

Scalable Computers

Boleslaw K. Szymanski

Department of Computer Science

Rensselaer Polytechnic Institute, Troy, NY12180, USA

Abstract

The driving force behind the development of parallel hardware and software is the desire to solve ever larger problems in an acceptable amount of computation time. Hence, one of the most important measures of quality of a parallel computer or algorithm is its scalability which describes the parallel system behavior when problems of increasing size are solved on increasingly large computer configurations. In this article, we explain the notion of scalability in parallel systems and discuss underlying issues in computer technology and software engineering. We start with the definitions of the parallel program speedup, efficiency and scalability. Then, we discuss bounds on the speedup such as Amdahl's Law, Berstis's Law, and communication and synchronization limits. We also reflect on how these bounds impact different classes of parallel architectures (SIMD vs. MIMD and shared vs. distributed memory). Next, we introduce the Bulk Synchronous Parallelism model and discuss how it fosters designing "immortal" parallel algorithms which can be proven optimal independently of the used technology. In conclusion, we describe current trends in parallel architectures and applications.

1 Introduction and Genesis of Parallel Computing

It is widely recognized that the computer technology has become a critical component of everyday life of a modern society. The computer has become

ubiquitous in manufacturing, services, products, entertainment. Computers have been changing ways in which we conduct business, produce goods and do science.

Parallel processing is currently a small fraction of overall computer technology and the Computer Revolution. Yet, there are two compelling reasons for parallel processing to have increasing importance in the future. The first one is that parallel processing supports the largest computations which became an integral part of sciences, medicine and manufacturing. Large-scale computer modeling enabled by parallel processing impacts decision making in banking, finance, military and government. Parallel computers empower decision makers, such as high-level managers, military leaders and chief scientists with the ability to gather, access, and synthesize information, as well as to simulate real-life processes to measure the impact of social, economical and industrial decisions. The quality of the simulations and synthesized information strongly depends on the amount of applied computational power. Today, even the largest uniprocessor computers are too slow for the more challenging problems of this kind.

The second reason for growing importance of parallel processing is that improvements in speed of sequential computing are approaching technological limits. The semiconductor industry has been doubling processor speed every 18 months for the last few decades, but this rate of improvement cannot last as processor design technology is maturing. In addition to economical forces (the cost of hardware needed to fabricate chips with smaller dimensions is an exponential function of the size improvement), the basic laws of physics limit the speed of a uniprocessor. The speed of signal transmission in a computer cannot exceed the speed of light in the transmission media, which is about 300,000 km/sec for silicon. Consequently, it takes one billionth of a second for a signal to propagate in a silicon chip of an inch in diameter. However, one signal propagation can support at most one floating point operation. Hence, a sequential computer built with a chip of such size can provide at most one gigaflops of performance, merely one-thousandth of the speed delivered by the fastest parallel computer available today.

The quest for higher-speed machines has been fueled by computationally intensive problems with profound economical and social impacts referred to as Grand Challenges [5]. Typically, such problems include:

- High-resolution weather forecasting crucial for agriculture, disaster pre-

vention, etc.

- Pollution studies that include cross-pollutant interactions, important in environmental protection.
- Global modeling of atmosphere-ocean-biosphere interactions to measure the long-term impact of human activities on the stability of the global ecosystem.
- Human genome sequencing that will assist in recognizing, preventing and fighting genetic diseases.
- The design of new and more efficient drugs to cure cancer, AIDS and other diseases.
- High-temperature superconductor design that can revolutionize computer design, electrical devices, etc.
- The aerodynamic design of aerospace vehicles (airflow modeling) and improvements in automotive engine design (ignition and combustion modeling) that can lead to more efficient use of depletable fossil fuels in transportation.
- The design of quantum switching devices important for building more powerful computers.

It is estimated that achieving interactive response time for Grand Challenge problems, in the order of minutes for smaller instances and hours for larger ones, will require a machine with performance measured in teraflops, i.e., in thousands of billion floating point operations per second. Today's largest machines can achieve this kind of performance on a limited range of algorithms for very large, highly localized, finely tuned, and often simplified applications. The real drawback is in the software and programmer's ability to find enough useful parallelism in an application to utilize all of the processors of a parallel computer most of the time. Yet, parallel processing is the only feasible option for sustained growth in computer performance in view of the inevitable slowdown in the rate of improvements in the semiconductor industry mentioned earlier.

