Developing cloud programs (i.e., designing and implementing software systems that successfully exploit the capabilities of massively distributed computational resources) presents formidable challenges. The difficulties arise from the multiplicity of possible logical interactions and temporal interleavings among numerous software and hardware components. Program bugs can be difficult to reproduce, and due to some cloud programs' nondeterministic performance behavior, analyzing and reasoning a system's behavior can exceed mere human ability.

As researchers and practitioners have gradually gained a better understanding of this problem, they have developed models of programming and computation that mitigate cloud systems' inherent complexity. These models, which are embodied in software/hardware systems, stand between developer and underlying computational resources, providing the programmer stylized design patterns, a relatively simpler way of thinking about distributed programming, and a flexible interface to applications, data, and resources.

The current generation of cloud programming models builds on classical predecessors that support interprocess communication based on shared memory and message passing. Although those earlier versions provide basic facilities for interaction among distributed tasks, they lack the capability to parallelize and distribute tasks automatically and to recover from faults. Those versions' modern descendants, including Hadoop MapReduce, Pregel, and GraphLab, provide greater sophistication and specifically address the demands of distributed programming and computing in cloud environments. Among other advantages, these current models relieve developers from concerns with many of the difficult aspects of distributed programming and allow programmers to focus on sequential portions of their application's algorithms.

Echoing Babbage's 1837 design for the first programmable computer, we distinguish today's cloud-programming models by

referring to them as **distributed analytics engines**. This unit examines the concepts and challenges of cloud computing and provides current examples of analytics engines for developing cloud applications.

Taxonomy of Programs

A computer program embodies a computational algorithm and comprises variable declarations, variable assignments, expressions, and flow-control statements, which are all typically expressed in a high-level programming language, such as Java or C++. Before execution, programs are usually compiled and then converted to machine instructions/code that a central processing unit (CPU) can run either sequentially or concurrently. A **sequential** program runs in program order (i.e., the original, programmer-specified statement sequence).

A **concurrent** program, in contrast, is a set of sequential programs that, while executing, share one or more CPUs in time. This **timesharing**, defined in Unit 3, "Virtualizing Resources for the Cloud," allows multiple programs to take turns in using a single computational resource. For instance, given that a CPU can run only one program at a time, running multiple sequential programs on one processor requires the operating system (OS) to employ a scheduling strategy, such as round-robin, to allocate the resource to each program for a specified interval.

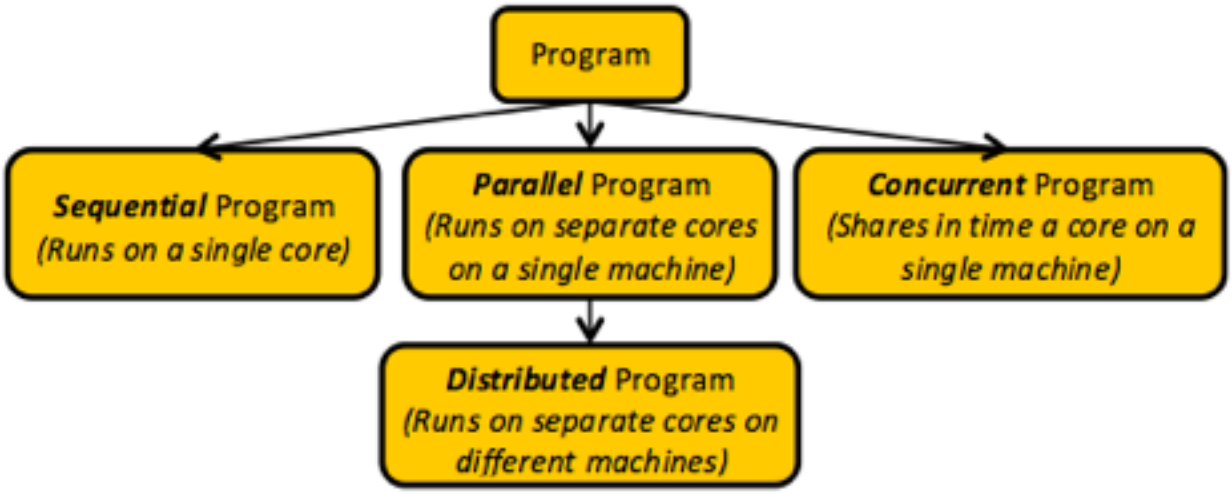

Figure 5.1: Our taxonomy of programs

The general term **application** refers, often ambiguously, to both sequential and concurrent programs, and the latter are sometimes called parallel programs. A parallel program is a set of sequential programs that overlap in time by running on separate CPUs. In multiprocessor systems, such as chip multicore machines, related sequential programs executing in different cores constitute a parallel program. By contrast, related sequential programs sharing a single CPU in time represent a **concurrent program**. For this unit, we further distinguish a parallel program comprising multiple sequential programs running on distinct, networked machines (versus different cores on one machine) as a **distributed program**. Figure 5.1 illustrates this program taxonomy.

Another term common in parallel/distributed programming is **multitasking**, which refers to overlapping one program's computation with that of another. This concept is central to all modern operating systems and describes a scheduling mechanism that enables an OS to juggle several programs at once. Multitasking has become so useful that almost all modern programming languages now support it through **multithreading** constructs.

The term "thread" was popularized by Pthreads (POSIX threads), [1] which is a specification of concurrency constructs that has been

widely adopted, especially in UNIX systems. [2] A **thread of execution** is the smallest sequence of instructions an OS can manage through its scheduler. Threads are intimately related to processes: a thread runs within the address space of a **process**, while a process runs within its own address space. Thus, a process can contain one or more threads, and the threads are not standalone work units. In principle, different processes do not share memory, while threads within a process all share their parent's address space.
Task is another term that refers to a small unit of work, and in this unit, we use it to mean "process." In addition, we collectively denote tasks (possibly only one) that belong to the same program/ application as a **job**. An application can encompass several jobs. For instance, a fluid dynamics application typically consists of three jobs, individually responsible for structural, fluid, and thermal analyses. In addition, each job can carry out its work through multiple tasks. Figure 5.2 demonstrates the concepts of processes, threads, tasks, jobs, and applications.

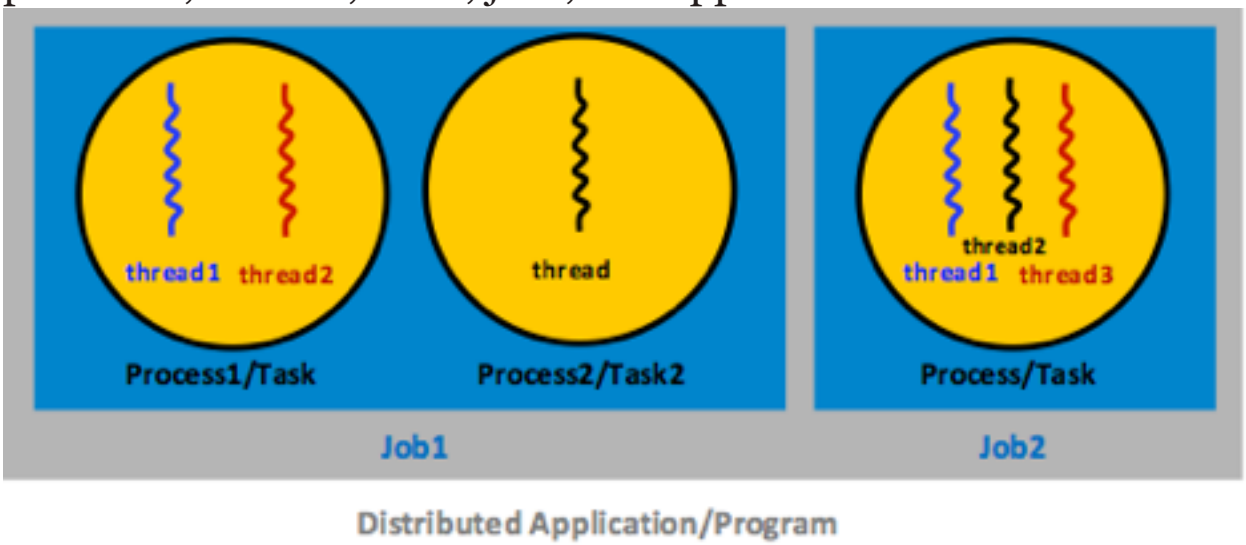

Figure 5.2: A demonstration of the concepts of processes, threads, tasks, jobs, and applications

Motivations for Distributed Programming

Why Parallelize and Distribute Programs?

Various analytic techniques at the algorithm and code levels can identify potential parallelism in sequential programs [1] and, in principle, every sequential program can be parallelized. A program can then be split into serial and parallel parts, as shown in Figure 5.3. Parallel parts can run concurrently on a single machine or be distributed across machines. Programmers typically transform sequential programs into parallel versions mainly to achieve higher computational speed, or throughput. In an ideal world, parallelizing a sequential program into an *n*-way distributed program would yield an *n*-fold decrease in execution time. Using distributed programs as opposed to sequential programs is crucial for multiple domains, especially science. For instance, simulating the folding of a single protein can take years if performed sequentially but only days if executed in parallel. The pace of scientific discovery in some domains depends on how fast certain scientific problems can be solved. Furthermore, some programs have real-time constraints such that if computation is not performed fast enough, the whole program may be rendered pointless. For example, predicting the direction of hurricanes and tornados using weather modeling must be done in a timely manner, or else the prediction will be wasted. In actuality, scientists and engineers have relied on distributed programs for decades to solve important and complex scientific problems, such as quantum mechanics, physical simulations, weather forecasting, oil and gas exploration, and molecular modeling, to mention a

few. This trend will probably continue, at least in the foreseeable future.

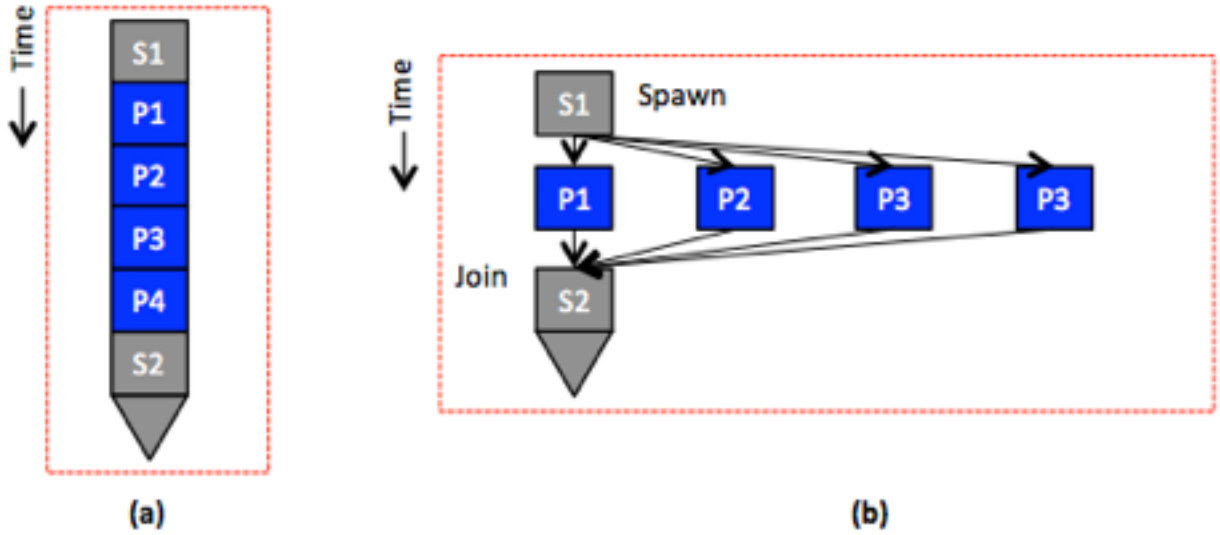

Figure 5.3: (a) A sequential program with serial (S1) and parallel (P1) parts. (b) A parallel/distributed program that corresponds to the sequential program in (a), whereby the parallel parts can be either distributed across machines or run concurrently on a single machine.

