., parallel computing is the answer.* Recently, researchers have argued that the cost of building chip manufacturing facilities has become so high that it is unlikely new foundaries will be constructed in the future. So, the speed of microprocessors is limited by the technological capabilities of the foundaries currently complete or under construction. This will prevent us from computing at the speed of light using a single microprocessor.

As we all know, this much heralded future of parallel computing has yet to arrive. Some of the reasons for the limited acceptance of parallel computing and computers are:

1. *Lack of a unifying model.* The RAM model for serial computing made it possible for one to develop algorithms using an abstract model. This model had the property that algorithms that worked well on the model worked well on real serial computers regardless of the vendor and to a large extent regardless of variations in instruction set, cache size, number of registers, etc. Good performance on the RAM model translated into good performance on a real computer and vice versa. In the case of parallel computing, there is no simple and acceptably accurate model that enjoys this property.

2. *Lack of program portability.* With the widespread use of high-level languages such as FORTRAN, Pascal, C, etc., compilers for these languages became readily available for virtually all serial computers. As a result, programs written in a high-level language can be ported with relatively little effort from one

vendor's serial computer to that of another. By-and-large, programs ported in this fashion tended to exhibit similar performance across all serial computers. In the parallel world, however, a program written in parallel C (say) for parallel computer X cannot be ported with moderate effort to run with similar performance on parallel computer Y even though Y has a C compiler. This is due to the fact that to get good performance on computer X, it is necessary to write the program using knowledge of several architectural features of computer X. These features include: memory organization, interconnection topology, interconnection technology (i.e., point-to-point, electronic and optical buses, etc.), ratio of computational speed and communication bandwidth, number of available processors, ability of the processors to overlap computation with communication, etc. The lack of program portability means that each time one changes ones parallel computer (even staying with the same vendor or basic architecture), all the programs need to be rewritten (or at least retuned).

3. *Lack of suitable performance metrics.* In the serial world, performance is often captured by providing the time and memory / space requirements of a program. For practical purposes, the memory requirements of a program are important only to the extent that there be sufficient memory available to solve instances of the desired size. In most situations there is no benefit to using less primary memory than might be available on the target computer. Hence, with the assumption that there is "enough" primary memory available, only time remains as a performance metric. Asymptotic analysis techniques have made it possible to compare alternate solutions to the same problem without necessarily resorting to experiment. Experimental measurements of performance measures generally conform to the analytical expectations.

   In the parallel world, in addition to time and space, we have a myriad of measures (speedup, scaled speedup, efficiency, isoefficiency, serial fraction, etc.). Because of the lack of a unifying model, it isn't possible to talk about a parallel algorithm independent of the architecture for which it is designed. As a result, performance metrics for parallel algorithms are also tied to the target parallel architectures. So, one considers the algorithm and the parallel computer on which it is to run as a *system* and then measures performance of the system. As a result, we really cannot talk about the performance a parallel matrix multiplication algorithm (say). Rather, we need to talk about the performance of a parallel matrix multiplication system (i.e., algorithm–architecture combination) and there are at least as many of these as there are different parallel architectures.

4. *Use of slow processors.* Some manufacturers of parallel computers have used custom processors while others have used off-the-shelf microprocessors. Those that have relied on custom technology have generally found it impossible to keep pace with performance improvements in off-the-shelf microprocessors. As a result, these manufacturers have found that shortly after introduction (and sometimes even before introduction), the performance of an individual processor of their parallel computer was significantly inferior to that of a mid-to-high-end workstation. With manufacturers that have relied on off-the-shelf microprocessors, there is generally a two-to-three year gap between the introduction of a new microprocssor and its application in a parallel computer. Once again, an individual processor of the parallel computer is found to perform much less effectively than a fast serial workstation.

   The fact that many parallel commercial parallel computers have been built with serial processors that are slower than those used in the fastest PCs and workstations has made it difficult to show spectacular performance gains over high-end state-of-the-art serial computers. In the absence of anomalous behaviour, a 50 processor parallel computer built with processors that are only one fifth as fast as the fastest serial computer can, at best, show a performance gain of 10. When communication and algorithm overheads are factored in, one can expect only a much more modest performance gain.

While the first three reasons are technological, the last is largely economic. In Section 2, we review the many performance metrics proposed for parallel systems. This review sheds some light on why so many metrics exist and also points out the similarities among some of these. We also point out that while the focus of recent research has been on optimizing these metrics, the run time should remain the primary metric. By using almost any other metric as the primary metric, we run the risk of favoring a parallel algorithm that is always slower over one that is always faster. In Section 3, we list criteria that a unifying model for parallel computing should satisfy. Examining the architectural disparities among the many commercial and proposed parallel computers, we conclude that it is highly improbable that one can formulate a suitably simple and accurate model that will satisfy the desired criteria. Two popular models for a subset of commercial architectures are examined. These models are the bulk-synchronous parallel model and the LogP model. Both of these are intended to be used as models of asynchronous distributed memory parallel computers. We point out that while these represent a major advance in modeling a popular class of parallel computers, neither of these satisfy all of the desired properties for a parallel computing model. So, these models should be used with caution when developing algorithms for commercial asynchronous distributed memory parallel computers.

2 Performance Metrics

2.1 Speedup

In parallel computing, as in serial computing, time and memory are the dominant performance metrics. Between alternate methods that use differing amounts of memory, we would prefer the faster method provided enough memory is available to run both methods. That is, there is no advantage to using less memory than might be made available to the application unless the use of less memory results in a reduction in execution time. So, the execution time or time complexity of a parallel program remains an important metric. In serial computing, an asymptotic time complexity analysis can be done using the RAM model. However, in the case of parallel asymptotic time complexity analysis, it is not enough to know that the P processors of the parallel computer can each be modeled as a RAM. We need to know additional architectural details such as the interconnection topology and memory access properties. As a result, when we talk about the complexity of a parallel algorithm we are really talking about the analysis of the algorithm with respect to a particular parallel computer architecture. The combination of the algorithm and the architecture defines a parallel system. So, we talk about the time complexity of the system.

Performance measurements in the parallel domain have been made more complex by our desire to know "how much faster are we running our application on a parallel computer?". That is, what benefit are we deriving from the use of parallelism or how much is the speedup that results from the use of parallelism? While there is general agreement that speedup is the ratio

$$\frac{\text{serial execution time}}{\text{parallel execution time}}$$

there is diversity in the definitions of serial and parallel execution times. This diversity results in at least five different definitions of speedup.

1. In *relative* [40], [41], [42], the serial time used is the execution time of the parallel program when run on a single processor of that parallel computer. So the relative speedup obtained by the parallel program, Q, when solving an instance I of size n using P processors is:

   $$\mathit{RelativeSpeedup}(I,P) = \frac{\text{time to solve } I \text{ using program } Q \text{ and 1 processor}}{\text{time to solve } I \text{ using program } Q \text{ and } P \text{ processors}}$$

   The relative speedup of a parallel system is not a fixed number. Rather, it depends on the characteristics of the instance I being solved as well as on the size P of the parallel computer being used. Typically, for instance characteristics we use simple quantities like the instance size (i.e., number of inputs and / or outputs). These generally do not uniquely determine the execution time. For example, the time to sort n numbers using most of the popular serial sorting methods depends not only on n but also on the initial order of the n numbers being sorted. Consequently, the relative speedup of a parallel system isn't uniquely determined by the instance size n and the parallel computer size P. So, we may further qualify relative speedup as maximum speedup, average or expected speedup, and minimum speedup. For any combination of instance and parallel computer sizes, the maximum relative speedup is defined as

   $$\mathit{MaximumRelativeSpeedup}(n,P) = \max_{|I|=n}\{\mathit{RelativeSpeedup}(I,P)\}$$

where $|I|$ is the size of instance I. Average and minimum relative speedup are defined similarly.

For many of the examples used in this paper, run time is uniquely determined by the number of inputs in an instance and in these cases, we do not make the distinction between maximum, average, and minimum speedups. For these examples, we may use the notation $RelativeSpeedup(n, P)$ to denote the relative speedup obtained on instances of size n when P processors are used.

2. In [18], [21], and [34], for example, parallel execution time is compared with the time needed by the fastest serial algorithm / program for the application. The fastest algorithm's execution time on a single processor of the parallel computer is used. Since for many applications, we may not know the fastest algorithm and since for some applications no one algorithm might be fastest for all instances, the run time of the sequential algorithm that is used in "practice" is used instead of the run time of the fastest algorithm. The resulting speedup is called the *real speedup.*

$$RealSpeedup(I, P) = \frac{\text{time to solve } I \text{ using best serial program and 1 processor}}{\text{time to solve } I \text{ using program } Q \text{ and } P \text{ processors}}$$

   We may define $MaximumRealSpeedup(n, P)$, $RealSpeedup(n, P)$, etc. in the same way as we did for relative speedup.

3. In yet another definition of speedup, the parallel execution time is compared with that of the fastest sequential algorithm run on the fastest serial computer. As in the case of real speedup, in reality, the comparison is done using the serial algorithm that would be used in "practice". The term *absolute speedup* is used for this measure [33], [40], [41], [42].

$$AbsoluteSpeedup(I, P) = \frac{\text{time to solve } I \text{ using best serial program and 1 fastest processor}}{\text{time to solve } I \text{ using program } Q \text{ and } P \text{ processors}}$$

   The definition of absolute speedup may be extended to $MaximumAbsoluteSpeedup(n, P)$, $AbsoluteSpeedup(n, P)$, etc.

