Distributed Programming for the Cloud: Models, Challenges and Analytics Engines

Mohammad Hammoud and Majd F. Sakr

July 18, 2013

Contents

Chapter 1

The effectiveness of cloud programs hinges on the manner in which they are designed, implemented and executed. Designing and implementing programs for the cloud requires several considerations. First, they involve specifying the underlying programming model, whether message-passing or shared-memory. Second, they entail developing synchronous or asynchronous computation model. Third, cloud programs can be tailored for graph or data parallelism, which require employing either data striping and distribution, or graph partitioning and mapping. Lastly, from architectural and management perspectives, a cloud program can be typically organized in two ways, master/slave or peer-to-peer. Such organizations define the program's complexity, efficiency and scalability.

Added to the above design considerations, when constructing cloud programs, special attention must be paid to various challenges like scalability, communication, heterogeneity, synchronization, fault-tolerance and scheduling. First, scalability is hard to achieve in large-scale systems (e.g., clouds) due to several reasons such as the inability of parallelizing all parts of algorithms, the high probability of load-imbalance, and the inevitability of synchronization and communication overheads. Second, exploiting locality and minimizing network traffic are not easy to accomplish on (public) clouds since network topologies are usually unexposed. Third, heterogeneity caused by two common realities on clouds, virtualization environments and variety in datacenter components, impose difficulties in scheduling tasks and masking hardware and software differences across cloud nodes. Fourth, synchronization mechanisms must guarantee mutual exclusive accesses as well as properties like avoiding deadlocks and transitive closures, which are highly likely in distributed settings. Fifth, fault-tolerance mechanisms, including task resiliency, distributed checkpointing and message logging should be incorporated since the likelihood of failures increases on large-scale (public) clouds. Finally, task locality, high parallelism, task elasticity and service level objectives (SLOs) need to be addressed in task and job schedulers for effective programs' executions.

While designing, addressing and implementing the requirements and challenges of cloud programs are crucial, they are difficult, require time and resource investments, and pose correctness and performance issues. Recently, distributed analytics engines such as MapReduce, Pregel and GraphLab were developed to relieve programmers from worrying about most of the needs to construct cloud programs, and focus mainly on the sequential parts of their algorithms. Typically, these analytics engines automatically parallelize sequential algorithms provided by users in high-level programming languages like Java and C++, synchronize and schedule constituent tasks and jobs, and handle failures, *all* without any involvement from users/developers. In this Chapter, we first define some common terms in the theory of distributed programming, draw a requisite relationship between distributed systems and clouds, and discuss the main requirements and challenges for building distributed programs for clouds. While discussing the main requirements for building cloud programs, we indicate how MapReduce, Pregel and GraphLab address each requirement. Finally, we close up with a summary on the Chapter and a comparison between MapReduce, Pregel and GraphLab.

1.1 Taxonomy of Programs

A computer program consists of variable declarations, variable assignments, expressions and flow control statements written typically using a high-level programming language such as Java or C++. Computer programs are compiled before executed on machines. After compilation, they are converted to machine instructions/code that run over computer processors either *sequentially* or *concurrently* in an in-order or out-of-order manners, respectively. A **sequential program** is a program that runs in the *program order*. The program order is the original order of statements in a program as specified by a programmer. A **concurrent program** is a set of sequential programs that *share in time* a certain processor when executed. Sharing in time (or timesharing) allows sequential programs to take turns in using a certain resource component. For instance, with a single CPU and multiple sequential programs, the Operating System (OS) can allocate the CPU to each program for a specific time interval; given that only one program can run at a time on the CPU. This can be achieved by using a specific CPU scheduler such as the round-robin scheduler [69].

Programs, being sequential or concurrent, are often named interchangeably as applications. A different term that is also frequently used alongside concurrent programs is **parallel programs**. Parallel programs are technically different than concurrent 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 multi-core machines, related sequential programs that are executed at different cores represent a parallel program, while related sequential programs that share the same CPU in

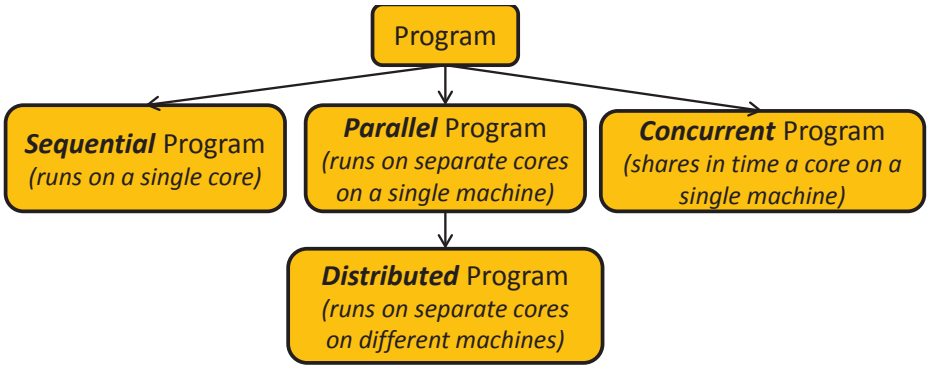

Figure 1.1: Our taxonomy of programs.

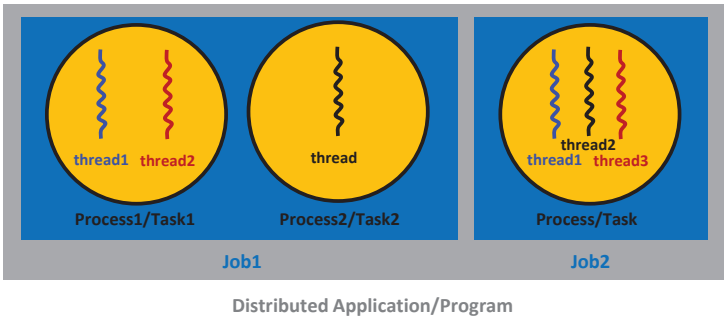

Figure 1.2: A demonstration of the concepts of processes, threads, tasks, jobs and applications.

time represent a concurrent program. To this end, we refer to a parallel program with multiple sequential programs that run on different networked machines (not on different cores at the same machine) as **distributed program**. Consequently, a distributed program can essentially include all types of programs. In particular, a distributed program can consist of multiple parallel programs, which in return can consist of multiple concurrent programs, which in return can consist of multiple sequential programs. For example, assume a set S that includes 4 sequential programs, P_1, P_2, P_3 and P_4 (i.e., $S = \{P_1, P_2, P_3, P_4\}$). A concurrent program, P', can encompass P_1 and P_2 (i.e., $P' = \{P_1, P_2\}$) whereby P_1 and P_2 share in time a single core. Furthermore, a parallel program, P'', can encompass P' and P_3 (i.e., $P'' = \{P', P_3\}$) whereby P' and P_3 overlap in time over multiple cores on the same machine. Lastly, a distributed program, P''', can encompass P'' and P_4 (i.e., $P''' = \{P'', P_4\}$) whereby P'' runs on different cores on the same machine and P_4 runs on a different machine as opposed to P''. In this Chapter we are mostly concerned with distributed programs. Fig. 1.1 shows our program taxonomy.

1.2 Tasks and Jobs in Distributed Programs

Another common term in the theory of parallel/distributed programming is *multitasking*. Multitasking is referred to overlapping the computation of one program with that of another. Multitasking is central to all modern Operating Systems (OSs), whereby an OS can overlap computations of multiple programs by means of a scheduler. Multitasking has become so useful that almost all modern programming languages are now supporting multitasking via providing constructs for

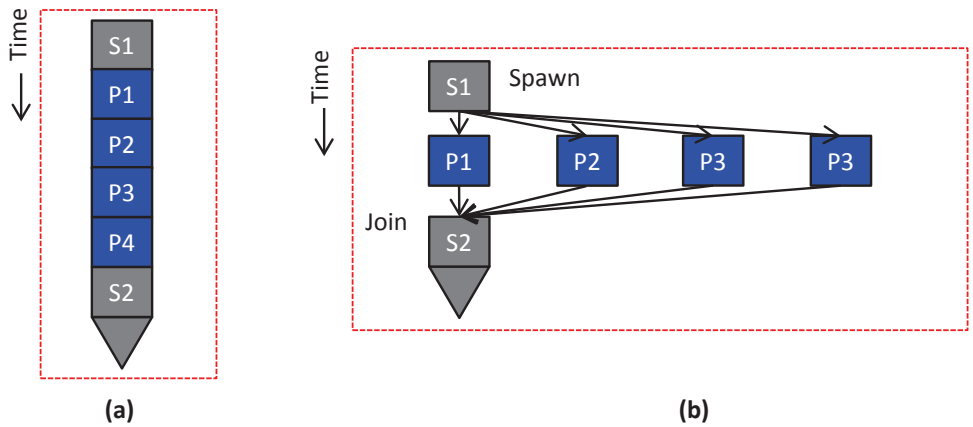

Figure 1.3: **(a)** A sequential program with serial (S_i) and parallel (P_i) parts. **(b)** A parallel/distributed program that corresponds to the sequential program in (a), whereby the parallel parts can be either distributed across machines or run concurrently on a single machine.

multithreading. A thread of execution is the smallest sequence of instructions that an OS can manage through its scheduler. The term thread was popularized by Pthreads (POSIX threads [59]), a specification of concurrency constructs that has been widely adopted, especially in UNIX systems [8]. A technical distinction is often made between **processes** and **threads**. A process runs using its own address space while a thread runs within the address space of a process (i.e., threads are parts of processes and not *standalone* sequences of instructions). A process can contain one or many threads. In principle, processes do not share address spaces among each other, while the threads in a process do share the process's address space. The term **task** is also used to refer to a small unit of work. In this Chapter, we use the term task to denote a process, which can include multiple threads. In addition, we refer to a group of tasks (which can only be 1 task) that belong to the same program/application as a **job**. An application can encompass multiple jobs. For instance, a fluid dynamics application typically consists of three jobs, one responsible for structural analysis, one for fluid analysis and one for thermal analysis. Each of these jobs can in return have multiple tasks to carry on the pertaining analysis. Fig. 1.2 demonstrates the concepts of processes, threads, tasks, jobs and applications.

1.3 Motivations for Distributed Programming

In principle, every sequential program can be parallelized by identifying sources of parallelism in it. Various analysis techniques at the algorithm and code levels can be applied to identify parallelism in sequential programs [67]. Once sources of parallelism are detected, a program can be split into *serial* and *parallel* parts as shown in Fig. 1.3. The parallel parts of a program can be run either concurrently or in parallel on a single machine, or in a distributed fashion across machines. Programmers parallelize their sequential programs primarily to run them faster and/or achieve higher throughput (e.g., number of data blocks read per hour). Specifically, in an ideal world, what programmers expect is that by parallelizing a sequential program into an *n-way* distributed

program, an *n-fold* decrease in execution time is obtained. Using distributed programs as opposed to sequential ones is crucial for multiple domains, especially for science. For instance, simulating a single protein folding can take years if performed sequentially, while it only takes days if executed in a distributed manner [67]. Indeed, the pace of scientific discovery is contingent on how fast some certain scientific problems can be solved. Furthermore, some programs have real time constraints by which if computation is not performed fast enough, the whole program might turn useless. For example, predicting the direction of hurricanes and tornados using weather modeling must be done in a timely manner or the whole prediction will be unusable. 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. We expect this trend to continue, at least for the foreseeable future.

Distributed programs have also found a broader audience outside science, such as serving search engines, Web servers and databases. For instance, much of the success of Google can be traced back to the effectiveness of its algorithms such as PageRank [42]. PageRank is a distributed program that is run within Google's search engine over thousands of machines to rank webpages. Without parallelization, PageRank cannot achieve its goals effectively. Parallelization allows also leveraging available resources effectively. For example, running a Hadoop MapReduce [27] program over a single Amazon EC2 instance will not be as effective as running it over a large-scale cluster of EC2 instances. Of course, committing jobs earlier on the cloud leads to fewer dollar costs, a key objective for cloud users. Lastly, distributed programs can further serve greatly in alleviating subsystem bottlenecks. For instance, I/O devices such as disks and network card interfaces typically represent major bottlenecks in terms of bandwidth, performance and/or throughput. By distributing work across machines, data can be serviced from multiple disks simultaneously, thus offering an increasingly 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 subsequently improves performances, increases throughput and reduces costs, especially on the cloud.

1.4 Models of Distributed Programs

Distributed programs are run on distributed systems which consist of networked computers. The cloud is a special distributed system. In this section, we first define distributed systems and draw a relationship between clouds and distributed systems. Second, in an attempt to answer the question of how to program the cloud, we present two traditional distributed programming models which can be used for that sake, the **shared-memory** and the **message-passing** programming

models. Third, we discuss the computation models that cloud programs can employ. Specifically, we describe the **synchronous** and **asynchronous** computation models. Fourth, we present the two main parallelism categories of distributed programs intended for clouds, **data parallelism** and **graph parallelism**. Lastly, we end the discussion with the architectural models that cloud programs can typically utilize, **master/slave** and **peer-to-peer** architectures.