Distributed programs have also found broad applications beyond science, such as search engines, Web servers, and databases. For instance, much of Google's success can be traced back to the effectiveness of algorithms such as PageRank, [2] a distributed program that ranks Web pages and runs in Google's search engine on thousands of machines. Without parallelization, PageRank could not achieve its goals. Parallelization also contributes in leveraging available resources. For example, running a Hadoop MapReduce [3] program on a single Amazon EC2 instance is not as effective as running it on a large-scale cluster of EC2 instances. Of course, committing jobs earlier on the cloud leads to a reduction in cost, which is a key objective for cloud users. Distributed programs also help alleviate subsystem bottlenecks. For instance, I/O devices, such as disks and network interface cards, typically represent major bottlenecks in terms of bandwidth, performance, and/or throughput. By distributing work across machines, data can be served from multiple disks

simultaneously, offering an increased aggregate I/O bandwidth, improving performance, and maximizing throughput. In summary, distributed programs play a critical role in rapidly solving various computing problems and effectively mitigating resource bottlenecks. This action improves performance, increases throughput, and reduces cost, especially in the cloud.

Distributed Systems and the Cloud

Distributed Systems and Clouds

Distributed programs run on networked computers. Networks of computers are ubiquitous. The Internet, high-performance computing (HPC) clusters, mobile phones, and in-car networks, among others, present common examples. Many networks of computers are deemed distributed systems. We define a **distributed system** as one in which networked computers communicate using message passing and/or shared memory and coordinate their actions to solve a particular problem or offer a specific service. Because a cloud is defined as a set of Internet-based software, platform, and infrastructure services offered through a cluster (or clusters) of networked computers (i.e., data centers), a cloud is thus a distributed system. Another consequence of our definition is that distributed programs (versus sequential or parallel) will be the norm in clouds. In particular, we define distributed programs in the section **Taxonomy of Programs** as parallel programs that run on separate processors at different machines. Thus, the only way for tasks in distributed programs to interact over a distributed system is either by sending and receiving messages explicitly or by reading and writing from/to a shared distributed memory supported by the underlying distributed system (e.g., by using distributed shared memory

[DSM] hardware architecture). [1] We next identify the different models by which distributed programs for clouds (or *cloud programs*) can be built and recognize some of the challenges that cloud programs must address.

Programming the Cloud

The effectiveness of cloud programs hinges on the manner in which they are designed, implemented, and executed. The development process must address several considerations:

1. Which underlying programming model is most appropriate, message passing or shared memory?
2. Does the application better fit a synchronous or asynchronous computation model?
3. What is the best way to configure data for computational efficiency: by using data parallelism or graph parallelism?
4. Which architectural and management structure will most enhance program complexity, efficiency, and scalability: master-slave or peer to peer?

For cloud programs in particular, several issues—spanning design, implementation, tuning, and maintenance—require special attention:

1. Computational scalability is hard to achieve in large systems (e.g., clouds) for several reasons, including inability to parallelize algorithms completely, high probability of load imbalance, and inevitability of synchronization and communication overheads.
2. Communication that exploits data locality and minimizes network traffic can be tricky, particularly on (public) clouds, where network topologies are usually hidden.
3. Two common cloud realities—virtual environments and data center component diversity—introduce heterogeneity that complicates scheduling tasks and masks hardware and software differences among cloud nodes.

4 To avoid deadlocks and transitive closures and to guarantee mutually exclusive access, which are highly desirable capabilities in distributed settings, the underlying system must provide, and the designer must exploit, effective synchronization mechanisms.
5 As failure likelihood increases with cloud scale, system designs must employ fault-tolerance mechanisms, including task resiliency, distributed checkpointing, and message logging.
6 For effective and efficient execution, task and job schedulers must support control of task locality, parallelism, and elasticity as well as service-level objectives (SLOs).

Addressing all of these development considerations and cloud issues imposes a major burden on programmers. Designing, developing, verifying, and debugging all (or even some) of these capabilities present inherently difficult problems and can introduce significant correctness and performance challenges, in addition to consuming significant time and resources.
Modern distributed analytics engines promise to relieve developers of these responsibilities. These engines provide application programming interfaces (APIs) that enable users to present their programs as simple, sequential functions. The engines then automatically create, parallelize, synchronize, and schedule tasks and jobs. They also handle failures without requiring user involvement. At the end of this unit, we detail how distributed analytics engines effectively abstract and address the challenges of developing cloud programs. In the next section, however, we first present the two traditional distributed programming models: shared memory and message passing. Second, we discuss the computation models that cloud programs can employ. Specifically, we explain the synchronous and asynchronous computation models. Third, we present the two main parallelism categories of cloud programs, data parallelism and graph parallelism. Last, we describe the architectural models

that cloud programs can typically utilize: master-slave and peer-to-peer architectures.

Models for Building Cloud Programs

Programming models embody concepts and offer tools that support developers in building large computational systems. A distributed programming model, in particular, facilitates translating sequential algorithms into distributed programs that can execute over distributed systems. The model's design determines how easily a programmer can specify an algorithm as a distributed program. A model that abstracts architectural/hardware details, automatically parallelizes and distributes computation, and transparently supports fault tolerance is considered easy to use.

A model's efficiency, however, depends on the effectiveness of its underlying techniques. For a distributed program running on a system of distributed computers, one essential requirement is a communication mechanism that enables coordinating component tasks across multiple, networked resources. Two traditional models, message passing and shared memory, meet this need, although in a relatively basic form. Additional challenges in the distributed programs typical of cloud environments have led to more sophisticated programming models that, when implemented as distributed analytics engines, can automatically parallelize and distribute tasks and can tolerate faults.

The Shared-Memory Model

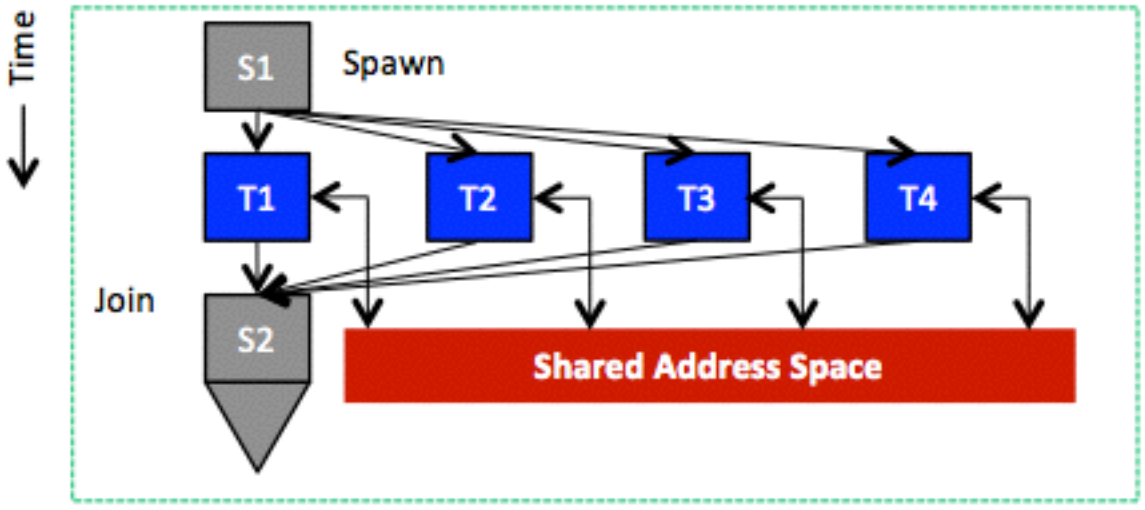

Figure 5.4: Tasks running in parallel and sharing an address space

The shared-memory model's key abstraction says that every task can access any location in an application's distributed memory space. Tasks thus communicate by reading from and writing to memory locations in the distributed memory space, as is the case with threads in a single process, where all threads share the process address space (Figure 5.4). With shared memory, tasks exchange data implicitly via sharing and not by explicitly sending and receiving messages. The shared-memory model, consequently, supports synchronization mechanisms that distributed programs must use to control the order in which various tasks can perform read/write operations. In particular, multiple tasks must not be able to write simultaneously to a shared-data location, which could corrupt data or make it inconsistent. This goal typically can be achieved using semaphores, locks, and/or barriers. A semaphore is a point-to-point synchronization mechanism that involves two parallel/distributed tasks. Semaphores use two operations: post and wait. The post operation acts like depositing a token, signaling that data has been produced. The wait operation blocks until signaled by the post operation that it can proceed to consume data. Locks protect critical sections, which are regions that only one task can access (typically write) at a time. Locks involve two operations,

lock and unlock, for acquiring and releasing a lock associated with a critical section. A lock can be held by only one task at a time, and other tasks cannot acquire it until released. Last, a barrier defines a point beyond which a task is not allowed to proceed until every other task reaches that point. The efficiency of semaphores, locks, and barriers is a critical and challenging goal in developing distributed/parallel programs for the shared-memory programming model (see the section **Synchronization**). Figure 5.5 shows an example that transforms a simple sequential program into a distributed one using the shared-memory programming model. The sequential program adds the elements of two arrays, `b` and `c`, storing the results in array `a`. Subsequently, any element greater than `0` in `a` is added to a grand sum. The corresponding distributed version assumes only two tasks and splits the work evenly between them. For every task, start and end variables are specified to correctly index the (shared) arrays, obtain data, and apply the given algorithm. Clearly, the grand sum is a critical section and so protected by a lock. In addition, no task can print the grand sum before every other task has finished its work, thus inserting a barrier before the printing statement. As shown in the program, communication between the two tasks is implicit (via reads and writes to shared arrays and variables) and synchronization is explicit (via locks and barriers). Last, as pointed out earlier, the underlying distributed system must provide data-sharing functionality. Specifically, the infrastructure must create the illusion that the memories of all computers in the system form a single, shared space that is addressable by all tasks. A common example of systems that offer such an underlying shared (virtual) address space on a cluster of computers (connected by a LAN) is called DSM. [2] [1] [3] A common programming language that can be used on DSMs and other distributed shared systems is OpenMP. [4]

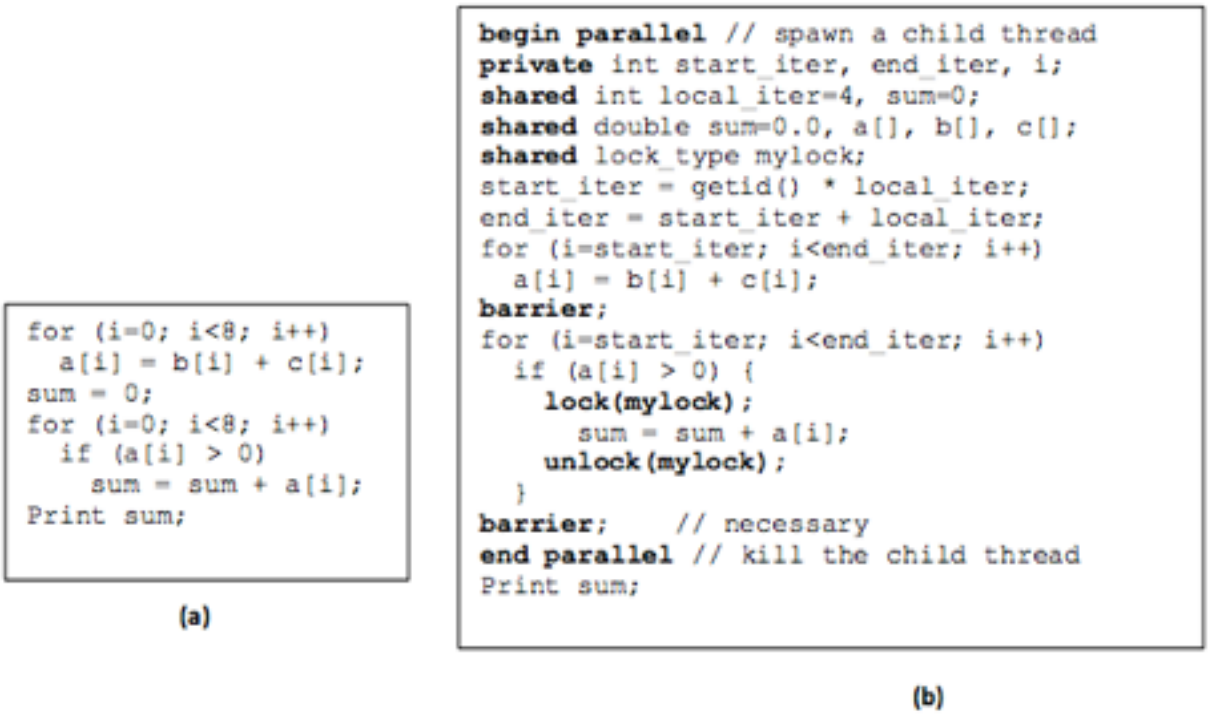

Figure 5.5: Sequential and shared-memory versions

The Message-Passing Programming Model

In the message-passing programming model, distributed tasks communicate by sending and receiving messages. Here, distributed tasks do not share an address space in which they can access each other's data (see Figure 5.6), and the key abstraction resembles processes that, unlike threads, each maintain a private memory space. To send and receive data via explicit messages, this model incurs communication overheads (e.g., variable network latency and potentially excessive data transfers). Balancing these overheads, the explicit message exchanges implicitly synchronize the operation sequence in communicating tasks. Figure 5.7 demonstrates an example that transforms the sequential program shown in Figure 5.5(a) into a distributed version that uses message passing. Initially, only a main task with **`id = 0`** can access the arrays **`b`** and **`c`**. Thus, assuming the existence of only two tasks, the main task first sends parts of the arrays to the other task (using an explicit send operation) in order