4. Let $t_{serial}^{best}(n)$ be the asymptotic complexity of the best serial algorithm for the problem and let $t_{parallel}^{Q}(n)$ be the asymptotic complexity of the parallel algorithm Q under the assumption that the parallel computer has available to

it as many processors as it can gainfully use. The *asymptotic real speedup* (see [24], for example) is defined as

$$AsymptoticRealSpeedup(n) = \frac{t^{best}_{serial}(n)}{t^{Q}_{parallel}(n)}$$

For problems such as sorting where the asymptotic complexity isn't uniquely characterized by the instance size n, we use the worst case complexity.

5. *Asymptotic relative speedup* (see [32], for example) differs from asymptotic real speedup in that for the serial complexity, we use the asymtotic time complexity of the parallel algorithm when run on a single processor.

Notice that unlike the remaining three speedup measures, asymptotic (real, relative) speedup is not a function of the number of processors available in the parallel system. This is because in asymptotic speedup, we assume that the parallel system has an unbounded number of processors.

When computing relative, real, or absolute speedup, we could use analytical time complexities as is done in [9], [20], [22], [24], [50], [51] (for example) or we could use actual measured run times as is done in [40] [41], [50], [51] (for example). When the former is done, we obtain the *analytical speedup* of the parallel system. In the latter case, we obtain the *measured speedup* of the system. The meaning of the terms analytical relative speedup, analytical real speedup, measured real speedup, etc. should be apparent. Since analytical run time measurements are usually accurate only to within a constant factor, the analytical run time of the "best" serial algorithm is the same on all serial computers modeled by RAMs (within a constant factor). So, usually there is no difference between analytical real speedup and analytical absolute speedup.

2.1.1 Asymptotic Speedup

Consider the matrix multiplication algorithm of [9] which multiplies two $n \times n$ matrices on an $n^3/\log n$ processor hypercube in $\Theta(\log n)$ time. Since this is the fastest the algorithm can run even if more processors are made available, $t^{Q}_{parallel}(n) = \log n$. What is its asymptotic real speedup? This depends on the complexity of the best serial algorithm and this complexity changes with time as faster matrix multiplication algorithms are discovered. Hence, except in the case of problems for which asymptotically optimal algorithms are known, asymptotic speedup is not time invariant. If $\Theta(n^\alpha)$ is the complexity of the fastest serial matrix multiplication algorithm currently known, then

$$AsymptoticRealSpeedup(n) = \Theta(n^\alpha / \log n)$$

Unfortunately, the goodness of our parallel multiplication algorithm / system (as measured by its asymptotic real speedup) declines as the years go by and asymptotically superior serial algorithms are discovered. This decline in goodness has nothing to do with deterioration in the parallel algorithm with respect to its execution time and memory requirements.

When the parallel matrix multiplication algorithm of [9] is run on a single processor, its complexity is $\Theta(n^3)$. So, the asymptotic relative speedup of the parallel matrix multiplication system of [9] is $\Theta(n^3/\log n)$.

2.1.2 Absolute Speedup

Since we have equated analytical absolute speedup to analytical real speedup, in this section, we are concerned only with measured absolute speedup. In practice, one would expect this to be a really important measure as it tells us how much faster I can solve my problem instance on the target parallel computer versus the fastest serial alternative. Unfortunately, very few experimental studies of speedup report this quantity. Besides difficulties associated with getting access to the fastest serial alternative, this alternative changes with time. This may happen because a faster serial algorithm is developed or because a faster serial computer is manufactured. Certainly in the time elapsed between the conducting of experiments and the publishing of the results, the speed of the fastest serial computer and hence the absolute speedup would have changed!

Since many parallel computers are built using processors that are slow relative to the fastest serial processors available at the time, these parallel computers exhibit absolute speedup less than one even when P is modestly large (say 100). For example, Ranka and Sahni [35] report that the time needed by a 64 processor nCube1 computer to perform image-template matching on a 256×256 image using a 16×16

template was approximately six times that required by a single processor Cray2. This is despite the fact that the nCube1 program achieved a relative speedup of almost 58. (Note, however, that a 64 processor nCube1 provided a cost-performance advantage over the Cray2 as its cost in 1990 was less than one-tenth that of a single processor Cray2.)

2.1.3 Real Speedup

Analytical real speedup is generally easy to determine as one need not write a program and measure execution time. There are two approaches to computing the analytical real speedup of a parallel system. In one, the asymptotic complexities of the serial and parallel algorithms are used and in the other, we use the number of operations or workload. Generally, we regard workload (total amount of computation done by the program) and run time as related by the speed of the computer. At times, we will need to make the distinction between workload and time because this assumed relationship between the two may not hold. For example, the effective speed of the computer may vary with workload as larger workloads may require more memory and may eventually require the use of slower secondary memory. Furthermore, as total workload changes, its mix may change causing a change in effective speed.

As an example of using asymtotic complexities, we note that two $n \times n$ matrices can be added in $\Theta(n^2/P)$ time using a shared memory computer with $P \leq n^2$ processors. The fastest serial algorithm to add the two matrices has complexity $\Theta(n^2)$. So, the analytical real speedup of the parallel shared memory algorithm is

$$AnalyticalRealSpeedup(n, P) = \Theta(P)$$

Notice the difference between this and asymptotic speedup. In the latter, the asymptotic complexity is computed assuming an unbounded supply of processors. As an example of using operation counts or workload, consider the problem of reading in two $n \times n$ matrices from a disk, adding them, and writing the result back to disk. Suppose that the time to read / write a matrix is n^2 units and that the time to add them using a serial algorithm is n^2/S where S is the speed of the processor. Further, suppose that the parallel addition algorithm, because of communication

size n	Number of Processors P					
	2	4	8	16	32	64
16	0.90	0.84				
32	1.06	1.12	1.04			
64	1.22	1.44	1.60	1.60		
512	1.46	2.36	3.76	5.44	6.72	7.04

Table 1: Connected components average real speedup on an nCube1 hypercube

and other overheads, takes $2n^2/(SP)$ time to add the matrices using P processors of speed S. The serial time is $3n^2 + n^2/S$ and the parallel time is $3n^2 + 2n^2/(SP)$. The analytical real speedup is

$$(3 + 1/S)/(3 + 2/(SP)) \tag{1}$$

While analytical real speedup can be computed without expending the effort that goes into actually programming an algorithm and measuring its execution time, analytical speedup isn't very useful for the practitioner who is very concerned about the constant factors associated with the speedup. The difference between a speedup of $P/8$ and $P/4$ (for our matrix addition example) is very important in practice. Even though in our matrix multiplication example, we performed a more accurate analysis than when asymptotic complexities are used, the predicted speedup may differ from the actually observed speedup as the analysis cannot account for all overheads encountered in an actual execution.

To obtain the measured real speedup of a parallel system, we need to program and run both the "best" serial algorithm and the parallel algorithm. The measured speedup varies with both the problem instance and the number of processors. Table 1 shows the measured average real speedups of the stripes partitioning method of [50] to compute the connected components of a dense graph. The algorithm was run on an nCube1 hypercube computer. The number of vertices in the graph is denoted by n and the number of processors is denoted by P. This table has been computed using data provided in [50].

Because parallel programs often contain overheads that are not present in the competing serial algorithm, parallel programs often exhibit no speedup when the number of processors and / or the instance size is small. For example, from Table 1, we see that the parallel connected components algorithm of [50] takes more time on graphs with 16 vertices when run on an nCube1 hypercube with $P > 1$ processors than taken by the competing "best" serial algorithm running on a single hypercube processor (i.e., speedup is less than one).

The communication and other overheads in a parallel program often increase as the number of processors is increased. As a result, if we fix the problem size and increase the number of processors, the speedup often increases for a while and then begins to decrease. So, there is often an optimal value of P beyond which the run time actually increases. This optimal value of P is a function of the problem size n. Examining Table 1, we see that when $n = 16$, the speedup is less with four processors than with two; when $n = 32$, the speedup is less with eight processors than with four; and (while not shown in the table) when $n = 64$, the speedup is less with 32 processors than it is with eight. Notice that if speedup declines as P increases, then run time increases as P increases. [12], [29], and [45] develop methods to determine the optimal number of processors to use.

Measuring real speedup on a distributed memory parallel computer is often a problem as there isn't sufficient memory associated with a single processor to execute the serial program on that processor while using instances that are large. This difficulty is less acute when one uses absolute speedup as the fastest serial computers usually have larger amounts of memory available.

Since in measured real speedup, all experiments (serial and parallel) are performed on the same computer, we eliminate the variance in absolute speedup that is due to changes in the fastest serial computer. However, real speedup is still subject to the influence of changes in the "best" serial algorithm. So, as better serial algorithms are developed, the real speedup obtained from the parallel algorithm decreases.

Measured real speedup more accurately reflects the benefits accruing from parallelism as it factors out the effects of speed difference between the individual processors of the parallel computer and that of the fastest serial computer. *However, when real speedup is used as a performance measure independent of actual parallel execution time, then it favors slow processors over fast ones.* Consider the analytical speedup formula of Equation 1. For $P > 2$, the speedup increases as S decreases. For instance, when $S = P = 10$, the speedup is 1.026 and when $S = 1$ and $P = 10$, the speedup is 1.25. Hence, one can observe larger speedups by simply using slower processors! The actual parallel run time is however, 3.02 on the faster parallel computer and 3.2 on the slower one. So, increasing real speedup should be secondary to reducing the actual execution time.