2 Speedup, efficiency and scalability of parallel computation

Let $s(p)$ denotes the *speedup* of a program when executed on p processors and $t(i)$ be the execution time of this program on an i processor machine. Then,

$$s(p) = \frac{t(1)}{t(p)}. \quad (1)$$

The *parallel efficiency* is defined as

$$e(p) = \frac{s(p)}{p}. \quad (2)$$

The speedup is linear when $s(p) = p$ (hence, the efficiency is 1), superlinear when $s(p) > p$, and sublinear (the most common case) when $s(p) < p$. Superlinear speedup is rarely encountered and it usually can be attributed to the larger total cache and main memory of a parallel machine than the cache and memory of the sequential machine used to measure $t(1)$. An interesting case of superlinearity arises in some combinatorial searches when knowledge of a bound for the sought value speeds up processing branches of the search tree. The parallel algorithm explores several branches concurrently and the bounds found in them speed up the search for all processors, whereas a sequential search may find these bounds later in the execution so branches processed earlier cannot benefit from them. Of course parallel execution can be emulated on a sequential machine but then the required memory will have to contain all the investigated branches at once, likely exceeding the memory size of the sequential computer. In the following discussion we will focus on the typical case of the sublinear speedup.

In Equations (1) and (2), we assumed the constant problem size. As in theory of algorithms, we are also interested in speedup and efficiency as a function of the problem size N . The corresponding definitions than become

$$S(N, p) = \frac{T(N, 1)}{T(N, p)} \quad \text{and} \quad E(N, p) = \frac{S(N, p)}{p}, \quad (3)$$

where $S(N, p)$, $T(N, p)$ and $E(N, p)$ are speedup, execution time and parallel efficiency of a program with the problem size N and executed on a p processor machine.

Similarly as in theory of algorithms, we often interested not only in the exact functions expressing S and E in terms of N or p but also in their $O()$ order. $O()$ notation can be formally defined as:

$$f(x) \in O(g(x)) \text{ iff } (\exists c, X : (\forall x > X : f(x) < c * g(x))).$$

In other words, there is such a constant c that for all sufficiently large values of arguments (bigger than X), value of $f(x)$ is smaller than $c * g(x)$. Instead of saying that $f(x) \in O(g(x))$ we often write $f(x) = O(g(x))$. For example, $f(x) = 0.0000001 * x^2 + x$ belongs to $O(x^2)$ (and also $f(x) = O(x^3)$, etc.) but not to $O(x)$, even though for small x , $f(x) \approx x$.

By definition, efficiency measures how well an investment in additional hardware (cost of p processors) pays off. For linear speedup, efficiency is 1, meaning that 100% of power of newly added processors contributes directly to the solution, an ideal case. In practice, adding processors introduces overheads such as load balance (the problem has to be divided between larger number of participating processors, so the partitioning may become uneven), synchronization (more processors must be synchronized), and communication (the partitions are smaller so more data must be exchanged between larger number of processors). Hence, the typical case is the sublinear speedup with efficiency $E(N, p) < 1$.

Since efficiency is a function of two arguments, we may consider how it behaves when one of them is fixed. If we set a problem size to a constant, say N_c , then clearly when the number of processors becomes large enough, some of them would be idle and the efficiency would suffer. In particular, $E(N_c, p) < \frac{T(N_c, 1)}{p}$ and $T(N_c, 1)$ is a constant, therefore efficiency asymptotically decreases to 0 when the number of processors increases to infinity. This very simple analysis indicates that for a fixed size problem there is a limit how much parallelism can be profitably applied. Kuck [11] defines several efficiency limits that can be used to quantify what "profitably" in this context means. The most modest limit is that at which $S(N_c, p) > S(N_c, p + 1)$ (or equivalently $E(N_c, p) > \frac{pE(N_c, p+1)}{p+1}$). More restrictive is the requirement that $E(N_c, p) > \frac{1}{2 \log_2 p}$, and the highest limit corresponds to the demand that $E(N_c, p) > \frac{p}{2}$. We will refer to these limits as *threshold*, *intermediate* and *high performance* limits, respectively. There is also the corresponding bound on the number of processors that can be used at each level of efficiency. These bounds are denoted as p_t , p_i and p_h , respectively. For scalable computers

Figure 1: Constant problem size speedup

and algorithms, typically $p_t > p_i > p_h \geq 1$ (see Figure 1). Sometimes however, the value of p_t could be smaller than p_i , or even p_h . For example, for an inherently sequential algorithm, $p_t = 1$ and therefore it is no bigger than p_h or p_i .

