# 1 Mathematics, Models and Architectures

Bill MCCOLL

Overview

Models of computation are at the heart of computer science. The notion of what it means to be computable can be precisely and mathematically defined in terms of the Turing machine model, first defined by Alan Turing in 1936. The architecture of sequential computers is based on the von Neumann model, first proposed by John von Neumann in 1945. Turing and von Neumann are today regarded as the founders of our computing industry, but they are also widely recognized as two of the greatest mathematicians of the twentieth century.

Models and mathematics play fundamental roles in computing. They guide how we design and analyze algorithms, how we design and compare architectures, and how we design software that can be automatically adapted to run efficiently on different architectures. For the past 50 years it has been recognized that a post-von-Neumann model will need to be a parallel model, in order to guide the design of parallel algorithms, software and architectures.

In this chapter we will explain some of the history of attempts to produce a new universal parallel model that could potentially supersede and replace the sequential von-Neumann model. We will also show how mathematics continues to be a central element in all of this research. At the end of the chapter we will suggest a number of new areas for fundamental mathematical research that can help guide and influence future work on parallel algorithms, software and architectures.

## 1.1 Introduction

In the nineteenth century and early twentieth century, “computation” was something done by people who were performing tasks such as counting and analyzing census data, or analyzing and predicting the motion of planets.

The concept of computation as a mathematically precise notion was formalized by several researchers in the 1930s – Emil Post, Alonzo Church, and perhaps most prominently, by Alan Turing. Their formalizations, or models, varied significantly – Post’s Machine, Church’s Lambda Calculus (Church 1941), and Turing Machines (Turing 1937). However, it was soon realized that all of these models defined essentially the same notion

[a] From *Mathematics of Future Computing and Communications*, edited by Liao Heng and Bill McColl © 2022 Cambridge University Press.

of what it means to be “computable”. Today, it remains the case that what we regard as computable is precisely that which corresponds to the Church–Turing thesis. We regard a problem to be computable, or computationally solvable, if and only if it can be solved on a Turing machine.

Besides defining the model, Turing also showed other remarkable mathematical results and insights obtained from using the model. For example, he showed that a single “universal” Turing machine could be designed that could simulate an arbitrary Turing machine on arbitrary input. This indicated that it was possible, in principle, to design and build “general-purpose” computers. This mathematical result was the key insight that would later provide the foundation for the launch of the computing industry and its massive global growth over the past 70 years.

Another spectacular mathematical result published by Turing in 1936 showed that many natural problems were not computable. For example, he showed that it was not possible to write a general program that would take a description of any other program and its input and correctly determine whether or not the program would halt on that input or continue running forever.

The halting problem was one of the first problems to be shown to be non-computable, or undecidable. Also in 1936, Church independently showed that a problem in the lambda calculus was undecidable. Since then, using these mathematically precise models of computation, many other natural and important problems have been shown to be undecidable.

These negative mathematical results, and their methods of proof, have massive ramifications for our computing industry. For example, using essentially the same mathematical methods as those used for the halting problem, we can show that it is not possible to write software that will tell if two given programs will always produce the same results. In fact, an even stronger result is Rice’s theorem from 1951 which says essentially that any non-trivial property of a Turing-complete formalism is undecidable. Informally, what this means is that if you have a programming model or computational model $M$ that is as general and powerful as a Turing machine, then no non-trivial questions about the behavior of programs of $M$ can be answered computationally!

Without a precise mathematical model of computation, none of these major theoretical achievements would have been possible. Today, theory and models remain at the heart of modern computer science as we shall see. They guide how we design and analyze algorithms, how we design and compare architectures, and how we design software that can be automatically adapted to run efficiently on different architectures.

Following Turing’s groundbreaking theoretical work in the 1930s on universal Turing machines, John von Neumann in 1945 proposed a practical design for a computer architecture (von Neumann 1945). Von Neumann’s model consisted of:

- Processing unit with arithmetic logic unit and registers.
- Control unit with instruction register and program counter.
- Memory for data and instructions.
- External mass storage.
- Input and output mechanism.

A central element of the von Neumann model is that it keeps both program instructions and data in read-write random–access memory (RAM). So unlike the Turing machine, which uses tapes to store information, this was a Random Access Machine model.

The von Neumann model has been the standard model for sequential computing for almost 75 years. The basic idea of the model is to decompose computations into a series of steps that will take place one after another, i.e. to decompose the computation temporally, and to use random-access mechanisms to access data.

For the past 50 years it has been recognized that a post-von-Neumann model will need to support parallel computing, in which large scale computations are decomposed both spatially and temporally. In the current era of “big data” are “artificial intelligence”, this translates into a need to decompose or partition large data-intensive computations and the data that they operate on. To achieve scalability and high performance, this of course needs to be done in a way that not only balances the load across the components of the computation, but also minimizes the communication and synchronization between those components.

A further complexity is added by the fact that modern computing systems are not only massively parallel but also heterogeneous. A typical computer architecture today may contain not only CPUs but also GPUs and other special-purpose chips for tensor or other AI computations. Besides the increased hardware complexity, this also adds further issues such as maximizing memory and I/O efficiency.

Despite much effort over the past 50 years, no single model has emerged that fully supports all of what is needed in terms of universality, scalability and performance, but also in terms of resilience and software productivity.

In this chapter we will explain some of the history of attempts to produce a new universal parallel model that could potentially supersede and replace the sequential von-Neumann model. We will also show how mathematics continues to be a central element in all of this research. At the end of the chapter we will suggest a number of areas for future fundamental mathematical research that can guide and influence work on algorithms, software and architectures.

## 1.2 Moving Beyond von Neumann

A successful universal model needs to address a large number of challenges. First and foremost, it needs to provide a convenient framework in which to express many or all of the standard algorithmic structures and patterns that exist across the broad field of computing. These include:

- Dense Linear Algebra.
- Sparse Linear Algebra.
- Spectral Methods, e.g. FFT.
- $N$-Body Methods.
- Structured Grids.
- Unstructured Grids.

- MapReduce, e.g. Monte Carlo.
- Graph Computing.
- Tensor Computing.
- Dynamic Programming and Tabulation.
- Combinatorial Search.
- Branch-and-Bound.
- Linear Programming.
- (Mixed) Integer Programming.
- SAT (Satisfiability) and SMT (Satisfiability Modulo Theories).
- Evolution and Genetic Strategies.
- Machine Learning.
- Deep Learning.
- Differentiable Programming.
- Reinforcement Learning.
- Finite State Machines.
- Discrete Event Simulation.

While lengthy, this list is by no means complete. It does however show the wide range of computations that must be supported by any candidate universal model. But that is not all, even more challenging is the fact that any such model must also support a wide range of other fundamental requirements.

For example, in order to overcome the challenge of the "memory wall", we need a model that can support the need to design communication-optimal algorithms and software, the need to determine optimal placement and data partitioning, and the need for cost modeling and analysis to optimize architectures.

Also, in order to support the goal of achieving a single unified model, and potentially a single unified API, we need:

- Algorithmic universality – the model needs to be general purpose.
- Heterogeneity – software should be portable across all architectures.
- Scalability – the model should work at all scales from a few cores up to many millions.
- Performance AND productivity – the programming model needs to be high-level.
- Analyzability and predictability – the model needs to support cost modeling of algorithms and architectures.
- Automatic mapping and scheduling for portability with optimal performance.
- Automatic handling of faults and tail latency.

In the following sections we will look at various proposals for models that have been made over the past 45 years, and discuss their capabilities and limitations. We will see that, despite much research effort, today we still do not have a single model that provides a compelling solution in all dimensions.

## 1.3 Programming-Oriented Models

The von Neumann model is an example of an architectural model that is neutral in terms of the particular styles of program that can be efficiently run on it. A von Neumann sequential machine is effective at handling not only simple numerical and scientific computations, but also complex data structures and all of the many types of applications that are required across the global computing industry. In contrast, a number of the early examples of models proposed as post-von-Neumann took a more programming-oriented approach – essentially first define the programming model and then try to design an architectural model to support it. In many cases, the resulting architectures have proved to be unable to deliver many of the other characteristics required of a model. We will later see that an alternative approach, that of defining a "bridging model" between the many styles of programming and the many types of architecture, has offered a more convenient way of achieving the many simultaneous goals. In this section we consider some of these programming-oriented models.

### 1.3.1 Dataflow

Around 1974, Jack Dennis (1974) and others suggested that programs should not be tied to a specific sequential temporal execution based on a program counter, as in the von Neumann model. Instead, they argued, the dataflow in a computation was the fundamental aspect, and operations should instead be scheduled and executed as soon as their required input data was available. In this dataflow model, instead of von Neumann assignment to variables, computations would be represented as graphs of single-assignment operations. Such graphs could be static or dynamic.

The dataflow or task parallelism approach has been tried many times over the past 40 years, most recently in the TensorFlow model for machine learning and AI computation. At modest scale the dataflow model can certainly be useful. However, the optimization of dataflow computations at large scale remains a major challenge. Also, as it is a relatively low-level programming model, the productivity of programmers is low, especially when trying to achieve high performance at scale with dataflow models.

As it has been proposed quite recently, it is perhaps important to elaborate further on the TensorFlow model (Dean et al. 2015), developed by Google. TensorFlow computations are expressed as stateful dataflow graphs consisting mainly of operations on tensors (multidimensional data arrays). TensorFlow is not intended to be a model for general purpose parallel computing, or even a model for a broad class of AI computations. It is only intended to support a class of tensor-based machine learning applications. However, these computations (particularly training) are often very computationally intensive, communication-intensive, and highly iterative. They therefore require a model that can offer high performance at scale.

Viewing TensorFlow as a parallel computing model, it is essentially the same as the dataflow models and frameworks that were proposed originally in the 1970s. As a special purpose computing model, TensorFlow provides much of what is needed to build deep learning applications on small-scale parallel systems. As the scale increases and there is a need to carry out the computation on large-scale distributed memory architectures, the standard problems of data distribution and communication minimization become the dominant challenges, as they always do.

In the case of deep learning at small scale, batch splitting for data parallelism can be used. However, at larger scale, this approach suffers from all of the standard problems in parallel computing: memory constraints, high latency, and inefficiency due to fine-grain small batch sizes. To address these issues we need to shift to more of a “model parallelism” approach, using the kind of standard parallel algorithm, software and architectural techniques that have been used in BSP, MPI and other frameworks for over 25 years, such as high performance point-to-point communications and optimized collective communications such as AllReduce implemented efficiently at scale on powerful networks. Today this trend is already underway in systems such as Mesh-TensorFlow (Shazeer et al. 2018) that are starting to use some basic high performance computing concepts.

In summary, TensorFlow provides a convenient special purpose framework for a certain class of tensor dataflow computations. Such special purpose frameworks are not intended to support a broad class of computations. As such they are like scripting languages that can be useful in certain areas, but do not replace more general purpose programming languages. In terms of scalability and performance, it is not clear whether such a dataflow approach can offer convenience in terms of simplicity and software productivity, while at the same time delivering high performance at scale.

## 1.3.2 Functional Programming

In 1977, John Backus, who had designed Fortran in the 1950s, won the Turing Award. His Turing Award lecture (Backus 1978) was entitled “Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs” In it he argued that the world should move on from the temporal imperative programming style of von Neumann and Fortran to a new era in which programming would be much more mathematical and abstract, one where computation would be functional.

Functional models of computation were not new, even in the 1970s. They have been around for a very long time. As noted earlier, the lambda calculus, a functional model of computation, was first introduced by Alonzo Church (1941), around the same time as Turing (1937) introduced the Turing Machine model. Haskell Curry’s work on combinatory logic (Curry & Feys 1958) was another early example of a functional model of computation, closely related to the work of Church. Although even here, Curry was not the first. The first work on combinatory logic was done by Schönfinkel in 1920 in Göttingen.

Not only are functional models of computation an old idea, so are functional programming languages! In 1958, just one year after the development of Fortran, John McCarthy and his colleagues developed the functional programming language Lisp which is based on the lambda calculus. Remarkably, even today, Fortran and Lisp are still both in widespread use around the world, more than 60 years after they were first developed. Since then many other functional programming languages have been developed, including Scheme, ML and Haskell. Today, many popular programming languages such as Python, Lua and Scala have incorporated important elements of functional programming.

Functional models of computation offer significant potential in terms of improving software productivity, as they provide a flexible high-level abstraction that makes programming easier using concepts such as first class and higher-order functions. The main challenge in this area is to provide the necessary software automation to ensure that the required load balancing, communication-efficiency, synchronization-efficiency, etc., can be achieved without the programmer having to specify how it should be done at scale on complex heterogeneous parallel architectures. To date, this remains an unsolved problem.

### 1.3.3 Data Parallelism

Data parallelism is perhaps the simplest model of parallel computation, where the control flow is sequential, as in the von Neumann model, but the data is distributed so that in each step the processors can operate independently on separate parts of the data, with no need for coordination, synchronization, communication or any of the other aspects that make parallel programming challenging.

SIMD machines, vector machines, GPUs and other architectures support various styles of data parallelism. Data parallelism is attractive in terms of programmer productivity, as it is a very simple programming model for simply structured parallel computations, however it lacks the flexibility and power to handle a broad class of computations in an efficient manner. Besides low-level GPU libraries, three recent examples of programming models that are predominantly data parallel are MapReduce, Spark and GraphBLAS. We will say a little about each of these.

