As demonstrated by nature, the ability to work in parallel as a group represents a very efficient way

to reach a common target; human beings have learned to aggregate themselves and to assemble

man-made devices in organizations in which each entity may have modest ability, but a network of entities can organize themselves to accomplish goals that an individual entity cannot. Thus, we should

not be surprised that the thought that individual systems should work in concert to solve complex

applications was formulated early on in the computer age.

Parallel computing allows us to solve large problems by splitting them into smaller ones and solving

them concurrently. Parallel computing was considered for many years the “holy grail” for solving

data-intensive problems encountered in many areas of science, engineering, and enterprise computing;

it required major advances in several areas, including algorithms, programming languages and environments,

performance monitoring, computer architecture, interconnection networks, and last but not

least, solid-state technologies.

Parallel hardware and software systems allow us to solve problems demanding more resources than

those provided by a single system and, at the same time, to reduce the time required to obtain a solution.

The speed-up measures the effectiveness of parallelization; in the general case the speed-up of the

parallel computation is defined as

S(N) = T (1)

T (N)

, (2.1)

with T (1) the execution time of the sequential computation and T (N) the execution timewhen N parallel

computations are carried out. Amdahl’s Law1 gives the potential speed-up of a parallel computation; it

states that the portion of the computation that cannot be parallelized determines the overall speed-up.

If α is the fraction of running time a sequential program spends on nonparallelizable segments of the

computation, then

S = 1

α

. (2.2)

To prove this result, call σ the sequential time and π the parallel time and start from the definitions

of T (1), T (N), and α:

T (1) = σ + π, T (N) = σ + π

N

, and α = σ

σ + π

. (2.3)

Then

S = T (1)

T (N)

= σ + π

σ + π/N

= 1 + π/σ

1 + (π/σ )   (1/N)

. (2.4)

But

π/σ = 1 − α

α

. (2.5)

Thus, for large N

S = 1 + (1 − α)/α

1 + (1 − α)/(Nα)

= 1

α + (1 − α)/N

≈ 1

α

. (2.6)

Amdahl’s law applies to a fixed problem size; in this case the amount of work assigned to each one of

the parallel processes decreases when the number of processes increases, and this affects the efficiency

of the parallel execution.

When the problem size is allowed to change, Gustafson’s Law gives the scaled speed-up with N

parallel processes as

S(N) = N − α(N − 1). (2.7)

As before, we call σ the sequential time; now π is the fixed parallel time per process; α is given by

Equation 2.3. The sequential execution time, T (1), and the parallel execution time with N parallel

processes, T (N), are

T (1) = σ + Nπ and T (N) = σ + π. (2.8)

Then the scaled speed-up is

S(N) = T (1)

T (N)

= σ + Nπ

σ + π

= σ

σ + π

+ Nπ

σ + π

= α + N(1 − α) = N − α(N − 1). (2.9)

Amdahl’s Law expressed by Equation 2.2 and the scaled speed-up given by Equation 2.7 assume that

all processes are assigned the same amount of work. The scaled speed-up assumes that the amount

of work assigned to each process is the same, regardless of the problem size. Then, to maintain the

same execution time, the number of parallel processes must increase with the problem size. The scaled

speed-up captures the essence of efficiency, namely that the limitations of the sequential part of a code

can be balanced by increasing the problem size.

Coordination of concurrent computations could be quite challenging and involves overhead, which

ultimately reduces the speed-up of parallel computations. Often the parallel computation involves multiple

stages, and all concurrent activities must finish one stage before starting the execution of the next

one; this barrier synchronization further reduces the speed-up.

The subtasks of a parallel program are called processes, whereas threads are lightweight subtasks.

Concurrent execution could be very challenging (e.g., it could lead to race conditions, an undesirable

effect in which the results of concurrent execution depend on the sequence of events). Often, shared

resources must be protected by locks to ensure serial access. Another potential problem for concurrent

execution of multiple processes or threads is the presence of deadlocks; a deadlock occurs when processes

or threads competing with one another for resources are forced to wait for additional resources

held by other processes or threads and none of the processes or threads can finish. The four Coffman

conditions must hold simultaneously for a deadlock to occur:

1. Mutual exclusion. At least one resource must be nonsharable, and only one process/thread may use

the resource at any given time.