to split the work evenly between the two tasks. The second task receives the required data (using an explicit receive operation) and performs a local sum. When done, it sends back its local sum to the main task. Likewise, the main task performs a local sum on its data part and collects the local sum of the other task before aggregating and printing a grand sum. As shown, for every send operation, there is a corresponding receive operation, and no explicit synchronization is needed. Last, the message-passing programming model does not necessitate any support from the underlying distributed system. Specifically, the interacting tasks require no illusion of a single, shared address space. A popular example of a message-passing programming model is provided by the message passing interface (MPI). [5] MPI is an industry-standard, message-passing library (more precisely, a specification of what a library can do) for writing message-passing programs. A popular high-performance and widely portable implementation of MPI is MPICH. [6]

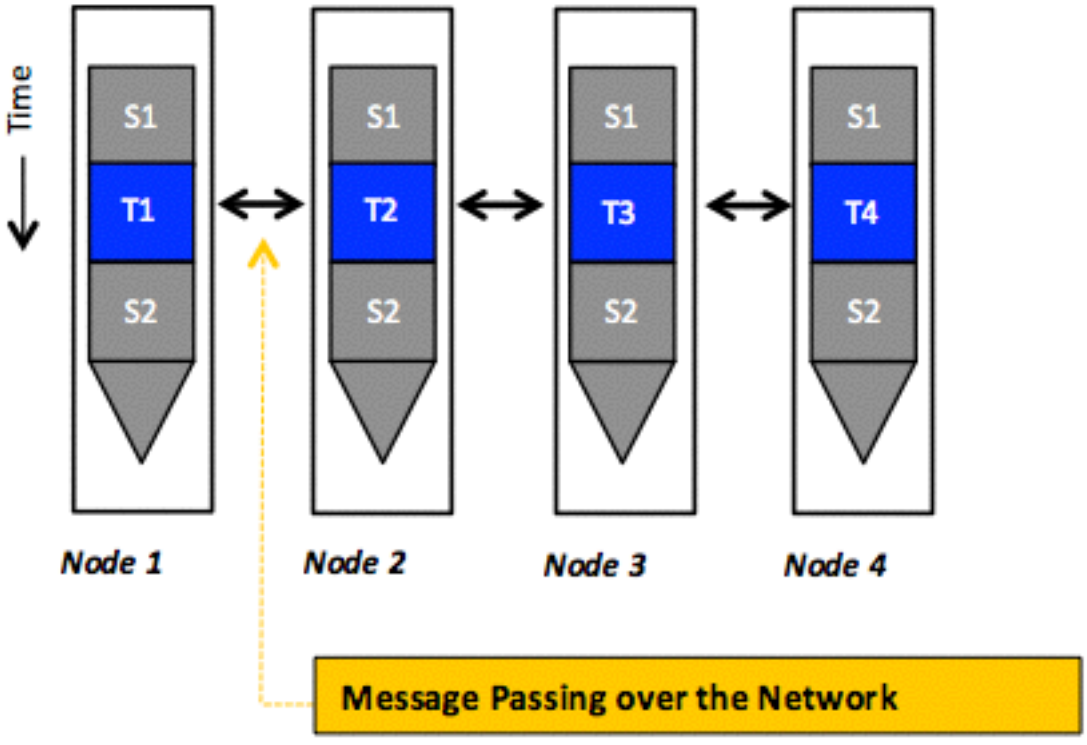

Figure 5.6: Tasks running in parallel using the message-passing programming model whereby the interactions happen only via sending and receiving messages over the network

Table 5.1 compares the shared-memory and the message-passing programming models in terms of five aspects: development effort, tuning effort, communication, synchronization, and hardware support.

Shared-memory programs are initially easier to develop because programmers need not worry about how data are laid out or communicated. Furthermore, the code structure of a shared-memory program is often quite similar to that of its sequential counterpart. Typically, programmers only insert additional directives to specify parallel/distributed tasks, scope of variables, and synchronization points. In contrast, message-passing programs require a shift in programmer thinking to consider, a priori, how to partition data across tasks, collect data, and communicate and aggregate results with explicit messaging.

How data are laid out and where it is stored begin to affect performance significantly as data and resources scale up. For instance, large-scale distributed systems such as the cloud imply nonuniform access latencies (e.g., accessing remote data takes much more time than accessing local data) thus encouraging programmers to keep data close to the tasks that use them. Although message-passing programmers must plan ahead to partition data across tasks, shared memory programmers will (most of the time) address that issue during postdevelopment, typically through data migration or replication. This adjustment can involve significant tuning effort compared to a message-passing design.

In large-scale systems, synchronization points can become performance bottlenecks: as the number of users attempting to access a critical section increases, associated delays and waits also increase. We return to synchronization and other challenges

involved in programming for the cloud in the section **Main Challenges in Building Cloud Programs**.

```
id = getpid();
local_iter = 4;
start_iter = id * local_iter;
end_iter = start_iter + local_iter;
if (id == 0)
  send_msg (P1, b[4..7], c[4..7]);
else
  recv_msg (P0, b[4..7], c[4..7]);
for (i=start_iter; i<end_iter; i++)
  a[i] = b[i] + c[i];
local_sum = 0;
for (i=start_iter; i<end_iter; i++)
  if (a[i] > 0)
    local_sum = local_sum + a[i];
if (id == 0) {
  recv_msg (P1, &local_sum1);
  sum = local_sum + local_sum1;
  Print sum;
}
else
  send_msg (P0, local_sum);
```

Figure 5.7. A distributed program that corresponds to the sequential program in Figure 5.5(a) and is coded using the message-passing programming model

Synchronous and Asynchronous Computation Models

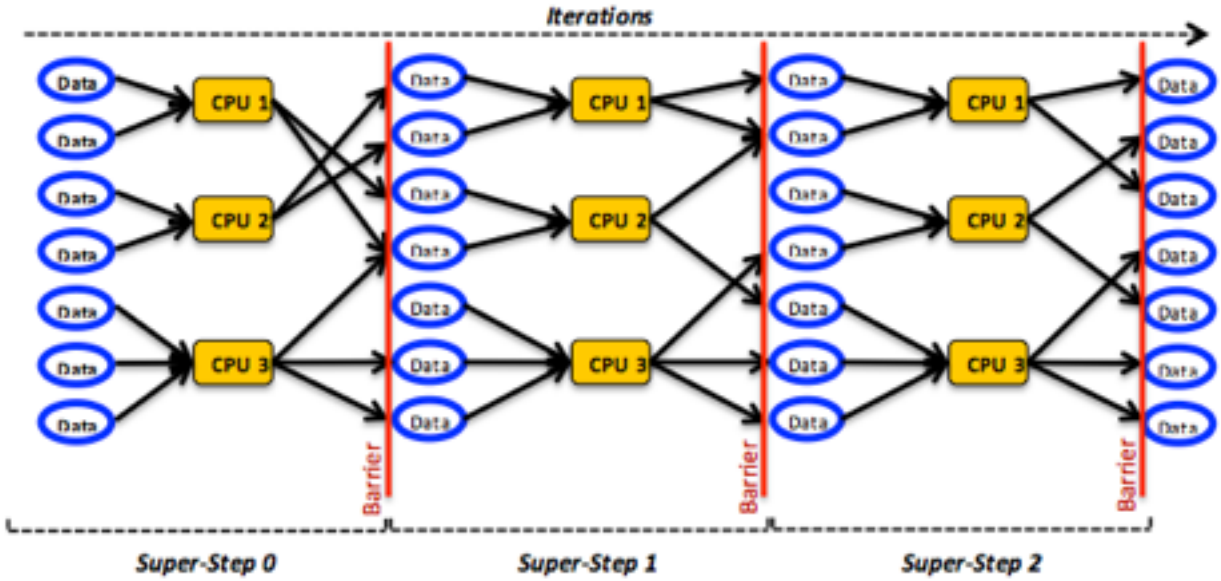

Figure 5.8: The bulk synchronous parallel (BSP) model

Independent of the programming model used, a developer can specify distributed computation as either **synchronous** or **asynchronous**. This distinction refers to the presence or absence of a (global) coordination mechanism that synchronizes task operations. A distributed program is synchronous if and only if the component tasks operate in lockstep. That is, for some constant `(c ≥ 1)`, if and only if any task has taken `(c + 1)` steps, then every other task must have taken at least *c* steps. [1] Subsequently, if any task has taken (c + 2) steps, then every other task must have taken at least (c + 1) steps. Clearly, this constraint requires a coordination mechanism through which task activities can be synchronized and their timing be accordingly enforced. Such mechanisms usually have an important effect on performance. Typically, in synchronous programs, distributed tasks must wait at predetermined points for the completion of certain computations or for the arrival of certain data. [3] A program that is not synchronous is an asynchronous program. Asynchronous programs impose no requirement to wait at predetermined points or for the arrival of certain data. Computational asynchrony obviously has less effect on

performance but implies that the correctness/validity of the program must be assessed.
MapReduce and Pregel programs, for example, involve synchronous computation, while those under GraphLab are asynchronous. Pregel employs the bulk synchronous parallel (BSP) model, [2] which is a synchronous model commonly employed for effectively implementing distributed programs. BSP combines three attributes: components, a router, and a synchronization method. A BSP component comprises a processor and data stored in local memory, but the model does not preclude other arrangements, such as holding data in remote memories. BSP is neutral about the number of processors, be it two or a million, and BSP programs can be written for **v** virtual distributed processors to run on **p** physical distributed processors, where **(v > p)**.
BSP builds on the message-passing programming model, sending and receiving messages through a router that, in principle, can only pass messages point-to-point between pairs of components (the model provides no broadcasting facilities, although developers can implement them using multiple point-to-point communications). To achieve synchrony, BSP splits every computation into a sequence of steps called **super-steps**. In every super-step, **S**, each component is assigned a task encompassing (local) computation. Components in **S** can send messages to **(c + 1)** components in super-step **(S + 1)** and are implicitly allowed to receive messages from components in super-step **(S – 1)**. In every super-step, tasks operate simultaneously and do not communicate with each other. Across super-steps, tasks move in a lockstep mode: no task in **(S + 1)** can start before every task in S commits. To satisfy this condition, BSP applies a global, barrier-style synchronization mechanism, as shown in Figure 5.8. Because BSP does not provide simultaneous

accesses to a single memory location, it does not require any synchronization mechanism beyond barriers.
Another primary concern in a distributed setting lies in allocating data so that computation will not be slowed by nonuniform memory access latencies or uneven loads among individual tasks. BSP promotes uniform access latencies by using local data: data are communicated across super-steps before triggering actual task computations, and the model thus divorces computation and communication. Such separation implies that no particular network topology is favored beyond the requirement to deliver high throughput. Butterfly, hypercube, and optical crossbar topologies are all acceptable.
Across tasks within a super-step, data volumes can still vary, and task loading depends mainly on the distributed program and the responsibilities it imposes on its constituent tasks. Accordingly, the time required to finish a super-step becomes bound by its slowest task (a super-step cannot commit before its slowest task completes). This limit presents a major challenge for the BSP model because it can create load imbalance, which usually degrades performance. Load imbalance can also be caused by heterogeneous clusters, especially on the cloud (see the section **Heterogeneity**). Note that although BSP suggests several design choices, it does not make their use obligatory. Indeed, BSP leaves many design choices open (e.g., barrier-based synchronization can be implemented at a finer granularity or completely switched off if unneeded for the given application).

