3

Principles of Scalable Performance

We study performance measures, speedup laws, and scalability principles in this chapter. Three speedup models are presented under different computing objectives and resource constraints. These include Amdahl's law (1967), Gustafson's scaled speedup (1988), and the memory-bounded speedup by Sun and Ni (1993).

The efficiency, redundancy, utilization, and quality of a parallel computation are defined, involving the interplay between architectures and algorithms. Standard performance measures and several benchmark kernels are introduced with relevant performance data.

The performance of parallel computers relies on a design that balances hardware and software. The system architects and programmers must exploit parallelism, pipelining, and networking in a balanced approach. Toward building massively parallel systems, the scalability issues must be resolved first. Fundamental concepts of scalable systems are introduced in this chapter. Case studies can be found in subsequent chapters, especially in Chapters 9 and 13.

3.1 PERFORMANCE METRICS AND MEASURES

In this section, we first study parallelism profiles and define the asymptotic speedup factor, ignoring communication latency and resource limitations. Then we introduce the concepts of system efficiency, utilization, redundancy, and quality of parallel computations. Possible tradeoffs among these performance metrics are examined in the context of cost-effectiveness. Several commonly used performance measures, MIPS, Mflops, and TPS, are formally defined.

3.1.1 Parallelism Profile in Programs

The degree of parallelism reflects the extent to which software parallelism matches hardware parallelism. We characterize below parallelism profiles, introduce the concept of average parallelism, and define an ideal speedup with infinite machine resources. Variations on the ideal speedup factor will be presented in subsequent sections from various application viewpoints and under different system limitations.

Degree of Parallelism The execution of a program on a parallel computer may use different numbers of processors at different time periods during the execution cycle. For each time period, the number of processors used to execute a program is defined as the *degree of parallelism* (DOP). This is a discrete time function, assuming only nonnegative integer values.

The plot of the DOP as a function of time is called the *parallelism profile* of a given program. For simplicity, we concentrate on the analysis of single-program profiles. Some software tools are available to trace the parallelism profile. The profiling of multiple programs in an interleaved fashion can in theory be extended from this study.

Fluctuation of the profile during an observation period depends on the algorithmic structure, program optimization, resource utilization, and run-time conditions of a computer system. The DOP was defined under the assumption of having an unbounded number of available processors and other necessary resources. The DOP may not always be achievable on a real computer with limited resources.

When the DOP exceeds the maximum number of available processors in a system, some parallel branches must be executed in chunks sequentially. However, parallelism still exists within each chunk, limited by the machine size. The DOP may be also limited by memory and by other nonprocessor resources. We consider only the limit imposed by processors in our discussions on speedup models.

Average Parallelism In what follows, we consider a parallel computer consisting of n homogeneous processors. The maximum parallelism in a profile is m. In the ideal case, $n >> m$. The *computing capacity* Δ of a single processor is approximated by the execution rate, such as MIPS or Mflops, without considering the penalties from memory access, communication latency, or system overhead. When i processors are busy during an observation period, we have DOP = i.

The total amount of work W (instructions or computations) performed is proportional to the area under the profile curve:

$$W = \Delta \int_{t_1}^{t_2} DOP(t)\, dt \tag{3.1}$$

This integral is often computed with the following discrete summation:

$$W = \Delta \sum_{i=1}^{m} i \cdot t_i \tag{3.2}$$

where t_i is the total amount of time that DOP = i and $\sum_{i=1}^{m} t_i = t_2 - t_1$ is the total elapsed time.

The *average parallelism A* is computed by

$$A = \frac{1}{t_2 - t_1} \int_{t_1}^{t_2} DOP(t)\, dt \tag{3.3}$$

In discrete form, we have

$$A = \left(\sum_{i=1}^{m} i \cdot t_i \right) \Big/ \left(\sum_{i=1}^{m} t_i \right) \tag{3.4}$$

Example 3.1 Parallelism profile and average parallelism of a divide-and-conquer algorithm (Sun and Ni, 1993)

As illustrated in Fig. 3.1, the parallelism profile of a divide-and-conquer algorithm increases from 1 to its peak value m = 8 and then decreases to 0 during the observation period (t_1, t_2).

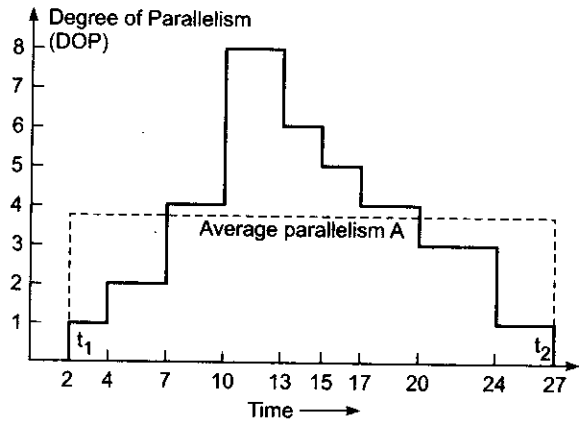


Fig. 3.1 Parallelism profile of a divide-and-conquer algorithm

In Fig. 3.1, the average parallelism $A = (1 \times 5 + 2 \times 3 + 3 \times 4 + 4 \times 6 + 5 \times 2 + 6 \times 2 + 8 \times 3)/(5 + 3 + 4 + 6 + 2 + 2 + 3) = 93/25 = 3.72$. In fact, the total workload $W = A\Delta\,(t_2 - t_1)$, and A is an upper bound of the asymptotic speedup to be defined below.

Available Parallelism There is a wide range of potential parallelism in application programs. Engineering and scientific codes exhibit a high DOP due to data parallelism. Manoj Kumar (1988) has reported that computation-intensive codes may execute 500 to 3500 arithmetic operations concurrently in each clock cycle in an idealized environment. Nicolau and Fisher (1984) reported that standard Fortran programs averaged about a factor of 90 parallelism available for very-long-instruction word architectures. These numbers show the optimistic side of available parallelism.

However, David Wall (1991) indicated that limits of instruction-level parallelism is around 5, rarely exceeding 7. Bulter et al. (1991) reported that when all constraints are removed, the DOP in programs may exceed 17 instructions per .cycle. If the hardware is perfectly balanced, one can sustain from 2.0 to 5.8 instructions per cycle on a superscalar processor that is reasonably designed. These numbers show the pessimistic side of available parallelism.

The above measures of available parallelism show that computation that is less numeric than that in scientific codes has relatively little parallelism even when basic block boundaries are ignored. A *basic block* is a sequence or block of instructions in a program that has a single entry and a single exit points. While compiler optimization and algorithm redesign may increase the available parallelism in an application, limiting parallelism extraction to a basic block limits the potential instruction-level parallelism to a factor of about 2 to 5 in ordinary programs. However, the DOP may be pushed to thousands in some scientific codes when multiple processors are used to exploit parallelism beyond the boundary of basic blocks.

Asymptotic Speedup Denote the amount of work executed with DOP = i as $W_i = i\Delta t_i$ or we can write $W = \sum_{i=1}^{m} W_i$. The execution time of W_i on a single processor (sequentially) is $t_i(1) = W_i/\Delta$. The execution time of W_i on k processors is $t_i(k) = W_i/k\Delta$. With an infinite number of available processors, $t_i(\infty) = W_i/i\Delta$ for $1 \le i \le m$. Thus we can write the *response time* as

$$T(1) = \sum_{i=1}^{m} t_i\ (1) = \sum_{i=1}^{m} \frac{W_i}{\Delta} \tag{3.5}$$

$$T(\infty) = \sum_{i=1}^{m} t_i \quad (\infty) = \sum_{i=1}^{m} \frac{W_i}{i\Delta} \tag{3.6}$$

The *asymptotic speedup* S_∞ is defined as the ratio of $T(1)$ to $T(\infty)$:

$$S_\infty = \frac{T(1)}{T(\infty)} = \frac{\sum_{i=1}^{m} W_i}{\sum_{i=1}^{m} W_i / i} \tag{3.7}$$

Comparing Eqs. 3.4 and 3.7, we realize that $S_\infty = A$ in the ideal case. In general, $S_\infty \leq A$ if communication latency and other system overhead are considered. Note that both S_∞ and A are defined under the assumption $n = \infty$ or $n >> m$.

3.1.2 Mean Performance

Consider a parallel computer with n processors executing m programs in various modes with different performance levels. We want to define the mean performance of such multimode computers. With a weight distribution we can define a meaningful performance expression.

Different execution modes may correspond to scalar, vector, sequential, or parallel processing with different program parts. Each program may be executed with a combination of these modes. *Harmonic mean* performance provides an average performance across a large number of programs running in various modes.

Before we derive the harmonic mean performance expression, let us study the *arithmetic mean* performance expression first derived by James Smith (1988). The execution rate R_i for program i is measured in MIPS rate or Mflops rate, and so are the various performance expressions to be derived below.

Arithmetic Mean Performance Let $\{R_i\}$ be the execution rates of programs $i = 1, 2, \ldots, m$. The *arithmetic mean execution rate* is defined as

$$R_a = \sum_{i=1}^{m} R_i / m \tag{3.8}$$

The expression R_a assumes equal weighting $(1/m)$ on all m programs. If the programs are weighted with a distribution $\pi = \{f_i | i = 1, 2, \ldots, m\}$, we define a *weighted arithmetic mean execution rate* as follows:

$$R_a^* = \sum_{i=1}^{m} (f_i R_i) \tag{3.9}$$

Arithmetic mean execution rate is proportional to the sum of the inverses of execution times; it is not inversely proportional to the sum of execution times. Consequently, the arithmetic mean execution rate fails to represent the real times consumed by the benchmarks when they are actually executed.

Harmonic Mean Performance With the weakness of arithmetic mean performance measure, we need to develop a mean performance expression based on arithmetic mean execution time. In fact, $T_i = 1/R_i$ is the mean execution time per instruction for program i. The *arithmetic mean execution time* per instruction is defined by

$$T_a = \frac{1}{m}\sum_{i=1}^{m} T_i = \frac{1}{m}\sum_{i=1}^{m}\frac{1}{R_i} \tag{3.10}$$

The *harmonic mean execution rate* across m benchmark programs is thus defined by the fact $R_h = 1/T_a$:

$$R_h = \frac{m}{\sum_{i=1}^{m}(1/R_i)} \tag{3.11}$$

Therefore, the harmonic mean performance is indeed related to the average execution time. With a weight distribution $\pi = \{f_i | i = 1, 2, \ldots, m\}$, we can define the *weighted harmonic mean execution rate as*:

$$R_h^* = \frac{1}{\sum_{i=1}^{m}(f_i/R_i)} \tag{3.12}$$

The above harmonic mean performance expressions correspond to the total number of operations divided by the total time. Compared to arithmetic mean, the harmonic mean execution rate is closer to the real performance.

Harmonic Mean Speedup Another way to apply the harmonic mean concept is to tie the various modes of a program to the number of processors used. Suppose a program (or a workload of multiple programs combined) is to be executed on an n-processor system. During the executing period, the program may use $i = 1, 2, \ldots, n$ processors in different time periods.

We say the program is executed in *mode i*, if i processors are used. The corresponding execution rate R_i is used to reflect the collective speed of i processors. Assume that $T_1 = 1/R_1 = 1$ is the sequential execution time on a uniprocessor with an execution rate $R_1 = 1$. Then $T_i = 1/R_i = 1/i$ is the execution time of using i processors with a combined execution rate of $R_i = i$ in the ideal case.

Suppose the given program is executed in n execution modes with a weight distribution $w = \{f_i | i = 1, 2, \ldots, n\}$. A *weighted harmonic mean speedup* is defined as follows:

$$S = T_1/T^* = \frac{1}{(\sum_{i=1}^{n} f_i/R_i)} \tag{3.13}$$

where $T^* = 1/R_h^*$ is the *weighted arithmetic mean execution time* across the n execution modes, similar to that derived in Eq. 3.12.

Example 3.2 Harmonic mean speedup for a multiprocessor operating in *n* execution modes (Hwang and Briggs, 1984)

In Fig. 3.2, we plot Eq. 3.13 based on the assumption that $T_i = 1/i$ for all $i = 1, 2, \ldots, n$. This corresponds to the ideal case in which a unit-time job is done by i processors in minimum time. The assumption can also be interpreted as $R_i = i$ because the execution rate increases i times from $R_1 = 1$ when i processors are fully utilized without waste.

The three probability distributions π_1, π_2, and π_3 correspond to three processor utilization patterns. Let $s = \sum_{i=1}^{n} i$. $\pi_1 = (1/n, 1/n, \ldots, 1/n)$ corresponds to a uniform distribution over the n execution modes, $\pi_2 = (1/s, 2/s, \ldots, n/s)$ favors using more processors, and $\pi_3 = (n/s, (n-1)/s, \ldots, 2/s, 1/s)$ favors using fewer processors.

The ideal case corresponds to the 45° dashed line. Obviously, π_2 produces a higher speedup than π_1 does. The distribution π_1 is superior to the distribution π_3 in Fig. 3.2.

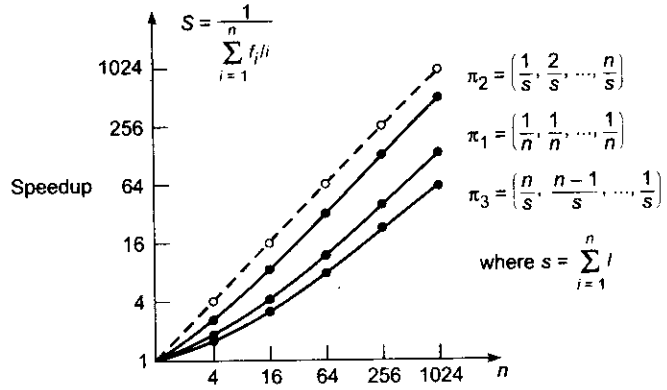


Fig. 3.2 Harmonic mean speedup performance with respect to three probability distributions: π_1 for uniform distribution, π_2 in favor of using more processors, and π_3 in favor of using fewer processors

Amdahl's Law Using Eq. 3.13, one can derive Amdahl's law as follows: First, assume $R_i = i$, $w = (\alpha, 0, 0, \ldots, 0, 1 - \alpha)$; i.e., $w_1 = \alpha$, $w_n = 1 - \alpha$, and $w_i = 0$ for $i \neq 1$ and $i \neq n$. This implies that the system is used either in a pure sequential mode on one processor with a probability α, or in a fully parallel mode using n processors with a probability $1 - \alpha$. Substituting $R_1 = 1$ and $R_n = n$ and w into Eq. 3.13, we obtain the following speedup expression:

$$S_n = \frac{n}{1 + (n-1)\alpha} \tag{3.14}$$

This is known as Amdahl's law. The implication is that $S \to 1/\alpha$ as $n \to \infty$. In other words, under the above probability assumption, the best speedup one can expect is upper-bounded by $1/\alpha$, regardless of how many processors are employed.