#### MapReduce

MapReduce (Dean & Ghemawat 2004) is one of the simplest models of computation ever proposed. It takes two of the most basic functions map and reduce and builds a whole system platform around that basic functionality. Originally developed by Google for simple big data computations, it gained prominence for a brief period in the form of Apache Hadoop. Today, Google and others have moved on beyond the limitations of this primitive model.

Perhaps the most worthwhile element of this minimalist model and platform was that it provided automatic support for redundancy and fault tolerance on cheap commodity clusters. However, this functionality was only achieved at a huge cost in terms of efficiency, since the computations on these MapReduce and Hadoop platforms often involved rescheduling and re-executing large-scale tasks and achieved data resilience using slow and inefficient distributed file systems.

For a very small class of parallel computations with very simple and light communication requirements, MapReduce could be used, but for any computations that required any kind of iteration, the MapReduce model would be hopelessly inefficient due to the requirement to write to, and read from, distributed persistent storage in order to communicate. Since most large-scale computations today such as graph computing, analytics and machine learning involve many rounds of iterative computation, this model is no longer relevant. Moreover, many modern parallel computations are also highly communication-intensive. Since most MapReduce implementations write communications to distributed persistent storage for fault recovery, this also severely limits the applicability of these MapReduce platforms.

### Spark

Recognizing the severe limitations of the MapReduce model for any iterative parallel computation, the Spark model (Zaharia et al. 2010) and platform were developed to address some of these shortcomings. The Spark model supports a class of data parallel computations and, like MapReduce, provides a degree of fault tolerance.

Spark's core idea is the Resilient Distributed Dataset (RDD) which can be viewed as a working set for certain types of parallel computation. Spark provides a form of distributed shared memory. The main benefit is that data management and communication is handled automatically, but of course this is also the main limitation, in that the algorithm designer or programmer has insufficient control over data distribution and communication to achieve high performance in many cases. Like MapReduce, Spark requires a cluster manager and a distributed storage system to handle the various automated elements.

The core programming model for Spark is functional programming on RDDs, using for example the Scala programming language. While this provides a nice high level programming abstraction, the Spark model also has some imperative programming features such as accumulators, and some shared variable mechanisms such as broadcast variables.

The Spark model was an important development in the area of models of parallel computation, as it provided a simple data parallel programming abstraction for a range of big data computations in which one is applying the same operation to all elements of a dataset. However, for large-scale communication-intensive parallel computations, the convenience of automatic management of memory and communications in a model such as Spark comes with a significant price in terms of performance.

### GraphBLAS

The Basic Linear Algebra Subprograms (BLAS) have, for many years, provided a convenient set of algorithmic software tools for high performance computing. The objective of work on GraphBLAS (Buluç et al. 2017) is to provide a similar framework of building blocks for graph computing. The starting motivation for this work is the simple observation that any weighted graph can be represented by its adjacency matrix, and with such a representation, we can easily transform linear algebra on matrices and vectors into graph computations such as breadth first search by simply changing the scalar functions or operators. For example, changing the standard operator pair (*,+) into the semiring (+,min) we can compute various functions on paths in graphs.

GraphBLAS provides a simple high level functional programming model for computations on sparse and dense graphs, with the potential to improve software productivity. Programs operate directly on vectors and matrices, both of which can be dense or sparse matrices. Implementations of GraphBLAS can take advantage of this sparsity to achieve high performance. They can also exploit the freedom that is provided by such a high level programming model, in order to optimize data distribution and the associated parallel communication and synchronization in distributed memory architectures. The various standard higher order functions are provided as data-centric primitives or building blocks:

- Matrix-vector product.
- Matrix-matrix product.
- Inner product.
- Data parallel elementwise addition and multiplication.
- Fold left and fold right.

It should be noted that the last two correspond closely to map and reduce, and can be generalized in that direction. Similarly, the basic mechanisms of BSP data-centric "Think Like a Vertex" graph frameworks such as Pregel, which we will discuss later, can also be modeled using GraphBLAS primitives. So, the GraphBLAS model may be extensible in ways that offer more flexibility than one would expect at first sight.

### 1.3.4 Message Passing

In 1977, Tony Hoare introduced the communicating sequential processes (CSP) model of concurrent computation (Hoare 1978). The CSP model is based on the fundamental idea that synchronization and communication should be tightly coupled. CSP computations use synchronous communication via channels for sending and receiving. Other models of concurrent computation were introduced around the same time, such as Robin Milner's Calculus of Communicating Systems (CCS) and other process algebras.

In the 1980s the message passing model was used in a number of early parallel computer architectures, including the Inmos Transputer with its associated programming language Occam, which was heavily influenced by CSP. This message passing approach came to be more generally referred to as "distributed memory parallel computation" in contrast to the kind of shared memory models that we will consider next.

In the 1990s there was a rapid proliferation of distributed memory message passing architectures and programming tools. In response to this, a committee was formed – the MPI Forum – with the goal of developing a standard Message Passing Interface (MPI) for point-to-point and collective communications in a distributed memory parallel architecture. The first version of the MPI standard was designed in 1993 consisting of a library of functions usable with standard sequential programming languages such as C. Since then a series of further updated versions have been defined, and some of them have been implemented.

Hoare's original CSP model can be seen as addressing the need for a sound mathematical framework for low-level communications. As such, it succeeds, in that complex low-level protocols can be analyzed and verified using the CSP model and its associated tools. In the case of the Transputer architecture and the Occam language, this very low-level message passing is fundamental to the hardware architecture. In the case of

MPI it is also a very low-level model in which the objective is simply to support parallel communications, rather than to help algorithm designers and programmers to structure, analyze and optimize their parallel computations.

As a low-level model, unstructured synchronous message passing certainly allows massively parallel computation to be achieved on distributed memory architectures, but it does so at considerable expense. Firstly, it permits the most serious types of parallel programming problems to arise, such as deadlock and race conditions. Secondly, it makes debugging extremely complex, as there is no easy way to capture and analyze the state of a large-scale unstructured message passing program. Thirdly, software productivity is extremely low, as it is such a low-level model.

We will see later that the BSP model enables us to overcome these many limitations of message passing. Interestingly, it does so by abandoning the fundamental element of CSP message passing, by decoupling synchronization and communication. Instead of message passing being a synchronous two-sided activity, in BSP it becomes an asynchronous (or more accurately bulk synchronous) one-sided activity.

### 1.3.5 Other Programming-Oriented Models

Some proposed models have gone through periods where they have rapidly achieved a degree of prominence, only to fade away equally rapidly as candidates for a post von Neumann model. One example of this was logic programming models, and related languages such as Prolog, that were first developed in the 1970s. Throughout the 1980s, concurrent logic programming was heavily promoted by Japan's "Fifth Generation Computer Systems" initiative as a foundation and model for future AI and other advanced computing applications. Despite very significant research funding, no evidence emerged that this approach could ever deliver high performance at scale.

## 1.4 An Algorithm-Oriented Model

In algorithm design and complexity theory, a cost model is normally assumed implicitly, based on the von Neumann computational model. If there is a need to define it more explicitly, then it is typically assumed to be the Unit-Cost Random Access Machine model. This simply says that it is a sequential model in which the cost of reading from, or writing to memory is constant, irrespective of the memory size. It is also assumed that all operations take constant (or unit) time. This simplifying (and idealized) assumption makes it easy for algorithms to be compared in terms of their running times, without a large amount of mathematical machinery being required. Of course, it is an idealized model in that for example, it will usually take longer to access physical memory A than physical memory B if the former is one billion times larger than the latter. Even in theoretical terms it is idealized in the sense that it does not, for example, reflect the costs on a Turing Machine. Nevertheless it is almost universally used informally in algorithm design.

Since the 1970s there has been a large and growing body of research on parallel algorithm design and parallel complexity theory. Much of the early work in this area also adopted a simple, idealized shared memory model – the Parallel Random Access Machine or PRAM. A PRAM consists of a collection of processors which compute synchronously in parallel and which communicate with a common global random access memory. In one time step, each processor can do (any subset of) the following:

- Read two values from the common memory.
- Perform a simple two-argument operation.
- Write a value back to the common memory.

There is no explicit communication between processors. Processors can only communicate by writing to, and reading from, the common memory. The processors have no local memory other than a small fixed number of registers which they use to temporarily store the argument and result values.

In a Concurrent Read Concurrent Write (CRCW) PRAM, any number of processors can read from, or write to, a given memory cell in a single time step. In a Concurrent Read Exclusive Write (CREW) PRAM, at most one processor can write to a given memory cell at any one time. In the most restricted model, the Exclusive Read Exclusive Write (EREW) PRAM, no concurrency is permitted either in reading or in writing. The CRCW PRAM model has a large number of variants which differ in the convention they adopt for the effect of concurrent writing. Three simple examples of such conventions are:

- Two or more processors can write so long as they write the same value.
- One of the processors attempting to write will succeed but the choice of which one will succeed will be made non-deterministically.
- The lowest numbered processor will succeed (assuming some appropriate numbering).

In other CRCW models, one might have the possibility of concurrent writing in which the memory location is updated to the sum of the written values, or to the minimum of the written values.

Despite being idealized in many ways, the PRAM model does provide a simple cost model for parallel algorithms that allows the theoretical degree of parallelism in a computation to be mathematically explored in certain ways. As a practical model of parallel computation it is however inadequate, since it ignores the costs of communication and synchronization in large-scale physical parallel architectures, and these are quite often the most important elements in the cost. While shared memory may be a realistic assumption when the number of processors is small, for example, 4, 8 or even 16, it is hopelessly unrealistic in situations where the number of processors is, for example, one million.

While shared memory is clearly not a viable model for large-scale general purpose parallel computing, and therefore cannot be a candidate for a universal post-von-Neumann model, it has a useful role to play in certain small-scale settings. Today, many/most chips are multicore, as it is cheap and easy to arrange a small number of general purpose cores on a single chip. In cases where the individual cores are very simple, as in accelerators, then the numbers of such cores within a single device may be quite high, but it remains the case that for systems in which the number of chips themselves is large, shared memory is not a viable model.

## 1.5 A Bridging Model

In 1990, Leslie Valiant published the first description of a new model of computation, the Bulk Synchronous Parallel (BSP) model (Valiant 1990*a*), based on the idea of computing in rounds, or supersteps. In the initial description, a strong emphasis was placed on demonstrating that the idealized PRAM shared memory model could be realized by simulation on certain forms of BSP machine with appropriate properties. This "Automatic-Mode BSP" was based on the ideas of using hashing to randomize the distribution of data, and of using "parallel slackness" to hide or tolerate latency to non-local memory. More importantly, Valiant showed that BSP had a simple cost model that could be used to capture the important costs of communication and synchronization in a parallel computation in a convenient way, enabling powerful analysis, comparison and optimization of parallel algorithms.

These two initial achievements of the BSP superstep model in demonstrating PRAM simulation via automatic memory management, and accurate cost modeling of parallel algorithms, were merely the first steps in a long line of BSP innovations that followed through the 1990s. Most of the subsequent research, rather than focusing on Automatic-Mode BSP, PRAM simulation, strobing barriers etc. instead pursued the alternative direction of BSP as an architectural and/or programming model. In this "Direct-Mode BSP", data distribution and the associated communications and synchronization would be controlled by the algorithm designer or programmer, with the system only automating the low-level scheduling of those communications to avoid congestion and maximize throughput.

Between 1990 and 1992, Leslie Valiant and I worked on ideas for a distributed memory BSP programming model. Between 1992 and 1997, I led a large research team at Oxford that developed various BSP programming libraries, languages and tools, and also numerous massively parallel BSP algorithms (McColl 1995). With interest and momentum growing, I then led an international group that developed and published the BSPlib Standard for BSP programming (Hill et al. 1998) in 1996. Since then, there have been hundreds of papers on BSP research, and BSP has become a standard model for parallel computing (Skillicorn et al. 1997). It is now widely used in research and in industry in many areas, including data analytics, graph computing, machine learning and HPC.

### 1.5.1 BSP Architectures

A BSP computer consists of a set of processor-memory pairs, a global communications network, and a mechanism for the efficient barrier synchronization of the processors. A BSP computer operates in the following way. A computation consists of a sequence of parallel supersteps, where each superstep consists of a sequence of steps, followed by a barrier synchronization at which point all data communications will be completed. During a superstep, each processor can perform a number of computation steps on values held locally at the start of the superstep, send and receive a number of messages, and handle various remote read/get and write/put requests.

The BSP computer is a two-level memory model, i.e. each processor has its own physically local memory module; all other memory is non-local, and is accessible in a uniformly efficient way. By uniformly efficient, we mean that the time taken for a processor to read from, or write to, a non-local memory element in another processor-memory pair should be independent of which physical memory module the value is held in. The algorithm designer and the programmer should not be aware of any hierarchical memory organization based on network locality corresponding to the particular structure of the communications network.

BSP computations have both a horizontal (spatial) structure and a vertical (temporal) structure. The horizontal structure arises from concurrency, and consists of a fixed number of virtual threads. These are each associated, at run-time, with a physical processor. The vertical structure arises from the progress of a computation through time. For BSP, this is a sequential composition of global supersteps, which conceptually occupy the full width of the executing architecture. Each superstep is further subdivided into three ordered phases consisting of:

- Computation locally in each thread, using only values stored in the local memory of each processor.
- Communication actions, e.g. put and get, amongst the threads, involving movement of data between processors.
- Barrier synchronization, which waits for all of the communication actions to complete, and which then makes the data that was moved available in the local memories of the destination processors.