2.1.4 Relative Speedup

Since the measurement of real speedup requires us to program not just the parallel algorithm but also the "best" serial algorithm, one is tempted to look at an easier way to report the speedup of ones parallel algorithm. In particular, one might avoid using the "best" serial algorithm and use the single processor execution time of the parallel algorithm itself (provided the parallel algorithm has been written with P a parameter that can be set to 1) or use the serial algorithm on which the parallel algorithm is "based". This is what is done when measuring relative speedup. Relative speedup, like real speedup, may be determined analytically or experimentally.

Relative speedup suffers from the same deficiencies as does real speedup. I.e., because of memory limitations one is often unable to obtain the execution time on a single processor and relative speedup is higher when the processors are slow than when they are fast [41]. In addition [41], it favors code that is inefficient when run on a single processor. This does not happen in the case of real speedup as for this measure, the single processor code used is the "best" available.

Suppose we have an inherently sequential problem that is optimally solved in $\Theta(n)$ serial time using the procedure $SequentialCode(n)$. A very inefficient serial code for this problem is:

```
for $i = 1$ to 1000000 do $SequentialCode(n)$;
```

A simple P, processor parallelization of this code is (assume P divides 1000000):

```
for $i = 1$ to $P$ pardo
for $i = 1$ to $1000000/P$ do $SequentialCode(n)$;
```

Both the serial and parallel codes execute procedure $SequentialCode(n)$ 1,000,000 times even though one execution would have sufficed. The relative speedup is P.

However, the real speedup is less than one for $P < 1000000$. So, it is trivially possible to obtain good relative speedups by simply starting with bad but easily parallelized serial code. The resulting parallel code is, of course, of little practical value.

As another example, consider the problem of multiplying two $n \times n$ matrices on a parallel computer system comprised of a host to which P parallel processors are attached as a peripheral. Suppose that the time to transfer the two matrices to the peripheral parallel processors and to receive the results back is $3n^2$ and the time to multiply the matrices on the parallel processors is $n^3/(SP)$ (assume $P \leq n^2$) where S is the speed of one of the parallel processors. The relative speedup is $(n^3/S)/(3n^2 + n^3/(SP))$ under the assumption that if we were to use a serial computer comprised of one of the parallel processors, then there would be no need to transfer the matrices to and from a peripheral. Simplifying this formula yields the formula $n/(3S + n/P)$ which implies that for fixed n and P, speedup increases as S decreases. Or, the speedup is more when we use slower processors. Suppose we change the program to one that is less efficient, i.e., one that does more work. For our matrix multiplication example, this could be done by adding redundant work like recomputing the matrix product several times. The execution time of the new code is kn^3/P where k is the degree of inefficiency. The relative speedup is now $(kn^3/S)/(3n^2 + kn^3/(SP)) = n/(3S/k + n/P)$ which is an increasing function of k. When $S = 1$, $n = 100$, and $P = 10$, the relative speedup is $100k/(3 + 10k)$. For $k = 1$, this is 7.69 and for $k = 10$, it is 9.71.

As in the case of real speedup, it is hazardous to use relative speedup as a parallel performance metric independent of actual run time. The potential for erroneous conclusions are even greater now as the speedup is not measured with respect to the best serial algorithm. So, it possible that a parallel algorithm that exhibits a relative speedup of 10 when solving a problem of size 1000 using 20 processors actually takes more time than the best serial algorithm running on a single processor of the same parallel computer.

2.2 Limits To Speedup

Suppose we want to add two 100×100 matrices manually. The matrices are initially written on a long wall and the result is also to be written on that wall. Suppose that if I were to do it by myself, it would take me a day. Using a friend, I could it do it in a little over half a day as it would take me a little bit of time to tell my friend which half he / she should work on and then it would take us half a day each to do the actual addition. The speedup is almost 2. If 10,000 people were standing as in a 100×100 matrix, each could be assigned a unique pair of matrix elements to add and the addition task could be accomplished in a liitle over one-ten-thousandth of a day. The speedup is almost 10,000. With a million people, the job cannot be done any faster as there isn't enough work to go around. In fact, because of the ensuing pandemonium, it may actually take more time resulting in a smaller speedup.

Since each problem instance may be assumed to be solvable by a finite amount of work, it follows that by increasing the number of processors indefinitely, we will reach a point when there isn't any work to be distributed to the newly added processors and no further speedup is possible. So, for any given instance of a problem, there is a limit to the speedup that is attainable. However, depending upon the amount of work available, the attainable speedup may be very large and we may be able to gainfully employ a very large number of processors. To attain ever increasing amounts of speedup, we must solve larger and larger instances (i.e., the workload represented by the instances must continue to increase). In our above matrix addition example, when adding 100×100 matrices we can continue to get increasing speedup up to 10,000 processors (i.e., people). Beyond this, we must move to a larger instance if we are to see more speedup. For 1000×1000 matrices, the speedup will continue to increase up to 1,000,000 processors.

The asymptotic real speedup of the matrix multiplication algorithm of [9] is $\Theta(n^3/\log n)$. As $n \to \infty$, speedup $\to \infty$. So, speedup is unbounded. Of course, as we increase n, the algorithm of [9] requires an increasing number of processors. The parallel computing literature is rich in examples of parallel algorithms that exhibit unbounded parallelism. Each, of course, requires that workload and number of processors increase indefinitely.

Perhaps the oldest and most quoted observation about limits to attainable speedup is Amdahl's Law [4]. This law makes the observation that if we are to solve a problem which contains both a serial component (i.e., portion of computing that cannot be parallelized) and a parallel component (i.e., portion that can be run on a P processor parallel computer with speedup P) then the observed speedup will be

$$\frac{s+p}{s+p/P}$$

where s and p are respectively the times needed to execute the serial and parallel components on a single processor (in general, s and p will be functions of some characteristics of the instance being solved, e.g., the size of the instance). The definition of speedup used by Amdahl corresponds to that of relative speedup.

The validity of Amdahl's law follows from the definition of relative speedup. The implications of Amdahl's law are different for different problems. Let us reconsider the matrix addition example of Section 2.1.3. The disk I / O represents the serial component and takes $3n^2$ time. Assuming no overheads, the parallel component which is the matrix addition takes $n^2/(SP)$ when P processors of speed S are available. The relative speedup is $(3+1/S)/(3+2/(SP))$ which approaches $1+1/(3S)$ as $P \to \infty$. So, regardless of how many processors we use, we cannot attain a speedup of even two when $S \geq 1$! This is because the serial component remains a constant fraction of the total serial work even as the workload increases.

When we are dealing with problems for which $c \leq s/(s+p) \leq 1$ for some constant c, the attainable speedup is bounded (even in the face of increasing workloads and number of processors). From Amdahl's law, we get

$$\text{speedup} = \frac{s+p}{s+p/P} = \frac{1}{\frac{s}{s+p}+\frac{p}{P(s+p)}} \leq \frac{1}{c+\frac{p}{P(s+p)}} \leq \frac{1}{c}$$

So, if 99% of the workload is parallelizable, then the maximum speedup attainable is 100; if 99.9% is parallelizable, the maximum attainable speedup is 1000; and if only 99.99% is parallelizable, the speedup cannot exceed 10,000. This reasoning was used by Amdahl to conclude that parallel computers could not deliver significant speedup in practice as to do this the serial component would have to be very very small.

In our earlier examples of unbounded parallelism, the ratio $\frac{s}{s+p} \to 0$ as we increase the workload and the bounded speedup implication of Amdahl's law doesn't apply. Consider our example of matrix multiplication using the host model (Section 2.1.4). Here, the serial component is the time needed to send the data and receive the results from the parallel processor ensemble. This time is $3n^2$. The parallel component is n^3. The ratio $\frac{s}{s+p}$ is $\frac{3}{n}$ which $\rightarrow 0$ as $n \rightarrow \infty$. Hence, the attainable speedup is unbounded. In fact, in Section 2.1.4, we showed that the realtive speedup is $(n^3/S)/(3n^2 + n^3/(SP))$ which is $n/(3 + \frac{n}{P})$ for $S = 1$. If we keep $n = P$ as we increase the workload and the number of processors, then the relative speedup becomes $n/4$ which $\rightarrow \infty$ as n is increased.

As the preceding examples show, there is no inconsistency between Amdahl's law and the attainability of unbounded speedup. Amdahl simply applied the relative speedup formula to the case when $0 < c \leq s/(s+p) \leq 1$ to demonstrate that under these circumstances, speedup is bounded. However, there are many problems for which the ratio $s/(s+p)$ diminishes to zero.

If we limit the question of maximum attainable speedup to a particular instance of a problem, then speedup is limited for those instances that have a fixed workload associated with them. Two factors contribute to limiting the attainable speedup.

1. *The total workload.* If the instance represents a total workload of u units and each processor does one unit of work in unit time, then at most u processors can be gainfully used and the parallel execution time can be as low as one. The serial execution time is u. So, speedup cannot exceed u.

2. *The serial component.* If s of the u workload units cannot be parallelized, the parallel run time cannot be reduced below $s + 1$ (Note, we are dealing with work units and assuming that one work unit can be executed in unit time using a single processor. As a result, the parallel component will take at least one unit of time when run on many processors). So, the speedup cannot exceed $u/(s + 1)$.

In several application areas, the workload associated with an instance can be changed so as to meet certain computational and accuracy requirements. For example, if we wish to predict the temperature in Gainesville one hour from now using a finite grid method, the grid resolution, number of time steps, etc. determine both the computational load and the accuracy of the predicted temperature.