Very similarly, we can consider the efficiency function for a fixed number of processors, p_c (see Figure 2). For small problem sizes, the speedup raises quickly when the problem size grows. However, even for well-parallelizable problems, the speedup $S_{max} = p_c$ is an asymptotic value, so for large problem sizes there is very little improvement in the speedup when the problem size grows. Similarly as in the previous plot, we can define two boundary values of the problem sizes, N_i which is the minimum size for which the speedup is at least $\frac{p_c}{2 \log_2 p_c}$, and N_h , for which the speedup is $\frac{p_c}{2}$. It should be noted, that for many algorithms the speedup may never reach the high performance limit, regardless of the problem size considered. Moreover, the available memory limits the maximum size of the problem that can be run on a machine with a fixed number of processors. If this limit is exceeded, the performance of the system will degrade significantly. However, the same degradation will be observed on a sequential machine, which usually has less memory than the

parallel machines, so, at least formally, the speedup may not suffer. Nevertheless, the execution time will increase in such a case, and therefore running programs with sizes beyond the memory limit is not practical (see [11] for more detailed discussion of this limit).

Figure 2: Constant processor number speedup

Combining the two presented plots, we obtain the three-dimensional plot of the speedup as a function of the problem size and the number of processors used. However, to normalize this new plot for different algorithms, we will use the execution time on a single processor $T(N, 1)$ as an argument in place of the problem size N . The value $\frac{T(N,1)}{p}$ defines the computational load on each processor, independently of the algorithm used. In Figure 3, we show the constant speedup curves on $p \times T(N, p)$ plane. On this plane, the constant computational workload curves are represented as straight lines originating at the origin of the axes (one such line is drawn in Figure 3). For some unspecified algorithm A, three curves are shown. For each computational load $T(N, 1)$, the threshold performance curve identifies the number of processors that deliver the highest speedup. The intermediate performance curve defines the minimum number of processors, p_i , for which the speedup $S(N_i, p_i) = \frac{p_i}{2^{\log_2 p_i}}$ for the given load $T(N_i, 1)$. The high performance curve

Figure 3: Constant speedup curves

represents the speedup equal to half of the number of processors used. As shown, typically high performance region is the smallest, contained within intermediate performance region which in turn is within threshold performance region. For the linear speedup, the constant performance curve is a straight line. Most often, however, such curves are convex, showing that the performance benefit of adding an additional processor to the system is decreasing with the size of the system.

How large the high performance region is depends both on the architecture and the algorithm. In Figure 3, algorithm B is less amenable to parallelization than algorithm A because its high performance curve is within the high performance curve of algorithm A. However, these curves can also represent performance of the same algorithm on two different architectures, in which case the curve B corresponds to less scalable architecture. Hence, intuitively, scalability of a computer describes its ability to use additional computing power (more processors) to solve larger problems fast. In other words, scalability measures how the number of processors used must increase to keep the solution time constant or small when the problem size increases.

Definition 1: A computer is scalable for the range of processors $[p_{smin}, p_{smax}]$ and the range of problem sizes $[N_{smin}, N_{smax}]$ if and only if $\forall p \in [p_{smin}, p_{smax}], N \in [N_{smin}, N_{smax}] : S_{lim}(p) \leq S(N, p) \leq S(N, p+1)$, where $S_{lim}(p)$ is either $\frac{p}{2}$ or $\frac{p}{2 \log_2 p}$.

The size of scalability region is a measure of scalability of an architecture and/or algorithm.

To illustrate these notions on a concrete algorithm, let's consider a simple *addition problem* which requires summing N numbers using a p processor machine. The algorithm is very simple: each processor first adds $\frac{N}{p}$ numbers in as many steps and then all processors sum their sub-results to find the answer. The first architecture that we consider has a ring interconnection which requires $p - 1$ steps to add the sub-results (each steps involves one

Figure 4: Tree and ring interconnections

addition and a communication of one value). The second one uses a tree interconnection, so the addition of sub-results takes about $2 \log_2 p$ steps (on each level of tree, except the bottom, there are two communication steps, each sending a value and two addition steps, see Figure 4). Hence, the execution time, speedup and efficiency in these two cases can be expressed as:

$$T_t(N, p) = O\left(\frac{N}{p} + \log_2 p\right) \text{ and } T_r(N, p) = O\left(\frac{N}{p} + p\right).$$

$$S_t(N, p) = O\left(\frac{p}{1 + \frac{p \log_2 p}{N}}\right) \text{ and } S_r(N, p) = O\left(\frac{p}{1 + \frac{p^2}{N}}\right).$$

$$E_t(N, p) = O\left(\frac{1}{1 + \frac{p \log_2 p}{N}}\right) \text{ and } E_r(N, p) = O\left(\frac{1}{1 + \frac{p^2}{N}}\right).$$

Setting the constant problem size to N_c yields a plot of the speedup as a function of the number of processors:

$$S_t(N_c, p) = \frac{p}{1 + \frac{p \log_2 p}{N_c}} \text{ and } S_r(N_c, p) = \frac{p}{1 + \frac{p^2}{N_c}}.$$

As seen from Figure 5, the speedups have unique maxima, which are achieved

Figure 5: Constant problem size speedup for addition algorithm

at:

$$p_{t,max} = N_c \ln 2 \text{ with the corresponding speedup } S_{t,max}(N_c) \approx \frac{N_c}{\log_2 N_c}.$$

$p_{r,max} = \sqrt{N_c}$ with the corresponding speedup $S_{r,max}(N_c) = \frac{\sqrt{N_c}}{2} = \frac{p_{r,max}}{2}$.