2. Hold and wait. At least one process/thread must hold one or more resources and wait for others.

3. No preemption. The scheduler or a monitor should not be able to force a process/thread holding a

resource to relinquish it.

4. Circular wait. Given the set of n processes/threads {P1, P2, P3, . . . , Pn}, P1 should wait for a

resource held by P2, P2 should wait for a resource held by P3, and so on and Pn should wait

for a resource held by P1.

There are other potential problems related to concurrency. When two or more processes or threads

continually change their state in response to changes in the other processes, we have a livelock condition;

the result is that none of the processes can complete its execution. Very often processes/threads running

concurrently are assigned priorities and scheduled based on these priorities. Priority inversion occurs

when a higher-priority process or task is indirectly preempted by a lower-priority one.

Concurrent processes/tasks can communicate using messages or shared memory. Multicore processors

sometimes use shared memory, but the shared memory is seldom used in modern supercomputers

because shared-memory systems are not scalable. Message passing is the communication method used

exclusively in large-scale distributed systems, and our discussion is restricted to this communication

paradigm.

Shared memory is extensively used by the system software; the stack is an example of shared memory

used to save the state of a process or thread. The kernel of an operating system uses control structures

such as processor and core tables for multiprocessor and multicore system management, process and

thread tables for process/thread management, page tables for virtual memory management, and so

on. Multiple application threads running on a multicore processor often communicate via the shared

memory of the system. Debugging amessage-passing application is considerably easier than debugging

a shared memory application.

We distinguish fine-grain from coarse-grain parallelism; in the former case relatively small blocks

of the code can be executed in parallel without the need to communicate or synchronize with other

threads or processes, while in the latter case large blocks of code can be executed in parallel. The speedup

of applications displaying fine-grain parallelism is considerably lower than that of coarse-grained

applications; indeed, the processor speed is orders of magnitude higher than the communication speed,

even on systems with a fast interconnect.

In many cases, discovering parallelism is quite challenging, and the development of parallel algorithms

requires a considerable effort. For example, many numerical analysis problems, such as solving

large systems of linear equations or solving systems of partial differential equations (PDEs), requires

algorithms based on domain decomposition methods.

Data parallelism is based on partitioning the data into several blocks and running multiple copies of

the same program concurrently, each running on a different data block – thus the name of the paradigm,

Same Program Multiple Data (SPMD).

Decomposition of a large problem into a set of smaller problems that can be solved concurrently is

sometimes trivial. For example, assume that we want to manipulate the display of a three-dimensional

object represented as a 3D lattice of (n   n   n) points; to rotate the image we would apply the

same transformation to each one of the n3 points. Such a transformation can be done by a geometric

engine, a hardware component that can carry out the transformation of a subset of n3 points

concurrently.

Suppose that we want to search for the occurrence of an object in a set of n images, or of a string

of characters in n records; such a search can be conducted in parallel. In all these instances the time

required to carry out the computational task using N processing elements is reduced by a factor of N.

A very appealing class of applications of cloud computing is numerical simulations of complex

systems that require an optimal design; in such instances multiple design alternatives must be compared

and optimal ones selected based on several optimization criteria. Consider for example the design of

a circuit using field programmable gate arrays (FPGAs). An FPGA is an integrated circuit designed

to be configured by the customer using a hardware description language (HDL), similar to that used

for an application-specific integrated circuit (ASIC). Because multiple choices for the placement of

components and for interconnecting them exist, the designer could run concurrently N versions of the

design choices and choose the one with the best performance, such as minimum power consumption.

Alternative optimization objectives could be to reduce cross-talk among the wires or to minimize the overall noise. Each alternative configuration requires hours ormaybe days of computing; hence, running

them concurrently reduces the design time considerably.

The list of companies that aimed to support parallel computing and ended up as casualties of this

effort is long and includes names such as Ardent, Convex, Encore, Floating Point Systems, Inmos,

Kendall Square Research, MasPar, nCube, Sequent, Tandem, and Thinking Machines. The difficulties

of developing newprogramming models and the effort to design programming environments for parallel

applications added to the challenges faced by all these companies.

From the very beginning itwas clear that parallel computing requires specialized hardware and system

software. It was also clear that the interconnection fabric was critical for the performance of parallel

computing systems.We nowtake a closer look at parallelism at different levels and themeans to exploit it.