When solving combinatorial problems using heuristics, the quality of the computed solution can often be improved by increasing the computational resources available to the heuristic. By changing the workload associated with an instance, we may also change (hopefully reduce) the fraction of the workload that cannot be parallelized. A generalization of Amdahl's Law to the case when the serial and parallel workloads (and hence the times spent on the serial and parallelizable parts of the computation) may be varied with variations in the number of processors is considered in [10]. In flexible workload environments, neither of the two factors that limit speedup may apply. For applications with flexible workload per instance, different speedup measures have been proposed.

2.3 Speedup Measures For Flexible Workload Instances

Recognizing that the speedup limits attributed to Amdahl's law apply only when $0 < c \leq s/(s+p) \leq 1$ and that in certain application domains users can increase the workload associated with an instance as more computational resources become available, different speedup formulations have been proposed. For applications in which the parallel component of the workload can be scaled linearly in P while keeping the serial component fixed, Gustafson [15] has proposed a speedup measure called *scaled speedup*. Gustafson's formula for scaled relative speedup is $(s+pP)/(s+p)$. In this, the serial execution time, $s + pP$, increases as we increase the number of processors but the parallel execution time, $s + p$, remains the same. As cited by Gustafson [15], this assumption is valid for grid problems where the grid resolution, number of iterations, etc. can be adjusted to increase the accuracy of the computation. Users are less interested in reducing the execution time than they are in improved accuracy. So, as the number of processors increases, finer grids and more time steps can be used to increase accuracy while keeping the parallel execution time unchanged. This increases the parallel workload while keeping the serial workload fixed. Notice that scaled speedup is simply the relative speedup of the scaled instance. Zhou [52] has observed that if s is held constant while $s + p \to \infty$ then the speedup increases linearly in P. The assumption that s remains the same as workload is scaled with P does not hold in all situations. The case when s increases linearly with P is considered in [10] and the case when s grows logarithmically in P is considered in

[16].

Sun and Ni [40], [42] have generalized Gustafson's scaled speedup to *fixed-time relative speedup* and proposed another speedup model called *memory-bounded relative speedup.* In fixed-time relative speedup, instance parameters such as grid resolution, number of time steps, number of starting points tried by an iterative improvement heuristic, number of temperature changes used by a simulated annealing method, etc. are adjusted so that the total time spent by the parallel computer when solving the adjusted instance is the same as that spent by the serial computer on the non-adjusted instance.

Let I denote the problem instance to be solved. Let T (T') be the time needed to solve the unadjusted (adjusted) instance using a single processor of the parallel computer and the given parallel program with P set to one. The fixed-time relative speedup is [40], [42]:

$$FixedTimeRelativeSpeedup(I,P) \quad = \tfrac{T'}{T} \quad = \frac{\text{serial time of adjusted instance}}{\text{parallel time of adjusted instance}}$$

$=$ relative speedup of adjusted instance

Fixed-time real and absolute speedup may be defined similarly. For fixed-time real speedup, we first determine the time T taken by the "best" serial program to solve the unadjusted instance using a single processor of the parallel computer. The instance parameters are adjusted so that the adjusted instance runs in this much time on the parallel computer. The time, T', needed to solve the adjusted instance using the "best" serial program is determined. The fixed-time real speedup for the instance is T'/T. For fixed-time absolute speedup, all serial run times are determined using the "fastest" serial computer and the "best" serial program.

In [40] and [42], fixed-time relative speedup is also stated in terms of workload. Let W be the workload of the non-adjusted instance (W is generally the maximum workload that can be executed on a serial computer in the amount of time T, one is willing to wait for the results.). This workload may be decomposed into components W_i by examining the execution profile of the parallel program under the assumptions (a) $P = \infty$, (b) synchronized execution steps all of the same duration and in each of which each active processor performs one operation, and (c) at each step, the parallel program executes, in parallel, all operations that are ready for execution.

W_i is the total work (i.e., number of steps in which i processors are active times i) that is done using i processors. It is easy to see that W_i is divisible by i and that $W = \sum_{i=1}^{m} W_i$ where m is the largest i for which W_i is non-zero. We shall say that W_i is the work that can be done with maximum parallelism i and that m is the maximum degree of parallelism [40], [42].

Let W' be the workload when the parameters to instance I are adjusted so that the run time on a P processor parallel computer is the same as the serial run time T for the unadjusted instance. Let W_i' be the component of the workload W' that can be done with maximum parallelism i and let m' be the maximum degree of parallelism of the adjusted instance. The time, T', required to solve the adjusted scaled instance using one processor of speed S (S is now the number of steps per unit of time) is

$$T' = W'/S = \sum_{i=1}^{m'} W_i'/S$$

When we attempt to solve the adjusted instance using P processors, the time needed for workload W_i' depends on the precedence relations among the operations that constitute W_i'. As a result, this time can vary from $\lceil W_i'/P \rceil /S$ to $W'/(iS) * \lceil i/P \rceil$. Sun and Ni ([40], [42]) assume the parallel time is the maximum possible. With this assumption, we get

$$T = W/S = \sum_{i=1}^{m'} \frac{W_i'}{iS} \lceil \frac{i}{P} \rceil + \mathit{overhead}(I,P)$$

where $\mathit{overhead}(I,P)$ is the overhead time introduced by the parallelization (in particular, this includes the communication overhead). So,

$$\mathit{FixedTimeRelativeSpeedup}(I,P) \quad = \tfrac{T'}{T} = \tfrac{W'}{W} = \frac{\sum_{i=1}^{m'} W_i'}{\sum_{i=1}^{m'} \frac{W_i'}{i} \lceil \frac{i}{P} \rceil + S * \mathit{overhead}(I,P)}$$

In memory-bounded relative speedup, the instance parameters are adjusted so that the adjusted instance when solved using P processors uses up as much memory per processor as used by the unadjusted instance when solved with a single processor. So, the aggregate memory usage by the adjusted instance may be up to P times that used by the non-adjusted instance. Let W_i^* represent the workload of the adjusted instance that can be done with maximum parallelism i and let m^* be the maximum degree of parallelism of the adjusted instance. Let $T^* = W^*/S$ be the time needed to solve the adjusted instance serially and let T be the time needed to solve the adjusted instance using P processors. The memory-bounded relative speedup for instance I is [40], [42]

$$MemoryBoundedRelativeSpeedup(I,P) \quad = \frac{T^*}{T} = \frac{W^*}{W} = \frac{\sum_{i=1}^{m^*} W_i^*}{\sum_{i=1}^{m^*} \frac{W_i^*}{i} \lceil \frac{i}{P} \rceil + S * overhead(I,P)}$$

Like fixed-time relative speedup, memory-bounded relative speedup for the problem instance I is simply the traditional relative speedup for the respective adjusted instance. Since the adjusted instance generally has a larger workload associated with it and the serial component of this workload is generally a smaller fraction of the total workload, the speedup for the adjusted instance is generally larger than for the original unadjusted instance. In memory-bounded real speedup, T' is the time needed to solve the adjusted instance using the "best" serial program and a single processor of the parallel computer while in memory-bounded absolute speedup, T' is the time needed to solve the adjusted instance using the "best" serial program and the "fastest" serial computer.

Sun and Gustafson [41] have provided a more general formulation of fixed-time and memory-bounded speedup that uses a different cost factor for the different kinds of work that might be done when solving instance I. They also show that under certain conditions, relative speedup favors slower parallel processors as well as inefficient code in the sense that these result in higher speedup values.

Sun and Gustafson [41] introduce two new speedup measures. One of these is called *generalized speedup*. It is defined as the ratio of parallel execution speed to sequential execution speed (speed may be measured, for example, as the average number of instructions or floating point operations executed per second). They observe that when the cost factor is the same for all kinds of work, generalized speedup equals relative speedup. This is also true when cost factors are different and the instance is not scaled. Define r_∞ to be the maximum speed attained by a single processor execution of the parallel algorithm as the workload is increased [43]. When using generalized speedup, we can avoid running large instances on a single processor (something that might be impossible because of memory size limitations and required run time) by using r_∞ as the sequential execution speed.

The other measure introduced in [41] is *sizeup*. For this, the instance being solved is adjusted as for fixed-time speedup. The sizeup is the ratio of the serial work represented by the adjusted instance to the serial work represented by the unadjusted instance, i.e., sizeup $= W'/W$. When the cost factor for different kinds of work is the same, work and time are related by a constant and sizeup is the same as fixed-time speedup.

2.4 Anomalous Speedup

In the literature (see [18], [24], for example), one can find many claims that real speedup cannot exceed P, the number of processors. This claim can be "proved" by showing how a single processor can simulate any parallel algorithm that runs in $t(I)$ time by a single processor in $Pt(I)$ time (by executing the steps of the P processors serially in round-robin fashion). Hence, there is a sequential algorithm with execution time $Pt(I)$. The best sequential algorithm for the problem therefore takes no more than this much time and so the real speedup is no more than P. There are at least two flaws in this argument:

1. There exist problems for which there are no sequential algorithms that are "best" on all instances. As a result, real speedup is computed relative to the algorithm that would be used in practice. With respect to this algorithm, it is entirely possible that other sequential algorithms are faster on certain instances and slower on others. The parallel version of these could therefore yield speedup greater than P on some instances and speedup less than one on others.

2. The simulation of the parallel algorithm by a single processor incurs some overhead. There are several possible sources for this overhead:

   (a) The parallel computer with P processors may have more aggregate memory than the single processor. As a result, the simulation may need to use secondary storage. Alternatively, even though only one processor is used for the computation, we may need to use the memory of the remaining processors for the data. This results in expensive fetches of data from remote memories which may not occur when the program is run using P processors.