In Fig. 3.3, we plot Eq. 3.14 as a function of n for four values of α. When $\alpha = 0$, the ideal speedup is achieved. As the value of α increases from 0.01 to 0.1 to 0.9, the speedup performance drops sharply.

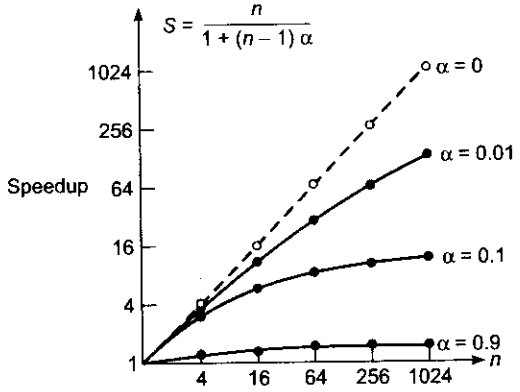


Fig. 3.3 Speedup performance with respect to the probability distribution $\pi = (\alpha, 0, \ldots, 0, 1 - \alpha)$ where α is the fraction of sequential bottleneck

For many years, Amdahl's law has painted a pessimistic picture for parallel processing. That is, the system performance cannot be high as long as the serial fraction α exists. We will further examine Amdahl's law in Section 3.3.1 from the perspective of workload growth.

3.1.3 Efficiency, Utilization, and Quality

Ruby Lee (1980) has defined several parameters for evaluating parallel computations. These are fundamental concepts in parallel processing. Tradeoffs among these performance factors are often encountered in real-life applications.

System Efficiency Let $O(n)$ be the total number of unit operations performed by an n-processor system and $T(n)$ be the execution time in unit time steps. In general, $T(n) < O(n)$ if more than one operation is performed by n processors per unit time, where $n \geq 2$. Assume $T(1) = O(1)$ in a uniprocessor system. The *speedup factor* is defined as

$$S(n) = T(1)/T(n) \tag{3.15}$$

The *system efficiency* for an n-processor system is defined by

$$E(n) = \frac{S(n)}{n} = \frac{T(1)}{nT(n)} \tag{3.16}$$

Efficiency is an indication of the actual degree of speedup performance achieved as compared with the maximum value. Since $1 \leq S(n) \leq n$, we have $1/n \leq E(n) \leq 1$.

The lowest efficiency corresponds to the case of the entire program code being executed sequentially on a single processor, the other processors remaining idle. The maximum efficiency is achieved when all n processors are fully utilized throughout the execution period.

Redundancy and Utilization The *redundancy* in a parallel computation is defined as the ratio of $O(n)$ to $O(1)$:

$$R(n) = O(n)/O(1) \tag{3.17}$$

This ratio signifies the extent of matching between software parallelism and hardware parallelism. Obviously $1 \leq R(n) \leq n$. The *system utilization* in a parallel computation is defined as

$$U(n) = R(n)E(n) = \frac{O(n)}{nT(n)} \tag{3.18}$$

The system utilization indicates the percentage of resources (processors, memories, etc.) that was kept busy during the execution of a parallel program. It is interesting to note the following relationships: $1/n \leq E(n) \leq U(n) \leq 1$ and $1 \leq R(n) \leq 1/E(n) \leq n$.

Quality of Parallelism The *quality* of a parallel computation is directly proportional to the speedup and efficiency and inversely related to the redundancy. Thus, we have

$$Q(n) = \frac{S(n)E(n)}{R(n)} = \frac{T^3(1)}{nT^2(n)O(n)} \tag{3.19}$$

Since $E(n)$ is always a fraction and $R(n)$ is a number between 1 and n, the quality $Q(n)$ is always upper-bounded by the speedup $S(n)$.

Example 3.3 A hypothetical workload and performance plots

In Fig. 3.4, we compare the relative magnitudes of $S(n)$, $E(n)$, $R(n)$, $U(n)$, and $Q(n)$ as a function of machine size n, with respect to a hypothetical workload characterized by $O(1) = T(1) = n^3$, $O(n) = n^3 + n^2 \log_2 n$, and $T(n) = 4n^3/(n + 3)$.

Substituting these measures into Eqs. 3.15 to 3.19, we obtain the following performance expressions:

$$\begin{aligned} S(n) &= (n + 3)/4 \\ E(n) &= (n + 3)/(4n) \\ R(n) &= (n + \log_2 n)/n \\ U(n) &= (n + 3)(n + \log_2 n)/(4n^2) \\ Q(n) &= (n + 3)^2/(16(n + \log_2 n)) \end{aligned}$$

The relationships $1/n \le E(n) \le U(n) \le 1$ and $0 \le Q(n) \le S(n) \le n$ are observed where the linear speedup corresponds to the ideal case of 100% efficiency.

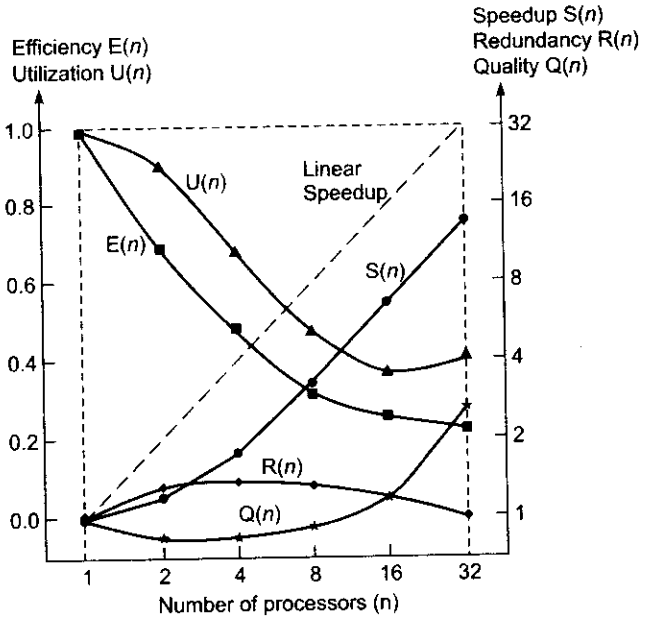


Fig. 3.4 Performance measures for Example 3.3 on a parallel computer with up to 32 processors

To summarize the above discussion on performance indices, we use the speedup $S(n)$ to indicate the degree of speed gain in a parallel computation. The efficiency $E(n)$ measures the useful portion of the total work performed by n processors. The redundancy $R(n)$ measures the extent of workload increase.

The utilization $U(n)$ indicates the extent to which resources are utilized during a parallel computation.

Finally, the quality $Q(n)$ combines the effects of speedup, efficiency, and redundancy into a single expression to assess the relative merit of a parallel computation on a computer system.

The speedup and efficiency of 10 parallel computers are reported in Table 3.1 for solving a linear system of 1000 equations. The table entries are excerpts from Table 2 in Dongarra's report (1992) on LINPACK benchmark performance over a large number of computers.

Either the standard LINPACK algorithm or an algorithm based on matrix-matrix multiplication was used in these experiments. A high degree of parallelism is embedded in these experiments. Thus high efficiency (such as 0.94 for the IBM 3090/6008 VF and 0.95 for the Convex C3240) was achieved. The low efficiency reported on the Intel Delta was based on some initial data.

Table 3.1 *Speedup and Efficiency of Parallel Computers for Solving a Linear System with 1000 Unknowns*

Computer Model	*No. of Processors* n	*Uni-processor Timing* T_1 (s)	*Multi-processor Timing* T_n (s)	*Speedup* $S = T_1/T_n$	*Efficiency* $E = S/n$
Cray Y-MP C90	16	0.77	0.069	11.12	0.69
NEC SX-3	2	0.15	0.082	1.82	0.91
Cray Y-MP/8	8	2.17	0.312	6.96	0.87
Fujitsu AP 1000	512	160.0	1.10	147.0	0.29
IBM 3090/600S VF	6	7.27	1.29	5.64	0.94
Intel Delta	512	22.0	1.50	14.7	0.03
Alliant FX/2800-200	14	22.9	2.06	11.1	0.79
nCUBE/2	1024	331.0	2.59	128.0	0.12
Convex C3240	4	14.9	3.92	3.81	0.95
Parsytec FT-400	400	1075.0	4.90	219.0	0.55

Source: Jack Dongarra, "Performance of Various Computers Using Standard Linear Equations Software," Computer Science Dept., Univ. of Tennessee, Knoxville, TN 37996-1301, March 11, 1992.

3.1.4 Benchmarks and Performance Measures

We have used MIPS and Mflops to describe the *instruction execution rate* and *floating-point capability* of a parallel computer. The MIPS rate defined in Eq. 1.3 is calculated from clock frequency and average CPI. In practice, the MIPS and Mflops ratings and other performance indicators to be introduced below should be measured from running benchmarks or real programs on real machines.

In this section, we introduce standard measures adopted by the industry to compare various computer performance, including *Mflops, MIPS, KLIPS, Dhrystone*, and *Whestone*, as often encountered in reported computer ratings.

Most computer manufacturers state peak or sustained performance in terms of MIPS or Mflops. These ratings are by no means conclusive. The real performance is always program-dependent or application-driven. In general, the MIPS rating depends on the instruction set, varies between programs, and even varies inversely with respect to performance, as observed by Hennessy and Patterson (1990).

To compare processors with different clock cycles and different instruction sets is not totally fair. Besides the native MIPS, one can define a *relative MIPS* with respect to a reference machine. We will discuss relative MIPS rating against the VAX/780 when Dhrystone performance is introduced below. For numerical computing, the LINPACK results on a large number of computers are reported in Chapter 8.

Similarly, the Mflops rating depends on the machine hardware design and on the program behavior. MIPS and Mflops ratings are not convertible because they measure different ranges of operations. The conventional rating is called the *native Mflops*, which does not distinguish unnormalized from normalized floating-point operations.

For example, a *real floating-point divide* operation may correspond to four *normalized floating-point divide* operations. One needs to use a conversion table between real and normalized floating-point operations to convert a native Mflops rating to a normalized Mflops rating.

The Dhrystone Results This is a CPU-intensive benchmark consisting of a mix of about 100 high-level language instructions and data types found in system programming applications where floating-point operations are not used (Weicker, 1984). The Dhrystone statements are balanced with respect to statement type, data type, and locality of reference, with no operating system calls and making no use of library functions or subroutines. Thus the Dhrystone rating should be a measure of the integer performance of modern processors. The unit *KDhrystones/s* is often used in reporting Dhrystone results.

The Dhrystone benchmark version 1.1 was applied to a number of processors. DEC VAX 11/780 scored 1.7 KDhrystones/s performance. This machine has been used as a reference computer with a 1 MIPS performance. The relative VAX/MIPS rating is commonly accepted by the computer industry.

The Whetstone Results This is a Fortran-based synthetic benchmark assessing the floating-point performance, measured in the number of *KWhetstones/s* that a system can perform. The benchmark includes both integer and floating-point operations involving array indexing, subroutine calls, parameter passing, conditional branching, and trigonometric/transcendental functions.

The Whetstone benchmark does not contain any vectorizable code and shows dependence on the system's mathematics library and efficiency of the code generated by a compiler.

The Whetstone performance is not equivalent to the Mflops performance, although the Whetstone contains a large number of scalar floating-point operations.

Both the Dhrystone and Whetstone are synthetic benchmarks whose performance results depend heavily on the compilers used. As a matter of fact, the Dhrystone benchmark program was originally written to test the CPU and compiler performance for a typical program. Compiler techniques, especially procedure in-lining, can significantly affect the Dhrystone performance.

Both benchmarks were criticized for being unable to predict the performance of user programs. The sensitivity to compilers is a major drawback of these benchmarks. In real-life problems, only application-oriented benchmarks will do the trick. We will examine the SPEC and other benchmark suites in Chapter 9.

The TPS and KLIPS Ratings On-line transaction processing applications demand rapid, interactive processing for a large number of relatively simple transactions. They are typically supported by very large databases. Automated teller machines and airline reservation systems are familiar examples. Today many such applications one web-based.

The throughput of computers for on-line transaction processing is often measured in *transactions per second* (TPS). Each transaction may involve a database search, query answering, and database update operations. Business computers and servers should be designed to deliver a high TPS rate. The TP1 benchmark was originally proposed in 1985 for measuring the transaction processing of business application computers. This benchmark also became a standard for gauging relational database performances.

Over the last couple of decades, there has been an enormous increase both in the diversity and the scale of computer applications deployed around the world. The world-wide web, web-based applications, multimedia applications and search engines did not exist in the early 19990s. Such scale and diversity have been made possible by huge advances in processing, storage, graphics display and networking capabilities over this period, which have been reviewed in Chapter 13.

For such applications, application-specific benchmarks have become more important than general purpose benchmarks such as Whetstone. For web servers providing 24 × 7 service for example, we may wish to benchmark—under simulated but realistic load conditions—performance parameters such as: *throughput* (in number of requests served and/or amount of data delivered) and *average response time.*

In artificial intelligence applications, the measure KLIPS (*kilo logic inferences per second*) was used at one time to indicate the reasoning power of an AI machine. For example, the high-speed inference machine developed under Japan's Fifth-Generation Computer System Project claimed a performance of 400 KLIPS.

Assuming that each logic inference operation involves about 100 assembly instructions, 400 KLIPS implies approximately 40 MIPS in this sense. The conversion ratio is by no means fixed. Logic inference demands symbolic manipulations rather than numeric computations. Interested readers are referred to the book edited by Wah and Ramamoorthy (1990).

3.2 PARALLEL PROCESSING APPLICATIONS

Massively parallel processing has become one of the frontier challenges in supercomputer applications. We introduce grand challenges in high-performance computing and communications and then assess the speed, memory, and I/O requirements to meet these challenges. Characteristics of parallel algorithms are also discussed in this context.

3.2.1 Massive Parallelism for Grand Challenges

The definition of massive parallelism varies with respect to time. Based on today's standards, any machine having hundreds or thousands of processors is a *massively parallel processing* (MPP) system. As computer technology advances rapidly, the demand for a higher degree of parallelism becomes more obvious.

The performance of most commercial computers is marked by their peak MIPS rate or peak Mflops rate. In reality, only a fraction of the peak performance is achievable in real benchmark or evaluation runs. Observing the sustained performance makes more sense in evaluating computer performance.

Grand Challenges We review below same of the grand challenges identified in the U.S. High-Performance Computing and Communication (HPCC) program, reveal opportunities for massive parallelism, assess past developments, and comment on future trends in MPP.

(1) The magnetic recording industry relies on the use of computers to study megmetostatic and exchange

interactions in order to reduce noise in metallic thin films used to coat high-density disks. In general, all research in science and engineering makes heavy demands on computing power.