Data and Graph Parallelism Models

A second consideration in developing distributed programs involves specifying the type of parallelism, data or graph

parallelism. The data parallelism design emphasizes the distributed nature of data and spreads it across multiple machines. Computation, meanwhile, can remain the same among all nodes and be applied on different data. Alternately, tasks on different machines can perform different computational tasks. When the tasks are identical, we classify the distributed program as single program, multiple data (SPMD); otherwise, we categorize it as multiple program, multiple data (MPMD).
The basic idea of **data parallelism** is simple: by distributing a large file across multiple machines, it becomes possible to access and process different parts of the file in parallel. As discussed in Unit 4, one popular technique for distributing data is file striping, in which a single file is partitioned and distributed across multiple servers. Another form of data parallelism is to distribute whole files (without partitioning) across machines, especially if files are small and their contained data exhibit very irregular structures. We note that data can be distributed among distributed tasks either explicitly, by using message passing, or implicitly, by using shared memory, assuming the underlying distributed system supports shared memory.

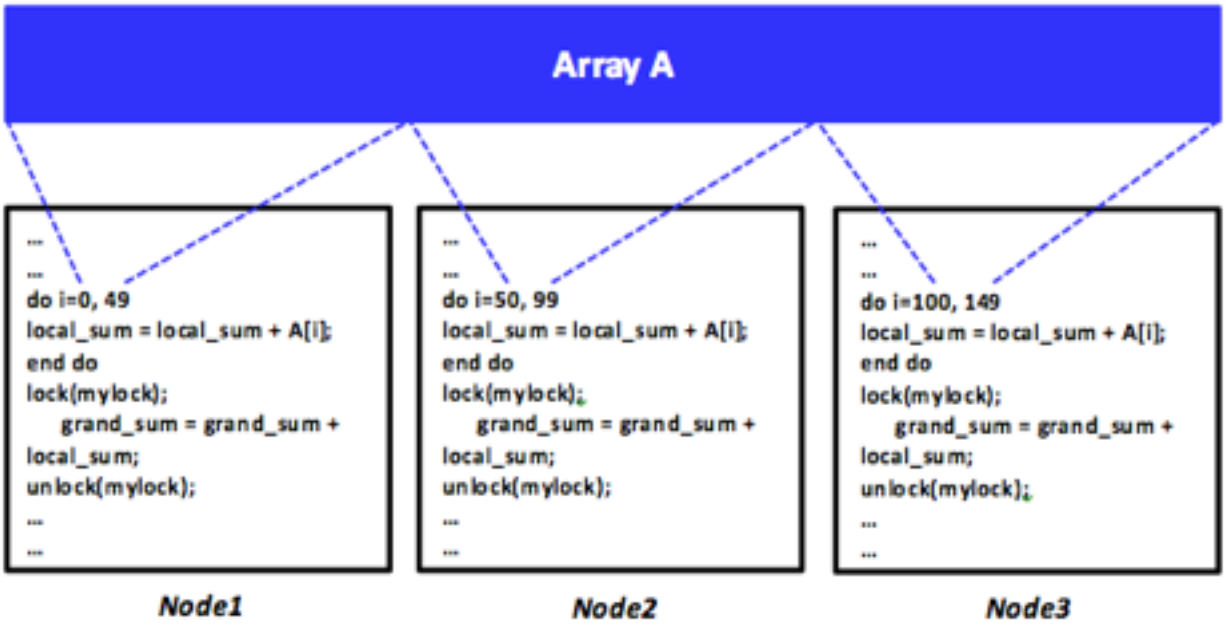

Figure 5.9: An SPMD distributed program using the shared-memory programming model

Data parallelism is achieved when each node runs one or many tasks on different pieces of distributed data. As a specific example, assume array A is shared among three machines in a distributed shared-memory system. Consider also a distributed program that simply adds all elements of array A. It is possible to command machines 1, 2, and 3 to run the addition task, each on one-third of array A, or 50 elements, as shown in Figure 5.9. The data can be allocated across tasks using the shared-memory programming model, which requires a synchronization mechanism. Clearly, such a program is SPMD. In contrast, array A can also be distributed evenly (using message passing) by a (master) task among three machines, including the master's machine, as shown in Figure 5.10. Each machine will run the addition task independently; nonetheless, summation results will have to be eventually aggregated at the master task in order to generate a grand total. In such a scenario, every task is similar in a sense that it is performing the same addition operation, yet on a different part of array A. The master task, however, is also distributing data to all tasks and aggregating summation results, thus making it slightly different from the other two tasks. Clearly, this makes the program MPMD. As will be discussed in the MapReduce section, MapReduce uses data parallelism with MPMD programs.

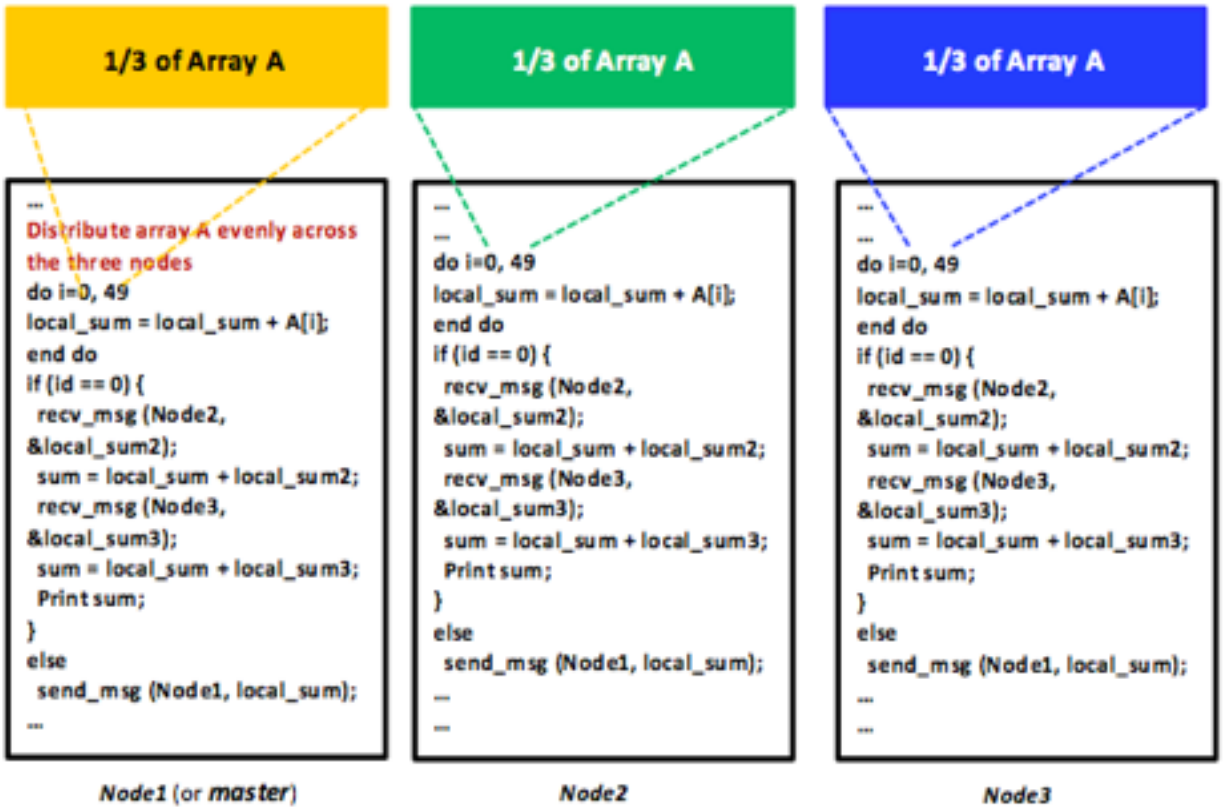

Figure 5.10: An MPMD distributed program using the message-passing programming model

Graph parallelism, on the other hand, focuses on distributing computation as opposed to data. Most distributed programs actually fall somewhere on a continuum between the two forms. Graph parallelism is widely used in many domains such as machine learning, data mining, physics, and electronic circuit design, among others. Many problems in these domains can be modeled as graphs in which vertices represent computations and edges encode data dependencies or communications. Recall that a graph

, called edges. Weights can be associated with vertices and edges to indicate the amount of work at each vertex and the communication data on each edge.

Consider a classical problem from circuit design: the common goal of keeping certain pins of several components electrically equal by wiring them together. If we assume **n** pins, then an

arrangement of **(n – 1)** wires, each connecting two pins, can be employed. Of all such arrangements, the one requiring the minimum number of wires is normally the most desirable. Obviously, this wiring problem can be modeled as a graph problem. In particular, each pin can be represented as a vertex, and each interconnection between a pair of pins

is the minimum. As *S* is acyclic and fully connected, it must result in a tree known as the *minimum spanning tree*. Consequently, solving the wiring problem morphs into solving the minimum spanning tree problem, a classical problem that is solvable with Kruskal's and Prim's algorithms, to mention a few. [1]

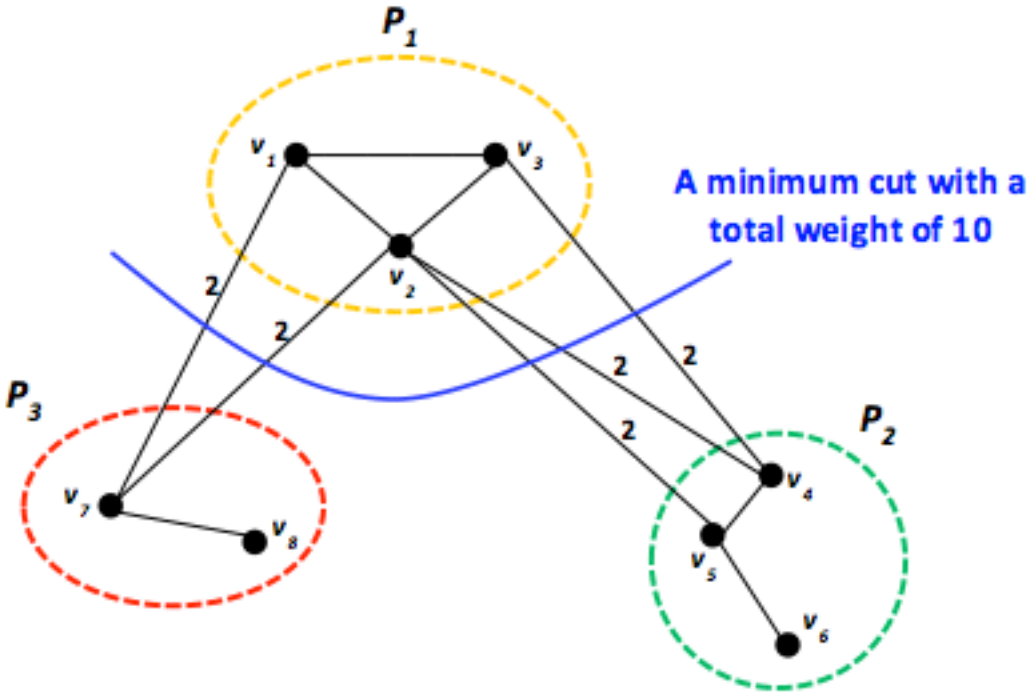

Figure 5.11: A graph partitioned using the edge-cut metric

Once modeled as a graph, a program can be distributed over machines in a distributed system using a graph-partitioning technique, which involves dividing the work (vertices) over distributed nodes for efficient distributed computation. As with data parallelism, the basic idea is simple: by distributing a large graph across multiple machines, it becomes possible to process

different parts of the graph in parallel, resulting in a graph-parallel design. The standard objective of graph partitioning is to distribute work uniformly over *p* processors by partitioning the vertices into *p* equally weighted partitions while minimizing internode communication reflected by edges. Such an objective is typically referred to as the standard edge-cut metric. [2] While the graph partitioning problem is NP-hard, [3] heuristics can achieve near optimal solutions. [2] [4] [5] As a specific example, Figure 5.11 demonstrates three partitions, P_1, P_2, and P_3, at which vertices $\{v_1, \ldots, v_8\}$ are divided using the edge-cut metric. Each edge has a weight of two corresponding to one unit of data communicated in each direction. Consequently, the total weight of the shown edge cut is 10. Other cuts will result in more communication traffic. Clearly, for communication-intensive applications, graph partitioning is critical and can play a dramatic role in dictating the overall application performance. Some of the challenges pertaining to graph partitioning are discussed in the section **Communication**. Both Pregel and GraphLab employ graph partitioning, and we further discuss each in later sections.