1.4.1 Distributed Systems and the Cloud

Networks of computers are ubiquitous. The Internet, High Performance Computing (HPC) clusters, mobile phone, and in-car networks, among others, are common examples of networked computers. Many networks of computers are deemed as 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 certain problem or offer a specific service. One significant consequence of our definition pertains to clouds. Specifically, since a cloud is defined as a set of Internet-based software, platform and infrastructure services offered through a cluster of networked computers (i.e., a datacenter), it becomes a distributed system. Another consequence of our definition is that distributed programs will be the norm in distributed systems such as the cloud. In particular, we defined distributed programs in Section 1.1 as a set of sequential 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 to either send and receive messages explicitly, or read and write from/to a shared distributed memory supported by the underlying distributed system. We next discuss these two possible ways of enabling distributed tasks to interact over distributed systems.

1.4.2 Traditional Programming Models and Distributed Analytics Engines

A distributed programming model is an abstraction provided to programmers so that they can translate their algorithms into distributed programs that can execute over distributed systems (e.g., the cloud). A distributed programming model defines how easily and efficiently algorithms can be specified as distributed programs. For instance, a distributed programming model that highly abstracts architectural/hardware details, automatically parallelizes and distributes computation, and transparently supports fault-tolerance is deemed an easy-to-use programming model. The efficiency of the model, however, depends on the effectiveness of the techniques that underlie the model. There are two classical distributed programming models that are in wide use, **shared-memory** and **message-passing**. The two models fulfill different needs and suit different circumstances. Nonetheless, they are elementary in a sense that they only provide a basic

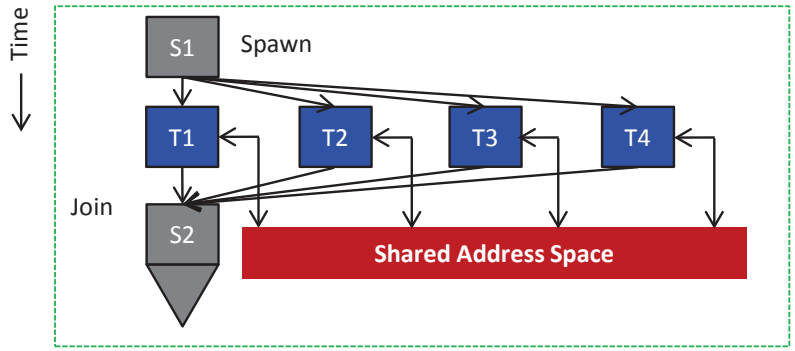

Figure 1.4: Tasks running in parallel and sharing an address space through which they can communicate.

interaction model for distributed tasks and lack any facility to automatically parallelize and distribute tasks, or tolerate faults. Recently, there have been other advanced models that address the inefficiencies and challenges posed by the shared-memory and the message-passing models, especially upon porting them to the cloud. Among these models are MapReduce [17], Pregel [49] and GraphLab [47]. These models are built upon the shared-memory and the message-passing programming paradigms, yet are more involved and offer various properties that are essential for the cloud. As these models highly differ from the traditional ones, we refer to them as **distributed analytics engines**.

1.4.2.1 The Shared-Memory Programming Model

In the shared-memory programming model, tasks can communicate by reading and writing to shared memory (or disk) locations. Thus, the abstraction provided by the shared-memory model is that tasks can access any location in the distributed memories/disks. This is similar to threads of a single process in Operating Systems, whereby all threads share the process address space and communicate by reading and writing to that space (see Fig. 1.4). Therefore, with shared-memory, data is not explicitly communicated but implicitly exchanged via sharing. Due to sharing, the shared-memory programming model entails the usage of **synchronization** mechanisms within distributed programs. Synchronization is needed to control the order in which read/write operations are performed by various tasks. In particular, what is required is that distributed tasks are prevented from simultaneously writing to a shared data, so as to avoid corrupting the data or making it inconsistent. This can be typically 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 with consuming data. Locks protect **critical sections**, or regions that at most 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, respectively. A lock can be held by only one task at a time, and other tasks cannot acquire it until released. Lastly,

a barrier defines a point at 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 (details on the challenges that pertain to synchronization are provided in Section 1.5.4).

Fig. 1.5 shows an example that transforms a simple sequential program into a distributed program using the shared-memory programming model. The sequential program adds up the elements of two arrays $\boldsymbol{b}$ and $\boldsymbol{c}$ and stores the resultant elements in array $\boldsymbol{a}$. Afterwards, if any element in $\boldsymbol{a}$ is found to be greater than 0, it is added to a grand sum. The corresponding distributed version assumes only two tasks and splits the work evenly across 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; hence, a lock is used to protect it. In addition, no task can print the grand sum before every other task has finished its part, thus a barrier is utilized prior to the printing statement. As shown in the program, the communication between the two tasks is implicit (via reads and writes to shared arrays and variables) and synchronization is explicit (via locks and barriers). Lastly, as pointed out earlier, sharing of data has to be offered by the underlying distributed system. Specifically, the underlying distributed system should provide an illusion that all memories/disks of the computers in the system form a single shared space 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 denoted as *Distributed Shared Memory* (DSM) [44,45,70]. A common programing language that can be used on DSMs and other distributed shared systems is OpenMP [55].

Other modern examples that employ a shared-memory view/abstraction are MapReduce and GraphLab. To summarize, the shared-memory programming model entails two main criteria: (1) developers need not explicitly encode functions that send/receive messages in their programs, and (2) the underlying storage layer provides a shared view to all tasks (i.e., tasks can transparently access any location in the underlying storage). Clearly, MapReduce satisfies the two criteria. In particular, MapReduce developers write only two sequential functions known as the map and the reduce functions (i.e., no functions are written or called that explicitly send and receive messages). In return, MapReduce breaks down the user-defined map and reduce functions into multiple tasks denoted as Map and Reduce tasks. All Map tasks are encapsulated in what is known as the Map phase, and all Reduce tasks are encompassed in what is called the Reduce phase. Subsequently, all communications occur only between the Map and the Reduce phases and under the full control of the engine itself. In addition, any required synchronization is also handled by the MapReduce engine. For instance, in MapReduce the user-defined reduce function cannot be applied before all the Map phase output (or intermediate output) are shuffled,

```
for (i=0; i<8; i++)
  a[i] = b[i] + c[i];
sum = 0;
for (i=0; i<8; i++)
  if (a[i] > 0)
    sum = sum + a[i];
Print sum;
```

(a)

```
begin parallel // spawn a child thread
private int start_iter, end_iter, i;
shared int local_iter=4, sum=0;
shared double sum=0.0, a[], b[], c[];
shared lock_type mylock;

start_iter = getid() * local_iter;
end_iter = start_iter + local_iter;
for (i=start_iter; i<end_iter; i++)
  a[i] = b[i] + c[i];
barrier;

for (i=start_iter; i<end_iter; i++)
  if (a[i] > 0) {
    lock(mylock);
      sum = sum + a[i];
    unlock(mylock);
  }
barrier;    // necessary

end parallel // kill the child thread
Print sum;
```

(b)

Figure 1.5: **(a)** A sequential program that sums up elements of two arrays and compute a grand sum on results that are greater than zero. **(b)** A distributed version of the program in (a) coded using the shared-memory programming model.

merged and sorted. Obviously, this requires a barrier between the Map and the Reduce phases, which the MapReduce engine internally incorporates. Second, MapReduce uses the Hadoop Distributed File System (HDFS) [27] as an underlying storage layer. As any typical distributed file system, HDFS provides a shared abstraction for all tasks, whereby any task can transparently access any location in HDFS (i.e., as if accesses are local). Therefore, MapReduce is deemed to offer a shared-memory abstraction provided internally by Hadoop (i.e., the MapReduce engine and HDFS).

Similar to MapReduce, GraphLab offers a shared-memory abstraction [24, 47]. In particular, GraphLab eliminates the need for users to explicitly send/receive messages in update functions (which represent the user-defined computations in it), and provides a shared view among vertices in a graph. To elaborate, GraphLab allows **scopes** of vertices to overlap and vertices to read and write from and to their scopes. The scope of a vertex v (denoted as S_v) is the data stored in v and in all v's adjacent edges and vertices. Clearly, this poses potential read-write and write-write conflicts between vertices sharing scopes. The GraphLab engine (and not the users) synchronizes accesses to shared scopes and ensures consistent parallel execution via supporting three levels of consistency settings, **Full Consistency**, **Edge Consistency** and **Vertex Consistency**. Under Full Consistency, the update function at each vertex has an exclusive read-write access to its vertex, adjacent edges and adjacent vertices. While this guarantees strong consistency and full correctness, it limits parallelism and consequently performance. Under Edge Consistency, the

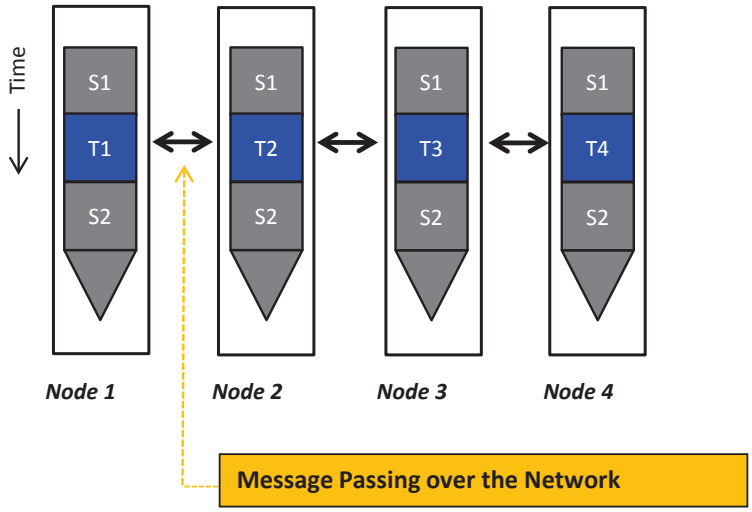

Figure 1.6: Tasks running in parallel using the message-passing programming model whereby the interactions happen only via sending and receiving messages over the network.

update function at a vertex has an exclusive read-write access to its vertex and adjacent edges, but only a read access to adjacent vertices. Clearly, this relaxes consistency and enables a superior leverage of parallelism. Finally, under Vertex Consistency, the update function at a vertex has an exclusive write access to only its vertex, hence, allowing all update functions at all vertices to run simultaneously. Obviously, this provides the maximum possible parallelism but, in return, the most relaxed consistency. GraphLab allows users to choose whatever consistency model they find convenient for their applications.