(2) Rational drug design is being aided by computers in the search for a cure for cancer, acquired immunodeficiency syndrome and other diseases. Using a high-performance computer, new potential agents have been identified that block the action of human immunodeficiency virus protease.

(3) Design of high-speed transport aircraft is being aided by computational fluid dynamics running on supercomputers. Fuel combustion can be made more efficient by designing better engine models through chemical kinetics calculations.

(4) Catalysts for chemical reactions are being designed with computers for many biological processes which are catalytically controlled by enzymes. Massively parallel quantum models demand large simulations to reduce the time required to design catalysts and to optimize their properties.

(5) Ocean modeling cannot be accurate without supercomputing MPP systems. Ozone depletion and climate research demands the use of computers in analyzing the complex thermal, chemical and fluid-dynamic mechanisms involved.

(6) Other important areas demanding computational support include digital anatomy in real-time medical diagnosis, air pollution reduction through computational modeling, the design of protein structures by computational biologists, image processing and understanding, and technology linking research to education.

Besides computer science and computer engineering, the above challenges also encourage the emerging discipline of computational science and engineering. This demands systematic application of computer systems and computational solution techniques to mathematical models formulated to describe and to simulate phenomena of scientific and engineering interest.

The HPCC Program also identified some grand challenge computing requirements of the time, as shown in Fig. 3.5. This diagram shows the levels of processing speed and memory size required to support scientific simulation modeling, advanced *computer-aided design* (CAD), and real-time processing of large-scale database and information retrieval operations. In the period since the early 1990s, there have been huge advances in the processing, storage and networking capabilities of computer systems. Some MPP systems have reached petaflop performance, while even PCs have gigabytes of memory. At the same time, computing requirements in science and engineering have also grown enormously.

Exploiting Massive Parallelism The parallelism embedded in the instruction level or procedural level is rather limited. Very few parallel computers can successfully execute more than two instructions per machine cycle from the same program. *Instruction parallelism* is often constrained by program behavior, compiler/OS incapabilities, and program flow and execution mechanisms built into modern computers.

On the other hand, *data parallelism* is much higher than instruction parallelism. Data parallelism refers to the situation where the same operation (instruction or program) executes over a large array of data (operands). Data parallelism has been implemented on pipelined vector processors, SIMD array processors, and SPMD or MPMD multicomputer systems.

In Table 1.6, we find SIMD data parallelism over 65,536 PEs in the CM-2. One may argue that the CM-2 was a bit-slice machine. Even if we divide the number by 64 (the word length of a typical supercomputer), we still end up with a DOP on the order of thousands in CM-2.

The vector length can be used to determine the parallelism implementable on a vector supercomputer.

In the case of the Cray Y/MP C-90, 32 pipelines in 16 processors could potentially achieve a DOP of $32 \times 5 = 160$ if the average pipeline has five stages. Thus a pipelined processor can support a lower degree of data parallelism than an SIMD computer.

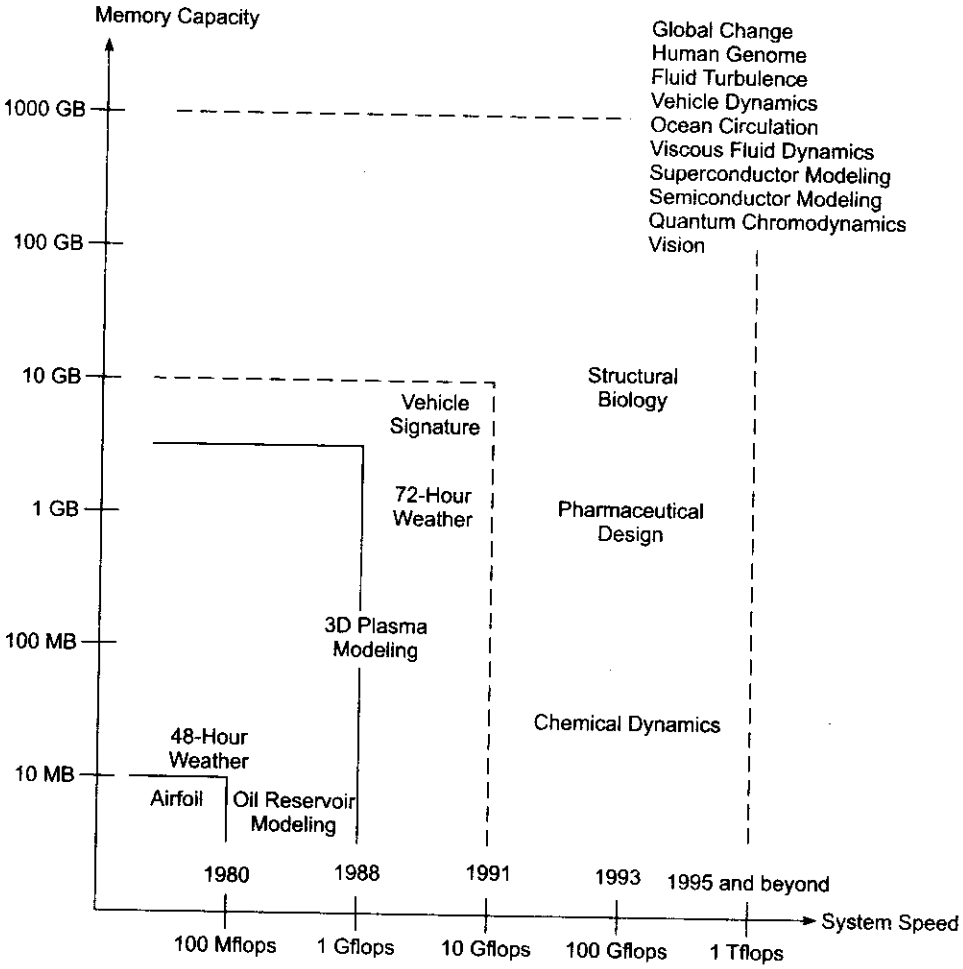


Fig. 3.5 Grand challenge requirements in computing and communications (Courtesy of U.S. High-Performance Computing and Communication Program, 1992)

On a message-passing multicomputer, the parallelism is more scalable than a shared-memory multiprocessor. As revealed in Table 1.4, the nCUBE/2 could achieve a maximum parallelism on the order of thousands if all the node processors were kept busy simultaneously.

The Past and the Future MPP systems started in 1968 with the introduction of the Illiac IV computer with 64 PEs under one controller. Subsequently, a massively parallel processor, called MPP, was built by Goodyear with 16,384 PEs. IBM built a GF11 machine with 576 PEs. The MasPar MP-1, AMT DAP610, and CM-2 were all early examples of SIMD computers.

Early MPP systems operating in MIMD mode included the BBN TC-2000 with a maximum configuration of 512 processors. The IBM RP-3 was designed to have 512 processors (only a 64-processor version was built). The Intel Touchstone Delta was a system with 570 processors.

Several subsequent MPP projects included the Paragon by Intel Supercomputer Systems, the CM-5 by Thinking Machine Corporation, the KSR-1 by Kendall Square Research, the Fujitsu VPP500 System, the Tera computer, and the MIT *T system.

IBM announced MPP projects using thousands of IBM RS/6000 and later Power processors, while Cray developed MPP systems using Digital's Alpha processors and later AMD Opteron processors as building blocks. Some early MPP projects are summarized in Table 3.2. We will study some of these systems in later chapters, and more recent advances in Chapter 13.

Table 3.2 *Early Representative Massively Parallel Processing Systems*

MPP System	*Architecture, Technology, and Operational Features*
Intel Paragon	A 2-D mesh-connected multicomputer, built with i860 XP processors and wormhole routers, targeted for 300 Gflops in peak performance.
IBM MPP Model	Use IBM RISC/6000 processors as building blocks, 50 Gflops peak expected for a 1024-processor configuration.
TMC CM-5	A universal architecture for SIMD/MIMD computing using SPARC PEs and custom-designed FPUs, control and data networks, 2 Tflops peak for 16K nodes.
Cray Research MPP Model	A 3D torus heterogeneous architecture using DEC Alpha chips with special communication support, global address space over physically distributed memory; first system offered 150 Gflops in a 1024-processor configuration in 1993; capable of growing to Tflops with larger configurations.
Kendall Square Research KSR-1	An ALLCACHE ring-connected multiprocessor with custom-designed processors, 43 Gflops peak performance for a 1088-processor configuration.
Fujitsu VPP500	A crossbar-connected 222-PE MIMD vector system, with shared distributed memories using VP2000 as a host; peak performance = 355 Gflops.

3.2.2 Application Models of Parallel Computers

In general, if the workload is kept unchanged as shown by curve α in Fig. 3.6a, then the efficiency E decreases rapidly as the machine size n increases. The reason is that the overhead h increases faster than the machine size. To maintain the efficiency at a desired level, one has to increase the machine size and problem size proportionally. Such a system is known as a *scalable computer* for solving *scaled problems*.

In the ideal case, we like to see a workload curve which is a linear function of n (curve γ in Fig. 3.6a). This implies linear scalability in problem size. If the linear workload curve is not achievable, the second choice is to achieve a sublinear scalability as close to linearity as possible, as illustrated by curve β in Fig. 3.6a, which has a smaller constant of proportionality than the curve γ.

Suppose that the workload follows an exponential growth pattern and becomes enormously large, as shown by curve θ in Fig. 3.6a. The system is considered poorly scalable in this case. The reason is that to keep a constant efficiency or a good speedup, the increase in workload with problem size becomes explosive and exceeds the memory or I/O limits.

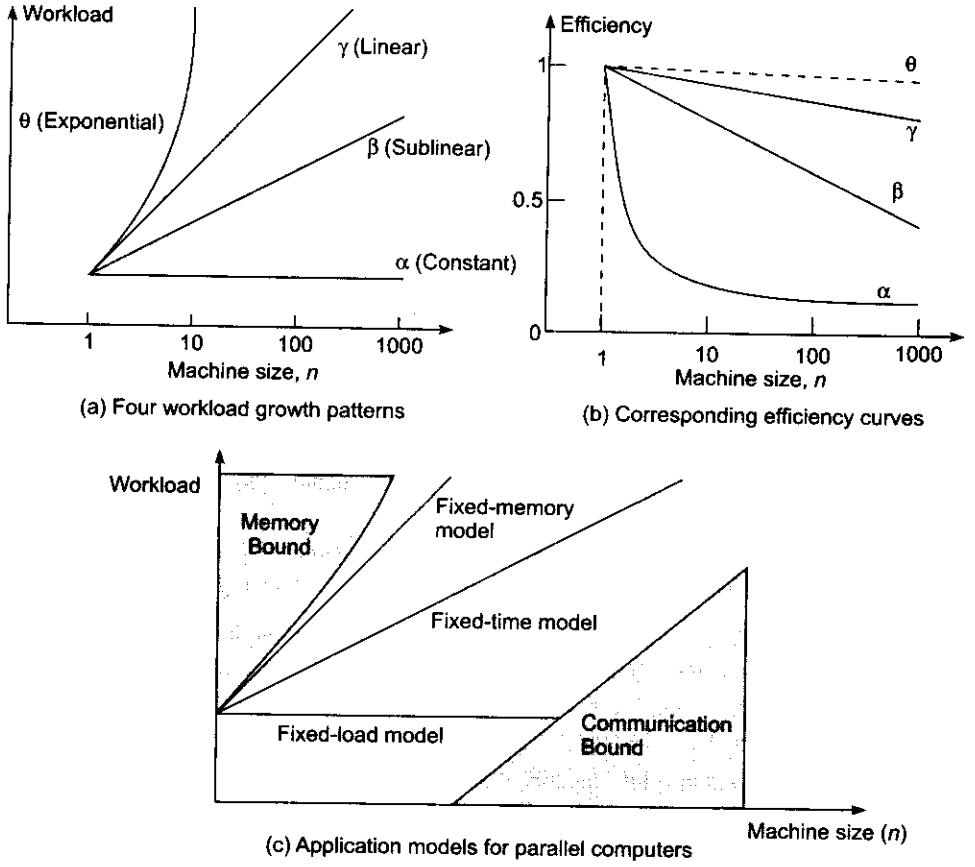


Fig. 3.6 Workload growth, efficiency curves, and application models of parallel computers under resources constraints

The Efficiency Curves Corresponding to the four workload patterns specified in Fig. 3.6a, four efficiency curves are shown in Fig. 3.6b, respectively. With a constant workload, the efficiency curve (α) drops rapidly. In fact, curve α corresponds to the famous Amdahl's law. For a linear workload, the efficiency curve (γ) is almost flat, as observed by Gustafson in 1988.

The exponential workload (θ) may not be implementable due to memory shortage or I/O bounds (if real-time application is considered). Thus the θ efficiency (dashed lines) is achievable only with exponentially increased memory (or I/O) capacity. The sublinear efficiency curve (β) lies somewhere between curves α and γ.

Scalability analysis determines whether parallel processing of a given problem can offer the desired improvement in performance. The analysis should help guide the design of a massively parallel processor. It is clear that no single scalability metric suffices to cover all possible cases. Different measures will be useful in different contexts, and further analysis is needed along multiple dimensions for any specific application.

A parallel system can be used to solve arbitrarily large problems in a fixed time if and only if its workload

pattern is allowed to grow linearly. Sometimes, even if minimum time is achieved with more processors, the system utilization (or efficiency) may be very poor.

Application Models The workload patterns shown in Fig. 3.6a are not allowed to grow unbounded. In Fig. 3.6c, we show three models for the application of parallel computers. These models are bounded by limited memory, limited tolerance of IPC latency, or limited I/O bandwidth. These models are briefly introduced below. They lead to three speedup performance models to be formulated in Section 3.3.

The *fixed-load model* corresponds to a constant workload (curve α in Fig. 3.6a). The use of this model is eventually limited by the communication bound shown by the shaded area in Fig. 3.6c.

The *fixed-time model* demands a constant program execution time, regardless of how the workload scales up with machine size. The linear workload growth (curve γ in Fig. 3.6a) corresponds to this model. The *fixed-memory model* is limited by the memory bound, corresponding to a workload curve between γ and θ in Fig. 3.6a.

From the application point of view, the shaded areas are forbidden. The communication bound includes not only the increasing IPC overhead but also the increasing I/O demands. The memory bound is determined by main memory and disk capacities.

In practice, an algorithm designer or a parallel computer programmer may choose an application model within the above resource constraints, as shown in the unshaded application region in Fig. 3.6c.

Tradeoffs in Scalability Analysis Computer cost c and programming overhead p (in addition to speedup and efficiency) are equally important in scalability analysis. After all, cost-effectiveness may impose the ultimate constraint on computing with a limited budget. What we have studied above was concentrated on system efficiency and fast execution of a single algorithm/program on a given parallel computer.

It would be interesting to extend the scalability analysis to multiuser environments in which multiple programs are executed concurrently by sharing the available resources. Sometimes one problem is poorly scalable, while another has good scalability characteristics. Tradeoffs exist in increasing resource utilization but not necessarily to minimize the overall execution time in an optimization process.