A superstep is shown in Figure 1.1.

If the target parallel computer has fewer processors than the virtual parallelism (parallel slackness), then a simple transformation can be used to convert the BSP program into a slimmer version. For example, if number of virtual processes is twenty times larger than the number of physical processors then we can map twenty virtual processes to each processor. Moreover, this can be done in a way that maximizes the number of communications that become internal operations.

## 1.5.2 BSP Cost Modeling

If we define a time step to be the time required for a single local operation, i.e. a basic operation (such as addition or multiplication) on locally held data values, then the performance of any BSP computer can be characterized by three parameters:

- $p$ = Number of processors.

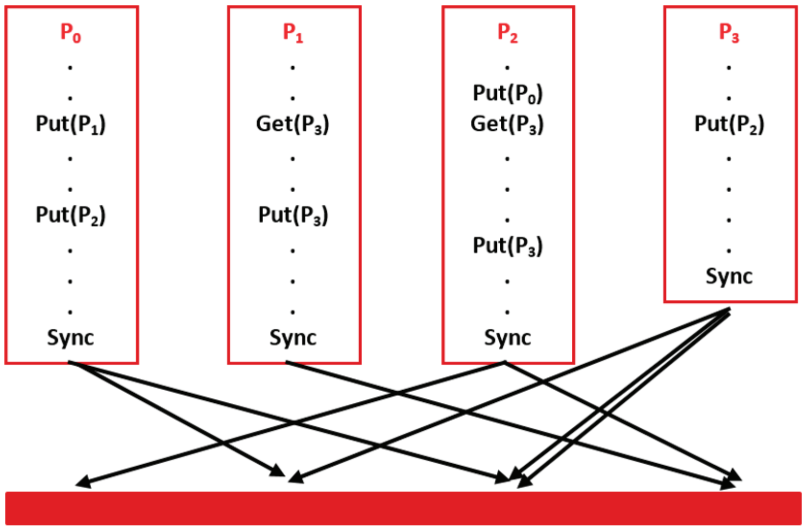


**Figure 1.1** Superstep

- $g$ = (Total number of local operations performed by all processors in one second)/ (Total number of words delivered by the communications network in one second, in a situation of continuous traffic).
- $L$ = Number of time steps for barrier synchronization.

There is also, of course, a fourth parameter $s$, the number of time steps per second. However, since the other parameters are normalized with respect to that one, it can be ignored in the design of algorithms and programs. The parameter $g$ corresponds to the frequency with which non-local memory accesses can be made; in a machine with a higher value of $g$ one must make non-local memory accesses less frequently. More formally, $g$ is related to the time required to realize $h$-relations in a situation of continuous message traffic. An $h$-relation is a communication pattern in which each processor sends up to $h$ messages and receives up to $h$ messages. The parameter $g$ is the value such that any $h$-relation can be performed in $g * h$ time steps.

Any parallel computing system can be regarded as a BSP computer, and can be benchmarked accordingly to determine its BSP parameters $L$ and $g$. The BSP model is therefore not prescriptive in terms of the physical architectures to which it applies. Every general purpose parallel architecture can be viewed by an algorithm designer or programmer as simply a point $(p, g, L)$ in the space of all BSP machines.

In theoretical terms, the BSP model can be regarded as a generalization of the PRAM model which permits the frequency of barrier synchronization, and hence the demands on the network, to be controlled. If a BSP architecture has a very small value of $g$, for example $g = 1$, then it can be regarded as a PRAM and we can use hashing to automatically achieve efficient memory management. The value of $L$ will determine the degree of parallel slackness required to achieve optimal efficiency. The case $g = L = 1$ corresponds to the idealized PRAM, where no parallel slackness is required.

The time for a superstep is determined as follows. Let the work $w$ be the maximum number of local computation steps executed by any processor during the superstep, and let $h_s$ be the maximum number of messages sent by any processor, and $h_r$ be the maximum number of messages received by any processor during the superstep. The time for the superstep is then at most $w + g * \max\{h_s, h_r\} + L$ steps. The total time required for a BSP computation is easily obtained by adding the times for each superstep. Analyzing and predicting the cost of a BSP program is, therefore, no more difficult than analyzing and predicting the cost of a sequential program.

By adding the time for each of $S$ supersteps we obtain an expression of the form $W + g * H + L * S$ where $W$, $H$, $S$ will typically be functions of $n$ and $p$. In designing an efficient BSP algorithm or program for a problem which can be solved sequentially in time $T(n)$ our goal will, in general, be to produce an algorithm requiring total time $W + g * H + L * S$ where $W(n, p) = T(n)/p$, $H(n, p)$ and $S(n, p)$ are as small as possible, and the range of values for $p$ is as large as possible. In many cases, this will require that we carefully arrange the data distribution so as to minimize the frequency of remote memory references. Another property of interest in BSP algorithm design is the space (or memory) efficiency of the computation. We use $M(n, p)$ to denote the maximum number of values which any one processor has to store at any point during the computation.

### 1.5.3 Why BSP?

**Universal. General purpose.**
The essence of the BSP approach to parallel programming is the notion of the superstep, in which communication and synchronization are completely decoupled. A "BSP program" is simply one which proceeds in phases, with the necessary global communications taking place between the phases. This approach to parallel programming is applicable to all kinds of parallel architectures and parallel algorithms. It provides a consistent, and very general, framework within which to develop portable parallel software for scalable parallel architectures.

**Simple.**
BSP programs are much the same as sequential programs. Only a bare minimum of extra information needs to be supplied to describe the use of parallelism.

**Easy to debug. No deadlock.**
Since communication and synchronization are decoupled in a BSP program, the programmer does not have to worry about problems such as deadlock, which can occur with synchronous message passing. Debugging a BSP program is also made much easier by this decoupling. The barrier at the end of a superstep provides an appropriate breakpoint at which the global state of the parallel computation is well defined and can be interrogated. Debugging and reasoning about the correctness of a BSP program are, therefore, not much more difficult than for a sequential program.

**Portable. Independent of target architectures.**
Unlike many parallel programming models, BSP is designed to be architecture-independ-

ent, so that programs run unchanged when they are moved from one architecture to another. Thus BSP programs are portable in a strong sense.

**Performance of a program on a given architecture is predictable.**
The execution time of a BSP program can be computed from the text of the program and a few simple parameters of the target architecture. This makes design space exploration possible, since the effect of a decision on performance can be easily determined.

**Analyzable.**
The BSP cost model makes it clear what strategies should be adopted to write efficient BSP programs. We should balance and minimize the computation between threads, since $W$ is a *maximum* over computation times. We should balance and minimize the communication between threads, since $h$ is a *maximum* over fan-in and fan-out of data. We should minimize the number of supersteps, since this determines the number of times $L$ appears in the final cost. The cost model also shows how it can be used to predict performance on a particular target architecture. The values of $W$ and $h$ for each superstep, and the number of supersteps can be determined by inspection of the program code. Values of $p$, $g$, and $L$ can then be inserted to give execution time before the program is executed.

**Easy to checkpoint for resilience.**
The fact that there we can easily capture a well-defined state at each barrier also means that we have a simple way in which to perform checkpointing and recovery from hardware failures. In contrast, capturing and checkpointing the state of a parallel computation in a message passing system such as MPI is incredibly difficult. In a later section, on the BSP cloned computing model, we will see that BSP superstep semantics not only allows us to handle hardware failures, but also to efficiently handle long tail latencies, which is a much more challenging problem.

**Communications can be optimized for global exchange.**
In large-scale parallel computing it is almost always the case that the parallelism in the software vastly exceeds the parallelism in the hardware. This is a simple consequence of the power of modern computing hardware. A single core is capable of performing over ten billion operations per second. In the example mentioned above, where a graph algorithm has one trillion parallel subtasks, these subtasks will be mapped onto a physical machine which has a much, much smaller degree of parallelism. For example, it might be mapped onto 200 16-core machines, in which case each core will be handling around 300 million small parallel subtasks. The same phenomenon of parallel slackness occurs in every area of large-scale parallel computing – sparse matrix computations, machine learning, modeling, optimization etc.

A major consequence of parallel slackness is that it will normally be the case that in any large-scale parallel computation, each processor will be communicating with every other processor as the computation proceeds. So, to achieve peak performance we need to design parallel software systems that are optimized for this global pattern of communication in which each processor is sending to all the others, and is receiving from all the others. This global pattern is called total exchange. BSP software systems are optimized for total exchange, rather than for single, individual pairwise communications, as in MPI and other asynchronous or synchronous message passing systems. When processors need to send and receive millions of values to/from every other processor in a single round of a parallel computation, this provides a massive improvement in efficiency and performance.

**Communications can be non-blocking, zero-copy, one-sided.**
Message passing systems such as MPI are based on synchronous communications between pairs of processors. In order for processor A to provide even a single value to processor B it needs to synchronize with B, and then communicate the value to B. This is incredibly costly, in terms of time, especially in situations where there are huge numbers of fine-grain communications that need to take place, as is increasingly the case in modern applications and services.

In designing the BSP software model, and in defining the BSPlib standard, we solved this problem by introducing non-blocking zero-copy one-sided communications as the default model for communications. This was made possible due to the superstep structure of BSP computations, with its barrier synchronization model. The shift from slow two-sided MPI communications to zero-copy one-sided BSP communications has resulted in major improvements in efficiency and performance, especially for modern fine-grain parallel computations such as sparse matrix computations, graph computing and machine learning.

In BSP, data communication can be carried out using Remote Direct Memory Access (RDMA) which is super-fast, both in terms of high-throughput and ultra-low-latency. BSP-style RDMA is now supported directly in communication networks such as InfiniBand. Some newer versions of MPI (MPI-2) have added primitives for one-sided communications, although to use them effectively requires the MPI program to be structured in a BSP style.

**Communications can be globally scheduled and optimized.**
In BSP, the system automatically optimizes communication and synchronization, whereas in MPI the programmer is responsible for this optimization, which can be very complex. An extremely expert MPI programmer can try to do this, and with the one-sided communications of MPI-2 may be able to achieve comparable performance to BSP in some cases. However, using BSP tools directly, this whole complexity is removed from the programmer, and the system can automatically schedule communications to ensure maximum network throughput and ultra-low latency, by eliminating congestion. This is possible in BSP because communication is non-blocking, and therefore the communications can be sent at any time during the superstep, and in any order. The system can schedule the total exchange of messages between nodes to ensure that the network is load balanced at all points.

For example, suppose there are four nodes $1, 2, 3, 4$ and each node has to send data to the other three. The BSP system can simply schedule the parallel communications in three rounds:

$$[1 \rightarrow 2\,,\ 2 \rightarrow 3\,,\ 3 \rightarrow 4\,,\ 4 \rightarrow 1]$$
$$[1 \rightarrow 3\,,\ 2 \rightarrow 4\,,\ 3 \rightarrow 1\,,\ 4 \rightarrow 2]$$

$$[1 \rightarrow 4\,,\, 2 \rightarrow 1\,,\, 3 \rightarrow 2\,,\, 4 \rightarrow 3].$$

This kind of schedule ensures that the network achieves the highest possible performance, and that there is no congestion at the nodes. The BSP system is able to easily ensure this optimality, whereas asking the programmer to do it is not only adding complexity to the software development, but also the programmer does not have the degree of control over network timing issues etc. to guarantee it, even if they have the one-sided communications primitives required available to them. Despite their best efforts, the programmer may therefore end up sending the communications in three rounds, but as:

$$\begin{array}{c}[1 \rightarrow 4\,,\, 2 \rightarrow 1\,,\, 3 \rightarrow 1\,,\, 4 \rightarrow 1]\\ [1 \rightarrow 2\,,\, 2 \rightarrow 4\,,\, 3 \rightarrow 2\,,\, 4 \rightarrow 2]\\ [1 \rightarrow 3\,,\, 2 \rightarrow 3\,,\, 3 \rightarrow 4\,,\, 4 \rightarrow 3]\end{array}$$

or some other bad schedule, resulting in congestion and poor performance. When the number of nodes is not 4, but 400 or 4000 this can be of huge importance.

**Communications can be combined and optimized.**
The high degree of parallel slackness in most BSP applications and services means that the number of separate messages to be sent from processor A to processor B during a single superstep can be very large. For example, in graph applications the number of separate messages may be hundreds of thousands or even millions. In traditional message passing, each of these messages would involve a separate synchronization and communication, leading to massive inefficiency. In such cases, the programmer would be forced to design additional software to combine the data to be sent into structures, in order to achieve acceptable performance levels. These message structures would need to be assembled and disassembled by the programmer.

In BSP this is not necessary, again due to the non-blocking nature of the communications, and the scheduling of communications by the system rather than by the programmer. Since the communications can be sent at any time during the superstep, and in any order, the BSP system is free to automatically combine any number of individual messages for a given destination, and send the whole set as a single message. The impact on performance of this kind of message combining can again be very significant, and it is transparent to the programmer, requiring no additional effort.