1.4.2.2 The Message-Passing Programming Model

In the message-passing programming model, distributed tasks communicate by sending and receiving messages. In other words, distributed tasks do not share an address space at which they can access each other's memories (see Fig. 1.6). Accordingly, the abstraction provided by the message-passing programming model is similar to that of processes (and not threads) in Operating Systems. The message-passing programming model incurs communication overheads (e.g., variable network latency and potentially excessive data transfers) for explicitly sending and receiving messages that contain data. Nonetheless, the explicit sends and receives of messages serve in implicitly synchronizing the sequence of operations imposed by the communicating tasks. Fig. 1.7 demonstrates an example that transforms the same sequential program shown in Fig. 1.5 (a) into a distributed program using message passing. Initially, it is assumed that only a main task with id = 0 has access to arrays $\boldsymbol{b}$ and $\boldsymbol{c}$. Thus, assuming the existence of only two tasks, the main task sends firstly parts of the arrays to the other task (using an explicit send operation) in order to evenly split the work among the two tasks. The other 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 part of data and collects the local sum of the other task before aggregating and printing a grand sum. As shown,

```
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 1.7: A distributed program that corresponds to the sequential program in Fig. 1.5 (a) coded using the message-passing programming model.

for every send operation, there is a corresponding receive operation. No explicit synchronization is needed.

Clearly, the message-passing programming model does not necessitate any support from the underlying distributed system due to relying on explicit messages. Specifically, no illusion for a single shared address space is required from the distributed system in order for the tasks to interact. A popular example of a message-passing programming model is provided by the **Message Passing Interface** (MPI) [50]. MPI is a message passing, industry-standard 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 [52]. A common analytics engine that employs the message-passing programming model is Pregel. In Pregel, vertices can communicate only by sending and receiving messages, which should be explicitly encoded by users/developers.

To this end, Table 1.1 compares between the shared-memory and the message-passing programming models in terms of 5 aspects, *communication*, *synchronization*, *hardware support*, *development effort* and *tuning effort*. Shared-memory programs are easier to develop at the outset because programmers need not worry about how data is laid out or communicated. Furthermore, the code structure of a shared-memory program is often not much different than its respective sequential one. Typically, only additional directives are added by programmers to specify paral-

lel/distributed tasks, scope of variables and synchronization points. In contrast, message-passing programs require a switch in the programmer's thinking, wherein the programmer needs to think a-priori about how to partition data across tasks, collect data, and communicate and aggregate results using explicit messaging. Alongside, scaling up the system entails less tuning (denoted as tuning effort in Table 1.1) of message-passing programs as opposed to shared-memory ones. Specifically, when using a shared-memory model, how data is laid out and where it is stored start to affect performance significantly. To elaborate, large-scale distributed systems like the cloud imply non-uniform access latencies (e.g., accessing remote data takes more time than accessing local data), thus enforces programmers to lay out data close to relevant tasks. While message passing programmers think about partitioning data across tasks during pre-development time, shared memory programmers do not. Hence, shared memory programmers need (most of the time) to address the issue during post-development time (e.g., through data migration or replication). Clearly, this might dictate a greater post-development tuning effort as compared to the message-passing case. Finally, synchronization points might further become performance bottlenecks in large-scale systems. In particular, as the number of users that attempt to access critical sections increases, delays and waits on such sections also increase. More on synchronization and other challenges involved in programming the cloud are presented in Section 1.5.

Table 1.1: A Comparison Between the Shared-Memory and the Message-Passing Programming Models.

Aspect	The Shared-Memory Model	The Message-Passing Model
Communication	Implicit	Explicit
Synchronization	Explicit	Implicit
Hardware Support	Usually Required	Not Required
Initial Development Effort	Lower	Higher
Tuning Effort Upon Scaling Up	Higher	Lower

1.4.3 Synchronous and Asynchronous Distributed Programs

Apart from programming models, distributed programs, being shared-memory or message-passing based, can be specified as either **synchronous** or **asynchronous** programs. A distributed program is synchronous if and only if the distributed tasks operate in a *lock-step mode*. That is, if there is some constant $c \geq 1$ and any task has taken $c + 1$ steps, every other task should have taken at least 1 step [71]. Clearly, this entails a coordination mechanism through which the activities of tasks can be synchronized and the lock-step mode be accordingly enforced. Such a mechanism usually has 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 [9]. A distributed program that is not synchronous is referred to as asynchronous. Asynchronous programs expose no requirements for waiting at predetermined points and/or for the arrival of specific data. Obviously, this has less effect on performance but implies that the correctness/validity of the program must be assessed. In short, the distinction between synchronous

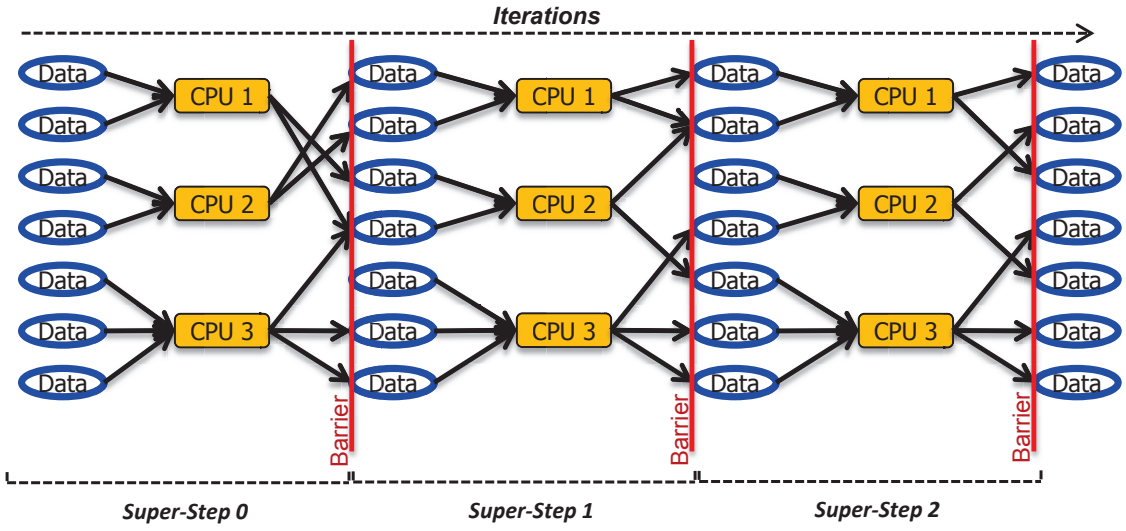

Figure 1.8: The Bulk Synchronous Parallel (BSP) model.

and asynchronous distributed programs refers to the presence or absence of a (global) coordination mechanism that synchronizes the operations of tasks and imposes a lock-step mode. As specific examples, MapReduce and Pregel programs are synchronous, while GraphLab ones are asynchronous.

One synchronous model that is commonly employed for effectively implementing distributed programs is the **Bulk Synchronous Parallel** (BSP) model [74] (see Fig. 1.8). The Pregel programs follow particularly the BSP model. BSP is defined as a combination of three attributes, *components*, a *router* and a *synchronization* method. A component in BSP consists of a processor attached with data stored in local memory. BSP, however, does not exclude other arrangements such as holding data in remote memories. BSP is neutral about the number of processors, be it two or millions. BSP programs can be written for v virtual distributed processors to run on p physical distributed processors, where v is larger than p. BSP is based on the message-passing programming model, whereby components can only communicate by sending and receiving messages. This is achieved through a router which in principle can only pass messages point-to-point between pairs of components (i.e., no broadcasting facilities are available, though it can be implemented using multiple point-to-point communications). Finally, as being a synchronous model, 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. Besides, components in super-step S are allowed to send messages to components in super-step $S + 1$, and are (implicitly) allowed to receive messages from components in super-step $S - 1$. Tasks within every super-step operate simultaneously and do not communicate with each other. Tasks across super-steps move in a lock-step mode as suggested by any synchronous model. Specifically, no task in super-step $S + 1$ is allowed to start before every task in super-step S commits. To satisfy this condition, BSP applies a global barrier-style synchronization mechanism as shown in Fig 1.8.

BSP does not suggest simultaneous accesses to the same memory location, hence, precludes the requirement for a synchronization mechanism other than barriers. Another primary concern in a distributed setting is to allocate data in a way that computation will not be slowed down by non-uniform memory access latencies or uneven loads among individual tasks. BSP promotes uniform access latencies via enforcing local data accesses. In particular, data are communicated across super-steps before triggering actual task computations. As such, BSP carefully segregates computation from communication. Such a segregation entails that no particular network topology is favored beyond the requirement that high throughput is delivered. Butterfly, hypercube and optical crossbar topologies can all be employed with BSP. With respect to task loads, data can still vary across tasks within a super-step. This typically depends on: (1) the responsibilities that the distributed program imposes on its constituent tasks, and (2) the characteristics of the underlying cluster nodes (more on this in Section 1.5.1). As a consequence, tasks that are lightly loaded (or are run on fast machines) will potentially finish earlier than tasks that are heavily loaded (or are run on slow machines). Subsequently, the time required to finish a super-step becomes bound by the slowest task in the super-step (i.e., a super-step cannot commit before the slowest task commits). This presents a major challenge for the BSP model as it might create load-imbalance, which usually degrades performance. Finally, it is worth noting that while 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 it is acceptable by the given application).

1.4.4 Data Parallel and Graph Parallel Computations

As distributed programs can be constructed using either the shared-memory or the message-passing programming models as well as specified as being synchronous or asynchronous, they can be tailored for different parallelism types. Specifically, distributed programs can either incorporate **data parallelism** or **graph parallelism**. Data parallelism is a form of parallelizing computation as a result of distributing data across multiple machines and running (in parallel) corresponding tasks on those machines. Tasks across machines may involve the same code or may be totally different. Nonetheless, in both cases, tasks will be applied to distinctive data. If tasks involve the same code, we classify the distributed application as *Single Program Multiple Data* (SPMD) application; otherwise we label it as *Multiple Program Multiple Data* (MPMD) application. Clearly, 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. One popular technique for distributing data is *file-striping*, by which a single file is partitioned and distributed across multiple servers. Another form of data parallelism is to distribute *whole* files (i.e., without striping) across machines, especially if files are small and their contained data

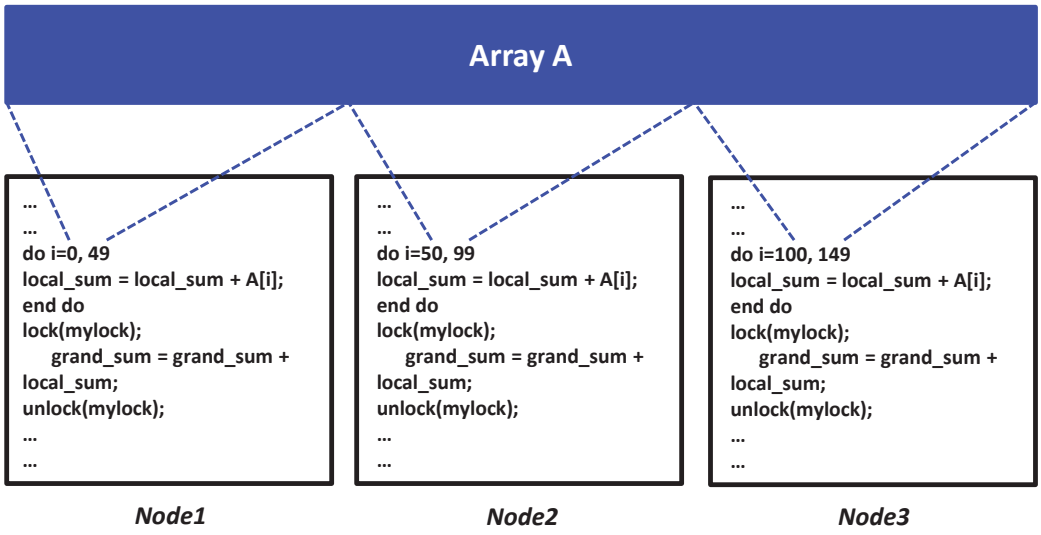

Figure 1.9: An SPMD distributed program using the shared-memory programming model.

exhibit very irregular structures. We note that data can be distributed among tasks either explicitly by using a message-passing model, or implicitly by using a shared-memory model (assuming an underlying distributed system that offers a shared-memory abstraction).

Data parallelism is achieved when each machine runs one or many tasks over different partitions of data. As a specific example, assume array $\boldsymbol{A}$ is shared among 3 machines in a distributed shared memory system. Consider also a distributed program that simply adds all elements of array $\boldsymbol{A}$. It is possible to charge machines 1, 2 and 3 to run the addition task, each on 1/3 of $\boldsymbol{A}$, or 50 elements, as shown in Fig. 1.9. 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 $\boldsymbol{A}$ can also be partitioned evenly and distributed accross 3 machines using the message-passing model as shown in Fig. 1.10. Each machine will run the addition task independently; nonetheless, summation results will have to be eventually aggregated at one main 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 $\boldsymbol{A}$. The main task, however, is further aggregating summation results, thus making it a little different than the other two tasks. Obviously, this makes the program MPMD.