Exploiting parallelism for higher performance demands both scalable architectures and scalable algorithms. The architectural scalability can be limited by long communication latency, bounded memory capacity, bounded I/O bandwidth, and limited processing speed. How to achieve a balanced design among these practical constraints is the major challenge of today's MPP system designers. On the other hand, parallel algorithms and efficient data structures also need to be scalable.

3.2.3 Scalability of Parallel Algorithms

In this subsection, we analyze the scalability of parallel algorithms with respect to key machine classes. An isoefficiency concept is introduced for scalability analysis of parallel algorithms. Two examples are used to illustrate the idea. Further studies of scalability are given in Section 3.4 after we study the speedup performance laws in Section 3.3.

Algorithmic Characteristics Computational algorithms are traditionally executed sequentially on uniprocessors. *Parallel algorithms* are those specially devised for parallel computers. The idealized parallel algorithms are those written for the PRAM models if no physical constraints or communication overheads are imposed. In the real world, an algorithm is considered efficient only if it can be cost effectively implemented on physical machines. In this sense, all machine-implementable algorithms must be architecture-dependent. This means the effects of communicaiton overhead and architectural constraints cannot be ignored.

We summarize below important characteristics of parallel algorithms which are machine implementable:

(1) *Deterministic versus nondeterministic*: As defined in Section 1.4.1, only deterministic algorithms are implementable on real machines. Our study is confined to deterministic algorithms with polynomial time complexity

(2) *Computational granularity*: As introduced in Section 2.2.1, granularity decides the size of data items and program modules used in computation. In this sense, we also classify algorithms as fine-grain, medium-grain, or coarse-grain.

(3) *Parallelism profile*: The distribution of the degree of parallelism in an algorithm reveals the opportunity for parallel processing. This often affects the effectiveness of the parallel algorithms.

(4) *Communication patterns and synchronization requirements*: Communication patterns address both memory access and interprocessor communications. The patterns can be *static* or *dynamic*, depending on the algorithms. Static algorithms are more suitable for SIMD or pipelined machines, while dynamic algorithms are for MIMD machines. The *synchronization frequency* often affects the efficiency of an algorithm.

(5) *Uniformity of the operations*: This refers to the types of fundamental operations to be performed. Obviously, if the operations are uniform across the data set, the SIMD processing or pipelining may be more desirable. In other words, randomly structured algorithms are more suitable for MIMD processing. Other related issues include data types and precision desired.

(6) *Memory requirement and data structures*: In solving large-scale problems, the data sets may require huge memory space. *Memory efficiency* is affected by data structures chosen and data movement patterns in the algorithms. Both time and space complexities are key measures of the granularity of a parallel algorithm.

The Isoefficiency Concept The workload w of an algorithm grows with s, the problem size. Thus, we denote the workload $w = w(s)$ as a function of s. Kumar and Rao (1987) have introduced an *isoefficiency* concept relating workload to machine size n needed to maintain a fixed efficiency E when implementing a parallel algorithm on a parallel computer. Let h be the total communication overhead involved in the algorithm implementation. This overhead is usually a function of both machine size and problem size, thus denoted $h = h(s, n)$.

The efficiency of a parallel algorithm implemented on a given parallel computer is thus defined as

$$E = \frac{w(s)}{w(s) + h(s,n)} \tag{3.20}$$

The workload $w(s)$ corresponds to useful computations while the overhead $h(s, n)$ are computations attributed to synchronization and data communication delays. In general, the overhead increases with respect to both increasing values of s and n. Thus, the efficiency is always less than 1. The question is hinged on relative growth rates between $w(s)$ and $h(s, n)$.

With a fixed problem size (or fixed workload), the efficiency decreases as n increase. The reason is that the overhead $h(s, n)$ increases with n. With a fixed machine size, the overhead h grows slower than the workload w. Thus the efficiency increases with increasing problem size for a fixed-size machine. Therefore, one can expect to maintain a constant efficiency if the workload w is allowed to grow properly with increasing machine size.

For a given algorithm, the workload w might need to grow polynomially or exponentially with respect to n in order to maintain a fixed efficiency. Different algorithms may require different workload growth rates to keep the efficiency from dropping, as n is increased. The isoefficiency functions of common parallel algorithms are polynomial functions of n; i.e., they are $O(n^k)$ for some $k \geq 1$. The smaller the power of n in the isoefficiency function, the more scalable the parallel system. Here, the system includes the algorithm and architecture combination.

Isoefficiency Function We can rewrite Eq. 3.20 as $E = 1/(1 + h(s, n)/w(s))$. In order to maintain a constant E, the workload $w(s)$ should grow in proportion to the overhead $h(s, n)$. This leads to the following condition:

$$w(s) = \frac{E}{1-E} \times h(s, n) \tag{3.21}$$

The factor $C = E/(1 - E)$ is a constant for a fixed efficiency E. Thus we can define the *isoefficiency function* as follows:

$$f_E(n) = C \times h(s, n) \tag{3.22}$$

If the workload $w(s)$ grows as fast as $f_E(n)$ in Eq. 3.21, then a constant efficiency can be maintained for a given algorithm-architecture combination. Two examples are given below to illustrate the use of isoefficiency functions for scalability analysis.

Example 3.4 Scalability of matrix multiplication algorithms (Gupta and Kumar, 1992)

Four algorithms for matrix multiplication are compared below. The problem size s is represented by the matrix order. In other words, we consider the multiplication of two $s \times s$ matrices A and B to produce an output matrix C = A × B. The total workload involved is $w = O(s^3)$. The number of processors used is confined within $1 \leq n \leq s^3$. Some of the algorithms may use less than s^3 processors.

The isoefficiency functions of the four algorithms are derived below based on equating the workload with the communication overhead (Eq. 3.21) in each algorithm. Details of these algorithms and corresponding architectures can be found in the original papers identified in Table 3.3 as well as in the paper by Gupta and Kumar (1992). The derivation of the communication overheads is left as an exercise in Problem 3.14.

The Fox-Otto-Hey algorithm has a total overhead $h(s, n) = O(n \log n + s^2 \sqrt{n})$. The workload $w = O(s^3) = O(n \log n + s^2 \sqrt{n})$. Thus we must have $O(s^3) = O(n \log n)$ and $O(s) = O(\sqrt{n})$. Combining the two, we obtain the isoefficiency function $O(s^3) = O(n^{3/2})$, where $1 \leq n \leq s^2$ as shown in the first row of Table 3.3.

Although this algorithm is written for the torus architecture, the torus can be easily embedded in a hypercube architecture. Thus we can conduct a fair comparison of the four algorithms against the hypercube architecture.

Berntsen's algorithm restricts the use of $n \leq s^{3/2}$ processors. The total overhead is $O(n^{4/3} + n \log n + s^2 n^{1/3})$. To match this with $O(s^3)$, we must have $O(s^3) = O(n^{4/3})$ and $O(s^3) = O(n)$. Thus, $O(s^3)$ must be chosen to yield the isoefficiency function $O(n^2)$.

The Gupta-Kumar algorithm has an overhead $O(n \log n + s^2 n^{1/3} \log n)$. Thus we must have $O(s^3) = O(n \log n)$ and $O(s^3) = O(s^2 n^{1/3} \log n)$. This leads to the isoefficiency function $O(n(\log n)^3)$ in the third row of Table 3.3.

The Dekel-Nassimi-Sahni algorithm has a total overhead $O(n \log n + s^3)$ besides a useful computation time of $O(s^3/n)$ for $s^2 \le n \le s^3$. Thus the workload growth $O(s^3) = O(n \log n)$ will yield the isoefficiency listed in the last row of Table 3.3.

Table 3.3 *Asymptotic Isoefficiency Functions of Four Matrix Multiplication Algorithms (Gupta and Kumar, 1992)*

Matrix Multiplication Algorithm	*Isoefficiency Function $f_E(n)$*	*Range of Applicability*	*Target Machine Architecture*
Fox, Otto, and Hey (1987)	$O(n^{3/2})$	$1 \le n \le s^2$	$A\ \sqrt{n} \times \sqrt{n}$ torus
Berntsen (1989)	$O(n^2)$	$1 \le n \le s^{3/2}$	A hypercube with $n = 2^{3k}$ nodes
Gupta and Kumar (1992)	$O(n(\log n)^3)$	$1 \le n \le s^3$	A hypercube with $n = 2^{3k}$ nodes and $k < \frac{1}{3} \log s$
Dekel, Nassimi, and Sahni (1981)	$O(n \log n)$	$s^2 \le n \le s^3$	A hypercube with $n = s^3 = 2^{3k}$ nodes

Note: Two $s \times s$ matrices are multiplied.

The above isoefficiency functions indicate the asymptotic scalabilities of the four algorithms. In practice, none of the algorithms is strictly better than the others for all possible problem sizes and machine sizes. For example, when these algorithms are implemented on a multicomputer with a long communication latency (as in Intel iPSC1), Berntsen's algorithm is superior to the others.

To map the algorithms on an SIMD computer with an extremely low synchronization overhead, the algorithm by Gupta and Kumar is inferior to the others. Hence, it is best to use the Dekel-Nassimi-Sahni algorithm for $s^2 \le n \le s^3$, the Fox-Otto-Hey algorithm for $s^{3/2} \le n \le s^2$, and Berntsen's algorithm for $n \le s^{3/2}$ for SIMD hypercube machines.

Example 3.5 Fast Fourier transform on mesh and hypercube computers (Gupta and Kumar, 1993)

This example demonstrates the sensitivity of machine architecture on the scalability of the FFT on two different parallel computers: *mesh and hypercube*. We consider the Cooley-Tukey algorithm for one-dimensional s-point fast Fourier transform.

Gupta and Kumar have established the overheads: $h_1(s, n) = O(n \log n + s \log n)$ for FFT on a hypercube machine with n processors, and $h_2(s, n) = O(n \log n + s\sqrt{n})$ on a $\sqrt{n} \times \sqrt{n}$ mesh with n processors.

For an s-point FFT, the total workload involved is $w(s) = O(s \log s)$. Equating the workload with overheads, we must satisfy $O(s \log s) = O(n \log n)$ and $O(s \log s) = O(s \log n)$, leading to the isoefficiency function $f_1 = O(n \log n)$ for the hypercube machine.

Similarly, we must satisfy $O(s \log s) = O(n \log n)$ and $O(s \log s) = O(s\sqrt{n})$ by equating $w(s) = h_2(s, n)$. This leads to the isoefficiency function $f_2 = O(\sqrt{nk}^{\sqrt{n}})$ for some constant $k \leq 2$.

The above analysis leads to the conclusion that FFT is indeed very scalable on a hypercube computer. The result is plotted in Fig. 3.7a for three efficiency values.

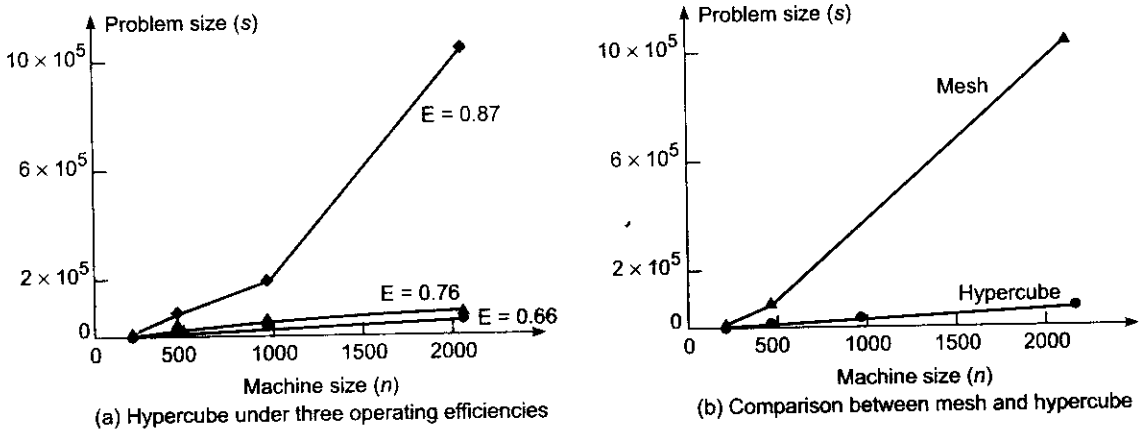


Fig. 3.7 Isoefficiency curves for FFT on two parallel computers (Courtesy of Gupta and Kumar, 1993)

To maintain the same efficiency, the mesh is rather poorly scalable as demonstrated in Fig. 3.7b.

This is predictable by the fact that the workload must grow exponentially in $O(\sqrt{nk}^{\sqrt{n}})$ for the mesh architecture, while the hypercube demands only $O(n \log n)$ workload increase as the machine size increases. Thus, we conclude that the FFT is scalable on a hypercube but not so on a mesh architecture.

If the bandwidth of the communication channels in a mesh architecture increases proportional to the increase of machine size, the above conclusion will change. For the design and analysis of FFT on parallel machines, readers are referred to the books by Aho, Hopcroft and Ullman (1974) and by Quinn (1987). We will further address scalability issues.from the architecture standpoint in Section 3.4.

3.3 SPEEDUP PERFORMANCE LAWS

Three speedup performance models are defined below. Amdahl's law (1967) is based on a fixed workload or a fixed problem size. Gustafson's law (1987) is applied to scaled problems, where the problem size increases with the increase in machine size. The speedup model by Sun and Ni (1993) is for scaled problems bounded by memory capacity.

3.3.1 Amdahl's Law for a Fixed Workload

In many practical applications that demand a real-time response, the computational workload is often fixed with a fixed problem size. As the number of processors increases in a parallel computer, the fixed load is distributed to more processors for parallel execution. Therefore, the main objective is to produce the results

as soon as possible. In other words, minimal turnaround time is the primary goal. Speedup obtained for time-critical applications is called fixed-load speedup.

Fixed-Load Speedup The ideal speedup formula given in Eq. 3.7 is based on a fixed workload, regardless of the machine size. Traditional formulations for speedup, including Amdahl's law, are all based on a fixed problem size and thus on a fixed load. The speedup factor is upper-bounded by a sequential bottleneck in this case.