For the ring architecture, the high performance level is obtained only at the optimum number of processors. The tree architecture reaches the high performance level for:

$$p_{t,h} = \frac{N_c}{\log_2 N_c}.$$

The numbers of processors for which intermediate performance is achieved for these architectures are defined as:

$$p_{t,i} \approx 2N_c \text{ and } p_{r,i} \approx \sqrt{N_c \log_2 N_c}.$$

It should be noted that tree architecture delivers much better performance for the given problem size than ring architecture. The fastest computation for the given problem size N_c is $\log_2 N_c$ for tree architecture and $2\sqrt{N_c}$ for ring architecture.

Figure 6: Constant speedup curves for addition algorithm

Plots of the constant speedup curves for this algorithm are shown in Figure 6. These plots were obtained from the general formula that defines the problem size N needed to achieve performance S with p processors:

$$N_t = \frac{S * p \log_2 p}{p - S} \text{ and } N_r = \frac{S * p^2}{p - S}. \quad (4)$$

It is easy to find the equations defining the high and intermediate performance limits from Equation (4):

$$N_{t,h} = p \log_2 p; \quad N_{t,i} = \frac{p \log_2 p}{2 \log_2 p - 1} \approx \frac{p}{2}$$

and

$$N_{r,h} = p^2; \quad N_{r,i} = \frac{p^2}{2 \log_2 p - 1} \approx \frac{p}{2 \log_2 p}$$

Hence, the region of scalability of the tree architecture is much larger than the ring architecture, demonstrating that the former is more scalable than the latter.

3 Scalability obstacles

The speedup that can be achieved by an application is limited by the following two parallel programming needs:

1. to synchronize processors during parallel execution, and
2. to distribute data and programs to processors before and during execution.

Synchronization introduces two kinds of overheads:

- *load imbalance* overhead that keeps some processors idle waiting for others to reach a synchronization point, and
- *communication overhead* associated with sending and receiving synchronization signals.

Load imbalance appears in several forms. First, for any given problem size N_c there is a limit into how many pieces a program of this size can be divided. This divisibility limit is rarely encountered in practice because the required number of processor is extremely large for large problems and it is clear that running small problems on a machine with a large number of processors cannot be effective. For example, in the addition problem presented in the previous section, the divisibility limit is N_c . For $p = N_c$ all addition operations involve sending arguments between processors. Using more processors than N_c leaves some of them without unique data and therefore idle.

The second form of load imbalance arises when the load does not divide evenly among processors. The best balance that the load distribution can achieve for a computational step of $T(N_c, 1)$ complexity is to execute it in $\lceil \frac{T(N_c, 1)}{p} \rceil$ time which is of course longer than $\frac{T(N_c, 1)}{p}$, but often the difference is negligible. For example, using $N_c - 1$ processors in the addition problem will force all processors (except one) to idle for one step during which the only processor with two numbers assigned to it adds them up.

The third form of load imbalance arises when at several points of program execution the number of parallel pieces that can be created is smaller than the number of available processors. The most common case of this situation happens when there are fragments of code that must be executed sequentially. This case is govern by the Amdahl's Law [2] which describes how the non-parallelizable part of the code limits the speedup. More precisely, if f is the fraction of the sequential execution time spent on code which is inherently sequential (so called the Amdahl's fraction) then:

$$s(p) = \frac{p}{(p - 1) * f + 1}. \quad (5)$$

Hence, independently of the number of processors used $s(p) \leq \frac{1}{f}$, (simply because $f * t(1) \leq t(p)$).

The Amdahl's Law can be generalized by introducing a sequence of fractions f_1, f_2, \dots, f_{p-1} , where f_i denotes a fraction of the execution time devoted to a fragment of code which can be split into i parallel task. Then, the speedup is

$$s(p) = \frac{p}{\sum_{i=1}^{p-1} f_i * (p/i - 1) + 1}. \quad (6)$$

Figure 7: Impact of the Amdahl's Law on the maximum speedup

The Amdahl's Law seems very pessimistic; after all, every program has sequential parts and even if these parts are small and limited to few percent of the code, still the speedup is limited to less than hundred times (see Figure 7).