As a real example, MapReduce uses data parallelism. In particular, input datasets are partitioned by HDFS into blocks (by default, 64MB per a block) allowing MapReduce to effectively exploit data parallelism via running a Map task per one or many blocks (by default, each Map task processes only 1 HDFS block). Furthermore, as Map tasks operate on HDFS blocks, Reduce tasks operate on the output of Map tasks denoted as **intermediate output** or **partitions**. In principle, each Reduce task can process one or many partitions. As a consequence, the data processed by Map and Reduce tasks become different. Moreover, Map and Reduce tasks are

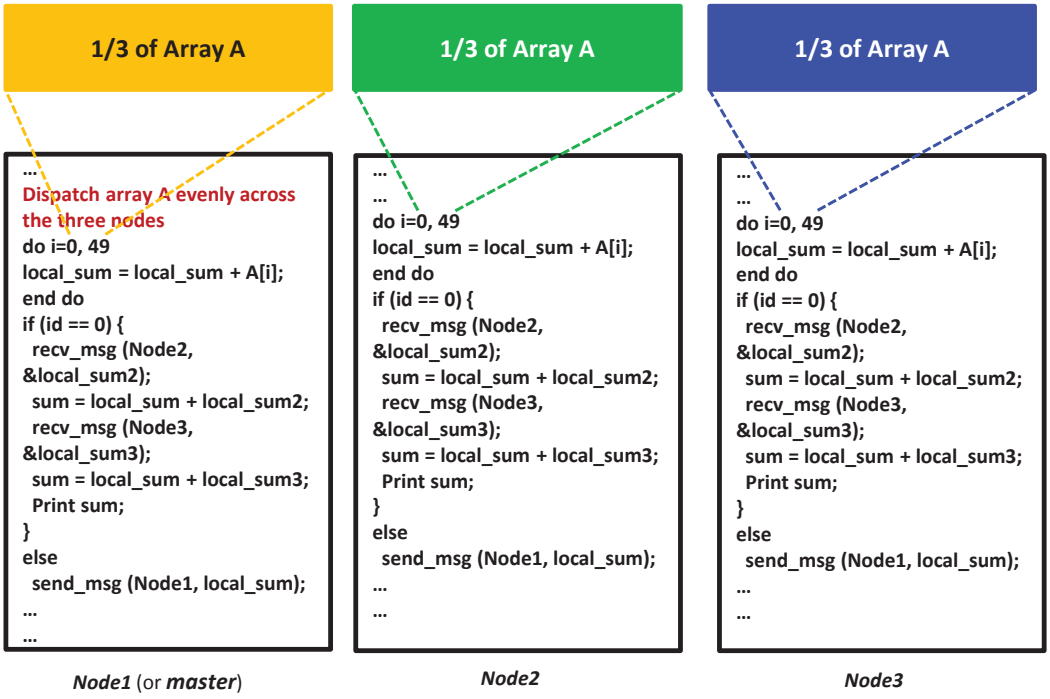

Figure 1.10: An MPMD distributed program using the message-passing programming model.

inherently dissimilar (i.e., the map and the reduce functions incorporate different binary codes). Therefore, MapReduce jobs lie under the MPMD category.

Graph parallelism contrasts with data parallelism. Graph parallelism is another form of parallelism that focuses more on distributing graphs as opposed to data. Indeed, most distributed programs fall somewhere on a continuum between data parallelism and graph parallelism. Graph parallelism is widely used in many domains such as machine learning, data mining, physics, and electronic circuit designs, 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 $\boldsymbol{G}$ is a pair ($\boldsymbol{V}$, $\boldsymbol{E}$), where $\boldsymbol{V}$ is a finite set of vertices and $\boldsymbol{E}$ is a finite set of pairwise relationships, $\boldsymbol{E} \subset \boldsymbol{V} \times \boldsymbol{V}$, called edges. Weights can be associated with vertices and edges to indicate the amount of work per each vertex and the communication data per each edge. To exemplify, let us consider a classical problem from circuit design. It is often the case in circuit design that pins of several components are to be kept electronically equivalent by wiring them together. If we assume $\boldsymbol{n}$ pins, then an arrangement of $\boldsymbol{n - 1}$ wires, each connecting two pins, can be employed. Of all such arrangements, the one that uses 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 ($\boldsymbol{u}$, $\boldsymbol{v}$) can be represented as an edge. A weight $\boldsymbol{w}(\boldsymbol{u}, \boldsymbol{v})$ can be set between $\boldsymbol{u}$ and $\boldsymbol{v}$ to encode the cost (i.e., the amount of wires needed) to connect $\boldsymbol{u}$ and $\boldsymbol{v}$. The problem becomes, how to find an acyclic subset, $\boldsymbol{S}$, of edges, $\boldsymbol{E}$, that connects all the vertices, $\boldsymbol{V}$, and whose total weight $\sum_{(u,v)\in S} \boldsymbol{w(u,v)}$ is the *minimum*. As $\boldsymbol{S}$ is acyclic and fully connected, it must result in a

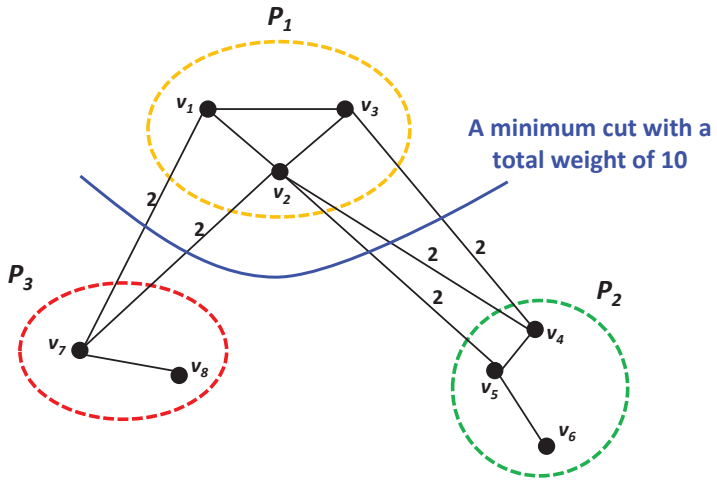

Figure 1.11: A graph partitioned using the edge cut metric.

tree known as the *minimum spanning tree*. Consequently, solving the wiring problem morphs into simply solving the minimum spanning tree problem. The minimum spanning tree problem is a classical problem and can be solved using Kruskal's or Prim's algorithms, to mention a few [15].

Once a problem is modeled as a graph, it can be distributed over machines in a distributed system using a **graph partitioning technique**. Graph partitioning implies dividing the work (i.e., the vertices) over distributed nodes for efficient distributed computation. As is the case 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. As such, graph partitioning enables what we refer to as *graph parallelism*. The standard objective of graph partitioning is to uniformly distribute the work over $\boldsymbol{p}$ processors by partitioning the vertices into $\boldsymbol{p}$ equally weighted partitions, while minimizing inter-node communication reflected by edges. Such an objective is typically referred to as the standard **edge cut metric** [34]. The graph partitioning problem is NP-hard [21], yet heuristics can be implemented to achieve near optimal solutions [34, 35,39]. As a specific example, Fig. 1.11 demonstrates 3 partitions, $\boldsymbol{P}_1$, $\boldsymbol{P}_2$ and $\boldsymbol{P}_3$ at which vertices $\boldsymbol{v}_1$,..., $\boldsymbol{v}_8$ are divided using the edge cut metric. Each edge has a weight of 2 corresponding to 1 unit of data being 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 very critical and can play a dramatic role in dictating the overall application performance. We discuss some of the challenges pertaining to graph partitioning in Section 1.5.3.

As real examples, both Pregel and GraphLab employ graph partitioning. Specifically, in Pregel each vertex in a graph is assigned a unique ID, and partitioning of the graph is accomplished via using a *hash(ID) mod N* function, where *N* is the number of partitions. The hash function is customizable and can be altered by users. After partitioning the graph, partitions are mapped to

cluster machines using a mapping function of a user choice. For example, a user can define a mapping function for a Web graph that attempts to exploit locality by co-locating vertices of the same Web site (a vertex in this case represents a Web page). In contrast to Pregel, GraphLab utilizes a *two-phase partitioning* strategy. In the first phase, the input graph is partitioned into k partitions using a hash-based random algorithm [47], with k being much larger than the number of cluster machines. A partition in GraphLab is called an **atom**. GraphLab does not store the actual vertices and edges in atoms, but *commands* to generate them. In addition to commands, GraphLab maintains in each atom some information about the atom's neighboring vertices and edges. This is denoted in GraphLab as **ghosts**. Ghosts are used as a caching capability for efficient adjacent data accessibility. In the second phase of the two-phase partitioning strategy, GraphLab stores the connectivity structure and the locations of atoms in an **atom index file** referred to as **meta-graph**. The atom index file encompasses k vertices (with each vertex corresponding to an atom) and edges encoding connectivity among atoms. The atom index file is split uniformly across the cluster machines. Afterwards, atoms are loaded by cluster machines and each machine constructs its partitions by executing the commands in each of its assigned atoms. By generating partitions via executing commands in atoms (and *not* directly mapping partitions to cluster machines), GraphLab allows future changes to graphs to be simply appended as additional commands in atoms without needing to re-partition the entire graphs. Furthermore, the same graph atoms can be reused for different sizes of clusters by simply re-dividing the corresponding atom index file and re-executing atom commands (i.e., only the second phase of the two-phase partitioning strategy is repeated). In fact, GraphLab has adopted such a graph partitioning strategy with the *elasticity* of clouds being in mind. Clearly, this improves upon the *direct and non-elastic* hash-based partitioning strategy adopted by Pregel. Specifically, in Pregel, if graphs or cluster sizes are altered after partitioning, the entire graphs need to be re-partitioned prior to processing.

1.4.5 Symmetrical and Asymmetrical Architectural Models

From architectural and management perspectives, a distributed program can be typically organized in two ways, **master/slave** (or **asymmetrical**) and **peer-to-peer** (or **symmetrical**) (see Fig. 1.12). There are other organizations, such as hybrids of asymmetrical and symmetrical, which do exist in literature [71]. For the purpose of our Chapter, we are only concerned with the master/slave and peer-to-peer organizations. In a master/slave organization, a central process known as the *master* handles all the logic and controls. All other processes are denoted as *slave* processes. As such, the interaction between processes is asymmetrical, whereby bi-directional connections are established between the master and all the slaves, and no interconnection is permitted between any two slaves (see Fig. 1.12 (a)). This requires that the master keeps track of every slave's network location within what is referred to as a *metadata* structure. In addition, this

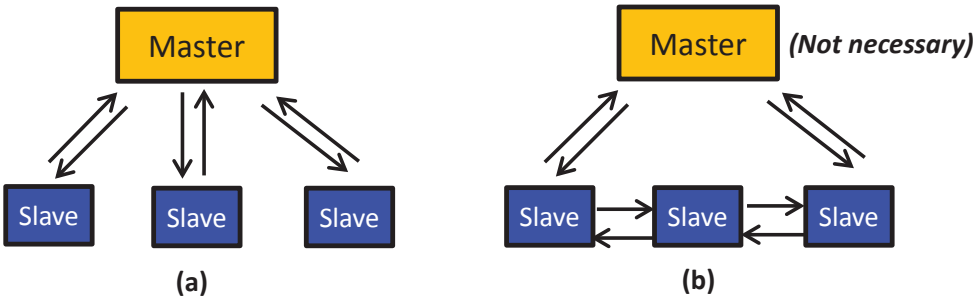

Figure 1.12: **(a)** A master/slave organization. **(b)** A peer-to-peer organization. The master in such an organization is optional (usually employed for monitoring the system and/or injecting administrative commands).

entails that each slave is always capable of identifying and locating the master.

The master in master/slave organizations can distribute the work among the slaves using one of two protocols, **push-based** or **pull-based**. In the push-based protocol, the master assigns work to the slaves without the slaves asking for that. Clearly, this might allow the master to apply fairness over the slaves via distributing the work equally among them. In contrast, this could also overwhelm/congest slaves that are currently experiencing some slowness/failures and are unable to keep up with work. Consequently, load-imbalance might occur, which usually leads to performance degradation. Nevertheless, smart strategies can be implemented by the master. In particular, the master can assign work to a slave if and only if the slave is *observed* to be *ready* for that. For this to happen, the master has to continuously monitor the slaves and apply some certain logic (usually complex) to accurately determine ready slaves. The master has also to decide upon the amount of work to assign to a ready slave so as fairness is maintained and performance is not degraded. In clouds, the probability of faulty and slow processes increases due to heterogeneity, performance unpredictability and scalability (see Section 1.5 for details on that). This might make the push-based protocol somehow inefficient on the cloud.

Unlike the push-based protocol, in the pull-based protocol, the master assigns work to the slaves only if they ask for that. This highly reduces complexity and potentially avoids load-imbalance, since the decision of whether a certain slave is ready to receive work or not is delegated to the slave itself. Nonetheless, the master still needs to monitor the slaves, usually to track the progresses 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).

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 (i.e., 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 upon each request). In contrary, using a master/slave organization simplifies decision making (e.g., allow a write transaction on a certain shared data). In particular, the master is always the sole entity which controls everything and can make any decision singlehandedly without bothering anyone else. This averts the employment of **voting mechanisms** [23, 71, 72], typically needed when only a *group* of entities (not a single entity) have 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. Lastly, as specific examples, Hadoop MapReduce and Pregel adopt master/slave organizations and apply the pull-based and the push-based protocols, respectively. We note, however, that recently, Hadoop has undergone a major overhaul to address several inherent technical deficiencies, including the reliability and availability of the JobTracker, among others. The outcome is a new version referred to as **Yet Another Resource Negotiator** (YARN) [53]. To elaborate, YARN still adopts a master/slave topology but with various enhancements. First, the resource management module, which is responsible for task and job scheduling as well as resource allocation, has been entirely detached from the master (or the JobTracker in Hadoop's parlance) and defined as a separate entity entitled as Resource Manager (RM). RM has been further sliced into two main components, the Scheduler (S) and the Applications Manager (AsM). Second, instead of having a single master for all applications, which was the JobTracker, YARN has defined a master per application, referred to as Application Master (AM). AMs can be distributed across cluster nodes so as to avoid application SPOFs and potential performance degradations. Finally, the slaves (or what is known in Hadoop as TaskTrackers) has remained effectively the same but are now called Node Managers (NMs).