### 1.5.4 Data-Centric BSP

In 2010, Google realized, after many years of experience with the limitations of MapReduce, that they needed a new model to support the many large-scale graph computing and analytics problems they were facing. They quickly realized that by using the BSP model in a data-centric way they could achieve the ambitious goals they had in terms of scalability and performance. The result was a new Google system named Pregel (Malewicz et al. 2010) which came with a new catchphrase “Think Like a Vertex”. Following the work at Google on Pregel, an open source data-centric BSP system for graph computing – Apache Giraph – was developed, and since then many other such systems have appeared. Today, data-centric BSP has become the standard way to perform large-scale graph computations in databases and other analytics systems.

As noted above, the basic idea behind data-centric BSP systems such as Pregel is to “Think Like a Vertex”. What this means is that the data structure, for example a graph with one billion vertices should be mapped to a virtual parallel computation in which vertices become local computations and edges become communications. As in all BSP systems, the computation proceeds in rounds or supersteps. In each round, each vertex will perform local computation, send messages to neighbors in the graph, and receive messages from neighbors, as shown in Figure 1.2. This iterative computation continues until some given termination condition is met.

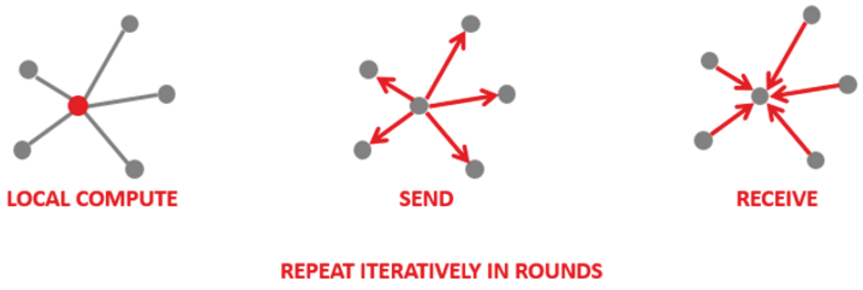


**Figure 1.2** Data-centric BSP

As in all BSP systems, superstep semantics ensures that we have no concurrency issues such as race conditions or deadlock. The degree of fine-grain virtual or logical parallelism in such a computation can of course be huge, perhaps one billion or one trillion. Practical parallel machines, on the other hand, will typically have only a few hundred or a few thousand cores. So, as noted previously, this massive scale virtual computation can be easily mapped onto a much smaller scale physical parallel machine in a straightforward way, and in a way that takes account of the structure of locality within the graph to optimize communications.

Data-centric BSP has been extensively used in recent years by many companies around the world for practical computations on graphs with billions of vertices and trillions of edges. Besides delivering massive scalability and extremely high performance, the model is also very easy to learn and to use effectively. For example, Google’s PageRank algorithm, which is central to their search methods can be described in just 15-20 lines of code, since the algorithm can be formulated in data-centric BSP as the following simple iterative computation:

- Read neighbor ranks.
- Update local rank.
- Send new rank to neighbor.

The database company TigerGraph recently announced that their data-centric BSP graph database was now the world's fastest transactional graph database in production, handling huge volumes of ecommerce and financial transactions.

Although we have emphasized graph analytics in discussing the data-centric BSP model, it is also applicable to many other classes of parallel computation. For example, it can be used in scientific and engineering computations such as stencil computations, $n$-body problems, finite elements, and circuit modeling. Also in machine learning and other areas of AI.

## 1.6 Parallel Algorithms and Complexity

The simplicity of the BSP cost model enables "cost-driven design" of parallel algorithms, parallel software and parallel architectures. As noted above, in designing an efficient BSP algorithm or program for a problem which can be solved sequentially in time $T(n)$, our goal will, in general, be to produce an algorithm requiring total time $W + g * H + L * S$ where $W(n,p) = T(n)/p$, $H(n,p)$ and $S(n,p)$ are as small as possible, and the range of values for $p$ is as large as possible.

### 1.6.1 BSP Algorithms for Common Parallel Patterns

Many static computations can be conveniently modeled by directed acyclic graphs (DAGs), where each node corresponds to some simple operation, and the arcs correspond to inputs and outputs. These DAGs often have a simple structure or pattern. In this section we will look at some of those patterns. In what follows, in the interests of simplicity and clarity, we will often ignore small constant factors.

#### Tree Computations

An important pattern in parallel computation is the tree pattern. Figure 1.3 shows a ternary tree.

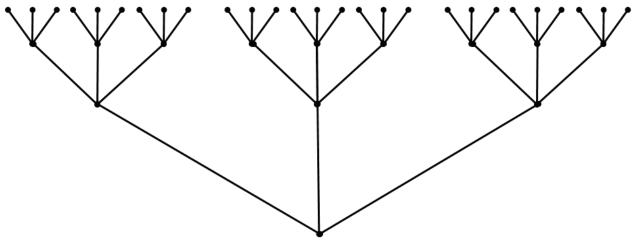


**Figure 1.3** Ternary tree pattern

Consider the simple problem of broadcasting a single value from one processor to all other $p - 1$ processors. We can use a balanced $k$-ary tree of height $s$, where $s = \log_k p$. This corresponds to a BSP algorithm with $s$ supersteps, each costing $g * k + L$. The total cost is therefore $s * (g * k + L)$, so if we choose $k = L/g$, then the cost will be

$2 * \log_k p * L$. If we choose $k = 2$, then the cost will be $\log_2 p(g * 2 + L)$. If we chose $k = p$, then we simply broadcast in a single round in which case the cost will be $g * p + L$. So, even for a very simple problem such as broadcasting a single value, there are a wide range of choices, and the cost model allows these to be easily analyzed and compared.

Consider now the related problem of broadcasting an array of $n$ values from one processor to all other $p - 1$ processors, where $n$ is much larger than $p$. For this problem we can use a different approach. In the first superstep, the source processor sends each of the other processors a distinct sub-array of size $n/p$. The cost of this step is $g * n + L$. In the second superstep, each processor sends its sub-array to all of the other processors. The cost of this step is again $g * n + L$, so the total cost is only $2 * (g * n + L)$.

So, given a particular problem of broadcasting an array of $n$ values on a $p$ processor machine with BSP parameters $g$ and $L$, for some particular values of $n$,$p$,$g$ and $L$, this kind of cost-driven design can be used to obtain the best possible solution. The same kind of analysis can be applied to find optimal algorithms for related tree-structured problems such as Reduce (single tree) and Prefix Sums (double tree).

## Butterfly Computations

Another important pattern in parallel computation is the butterfly pattern. Figure 1.4 shows an 8-point butterfly.

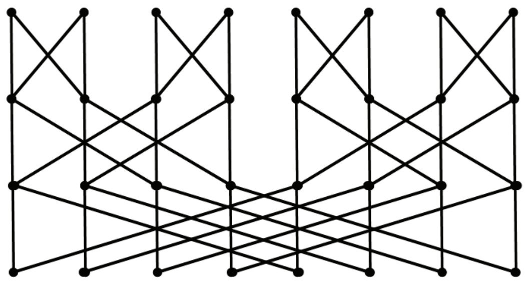


**Figure 1.4** 8-point butterfly

Consider the problem of computing the Fast Fourier Transform (FFT) of $n$ points. It is well known that this algorithm has sequential complexity $O(n \log n)$ and that it can be represented as a $(\log n)$-level butterfly graph with a bit reversal permutation applied to the inputs. It can be implemented on a $p$-processor BSP machine in $(\log n)/(\log(n/p))$ supersteps, in each of which each processor sequentially computes the next $\log(n/p)$ levels of the butterfly graph on its $n/p$ points in time $(n/p) * \log(n/p)$. The cost of each superstep is $(n/p) * \log(n/p) + g * (n/p) + L$, and therefore the total time required is at most $((n \log n)/p) * (1 + g/\log(n/p) + L/((n/p)\log(n/p)))$. In most practical situations, the number of processors $p$ will be no more than $n^{1/2}$. In such cases, the number of supersteps would be at most two, and the total time required would be $(n \log n)/p + g * (n/p) + L$.

## 2D Computations

Another important pattern in parallel computation is the two-dimensional grid DAG. Figure 1.5 shows a $9 \times 9$ 2D grid in which all of the 144 arcs are directed downwards.

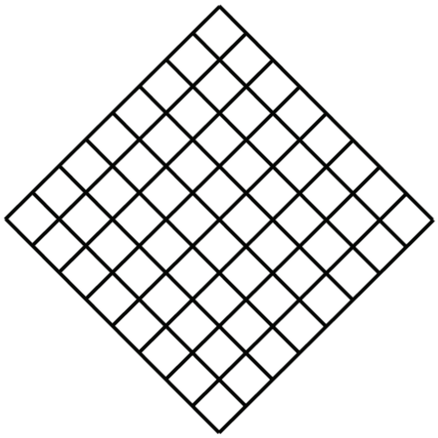


**Figure 1.5** 2D grid DAG

Let $D_n$ denote the 2D Grid DAG which has $n^2$ nodes $v_{i,j}$, $0 \leq i, j < n$, and arcs from $v_{i,j}$ to $v_{i+1,j}$ and $v_{i,j+1}$where those nodes exist. In McColl (1995) it is shown that the graph $D_n$ can be efficiently scheduled for a $p$ processor BSP computer by partitioning $D_n$ into $p^2$ subgraphs, each of which is isomorphic to $D_{n/p}$. The resulting schedule shows that any computation that can be mapped directly onto $D_n$ has a BSP implementation with cost at most $n^2/p + g * n + L * p$. Note that for such a BSP algorithm, in some cases, using more processors will actually *increase* the runtime. For example, consider a BSP architecture based on a simple ring. Such an architecture will have parameters $g = L = p$. For such a machine the runtime will be $n^2/p + n * p + p^2$. This runtime is minimized when $p = n^{1/2}$. Increasing the number of processors beyond this value will increase the runtime.

In McColl (1995) it is shown that the problem of solving a triangular $n \times n$ linear system can be mapped onto $D_n$, so we obtain the above upper bound for that problem. Many other problems can be similarly mapped onto $D_n$. Dynamic programming algorithms are often used in combinatorial search problems such as string comparison and computing string edit distance. These dynamic programming algorithms use tabulation to successively compute the entries of a 2D cost matrix. Such computations map directly onto $D_n$.

## 3D Computations

The 2D Grid pattern can be easily extended to higher dimensions. Figure 1.6 shows a $6 \times 6 \times 6$ 3D Grid DAG in which all of the 540 arcs are directed downwards.

Let $C_n$ denote the 3D Grid DAG which has $n^3$ nodes $v_{i,j,k}$, $0 \leq i, j, k < n$, and arcs from $v_{i,j,k}$ to $v_{i+1,j,k}$, $v_{i,j+1,k}$ and $v_{i,j,k+1}$where those nodes exist. In McColl (1995) it

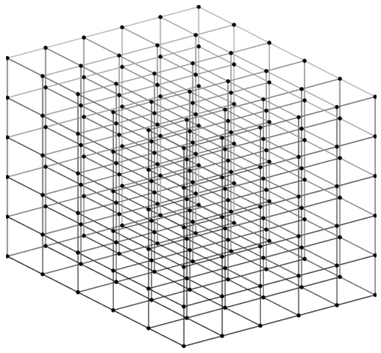


**Figure 1.6** 3D grid DAG

is shown that the graph $C_n$ can be efficiently scheduled for a $p$-processor BSP computer by partitioning $C_n$ into $p^{3/2}$ subgraphs, each of which is isomorphic to $C_{n/p^{1/2}}$ . The resulting schedule shows that any computation that can be mapped directly onto $C_n$ has a BSP implementation with cost at most $n^3/p + g * (n^2/p^{1/2}) + L * p^{1/2}$.

In McColl (1995) it is shown that the problem of computing the LU decomposition of an $n \times n$ linear system can be mapped onto $C_n$, so we obtain the above upper bound for that problem. Many other problems can be similarly mapped onto $C_n$ or onto closely related 3D graphs. One such example is the Algebraic Path Problem (APP) over a closed semi-ring, which includes as special cases the problems of computing shortest paths and computing transitive closures. The standard Floyd-Warshall dynamic programming algorithm for the APP can be mapped directly onto a minor variant of $C_n$ in which some of the arc directions are modified. With this approach we get the above upper bound for all of the various instances of the APP too.

We have explicitly considered 2D and 3D computations. However, the same general scheduling techniques can be easily extended to similar but higher dimensional directed acyclic graphs.

### 1.6.2 Communication-Optimality and Immortal Algorithms

To further illustrate some of the important issues in BSP algorithm design, we will consider several fundamental computational problems involving vectors and matrices.

#### Dense Matrix-Vector Multiplication

Consider the problem of multiplying an $n \times n$ matrix M by an $n$-element vector $v$ on $p$ processors, where $M, v$ are both dense. In McColl (1995) it is shown that by using a “block-block” or a “block-grid” distribution of the matrix $M$ we can produce a simple BSP algorithm with cost $n^2/p + g * (n/p^{1/2}) + L$.

To compute $u = M.v$, the matrix $M$ and vectors $u, v$ are partitioned uniformly across the $p$ processors. In the first superstep, each processor gets all the vector elements $v_j$ for which it holds a corresponding matrix element $m_{i,j}$. In the second superstep, each processor computes partial sums of $m_{i,j}.v_j$ products and sends each of those partial sums to the processor responsible for computing $u_i$. In the third and final superstep, each processor computes the value of its $u$ vector elements by adding up the relevant partial sums received.