The third design consideration is one of organizational structure. An application developer typically organizes a distributed program in a **master-slave** (asymmetric) or **peer-to-peer** (symmetric) architecture, as shown in Figure 5.12. Other organizations, such as hybrids, may be appropriate for special circumstances. [1] For the purpose of this unit, we are only concerned with the master-slave and peer-to-peer organizations.

Asymetric Master-Slave Organization

In a master-slave organization, a central process, called the *master*, handles all the logic and controls, and all other processes are denoted as slave processes. Thus, interaction between processes is asymmetrical: bidirectional connections enable the

master to communicate with each slave, and no interconnection is permitted between any two slaves (see Figure 5.12(a)). This situation requires the master to keep track of each slave's network location in what is called a *metadata structure*, and, further, that each slave can always identify and locate the master.

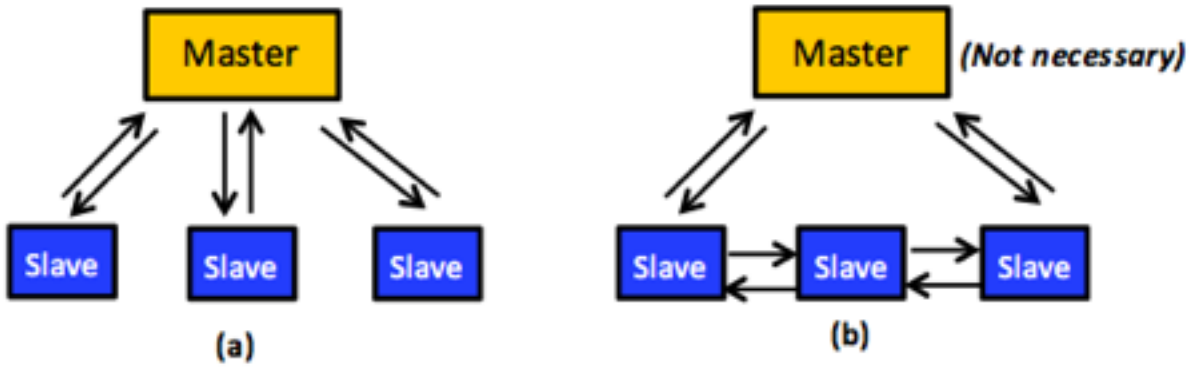

Figure 5.12. (a) A master-slave organization. (b) A peer-to-peer organization. The master in such an organization is optional (usually employed for monitoring the system and/or injecting administrative commands).

In asymmetric organizations, the master can distribute work among the slaves using one of two following protocols:

1 The **push-based** strategy assigns work to slaves unilaterally, without their asking. Clearly, this situation allows the master to apply fairness constraints over the slaves via distributing the work equally among them. Alternately, this arrangement could also overwhelm/congest slaves currently experiencing slowness/failures and who are thus unable to keep up with work. Consequently, load imbalance might occur, which usually leads to performance degradation. Nevertheless, the master can implement smart strategies. In particular, the master can assign work if and only if the slave is observed to be ready. For this tactic to work, the master must continuously monitor all slaves and apply some logic (usually complex) to accurately identify available slaves. To maintain fairness and enhance performance, the master must also decide upon the amount of work to assign. In clouds, the probability of faulty and

slow processes increases due to heterogeneity, performance unpredictability, and scalability (see the section **Main Challenges in Building Cloud Programs**). These limitations can make the push-based protocol inefficient on the cloud.

2. The **pull-based** strategy, on the other hand, requires slaves to request assignment of work. This protocol significantly reduces complexity and potentially avoids load imbalance because the decision of whether a particular slave is ready or not is delegated to the slave itself. Nonetheless, the master still needs to monitor the slaves, usually to track the progress of tasks at slaves and/or apply fault-tolerance mechanisms (e.g., to effectively address faulty and slow tasks, commonly present in large-scale clouds). Hadoop MapReduce and Pregel utilize the pull-based protocol.

To this end, we note that the master-slave organization suffers from a single point of failure (SPOF). Specifically, if the master fails, the entire distributed program comes to a grinding halt. Furthermore, having a central process (the master) for controlling and managing everything might not scale well beyond a few hundred slaves, unless efficient strategies are applied to reduce the contention on the master (e.g., caching metadata at the slaves so as to avoid accessing the master on each request). In contrart, using a master-slave organization simplifies decision making (e.g., allowing a write transaction on a certain shared data). In particular, the master is always the sole entity that controls everything and can make any decision singlehandedly without bothering anything else. This simplicity averts the employment of voting mechanisms, [3] [1] [2] which are typically needed when only a group of entities (not a single entity) has to make decisions. The basic idea of voting mechanisms is to require a task to request and acquire the permission for a certain action from at least half of the tasks plus one (a majority). Voting mechanisms usually complicate implementations of distributed programs.

Symmetric Peer-to-Peer Organization

In symmetric organizations, all tasks are equal, with logic, control, and work distributed evenly among them. Specifically, each task can communicate directly with those around it, without having to contact a master process (see Figure 5.12(b)). A master may be adopted, however, but only for purposes such as monitoring the system and/or injecting administrative commands. In other words, peer tasks do not require a master to function correctly. Moreover, although tasks communicate with one another, their work can be totally independent and may even be unrelated. Peer-to-peer organizations eliminate the potential for an SPOF and bandwidth bottlenecks, thus they typically exhibit good scalability and robust fault tolerance. Making decisions in peer-to-peer organizations, however, must be carried out collectively, usually through voting mechanisms. This arrangement typically implies increased implementation complexity as well as higher communication overhead and latency, especially in large-scale systems such as the cloud. GraphLab, which we discuss in later sections, employs a peer-to-peer organization.

Challenges in Building Cloud Programs

Designing and implementing a distributed program involves, as we have seen, choosing a programming model and addressing issues of synchrony, parallelism, and architecture. Beyond these matters, when developing cloud programs, a designer must also pay careful attention to several other challenges critical to cloud environments. We next discuss challenges associated with

scalability, communication, heterogeneity, synchronization, fault tolerance, and scheduling.

Scalability

A distributed program is considered to be scalable if it remains effective when the quantities of users, data, and resources increase significantly. To get a sense of the problem scope, consider the many popular applications and platforms currently offered to millions of users as Internet-based services. Along the data dimension, in this time of big data and the "era of tera" (Intel's phrase) [1] distributed programs typically cope with Web-scale data on the order of hundreds or thousands of gigabytes, terabytes, or petabytes. Google, for example, processes 20PBs of data per day. [2] Globally, in 2010, data sources generated approximately 1.2ZB (1.2 million petabytes), and the 2020 predictions expect an increase of nearly 44 times that amount. [3] Internet services, such as e-commerce and social networks, handle data volumes generated by millions of users daily. [2] Regarding resources, cloud data centers already host tens to hundreds of thousands of machines (e.g., according to estimates, Amazon EC2 hosts almost half a million machines), [5] and projections anticipate yet another multifold scaling of machine counts.

The reality of execution on n nodes never meets the ideal of n-fold performance escalation. Several reasons intervene:

1. As shown in Figure 5.13, some program parts can never be parallelized (e.g., initialization).
2. Load imbalance among tasks is highly likely, especially in distributed systems, such as clouds, in which heterogeneity (see the section Heterogeneity) is a major factor. As depicted in Figure 5.13(b), load imbalance usually delays programs so that a program becomes bound to its slowest task.

Specifically, even if all tasks in a program finish, the program cannot commit before the last task finishes.

3 Other serious overheads, such as communication and synchronization overheads, can significantly impede scalability.

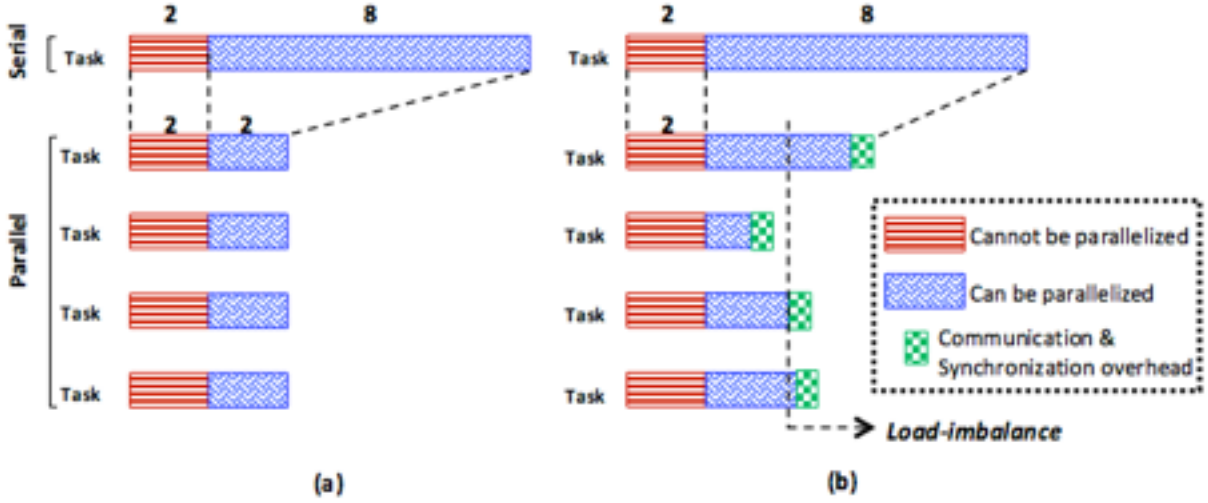

Figure 5.13: Parallel speedup: (a) ideal case and (b) real case

These issues are important when comparing the performance of distributed and sequential programs. A widely used expression that describes speedups and, additionally, accounts for various overheads is Amdahl's law. To illustrate the calculation, we assume that a sequential version of a program **T** takes **Ts** time units, while a parallel/distributed version takes **Tp** time units using a cluster of **n** nodes. In addition, we suppose that fraction **s** of the program is not parallelizable, leaving the **(1 - s)** portion parallelizable. According to Amdahl's law, the speedup of the parallel/distributed execution of **P** versus the sequential one can be defined as follows:

Although the formula is apparently simple, it exhibits a crucial implication: if we assume a cluster with an unlimited number of machines and a constant s, we can express the maximum achievable speedup by simply computing the speedup of **P** with an infinite number of processors as follows:

To understand the essence of this analysis, let us assume a serial fraction **s** of only 2%. Applying the formula with, say, an unlimited number of machines will result in a maximum speedup of only 50. Reducing **s** to 0.5% would result in a maximum speedup of 200. Consequently, we observe that attaining scalability in distributed systems is extremely challenging because it requires **s** to be almost 0, and this analysis ignores the effects of load-imbalance, synchronization, and communication overheads. In practice, synchronization overheads (e.g., performing barrier synchronization and acquiring locks) increase with an increasing number of machines, often super linearly. [4] Communication overheads also grow dramatically in large-scale distributed systems because all machines cannot share short physical connections. Load imbalance becomes a big factor in heterogeneous environments, as we discuss shortly. Although this is truly challenging, we point out that with Web-scale input data, the overheads of synchronization and communication can be greatly reduced if they contribute much less towards overall execution time than does computation. Fortunately, with many big-data applications, this latter situation is the case.