   (b) The cache hit frequency may decrease during the simulation as the cache size of the single processor is only equal to that of one of the parallel processors. Similarly, several operations done directly from registers in the case of parallel execution may require memory access during the single processor simulation.

(c) Cost of cycling through the programs of the P processors in round-robin fashion. For instance, we need to keep track of P program counters.

As a result, the single processor actually takes $cPt(I)$ time for some $c > 1$. The speedup between the single processor time and the parallel time is $cPt(I)/t(I) > P$.

Lai and Sahni [23] prove that it is possible for parallel branch-and-bound algorithms to exhibit unbounded real speedup (i.e., speedup $>> P$) on certain instances. Speedups in excess of P are also possible for the case of relative speedup and absolute real speedup. The "proof" that speedup cannot exceed P does not apply to relative speedup as the serial simulation of the parallel program is different from the parallel program itself and in relative speedup, we use the ratio of the serial time and parallel time of the same program.

Some additional papers in which one can find analytical and / or experimental observances of speedup that exceeds P are [2], [3], [17], [27], [30], [36], [39], and [44].

2.5 Cost Normalized Speedup

Often, in addition to knowing how much faster the parallel system runs, we want to get a measure of the cost at which this improved performance has been accomplished. So, we define a cost normalized speedup:

$$CostNormalizedSpeedup(I,P) = \frac{speedup(I,P)}{\text{(cost of parallel system)/(cost of serial system)}}$$

To obtain the cost normalized speedup, we need to know the costs of the parallel and serial systems in addition to the speedup. The cost of the parallel system should include the hardware and software cost. Should it also include the maintenance cost, operating cost, cost of the support personnel, etc. ? Because of differences in discounts, the hardware cost varies from installation to installation. The same computer bought a year ago may cost substantially less today. Should we use last year's cost (i.e., its cost to us) or today's (i.e., its replacement cost) or next year's projected cost (i.e., its cost to future purchasers of the system)? Furthermore, the parallel hardware may see a different number of users than the serial hardware and we may wish to amortize the hardware cost over the users. The software cost may be difficult to determine unless the software was bought off-the-shelf (an unlikely scenario in parallel computing but a possiblity in serial computing). Even with software, we have the issue of using total cost over amortized cost, today's cost over tomorrow's cost, amortizing the maintenance cost, factoring in the learning cost, etc. The serial system cost depends on the version of speedup in use. So, once we known which version of speedup is in use, the serial system is well defined. However, actually determining the serial system cost to use might be quite difficult. The reasons for this are the same as those for a parallel system. Even though cost normalized speedup appears to be an attractive measure, the difficulties associated with determining the cost to attribute to the parallel and serial systems has prevented this measure from becoming a commonly reported one.

2.6 Efficiency

Efficiency is a performance metric closely related to speedup. It is the ratio of speedup and the number of processors P. Depending on the variety of speedup used, one gets a different variety of efficiency. Since speedup can exceed P, efficiency can exceed 1. Efficiency, like speedup, should not be used as a performance metric independent of run time. The reasons for this are the same as for not using speedup as a metric independent of run time. Since speedup favors slow processors and inefficient code, efficiency favors these too. Also, the easiest way to get a relative efficiency of one is to use $P = 1$. A real efficiency of 1 can be obtained by using the best serial algorithm and $P = 1$. This, however, results in no speedup.

Alternate formulations of efficiency can be found in the literature. For example, Carmona and Rice [6] define efficiency as the ratio of work accomplished (wa) by a parallel algorithm and the work expended (we) by the algorithm. If we define the work accomplished by the parallel algorithm to be the work that would have been done by the "best" serial algorithm and the work expended to be the product of the parallel execution time, the speed (S) of an individual parallel processor, and the number P of processors, then assuming all processors have the same speed,

$$we = (\text{parallel time}) * S * P$$

$$wa = (\text{best sequential time}) * S$$

$$\frac{wa}{we} = \frac{\text{best sequential time}}{P * \text{parallel time}}$$

which equals real speedup divided by P. So, $\frac{wa}{we}$ equals the real efficiency. Similarly, if wa is defined as the work done by the parallel algorithm when executed on a single processor, then $\frac{wa}{we}$ equals the relative efficiency (i.e., relative speedup divided by P).

Another alternate, but equivalent, formulation of efficiency is provided in [6]. Define *wasted work*, ww, to be $we - wa$. Then efficiency may be written as

$$\frac{wa}{we} = \frac{wa}{ww + wa} = \frac{1}{1 + \frac{ww}{wa}}$$

2.7 Scalability

We have noted earlier that for many parallel systems (i.e., parallel algorithm – parallel computer combinations), speedup declines when the problem size is fixed and the number of processors is increased and that speedup increases when the number of processors is fixed and the problem size increases. The term *scalability of a parallel system* is used to refer to the change in performance of the parallel system as the problem size and computer size increase. Intuitively, a parallel system is scalable if its performance continues to improve as we scale (i.e., increase) the size of the system (i.e., problem size as well as machine size increase).

2.7.1 Asymptotic Scalability

Nussbaum and Agrawal [32] suggest using the ratio of asymptotic relative speedup of the parallel system and asymptotic relative speedup of an "ideal" system comprised of a "similar" parallel algorithm and an EREW PRAM.

Since many parallel algorithms are closely tied to the architecture on which they are to run, it is often neither possible nor desirable to execute exactly the same algorithm on parallel computers with differing architectures. For example, Dekel et al. [9] describe a $\Theta(n)$ algorithm to multiply two $n \times n$ matrices on an $n \times n$ mesh connected computer. The algorithm is based on the classical $\Theta(n^3)$ serial algorithm for matrix multiplication. The speedup obtained by the mesh algorithm of [9] is $\Theta(\frac{n^3}{n}) = \Theta(n^2)$. This is the maximum speedup obtainable using classical matrix multiplication on a mesh of any size. The mesh implementation involves several steps such as data alignment and interprocessor data shifts that are peculiar to the mesh architecture. These steps are unnecessary on an EREW PRAM. As a result, when obtaining the asymptotic relative speedup of the algorithm on an EREW PRAM, we need to use an algorithm that does "essentially" the same computations but avoids the communication necessitated work done on the mesh. Therefore, we need to use a "similar" algorithm, i.e., one that is based on the same serial algorithm rather than one based on an asymptotically faster serial algorithm. The run time of the "similar" parallel algorithm on an EREW PRAM with $n^3/\log n$ processors in $\Theta(\log n)$. The speedup is $\Theta(n^3/\log n)$ and this is the maximum speedup attainable by the "similar" parallel algorithm on an EREW PRAM.

The asymptotic scalability of the mesh matrix multiplication system is therefore

$$\frac{\text{asymptotic relative speedup of parallel system}}{\text{asymptotic relative speedup of ideal system}}$$

$$= \frac{\text{asymptotic relative speedup of parallel system}}{\text{asymptotic relative speedup of EREW PRAM system}} = \frac{\log n}{n}$$

As n increases, the scalability approaches zero. This means that the performance of the mesh matrix multiplication system continues to deteriorate relative to that of an "ideal" matrix multiplication system.

The hypercube matrix multiplication system of [9] has a parallel execution time of $\Theta(\log n)$ and uses $n^3/\log n$ processors. Its asymptotic relative speedup is $\Theta(n^3/\log n)$. So, the scalability of the hypercube system is 1. So, under the difinition of Nussbaum and Agarwal, the hypercube matrix multiplication system is more scalable than the mesh system. From this, we can conclude that the hypercube system comes closer (in this case it actually equals) to achieving the "ideal parallelism" that is possible for the algorithm.

Despite the conclusion that the scalability of the hypercube system is better than that of the mesh system, the efficiency of both systems is the same, i.e., one. The added speedup results from the use of additional processors. Furthermore, if we increase our matrix size from $n \times n$ to $2n \times 2n$, the workload (i.e., the serial work n^3)

increases by a factor of eight, the number of processors in the mesh computer needs to increase by a factor of four to get maximum speedup and retain an efficiency of one. However, the number of processors in the hypercube computer needs to increase by a factor of almost eight to maintain efficiency at one and obtain maximum speedup. If we use efficiency as our performance metric then to avoid a degradation in efficiency, the hypercube size needs to increase faster than the mesh size as problem size is increased. Yet, using asymptotic scalability, the mesh system's scalability is poor relative to that of the hypercube system.

The notion of asymptotic speedup doesn't incorporate the number of processors in either the parallel system or the ideal EREW PRAM system. So, a matrix multiplication algorithm that takes $\Theta(\log n)$ time using n^3 processors has the same asymptotic scalability as one that uses $n^3/\log n$ processors and takes $\Theta(\log n)$ time. The relative efficiency of the former system is $\Theta(1/\log n)$ while that of the latter is $\Theta(1)$. The efficiency of the first system deteriorates as n increases while that of the latter remains steady. Yet, both are equally scalable.

2.7.2 Isoefficiency

Kumar, Nageshwara, and Ramesh [20] proposed a scalability measure based on efficiency. In this, the scalability, or *isoefficiency*, of a parallel system is defined to be the rate at which workload must increase relative to the rate at which the number of processors is increased so that the efficiency of the parallel system remains the same. Isoefficiency is generally a function of the instance and number of processors in the parallel computer. Depending on the version of efficiency used, we get different versions of isoefficiency (e.g., real isoefficiency, absolute isoefficiency, relative isoefficiency). When using real efficiency, the workload is defined to be the total work done by the best serial algorithm for the problem. When using relative efficiency, the workload is defined to be the total work done by the parallel algorithm when run on a single processor. Several isoeffiency studies can be found in the literature (see [14], [20], [22], [50], and [51], for example).