In a Peer-to-Peer organization, logic, control and work are distributed evenly among tasks. That is, all tasks are equal (i.e., they all have the same capability) and no one is a boss. This makes peer-to-peer organizations symmetrical. Specifically, each task can communicate directly with tasks around it, without having to contact a master process (see Fig. 1.12 (b)). A master may be adopted, however, but only for purposes like monitoring the system and/or injecting administrative commands. In other words, as opposed to a master/slave organization, the presence of a master in a peer-to-peer organization is not requisite for the peer tasks to function correctly. Moreover, although tasks communicate with one another, their work can be totally independent and could even be unrelated. Peer-to-Peer organizations eliminate the potential for SPOF and bandwidth bottlenecks, thus typically exhibit good scalability and robust fault-tolerance. In contrary, making decisions in peer-to-peer organizations has to be carried out collectively using usually voting mechanisms. This typically implies increased implementation complexity as well as more communication overhead and latency, especially in large-scale systems such as the

cloud. As a specific example, GraphLab employs a peer-to-peer organization. Specifically, when GraphLab is launched on a cluster, one instance of its engine is started on each machine. All engine instances in GraphLab are symmetric. Moreover, they all communicate directly with each other using a customized asynchronous Remote Procedure Call (RPC) protocol over TCP/IP. The first triggered engine instance, however, will have an additional responsibility of being a monitoring/master engine. The other engine instances across machines will still work and communicate directly without having to be coordinated by the master engine. Consequently, GraphLab satisfies the criteria to be a peer-to-peer system.

1.5 Main Challenges In Building Cloud Programs

Designing and implementing a distributed program for the cloud involves more than just sending and receiving messages and deciding upon the computational and architectural models. While all these are extremely important, they do not reflect the whole story of developing programs for the cloud. In particular, there are various challenges that a designer needs to pay careful attention to and address before developing a cloud program. We next discuss **heterogeneity**, **scalability**, **communication**, **synchronization**, **fault-tolerance** and **scheduling** challenges exhibited in building cloud programs.

1.5.1 Heterogeneity

The cloud datacenters are composed of various collections of components including computers, networks, Operating Systems (OSs), libraries and programming languages. In principle, if there is variety and difference in datacenter 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. This is mainly due to two major reasons. First, cloud providers typically keep multiple generations of IT resources purchased over different time frames. Second, cloud providers are increasingly applying the virtualization technology on their clouds for server consolidation, enhanced system utilization and simplified management. Public clouds are primarily virtualized datacenters. Even on private clouds, it is expected that virtualized environments will become the norm [83]. Heterogeneity is a direct cause of virtualized environments. For example, co-locating virtual machines (VMs) on similar physical machines may cause heterogeneity. Specifically, if we suppose two identical physical machines **A** and **B**, placing 1 VM over machine **A** and 10 VMs over machine **B** will stress machine **B** way more than machine **A**, assuming all VMs are identical and running the same programs. Having dissimilar VMs and diverse demanding programs are even more probable on the cloud. An especially compelling setting is Amazon EC2. Amazon EC2 offers 17 VM types [1] (as of March 4, 2013) for millions of users with different programs. Clearly, this 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. First, distributed programs must be designed in a way that masks the heterogeneity of the underlying hardware, networks, OSs, and the programming languages. This is a necessity for distributed tasks to communicate, or otherwise, the whole concept of distributed programs will not hold (recall that what defines distributed programs is passing messages). To elaborate, messages exchanged between tasks would usually contain primitive data types such as integers. Unfortunately, not all computers store integers in the same order. In particular, some computers might use the so-called big-endian order, in which the most significant byte comes first, while others might 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 so as 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) [16] is an example of a 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.

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 such a problem. Same 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. Lastly, 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 problem that requires a great deal of attention from distributed programmers is **performance variation** [20, 60] on the cloud. Performance variation entails that running the same distributed program on the same cluster twice can result in largely different execution times. It has been observed that execution times can vary by a factor of 5 for the same application on the same private cluster [60]. Performance variation is mostly caused by the heterogeneity of clouds imposed by virtualized environments and resource demand spikes and lulls typically experienced

over time. As a consequence, VMs on clouds rarely carry work at the same speed, preventing thereby tasks from making progress at (roughly) constant rates. Clearly, this can create tricky load-imbalance and subsequently degrade overall performance. As pointed out earlier, load-imbalance makes a program's performance contingent on its slowest task. Distributed programs can attempt to tackle slow tasks by detecting them and scheduling corresponding *speculative* tasks on fast VMs so as they 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 getting killed. For instance, Hadoop MapReduce follows a similar strategy for solving the same problem, known as **speculative execution** (see Section 1.5.5). Unfortunately, distinguishing between slow and fast tasks/VMs is very 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 any detectably slow node is faulty and differentiating between faulty and slow nodes is hard [71]. Because of that, speculative execution in Hadoop MapReduce does not perform very well in heterogeneous environments [11, 26, 73].

1.5.2 Scalability

The issue of scalability is a dominant subject in distributed computing. A distributed program is said to be scalable if it remains effective when the quantities of users, data and resources are increased significantly. To get a sense of the problem scope at hand, as per users, in cloud computing, most popular applications and platforms are currently offered as Internet-based services with *millions* of users. As per data, in the time of Big Data, or *the Era of Tera* as denoted by Intel [13], distributed programs typically cope with Web-scale data in the order of hundreds and thousands of GBs, TBs or PBs. Also, Internet services such as e-commerce and social networks deal with sheer volumes of data generated by millions of users every day [83]. As per resources, cloud datacenters already host tens and hundreds of thousands of machines (e.g., Amazon EC2 is estimated to host almost half a million machines [46]), and projections for scaling up machine counts to extra folds have already been set forth.

As pointed out in Section 1.3, upon scaling up the number of machines, what programmers/users expect is escalated performance. Specifically, programmers expect from distributed execution of their programs on $\boldsymbol{n}$ nodes, versus on a single node, an $\boldsymbol{n}$-fold improvement in performance. Unfortunately, this never happens in reality due to several reasons. First, as shown in Fig. 1.13, parts of programs can never be parallelized (e.g., initialization parts). Second, load-imbalance among tasks is highly likely, especially in distributed systems like clouds. One of the reasons for load-imbalance is the heterogeneity of the cloud as discussed in the previous section. As depicted in Fig. 1.13 (b), load-imbalance usually delays programs, wherein a program becomes bound to the slowest task. Particularly, even if all tasks in a program finish, the program

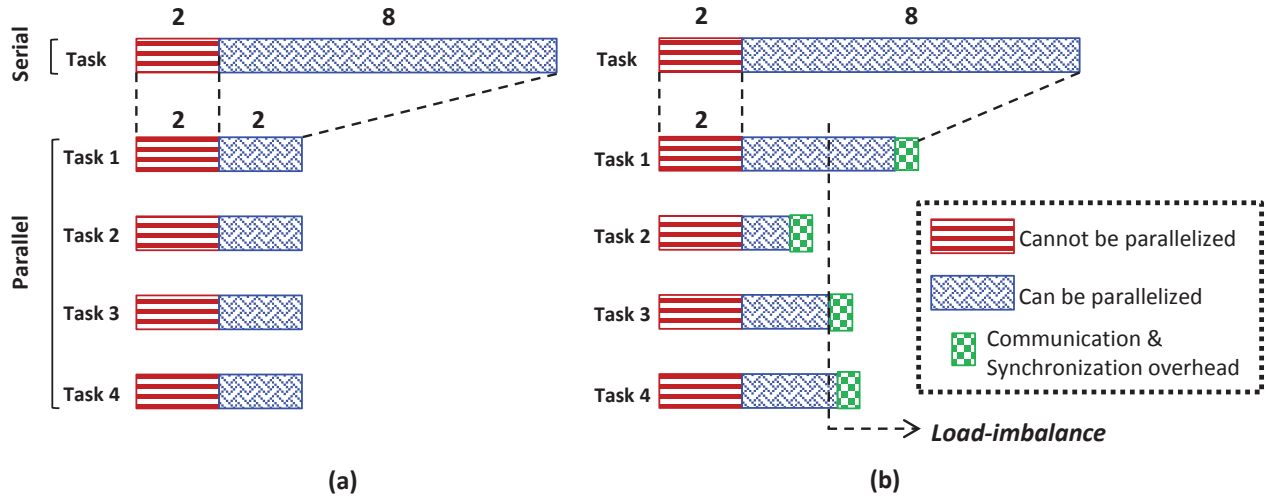

Figure 1.13: Parallel speedup. **(a)** Ideal case. **(b)** Real case.

cannot commit before the last task finishes (which might greatly linger!). Lastly, other serious overheads such as communication and synchronization can highly impede scalability. Such overheads are significantly important when measuring speedups obtained by distributed programs compared to sequential ones. A standard law that allows measuring speedups attained by distributed programs and, additionally, accounting for various overheads is known as **Amdahl's law**.

For the purpose of describing Amdahl's law we assume that a sequential version of a program ***P*** takes $\boldsymbol{T}_s$ time units, while a parallel/distributed version takes $\boldsymbol{T}_p$ time units using a cluster of ***n*** nodes. In addition, we suppose that ***s*** fraction of the program is not parallelizable. Clearly, this makes ***1 - s*** fraction of the program parallelizable. According to Amdahl's law, the speedup of the parallel/distributed execution of ***P*** versus the sequential one can be defined as follows:

$$Speedup_p = T_s/T_p = T_s/(T_s \times \text{s} + T_s \times \text{(1-s)/n}) = 1/(\text{s} + \text{(1-s)/n})$$

While the formula is apparently simple, it exhibits a crucial implication. In particular, if we assume a cluster with an unlimited number of machines and a constant ***s***, we can use the formula to express the maximum speedup that can be achieved by simply computing the $\lim_{n\to\infty} Speedup_p$ as follows:

$$\lim_{n\to\infty} Speedup_p = \lim_{n\to\infty} 1/(\text{s} + \text{(1-s)/n}) = 1/\text{s}$$

To understand the essentiality of the formula's implication, let us assume a serial fraction ***s*** of only 2%. Applying the formula with an assumingly 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 realize that attaining scalability in distributed systems is quite challenging, as it requires s to be almost 0, let alone 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 [67]. Communication overheads also grow dramatically since machines in large-scale distributed systems cannot be interconnected with very short physical distances. Load-imbalance becomes a big factor in heterogeneous environments as explained shortly. While this is truly challenging, we point out that with Web-scale input data, the overheads of synchronization and communication can be highly reduced if they contribute way less towards the overall execution time as compared to computation. Fortunately, this is the case with many Big Data applications.

1.5.3 Communication

As defined in Section 1.4.1, distributed systems are composed of networked computers that can communicate by explicitly passing messages or implicitly accessing shared memories. Even with distributed shared memory systems, messages are internally passed between machines, yet in a manner that is totally transparent to users. Hence, it all boils down essentially to passing messages. Consequently, it can be argued that the only way for distributed systems to communicate is by passing messages. In fact, Coulouris *et al.* [16] adopts such a 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, faster delivery of messages entails minimized execution times, reduced costs (as cloud applications can commit earlier), and higher QoS, especially for audio and video applications. This makes the issue of communication a principal theme in developing distributed programs for the cloud. Indeed, it will not be surprising if some people argue that communication is at the heart of the cloud and is one of its major bottlenecks.