A simple input-output complexity argument can be used to show that for any BSP algorithm that computes $u = M.v$, if the parallel work $W(n, p) = n^2/p$ then the parallel communication $H(n, p)$ must be at least $n/p^{1/2}$. Noting that the $n^2$ sequential computation cost is itself optimal we see that the above method is in a strong sense a best possible BSP algorithm for this problem. It simultaneously achieves the optimal computation cost $W(n, p) = n^2/p$, the optimal communication cost $g * H(n, p) = g * (n/p^{1/2})$, and the optimal synchronization cost $S(n, p) * L = L$, all to within small constant factors. It is therefore an "immortal algorithm" – an algorithm that achieves the best possible parallel performance for all values of $n$, $p$, $g$ and $L$.

### Sparse Matrix-Vector Multiplication

Consider again the problem of computing $u = M.v$, but now in the case where the matrix $M$ is sparse. Sparse matrix-vector multiplication is a problem of fundamental importance in computing. It is at the heart of many applications which use iterative methods to solve very large linear systems. To enable the multiplication to be performed repeatedly with no additional redistribution we require that the result vector $u$ should be distributed across the parallel machine in the same way as the input vector $v$, that is, the processor holding $v_i$ at the start of the computation should hold $u_i$ at the end of the computation.

Let $C(r, d)$ denote the adjacency matrix of the directed $r$-ary, $d$-dimensional hypercube graph. The nodes of this graph form a $d$-dimensional grid of $n = r^d$ points which are numbered lexicographically. Each node has directed arcs to itself and to its immediate neighbors in each dimension. For the purposes of discussion we will consider just four $n \times n$ sparse matrices. Three of the four are instances of $C(r, d)$. They are:

- 2D-MESH = $C(n^{1/2}, 2)$.
- 3D-MESH = $C(n^{1/3}, 3)$.
- HYPERCUBE = $C(2, \log n)$.

Matrices of this kind are often used to model finite-difference operators in the solution of partial differential equations. The fourth matrix has a random structure in which each row and each column contains four non-zeros. We will refer to it as the EXPANDER matrix. [Note. The value four is not particularly significant. We could have chosen any small integer value greater than one.]

In McColl (1995) it is shown that significant reductions in communication cost $H(n, p)$ can be achieved by using a data distribution based on an efficient decomposition of the underlying graph. For example, the nodes of the 2D-MESH graph can be partitioned into $p$ regions, each of which corresponds to a 2D-MESH on $n/p$ nodes. The corresponding data distribution for matrix elements gives a BSP algorithm for $u = M.v$, where $M$ is the 2D-MESH matrix, with total cost $n/p + g * (n^{1/2}/p^{1/2}) + L$. The same approach, applied to the 3D-MESH matrix, gives a BSP algorithm with total cost $n/p + g * (n^{2/3}/p^{2/3}) +$

$L$, and, applied to the HYPERCUBE matrix, gives a BSP algorithm with total cost $(n \log n)/p + g * ((n \log p)/p) + L$. In each case we minimize the communication cost of the algorithm by partitioning the nodes of the graph into $p$ equal sized subsets in a way which minimizes the number of arcs between different subsets. Lower bounds on the efficiency of such partitions can be derived from known isoperimetric inequalities in mathematics.

Expander graphs are ubiquitous in theoretical computer science. They are sparse graphs with extremely high connectivity. Expanders have no small cuts. In geometrical terms, this means that they have very high isoperimetry. Simple counting arguments can be used to show that most 3-regular or 4-regular graphs are indeed expanders, so a random regular graph will almost certainly have high isoperimetry. For the EXPANDER matrix, the best upper bound which we have is the trivial one, $n/p + g * (n/p) + L$ which can be obtained by randomly distributing the matrix elements. As noted in McColl (1995), it can be shown that, due to its extremely high connectivity, for the EXPANDER matrix there is no partition of the nodes into $p$ equal sized subsets which gives a value for $H(n, p)$ which is less than the trivial $n/p$. Therefore, for the EXPANDER matrix, $u = M.v$ is an inherently non-local problem.

### Matrix Multiplication

We now consider the problem of multiplying two $n \times n$ dense matrices $A, B$ on $p$ processors.

For $p \leq n^2$, the standard $n^3$ sequential algorithm can be adapted to run on a $p$-processor BSP machine as follows. Each processor computes an $(n/p^{1/2}) \times (n/p^{1/2})$ submatrix of $C = A.B$. To do so it will require $n^2/p^{1/2}$ elements from $A$ and the same number from $B$. If $A$ and $B$ are both distributed uniformly across the $p$ processors, with each processor holding $n^2/p$ of the elements from each matrix, then the total time required for this algorithm will be $n^3/p + g * (n^2/p^{1/2}) + L$.

As a BSP implementation of the standard $n^3$ matrix multiplication algorithm, this simple method (Method 1) is clearly optimal in terms of its computation cost and its synchronization cost. It does, however, require memory size $M(n, p) = n^2/p^{1/2}$ which is $p^{1/2}$ times larger than the memory required on each processor to hold the input and output matrices.

We can, however, make a simple adaptation to this method in order to reduce the memory requirement. Instead of getting all $p^{1/2}$ blocks from $A$ and all $p^{1/2}$ blocks from $B$ in a single superstep, we can instead get one block from $A$ and one block from $B$ at a time, perform the block product, and then delete the $A$ and $B$ blocks to reuse the memory. In the next superstep, we get the next pair of $A$ and $B$ blocks, and so on. This adaptation has no effect on the total computation cost or on the total communication cost, but it does increase the synchronization cost. For this adaptation (Method 2), the BSP cost is $n^3/p + g * (n^2/p^{1/2}) + L * p^{1/2}$ and the memory cost is $M(n, p) = n^2/p$. So for this method we have optimal computation cost and optimal memory cost.

Yet another method (Method 3) is to map the matrix multiplication algorithm onto the 3D directed acyclic graph $C_n$ that we considered earlier. The product $C = A.B$ can be computed using the following set of definitions. For all $0 < i, j, k \leq n$,

$a_{i,j,k} = a_{i,j-1,k}$
$b_{i,j,k} = b_{i-1,j,k}$
$c_{i,j,k} = c_{i,j,k-1} + (a_{i,j,k} * b_{i,j,k})$

where $a_{i,0,k} = a_{i,k}$, $b_{0,j,k} = b_{k,j}$ and $c_{i,j,0} = 0$. These definitions can be directly translated into a labeled version of the 3D directed acyclic graph $C_n$. The BSP cost of Method 3 is therefore at most $n^3/p + g * (n^2/p^{1/2}) + L * p^{1/2}$ which is the same as Method 2. It also has the same optimal memory cost as Method 2.

All of the three methods we have described have the same communication cost $g * H(n,p) = g * (n^2/p^{1/2})$. This communication cost can, however, be reduced further if we use the 3D algorithm described in McColl (1995), which Leslie Valiant and I developed for the BSP model, although other similar methods were also developed for other models. The cost of this 3D algorithm (Method 4) is $n^3/p + g * (n^2/p^{2/3}) + L$. It is clearly optimal in terms of computation cost and synchronization cost. Moreover, an input-output complexity argument based on the isoperimetric inequality result of Loomis and Whitney (1949) can be used to show that for any BSP implementation of the standard $n^3$ sequential algorithm, if $W(n,p) = n^3/p$ then $H(n,p) \geq n^2/p^{2/3}$. So Method 4 provides a BSP realization of the standard $n^3$ matrix multiplication method which simultaneously achieves the optimal values for computation cost $W(n,p)$, communication cost $g * H(n,p)$ and synchronization cost $L * S(n,p)$. The memory requirement of this algorithm is, however, slightly inferior to Methods 2 and 3. Its memory complexity $M(n,p)$ is $n^2/p^{2/3}$. In closing we note that there is no single BSP algorithm that is optimal in all four dimensions (computation, communication, synchronization, memory). In Irony et al. (2004) it is shown that if $M(n,p) = O(n^2/p)$ then $H(n,p) = \Omega(n^2/p^{1/2})$. So any optimal memory algorithm must be suboptimal in communication. Depending on whether we wish to optimize memory or communication, we can choose Method 2 or Method 4.

As we have seen, the design, analysis and optimization of parallel algorithms relies heavily on mathematical analysis and theorems such as isoperimetric inequalities. This stems from the fact that viewed through the correct mathematical lens, we see that in the graph structure of parallel computations, the region corresponding to each processor has a surface that corresponds to the communication into and out of that processor, while the volume of the region corresponds to the operations performed by that processor.

## 1.6.3 Recursive Algorithms and Automatic Tradeoffs

For the problem of multiplying two $n \times n$ matrices $A$ and $B$, we can regard the matrices as each composed of four square $n/2 \times n/2$ submatrices. In this setting, the $n \times n$ product matrix $C = A.B$ can be computed using eight $n/2 \times n/2$ matrix multiplications and four $n/2 \times n/2$ matrix additions. These sub-problems can then be further recursively subdivided in the same way. Strassen showed that in such a recursive matrix multiplication algorithm, the number of multiplications can be reduced from 8 to 7 if we allow the number of additions to increase from 4 to 15. The effect of this change is to reduce the asymptotic complexity from $O(n^3)$ to $O(n^\alpha)$ where $\alpha = \log_2 7$.

Consider the recursive $O(n^3)$ algorithm. After $k$ levels of recursion we have $8^k$ matrix multiplication sub-problems. If we have $p$ processors then we might stop the recursion when $8^k = p$ since the number of sub-problems would match the number of processors. At that stage the size of the square submatrices would be $n/p^{1/3} \times n/p^{1/3}$. In McColl & Tiskin (1999) it is shown that the resulting recursive algorithm is essentially Method 4 above. It has optimal computation, communication and synchronization cost.

Now suppose instead that we had continued the recursion beyond that point, and stopped it when $4^k = p$. At that stage, the size of the square submatrices would be $n/p^{1/2} \times n/p^{1/2}$. In this case, McColl & Tiskin (1999) show that the resulting recursive algorithm is essentially Method 2 above. It can be realized with optimal memory cost.

If we terminate the recursion at some level $k$ where $p^{1/3} < 2^k < p^{1/2}$, then we will have a BSP algorithm that requires less memory than Method 4 and less communication than Method 2. So the results of McColl & Tiskin (1999) provides us with a single unified recursive algorithm for BSP matrix multiplication that we can use to automatically achieve a tradeoff between memory optimality and communication optimality. The results of McColl & Tiskin (1999) also apply to Strassen's sub-cubic matrix multiplication algorithm.

We noted previously that our BSP algorithm for dense matrix-vector multiplication was an "immortal algorithm" in the sense that it simultaneously achieved the optimal computation cost, the optimal communication cost, and the optimal synchronization cost, all to within small constant factors. So, it achieves the best possible parallel performance for all values of $n$, $p$, $g$ and $L$. For matrix multiplication, Method 4 above is also an immortal algorithm in this sense. However, it does require more memory than Method 2. If we redefine the term immortal to also require memory optimality then the result in Irony et al. (2004) shows that there is no such algorithm for matrix multiplication. Given that fact, the recursive algorithm in McColl & Tiskin (1999) is as close to "immortality" as can be achieved in that it can be simply and automatically tuned to deliver optimality in any of the four dimensions.

## 1.7 Networks and Communications

A large amount of work has been done in recent years on the development of efficient routing methods, on the efficient embedding of one network in another, and on the demonstration of work-preserving emulations of one network by another. We will focus our attention here on the routing problem.

We consider the problem of routing $h$-relations on a $p$-processor network. We are interested in the development of distributed routing methods in which the routing decisions made at a node at some point in time are based only on information concerning the packets that have already passed through the node at that time. In the non-distributed case where global information is available everywhere, the problem of routing is easier and well understood.

### 1.7.1 Oblivious Routing

Let us first consider deterministic methods for distributed routing. We define a routing method to be *oblivious* if the path taken by each packet is entirely determined by its source and destination. It is known (Borodin & Hopcroft 1985) that, for a 1-relation no deterministic oblivious routing method can do better than proportional to $(p^{1/2}/d)$ time steps, in the worst case, for any degree $d$ graph. The most obvious examples of deterministic oblivious approaches are greedy methods in which one sends all packets to their destination by a shortest path through the network.

For 1-relations, the performance of greedy routing on a (fixed-degree) butterfly network can be summarized as follows. All 1-relations can be realized in $O(p^{1/2})$ steps, which, as we have observed, is an optimal worst case bound for any such fixed degree network. A large number of specific 1-relations which arise in practical parallel computation, e.g. the bit-reversal permutation and the transpose permutation, provably require proportional to $p^{1/2}$ steps.

What about the "average case"? Define a *random 1-mapping* to be the routing problem where each processor has a single packet which is to be sent to a random destination. Greedy routing of a random 1-mapping on a butterfly will terminate in $O(\log p)$ steps. Moreover, the fraction of all random 1-mappings which do not finish in $O(\log p)$ steps is incredibly small, despite the fact that most of the 1-relations which seem to arise in practice do not finish in this time. We can probably conclude from these results that "typical" routing problems, in a practical sense, is a rather different concept from "typical" routing problems in a mathematical sense.

The performance of greedy routing on a $(\log p)$-degree hypercube is very similar to the case of the butterfly. All 1-relations can be realized in $O(p^{1/2}/\log p)$ steps, which is an optimal worst case bound for any $(\log p)$-degree network. For the average case, where each packet has a random destination, greedy routing will terminate in $O(\log p)$ steps. In the case of the hypercube, there are exponentially many shortest paths for a greedy method to choose from, but even randomizing among these choices still gives no better than $O(p^{\alpha})$, $\alpha > 0$, steps for many 1-relations.