Consider the problem of multiplying two $n \times n$ matrices on an $m \times m$ mesh computer. Assume that the parallel matrix multiplication algorithm is based on the classical algorithm of complexity $\Theta(n^3)$. Then, the workload is cn^3. Assuming a computational rate of one unit of workload per unit of time, the serial execution time is cn^3. If m divides n the workload may be evenly distributed over the $P = m^2$ processors by assigning each processor an $\frac{n}{m} \times \frac{n}{m}$ portion of the input and output matrices. Following the development in [9], the parallel execution time is $cn^3/m^2 + bn^2/m$ where bn^2/m accounts for the communication and other overheads in the parallel algorithm. The relative speedup is therefore

$$\frac{cn^3}{cn^3/m^2 + bn^2/m} \tag{2}$$

and the relative efficiency is

$$\frac{cn^3}{m^2(cn^3/m^2 + bn^2/m)} = \frac{1}{1 + \frac{bm}{cn}} \tag{3}$$

For the relative efficiency to remain constant, $bm/(cn)$ should remain constant. For this, the workload n^3 should increase at the rate $(bm/c)^3 = (b/c)^3 P^{1.5}$. The isoefficiency, $ie(W = n^3, P)$, of the mesh matrix multiplication system is therefore $(b/c)^3 P^{1.5}$. So, the workload must increase at the rate $(b/c)^3 P^{1.5}$ to maintain constant efficiency. If the workload does not increase at this rate (or higher), then the parallel system's efficiency will decline.

Barring any anomalous behavior, the workload for an arbitrary problem must increase at least linearly in P as otherwise processor starvation will occur for large P and efficiency will decline. Hence, in the absence of anomalous behavior, $ie(W, P)$ is $\Omega(P)$. Parallel algorithms with smaller $ie(W, P)$ are more scalable than those with larger $ie(W, P)$.

As noted in [20], [22], [51], the concept of isoefficiency is useful because it allows one to test parallel programs using a small number of processors and then predict the performance for a larger number of processors. Thus it is possible to develop parallel programs on small hypercubes and also do a performance evaluation using smaller problem instances than the production instances to be solved when the program is released for commercial use. From this performance analysis and the isoefficiency analysis one can obtain a reasonably good estimate of the program's performance in the target commercial environment where the multicomputer may have many more processors and the problem instances may be much larger. So with this technique we can eliminate (or at least predict) the often reported observation that while a particular parallel program performed well on a small multicomputer it was found to perform poorly when ported to a large multicomputer. Isoefficiency may also be used to analyze the effects of changing processor and communication speed on the performance of the parallel system [22].

The isoefficiency of the stripes partitioning method of [50] for the connected components problem is $P^2 \log^2 P$. The measured real efficiency for dense graphs with $n = 64$ vertices and $P = 8$ processors is 0.2. Since the workload in n^2, to obtain an efficiency of 0.2 with $P' = 16$ processors, the workload, n'^2, should be at least $n^2 P'^2 \log^2 P' / (P^2 \log^2 P)$. So, n' should be at least $nP' \log P' / (P \log P) = 64 * 16 * 4/(8 * 3) = 170\frac{2}{3}$. The experimental results of [50] indicate that with 16 processors, the efficiency becomes 0.2 for a value of n somewhere between 128 and 256!

As was the case for speedup and efficiency, isoefficiency should not be used as the sole performance metric of a parallel system. Let us consider using isoefficiency to compare two matrix multiplication systems. Both systems are comprised of the same number of identical processors. One has speedup and efficiency given by Equations 2 and 3, respectively. Suppose that the other system has a parallel execution time of $2cn^3/m^2 + (b/2)n^2/m^2$ for $m \leq n$. In the second system, we have managed to reduce the communication overhead by a factor of two but this has come at the expense of doubling the time spent on computation. The isoefficiency of the second system can be obtained from that of the first by replacing b by $b/2$ and c by $2c$. Doing this, we determine the isoefficiency of the second system to be $(b/(4c))^3 P^{1.5}$ which is better than the isoefficiency of the first system by a factor of 64. The second system is therefore considerably more scalable than the first. However, if we compare the run times, we see that when $b = 4c$,

$$\frac{\text{run time of system 2}}{\text{run time of system 1}}$$

$$= \frac{2n^3 + 2n^2}{n^3 + 4n^2}$$

which is bigger than one for $n > 2$. Hence, the more scalable system runs slower than the less scalable system almost always. In fact, for large n, the more scalable system takes almost twice as much time.

Notice that the preceding analysis also applies to real efficiency. For this, we need to replace the serial time with that of the "best" serial algorithm or of the serial algorithm used "in practice". Using the latter, the serial time is cn^3.

2.7.3 Isospeed

Sun and Rover [43] use the average workload per processor needed to sustain a specified computational speed as a measure of scalability. Workload, W, is defined as the work (say floating point operations, instruction count, etc.) done by a serial algorithm. So, it excludes extraneous computations that might be introduced during parallelization as well as communication and other overheads encountered during parallel execution. One consequence of using operation counts such as total number of floating point operations as a measure of workload is that the speed of a serial program may vary with the value of W. This is because some of the excluded workload (such as loop initialization, set-up of vector operations, procedure invocation and return, certain special case tests, etc.) is independent of the total number of floating point operations and when W is large this excluded work has less impact on the measured MFLOPs than it does when W is small. W may also impact the speed of a serial processor because of architectural considerations such as the requirement for remote memory access when W is large [43].

Suppose that a P processor parallel system achieves an average per processor speed of xMFLOPS using a workload of W. The parallel execution time of the system is then $T = W/(Px)$ seconds. If the number of processors is increased to P', then the average speed will generally decrease unless the workload is increased. This is so as for many applications, overheads (such as interprocessor communication) increase when the workload is fixed and the number of processors is increased. So, to obtain a speed of xMFLOPS, it is necessary to increase the workload to W'. The time needed to perform the increased workload using P' processors is $T' = W'/(P'x)$. The scalability or *isospeed*, $\Psi(P, P')$, of the parallel system is defined to be [43] [44]

$$\Psi(P, P') = \frac{W/P}{W'/P'} = \frac{T}{T'} \tag{4}$$

Since the average speed (xMFLOPS) attained by a parallel system is a function of the workload, $\Psi(P, P')$ is really also a function of the initial workload W. To remove this dependence on W, Sun and Rover [43] propose a particular value of x to use. They define r_∞ to be the maximum speed attained by a single processor execution of the parallel algorithm as the workload is increased. They use $x = r_\infty/2$ as the average speed that is to be attained during the parallel execution.

As noted earlier, W'/P' is generally $\geq W/P$. So, $\Psi(W, P) \leq 1$. Parallel systems with isospeed closer to one are more scalable than those with isospeed much smaller than one. Recall that when the isoefficiency metric is used, systems with small isoefficiency value are considered more scalable than those with a large one.

Sun and Zhu [44] have noted that isospeed is closely related to isoefficiency when the speed of serial computation is independent of the workload. To see this, note that under this assumption, workload and serial time are related by a constant, the speed, S, of the serial computer. Suppose that a workload of W can be performed at an average speed of xMFLOPs per processor when P processors are used and that a workload of W' is needed to achieve the same average speed when $P' > P$ processors are used. The relative efficiency of the first parallel system is

$$\frac{\text{serial time}}{P * \text{parallel time}} = \frac{\frac{W}{S}}{P\frac{W}{Px}} = \frac{x}{S}$$

This equals the relative effeciency of the second system. So, the rate at which the workload must increase relative to the rate at which the number of processors is increased so as to maintain constant efficiency is

$$ie(W, P) = \frac{W'/W}{P'/P} = \frac{W'/P'}{W/P} = \frac{1}{\Psi(W, P)}$$

3 Models

Uniprocessor computing has benefitted significantly from the existence of a simple theoretical model, i.e., the RAM, of a uniprocessor computer. This model has made it possible to (a) develop uniprocessor algorithms and (b) establish algorithm correctness and expected performance relatively independent of the specific uniprocessor computer on which the program is to be run. Optimizing for machine dependent features such as cache size, number of registers, etc. is handled by the compiler. Generally, at the algorithm or programmer level, one does not take these considerations into account. Even though a model for RAMs with hierarchical memories has been proposed [1], this has not found significant use. The simplicity of the RAM model and its accuracy in modeling a uniprocessor computer coupled with the fact that uniprocessor compiler technology is sufficiently sophisticated to bridge the gap between the model and specific uniprocessor computers has made widespread and efficient uniprocessor computing possible.

Extrapolating from the uniprocessor case, we arrive at the following minmal requirements that a model for parallel computers should meet:

1. it should be conceptually simple to understand and use
2. algorithms determined to be correct for the model should be correct for the target architectures
3. the observed performance of the algorithm on a real parallel computer should correspond to the performance predicted from the model
4. the model should be sufficiently close to real architectures that compiler technology can bridge the gap between the two and perform optimizations specific to the real architecture (one consequence of this requirement is that code for parallel computers is portable).

When we look at parallel computers, we find there are a very large number of abstract models. However, none enjoys the simplicity and accuracy of the RAM. Despite this, none even attempts to serve as a model for all classes of parallel computers. Further, parallel compiler technology is unable to bridge the gap between the available parallel computer models and commercial parallel computers in the modeled class.