Distributed programs can mainly apply two techniques to address the communication bottleneck on the cloud. First, *the strategy of distributing/partitioning the work across machines should attempt to co-locate highly communicating entities together*. This can mitigate the pressure on the cloud network and subsequently improve performance. Such an aspired goal is not as easy as it might appear, though. 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 Section 1.4.4). Unfortunately, by carefully inspecting such a strategy, we realize a serious shortcoming that directly impacts communication. To exemplify, Fig. 1.11 in Section 1.4.4 shows that the minimum cut resulted from the edge cut metric overlooks the fact that some edges may represent the same information flow. In particular, v_2 at P_1 in the figure sends the same message twice to P_2 (specifically to v_4 and v_5 at P_2), while it

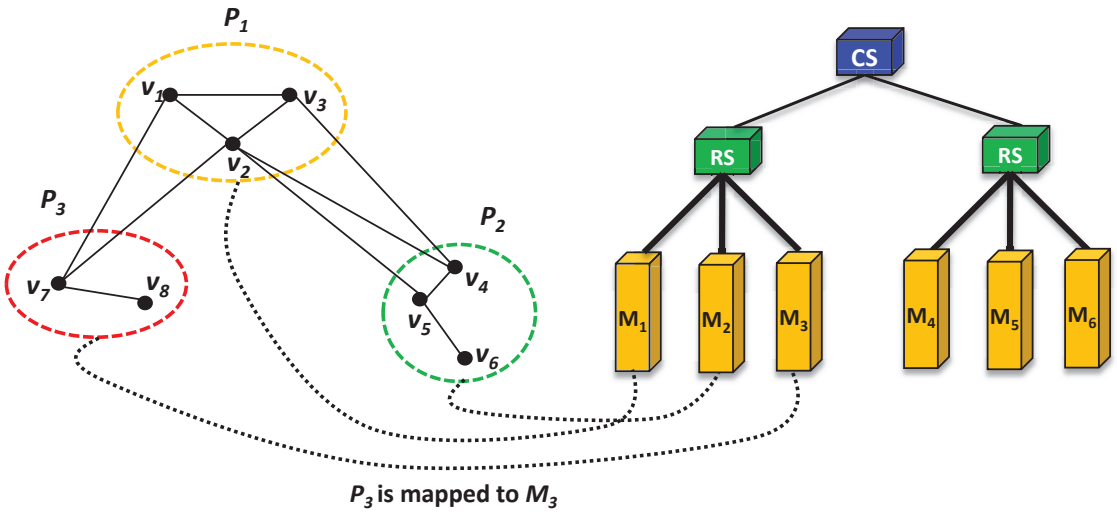

Figure 1.14: Effective mapping of graph partitions to cluster machines. A mapping of P_1 to the other rack while P_2 and P_3 remain on the same rack causes more network traffic and potentially degraded performance.

suffices to communicate the message only once, since $\boldsymbol{v}_4$ and $\boldsymbol{v}_5$ will exist on the same machine. Likewise, $\boldsymbol{v}_4$ and $\boldsymbol{v}_7$ can communicate messages to $\boldsymbol{P}_1$ only once but they do it twice. Therefore, the standard edge cut metric causes an over-count of the true volume of communication and consequently incurs superfluous network traffic. As an outcome, 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 render the bottleneck. In particular, it might happen that while the communication volume is minimized, some machines receive heavier partitions (i.e., partitions with more vertices) than others. An ideal, yet a challenging approach, is to minimize communication overheads while circumventing computation skew among machines. To summarize, this technique strives for *effective partitioning of work across machines so as highly communicating entities are co-located together*.

The second technique is *effective mapping of partitions*. Specifically, the mapping strategy of partitions to machines, whether graph or data partitions, should be done in a way that is totally aware of the underlying network topology. This dictates the number of switches that a message will hit before it reaches its destination. As a specific example, Fig. 1.14 demonstrates the same graph shown previously in Fig. 1.11 and a simplified cluster with a tree-style network and 6 machines. The cluster network consists of two rack switches (RSs), each connecting 3 machines, and a core switch (CS) connecting the two RSs. A salient point is that the bandwidth between two machines is dependent on their relative locations in the network topology. For instance, machines that are on the same rack have higher bandwidth between them as opposed to machines that are off-rack. As such, it pays to minimize network traffic across racks. If $\boldsymbol{P}_1$, $\boldsymbol{P}_2$ and $\boldsymbol{P}_3$ are mapped to $\boldsymbol{M}_1$, $\boldsymbol{M}_2$ and $\boldsymbol{M}_3$, respectively, less network latency will be incurred when $\boldsymbol{P}_1$, $\boldsymbol{P}_2$ and $\boldsymbol{P}_3$ communicate versus if they are mapped across the two racks. More precisely, for $\boldsymbol{P}_1$ to communicate with $\boldsymbol{P}_2$ on the same rack, only one hop is incurred to route a message from $\boldsymbol{P}_1$

to $\boldsymbol{P}_2$. In contrast, for $\boldsymbol{P}_1$ to communicate with $\boldsymbol{P}_2$ on different racks, two hops are incurred per each message. Clearly, a less number of hops results in a better network latency and improved overall performance. Unfortunately, this is not as easy as it might appear to achieve on clouds, especially on public clouds, for one main reason. That is, clouds such as Amazon EC2 do not expose their network topologies. Nevertheless, the network topology can still be learned (though not very effectively) using a benchmark like Netperf [54] to measure point-to-point TCP stream bandwidths between all pairs of cluster nodes [32]. This enables estimating the relative locality of nodes and arriving at a reasonable inference regarding the rack topology of the cluster.

1.5.4 Synchronization

Distributed tasks should be allowed to simultaneously operate on shared data without corrupting data or causing any inconsistency. For instance, GraphLab allows multiple tasks to operate on different vertices of the same graph simultaneously. This might lead to race-conditions whereby two tasks might try to modify data on a shared edge at the same time, resulting in a corrupted value. Consequently, *synchronization* solutions for providing distributed *mutual exclusive accesses* by tasks will be required. Synchronization acts as a mechanism through which programmers can control the sequence of operations (reads and writes) that are performed by tasks. As discussed in Section 1.4.2.1, there are three types of synchronization methods that are in wide use, semaphores, locks and barriers. The efficiency of such methods is a critical goal in developing distributed programs. For instance, as pointed out in Section 1.4.2.1 and exemplified by the BSP model (see Section 1.4.3), a barrier defines a point at which no task is allowed to continue unless all other tasks reach that point. While this is easy to implement, the whole execution time of a distributed program becomes dependent on the slowest task. In distributed systems such as the cloud, where heterogeneity is the norm, this can cause serious performance degradation. The challenge becomes how to apply synchronization methods and at the same time avert performance degradation.

In addition to ensuring mutual exclusion, there are other properties that need to be guaranteed for distributed programs when it comes to synchronization. To start with, if one task shows interest in getting access to a critical section, eventually it should succeed. If two tasks show interest in getting access to a critical section simultaneously, one of them should only succeed. This is denoted as the ***deadlock-free*** property and has to be delivered by any mutual exclusion mechanism. Things, however, might not go always as expected. For instance, if task $\boldsymbol{A}$ succeeds in acquiring lock1 and, at about the same time, task $\boldsymbol{B}$ succeeds in acquiring lock2; then if task $\boldsymbol{A}$ attempts to acquire lock2 and task $\boldsymbol{B}$ attempts to acquire lock1, we end up with what is known as a ***deadlock***. Avoiding deadlocks is a real challenge in developing distributed programs, especially when the number of tasks is scaled up. To build upon the example of tasks $\boldsymbol{A}$ and $\boldsymbol{B}$, 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. Circular waits lead normally to stark deadlocks that might bring the whole distributed programs/systems to grinding halts. Lastly, we note that the "wait on" relation is at the heart of every mutual exclusion mechanism. In particular, no mutual exclusion protocol can preclude it; no matter how clever it is [36]. 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 suggests another major challenge that distributed programs need to address, that is, ***fault-tolerance***.

1.5.5 Fault-tolerance

A basic feature that distinguishes distributed systems such as the cloud from uniprocessor systems is the concept of **partial failures**. Specifically, in distributed systems if a node or component fails, the whole system can continue functioning. On the other hand, if one component (e.g., the RAM) fails in a uniprocessor system, the whole system will fail. A crucial objective in designing distributed systems/programs is to construct them in a way that they can automatically tolerate partial failures without seriously affecting performance. A key technique for masking faults in distributed systems is to use hardware redundancy such as the RAID technology [56]. In most cases, however, distributed programs cannot only depend on the underlying hardware fault-tolerance techniques of distributed systems. Thus, they usually apply their own fault-tolerance techniques. Among these techniques is **software redundancy**.

A common type of software redundancy is **task redundancy** (or **resiliency** or **replication**). Task replication is applied as a protection against task failures. In particular, tasks can be replicated as **flat** or **hierarchical groups**, exemplified in Fig. 1.15. In flat groups (see Fig. 1.15 (a)), all tasks are identical in a sense that they all carry the same work. Eventually, only the result of one task is considered and the other results are discarded. Obviously, flat groups are symmetrical and preclude single point of failures (SPOFs). Particularly, 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 might be required. As discussed earlier, voting mechanisms incur implementation complexity, communication delays and performance overheads.

A hierarchical group (see Fig. 1.15 (b)) usually employs a coordinator task and specifies the

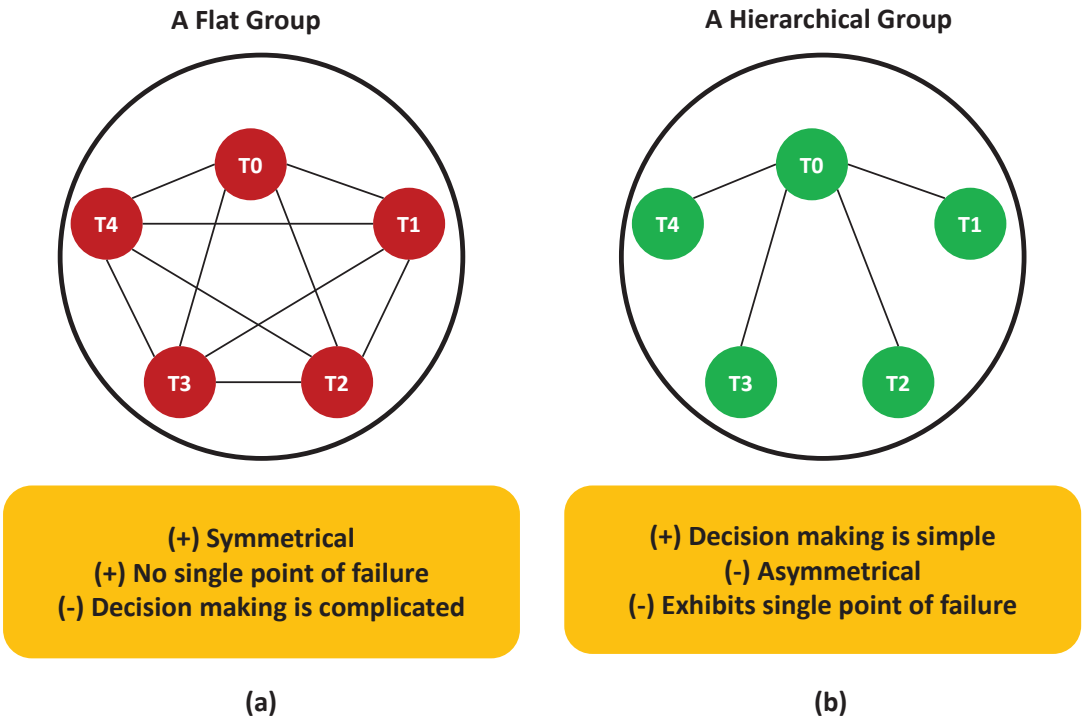

Figure 1.15: Two classical ways to employ task redundancy. (a) A flat group of tasks. (b) A hierarchical group of tasks with a central process (i.e., T0, whereby **T*i*** stands for task ***i***.).

rest of the tasks as workers. In this model, when a user request is made, it gets first forwarded to the coordinator who, in return, decides which worker is best suited to fulfill it. Clearly, hierarchical groups reflect opposite properties as compared to flat ones. 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.

As a real example, Hadoop MapReduce applies task resiliency to recover from task failures and mitigate the effects of slow tasks. Specifically, Hadoop MapReduce suggests monitoring and *replicating* tasks in an attempt to detect and treat slow/faulty ones. To detect slow/faulty tasks, Hadoop MapReduce depends on what is denoted as the **heartbeat mechanism**. As pointed out earlier, MapReduce adopts a master/slave architecture. Slaves (or TaskTrackers) send their heartbeats every 3 seconds (by default) to the master (or the JobTracker). The JobTracker employs an *expiry thread* that checks these heartbeats and decides whether tasks at TaskTrackers are *dead* or *alive*. If the expiry thread does not receive heartbeats from a task in 10 minutes (by default), the task is deemed dead. Otherwise, the task is marked alive. Alive tasks can be slow (referred interchangeably to as **stragglers**) or *not-slow*. To measure the slowness of tasks, the JobTracker calculates task progresses using a *progress score* per each task between 0 and 1. The progress scores of Map and Reduce tasks are computed differently. For a Map task, the progress score is a function of the input HDFS block read so far. For a Reduce task the progress score is more

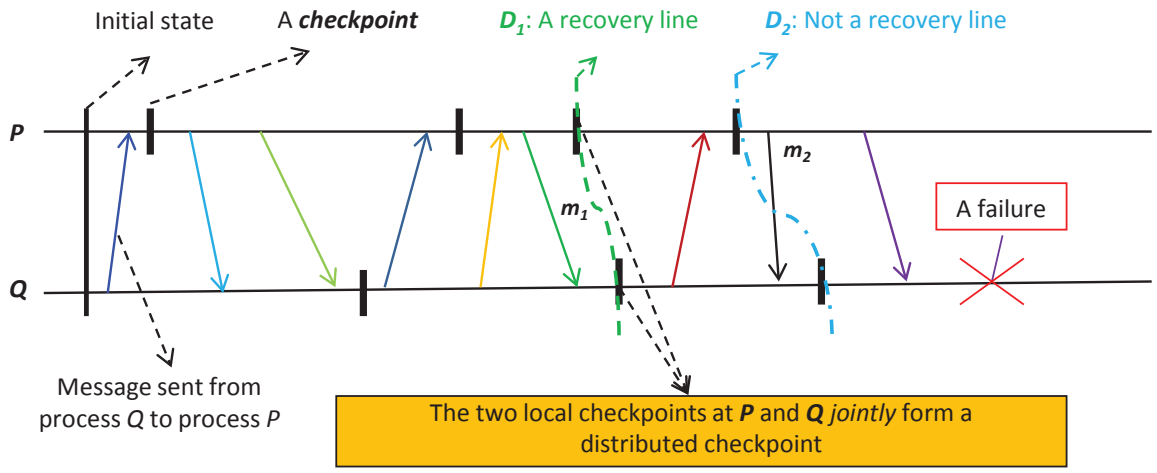

Figure 1.16: Demonstrating distributed checkpointing. D_1 is a valid distributed checkpoint while D_2 is not due to being inconsistent. Specifically, D_2's checkpoint at ***Q*** indicates that m_2 has been received, while D_2's checkpoint at ***P*** does not indicate that m_2 has been sent.

involved. To elaborate, the execution of a Reduce task is split into three main stages, the *Shuffle*, the *Merge & Sort*, and the *Reduce* stages. Hadoop MapReduce assumes that each of these stages accounts for 1/3 of a Reduce task's score. Per each stage, the score is the fraction of data processed so far. For instance, a Reduce task halfway through the Shuffle stage will have a progress score of 1/3 × 1/2 = 1/6. On the other hand, a Reduce task halfway through the Merge & Sort stage will have a progress score of 1/3 + (1/2 × 1/3) = 1/2. Finally, a Reduce task halfway through the Reduce stage will have a progress score of 1/3 + 1/3 + (1/3 × 1/2) = 5/6. After slow tasks are detected, corresponding backup (or *speculative*) tasks are run simultaneously. Hadoop allows a maximum of one speculative task per an original slow task. The speculative tasks compete with the original ones and optimistically finish earlier. The tasks that finish earlier, being speculative or original, are committed while the others are killed. This type of task resiliency is known in Hadoop MapReduce as **speculative execution**. Speculative execution is turned on by default in Hadoop, but can be enabled or disabled independently for Map and Reduce tasks, on a cluster-wide basis or on a per-job basis. To this end, we indicate two points: (1) speculative tasks are identical due to running the same function on the same data, and (2) no coordinator task is specified among the speculative tasks, yet a coordinator is still used, but not from among the tasks (i.e., the JobTracker). As such, we refer to the strategy adopted by Hadoop MapReduce as a hybrid of flat and hierarchical task groups.