Fortunately, often the execution time of sequential parts of the algorithm do not change, or change slowly with the growth of the problem size, whereas execution time of parallelizable parts changes rapidly when the problem size is increased. Hence, the Amdahl's fraction is dependent on the problem size and as such, it could be written as $f(N)$. For a wide class of problems $f(N)$ can be made arbitrary small by selecting sufficiently large problem size. Consequently, for such problems, the speedup can be made arbitrary large, showing that the algorithm is scalable. More formally, this requirement can be stated as follows.

s

Definition 2: An algorithm is computationally scalable if

$$\forall p > 1 : \lim_{N \rightarrow \infty} f_p(N) = 0. \quad (7)$$

Definition 2 uses the generalized Amdahl's Law (compare Equation 6) and it simplifies to the requirement:

$$\lim_{N \rightarrow \infty} f(N) = 0.$$

for the original Amdahl's fraction $f(N)$.

Often the problems computed on parallel machines are too large to fit on a uniprocessor, so measuring the Amdahl's fraction for them is either impossible or difficult. John Gustafson [8] proposed different measure, A , that represents a fraction of parallel execution time during which the parallel machine executed the sequential part of the code. Therefore the speedup is:

$$s(p) = p - (p - 1) * A. \quad (8)$$

The nice feature of this formula is that it clearly shows how to improve

Figure 8: The Amdahl's and Berstis's fractions for the same program

the speedup. If we start adding processors (i.e., increasing p) but keep the work of all processors the same, then most likely A will stay the same and the speedup will grow. Likewise, with the constant number of processors, we decrease A by increasing the problem size. The final conclusion is similar to what the Amdahl's Law implies: by selecting a problem large enough to keep

all processors occupied for a long time, the impact of the sequential parts of the program could be made negligible.

It should be noted, that the Amdahl's and Berstis's fractions are different even for the same program. Consider a program that requires initial setup and post-processing cleanup steps that must be done sequentially (see Figure 8) and each requires one time unit. The parallelizable part of the program requires six time units. On a sequential machine, the Amdahl's fraction, f , is equal to $2/8 = 0.25$, so the speedup on a six processor machine is then (see Equation 5) $s(p) = 8/3$. However, during parallel execution two time units are spent on sequential execution, so the Berstis's fraction is $A = 2/3 \neq f$ but the speedup is the same, $8/3$ according to Equation 8.

There is yet another form of load imbalance that is very difficult to deal with. This is a load imbalance that arises at run time in response to evolving computation. To eliminate it, the program must change data and task allocation to processors during execution, which increases execution time. The program must also devote part of its execution time to monitoring load balance and to finding new data and task distributions. Monitoring, finding new distribution and moving data and tasks, all contribute to the overhead which decreases speedup and efficiency of parallel computation. These issues are still unresolved and they are the subject of on-going research (for example, see [3]).

Communication overhead associated with sending and receiving synchronization signals is of the same nature as communication overhead arising during distribution of data and tasks to parallel processors. The initial distribution cost is usually a small percentage of the overall computation costs because it is proportional to the data size N , whereas the computation is much faster growing function of N , often in $O(N^2)$ or higher. Hence, we will concentrate on assessing communication overhead incurred during parallel execution. This overhead is dependent on the volume of data communicated, its granularity and on the properties of interconnecting network of a parallel machine.

Similarly like in case of the Amdahl's fraction, there is a large body of algorithms that have the amount of computation growing faster than the volume of data that need to be communicated. A very important subset of such algorithms include scientific computation that evaluate a function at many points of a physical domain. In such computation the volume of data to be communicated is proportional to the surface of a subdomain assigned

to a single processor. The computation is proportional to the volume of each subdomain. Hence, for two-dimensional and three-dimensional domains, the communication to computation ratios $r(N)$ are

$$r_{2D}(N) = O\left(\frac{1}{\sqrt{T}(N, p_c)}\right) \text{ and } r_{3D}(N) = O(T(N, p_c)^{-1/3}).$$

Although $r_{3D}(N)$ decreases slower than $r_{2D}(N)$, the decrease is still significant and converges to 0 when problem size grows to infinity.

s

Definition 3: An algorithm is *efficiently parallelizable* if its communication to computation ratio $r(N)$ satisfies the condition:

$$\lim_{N \rightarrow \infty} r(N) = 0.$$

s

Conclusion: A computation is *scalable* if an efficiently parallelizable and computationally scalable algorithm is run on an architecture with a scalable interconnection network.