The difficulties involved in attempting to formulate a single simple and accurate model for parallel computers can be gauged by examining the variations in commercial and proposed parallel computers:

1. There are synchronous, asynchronous, and semi-asynchronous parallel computers.
2. Some have shared memory, some have distributed memory, some have both.
3. Some operate in single-instruction-multiple-data (SIMD) mode while some others operate in multiple-instruction-multiple-data (MIMD) mode. In the

SPMD mode, all processors of the parallel computer execute the same program. However, different processors can be executing different parts of this program at any given time.

4. Parallel computers differ in the interconnection network used. Some of the common interconnection networks are: hypercube, omega, mesh, and fat tree.
5. Differences in routing methods. For example, some computers might route messages through the interconnection network using a store-and-forward scheme while others might use wormhole routing.
6. Recently proposed architectures that use reconfigurable electronic and optical buses use the interconnection network to perform tasks that can be done only by processor computation in other parallel architectures [5], [25], [28], [37], [46], and [47].

The simplest and oldest of the parallel computer models is the PRAM. This simply extends the RAM model by permitting several processors to share the same memory. The PRAM model is a synchronous shared memory model (SMM). Since it makes no provision for the stated variations in 2 – 6 above, it cannot possibly model all synchronous parallel computers accurately. In fact, even to model shared memory synchronous parallel computers, one needs to consider variations such as EREW-PRAM, CREW-PRAM, and CRCW-PRAM that account for differences in memory access conflict resolution schemes. The relationship between different basic PRAM models is discussed in [11]. Other variations of the PRAM model have been introduced to model shared memory parallel computers that are asynchronous (APRAM model [7]) and semi-asynchronous (PPRAM – phase PRAM [13]).

Since most commercial parallel computers are a collection of processor-memory pairs that operate asynchronously and communicate via an interconnection network, efforts have been made to develop models for these computers that are more accurate than the asynchronous PRAM. Two attempts in this direction are the bulk-synchronous parallel (BSP) model [48] and the LogP model [8]. The bulk-synchronous model is actually a semi-asynchronous model in which the processors (or subsets of processors) work asynchronously for L time units and then are synchronized by some hardware mechanism. In this model, a parallel program is a sequence of supersteps. Each superstep is made up of computational and routing tasks. During a superstep, processors can execute computational tasks, and send and receive messages from other processors. The activities within a superstep are done asynchronously. Synchronization is done following each superstep. The parallel computer starts the first superstep and then checks for its completion after L time units. In case the superstep is complete, the next superstep is initiated. If not, another L time units are alloted the first superstep and a test for completion is done at the end of these L units. This process is repeated until the first superstep completes. Succeeding supersteps are handled in the same way. Interprocessor communication is handled by a "network topology insensitive" router that realizes h-relations (in an h-relation, each processor sends at most h messages and receives at most h messages). The cost of realizing an h-relation is gh where g is a measure of the network bandwidth. Note that because of the semi-asynchronous nature of the model, the actual cost of an h-relation is $\lceil gh/L \rceil$ supersteps.

The BSP model comes closer to modeling distributed memory asynchronous computers than any of the PRAM variations. It achieves increased model accuracy (or realism) while maintaing simplicity at the expense of being insensitive to the network topology. Unfornuately, this very reason makes it is unable to account for programming differences attributable to differences in the network topology. These differences become important when we are modeling computations that require the simultaneous transfer of messages between several pairs of processors. We shall elaborate on this when we consider the LogP model which is also insensitive to the network topology.

In the LogP model [8], an asynchronous network of processors is modeled by the following parameters:

1. **L**atency $\cdots$ The maximum time needed for a "small" message to travel between any two processors.

2. **o**verhead $\cdots$ The amount of time a processor spends when sending or receiving a message. During this time, the processor cannot engage in other activities.

3. **g**ap $\cdots$ The minimum time between successive message sends and successive message receives by the same processor.

4. **P** $\cdots$ The number of processors.

The interconnection network in the LogP model has a finite capacity and at no time can there be more than $\lceil L/g \rceil$ messages with the same processor as source or more than $\lceil L/g \rceil$ messages with the same processor as destination. Since, by definition, all messages reach their destination within L units of time. Further, a source can transmit a message only every g time units. So, by the time the source transmits $\lceil L/g \rceil$ messages, the first must have reached its destination. Hence, the number of in-transit messages from any source cannot exceed this amount. Also, since a destination can receive a message only every g units and messages take at most L time to reach their destination, the presence of more than $\lceil L/g \rceil$ messages with the same destination implies the need for buffers to hold messages that cannot be received by the destination.

Both the bulk-synchronous and LogP models ignore the specifics of the interconnection network and so attempt to promote the development of programs/algorithms that are portable across a range of interconnection networks. Unfortunately, ignoring the specifics of the interconnection network can result in the development of algorithms that perform well on the model but not on the target parallel computer. While current wormhole and cut-through routing technology has resulted in router performance (on a single point-to-point message transfer) that is relatively independent of the underling routing (interconnection) network, the ability of a network to realize several point-to-point connections simultaneously is quite sensitive to the topology and the router. The path used to realize one source-to-destination message transfer may block a simultaneous transfer between another source-destination pair.

As an example of this, consider the optimal all-to-all broadcast algorithm developed for the LogP model in [19]. In this, each processor sends a message to each of the remaining processors. The proposed algorithm (assuming $g \geq o$) is:

Processor i sends its message to processors $(i+1) mod P$, $\cdots$, $(i+P-1) mod P$ at time 0, g, $2g$, $\cdots$, $(P-2)g$.

Using this algorithm and the assumptions of the model, the message from processor i is received by the destination processors at times $L+2o$, $L+2o+g$, $\cdots$, $L+2o+(P-2)g$. As shown in [19], this is optimal for the model as no processor can receive its first message before time $L + 2o$ and each succeeding message must be received with a gap of at least g. Hence, the earliest time a processor can receive all of its messages is $L + 2o + (P - 2)g$.

Now, let's see how this algorithm performs when we take network topology into account. First lets consider a ring topology. For simplicity, we assume this is a unidirectional link and processor i can send a message to processor $(i + 1) mod P$ directly. For processor i to send a message to processor j, a communication path between the two processors needs to be established and reserved for the duration of the transmission. We assume that all links on the established communication path are reserved for L units of time. Since there is only one outbound link from processor i, we set $g = L$ (the processor must wait for the transmission to complete before it can initiate another transmission on its outbound link). Further, we assume $o = 0$. Consider processor i for some i. As predicted by the model, messages sent by this processor reach their destinations at times L, $2L$, $\cdots$, $(P - 1)L$. The first message transmittal ties up one link for L units of time. The second ties up two links for L time, the third ties up three links for L time, and so on. Using the sum of the products of links used and time for which they are used as our resource metric, the broadcast from processor i uses $P(P - 1)L/2$ resource. When we consider the broadcasts from all P processors, the resource requirement becomes $P^2(P-1)L/2$. Since we have only P links in the ring, even with 100% link usage during the entire broadcast algorithm, it would take $P^2(P-1)L/2/P = P(P-1)L/2$ time to muster this much resource. So, the algorithm cannot complete before time $P(P - 1)L/2$ (rather than the predicted time of $(P - 1)L$). The predicted run time and actual run time are off by a factor of $P/2$. Furthermore, by circulating the P different messages around the ring, we can accomplish the all-to-all broadcast in $(P - 1)L$ time (the actual time will be less as messages are sent only to neighbor processors). Hence, the optimal LogP algorithm is *not optimal* for the ring connected parallel computer. It is slower than the optimal by a factor of $P/2$!

The above analysis can be extended to a $\sqrt{P} \times \sqrt{P}$ mesh connected computer with wraparound rows and columns. One way to accomplish the all-to-all broadcast is to first circulate the data along rows as in a ring. This takes $(\sqrt{P} - 1)L$ time. Next, we have $\sqrt{P}$ rounds of data circulation along columns. In round i data that was initially in column i is circulated. Each round takes $(\sqrt{P}-1)L$ time. The total time is $(P-1)L$. Again, the actual time could be a lot less as all comunications are between adjacent processors. Using the optimal LogP algorithm, each processor uses $O(P^{1.5}L)$ units of resource. So, the P processors together need $O(P^{2.5}L)$ resource units. The mesh has $O(P)$ links. Hence with 100% link utilization, the expected run time is $O(P^{1.5}L)$ rather than the predicted time of $(P-1)L$. The optimal LogP broadcast algorithm runs slower than the optimal mesh broadcast algorithm by a factor of $O(\sqrt{P})$.

So, even though the LogP model is a more accurate model of an asynchronous distributed memory message passing parallel computer than the asynchronous PRAM, it isn't accurate enough that efficient LogP algorithms are efficient algorithms for real asynchronous distributed memory message passing parallel computers. The differences between the optimal LogP all-to-all broadcast algorithm and the optimal ring and mesh algorithms are sufficient that it is unlikely that parallel compiler technology will ever bridge the gap.