Communication

Even with distributed shared-memory systems, such as DSM, messages are passed internally between machines, albeit in a manner totally transparent to users. Hence, coordination all boils down to passing messages. We can argue, then, that the only way distributed systems can communicate is by passing messages. In fact, Coulouris and associates [1] adopt just this definition for

distributed systems. Distributed systems, such as the cloud, rely heavily on the underlying network to deliver messages rapidly enough to destination entities for three main reasons: performance, cost, and quality of service (QoS). Specifically, fast message delivery minimizes execution times, reduces costs (because cloud applications can commit earlier), and raises QoS, especially for audio and video applications. This condition makes the issue of communication a principal theme in developing cloud programs. Indeed, some might argue that communication lies at the heart of the cloud and constitutes one of its major bottlenecks. Distributed programs can apply two techniques to address cloud communication bottlenecks.

Colocation

Distributing/partitioning work across machines attempts to place highly communicating entities together. This strategy can mitigate pressure on the cloud network and subsequently improve performance. Realizing this goal, however, is not as easy as it might seem. For instance, the standard edge-cut strategy seeks to partition graph vertices into p equally weighted partitions over p processors so that the total weight of the edges crossing between partitions is minimized (see the section **Data Parallel and Graph Parallel Computations**).

Carefully inspecting this strategy, we recognize a serious shortcoming that directly impacts communication. As Figure 5.10 in section **Data Parallel and Graph Parallel Computations** shows, the minimum cut resulted from the edge-cut metric overlooks the fact that some edges may represent the same information flow. In particular, **v2** at **P1** in the figure sends the same message twice to **P2** (specifically to **v4** and **v5** at **P2**), while it suffices to communicate the message only once because **v4** and **v5** will exist on the same machine. Likewise, **v4** and **v7** can communicate messages to P_1 only once, but they do it twice.

The standard edge-cut metric thus overcounts communication volume and consequently leads to superfluous network traffic. As

a result, interconnection bandwidth can be potentially stressed and performance degraded. Even if the total communication volume (or the number of messages) is minimized more effectively, load imbalance can generate a bottleneck. In particular, it may happen that, although communication volume is minimized, some machines receive larger partitions (with more vertices) than others. An ideal, yet challenging, approach is to minimize communication overheads while circumventing computation skew among machines. This latter strategy strives for effective partitioning of work across machines so that highly communicating entities are colocated.

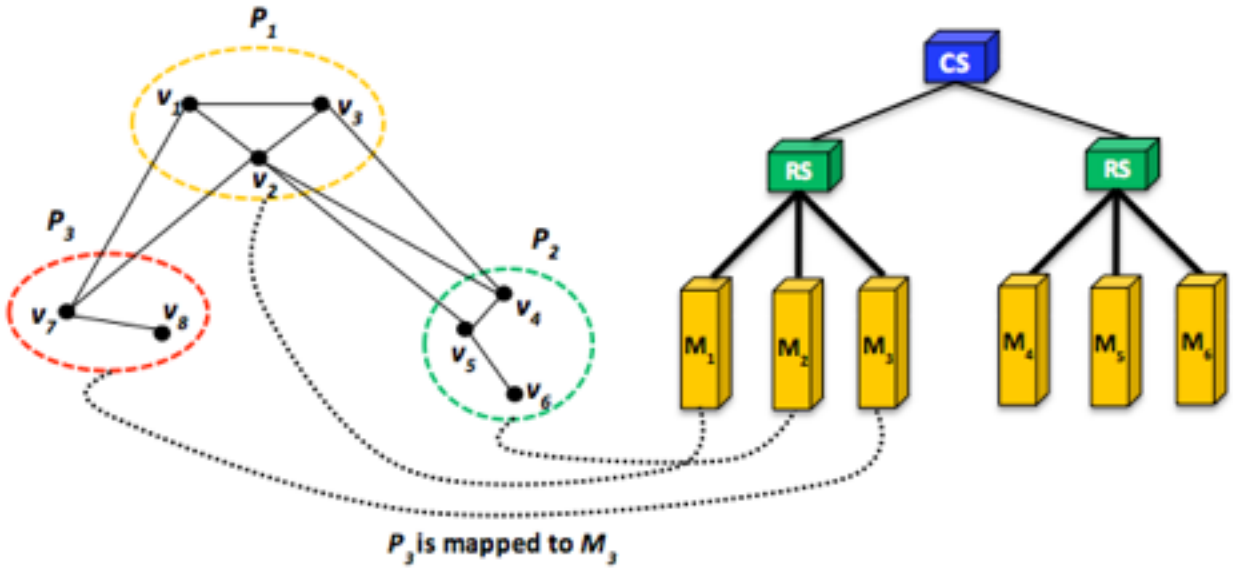

Figure 5.14: Effective mapping of graph partitions to cluster machines. A mapping of **P1** to the other rack while **P2** and **P3** remain on the same rack causes more network traffic and potentially degraded performance.

Effective Partition Mapping

To be most effective, the strategy for mapping partitions—whether graph or data partitions—to machines should be totally aware of the underlying network topology. This goal usually requires determining the number of switches a message will hit before reaching its destination. As a specific example, Figure 5.14 demonstrates the same graph shown in Figure 5.10 and a simplified cluster with a tree-style network and six machines. The cluster network consists of two rack switches (RSs), each

connecting three machines, and a core switch (CS) connecting the two RSs. Note that the bandwidth between any two machines depends on their relative locations in the network topology. For instance, machines on the same rack share a higher bandwidth connection as opposed to machines that are off-rack. Thus, it pays to minimize network traffic across racks. If **P1**, **P2**, and **P3** are mapped to **M1**, **M2**, and **M3**, respectively, less network latency will be incurred when **P1**, **P2**, and **P3** communicate versus if they are mapped across the two racks. More precisely, for **P1** to communicate with **P2** on the same rack, only one hop is incurred to route a message from **P1** to **P2**. In contrast, for **P1** to communicate with **P2** on different racks, two hops are incurred per each message. Clearly, fewer hops reduce network latency and improve overall performance. Unfortunately, this objective is not as easy to achieve on clouds as it might seem for one main reason: especially on public systems, such as Amazon EC2, network topologies remain hidden. Nevertheless, the network topology can still be learned (though not effectively) using a benchmark such as Netperf [2] to measure point-to-point TCP stream bandwidths between all pairs of cluster nodes. [3] This approach enables estimating the relative locality of nodes and arriving at a reasonable inference regarding the cluster's rack topology.

Heterogeneity

Cloud data centers comprise various collections of components, including computers, networks, operating systems (OSs), code libraries, and programming languages. In principle, if there is variety and difference in data center components, the cloud is referred to as a heterogeneous cloud. Otherwise, the cloud is

denoted as a homogenous cloud. In practice, homogeneity does not always hold, mainly due to two reasons:

1. Cloud providers typically maintain multiple generations of IT resources, purchased over different time frames.
2. Cloud providers are increasingly applying virtualization technology on their clouds to consolidate servers, enhance system utilization, and simplify management. Public clouds are primarily virtualized data centers. Even on private clouds, virtualized environments are expected to become the norm. [1]

Heterogeneity is a direct result of virtualized environments, and colocating virtual machines (VMs) on similar physical machines can cause heterogeneity. Consider, for example, two identical physical machines, A and B. Even assuming identical VMs running the same programs, placing one VM on machine A and 10 VMs on machine B will stress the second machine more than the first. Having dissimilar VMs and diverse, demanding programs is even more probable on the cloud, and the situation is worse there. An especially compelling setting is Amazon EC2, which offers 17 VM types [5] (see Unit 4) for millions of users with different programs. Clearly, this situation creates even more heterogeneity. In short, heterogeneity is already and will continue to be the norm on the cloud.

Heterogeneity poses multiple challenges for running distributed programs on the cloud. Distributed programs must be designed to mask heterogeneity of the underlying hardware, networks, OSs, and programming languages. The illusion of homogeneity allows distributed tasks to communicate; otherwise message passing and the whole concept of distributed programs fail. Consider the data representation problem: messages exchanged between tasks usually contain primitive data types, such as integers. Unfortunately, not all computers store integers in the same order. In particular, some computers use the so-called big-endian order, in which the most significant byte comes first, while others use the

so-called little-endian order, in which the most significant byte comes last. The floating-point numbers can also differ across computer architectures. Another issue is the set of codes used to represent characters. Some systems use ASCII characters, while others use the Unicode standard. In a word, distributed programs have to work out such heterogeneity to exist. The part that can be incorporated in distributed programs to work out heterogeneity is commonly referred to as middleware. Fortunately, most middleware are implemented over the Internet protocols, which themselves mask the differences in the underlying networks. The Simple Object Access Protocol (SOAP) [2] is an example of middleware. SOAP defines a scheme for using Extensible Markup Language (XML), a textual self-describing format, to represent contents of messages and allow distributed tasks at diverse machines to interact. Another example is Representational State Transfer, or REST.

In general, code suitable for one machine might not be suitable for another machine on the cloud, especially when instruction-set architectures (ISAs) vary across machines. Ironically, the virtualization technology, which induces heterogeneity, can effectively serve in solving the problem. Some VMs can be initiated for a user cluster and mapped to physical machines with different underlying ISAs. Afterwards, the virtualization hypervisor will take care of emulating any difference between the ISAs of the provisioned VMs and the underlying physical machines (if any). From a user's perspective, all emulations occur transparently. Last, users can always install their own OSs and libraries on system VMs, like Amazon EC2 instances, thus ensuring homogeneity at the OS and library levels.

Another serious heterogeneity problem that requires attention from distributed programmers is performance variation [3] [4] on the cloud. Performance variation describes the situation in which running the same distributed program twice on the same cluster can result in different execution times. For example, execution

times can vary by a factor of five for the same application on the same private cluster. [4] Performance variation is due mostly to cloud heterogeneity, imposed by virtualized environments, and spikes and lulls in resource demand over time. As a consequence, cloud VMs rarely carry work at the same speed, thereby preventing tasks from making progress at (approximately) constant rates. Clearly, this situation can create tricky load imbalance and degrade overall performance because load imbalance makes a program's performance contingent on its slowest task. Distributed programs can attempt to provide relief by detecting slow tasks and scheduling corresponding speculative tasks on fast VMs so that the latter finish earlier. Specifically, two tasks with the same responsibility can compete by running at two different VMs, with the one that finishes earlier getting committed and the other killed. Hadoop MapReduce follows a similar strategy for solving the same problem, called *speculative execution*. Unfortunately, distinguishing between slow and fast tasks/VMs is challenging on the cloud. It could happen that a certain VM running a task is temporarily passing through a demand spike, or it could be the case that the VM is simply faulty. In theory, not every node that is detected to be slow is faulty, and differentiating between faulty and slow nodes is hard. [6] Because of this problem, Hadoop MapReduce does not perform well in heterogeneous environments. [7] [8] [9] Reasons for that and details on Hadoop's speculative execution are presented in Hadoop MapReduce section.

Synchronization

To achieve maximum performance, distributed tasks need the ability to operate simultaneously on shared data without risking data corruption or inconsistency. Synchronization mechanisms

address this requirement by allowing programmers to control the sequence of operations (reads and writes) that tasks perform. For instance, GraphLab allows multiple tasks to operate on different vertices of the same graph simultaneously. This capability could lead to race conditions in which two tasks try to modify data on a shared edge at the same time, resulting in a corrupted value. The solution lies in a synchronizing means to assure that distributed tasks can obtain mutually exclusive data access, the **mutual exclusion** property.