### 1.7.2 Randomized Routing

We have seen that for the butterfly and hypercube, the performance of greedy routing on random 1-mappings is much better than on "worst case 1-relations", such as the bit-reversal permutation in the case of the butterfly. Around 1980, Valiant made the simple and striking observation that one could achieve efficient distributed routing, in terms of worst case performance, if one could reduce a 1-relation to something like the composition of two random 1-mappings. The resulting technique which emerged from this observation has come to be known as two-phase randomized routing (Valiant 1990*b*).

Using this approach, a 1-relation is realized by initially sending each packet to a random node in the network, using a greedy method. From there it is forwarded to the desired destination, again by a greedy method. Both phases of the routing correspond closely to the realization of a random 1-mapping. Extensive investigation of this method, in terms of the number of steps required, size of buffers required etc., has shown that it performs extremely well, both in theory and in practice. Using randomized routing one can show that with high probability, every 1-relation can be realized on a $p$-processor butterfly, 2D array and hypercube in a number of steps proportional to the diameter of the network. For these fixed-degree networks, this result is essentially optimal. For the ($\log p$)-degree hypercube network, the following even stronger result can be obtained. With high probability, every ($\log p$)-relation can be realized on a $p$-processor hypercube in $O(\log p)$ steps. Randomized routing can also be used to achieve good worst case performance on other networks such as the shuffle-exchange network, the cube-connected-cycles, and on fat trees.

An interesting theoretical alternative to using randomized routing on a standard, well-defined network such as a butterfly, is to use deterministic routing on a “randomly wired network”. In Leighton & Maggs (1989), Upfal (1992) it is shown that a simple deterministic routing algorithm can be used to realize a 1-relation in $O(\log p)$ steps on a randomly wired, bounded degree network known as a multi-butterfly. An important feature of multi-butterflies is that they have powerful expansion properties. In addition to permitting fast deterministic routing, such expander graphs also have very strong fault tolerance properties.

### 1.7.3 Networks, Routing and BSP

The use of the parameters $L$ and $g$ to characterize the communications performance of a BSP computer contrasts sharply with the way in which communications performance is described for most distributed memory architectures produced and sold today. A major feature of the BSP model is that it lifts considerations of network performance from the local level to the global level. We are thus no longer particularly interested in whether the network is a 2D array, a butterfly or a hypercube, or whether it is implemented in VLSI or in some optical technology. Our interest instead is in the global parameters of the network, such as $L$ and $g$, which describe its ability to support non-local data communications in a uniformly efficient manner.

In the design and implementation of a BSP computer, the values of $L$ and $g$ which can be achieved will depend on the capabilities of the available technology and the amount of money that one is willing to spend on the communications network. As the computational performance of machines continues to grow, we will find that to keep $L$ and $g$ low it will be necessary to continually increase our investment in the communications hardware as a percentage of the total cost of the machine. In asymptotic terms, the values of $L$ and $g$ one might expect for various p-processor networks are as shown in Table 1.1 (ignoring small constant factors).

These asymptotic mathematical estimates are based on the degree and diameter properties of the corresponding graph, and on the use of a fast routing method such as randomized routing. In a practical setting, the channel capacities, routing methods used, physical implementation etc. would also have a significant impact on the actual values of $L$ and $g$ which could be achieved on a given machine. New optical technologies may offer the prospect of further reductions in the values of $L$ and $g$ which can be achieved,

**Table 1.1** Network $L$ and $g$ values

| Network | $L$ | $g$ |
|---|---|---|
| Ring | $p$ | $p$ |
| 2D Array | $p^{1/2}$ | $p^{1/2}$ |
| Butterfly | $\log p$ | $\log p$ |
| Hypercube | $\log p$ | 1 |
| Completely Connected | 1 | 1 |

by providing a more efficient means of non-local communication than is possible with VLSI.

Choosing the right values of $L$ and $g$ for a network architecture will depend on the class of computations that the parallel machine needs to support. If an application has a BSP cost $W + g * H + L * S$ then a network architecture where $g$ is less than $W/H$ and $L$ is less than $W/S$ will ensure that the machine is reasonably balanced for that application, and that the utilization of the available computation capacity will be acceptable. Of course, the lower the values of $L$ and $g$, the higher the overall utilization will be, up to a point. However, attempting to achieve the lowest possible $L$ and $g$ will not, in general, be worthwhile, as the cost involved will increase dramatically, and may only provide a minimal increase in performance. Simple mathematical analysis, using the BSP cost model, can easily determine all of these tradeoffs in a precise way.

## 1.8 Resilience-Oriented Models

Resilient, predictable architectures are essential for large-scale parallel computation. As scale increases, the number of faults and long tails increases correspondingly. We therefore need a model that can efficiently handle fault tolerance and tail tolerance at scale. In 2017, a new BSP-based bridging model was developed that, for the first time, solves this problem. This new Cloned Computing model supports high-performance parallel computing with fault tolerance and tail tolerance (McColl 2018).

The model is general purpose – it applies to all forms of large-scale parallel computing, including communication-intensive, highly iterative computations. It provides a new, simple model enabling large-scale parallel software to be run with automatic fault and tail tolerance, with no program changes for load balanced computations, and only a simple Boolean parameter addition for non-load-balanced computations. It also provides an accurate cost model enabling large-scale parallel software to be automatically optimized for any architecture. Finally, the model enables high performance nonstop and realtime parallel computation. Unlike checkpointing-based recovery, the new model enables computations to run continuously without interruption and meeting realtime requirements, with high probability. Below we will provide an overview of this new model.

## 1.8.1 Fault and Tail Tolerance

In many scenarios in large-scale parallel computing, it is not possible to simply run the computation again, or to use simple methods such as checkpointing, to handle faults or long latencies (tails). For example, with petaflop and exaflop computations it will be very costly to "just run again or use checkpointing". Another example is where we are running infinite continuous nonstop computations. In such a case, "just run again or use checkpointing" is impossible, as it is in the case where we are running realtime computations. Large-scale parallel computing requires a model that

- can be used to efficiently run any parallel computation, at any scale,
- can be used on cost-effective commodity architectures,
- can be used to efficiently run computations continuously, without interruptions,
- offers high performance,
- offers high availability, with automatic fault tolerance and tail tolerance, self-healing and self-optimizing.

Cloud computing architectures offer parallelism, scale and cost-effectiveness, but unfortunately today's cloud architectures are very unpredictable. Maximum latency can often be more than 100 times greater than average latency for identical tasks. In some cases, the multiple can even be 1000 times or more, as shown in Figure 1.7.

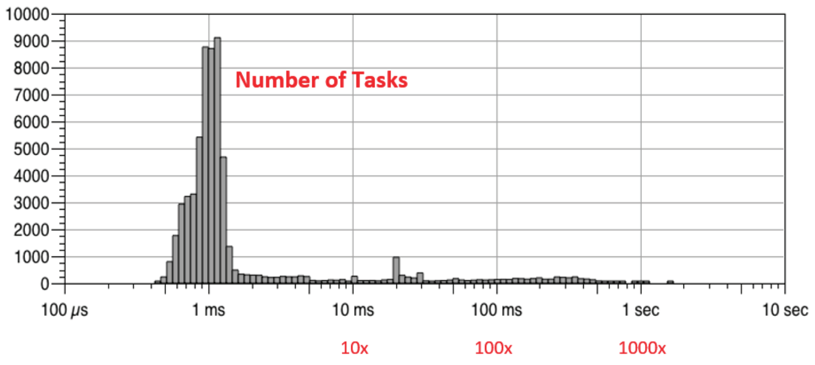


**Figure 1.7** Tail latency

These long tails are a major problem, and quite different from the related problem of handling faults. However, with modern software container technology, containers can be relaunched very quickly – in seconds rather than the minutes normally required to relaunch a virtual machine or physical server. So containers provide a means of restarting computations quickly, but there remains the challenge of deciding when to restart.

In this new era of large-scale computing with frequent long tails, traditional checkpointing-based approaches to recovery are inadequate for a variety of simple reasons:

- The latency of writing to, and reading from, resilient storage is very high.

- We need to choose frequent checkpointing or long recovery times – both are very bad, and there is no good tradeoff.
- We need to choose frequent recovery or long tail limits – both are very bad.
- With checkpointing, nonstop performance or continuous realtime performance are both impossible, due to stopping and restarting.

Are there alternatives to checkpointing for parallel computing with fault tolerance and tail tolerance?

Previous work at Google and UC Berkeley has shown that this can be achieved for a certain class of simple tree-structured, data parallel, constant-round parallel computations, e.g. MapReduce computations that use disk-based data communications systems such as Hadoop HDFS, and that have a small number of tasks and execute in a small number of rounds. This work was an important first step, but did not consider the following:

- General purpose parallel computations that may have thousands of tasks, and may execute for thousands of rounds.
- The challenge of supporting long or continuous parallel computations that need to run nonstop over a huge number of rounds.
- Computations where almost all tasks are costly, rather than just a small percentage of tasks.
- General purpose parallel computing with high performance point-to-point data communications systems such as BSP, MPI, RDMA.

The new Cloned Computing model for general purpose parallel computing addresses all of these challenges and offers all of the following:

- A solution for all large-scale parallel computations, including those that are general dataflow graphs, or communication-intensive, or highly iterative, multi-round.
- Scalability, high performance, cost-effectiveness, and high availability (automatic fault tolerance and tail tolerance).
- A cost model and automatic optimization for general purpose parallel computations run on large-scale commodity architectures where faults and long tails are common.

### 1.8.2 The Cloned Computing Model

Many modern large-scale parallel computing applications are both highly iterative and communication-intensive. For example, Big Data Analytics, Graph Computing, Machine Learning, AI, HPC, Modeling, Genomics, Network Optimization, Simulation. These applications mostly "compute in rounds" irrespective of whether the actual software they are written in is MPI, BSP, MapReduce, Spark, Pregel, Giraph, or other parallel programming models and systems. This BSP-style superstep parallelism has proven to be capable of delivering the highest levels of performance for all kinds of applications. Moreover, many of the most important modern large-scale commercial parallel applications such as Machine Learning, AI, Network Optimization, and Graph Analytics, are highly iterative, involving thousands of rounds, and can be very naturally and easily expressed as BSP computations.

The Cloned Computing model extends the BSP model in a major way to support high performance fault tolerance and tail tolerance. An implementation of the model runs parallel programs that compute in rounds using a new execution model which for convenience we will call Nonstop BSP. A Nonstop BSP program has four core features:

- It is a BSP program and is structured to compute in rounds. In each round, each of the processes computes on data in local memory, globally communicates across the network, synchronizes.
- The number of processes in a program is the parallelism parameter $p$.
- The number of rounds in a program is the parameter $R$, which may be finite or infinite.
- Each process has a BSP synchronization mechanism at the end of each round. This mechanism is parameterized by a (typically Boolean) value indicating whether or not the process is one that can set the minimum time value for the round. This parametric value can be changed dynamically during the execution of a program. If a program has the parameter set so that no process can set the minimum time value in any round then the cloned computing system will execute the program as a normal BSP program without fault tolerance or tail tolerance.

In Figure 1.1 we showed a simple example of a non-load-balanced BSP program with four processes. For that program, we might have an expected time per round of 4 seconds for $p_0$, $p_1$ and $p_2$ but only 100ms for $p_3$. In order to turn this into a Nonstop BSP program for execution, the only change required is to parameterize the synchronization primitives in order to show whether or not the associated process is one that can set the minimum time value for the round. So, for this example, we might set the Sync parameter to False for $p_3$, and to True for the others, as shown in Figure 1.8.

The following provides a simple high level overview of a cloned architecture with $P$ processors, running a $p$-process parallel program, where $p \leq P$.

- Each of the $P$ processing elements can run one or more of the processes during a single round.
- Multiple instances of a process are referred to as clones.
- Each of the $p$ processes is run as the first process on at least one of the $P$ elements.
- The first process to be run on an element is the only one that can possibly set MinTime for the round.
- During each round, if the first process to be run on each element has its synchronization mechanism set indicating that it can set MinTime, then it monitors its elapsed time for the current round (Nonstop BSP) and attempts to write MinTime when it ends. The first such process to write its time sets MinTime for round.
- Each process can have access not only to its own local data and state, but also to other information including
  - A copy of MinTime for the round;
  - Its elapsed time for the round;
  - Which clones of other processes it may need to communicate with;

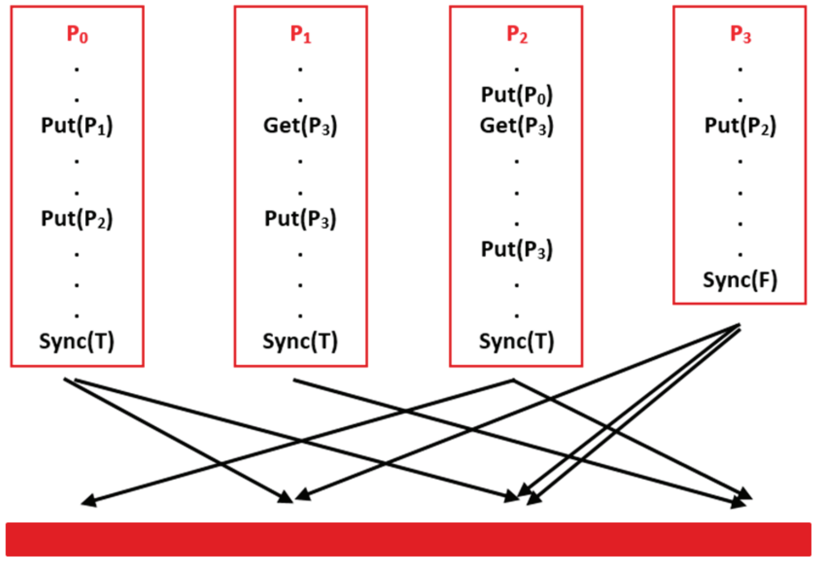


**Figure 1.8** Superstep with Sync parameter

  – Standby resources available;
  – Where other clones of the process are located.