The scalability of parallel architectures is discussed in the following section, here we discuss scalability of interconnection networks. The basic problem with the networks is that they have either scalable cost or scalable performance but not both. On one extreme are scalable performance networks, such as the fully interconnected network or a crossbar switch. In such networks, the distance between any two components is independent of the number of interconnected components. However, the cost for such networks grows proportionally to the square of the number of interconnected components. Hence, the machine with p processors will cost $p + c * p^2$, where c is the ratio of the cost of a processor to the cost of wire or a switch. As a result, as the size of the machine increases, the proportion of the money spent on the computing power p decreases as $\frac{1}{1+cp}$ making the computation more and more expensive. On the other hand, the cost scalable solution, for example a ring or a tree, provides the diminishing network performance with the growth of the number of components because the total bandwidth for the data communication is constant and shared among all processors.

The solutions used most commonly today are located between these two extremes, using interconnections such as a hypercube, a switching network with $\log_2 p$ stages or a fat tree [14]. In a hypercube, processors are arranged into a $\log_2 p$ -dimensional cube, each having $\log_2 p$ direct neighbors (see Figure 9). Switching networks have p switches at each stage and shuffle messages at each switch, effectively emulating the fully interconnected network. Finally, fat trees provide higher level nodes with multiple links to accommodate higher communication traffic flowing through them (see Figure 9). All these interconnection networks incur the cost of $O(p \log_2 p)$ for connecting p components. The distance between any two components is $O(\log_2 p)$. As a result, both the communication overhead and the ratio of interconnection to the total cost of the machine grow slowly with the size of the parallel machine.

Figure 9: Scalable interconnection networks

4 Classification of parallel machines and limitations of their scalability

The basic classification of parallel machines was proposed by Flynn [7] who characterized classes of architectures according to their instruction and data streams. Sequential machines are Single Instruction Single Data class, or SISD in short. Very Large Instruction Word (VLIW) machines [6] are Multiple Instruction Single Data (MISD) because their instruction word contains several ordinary instructions and their processors include several copies of functional units that execute these instructions concurrently. VLIW machines rely on a compiler to pack several instructions into one word. However, the code produced by the compilers of high level languages contains only short sequences of instructions uninterrupted by conditional jumps. Therefore a complex program analysis, called *trace scheduling* [4] is needed to support even modest parallelism of the order of several to a few dozen. Hence, MISD architectures are inherently non-scalable.

The next class of parallel machines is characterized by a Single Instruction and Multiple Data streams (SIMD). The same instruction is executed by all processors, each working on different data. For limited class of algorithms, including scientific computations with a compile-time defined data structures, these machines are scalable. Their limitations are discussed below (see also [9]).

The final class of machines processes Multiple Instruction and Multiple Data (MIMD) streams. They come in two flavors: either with memory shared among all processors or memory distributed to exclusive use of individual processors. Shared memory machines are easy to program because they directly support global address space for parallel programs. However, modern processors are equipped with caches and maintaining cache coherence is expensive and limits scalability of these architectures.

In distributed memory machines the programmer is explicitly responsible for distributing data and tasks to each processor and for moving data around the processors during execution. The data movements are encoded as the communication primitives that either are a part of communication library (such as Message Passing Interface (MPI) Library [12]), or are embedded directly into the programming language [14]. As discussed in the

previous section, providing a scalable interconnection network is a challenge, but in general MIMD distributed memory machines are considered the most scalable.

To reap the benefits of both shared memory simple programming model owing to global address space and scalability supported by distributed memory, several modern architectures support the shared memory model via distributed memory hardware [10]. Such an approach was encouraged by increasing similarities between shared memory architectures with large local caches and the distributed memory machines. The scalability and performance of these machines are still not fully understood but more evaluations and research on these topics is under way.

The issue of scalability cannot be considered without looking at parallel programming paradigms. The increasing importance of parallel processing prompted growth in the body of standardization in parallel programming languages and tools. Yet, there is no evidence of convergence of the supported programming paradigms to a single model. Currently there are two most popular models for parallel program design: data parallelism and message passing.

Data parallelism is popular because of its simplicity. In this model, a single program (and therefore a single thread of execution) is replicated on many processors and each copy operates on a separate part of data. Depending on the tightness with which the execution of programs is synchronized, there are two modes of using data parallelism. When each instruction of the program is synchronically executed on all processors, then Single Instruction Multiple Data (SIMD) mode is used. Such tight synchronization requires hardware support.

From the software engineering point of view, SIMD machines are easy to program because there is a single flow of control on all processors. The main focus of parallelization is to find large data structures that can be distributed to all processors to keep them all occupied. Another concern is to minimize the data movements necessary to provide data to processors that are to execute them. Due to the small granule of parallelism (single instruction) SIMD machines consist of a very large number of simple processors (tens or hundred thousand of processors in a single machine is not unusual). Each of these processors must either execute the same statement as all the others or idle, so SIMD machines achieve poor efficiency on programs that do not contain sufficiently large data structures. They also do not perform well

on programs which require irregular data references (list structures, dynamic memory, etc.). The consensus is that SIMD architecture has a very specialized niche of applications (e.g., visual information and scene processing), but it is not the best choice for general parallel processing.