We consider below both the cases of DOP $< n$ and of DOP $\geq n$. We use the ceiling function $\lceil x \rceil$ to represent the smallest integer that is greater than or equal to the positive real number x. When x is a fraction, $\lceil x \rceil$ equals 1. Consider the case where DOP $= i > n$. Assume all n processors are used to execute W_i exclusively. The execution time of W_i is

$$t_i(n) = \frac{W_i}{i\Delta}\left\lceil \frac{i}{n} \right\rceil \tag{3.23}$$

Thus the response time is

$$T(n) = \sum_{i=1}^{m} \frac{W_i}{i\Delta}\left\lceil \frac{i}{n} \right\rceil \tag{3.24}$$

Note that if $i < n$, then $t_i(n) = t_i(\infty) = W_i/i\Delta$. Now, we define the *fixed-load speedup factor* as the ratio of $T(1)$ to $T(n)$:

$$S_n = \frac{T(1)}{T(n)} = \frac{\sum_{i=1}^{m} W_i}{\sum_{i=1}^{m} \frac{W_i}{i}\left\lceil \frac{i}{n} \right\rceil} \tag{3.25}$$

Note that $S_n \leq S_\infty \leq A$, by comparing Eqs. 3.4, 3.7, and 3.25.

A number of factors we have ignored may lower the speedup performance. These include communication latencies caused by delayed memory access, interprocessor communication over a bus or a network, or operating system overhead and delay caused by interrupts. Let $Q(n)$ be the lumped sum of all system overheads on an n-processor system. We can rewrite Eq, 3.25 as follows:

$$S_n = \frac{T(1)}{T(n) + Q(n)} = \frac{\sum_{i=1}^{m} W_i}{\sum_{i=1}^{m} \frac{W_i}{i}\left\lceil \frac{i}{n} \right\rceil + Q(n)} \tag{3.26}$$

The overhead delay $Q(n)$ is certainly application-dependent as well as machine-dependent. It is very difficult to obtain a closed form for $Q(n)$. Unless otherwise specified, we assume $Q(n) = 0$ to simplify the discussion.

Amdahl's Law Revisited In 1967, Gene Amdahl derived a fixed-load speedup for the special case where the computer operates either in sequential mode (with DOP = 1) or in perfectly parallel mode (with DOP $= n$). That is, $W_i = 0$ if $i \neq 1$ or $i \neq n$ in the profile. Equation 3.25 is then simplified to

$$S_n = \frac{W_1 + W_n}{W_1 + W_n/n} \tag{3.27}$$

Amdahl's law implies that the sequential portion of the program W_i does not change with respect to the machine size n. However, the parallel portion is evenly executed by n processors, resulting in a reduced time.

Consider a normalized situation in which $W_1 + W_n = \alpha + (1 - \alpha) = 1$, with $\alpha = W_1$ and $\alpha = W_n$. Equation 3.27 is reduced to Eq. 3.14, where α represents the percentage of a program that must be executed sequentially and $1 - \alpha$ corresponds to the portion of the code that can be executed in parallel.

Amdahl's law is illustrated in Fig. 3.8. When the number of processors increases, the load on each processor decreases. However, the total amount of work (workload) $W_1 + W_n$ is kept constant as shown in Fig. 3.8a. In Fig. 3.8b, the total execution time decreases because $T_n = W_n/n$. Eventually, the sequential part will dominate the performance because $T_n \to 0$ as n becomes very large and T_1 is kept unchanged.

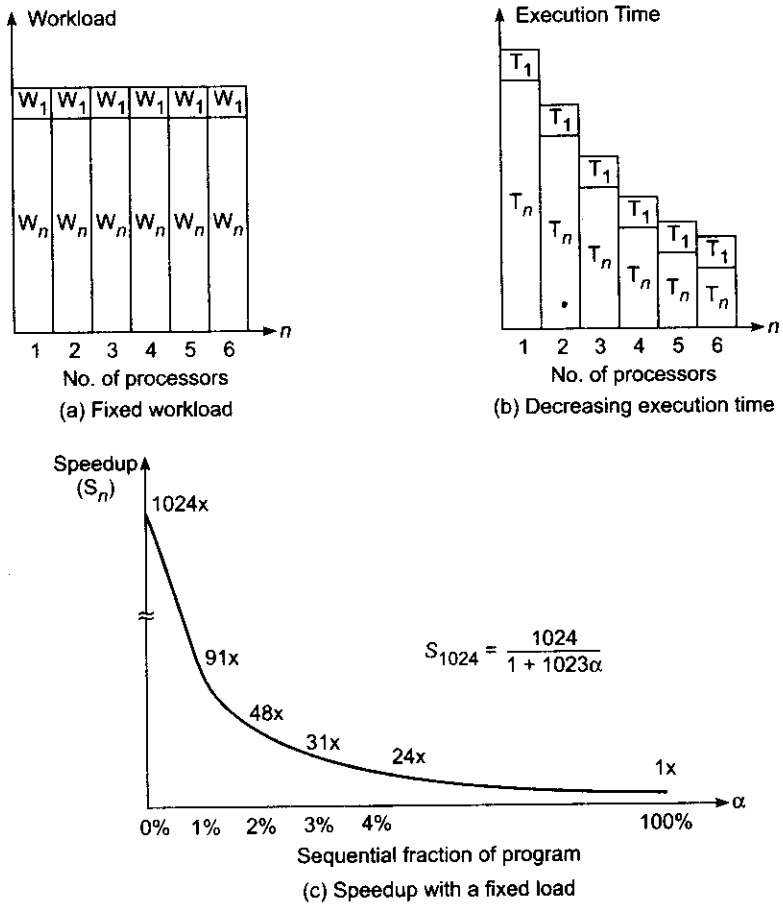


Fig. 3.8 Fixed-load speedup model and Amdahl's law

Sequential Bottleneck Figure 3.8c plots Amdahl's law using Eq. 3.14 over the range $0 \le \alpha \le 1$. The maximum speedup $S_n = n$ if $\alpha = 0$. The minimum speedup $S_n = 1$ if $\alpha = 1$. As $n \to \infty$, the limiting value of $S_n \to 1/\alpha$. This implies that the speedup is upper-bounded by $1/\alpha$, as the machine size becomes very large.

The speedup curve in Fig. 3.8c drops very rapidly as α increases. This means that with a small percentage of the sequential code, the entire performance cannot go higher than $1/\alpha$. This α has been called *the sequential bottleneck* in a program.

The problem of a sequential bottleneck cannot be solved just by increasing the number of processors in a system. The real problem lies in the existence of a sequential fraction of the code. This property has imposed a pessimistic view on parallel processing over the past two decades.

In fact, two major impacts on the parallel computer industry were observed. First, manufacturers were discouraged from making large-scale parallel computers. Second, more research attention was shifted toward developing parallelizing compilers which would reduce the value of α and in turn boost the performance.

3.3.2 Gustafson's Law for Scaled Problems

One of the major shortcomings in applying Amdahl's law is that the problem (workload) cannot scale to match the available computing power as the machine size increases. In other words, the fixed load prevents scalability in performance. Although the sequential bottleneck is a serious problem, the problem can be greatly alleviated by removing the fixed-load (or fixed-problem-size) restriction. John Gustafson (1988) has proposed a fixed-time concept which leads to a scaled speedup model.

Scaling for Higher Accuracy Time-critical applications provided the major motivation leading to the development of the fixed-load speedup model and Amdahl's law. There are many other applications that emphasize accuracy more than minimum turnaround time. As the machine size is upgraded to obtain more computing power, we may want to increase the problem size in order to create a greater workload, producing more accurate solution and yet keeping the execution time unchanged.

Many scientific modeling and engineering simulation applications demand the solution of very large-scale matrix problems based on some partial differential equation (PDE) formulations discretized with a huge number of grid points. Representative examples include the use of finite-element method to perform structural analysis or the use of finite-difference method to solve computational fluid dynamics problems in weather forecasting.

Coarse grids require less computation, but finer grids require many more computations, yielding greater accuracy. The weather forecasting simulation often demands the solution of four-dimensional PDEs. If one reduces the grid spacing in each physical dimension (X, Y, and Z) by a factor of 10 and increases the time steps by the same magnitude, then we are talking about an increase of 10^4 times more grid points. The workload thus increases to at least 10,000 times greater.

With such a problem scaling, of course, we demand more computing power to yield the same execution time. The main advantage is not in saving time but in producing much more accurate weather forecasting. This problem scaling for accuracy has motivated Gustafson to develop a fixed-time speedup model. The scaled problem keeps all the increased resources busy, resulting in a better system utilization ratio.

Fixed-Time Speedup In accuracy-critical applications, we wish to solve the largest problem size possible on a larger machine with about the same execution time as for solving a smaller problem on a smaller machine. As the machine size increases, we have to deal with an increased workload and thus a new parallelism profile. Let m' be the maximum DOP with respect to the scaled problem and W_i' be the scaled workload with DOP = i.

Note that in general $W_i' > W_i$ for $2 \le i \le m'$ and $W_1' = W_1$. The fixed-time speedup is defined under the assumption that $T(1) = T'(n)$, where $T'(n)$ is the execution time of the scaled problem and $T(1)$ corresponds to the original problem without scaling. We thus obtain

$$\sum_{i=1}^{m} W_i = \sum_{i=1}^{m'} \frac{W_i'}{i} \left\lceil \frac{i}{n} \right\rceil + Q(n) \tag{3.28}$$

A general formula for fixed-time speedup is defined by $S_n' = T(1)/T'(n)$, modified from Eq. 3.26:

$$S_n' = \frac{\sum_{i=1}^{m'} W_i'}{\sum_{i=1}^{m'} \frac{W_i'}{i} \left\lceil \frac{i}{n} \right\rceil + Q(n)} = \frac{\sum_{i=1}^{m'} W_i'}{\sum_{i=1}^{m} W_i} \tag{3.29}$$

Gustafson's Law Fixed-time speedup was originally developed by Gustafson for a special parallelism profile with $W_i = 0$ if $i \neq 1$ and $i \neq n$. Similar to Amdahl's law, we can rewrite Eq. 3.29 as follows, assuming $Q(n) = 0$,

$$S_n' = \frac{\sum_{i=1}^{m'} W_i'}{\sum_{i=1}^{m} W_i} = \frac{W_1' + W_n'}{W_1 + W_n} = \frac{W_1 + nW_n}{W_1 + W_n} \tag{3.30}$$

where $W_n' = nW_n$ and $W_1 + W_n = W_1' + W_n'/n$, corresponding to the fixed-time condition. From Eq. 3.30, the parallel workload W_n' has been scaled to n times W_n in a linear fashion.

The relationship of a scaled workload to Gustafson's scaled speedup is depicted in Fig. 3.9. In fact, Gustafson's law can be restated as follows in terms of $\alpha = W_1$ and $1 - \alpha = W_n$ under the same assumption $W_1 + W_n = 1$ that we have made for Amdahl's law:

$$S_n' = \frac{\alpha + n(1-\alpha)}{\alpha + (1-\alpha)} = n - \alpha(n-1) \tag{3.31}$$

In Fig. 3.9a, we demonstrate the workload scaling situation. Figure 3.9b shows the fixed-time execution style. Figure 3.9c plots S_n' as a function of the sequential portion α of a program running on a system with $n = 1024$ processors.

Note that the slope of the S_n curve in Fig. 3.9c is much flatter than that in Fig. 3.8c. This implies that Gustafson's law does support scalable performance as the machine size increases. The idea is to keep all processors busy by increasing the problem size. When the problem can scale to match available computing power, the sequential fraction is no longer a bottleneck.

3.3.3 Memory-Bounded Speedup Model

Xian-He Sun and Lionel Ni (1993) have developed a memory-bounded speedup model which generalizes Amdahl's law and Gustafson's law to maximize the use of both CPU and memory capacities. The idea is to solve the largest possible problem, limited by memory space. This also demands a scaled workload, providing higher speedup, higher accuracy, and better resource utilization.

Memory-Bound Problems Large-scale scientific or engineering computations often require larger memory space. In fact, many applications of parallel computers are memory-bound rather than CPU-bound

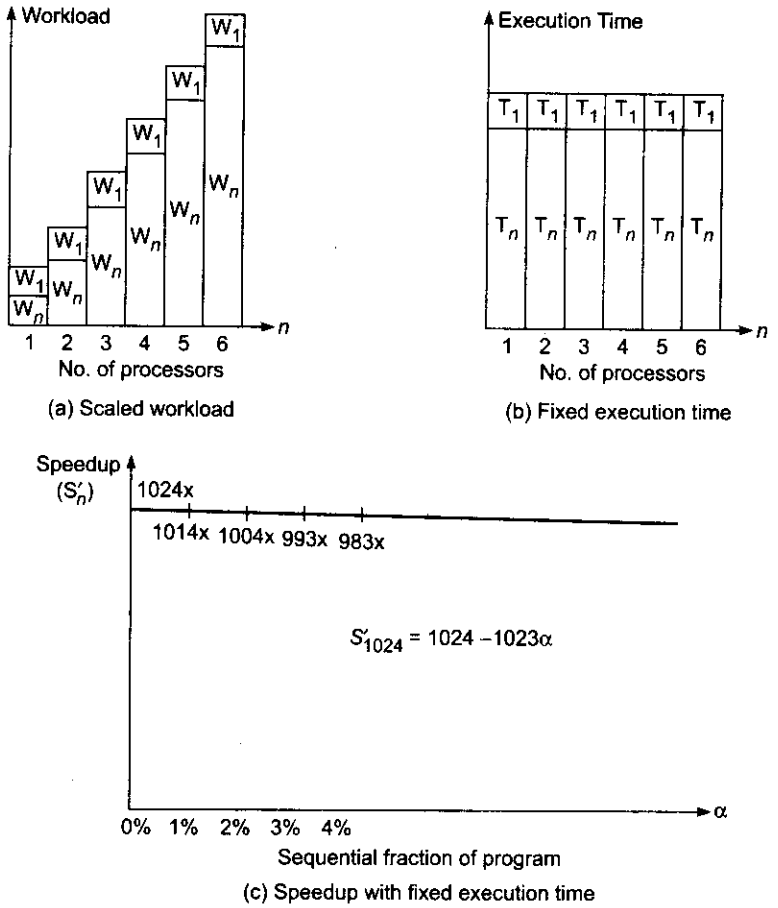


Fig. 3.9 Fixed-time speedup model and Gustafson's law

or I/O-bound. This is especially true in a multicomputer system using distributed memory. The local memory attached to each node may be relatively small. Therefore, each node can handle only a small subproblem.

When a large number of nodes are used collectively to solve a single large problem, the total memory capacity increases proportionally. This enables the system to solve a scaled problem through program partitioning or replication and domain decomposition of the data set.

Instead of keeping the execution time fixed, one may want to use up all the increased memory by scaling the problem size further. In other words, if you have adequate memory space and the scaled problem meets the time limit imposed by Gustafson's law, you can further increase the problem size, yielding an even better or more accurate solution.

A memory-bounded model was developed under this philosophy. The idea is to solve the largest possible problem, limited only by the available memory capacity. This model may result in an increase in execution time to achieve scalable performance.

Fixed-Memory Speedup Let M be the memory requirement of a given problem and W be the computational workload. These two factors are related to each other in various ways, depending on the address space and architectural constraints. Let us write $W = g(M)$ or $M = g^{-1}(W)$, where g^{-1} is the inverse of g.