Fault tolerance in distributed programs is not only concerned with tolerating faults, but further with recovering from failures. The basic idea of **failure recovery** is to replace a flawed state with a flaw-free state. One way to achieve this is by using backward-recovery. Backward-recovery requires that the distributed program/system is brought from its present flawed state to a previously correct state. This can be accomplished by recording the state at each process from time to time. Once a failure occurs, recovery can be started from the last recorded correct state, typically

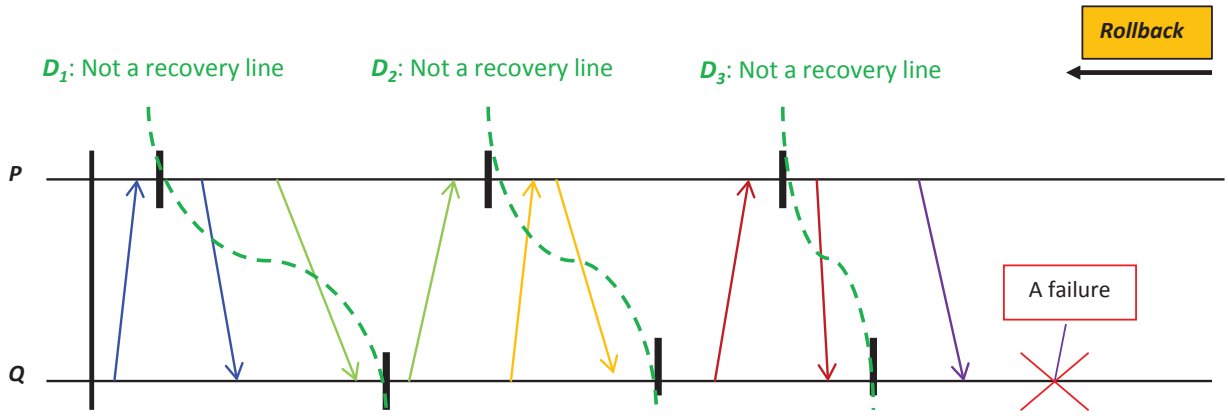

Figure 1.17: The domino effect that might result from rolling back each process (e.g., processes ***P*** and ***Q***) to a saved local checkpoint in order to locate a recovery line. Neither D_1, D_2 nor D_3 are recovery lines because they exhibit inconsistent global states.

denoted as the recovery line. Every time the state is recorded at a process, a checkpoint is said to be obtained. The checkpoints of a distributed program at different processes in a distributed system are known as a **distributed checkpoint**. The process of capturing a distributed checkpoint is not easy for one main reason. Specifically, a distributed checkpoint needs to maintain a *consistent global state*. More precisely, a distributed checkpoint 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*** should have been sent from somewhere. Fig. 1.16 demonstrates two distributed checkpoints, $\boldsymbol{D}_1$, which maintains a consistent global state and $\boldsymbol{D}_2$, which does not maintain a consistent global state. The $\boldsymbol{D}_1$'s checkpoint at ***Q*** indicates that ***Q*** has received a message $\boldsymbol{m}_1$, and the $\boldsymbol{D}_1$'s checkpoint at ***P*** indicates that ***P*** has sent $\boldsymbol{m}_1$, hence, making $\boldsymbol{D}_1$ consistent. In contrary, the $\boldsymbol{D}_2$'s checkpoint at ***Q*** indicates that message $\boldsymbol{m}_2$ has been received, and the $\boldsymbol{D}_2$'s checkpoint at ***P*** does not indicate that $\boldsymbol{m}_2$ has been sent from ***P***. Therefore, $\boldsymbol{D}_2$ cannot be treated as a recovery line due to being inconsistent.

The distributed program/system can inspect whether a certain distributed checkpoint is consistent or not by *rolling back* each process to its most recently saved state. 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 Fig. 1.16 will roll back until hitting $\boldsymbol{D}_1$. As $\boldsymbol{D}_1$ reflects a global consistent state, a recovery line is said to be discovered. Unfortunately, the process of cascaded rollbacks is challenging since it might lead to what is called the **domino effect**. As a specific example, Fig. 1.17 exhibits a case where a recovery line cannot be found. In particular, every distributed checkpoint in Fig. 1.17 is indeed inconsistent. This makes distributed checkpointing a costly operation which might not converge to an acceptable recovery solution. Accordingly, many fault-tolerant distributed systems combine checkpointing with **message logging**. One way to achieve that is by logging messages of a process before sending them off and after a checkpoint has been taken. Obviously, this solves the problem of $\boldsymbol{D}_2$ at Fig. 1.16, for example. In particular,

after the $\boldsymbol{D}_2$'s checkpoint at $\boldsymbol{P}$ is taken, the send of $\boldsymbol{m}_2$ will be marked in a log message at $\boldsymbol{P}$, which if merged with $\boldsymbol{D}_2$'s checkpoint at $\boldsymbol{Q}$, can form a global consistent state. Apart from distributed analytics engines, Hadoop Distributed File System (HDFS) combines distributed checkpointing (i.e., the ***image file***) and message logging (i.e., the ***edit file***) to recover slave (or NameNode in HDFS's parlance) failures [27]. As examples from analytics engines, distributed checkpointing alone is adopted by GraphLab, while message logging and distributed checkpointing combined together are employed by Pregel.

1.5.6 Scheduling

The effectiveness of a distributed program hinges on the manner in which its constituent tasks are scheduled over distributed machines. Scheduling in distributed programs is usually categorized into two main classes, **task scheduling** and **job scheduling**. To start with, as defined in Section 1.2, a job can encompass one or many tasks. Tasks are the finest unit of granularity for execution. Many jobs from many users can be submitted simultaneously for execution on a cluster. Job schedulers decide on which job should go next. For instance, Hadoop MapReduce adopts a *First In, First Out* (FIFO) job scheduler, whereby jobs are run according to the order of which they have been received. With FIFO, *running* jobs cannot be preempted so as to allow *waiting* jobs to proceed and, consequently, achieve certain objectives (e.g., avoiding job starvations and/or sharing resources effectively). As a result, simultaneous sharing of cluster resources is greatly limited. In particular, as long as a job occupies the *whole* cluster resources, no other job is allowed to carry on. That is, the next job in the FIFO work queue is not allowed to start up unless some resources become free and the currently running job has no more tasks to execute. Clearly, this might lead to a fairness issue wherein a very long job can block the whole cluster for a very long time starving all small jobs. Hadoop MapReduce, however, employs multiple other job schedulers besides FIFO (e.g., Capacity [28] and Fair [29] schedulers). After a job is granted the cluster, the decision morphs into how the job's tasks should be scheduled. Tasks can be scheduled either close to the data that they are supposed to process or *anywhere*. When tasks are scheduled nearby their data, ***locality*** is said to be exploited. For example, Hadoop MapReduce schedules Map tasks in the vicinity of their uniform-sized input HDFS blocks and Reduce tasks at *any* cluster nodes, irrespective of the locations of their input data. As opposed to MapReduce, Pregel and GraphLab do not exploit any locality when scheduling tasks/vertices.

Task schedulers must account for the heterogeneity of the underlying cloud or, otherwise, performance might degrade significantly. To elaborate, similar tasks that belong to the same job can be scheduled at nodes of variant speeds in a heterogeneous cloud. As reiterated in Sections 1.4.3 and 1.5.1, this can create load-imbalance and make jobs move at the pace of their slowest tasks. Strategies such as speculative execution in Hadoop MapReduce can (minimally) address such

a challenge (see Section 1.5.5). In addition to considering heterogeneity, task schedulers must seek to enhance system utilization and improve task parallelism. Specifically, tasks should be uniformly distributed across cluster machines in a way that fairly utilizes the available cluster resources and effectively increases parallelism. Obviously, this presents some contradictory objectives. To begin with, by evenly distributing tasks across cluster machines, locality might be affected. For instance, machines in a Hadoop cluster can contain different numbers of HDFS blocks. If at one machine, a larger number of HDFS blocks exist as opposed to others, locality would entail scheduling all respective Map tasks at that machine. This might make other machines less loaded and utilized. In addition, this can reduce task parallelism as a consequence of accumulating many tasks on the same machine. If locality is relaxed a little bit, however, utilization can be enhanced, loads across machines can be balanced and task parallelism can be increased. Nonetheless, this would necessitate moving data towards tasks, which if done injudiciously, might increase communication overhead, impede scalability and potentially degrade performance. In fact, with datacenters hosting thousands of machines, moving data frequently towards distant tasks might become one of the major bottlenecks. As such, an optimal task scheduler would strike a balance between system utilization, load balancing, task parallelism, communication overhead and scalability so as performance is improved and costs are reduced. Unfortunately, in practice, this is very hard to accomplish. In reality, most task schedulers attempt to optimize one objective and overlook the others.

Another major challenge when scheduling jobs and tasks is to meet what is known as **Service-Level Objectives** (SLOs). SLOs reflect the performance expectations of end-users. Amazon, Google and Microsoft have identified SLO violations as a major cause of user dissatisfaction [31, 64, 79]. For example, SLO can be expressed as a maximum latency for allocating the desired set of resources to a job, a soft/hard deadline to finish a job, or GPU preferences of some tasks, among others. In multi-tenant heterogeneous clusters, SLOs are hard to achieve, especially upon the arrival of new jobs while others are executing. This might require *suspending* currently running tasks in order to allow the newly arrived ones to proceed and, subsequently, meet their specified SLOs. The capability of suspending and resuming tasks is referred to as **task elasticity**. Unfortunately, most distributed analytics engines at the moment; including Hadoop MapReduce, Pregel and GraphLab do not support task elasticity. Making tasks elastic is quite challenging. It demands identifying *safe* points where a task can be suspended. A safe point in a task is a point at which the correctness of the task is not affected, and its committed work is not *all* repeated when it is suspended then resumed. In summary, meeting SLOs, enhancing system utilization, balancing load, increasing parallelism, reducing communication traffic and facilitating scalability are among the objectives that make job and task scheduling one of the major challenges in

developing distributed programs for the cloud.

1.6 Summary

To this end, we conclude our discussion on distributed programming for the cloud. As a recap, we commenced our treatment for the topic with a brief background on the theory of distributed programming. Specifically, we categorized programs into sequential, parallel, concurrent and distributed programs, and recognized the difference between processes, threads, tasks and jobs. Second, we motivated the case for distributed programming and explained why cloud programs (a special type of distributed programs) are important for solving complex computing problems. Third, we defined distributed systems and indicated the relationship between distributed systems and clouds. Fourth, we delved into the details of the models that cloud programs can adopt. In particular, we presented the distributed programming (i.e., shared-memory or message-passing), the computation (i.e., synchronous or asynchronous), the parallelism (i.e., graph-parallel or data-parallel), and the architectural (i.e., master/slave or peer-to-peer) models in detail. Lastly, we discussed the challenges with heterogeneity, scalability, communication, synchronization, fault-tolerance and scheduling which are encountered when constructing cloud programs.

Throughout our discussion on distributed programming for the cloud, we also indicated that it is extremely advantageous to relieve programmers from worrying about specifying and implementing the distributed programming model, the computation model, the architectural model, the graph or data partitioning algorithm (e.g., file striping or edge cut metric), the fault-tolerance mechanism (i.e., task resiliency and/or distributed checkpointing and/or message logging), the low level synchronization methods (e.g., semaphores, locks and/or barriers) and properties (e.g., avoiding deadlocks and transitive closures), and the job and task scheduling algorithms (e.g., FIFO, locality-aware and elastic schedulers). Designing, developing, verifying and debugging all (or some of) these requirements might induce tremendous correctness and performance hurdles, let alone the time and resources that need to be invested and the inherent difficulty involved. As a consequence, we presented Hadoop MapReduce, Pregel and GraphLab as being easy-to-use, effective and popular distributed analytics engines for building cloud programs.

Table 1.2: A Comparison Between the Hadoop MapReduce, the Pregel and the GraphLab Analytics Engines.

Aspect	Hadoop MapReduce	Pregel	GraphLab