Data parallelism can be also used in a loosely synchronized mode, when the program execution consists of two stages:

1. computational stage, when copies of the same program are executed in parallel on each processor locally. The execution can differ in the conditional branches taken, number of loop iteration executed, etc.,
2. data exchange stage, when all processors concurrently engage in exchanging non-local data.

It should be noted that the data exchange stage is very simple in case of shared memory machines (when it can be enforced by use of locks or barriers). The frequency of synchronization in SPMD model can be adjusted to correspond to the latency of the interconnection network. The SPMD model is very adequate for scientific computing which often requires applying basically the same algorithm at many points of computational domain. SPMD parallel programs are conceptually simple because the same program executes on all processors, but they are more complex than SIMD programs.

For more complex applications, running a single program across the parallel machine may be unnecessarily restrictive. In particular, dynamically changing programs with unpredictable execution times result in poorly balanced parallel computations when implemented in SPMD mode. This is because in SPMD mode, processors synchronize at the data exchange stage, and none of the processors can proceed to the next computational stage until all others reach the data exchange stage.

The SPMD model was abstracted into a Bulk-Synchronous Parallelism model [15]. This model provides an abstraction of parallel algorithm description that lends itself to performance and scalability analysis. The model also became a basis for a library that facilitates the creation of portable parallel software [13].

The BSP model consists of three components:

1. Processors that execute sequential code.

2. A router which provides point to point communication between pairs of processors.
3. A synchronization mechanism that synchronizes all or a subset of the processors at selected points of execution.

In the BSP model, computation consists of a sequence of supersteps. In each superstep, a processor performs some local computation and transmits messages to other processors. After each superstep, all processors synchronize and the next superstep starts. In the BSP model, the data transmitted are not guaranteed to be available at the destination until after the end of the superstep at which they were sent.

A BSP computer is characterized by the four parameters: number of processors p , processor speed s , synchronization periodicity L , and ratio of the global computation to communication speed g . The synchronization periodicity L is the smallest number of processor operations that can be executed between successive synchronization operations. The ratio r is equal to the total number of local operations performed by all processors in a time unit divided by the total number of words delivered by the communication network in the same time unit. Processor speed s is measured in flops (floating point operations per second). Synchronization parameter, L , is measured in floating point operations and ratio g is measured in floating point operations per word. Values of parameters L and g depend on the number of processors p , so we will often write them as functions $L(p), g(p)$.

BSP parameters allow for algorithm performance and scalability analysis. For example, consider a superstep that needs to communicate h words of data. Since it takes $g * h$ time units for the communication network to deliver the data to its destination and L units to synchronize all the processors performing the superstep, at least $L + g * h$ units of computation are needed to keep the processors busy; an amount of computation less than this threshold would result in idling of some processor, and therefore would be a source of inefficiency.

In terms of the BSP parameters, parallel machines are often characterized by large values of s , owing to their fast processors, and low values of L and g thanks to communication links with low latency and large bandwidth. A general purpose network of workstations, on the other hand, is characterized by values of s that are somewhat lower than for the parallel machines and values of L and g that are much larger than the corresponding values

for the parallel machines because of high latency and low bandwidth of its communication. The importance of interconnection networks for architecture scalability can be seen from the difference in dependence of L and g on the number of interconnected components. The asymptotic values of BSP parameters for various architectures are shown in Figure 10. The ideal values, $L = g = O(1)$ can be achieved only on an idealized PRAM (Parallel Random Access Machine) which has an unattainable scalable shared memory. Thanks to this memory, PRAM can execute communication and synchronization in one instruction cycle. However, modern networks using optical links and hypercube or staged switches are relatively close to this ideal.

Figure 10: BSP parameters for selected architectures

To illustrate scalability analysis using BSP consider once again the addition problem introduced in Section 2. There is one superstep with N/p computational steps followed by $\log_2 p$ supersteps, each requiring one-word communication and one addition. Hence, the BSP cost of the algorithm is

$$O(N/p + (g + L + 1) \log_2 p).$$

To keep the other steps of the same order as the first one, the number of

processors used must satisfy the following inequalities:

$$\log_2 p \leq N/p \text{ so } p \in O\left(\frac{N}{\log_2 N}\right),$$

$$(L + g) \log_2 p \leq N/p \text{ so } pg(p) \log_2 p, pL(p) \log_2 p \in O(N).$$

For the tree architecture, $L(p) = g(p) = O(\log_2 p)$, therefore $p \in O\left(\frac{N}{\log_2^2 N}\right)$. However, for ring architecture $L(p) = g(p) = O(p)$ and, consequently, $p \in O\left(\sqrt{\frac{N}{\log_2 N}}\right)$. Hence, as we found it earlier in Section 2, the tree architecture is more scalable than the ring one and more processors can be used efficiently for the given problem size by the former than the latter.