In a multicomputer, the total memory capacity increases linearly with the number of nodes available. We write $W = \sum_{i=1}^{m^*} W_i$ as the workload for sequential execution of the program on a single node, and $W^* = \sum_{i=1}^{m^*} W_i^*$ as the scaled workload for execution on n nodes, where m^* is the maximum DOP of the scaled problem. The memory requirement for an active node is thus bounded by $g^{-1}\left(\sum_{i=1}^{m} W_i\right)$.

A *fixed-memory speedup* is defined below similarly to that in Eq. 3.29.

$$S_n^* = \frac{\sum_{i=1}^{m^*} W_i^*}{\sum_{i=1}^{m^*} \frac{W_i^*}{i} \left\lceil \frac{i}{n} \right\rceil + Q(n)} \tag{3.32}$$

The workload for sequential execution on a single processor is independent of the problem size or system size. Therefore, we can write $W_1 = W_1' = W_1^*$ in all three speedup models. Let us consider the special case of two operational modes: *sequential* versus *perfectly parallel* execution. The enhanced memory is related to the scaled workload by $W_n^* = g^*(nM)$, where nM is the increased memory capacity for an n-node multicomputer.

Furthermore, we assume $g^*(nM) = G(n)g(M) = G(n)W_n$, where $W_n = g(M)$ and g^* is a homogeneous function. The factor $G(n)$ reflects the increase in workload as memory increases n times. Now we are ready to rewrite Eq. 3.32 under the assumption that $W_i = 0$ if $i \neq 1$ or n and $Q(n) = 0$:

$$S_n^* = \frac{W_1^* + W_n^*}{W_1^* + W_n^*/n} = \frac{W_1 + G(n)W_n}{W_1 + G(n)W_n/n} \tag{3.33}$$

Rigorously speaking, the above speedup model is valid under two assumptions: (1) The collection of all memory forms a global address space (in other words, we assume a shared distributed memory space); and (2) All available memory areas are used up for the scaled problem. There are three special cases where Eq. 3.33 can apply:

Case 1: $G(n) = 1$. This corresponds to the case where the problem size is fixed. Thus, the fixed-memory speedup becomes equivalent to Amdahl's law; i.e. Eqs. 3.27 and 3.33 are equivalent when a fixed workload is given.

Case 2: $G(n) = n$. This applies to the case where the workload increases n times when the memory is increased n times. Thus, Eq. 3.33 is identical to Gustafson's law (Eq. 3.30) with a fixed execution time.

Case 3: $G(n) > n$. This corresponds to the situation where the computational workload increases faster than the memory requirement. Thus, the fixed-memory model (Eq. 3.33) will likely give a higher speedup than the fixed-time speedup (Eq. 3.30).

The above analysis leads to the following conclusions: Amdahl's law and Gustafson's law are special cases of the fixed-memory model. When computation grows faster than the memory requirement, as is often true in some scientific simulation and engineering applications, the fixed-memory model (Fig. 3.10) may yield an even higher speedup (i.e., $S_n^* \geq S_n' \geq S_n$) and better resource utilization.

The fixed-memory model also assumes a scaled workload and allows an increase in execution time. The increase in workload (problem size) is memory-bound. The growth in machine size is limited by increasing

communication demands as the number of processors becomes large. The fixed-time model can be moved very close to the fixed-memory model if available memory is fully utilized.

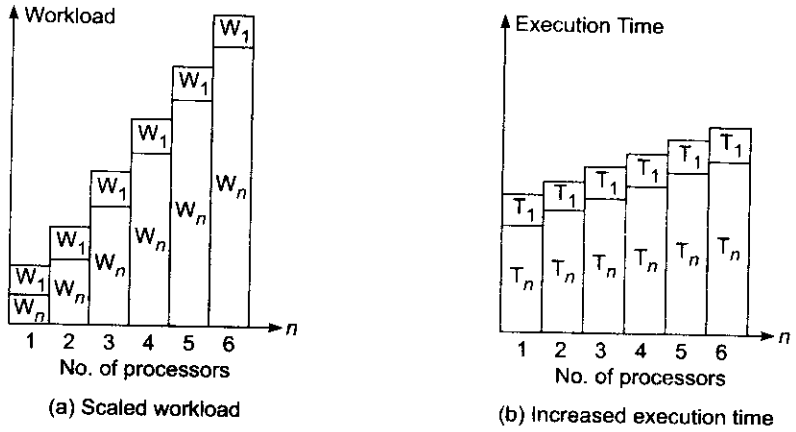


Fig. 3.10 Scaled speedup model using fixed memory (Courtesy of Sun and Ni; reprinted with permission from *ACM Supercomputing*, 1990)

Example 3.6 Scaled matrix multiplication using global versus local computation models (Sun and Ni, 1993)

In scientific computations, a matrix often represents some discretized data continuum. Enlarging the matrix size generally leads to a more accurate solution for the continuum. For matrices with dimension n, the number of computations involved in matrix multiplication is $2n^3$ and the memory requirement is roughly $M = 3n^2$.

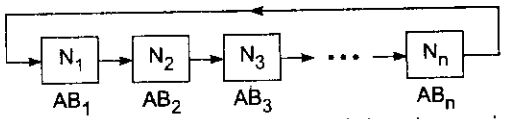
As the memory increases n times in an n-processor multicomputer system, $nM = n \times 3n^2 = 3n^3$. If the enlarged matrix has a dimension of N, then $3n^3 = 3N^2$. Therefore, $N = n^{1.5}$. Thus $G(n) = n^{1.5}$, and the scaled workload $W_n^* = G(n)W_n = n^{1.5}W$. Using Eq. 3.33, we have

$$S^* = \frac{W_1 + n^{1.5}W_n}{W_1 + \dfrac{n^{1.5}W_n}{n}} = \frac{W_1 + n^{1.5}W_n}{W_1 + n^{0.5}W_n} \tag{3.34}$$

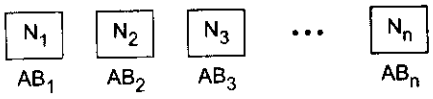
under the *global computation model* illustrated in Fig. 3.11a,. where all the distributed memories are used as. a common memory shared by all processor nodes.

As illustrated in Fig. 3.11b, the node memories are used locally without sharing. In such a *local computation model*, $G(n) = n$, and we obtain the following speedup:

$$S_n^* = \frac{W_1 + nW_n}{W_1 + W_n} \tag{3.35}$$



(a) Global computation with distributed shared memories



(b) Local computation with distributed private memories

Fig. 3.11 Two models for the distributed matrix multiplication

The above example illustrates Gustafson's scaled speedup for local computation. Comparing the above two speedup expressions, we realize that the fixed-memory speedup (Eq. 3.34) may be higher than the fixed-time speedup (Eq. 3.35). In general, many applications demand the use of a combination of local and global addressing spaces. Data may be distributed in some nodes and duplicated in other nodes. Data duplication is added deliberately to reduce communication demand. Speedup factors for these applications depend on the ratio between the global and local computations.

3.4 SCALABILITY ANALYSIS AND APPROACHES

The performance of a computer system depends on a large number of factors, all affecting the scalability of the computer architecture and the application program involved. The simplest definition of *scalability* is that the performance of a computer system increases linearly with respect to the number of processors used for a given application,

Scalability analysis of a given computer system must be conducted for a given application program/ algorithm. The analysis can be performed under different constraints on the growth of the problem size (workload) and on the machine size (number of processors). A good understanding of scalability will help evaluate the performance of parallel computer architectures for large-scale applications.

3.4.1 Scalability Metrics and Goals

Scalability studies determine the degree of matching between a computer architecture and an application algorithm. For different (architecture, algorithm) pairs, the analysis may end up with different conclusions. A machine can be very efficient for one algorithm but bad for another, and vice versa.

Thus, a good computer architecture should be efficient in implementing a large class of application algorithms. In the ideal case, the computer performance should be linearly scalable with an increasing number of processors employed in implementing the algorithms.

Scalability metrics Identified below are the basic metrics (Fig. 3.12) affecting the scalability of a computer system for a given application:

- *Machine size* (n)—the number of processors employed in a parallel computer system. A large machine size implies more resources and more computing power.

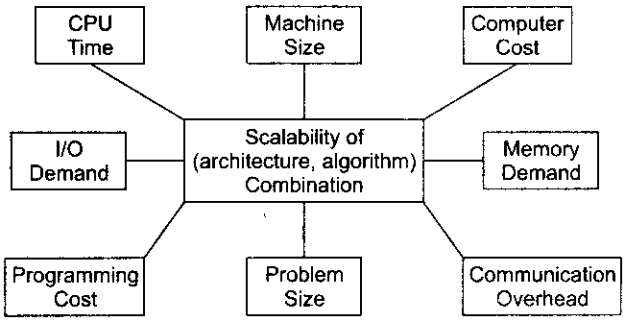


Fig. 3.12 Scalability metrics

- *Clock rate* (f)—the clock rate determines the basic machine cycle. We hope to build a machine with components (processors, memory, bus or network, etc.) driven by a clock which can scale up with better technology.
- *Problem size* (s)—the amount of computational workload or the number of data points used to solve a given problem. The problem size is directly proportional to the *sequential execution time* $T(s, 1)$ for a uniprocessor system because each data point may demand one or more operations.
- *CPU time* (T)—the actual CPU time (in seconds) elapsed in executing a given program on a parallel machine with n processors collectively. This is the *parallel execution time*, denoted as $T(s, n)$ and is a function of both s and n.
- *I/O demand* (d)—the input/output demand in moving the program, data, and results associated with a given application run. The I/O operations may overlap with the CPU operations in a multiprogrammed environment.
- *Memory capacity* (m)—the amount of main memory (in bytes or words) used in a program execution. Note that the memory demand is affected by the problem size, the program size, the algorithms, and the data structures used.

  The memory demand varies dynamically during program execution. Here, we refer to the maximum number of memory words demanded. Virtual memory is almost unlimited with a 64-bit address space. It is the physical memory which may be limited in capacity.
- *Communication overhead* (h)—the amount of time spent for interprocessor communication, synchronization, remote memory access, etc. This overhead also includes all noncompute operations which do not involve the CPUs or I/O devices. This overhead $h(s, n)$ is a function of s and n and is not part of $T(s, n)$. For a uniprocessor system, the overhead $h(s, 1) = 0$.
- *Computer cost* (c)—the total cost of hardware and software resources required to carry out the execution of a program.
- *Programming overhead* (p)—the development overhead associated with an application program. Programming overhead may slow down software productivity and thus implies a high cost. Unless otherwise stated, both computer cost and programming cost are ignored in our scalability analysis.

Depending on the computational objectives and resource constraints imposed, one can fix some of the above parameters and optimize the remaining ones to achieve the highest performance with the lowest cost.

The notion of scalability is tied to the notions of speedup and efficiency. A sound definition of scalability must be able to express the effects of the architecture's interconnection network, of the communication patterns

inherent to algorithms, of the physical constraints imposed by technology, and of the cost effectiveness or system efficiency. We introduce first the notion of speedup and efficiency. Then we define scalability based on the relative performance of a real machine compared with that of an idealized theoretical machine.

Speedup and Efficiency Revisited For a given architecture, algorithm, and problem size s, the *asymptotic speedup* $S(s, n)$ is the best speedup that is attainable, varying only the number (n) of processors. Let $T(s, 1)$ be the sequential execution time on a uniprocessor, $T(s, n)$ be the minimum parallel execution time on an n-processor system, and $h(s, n)$ be the lump sum of all communication and I/O overheads. The asymptotic speedup is formally defined as follows:

$$S(s, n) = \frac{T(s, 1)}{T(s, n) + h(s, n)} \tag{3.36}$$

The problem size is the independent parameter, upon which all other metrics are based. A meaningful measurement of asymptotic speedup mandates the use of a good sequential algorithm, even it is different from the structure of the corresponding parallel algorithm. The $T(s, n)$ is minimal in the sense that the problem is solved using as many processors as necessary to achieve the minimum runtime for the given problem size.

In scalability analysis, we are mainly interested in results obtained from solving large problems. Therefore, the run times $T(s, n)$ and $T(s, 1)$ should be expressed using order-of-magnitude notations, reflecting the asymptotic behavior.

The *system efficiency* of using the machine to solve a given problem is defined by the following ratio:

$$E(s, n) = \frac{S(s, n)}{n} \tag{3.37}$$

In general, the best possible efficiency is one, implying that the best speedup is linear, or $S(s, n) = n$. Therefore, an intuitive definition of scalability is: *A system is scalable if the system efficiency* $E(s, n) = 1$ *for all algorithms with any number of n processors and any problem size s.*

Mark Hill (1990) has indicated that this definition is too restrictive to be useful because it precludes any system from being called scalable. For this reason, a more practical efficiency or scalability definition is needed, comparing the performance of the real machine with respect to the theoretical PRAM model.

Scalability Definition Nussbaum and Agarwal (1991) have given the following scalability definition based on a PRAM model. The scalability $\Phi(s, n)$ of a machine for a given algorithm is defined as the ratio of the asymptotic speedup $S(s, n)$ on the real machine to the asymptotic speedup $S_I(s, n)$ on the ideal realization of an EREW PRAM.

$$S_I(s, n) = \frac{T(s, 1)}{T_I(s, n)}$$

where $T_I(s, n)$ is the parallel execution time on the PRAM, ignoring all communication overhead. The scalability is defined as follows:

$$\Phi(s, n) = \frac{S(s, n)}{S_I(s, n)} = \frac{T_I(s, n)}{T(s, n)} \tag{3.38}$$

Intuitively, the larger the scalability, the better the performance that the given architecture can yield running the given algorithm. In the ideal case, $S_I(s, n) = n$, the scalability definition in Eq. 3.38 becomes identical to the efficiency definition given in Eq. 3.37.

Example 3.7 Scalability of various machine architectures for parity calculation (Nussbaum and Agarwal, 1991)

Table 3.4 shows the execution times, asymptotic speedups, and scalabilities (with respect to the EREW-PRAM model) of five representative interconnection architectures: linear array, 2-D and 3-D meshes, hypercube, and Omega network, for running a parallel parity calculation.

Table 3.4 *Scalability of Various Network-Based Architectures for the Parity Calculation*

Metrics	*Machine Architecture*				
	Linear array	*2-D mesh*	*3-D mesh*	*Hypercube*	*Omega Network*
$T(s, n)$	$s^{1/2}$	$s^{1/3}$	$s^{1/4}$	$\log s$	$\log^2 s$
$S(s, n)$	$s^{1/2}$	$s^{2/3}$	$s^{3/4}$	$s/\log s$	$s/\log^2 s$
$\Phi(s, n)$	$\log s/s^{1/2}$	$\log s/s^{1/3}$	$\log s/s^{1/4}$	1	$1/\log s$

This calculation examines s bits, determining whether the number of bits set is even or odd using a balanced binary tree. For this algorithm, $T(s, 1) = s$, $T_I(s, n) = \log s$, and $S_I(s, n) = s/\log s$ for the ideal PRAM machine.