As discussed in the section Shared-Memory Programming Model, three synchronization methods are widely used: semaphores, locks, and barriers. Applying these methods efficiently is a critical goal in developing distributed programs. For instance, although a barrier is easy to implement, a distributed program's overall execution time then becomes dependent on the slowest task. In distributed systems such as the cloud, in which heterogeneity is the norm, this situation can seriously degrade performance. The challenge is to employ synchronization methods without paying performance penalties.

In addition to mutual exclusion, synchronization mechanisms must also guarantee other properties for distributed programs. To begin, if one task attempts to access a critical section, it should eventually succeed. If two tasks try simultaneously to access a critical section, only one should succeed. However, things may not always go as expected. For instance, if task `A` succeeds in acquiring `lock1` and, at about the same time, task `B` succeeds in acquiring `lock2`; then if task A attempts to acquire `lock2`, and task `B` attempts to acquire `lock1`, we have what is known as a **deadlock**. Avoiding such stalemates presents a significant challenge in developing distributed programs, especially when the number of tasks scales up, and any mutual exclusion mechanism must ensure the deadlock-free property.

To build upon the example of tasks `A` and `B`, let us assume a larger set of tasks **{A, B, C..., Z}**. In ensuring mutual exclusion, task **A** might wait on task **B**, if **B** is holding a lock required by **A**. In return, task **B** might wait on task **C**, if **C** is holding a lock required by **B**. The "wait on" sequence can carry on all the way up to task **Z**. Specifically, task **C** might wait on task **D**, and task **D** might wait on task **E**, all the way until task **Y**, which might also wait on task **Z**. Such a "wait-on" chain is usually referred to as transitive closure. When a transitive closure occurs, a circular wait is said to arise, which situation normally leads to stark deadlocks that can bring an entire distributed program/system to a grinding halt. Lastly, we note that the wait-on relation lies at the heart of every mutual exclusion mechanism. In particular, no mutual exclusion protocol, no matter how clever, can preclude it. [2] In normal scenarios, a task expects to "wait on" for a limited (reasonable) amount of time. But what if a task that is holding a lock/token crashes? This scenario brings us to another major challenge for distributed programs, namely, fault tolerance.

Did You Know?

Mutual exclusion in distributed systems can be categorized into two main classes, **token based** and **permission based**. In the token-based approach, mutual exclusion is accomplished by passing a single message denoted as token between tasks of a distributed program. Obtaining the token can be deemed as acquiring the lock. As such, a task that holds the token can access the shared data, while every other task will wait until its turn arrives. Tasks using the token-based approach are usually organized logically as a ring. When a task is done, it passes the token to the next task on the ring. The next task can either choose to access the shared data or simply pass the token to the subsequent task on the ring. The token-based approach avoids starvation because it can fairly ensure that each task will have the chance to access the shared data. On the other hand, it suffers

from a reliability issue. In particular, if the token is lost on the network (e.g., due to a network failure), or the task that is currently holding the token crashes, an intricate distributed procedure usually has to be involved to make sure the distributed program will continue functioning properly. Losing a token in a distributed system becomes more challenging if the system is scaled up. This problem stems from the increasing probabilities of machine and network failures in large-scale systems. In the permission-based approach, distributed mutual exclusion can be achieved by requiring tasks to ask for permissions (e.g., locks) in order to access shared data. This approach can be implemented using either **centralized** or **decentralized** algorithms. In centralized algorithms, a coordinator for granting permissions is employed. A task can always make requests to the coordinator, asking for permissions. The coordinator can either provide or deny permissions, depending on whether there are tasks already accessing the requested data. The coordinator ensures that only one task can write on a shared piece of data at a time, yet can allow multiple reads from multiple tasks on the same data to proceed. Centralized algorithms are easy to implement, robust to starvation, and exhibit fairness. In particular, permissions can be provided in the order that they are asked for and for specified allotted times, which ensures that every task gets a chance to make requests. Nonetheless, centralized algorithms suffer from serious problems. First, the coordinator exposes a single point of failure (SPOF). That is, if the coordinator fails, the whole system will go down. Second, the coordinator can become a performance bottleneck, especially when scaling up the quantities of nodes and users. To address these two main drawbacks of centralized algorithms, decentralized algorithms suggest splitting the central coordinator into multiple coordinators. [1] Subsequently, for a task to acquire (write) permission, it needs to get a majority vote from coordinators (see the **Symmetrical and Asymmetrical Architectural Models** section for an introduction on voting

mechanisms). Clearly, obtaining these permissions makes the distributed program less vulnerable to an SPOF. More precisely, a distributed program with a decentralized, mutually exclusive algorithm can tolerate K out of 2K + 1 coordinator failures. [1] Also, decentralized algorithms remove the performance bottleneck revealed in centralized algorithms. In contrary, decentralized algorithms are more complex to implement and maintain than centralized ones. In general, implementation and maintenance complexities can impede scalability, especially if the number of control messages increases dramatically.

Fault Tolerance

One basic feature that distinguishes clouds and other distributed systems from uniprocessor systems is the concept of partial failures. Specifically, if one node or component fails in a distributed system, the whole system may be able to continue functioning. On the other hand, if one component (e.g., the RAM) fails in a uniprocessor system, the whole system will also fail. A crucial objective in designing distributed systems/programs is to construct them in a way that they can tolerate partial failures automatically, without seriously affecting performance. A key technique for masking faults in distributed systems is to use hardware redundancy, such as the RAID technology (see Unit 4). In most cases, however, distributed programs cannot depend solely on the underlying hardware fault-tolerance techniques of distributed systems. Among the popular techniques that the distributed programs can apply is software redundancy.

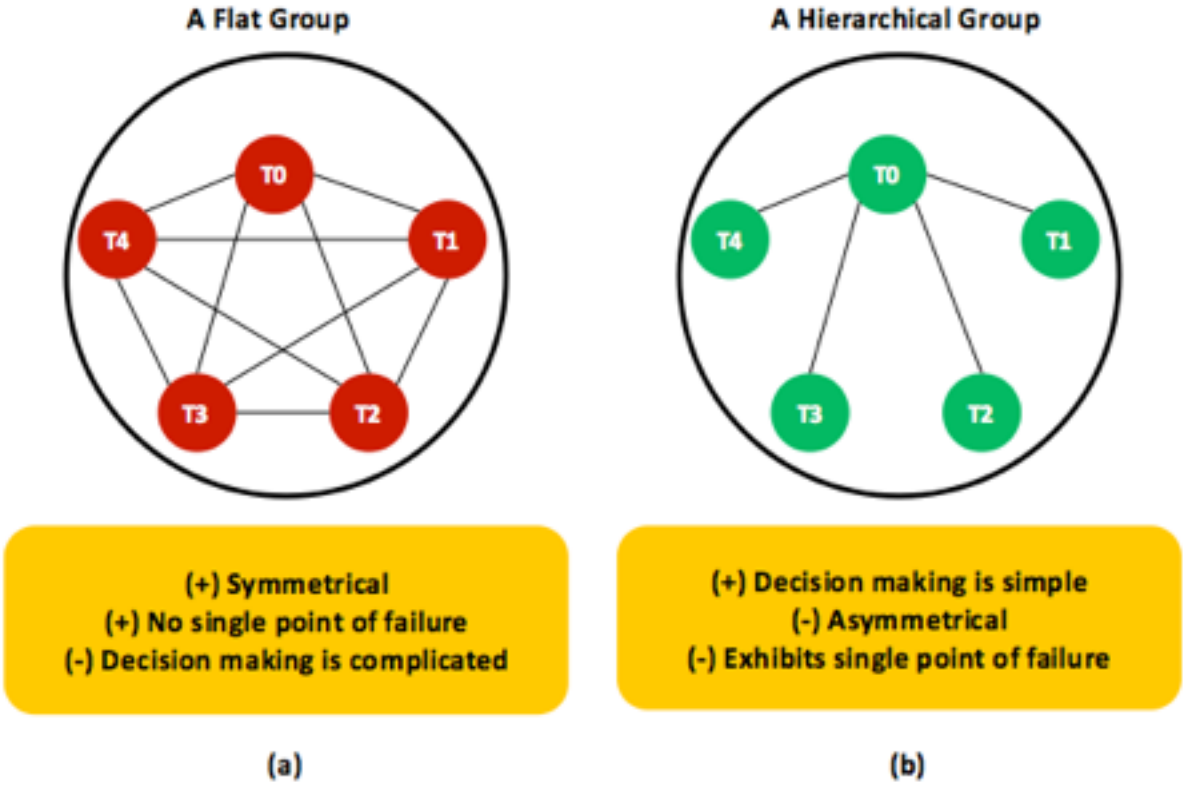

Figure 5.15: Two classical ways to employ task redundancy. (a) A flat group of tasks. (b) A hierarchical group of tasks with a central process (i.e., T0, whereby T1 stands for task 1.).

One common type of software redundancy is task redundancy (also called resiliency, or **replication**), which protects against task failures and slowness. Tasks can be replicated as flat or hierarchical groups, exemplified in Figure 5.15. In flat groups (see Figure 5.15(a)), all tasks are identical in that they all carry the same work. Eventually, only the result of one task is accepted, and the other results are discarded. Obviously, flat groups are symmetrical and preclude SPOFs: if one task crashes, the application will stay in business, yet the group will become smaller until recovered. However, if for some applications a decision is to be made (e.g., acquiring a lock), a voting mechanism may be required. As discussed earlier, voting mechanisms incur implementation complexity, communication delays, and performance overheads.

A hierarchical group (see Figure 5.15(b)) usually employs a coordinator task and specifies the rest of the tasks as workers. In this model, when a user request is made, it gets first forwarded to the coordinator who, in turn, decides which worker is best suited to fulfill the request. Clearly, hierarchical and flat groups reflect opposing properties. In particular, the coordinator is an SPOF and a potential performance bottleneck (especially in large-scale systems with millions of users). In contrast, as long as the coordinator is protected, the whole group remains functional. Furthermore, decisions can be easily made, solely by the coordinator without bothering any worker or incurring communication delays and performance overheads. A hybrid of flat and hierarchical task groups is adopted by Hadoop MapReduce but only for task failures. Details on that adoption are provided in the section on Hadoop MapReduce.

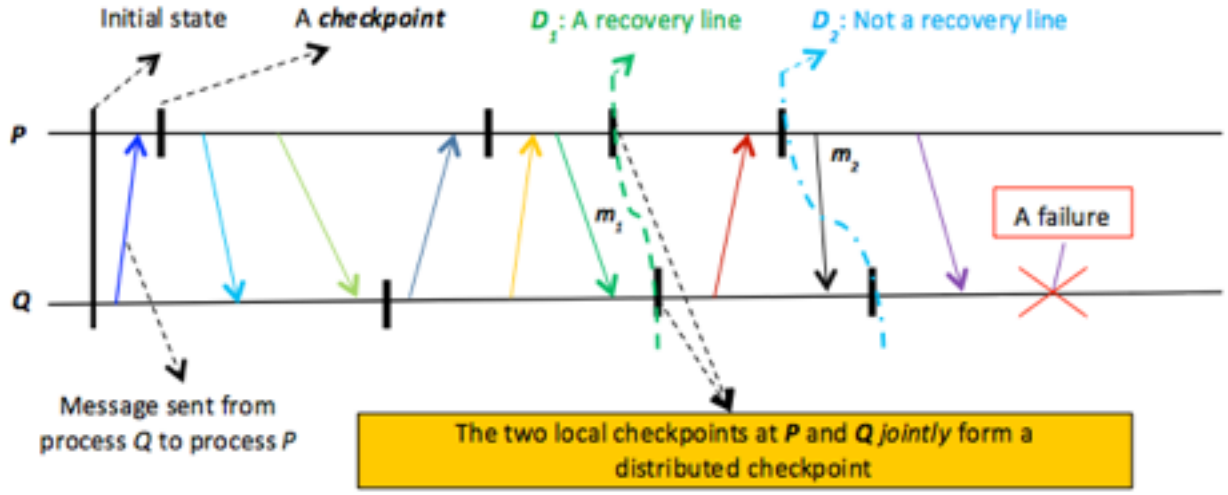

Figure 5.16: Demonstrating distributed checkpointing. **D1** is a valid distributed checkpoint, while **D2** is not because it is inconsistent. Specifically, **D2**'s checkpoint at Q indicates that **m2** has been received, while **D2**'s checkpoint at P does not indicate that **m2** has been sent.