- Cloned computing architectures have a polynomial “Predictability” exponent $D$, indicating that an expected fraction $P/t^D$ of the processing elements will fail to complete any round in less than $t$*MinTime.
- The cloned computation has a TailLimit $T$. Any process that fails to complete before $T$*MinTime is marked as a fault/tail, others are marked as live.
- A round is successful if at least one clone of each of the $p$ processes completes.
- The inter-process communications can be handled in several ways. For example, the first clone of a process to complete can handle the global communications for all the clones of that process. A number of other variations are also possible, depending on other objectives such as balancing communications at endpoints, increasing network latency resilience and other factors.
- Process faults and tails can be handled by transferring state from another live clone of the same process. To improve the speed with which this state transfer and relaunch can be achieved, it may be convenient to maintain a pool of spare containers that are ready to run. This can be done in several ways. For example, by having a static pool of containers directly associated with the various processes, or by having a dynamic pool of containers each of which can be used with any process. The choice between having a static pool, a dynamic pool, or no pool of spare containers can be made based on a tradeoff between speed of relaunch and efficiency of container utilization.

### 1.8.3 Vertical and Horizontal Cloning

To ensure fault tolerance and tail tolerance, processes can be cloned vertically or horizontally, or both. We noted above that in a cloned architecture with $P$ processing elements running a $p$-process parallel program, where $p \leq P$.

- Each of the $P$ processing elements can run one or more of the processes during a single round.
- Multiple instances of a process are referred to as clones.
- Each of the $p$ processes is run as the first process on at least one of the $P$ elements.

Let's consider first the case where $p = P$, and look at vertical cloning. Suppose $p = 10$, and we have processes numbered from 0 to 9 as shown in Figure 1.9.

0
1
2
3
4
5
6
7
8
9

**Figure 1.9** Processes 0 to 9

Let us further assume that the program is perfectly load balanced, so that every process takes exactly the same time as all the others, in every round. Then with no faults or tails we expect that all processes will complete the round at the same time, as shown in Figure 1.10.

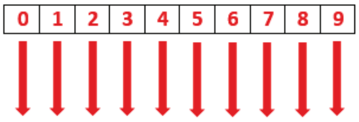


**Figure 1.10** Processes 0 to 9 with no faults or tails

If, however, there are faults or tails then the processes may take quite different amounts of time, as shown in Figure 1.11.

To address this challenge we can use vertical cloning, where multiple process instances run consecutively on the same processing element (no extra elements required), as shown in Figure 1.12.

In the example shown, each process appears twice, but vertical cloning can be used with any multiple. The multiples do not even need to be uniform, although this may normally be the case. For example, we might have the cloning shown in Figure 1.13 where each process has three additional clones.

In such uniform cases, we say that the degree of vertical cloning is the parameter $VC$. In the example shown in Figure 1.13, we have $VC = 4$. The first row of processes are the only ones that can try to set MinTime for the round, and can do so only if their synchronization parameter is set accordingly.

Next we introduce the idea of horizontal cloning. Again suppose $p = 10$, and we have processes numbered from 0 to 9 as shown in Figure 1.9. As in the case of vertical

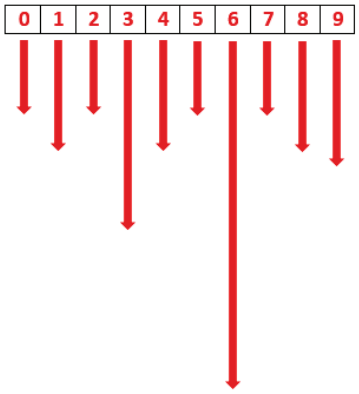


**Figure 1.11** Processes 0 to 9 with tails

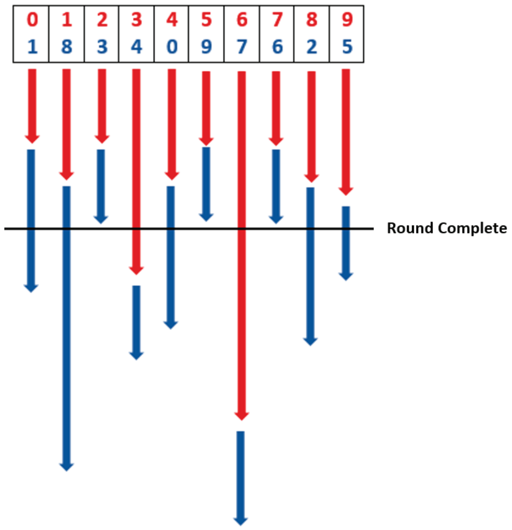


**Figure 1.12** Processes 0 to 9 with vertical cloning

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 1 | 8 | 3 | 4 | 0 | 9 | 7 | 6 | 2 | 5 |
| 7 | 5 | 4 | 9 | 1 | 6 | 8 | 0 | 3 | 2 |
| 2 | 3 | 6 | 1 | 7 | 4 | 0 | 5 | 9 | 8 |

**Figure 1.13** Vertical cloning

cloning, let us further assume that the program is perfectly load balanced, so that every process takes exactly the same time as all the others, in every round. Instead of running multiple process instances on the same element, we can instead run multiple instances of processes on different elements concurrently, by using more processing elements, as shown in Figure 1.14.

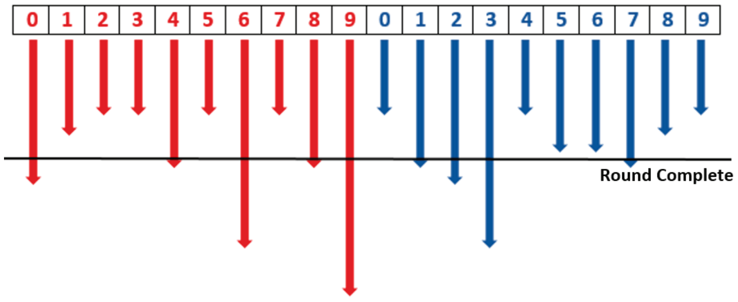
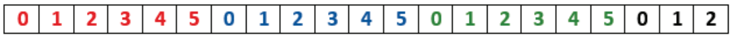


**Figure 1.14** Horizontal cloning

In the example shown in Figure 1.14, each process appears twice, but horizontal cloning can be used with any multiple. The multiples do not need to be uniform. For example, for $p = 6$ and $P = 21$ we might have the cloning structure shown in Figure 1.15.

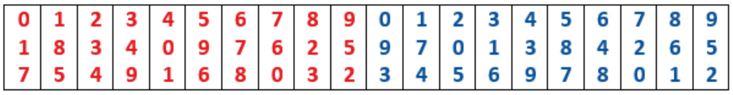


| | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 0 | 1 | 2 | 3 | 4 | 5 | 0 | 1 | 2 | 3 | 4 | 5 | 0 | 1 | 2 |

**Figure 1.15** Horizontal cloning with HC=3.5

We say that the level of horizontal cloning is the parameter $HC = P/p$. In the examples shown in Figures 1.14 and 1.15, the values of $HC$ are 2 and 3.5.

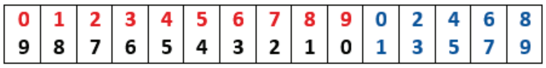
Vertical and horizontal cloning can be combined. For example, we might have $VC = 3$ and $HC = 2$, as shown in Figure 1.16.



| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 1 | 8 | 3 | 4 | 0 | 9 | 7 | 6 | 2 | 5 | 9 | 7 | 0 | 1 | 3 | 8 | 4 | 2 | 6 | 5 |
| 7 | 5 | 4 | 9 | 1 | 6 | 8 | 0 | 3 | 2 | 3 | 4 | 5 | 6 | 9 | 7 | 8 | 0 | 1 | 2 |

**Figure 1.16** Horizontal cloning with VC=3 and HC=2

As another example, we might have $VC = 2$ and $HC = 1.5$, as shown in Figure 1.17.



| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 | 2 | 4 | 6 | 8 |
| 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 1 | 3 | 5 | 7 | 9 |

**Figure 1.17** Horizontal cloning with VC=2 and HC=1.5

As in the case of simple vertical cloning, the first row of processes are the only ones that can try to set MinTime for the round, and can do so only if their synchronization parameter is set accordingly.

### 1.8.4 Resilience

Cloning is a powerful way of increasing the resilience of large-scale parallel computing architectures. To see, for example, the power of horizontal cloning, consider the simple case of a cloned computation in which $VC = 1$ and $HC = 2$, i.e. we have two copies of each process running concurrently. For example, when $p = 10$, we have the arrangement shown in Figure 1.18.

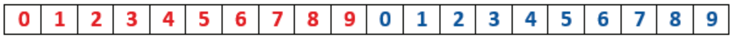


**Figure 1.18** Horizontal cloning with VC=1 and HC=2

There are at least two ways in which we can implement such an arrangement. The traditional approach would be simply to run two separate disjoint copies of the 10-process computation, with no significant communication between them, as shown in Figure 1.19. This is often the approach taken in fault tolerant computing, where separate redundant copies of a computation are run to increase resilience against failures.

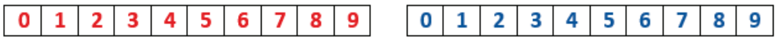


**Figure 1.19** Disjoint redundancy

With this new cloned computing model, we have the opportunity to run both computations together as a single combined computation, executing in rounds, with communications possible between any of the $2p$ processes, and with automatic fault tolerance and tail tolerance. The cost will be approximately the same as the disjoint redundant computation, but as we will see, the computation will be significantly more resilient. Consider first the disjoint computation, consisting of two separate $p$-process computations, which we will refer to as Left and Right. This is shown in Figure 1.20.

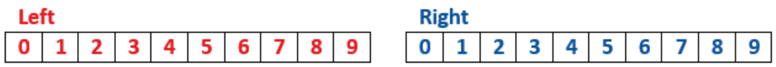


**Figure 1.20** Disjoint Left and Right computations

If, in any round, a process in Left and a process in Right both experience a fault or long tail, then the computation will have to stop. Note that it does not have to be the same process. For example, if process 4 in Left and process 2 in Right both experience a fault or long tail, then the computation has to stop. Now consider the combined cloned computation of the new model, as shown in Figure 1.18.

With this new model, as long as the two copies of any process do not both experience a fault or long tail during the same round, there is no need to stop the computation. For example, if process clones 0,3,4,6 from the Left group and process clones 1,5,8,9 from the Right group all experience a fault or long tail during the same round, as shown in Figure 1.21, then the computation can still continue without interruption.

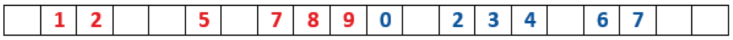


**Figure 1.21** Cloned computation with faults and long tails

For the disjoint computation, the probability of the $2p$-process system surviving $f$ failures is $2 * \binom{p}{f}/\binom{2p}{f}$. In the combined cloned computation, the probability is $2^f * \binom{p}{f}/\binom{2p}{f}$. The computation is therefore significantly more reliable than the disjoint computation.

For $p = 1000$ and $f = 2$, the disjoint computation survives with probability close to 0.5, whereas the combined computation survives with probability 0.9995. For $p = 10,000$ and $f = 2$, the disjoint computation survives with probability close to 0.5 and the combined one survives with probability 0.99995. Cloned computations get more and more resilient as the scale of the system increases.

The model can be easily extended to higher levels of replication. If, instead of two copies we use $r$ copies, then the probability of a cloned computation surviving $f$ faults or long tails, is given by the following formula

$$\text{Availability}(r, p, f) = 1 - N(r, p, f)/\binom{rp}{f}$$

where

$$N(r, p, f) = 0 \text{ if } p < 1 \text{ or } f < r$$

and

$$N(r, p, f) = \binom{r * (p-1)}{f-r} + \sum_{i=0}^{r-1} \binom{r}{i} * N(r, p-1, f-i).$$

In the above formulae, $r$ is the horizontal clone level $HC$, $p$ is the parallelism, and $f$ is the number of failures.

## 1.8.5 Cloned Computing Cost Model

The model has a cost model that enables optimal parameters to be calculated to achieve optimal performance, ensure nonstop resilience, and to allow tuning to achieve the best possible tradeoffs between performance, cost and resilience. The following is a list of the parameters that together make up this new cost model.

- Program Parameters:
  - Parallelism $p$;
  - Rounds $R$.
- Network Parameters:
  - Network Throughput $g$;
  - Network Latency $L$.

- Cloned Computing Parameters:
  - Predictability $D$;
  - TailLimit $T$;
  - VerticalCloneLevel $VC$;
  - HorizontalCloneLevel $HC$;
  - StandbyLevel $S$.