On real architectures, the parity algorithm's performance is limited by network diameter. For example, the linear array has a network diameter equal to $n - 1$, yielding a total parallel running time of $s/n + n$. The optimal partition of the problem is to use $n = \sqrt{s}$ processors so that each processor performs the parity check on $\sqrt{s}$ bits locally. This partition gives the best match between computation costs and communication costs with $T(s, n) = s^{1/2}$, $S(s, n) = s^{1/2}$ and thus scalability $\Phi(s, n) = \log s/s^{1/2}$.

The 2D and 3D mesh architectures use a similar partition to match their own communication structure with the computational loads, yielding even better scalability results. It is interesting to note that the scalability increases as the communication latency decreases in a network with a smaller diameter.

The hypercube and the Omega network provide richer communication structures (and lower diameters) than meshes of lower dimensionality. The hypercube does as well as a PRAM for this algorithm, yielding $\Phi(s, n) = 1$.

The Omega network (Fig. 2.24) does not exploit locality: communication with all processors takes the same amount of time. This loss of locality hurts its performance when compared to the hypercube, but its lower diameter gives it better scalability than any of the meshes.

Although performance is limited by network diameter for the above parity algorithm, for many other algorithms the network bandwidth is the performance-limiting factor. The above analysis assumed unit communication time between directly connected communication nodes. An architecture may be scalable for one algorithm but unscalable for another. One must examine a large class of useful algorithms before drawing a scalability conclusion on a given architecture.

3.4.2 Evolution of Scalable Computers

The idea of massive parallelism is rather old, the technology is advancing steadily, and the software is relatively unexplored, as was observed by Cybenko and Kuck (1992). One evolutional trend is to build scalable supercomputers with distributed shared memory and standardized UNIX/LINUX for parallel processing. In this section, we present the evolutional path and some scalable computer design concepts; recent advances in this direction are discussed in Chapter 13.

The Evolutional Path Figure 3.13 shows the early evolution of supercomputers with four-to-five-year gestation and of micro-based scalable computers with three-year gestation. This plot shows the peak performance of Cray and NEC supercomputers and of Cray, Intel, and Thinking Machines scalable computers versus the introduction year. The marked nodes correspond to machine models with increasing size and cost.

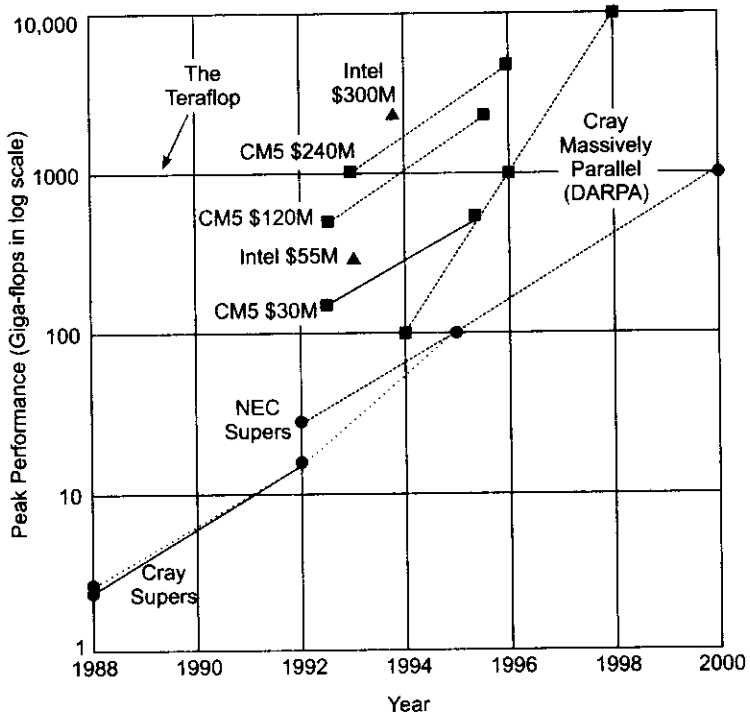


Fig. 3.13 The performance (in Gftops) of various computers manufactured during 1990s by Cray Research, Inc., NEC, Intel, and Thinking Machines Corporation (Courtesy of Gordon Bell; reprinted with permission from the *Communications of ACM*, August 1992)[1]

In 1988, the Cray Y-MP 8 delivered a peak of 2.8 Gflops. By 1991, the Intel Touchstone Delta, a 672-node multicomputer, and the Thinking Machines CM-2, a 2K PE SIMD machine, both began to supply an order-of-magnitude greater peak power (20 Gflops) than conventional supercomputers. By mid-1992, a completely new generation of computers were introduced, including the CM-5 and Paragon.

[1] Thinking Machines Corporation has since gone out of business.

In the past, the IBM System/360 provided a 100:1 range of growth for its various models. DEC VAX machines spanned a range of 1000:1 over their lifetime. Based on past experiences, Gordon Bell has identified three objectives for designing scalable computers. Implications and case studies of these challenges will be further discussed in subsequent chapters.

Size Scalability The study of system scalability started with the desire to increase the machine size. A size-scalable computer is designed to have a scaling range from a small to a large number of resource components. The expectation is to achieve linearly increased performance with incremental expansion for a well-defined set of applications. The components include computers, processors or processing elements, memories, interconnects, switches, cabinets, etc.

Size scalability depends on spatial and temporal locality as well as component bottleneck. Since very large systems have inherently longer latencies than small and centralized systems, the locality behavior of program execution will help tolerate the increased latency. Locality will be characterized in Chapter 4. The bottleneck-free condition demands a balanced design among processing, storage, and I/O bandwidth.

For example, since MPPs are mostly interconnected by large networks or switches, the bandwidth of the switch should increase linearly with processor power. The I/O demand may exceed the processing bandwidth in some real-time and large-scale applications.

The Cray Y-MP series scaled over a range of 16 processors (the C-90 model) and the current range of Cray supercomputers offer a much larger range of scalability (see Chapter 13). The CM-2 was designed to scale between 8K and 64K processing elements. The CM-5 scaling range was 1024 to 16K computers. The KSR-1 had a range of 8 to 1088 processor-memory pairs. Size-scalability cannot be achieved alone without considering cost, efficiency, and programmability on reasonable time scale.

Generation (Time) Scalability Since the basic processor nodes become obsolete every three years, the time scalability is equally important as the size scalability. Not only should the hardware technology be scalable, such as the CMOS circuits and packaging technologies in building processors and memory chips, but also the software/algorithm which demands software compatibility and portability with new hardware systems.

DEC claimed that the Alpha microprocessor was generation-scalable for 25 years. In general, all computer characteristics must scale proportionally: processing speed, memory speed and size, interconnect bandwidth and latency, I/O, and software overhead, in order to be useful for a given application.

Problem Scalability The problem size corresponds to the data set size. This is the key to achieving scalable performance as the program granularity changes. A problem scalable computer should be able to perform well as the problem size increases. The problem size can be scaled to be sufficiently large in order to operate efficiently on a computer with a given granularity.

Problems such as Monte Carlo simulation and ray tracing are "perfectly parallel", since their threads of computation do not come together over long spells of computation. Such an independence among threads is very much desired in using a scalable MPP system. In general, the *problem granularity* (operations on a grid point/data required from adjacent grid points) must be greater than a *machine's granularity* (node operation rate/node-to-node communication data rate) in order for a multicomputer to be effective.

Example 3.8 Problem scaling for solving Laplace equation on a distributed memory multicomputer (Gordon Bell, 1992)

Laplace equations are often used to model physical structures. A 3-D Laplace equation is specified by

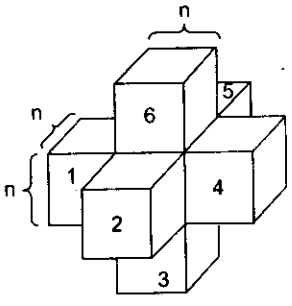
$$\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} = 0 \tag{3.39}$$

We want to determine the problem scalability of the Laplace equation solver on a distributed-memory multicomputer with a sufficiently large number of processing nodes. Based on finite-difference method, solving Eq. 3.39 requires performing the following averaging operation iteratively across a very large grid, as shown in Fig. 3.14:

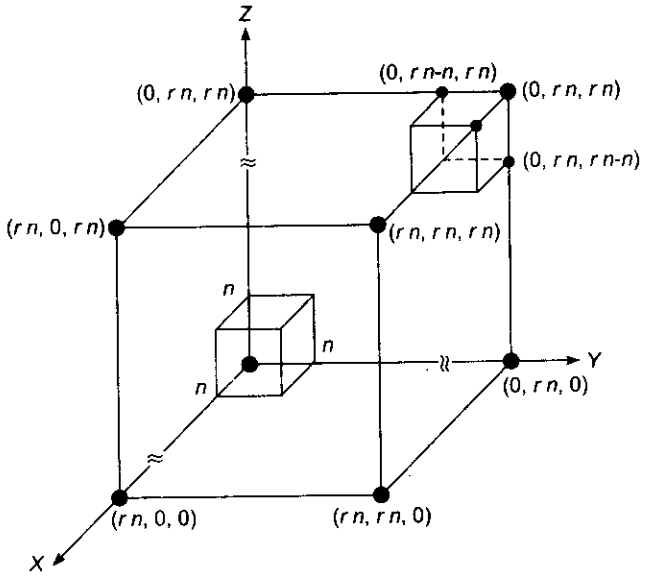
$$u_{i,j,k}^{(m)} = \frac{1}{6}\left[u_{i-1,j,k}^{(m-1)} + u_{i+1,j,k}^{(m-1)} + u_{i,j-1,k}^{(m-1)} + u_{i,j+1,k}^{(m-1)} + u_{i,j,k-1}^{(m-1)} + u_{i,j,k+1}^{(m-1)}\right] \tag{3.40}$$

where $1 \le i, j, k \le N$ and N is the number of grid points along each dimension. In total, there are N^3 grid points in the problem domain to be evaluated during each iteration m for $1 \le m \le M$.

The three-dimensional domain can be partitioned into p subdomains, each having n^3 grid points such that $pn^3 = N^3$, where p is the machine size. The computations involved in each subdomain are assigned to one node of a multicomputer. Therefore, in each iteration, each node is required to perform $7n^3$ computations as specified in Eq. 3.40.



(a) Six cube subdomains adjacent to a cube subdomain at the center



(b) An $N \times N \times N$ grid partitioned into p subdomains, each being an n^3 cube, where $p = r^3 = N^3/n^3$

Fig. 3.14 Partitioning of a 3D domain for solving the Laplace equation

Each subdomain is adjacent to six other subdomains (Fig. 3.14a). Therefore, in each iteration, each node needs to exchange (send or receive) a total of $6n^2$ words of floating-point numbers with its neighbors. Assume each floating-point number is double-precision (64 bits, or 8 bytes). Each processing node has the capability of performing 100 Mflops (or 0.01 μs per floating-point operation). The internode communication latency is assumed to be 1 μs (or 1 megaword/s) for transferring a floating-point number.

For a balanced multicomputer, the computation time within each node and inter-node communication latency should be equal. Thus $0.07n^3\mu$s equals $6n^2$ μs communication latency, implying that n has to be at least as large as 86. A node memory of capacity $86^3 \times 8 = 640\text{K} \times 8 = 5120$ Kwords = 5 megabytes is needed to hold each subdomain of data.

On the other hand, suppose each message exchange takes 2 μs (one receive and one send) per word. The communication latency is doubled. We desire to scale up the problem size with an enlarged local memory of 32 megabytes. The subdomain dimension size n can be extended to at most 160, because $160^3 \times 8 = 32$ megabytes. This size problem requires 0.3 s of computation time and 2×0.15 s of send and receive time. Thus each iteration takes 0.6 (0.3 + 0.3) s, resulting in a computation rate of 50 Mflops, which is only 50% of the peak speed of each node.

If the problem size n is further increased, the effective Mflops rate and efficiency will be improved. But this cannot be achieved unless the memory capacity is further enlarged. For a fixed memory capacity, the situation corresponds to the memorybound region shown in Fig. 3.6c. Another risk of problem scaling is to exacerbate the limited I/O capability which is not demonstrated in this example.

To summarize the above studies on scalability, we realize that the machine size, problem size, and technology scalabilities are not necessarily orthogonal to each other. They must be considered jointly. In the next section, we will identify additional issues relating scalability studies to software compatibility, latency tolerance, machine programmability, and cost-effectiveness.

3.4.3 Research Issues and Solutions

Toward the development of truly scalable computers, much research is being done. In this section, we briefly identify several frontier research problems. Partial solutions to these problems will be studied in subsequent chapters.

The Problems When a computer is scaled up to become an MPP system, the following difficulties can arise:

- Memory-access latency becomes too long and too nonuniformly distributed to be considered tolerable.
- The IPC complexity or synchronization overhead becomes too high to be useful.
- The multicache inconsistency problem becomes out of control.
- The processor utilization rate deteriorates as the system size becomes large.
- Message passing (or page migration) becomes too time-consuming to benefit resource sharing in a large distributed system.
- Overall system performance becomes saturated with diminishing return as system size increases further.

Some Approaches In order to overcome the above difficulties, listed below are some approaches being pursued by researchers:

- Searching for latency reducing and fast synchronization techniques.
- Using weaker memory consistency models.
- Developing scalable cache coherence protocols.
- Realizing shared virtual memory system.
- Integrating multithreaded architectures for improved processor utilization and system throughput.
- Expanding software portability and standardizing parallel and distributed UNIX/LINUX systems.

Scalability analysis can be carried out either by analytical methods or through trace-driven simulation experiments. In Chapter 9, we will study both approaches toward the development of scalable computer architectures that match program/ algorithmic behaviors. Analytical tools include the use of Markov chains, Petri nets, or queueing models. A number of simulation packages have already been developed at Stanford University and at MIT.

Supporting Issues Besides the emphases of scalability on machine size, problem size and technology, we identify below several extended areas for continued research and development:

(1) *Software scalability*: As problem size scales in proportion to the increase in machine size, the algorithms can be optimized to match the architectural constraints. Software tools are being developed to help programmers in mapping algorithms onto a target architecture.

A perfect match between architecture and algorithm requires matching both computational and communication patterns through performance-tuning experiments in addition to simple numerical analysis. Optimizing compilers and visualization tools should be designed to reveal opportunities for algorithm/program restructuring to match with the architectural growth.

(2) *Reducing communication overhead:* Scalability analysis should concern both useful computations and available parallelism in programs. The most difficult part of the analysis is to estimate the communication overhead accurately. Excessive communication overhead, such as the time required to synchronize a large number of processors, wastes system resources. This overhead grows rapidly as machine size and problem size increase.