In distributed programs, fault tolerance concerns not only surviving faults but also recovering from failures. The basic idea here is to replace a flawed state with a flaw-free state, and one way to achieve this goal is through backward recovery. This strategy

requires that the distributed program/system is brought from a current, flawed state to a previously correct state and relies on periodically recording the system's state at each process, which is called obtaining a **checkpoint**. When a failure occurs, recovery can be started from the last recorded correct state, typically called the **recovery line**.

Checkpoints of a distributed program at different processes in a distributed system constitute a distributed checkpoint. The process of capturing a distributed checkpoint is not easy because of one main reason. Specifically, a distributed checkpoint must maintain a consistent global state; that is, it should maintain the property that if a process **P** has recorded the receipt of a message, **m**, then there should be another process **Q** that has recorded the sending of **m**. After all, **m** must have come from a known process. Figure 5.16 demonstrates two distributed checkpoints, **D_1**, which maintains a consistent global state, and **D_2**, which does not. **D_1**'s checkpoint at **Q** indicates that **Q** has received a message, **m_1**, and the **D_1**'s checkpoint at **P** indicates that **P** has sent **m_1**, hence, making **D_1** consistent. In contrast, **D_2**'s checkpoint at **Q** indicates that message **m_2** has been received, and **D_2**'s checkpoint at **P** does not indicate that **m_2** has been sent from **P**. Therefore, **D_2** must be considered inconsistent and cannot be used as a recovery line.

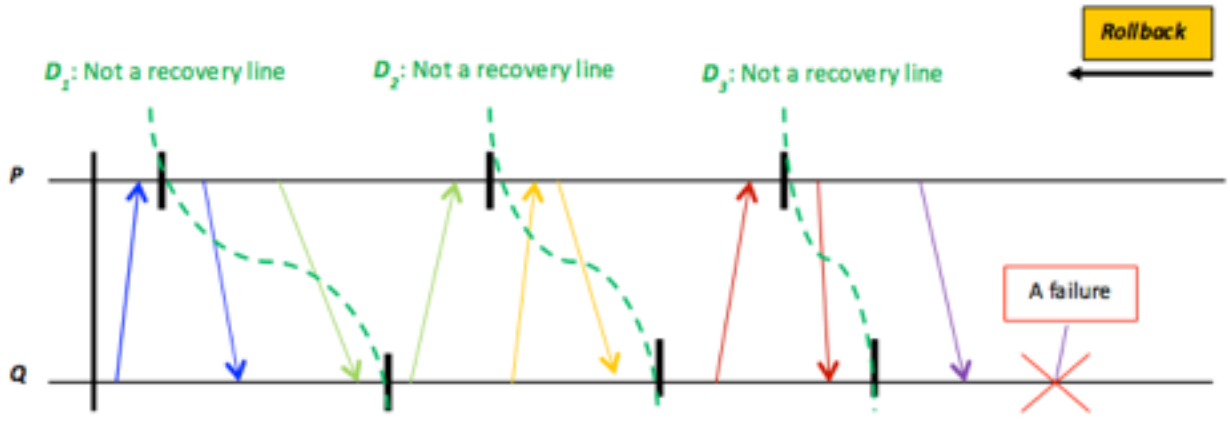

Figure 5.17: The domino effect that might result from rolling back each process (e.g., processes **P** and **Q**) to a saved, local checkpoint in order to locate a recovery line.

Neither **D1**, **D2**, nor **D3** are recovery lines because they exhibit inconsistent global states.

By rolling back each process to its most recently saved state, a distributed program/system can inspect a candidate distributed checkpoint to determine its consistency. When local states jointly form a consistent global state, a recovery line is said to be discovered. For instance, after a failure, the system exemplified in Figure 5.16 will roll back until hitting **D1**. Because **D1** reflects a global consistent state, we have obtained a recovery line. Unfortunately, the process of cascaded rollbacks is challenging because it can lead to a domino effect. As a specific example, Figure 5.17 exhibits a case in which a recovery line cannot be found. In particular, every distributed checkpoint in Figure 5.17 is indeed inconsistent. This pitfall makes distributed checkpointing a costly operation that may not converge to an acceptable recovery solution. Many fault-tolerant distributed systems thus combine checkpointing with message logging, recording each process message before sending and after a checkpoint has been taken. This tactic solves the problem of **D2** in Figure 5.16, for example. In particular, after **D2**'s checkpoint at **P** is taken, the send of **m2** will be marked in a log message at **P**, which, if merged with **D2**'s checkpoint at **Q**, can form a global consistent state. The Hadoop Distributed File System (HDFS) itself combines distributed checkpointing (the image file) and message logging (the edit file) to recover NameNode failures (see Unit 4). Pregel and GraphLab, discussed in later sections, apply only distributed checkpointing.

Scheduling

The effectiveness of a distributed program hinges on the manner in which its constituent tasks are scheduled over distributed machines. This scheduling is usually categorized into two main

classes, one for tasks and one for jobs. Tasks are the finest unit of execution granularity, and a job can encompass one or many tasks. Multiple users can submit numerous jobs simultaneously for execution on a cluster, and job schedulers determine which should go next. Hadoop MapReduce, for instance, utilizes a first-in, first-out (FIFO) job scheduler, whereby jobs run in order of receipt, and a scheduled job will occupy the whole cluster until the job has no more tasks to schedule. Hadoop MapReduce also employs other job schedulers, such as the Capacity and Fair Schedulers (see the Hadoop MapReduce section). After a job is granted the cluster, the scheduling decision morphs into how to schedule the job's component tasks. Tasks can be scheduled either close to the data that they will process, or anywhere. When tasks are scheduled near their data, locality is considered to be exploited. For example, Hadoop MapReduce incorporates two types of tasks, map and reduce tasks. Map tasks are scheduled in the vicinity of their uniform-sized input HDFS blocks, while reduce tasks are scheduled at any cluster nodes (anywhere), irrespective of their input data locations. Pregel and GraphLab, on the other hand, do not exploit any locality when scheduling tasks. Hadoop MapReduce, Pregel, and GraphLab task schedulers are detailed in the sections Hadoop MapReduce, Pregel, and GraphLab, respectively.

To avoid significant performance degradation, task schedulers must also account for heterogeneity in the underlying cloud system. Similar tasks that belong to the same job, for example, can be scheduled in a heterogeneous cloud at nodes of differing speed. This procedure, however, can introduce load imbalance and make jobs progress at the pace of their slowest tasks. Strategies such as Hadoop MapReduce's speculative execution can mitigate such problems (see the Hadoop MapReduce section).

In addition, task schedulers must seek to enhance system utilization and improve task parallelism. The objective here is to distribute tasks uniformly across cluster machines in a way that

utilizes the available resources fairly and increases parallelism effectively, but this goal presents some contradictory priorities. To begin, by evenly distributing tasks across cluster machines, locality may be affected. Machines in a Hadoop cluster, for instance, can contain different numbers of HDFS blocks. If one machine has a significantly larger number of blocks compared to other machines, locality would imply scheduling all map tasks in that machine. This disposition might make other machines less loaded and utilized. In addition, this strategy can reduce task parallelism by accumulating many tasks on the same machine. If locality is relaxed somewhat, utilization could be enhanced, loads across machines could be balanced, and task parallelism could be increased. However, relaxing locality would necessitate moving data towards tasks, and, if done injudiciously, relaxation could raise communication overheads, thereby impeding scalability and potentially degrading performance. In fact, with data centers hosting thousands of machines, moving data frequently towards distant tasks might become a major bottleneck. To improve performance and reduce costs, an optimal task scheduler should strike a balance between system utilization, load balancing, task parallelism, communication overheads, and scalability. Unfortunately, in practice, this ideal is hard to realize, and, indeed, most task schedulers attempt to optimize one objective and overlook the others.

Another major challenge when scheduling jobs and tasks is to meet what are called service-level objectives (SLOs), which reflect the performance expectations of end users. Amazon, Google, and Microsoft have identified SLO violations as a major cause of user dissatisfaction. [1] An SLO might be expressed, for example, as a maximum acceptable latency for allocating desired resources to a job, a soft/hard deadline to finish a job, or GPU preferences for certain tasks. In multitenant, heterogeneous clusters, SLOs are hard to achieve, especially when new jobs arrive while others are executing. This situation could require suspending currently

running tasks and allowing the new ones to proceed in order to meet their own specified SLOs. The capability to suspend and resume tasks is called *task elasticity*. Unfortunately, most distributed analytics engines—including Hadoop MapReduce, Pregel, and GraphLab—do not yet support task elasticity. Enabling elasticity is quite challenging and requires identifying safe points at which a task can be suspended without affecting its correctness and such that its committed work need not be repeated on resumption. Clearly, this capability resembles context switching in modern operating systems.

Introduction to Distributed Programming for the Cloud Summary

- Computer programs can be classified as sequential, parallel, distributed, and concurrent programs.
- A sequential program runs in program order on a single CPU (or, more specifically, a core).
- A concurrent program is a set of sequential programs that share one or more CPUs in time during execution.
- A parallel program is a set of sequential programs that overlap in time by running on separate CPUs in parallel.
- Parallel programs that run on separate CPUs on distinct networked machines are called distributed programs.
- A thread is the smallest sequence of instructions that an OS can manage through its scheduler. Multiple threads constitute a process, all of which share the same address space of the processor. Multiple processes constitute a task, while multiple tasks are typically grouped together as a job.

- Distributed programming is crucial to solving many problems, such as scientific workloads, big data computation, databases, and search engines.
- Multiple concerns dictate distributed program design for clouds; namely programming model, computation model, and program architecture.
- Typical programming models include the shared-memory model and the message-passing model.
- The shared-memory model assumes a shared address space, which is accessible by all tasks. Tasks communicate with each other by reading and writing to this shared address space. Communication among the tasks must be explicitly synchronized (using constructs such as barriers, semaphores, and locks). OpenMP is an example of a shared-memory model programming language.
- In the message-passing model, tasks do not share an address space and can only communicate to each other by explicitly sending and receiving messages. MPI is an example of a message-passing model programming language.
- Programming models are also classified as synchronous and asynchronous, based on the orchestration of the various tasks that are running in parallel. Synchronous programming models force all component tasks to operate in lock-step mode, while asynchronous models do not.
- Programs can also be classified according the type of parallelism they embody. They can either be data parallel or graph parallel.
- Data-parallel models focus on distributing the data over multiple machines while running the same code on each, also called the single-program, multiple-data (SPMD) model.
- Graph parallelism models focus on distributing computation as opposed to data, also called the multiple-program, multiple-data (MPMD) model.

- Tasks in a distributed programming model can be arranged into two distinct architectural models: asymmetric/master-slave and symmetric/peer-to-peer architectures.
- The master-slave organization requires one or more tasks to be specifically designated as the master tasks, which will coordinate the execution of the program among the slave tasks.
- The peer-to-peer organization consists of a set of tasks which are all equal but requires more complicated schemes to organize computation and make decisions.
- Major challenges in building cloud programs include managing scalability, communication, heterogeneity, synchronization, fault tolerance, and scheduling.
- Programs cannot be infinitely sped up by virtue of Amdahl's law, which expresses the limit on the speedup of a program as a function of the fraction of the program's time spent executing code that is serial in nature.
- Efficiently managing communication among distributed tasks dictates performance for many applications. Strategies to improve communication bottlenecks in the cloud include colocating highly communicating tasks and effectively managing the partitioning of data to map data to nodes that are closest to it.
- Clouds bring heterogeneity in terms of the underlying physical hardware, which is typically masked from the end user through virtualization. Programs running on the cloud that can account for and adjust based on heterogeneous hardware can benefit in terms of performance.
- Robust synchronization techniques are a must in distributed programming to deal with issues such as deadlocks.
- Fault tolerance poses a serious challenge in programming for clouds. Programs must anticipate for and recover against failures of software and hardware while running in the cloud.

- Task and job scheduling techniques take into account the unique nature of cloud resources in order to maximize performance.