Li, Mills, and Reif [26] have compiled a list of parallel computer models and characterized these models by resource metrics (degree of asynchrony, memory organization, latency, bandwidth, etc.). Their list includes fourteen models PRAM, VRAM (vector RAM), LPRAM (local-memory PRAM), BPRAM (block PRAM), PPRAM, PLPRAM (phase LPRAM), APRAM, BSP (bulk-synchronous parallel), LogP, PMH (parallel memory hierarch), P-HMM (parallel hierarchical memory model), H-PRAM (hierarchical-PRAM), and LogP-HMM. Yet other models can be found in [38], [31] and [49]. *It is evident that the number of parallel computer models exceeds the number of different parallel computer architectures that have been either proposed or built.* Each model attempts to provide an abstraction that can be used to develop algorithms and programs for a class of parallel computers. The abstraction is generally simpler to work with than any instance of the modeled class. However, this simplification is obtained at the expense of introducing modeling inaccuracy. Given the inability of parallel compilers to compensate for this inaccuracy, algorithms developed for the model often result in inefficient code on the target parallel computer. As a result, the applicability of the model is severely limited.

4 Conclusion

We have seen that a large number of performance metrics have been proposed for parallel systems. Each of these exposes some interesting aspect of the parallel system that is not exposed by the other metrics. When measuring the performance of a parallel system, we should keep in mind that the primary performance metric is run time. Speedup, efficiency, and scalability are important only to the extent that they result in improved run time.

The area of parallel computer models has received significant impetus by the introduction of the BSP and LogP models. These models are simple and more realistic models for distributed memory message passing asynchronous parallel computers than earlier PRAM based models that completely ignored interprocessor communication needs. However, since these models model the network with a limited number of parameters, distinctly different networks map into the same set of parameters. As a result, one needs to be very careful when attempting to use an efficient algorithm for the model on a real parallel computer. Without appropriate care, the result can be a rather inefficient parallel system.

Acknowledgement

The authors are grateful to Professor Xian-He Sun for providing valuable comments on an earlier version of this paper.

References

[1] A. Aggarwal, B. Alpern, A. Chandra, and M. Snir, A model for hierarchical memory, *Proceedings 19th ACM STOC*, 305-314, 1987.

[2] S. Akl, M. Conrad, and A. Ferreira, Data-movement-intensive problems: two folk theorems in parallel computation revisited, *Theoretical Computer Science*, 323-327, 1992.

[3] S. Akl and L. Lindon, Paradigms admitting superunitary behaviour in parallel computation, Queen's University at Kingston, Technical Report, 1993.

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

[5] Y. Ben-Asher, D. Peleg, R. Ramaswami, and A. Schuster, The power of reconfiguration, *Journal of Parallel and Distributed Computing*, 13, 139-153, 1991.

[6] E. Carmona and M. Rice, Modeling the serial and parallel fractions of a parallel algorithm, *Jr. of Parallel and Distributed Computing*, 13, 286-298, 1991.

[7] R. Cole and O. Zajicek, The APRAM: Incorporating asynchrony into the PRAM model, *Proceedings ACM SPAA*, 169-178, 1989.

[8] D. Culler, R. Karp, and D. Patterson, LogP: Towards a rea;listic model of parallel computation, *Proceedings ACM Symp. on Principles and Practices of Parallel Programmin*, 169-178, 1993.

[9] E. Dekel, D. Nassimi, and S. Sahni, Parallel Matrix and Graph Algorithms, *SIAM Computing*, vol 10, no. 4, pp 657-675, 1981.

[10] M. Driscoll and W. Daasch, Accurate predictions of parallel program execution time, *Jr. of Parallel and Distributed Computing*, 25, 16-30, 1995.

[11] D. Eppstein and Z. Galil, Parallel algorithmic techniques for combinatorial computation, *Ann. Rev. Comput. Sci.*, 233-283, 1988.

[12] H. Flat and K. Kennedy, Performance of parallel processors, *Parallel Computing*, 12, 1-20, 1989.

[13] P. Gibbons, A more practical PRAM model, *Proceedings ACM SPAA*, 158-168, 1989.

[14] A. Gupta and V. Kumar, Scalability of parallel algorithms for matrix multiplication, *Proc. International Conference on Parallel Processing*, III-115–III-119, 1993.

[15] J. Gustafson, Reevaluating Amdahl's Law, *CACM*, 31, 5, 532-533, 1988.

[16] D. Hall and M. Driscoll, Hardware for fast global operations on multicomputers, *Proc. 9th Intl. Parl. Proc. Symposium*, 673-679, 1995.

[17] D. Helmbold and C. McDowell, Modeling speedup(n) greater than n, *Proc. 1989 International Conference on Parallel Processing*, Vol. III, 219-225, 1989.

[18] J. JaJa, *An introduction to parallel algorithms*, Addison Wesley, 1992.

[19] R. Karp, A. Sahay, and E. Santos, Optimal broadcast and summation in the LogP model, Technical Report, University of California, Berkeley, 1993.

[20] V. Kumar, V. Nageshwara, and K. Ramesh, Parallel depth first search on the ring architecture, *Proc. 1988 Intl. Conf. on Parallel Processing*, Penn. State Univ. Press, 128-132, 1988.

[21] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to parallel computing*, Benjamin / Cummings, California, 1994.

[22] V. Kumar and A. Gupta, Analyzing scalability of parallel algorithms and architectures, *Journal of Parallel and Distributed Computing*, 22, 3, 379-391, 1994.

[23] S. Lai and S. Sahni, Anomalies in parallel branch-and-bound, *CACM*, 27, 6, 594-602, 1984.

[24] T. Leighton, *Introduction to parallel algorithms and architectures*, Morgan Kaufman, California, 1992.

[25] H. Li and M. Maresca, Polymorphic-torus architecture for computer vision, *IEEE Trans. on Pattern & Machine Intelligence*,

[26] Z. Li, P. Mills, and J. Reif, Models and resources metrics for parallel and distributed computation, *Proc. 28th Annual Hawaii International Conference on System Sciences*. 11, 3, 133-143, 1989.

[27] D. McBurney and M. Sleep, Transputer-based experiments with the ZAPP architecture, *Lecture Notes in Computer Science*, Springer Verlag, 258, 242-259, 1987.

[28] R. Miller, V. K. Prasanna Kumar, D. Resis and Q. Stout, Meshes with reconfigurable buses, Proceedings 5th MIT Conference On Advanced Research IN VLSI, 1988, pp 163-178.

[29] D. Nicol and F. Willard, Problem size, parallel architecture, and optimal speedup, *Jr. of Parallel and Distributed Computing*, 5, 404-420, 1988.

[30] D. Nicol, Inflated speedups in parallel simulations via malloc( ), *International Journal on Simulation*, 2, 413-426, 1992.

[31] M. Nodine and J. Vitter, Large-scale sorting in parallel machines, *Proceedings ACM SPAA*, 29-39, 1991.

[32] D. Nussbaum and A. Agarawal, Scalability of parallel machines, *CACM*, 34, 3, 57-61, 1991.

[33] J. Ortega and R. Voight, Solution of partial differential equations on vector and parallel computers, *SIAM Review*, 1985.

[34] S. Ranka and S. Sahni, *Hypercube algorithms*, Springer-Verlag, New York, 1990.

[35] S. Ranka and S. Sahni, Image template matching on MIMD hypercube multi-computers, *Jr. of Parallel and Distributed Computing*, 10, 1990, pp. 79-84.

[36] V. Rao and V. Kumar, On the efficiency of parallel backtracking, *IEEE Trans. on Parallel and Distributed Systems*, 4, 4, 427-437, 1993.

[37] S. Sahni, Data manipulation on the distributed memory bus computer. To appear in *Parallel Processing Letters.*

[38] D. Skillicorn, Models for practical parallel computation, *Intl. Jr. of Parallel Programming*, 20, 2, 133-158, 1991.

[39] E. Speckenmeyer, B. Monien, and O. Vornberger, Superlinear speedup for parallel backtracking, *Lecture Notes in Computer Science*, Springer Verlag, 297, 985-993, 1988.

[40] X. Sun and L. Ni, Another view on parallel speedup, *Proceedings Supercomputing 90*, 324-333, 1990.

[41] X. Sun and J. Gustafson, Toward a better parallel performance metric, *Parallel Computing*, 17, 1093-1109, 1991.

[42] X. Sun and L. Ni, Scalable problems and memory-bounded speedup, *Jr. of Parallel and Distributed Computing*, 19, 27-37, 1993.

[43] X. Sun and D. Rover, Scalability of parallel algorithm-machine combinations, *IEEE Trans. on Parallel and Distributed Systems*, 5, 6, 599-613, 1994.

[44] X. Sun and J. Zhu, Shared virtual memory and generalized speedup, *Proc. 8th International Parallel Processing Symposium*, 637-643, 1994.

[45] Z. Tang and G. Li, Optimal granularity of grid iteration problems, *Proc. 1990 ICPP Conference*, I-111–I-118, 1990.

[46] R. Thiruchelvan, J. Trahan, and R. Vaidyanathan, On the power of segmenting and fusing buses, *Proc. 1993 International Parallel Processing Symposium*, 1993, 79-83.

[47] Vaidyanathan, R., Sorting on PRAMs with reconfigurable buses, *Information Processing Letters*, 42, 1992, 203-208.

[48] L. Valiant, A bridging model for parallel computation, *C. ACM*, 33, 8, 103-111, 1990

[49] Algorithms for parallel memory II, *Algorithmica*, 1994.

[50] J. Woo and S. Sahni, Hypercube computing: Connected components, *Jr of Supercomputing*, 3, 209-234, 1989.

[51] J. Woo and S. Sahni, Computing biconnected components on a hypercube, *Jr. of Supercomputing*, 5, 1, 73-87, 1991.

[52] X. Zhou, Bridging the gap between Amdahl's law and Sandia laboratory's result, *CACM*, 32, 6, 1014-1015, 1989.