Similar analysis can be used to analyse the different algorithms to find out under what conditions one is better than the other. A somewhat simplified example of this kind of analysis would be to consider another algorithm for the addition problem in which all processors send their sub-results to one processor which then does the final additions. Hence, there are just two supersteps and the cost is $O(N/p + p + L + gp)$, because the processor receiving the partial sums must read them in sequentially. Hence, the restrictions for the number of processors yield: $p \in O(\sqrt{N})$ and $pL(p), p^2g(p) \in O(N)$. For ring architectures the corresponding limit is $p \in N^{1/3}$. Comparing it with the limit for the first algorithm, we find that when $\log_2^3 N > N$ (which is true for $N < 16$), the second algorithm is better than the first.

The BSP analysis indicates that the most scalable parallel architecture would have the functions $g(p), L(p)$ as close as possible to $O(1)$. As seen in Figure 10, the modern architectures move towards this goal.

5 Conclusion and Future Trends

There is a clear trend towards widening the base of parallel processing both in hardware and software. On the hardware side, that means using off-the-shelf commercially available components (processor, interconnection switches) which benefit from rapid pace of technological advancement fueled by the large customer base. The other effect is the convergence of different architectures thanks to spreading the successful solutions among all of them. Workstations interconnected by a fast network approach the performance of parallel machines. Shared memory machines with multilevel caches and sophisticated

prefetching strategies execute programs with efficiency similar to distributed memory machines.

On the software side, the widening the base of the users currently relies on standardization of parallel programming tools. By protecting the programmer's investment in software, standardization promotes development of libraries, tools and application kits that in turn will attract more end-users to parallel processing. Standardization supports scalable solutions that can be applied across different architectures and configurations of parallel machines, depending on the data size of the problem instant at hand. These trends increase importance of predictable performance and scalability analysis of both algorithms and architectures.

References

- [1] George S. Almasi and Allan Gottlieb, *Highly Parallel Computing*, Benjamin Cummings, Redwood City, CA, 1994.
- [2] G.M. Amdahl, "The Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," *AFIPS Proc. Summer Joint Computer Conference*, vol. 30, 1967.
- [3] C. Bottasso et al., "The Quality of Partitions by an Iterative Load Balancer," *Languages, Compilers and Run-Time Systems for Scalable Computers*, B.K. Szymanski (editor), Kluwer Academic Publishers, Reading MA, pp. 265-278, 1996.
- [4] John Ellis, *Bulldog: A Compiler for VLIW Architecture*, MIT Press, Boston, MA, 1985.
- [5] *Evolving High Performance Computing and Communications Initiative to Support the Nations's Information Infrastructure*, National Research Council, National Academy Press, Washington, D.C. 1995.
- [6] Joseph A. Fisher and John J. O'Donnell, "VLIW Machines: Multiprocessors We Can Actually Program," *COMPCON Proc.*, pp. 299-305, 1984.

- [7] Michael J. Flynn, "Some Computer Organizations and Their Effectiveness," *IEEE Transactions on Computers*, vol. C-21, pp. 948-960, 1972.
- [8] John L. Gustafson, "Reevaluating Amdahl's Law," *Communications of the ACM*, vol. 31, no. 5, pp. 532-3, 1988.
- [9] John Hennessy and David A. Patterson, *Computer Architectures: A Quantitative Approach*, Morgan Kaufman Publishers, San Mateo, CA, 1990.
- [10] Kai Hwang, *Advanced Computer Architectures: Parallelism, Scalability, Programmability*, McGraw-Hill, New York, 1993.
- [11] David J. Kuck, *High Performance Computing*, Oxford Press, New York, NY, 1996.
- [12] Peter S. Pacccho, *Parallel Programming with MPI*, Morgan Kaufman, San Mateo, CA, 1997.
- [13] Richard Miller and J.L. Reed, *The Oxford BSP Library: Users' Guide*, Version 1.0, Oxford Parallel Technical Report, Oxford University, Oxford, UK, 1994.
- [14] Michael Quinn, *Parallel Computing, Theory and Practice*, McGraw-Hill, New York, NY, 1994.
- [15] Leslie Valiant, "A bridging model for parallel computation," *Communication of the ACM*, vol. 33, no. 8, pp. 103-11, 1990.