Furthermore, the run time conditions are often difficult to capture. How to reduce the growth of communication overhead and how to tolerate the growth of memory-access latency in very large systems are still wide-open research problems.

(3) *Enhancing programmability*: The computing community generally agrees that multicomputers are more scalable; multiprocessors may be more easily programmed but are less scalable than multicomputers. It is the centralized-memory versus distributed private-memory organization that makes the difference. In the ideal case, we want to build machines which will retain the advantages of both architectures. This implies a system with shared distributed memory and simplified message communication among processor nodes. Hsterogencous programming paradigms are needed for future systems.

(4) *Providing longevity and generality*: Other scalability issues include *longevity*, which requires an architecture with sufficiently large address space, and *generality*, which supports a wide variety of languages and binary migration of software.

Performance, scalability, programmability, and generality will be studied throughout the book for general-purpose parallel processing applications, unless otherwise noted.

Summary

With rapid advances in technology, scalability becomes an important criterion for any modern computer system—and especially so for a parallel processing system. However, system scalability can only be defined in terms of system performance, and therefore issues of scalability and system performance are very closely interrelated. In this chapter, we have studied some basic issues related to the performance and scalability of parallel processing systems.

The main performance metric considered is the execution time of a parallel program which has a specific parallelism profile. As a program executes, the degree of parallelism in it varies with time, and therefore we can calculate the average degree of parallelism in the program. The parallelism profile also allows us to estimate the speedup achievable on the system as the number of processors is increased.

Apart from speedup, we also defined system efficiency and system utilization as asymptotic functions of the number of processors. On an n processor system, efficiency is defined as the speedup achieved divided by n (which is the ideal case speedup). System utilization, on the other hand, indicates the fraction of processor cycles which was actually utilized during program execution on the n processor system.

Benchmark programs are very useful tools in measuring the performance of computer systems. We looked at certain well-known benchmark programs, although it is also true that no two applications are identical and that therefore, in the final analysis, application specific benchmark programs are more useful.

We took a brief look at so-called 'grand challenge' applications of high performance computer systems; these are applications which are likely to have major impact in science and technology. Massively parallel processing (MPP) systems are increasingly being applied to such problems; clearly performance and scalability are important criteria for all such applications.

We then looked at some speedup performance laws governing parallel applications. Amdahl's law states in essence that, for a problem of a given fixed size, as the number of processors is increased, the speedup achievable is limited by the program fraction which must necessarily run as a sequential program, i.e. on one processor. Gustafson's law, on the other hand, studies also the effect of increasing the problem size as the system size is increased, resulting in the so-called fixed time speedup model. The third model studied was the memory-bounded speedup model proposed by Sun and Ni.

The specific metrics which affect the scalability of a computer system for a given application are—machine size in number of processors, processor clock rate, problem size, processor time consumed, I/O requirement, memory requirement, communication requirement, system cost, and programming cost of the application. Open research issues related to scalability in massively parallel systems were reviewed.

Exercises

Problem 3.1 Consider the parallel execution of the same program in Problem 1.4 on a four-processor system with shared memory. The program can be partitioned into four equal parts for balanced execution by the four processors. Due to the need for synchronization among the four program parts, 50000 extra instructions are added to each divided program part.

Assume the same instruction mix as in Problem 1.4 for each divided program part.

The CPI for the memory reference (with cache miss) instructions has been increased from 8 to 12 cycles due to contentions. The CPIs for the remaining instruction types do not change.

(a) Repeat part (a) in Problem 1.4 when the program is executed on the four-processor system.

(b) Repeat part (b) in Problem 1.4 when the program is executed on the four-processor system.

(c) Calculate the speedup factor of the four-processor system over the uniprocessor system in Problem 1.4 under the respective trace statistics.

(d) Calculate the efficiency of the four-processor system by comparing the speedup factor in part (c) with the ideal case.

Problem 3.2 A uniprocessor computer can operate in either scalar or vector mode. In vector mode, computations can be performed nine times faster than in scalar mode. A certain benchmark program took time T to run on this computer. Further, it was found that 25% of T was attributed to the vector mode. In the remaining time, the machine operated in the scalar mode.

(a) Calculate the effective speedup under the above condition as compared with the condition when the vector mode is not used at all. Also calculate α, the percentage of code that has been vectorized in the above program.

(b) Suppose we double the speed ratio between the vector mode and the scalar mode by hardware improvements. Calculate the effective speedup that can be achieved.

(c) Suppose the same speedup obtained in part (b) must be obtained by compiler improvements instead of hardware improvements. What would be the new vectorization ratio α that should be supported by the vectorizing compiler for the same benchmark program?

Problem 3.3 Let α be the percentage of a program code which can be executed simultaneously by n processors in a computer system. Assume that the remaining code must be executed sequentially by a single processor. Each processor has an execution rate of x MIPS, and all the processors are assumed equally capable.

(a) Derive an expression for the effective MIPS rate when using the system for exclusive execution of this program, in terms of the parameters n, α, and x.

(b) If $n = 16$ and $x = 400$ MIPS, determine the value of α which will yield a system performance of 4000 MIPS.

Problem 3.4 Consider a computer which can execute a program in two operational modes: *regular mode* versus *enhanced mode*, with a probability distribution of $\{\alpha, 1 - \alpha\}$, respectively.

(a) If α varies between a and b and $0 \leq a < b \leq 1$, derive an expression for the *average speedup factor* using the harmonic mean concept.

(b) Calculate the speedup factor when $a \rightarrow 0$ and $b \rightarrow 1$.

Problem 3.5 Consider the use of a four-processor, shared-memory computer for the execution of a program mix. The multiprocessor can be used in four execution modes corresponding to the active use of one, two, three, and four processors. Assume that each processor has a peak execution rate of 500 MIPS.

Let f_i be the percentage of time that i processors will be used in the above program execution and $f_1 + f_2 + f_3 + f_4 = 1$. You can assume the execution rates R_1, R_2, R_3, and R_4, corresponding to the distribution (f_1, f_2, f_3, f_4), respectively.

(a) Derive an expression to show the harmonic mean execution rate R of the multiprocessor in terms of f_i and R_i for $i = 1, 2, 3, 4$. Also show an expression for the harmonic mean execution time T in terms of R.

(b) What would be the value of the harmonic mean execution time T of the above program mix given $f_1 = 0.4, f_2 = 0.3, f_3 = 0.2, f_4 = 0.1$ and $R_1 = 400$ MIPS, $R_2 = 800$ MIPS, $R_3 = 1100$ MIPS, $R_4 = 1500$ MIPS? Explain the possible causes of observed R_i values in the above program execution.

(c) Suppose an intelligent compiler is used to enhance the degree of parallelization in the above program mix with a new distribution $f_1 = 0.1$, $f_2 = 0.2$, $f_3 = 0.3$, $f_4 = 0.4$. What would be the harmonic mean execution time of the same program under the same assumption on $\{R_i\}$ as in part (b)?

Problem 3.6 Explain the applicability and the restrictions involved in using Amdahl's law, Gustafson's law, and Sun and Ni's law to estimate the speedup performance of an n-processor system compared with that of a single-processor system. Ignore all communication overheads.

Problem 3.7 The following Fortran program is to be executed on a uniprocessor, and a parallel version is to be executed on a shared-memory multiprocessor.

```
L1:      Do 10 I = 1, 1024
L2:          SUM(I) = 0
L3:          Do 20 J = 1, I
L4:  20          SUM (I) = SUM (I) + I
L5:  10  Continue
```

Suppose statements 2 and 4 each take two machine cycle times, including all CPU and memory-access activities. Ignore the overhead caused by the software loop control (statements L1, L3, and L5) and all other system overhead and resource conflicts.

(a) What is the total execution time of the program on a uniprocessor?

(b) Divide the outer loop iterations among 32 processors with prescheduling as follows: Processor 1 executes the first 32 iterations ($I = 1$ to 32), processor 2 executes the next 32 iterations ($I = 33$ to 64), and so on. What are the execution time and speedup factors compared with part (a)? (Note that the computational workload, dictated by the J-loop, is unbalanced among the processors.)

(c) Modify the given program to facilitate a balanced parallel execution of all the computational workload over 32 processors. By a balanced load, we mean an equal number of additions assigned to each processor with respect to both loops.

(d) What is the minimum execution time resulting from the balanced parallel execution on 32 processors? What is the new speedup over the uniprocessor?

Problem 3.8 Consider the multiplications of two $n \times n$ matrices $A = (a_{ij})$ and $B = (b_{ij})$ on a scalar uniprocessor and on a multiprocessor, respectively. The matrix elements are floating-point numbers, initially stored in the main memory in row-major order. The resulting product matrix $C = (c_{ij})$ where $C = A \times B$, should be stored back to memory in contiguous locations.

Assume a 2-address instruction format and an instruction set of your choice. Each load/store instruction takes, on the average, 4 cycles to complete. All ALU operations must be done sequentially on the processor with 2 cycles if no memory reference is required in the instruction. Otherwise, 4 cycles are added for each memory reference to fetch an operand. Branch-type instructions require, on the average, 2 cycles.

(a) Write a minimal-length assembly-language program to perform the matrix multiplication on a scalar processor with a load-store architecture and floating-point hardware.

(b) Calculate the total instruction count, the total number of cycles needed for the program execution, and the average cycles per instruction (CPI).

(c) What is the MIPS rate of this scalar machine, if the processor is driven by a 400-MHz clock?

(d) Suggest a partition of the above program to execute the divided program parts on an N-processor shared-memory system with minimum time. Assume $n = 1000N$. Estimate the potential speedup of the multiprocessor over the uniprocessor, assuming the same type of processors are used in both systems. Ignore the memory-access conflicts, synchronization and other overheads.

(e) Sketch a scheme to perform distributed matrix computations with distributed data sets on an N-node multicomputer with distributed memory. Each node has a computer equivalent to the scalar processor used in part (a).

(f) Specify the message-passing operations required in part (e). Suppose that, on the average, each message passing requires 100 processor cycles to complete. Estimate the total execution time on the multicomputer for the distributed matrix multiplication. Make appropriate assumptions if needed in your timing analysis.

Problem 3.9 Consider the interleaved execution of the four programs in Problem 1.6 on each of the three machines. Each program is executed in a particular mode with the measured MIPS rating.

(a) Determine the arithmetic mean execution time per instruction for each machine executing the combined workload, assuming equal weights for the four programs.

(b) Determine the harmonic mean MIPS rate of each machine.

(c) Rank the machines based on the harmonic mean performance. Compare this ranking with that obtained in Problem 1.6.

Problem 3.10 Answer or prove the following statements related to speedup performance law:

(a) Derive the fixed-memory speedup expression S_n^* in Eq. 3.33 under reasonable assumptions.

(b) Derive Amdahl's law (S_n in Eq. 3.14) as a special case of the S_n^* expression.

(c) Derive Gustafson's law (S'_n in Eq. 3.31) as a special case of the S_n^* expression.

(d) Prove the relation $S_n^* \geq S'_n \geq S_n$ for solving the same problem on the same machine under different assumptions.

Problem 3.11 Prove the following relations among the speedup $S(n)$, efficiency $E(n)$, utilization $U(n)$, redundancy $R(n)$, and quality $Q(n)$ of a parallel computation, based on the definitions given by Lee (1980):

(a) Prove $1/n \leq E(n) \leq U(n) \leq 1$, where n is the number of processors used in the parallel computation.

(b) Prove $1 \leq R(n) \leq 1/E(n) \leq n$.

(c) Prove the expression for $Q(n)$ in Eq. 3.19.

(d) Verify the above relations using the hypothetical workload in Example 3.3.

Problem 3.12 Repeat Example 3.7 for sorting s numbers on five different n-processor machines using the linear array, 2D-mesh, 3D-mesh, hypercube, and Omega network as interprocessor communication architectures, respectively.

(a) Show the scalability of the five architectures as compared with the EREW-PRAM model.

(b) Compare the results obtained in part (a) with those in Example 3.7. Based on these two benchmark results, rank the relative scalability of the five architectures. Can the results be generalized to the performance of other algorithms?

Problem 3.13 Consider the execution of two benchmark programs. The performance of three computers running these two benchmarks are given below:

Benchmark	*Millions of floating-point operations*	*Computer 1* T_1 *(sec.)*	*Computer 2* T_2 *(sec.)*	*Computer 3* T_3 *(sec.)*
Problem 1	100	1	10	20
Problem 2	100	1000	100	20
Total time		1001	110	40

(a) Calculate R_a and R_h for each computer under the equal-weight assumption $f_1 = f_2 = 0.5$.

(b) When benchmark 1 has a constant R_1 = 10 Mflops performance across the three computers, plot R_a and R_h as a function of R_2, which varies from 1 to 100 Mflops under the assumption $f_1 = 0.8$ and $f_2 = 0.2$.

(c) Repeat part (b) for the case $f_1 = 0.2$ and $f_2 = 0.8$.

(d) From the above performance results under different conditions, can you draw a conclusion regarding the relative performance of the three machines?

Problem 3.14 In Example 3.5, four parallel algorithms are mentioned for multiplication of $s \times s$ matrices. After reading the original papers describing these algorithms, prove the following communication overheads on the target machine architectures:

(a) Prove that $h(s, n) = O(n \log n + s^2\sqrt{n})$ when mapping the Fox-Otto-Hey algorithm on a $\sqrt{n} \times \sqrt{n}$ torus.

(b) Prove that $h(s, n) = O(n^{4/3} + n \log n + s^2 n^{1/3})$ when mapping Berntsen's algorithm on a hypercube with $n = 2^{3k}$ nodes, where $k \leq \frac{1}{2} \log s$.

(c) Prove that $h(s, n) = O(n \log n + s^3)$ when mapping the Dekel-Nassimi-Sahni algorithm on a hypercube with $n = s^3 = 2^{3k}$ nodes.

Problem 3.15 Xian-He Sun (1992) has introduced an *isospeed* concept for scalability analysis. The concept is to maintain a fixed speed for each processor while increasing the problem size. Let W and W' be two workloads corresponding to two problem sizes. Let N and N' be two machine sizes (in terms of the number of processors). Let T_N and $T_{N'}$ be the parallel execution times using N and N' processors, respectively.

The isospeed is achieved when $W/(NT_N) = W'/(N'T_{N'})$. The *isoefficiency* concept defined by Kumar and Rao (1987) is achieved by maintaining a fixed efficiency through $S_N(W)/N = S_{N'}(W')/N'$, where $S_N(W)$ and $S_{N'}(W')$ are the corresponding speedup factors.

Prove that the two concepts are indeed equivalent if (i) the speedup factors are defined as the ratio of *parallel speed* R_N to *sequential speed* R_1 (rather than as the ratio of sequential execution time to parallel execution time), and (ii) $R_1(W) = R_1(W')$. In other words, isoefficiency is identical to isospeed when the sequential speed is fixed as the problem size is increased.