Programming Model	Shared-Memory	Message-Passing	Shared-Memory
Computation Model	Synchronous	Bulk-Synchronous	Asynchronous
Parallelism Model	Data-Parallel	Graph-Parallel	Graph-Parallel
Architectural Model	Master/Slave	Master/Slave	Peer-To-Peer
Task/Vertex Scheduling Model	Pull-Based	Push-Based	Push-Based

Hadoop MapReduce, Pregel and GraphLab were created to remove the burdens of creating

and managing distributed programs from the shoulders of users/developers and delegate all that to the analytics engines themselves. Specifically, such engines eliminate the need for users to: (1) design and implement the programming model and overcome all its associated synchronization and consistency issues, (2) develop the computation model, (3) specify the parallelism model and encode the accompanying partitioning and mapping algorithms, (4) architect the underlying organization (i.e., master-slave or peer-to-peer), and (5) apply the task/vertex scheduling strategy (i.e., push-based or pull-based). With respect to all these aspects, Table 1.2 compares Hadoop MapReduce, Pregel and GraphLab. In short, Hadoop MapReduce is regarded as a data-parallel engine, while Pregel and GraphLab are characterized as graph-parallel engines. Furthermore, while both MapReduce and Pregel suggest synchronous computation models, GraphLab promotes an asynchronous model. MapReduce suits more loosely-connected/embarrassingly parallel applications (i.e., applications with no or little dependency/communication between the parallel tasks), which involve vast volumes of data and non-iterative computations. On the other hand, Pregel and GraphLab suit more strongly-connected applications (i.e., applications with high degrees of dependency between parallel tasks/vertices), which involve iterative computations and little data per a task/vertex. Asynchronous computation can lead to theoretical and empirical gains in algorithms and system performance. Thus, for graph and iterative oriented applications that do not converge synchronously (e.g., greedy graph coloring) or converge way faster when executed asynchronously (e.g., dynamic PageRank), GraphLab becomes a superior option as opposed to Pregel. Finally, for graph and iterative oriented applications that converge faster if serializability (which entails that for every parallel/distributed execution, there is an equivalent sequential execution) is ensured or even necessitate serializability for correctness (e.g., Gibbs sampling, a very common algorithm in Machine Learning and Data Mining problems), GraphLab remains a supreme alternative versus Pregel.

References

[1] "Amazon Elastic Compute Cloud, http://aws.amazon.com/ec2/."

[2] "Amazon Elastic MapReduce, http://aws.amazon.com/elasticmapreduce/."

[3] "Amazon Simple Storage Service, http://aws.amazon.com/s3/."

[4] Amazon, "Amazon Web Services: Overview of Security Processes," *Amazon Whitepaper,* May 2011.

[5] M. Bailey, "The Economics of Virtualization: Moving Toward an Application-Based Cost Model," *VMware Sponsored Whitepaper,* 2009.

[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt and A. Warfield, "Xen and the Art of Virtualization," *SOSP,* Oct. 2003.

[7] J. R. Bell, "Threaded Code," *Communications of the ACM,* 1973.

[8] M. Ben-Ari, "Principles of Concurrent and Distributed Programming," *Addison-Wesley, Second Edition,* March 6, 2006.

[9] D. P. Bertsekas and J. N. Tsitsiklis, "Parallel and Distributed Computation: Numerical Methods," *Athena Scientific; First Edition,* January 1, 1997.

[10] C. Boulton, "Novell, Microsoft Outline Virtual Collaboration," *Serverwatch,* 2007.

[11] T. D. Braun, H. J. Siegel, N. Beck, L. L. Blni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, and R. F. Freund, "A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems," *JPDC,* June 2001.

[12] P. M. Chen and B. D. Nobel, "When Virtual Is Better Than Real," *HOTOS,* May 2001.

[13] S. Chen and S. W. Schlosser, "MapReduce Meets Wider Varieties of Applications," *IRP-TR-08-05, Intel Research,* 2008.

[14] D. Chisnall, "The Definitive Guide to the Xen Hypervisor," *Prentice Hall, 1st Edition* Nov. 2007.

[15] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, "Introduction to Algorithms," *The MIT Press, Third Edition* July 31, 2009.

[16] G. Coulouris, J. Dollimore, T. Kindberg and G. Blair, "Distributed Systems: Concepts and Design," *Addison-Wesley, 5 Edition* May 2011.

[17] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing On Large Clusters," *OSDI,* Dec. 2004.

[18] R. B. K. Dewar, "Indirect Threaded Code," *Communications of the ACM,* June 1975.

[19] T. W. Doeppner, "Operating Systems In Depth: Design and Programming," *Wiley, 1st Edition* Nov. 2010.

[20] B. Farley, V. Varadarajan, K. Bowers, A. Juels, T. Ristenpart and M. Swift "More for Your Money: Exploiting Performance Heterogeneity in Public Clouds," *SOCC,* 2012.

[21] M. R. Garey, D. S. Johnson and L. Stockmeyer, "Some simplified¡ i¿ NP¡/i¿-complete graph problems," *Theoretical Computer Science,* 1976.

[22] S. Ghemawat, H. Gobioff, and S. T. Leung, "The Google File System," *SOSP,* Oct. 2003.

[23] D. K. Gifford, "Weighted Voting for Replicated Data," *In Proceedings of the Seventh ACM Symposium on Operating Systems Principles,* December 1979.

[24] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs," *OSDI,* 2012.

[25] M. Gschwind, E. R. Altman, S. Sathaye, P. Ledak and D. Appenzeller, "Dynamic and Transparent Binary Translation," *IEEE Computer,* 2000.

[26] Z. Guo and G. Fox, "Improving MapReduce Performance in Heterogeneous Network Environments and Resource Utilization," *In Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID 2012),* May 2012.

[27] "Hadoop. http://hadoop.apache.org/."

[28] "Hadoop Capacity Scheduler. http://hadoop.apache.org/docs/stable/capacity_scheduler.html."

[29] "Hadoop Fair Scheduler. http://hadoop.apache.org/docs/r1.1.2/fair_scheduler.html."

[30] "Hadoop Tutorial. http://developer.yahoo.com/hadoop/tutorial/."

[31] J. Hamilton, "The Cost of Latency," *http://perspectives.mvdirona.com/2009/10/31/TheCostOfLatency.aspx,*

[32] M. Hammoud, M.S. Rehman and M.F. Sakr, "Center-of-Gravity Reduce Task Scheduling to Lower MapReduce Network Traffic," *CLOUD,* 2012.

[33] M. Hammoud and M.F. Sakr, "Locality-Aware Reduce Task Scheduling for MapReduce," *CloudComm,* 2011.

[34] B. Hendrickson and T. G. Kolda, "Graph Partitioning Models for Parallel Computing," *Parallel Computing,* 2000.

[35] B. Hendrickson and R. Leland, "The Chaco User's Guide Version 2.0," *Technical Report SAND95-2344, Sandia National Laboratories,* 1995.

[36] M. Herlihy and N. Shavit, "The Art of Multiprocessor Programming," *Morgan Kaufmann, First Edition,* March 14, 2008.

[37] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F.B. Cetin, and S. Babu, "Starfish: A Self-Tuning System for Big Data Analytics," *CIDR,* 2011.

[38] S. Ibrahim, H. Jin, L. Lu, S. Wu, B. He, and L. Qi, "LEEN: Locality/Fairness-Aware Key Partitioning for MapReduce in the Cloud," *CloudComm,* Dec. 2010.

[39] G. Karypis and V. Kumar, "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs," *SIAM Journal on Scientific Computing,* 1998.

[40] P. Klint, "Interpretation Techniques," *Software Practice and Experience,* 1981.

[41] P. M. Kogge, "An Architecture Trail to Threaded-Code Systems," *IEEE Computer,* March 1982.

[42] A. N. Langville and C. D. Meyer, "Google's PageRank and Beyond: The Science of Search Engine Rankings," *Princeton University Press,* February 6, 2012.

[43] "Learn About Java Technology, http://www.java.com/en/about/."

[44] K. Li, "Shared Virtual Memory on Loosely Coupled Multiprocessors," *Yale University, New Haven, CT (USA)* 1986.

[45] K. Li and P. Hudak, "Memory Coherence in Shared Virtual Memory Systems," *Transactions on Computer Systems (TOCS)* 1989.

[46] H. Liu, "Amazon Data Center Size," *http://huanliu.wordpress.com/2012/03/13/amazon-data-center-size/* March 2012.

[47] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola and J. M. Hellerstein, "Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud," *Proceedings of the VLDB Endowment* 2012.

[48] P. Magnusson and D. Samuelsson, "A Compact Intermediate Format for SIMICS," *Swedish Institute of Computer Science, Tech. Repoart R94:17* Sep. 1994.

[49] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser and G. Czajkowski, "Pregel: A System for Large-Scale Graph Processing," *In Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* June 2010.

[50] "Message Passing Interface, http://www.mcs.anl.gov/research/projects/mpi/."

[51] Microsoft Corporation, "ECMA C# and Common Language Infrastructure Standards," Oct. 2009.

[52] "MPICH, http://www.mpich.org/."

[53] A. C. Murthy, C. Douglas, M. Konar, O. O'Malley, S. Radia, S. Agarwal, and K. V. Vinod, "Architecture of Next Generation Apache Hadoop MapReduce Framework," *Apache Jira* 2011.

[54] Netperf, "http://www.netperf.org/."

[55] "OpenMP, http://openmp.org/wp/."

[56] D. Patterson, G. Gibson, and R. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," *ACM, volume 17,* 1988.

[57] I. Pratt, K. Fraser, S. Hand, Limpach, A. Warfield, Magenheimer, Nakajima and Mallick, "Xen 3.0 and the Art of Virtualization," *Proceedings of the Linux Symposium (Volume Two),* July 2005.

[58] G. J. Popek and R. P. Goldberg, "Formal Requirements for Virtualizable Third Generation Architectures," *Communications of the ACM,* July 1974.

[59] "POSIX Threads Programming, https://computing.llnl.gov/tutorials/pthreads/."

[60] M. S. Rehman and M. F. Sakr, "Initial Findings for Provisioning Variation in Cloud Computing," *CloudCom,* Nov. 2010.

[61] "Java RMI, http://www.oracle.com/technetwork/java/javase/tech/index-jsp-138781.html."

[62] T. H. Romer, D. Lee, G. M. Voelker, A. Wolman, W. A. Wong, J.-L. Baer, B. N. Bershad, and H. M. Levy, "The Structure and Performance of Interpreters," *ASPLOS,* 1996.

[63] N.B. Rizvandi, A.Y. Zomaya, A.J. Boloori, and J. Taheri, "Preliminary Results: Modeling Relation between Total Execution Time of MapReduce Applications and Number of Mappers/Reducers," *Tech. R. 679, The University of Sydney,* 2011.

[64] E. Schurman and J. Brutlag, "The User and Business Impact of Server Delays, Additional Bytes, and Http Chunking in Web Search," *Velocity Conference,* 2009.

[65] R. L. Sites, A. Chernoff, M. B. Kirk, M. P. Marks, and S. G. Robinson, "Binary Translation," *Communications of the ACM,* Feb. 1993.

[66] J. E. Smith and R. Nair, "Virtual Machines: Versatile Platforms for Systems and Processes," *Morgan Kaufmann,* 2005.

[67] Y. Solihin, "Fundamentals of Parallel Computer Architecture," *Solihin Books,* 2009.

[68] S. Soltesz, H. Potz, M. E. Fiuczynski, A. Bavier and L. Peterson, "Container-Based Operating System Virtualization: a Scalable, High-Performance Alternative to Hypervisors," *EuroSys,* March 2007.

[69] A. S. Tanenbaum, "Modern Operating Systems," *Prentice Hall, Third Edition ,* December 21, 2007.

[70] A. S. Tanenbaum, "Distributed Operating Systems," *Prentice Hall, First Edition,* September 4, 1994.

[71] A. S. Tanenbaum and M. V. Steen, "Distributed Systems: Principles and Paradigms," *Prentice Hall, Second Edition,* October 12, 2006.

[72] R. H. Thomas, "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases," *ACM Transactions on Database Systems (TODS),* 1979.

[73] M. Tsugawa and J. A. B. Fortes, "A Virtual Network (ViNe) Architecture for Grid Computing," *IPDPS'06,* 2006.

[74] L. G. Valiant, "A Bridging Model for Parallel Computation," *Communications of the ACM,* 1990.

[75] "VMWare, http://www.vmware.com."

[76] VMWare, "Understanding Full Virtualization, Paravirtualization, and Hardware Assist," *VMware Whitepaper,* Nov. 2007.

[77] VMWare, "VMware vSphere: The CPU Scheduler in VMware ESX 4.1," *VMware Whitepaper,* 2010.

[78] VMWare, "Understanding Memory Resource Management in VMware ESX Server," *VMware Whitepaper,* 2009.

[79] A. Wang, S. Venkataraman, S. Alspaugh, R. Katz, I. Stoica, "Cake: Enabling High-level SLOs on Shared Storage Systems," *SOCC,* 2012.

[80] "Xen Open Source Community, http://www.xen.org."

[81] "Xen 4.0 Release Notes, http://wiki.xen.org/wiki/Xen_4.0_Release_Notes."

[82] "Xen 4.1 Release Notes, http://wiki.xen.org/wiki/Xen_4.1_Release_Notes."

[83] M. Zaharia, A. Konwinski, A. Joseph, R. Katz, I. Stoica, "Improving Mapreduce Performance in Heterogeneous Environments," *OSDI,* 2008.

[84] F. Zhou, M. Goel, P. Desnoyers and R. Sundaram, "Scheduler Vulnerabilities and Attacks in Cloud Computing,"

arXiv:1103.0759v1 [cs.DC], March 2011.