Given $D$, $p$, $R$ we can automatically compute the most appropriate values for $T$, $VC$, $HC$, $S$ to optimize performance and ensure nonstop resilience. The cost is at most $T * HC *$ Perfect, where Perfect is the cost for an idealized architecture where $D$ is infinite (no failures or tails ever). We can place this new cost model at the top level of a hierarchy of cost models for different models of parallel computing, including PRAM (idealized shared memory), MapReduce, and standard BSP with no fault tolerance or tail tolerance.

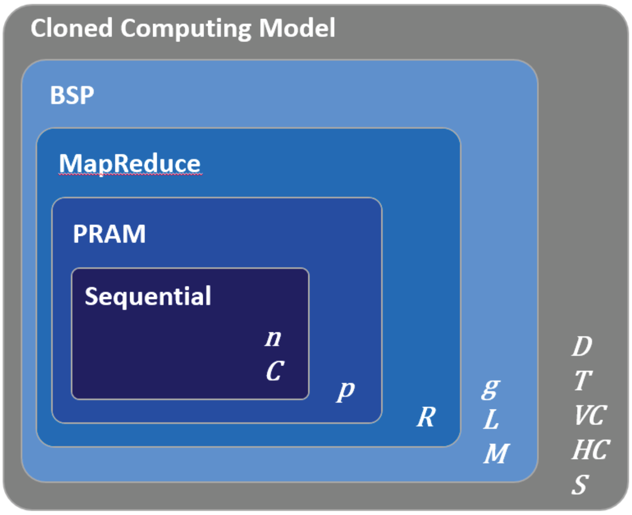


**Figure 1.22** Cost model hierarchy

The model parameters in the cost model hierarchy shown in Figure 1.22 are as follows:

- $n$ = Problem Size.
- $C$ = (Sequential) Complexity.
- $p$ = Parallelism.
- $R$ = Rounds.
- $g$ = Network Throughput.
- $L$ = Network Latency.
- $M$ = Local Memory.
- $D$ = Predictability.
- $T$ = TailLimit.

- $VC$ = VerticalClone Level.
- $HC$ = HorizontalClone Level.
- $S$ = StandbyLevel.

For the cloned computing model:

- The parameters $p$, $R$ and data distribution are decided by the programmer.
- The parameters $g$, $L$, $M$, $D$ are parameters determined by the infrastructure (hardware+software).
- Using the cost model, optimal $T$, $VC$, $HC$, $S$ parameters can be automatically calculated to guarantee a given Quality of Service Level (performance and resilience).

So, the model, in addition to providing high-performance general-purpose parallel computing with fault tolerance and tail tolerance, is also easy to use. Any large-scale parallel software can be automatically run as a cloned computation. Just add a Boolean to sync, and make it always true if you have a load balanced program. Also, any large-scale parallel software can be automatically optimized for cloned execution. Given program Parallelism $p$ and Rounds $R$, and Predictability $D$, we can automatically generate the optimal values of TailLimit, VerticalClone level, HorizontalClone level, StandbyLevel.

### 1.8.6 Why Cloned Computing?

**General purpose.**
Applies to all forms of large-scale parallel computing, including communication-intensive, highly iterative computations. Previous approaches only covered a very small class of MapReduce parallel computations.

**Automated.**
Provides a new, simple programming model (Nonstop BSP) enabling large-scale parallel software to be automatically run on the fault tolerant and tail tolerant system with no changes for load balanced computations, and only a simple Boolean parameter addition for non-load-balanced computations. The new model provides an accurate cost model enabling large-scale parallel software to be automatically optimized for any infrastructure.

**High Performance. Nonstop. Predictable.**
The new model enables high performance, nonstop, predictable, and realtime parallel computation. Unlike checkpointing-based recovery, it enables computations to run continuously without interruption and meeting realtime requirements, with high probability.

The simple diagrams in Figure 1.23 show the potential impact of this new model in reducing the time taken to perform five rounds of a load balanced parallel computation.

### 1.8.7 Coded Computing

As we have seen, cloned computing provides a solution to the resilience and predictability problem that can be applied to any parallel computation. This goes far beyond previous

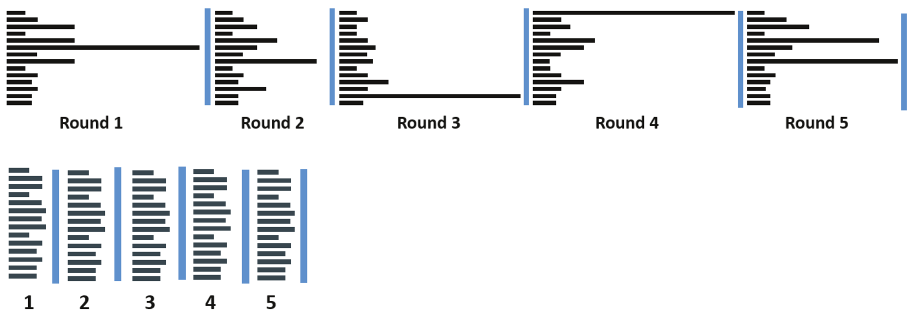


**Figure 1.23** Impact of cloned computing

results that only applied to simple tree-structured parallel computations such as MapReduce and other related forms of master-worker distributed computing. Another approach to the resilience and predictability problem that has been explored recently is "Coded Computing" (Yu et al. 2019). This is aimed at the more limited range of tree-structured parallel computations, but targets not only the tail latency (straggler) problem but also issues of communication, security and data privacy. It can be seen as a complementary approach to cloned computing, addressing a narrower scope of computations but a broader set of issues.

As noted in the discussion of cloned computing, the performance of a modern parallel and distributed systems is significantly affected by anomalous system behavior and bottlenecks – a form of "system noise". Given the individually unpredictable nature of the nodes in these systems, we are faced with the challenge of achieving fast results in the face of massive uncertainty.

One way of tackling this challenge is using coding theoretic techniques. The role of codes in providing resiliency against noise has been studied for decades in several other engineering contexts, and is part of our everyday infrastructure (smartphones, laptops, WiFi and cellular systems etc.). Coding is also now routinely used to transform the storage layer of distributed systems in modern datacenters supporting regenerating with locally repairable codes for distributed storage.

The basic idea of coded computing is to be able to compute in a redundant way so that the computation can be completed even without collecting the results from the stragglers. The redundancy in coded computation can be used not only to address the straggler problem in master-worker computations, but also communication bottlenecks. Coded computing has been used in a number of tree-structured master-worker parallel computations, including FFT, Matrix Multiplication, Machine Learning, and MapReduce. The method is based on the use of Lagrange interpolation polynomials and related mathematical techniques.

## 1.9 New Research Directions

As we have seen, despite more than 40 years of effort, we still do not have a universal model of computation that can handle all of the standard algorithmic structures and patterns that it needs to support, and all of the other requirements. No single model has yet demonstrated that it is satisfactory in all dimensions. We are still looking for a model that can address the spatial and temporal decomposition of computations on current and future heterogeneous massively parallel architectures.

### 1.9.1 Research Challenges for Current Models

The dataflow approach (such as TensorFlow) has been tried many times over the past 40 years, and at modest scale can be useful. However, the optimization of dataflow computations at large scale remains a major challenge. Also, as it is a relatively low-level programming model, the productivity of programmers is low, especially when trying to achieve high performance at scale with dataflow models.

The functional /algebraic approach (such as GraphBLAS) offers considerable potential in terms of improving software productivity, as it provides a flexible high-level functional abstraction that makes programming easier via higher-order functions. The main challenge in this area is to provide the necessary software automation to ensure that the required load balancing, communication-efficiency, synchronization-efficiency, etc., can be achieved without the programmer having to specify how it should be done at scale on complex heterogeneous parallel architectures.

The BSP model has clearly demonstrated massive scalability and very high performance at scale. It has also provided a foundation for other models that have come later, such as Data-Centric BSP (Pregel and many others) and nonstop resilient BSP (Cloned Computing). Data-centric BSP is now routinely used today at many companies for data analytics on graphs with billions of nodes and trillions of edges. It is also used in world-leading commercial graph database systems. Being a lower-level abstraction than the functional approach means that the programmer has potentially more flexibility, but as always this comes at the price of increased programming complexity.

As research on BSP-based models has shown, the challenges in developing a model and associated framework are not only concerned with the programming model or style. Indeed, as the programming model becomes simpler and more abstract in order to improve software productivity, these other aspects become more and more important. For example, it is easy to propose an appealing model where the programmer is presented with a simple algebraic programming abstraction based on higher-order functions. The real challenge in such a case is that any viable implementation of the model needs to offer not only a scalable high performance low-level execution engine, but also a range of powerful new technologies for cost-model driven software automation that together enable the highest levels of scalability and performance to be achieved. We need to provide not only a run-time layer for scalable heterogeneous parallel computing, but also the parallelization and optimization layer.

Can a new model be developed that can combine the software productivity advantages of simple functional models, with the scalability, performance and resilience benefits of lower-level BSP-based models? Or is there something else that is even better that we have just not thought of yet? That remains a research challenge for the future.

It is remarkable to reflect that the early work of von Neumann in the 1940s and, before that, Turing in the 1930s, provided us with a universal model of sequential computation that has endured to this day. The stability that it provided has been essential to the remarkable growth of the global software and hardware industries over the past 60-70 years. Despite the desire for a post-von Neumann model, we are not there yet.

### 1.9.2 Scale Simplifies

Before looking at specific new research directions, it is perhaps appropriate to consider a very simple and fundamental question. Does increasing scale increase the complexity of the challenge of producing a model that can achieve all the various aspects required – Universality, Scalability, Performance, Portability, Predictability, Analyzability, Productivity etc.? Or does scale simplify the problem?

It is natural to think that increasing scale will make problems more complex and challenging, but paradoxically, as scale increases, fewer and fewer of the approaches that work at small scale can be used. An expert programmer can attempt to explicitly define every aspect of a system with a few cores. However, when the number of cores is one million or more, this becomes an unrealistic task, unless a large part of the process is uniform and automated. So, at massive scale, we can expect that few models will work, and those that do will have to be simple, uniform and highly automated. So where should we be looking for new directions?

### 1.9.3 Communication-Light Models

As we have seen, many computational problems are communication-light. In BSP terms, these are problems with a cost $W + g * H + L * S$ where $H$ and $S$ are much smaller than $W$. For example, there are many important computational problems that are "embarrassingly parallel", i.e. where $H$ and $S$ are constant for any problem size, and in some cases may be zero. A simple example of such a communication-light problem would be data-parallel video encoding.

There are also many important problems where $S$ is constant or small, and where $H$ grows much more slowly than $W$. For example, we have many "high arithmetic intensity" parallel matrix computations where $W = O(n^3/p)$ but $H$ is only $O(n^2/p^{1/2})$ or less. In these communication-light scenarios, a number of the models we have considered will provide good scalability and performance. We can certainly use BSP or message passing, but other less flexible and more automated models such as MapReduce and Spark can also be used in many such cases. Using simpler models in these cases also allows us to more easily provide resilience with automated fault tolerance.

For communication-light parallel computation, the most extreme model is perhaps the disaggregated "serverless" architecture. Serverless computing (Jonas et al. 2019) has been offered in recent years by most cloud computing vendors. It is often referred to as "Function-as-a-Service" computing.

The central idea behind the serverless model is that, instead of cloud users renting server capacity by the hour and storage capacity by the GB, the cloud vendor instead allows them to launch and execute large numbers of short-lived stateless function calls and tasks. Users are billed based on the number of such calls executed, which may be hundreds, thousands, or millions per hour, depending on the user's needs.

This model has proved to be very popular, both with cloud vendors and with users. Cloud vendors can focus on optimizing their infrastructure to efficiently support serverless computing. They can control the launching, scheduling, placement, and termination of the function calls. Users, on the other hand, are able to focus on their business or applications without the need to manage increasingly complex computing infrastructure. Users also benefit from only paying for the resources they actually use, rather than the resources allocated. Users do not pay for any idle capacity that is wasted, and vendors work hard to ensure that any amount wasted is small.

Since serverless computing involves only stateless function calls and tasks, any data sharing that is required needs to be handled by the storage system rather than by direct communication between the functions/tasks. The natural cloud architecture to support serverless computing is therefore one in which one has a pool of computation resources and a separate pool of storage resources. One major benefit of this kind of "disaggregated" architecture is that each of the pools can be scaled, allocated, managed and priced independently, greatly reducing the cost required to set up and operate such an infrastructure. On the other hand, as in MapReduce architectures, any computation that is not communication-light will have low performance due to the constant need to communicate indirectly by reading from and writing to the remote storage pool.

In order to achieve the full potential and benefits of serverless computing for both cloud vendors and users, we need new cloud hardware and software architectures that:

- Minimize the overheads involved in launching and running short-lived containers and other forms of lightweight virtualization sandbox;
- Ensure strict multi-tenant isolation;
- Support massive data movements (data shuffling) between compute pools and remote storage pools with powerful new fabrics offering high bandwidth and low latency at scale;
- Support high performance fine-grained storage services at scale;
- Automate optimal resource allocation across heterogeneous compute pools (CPUs, GPUs, accelerators) based on static code analysis, profiling, and other mechanisms;
- Minimize sub-second-level tail latencies for predictable performance.

### 1.9.4 Communication-Heavy Models

While serverless computing may be an option for communication-light parallel computations, it is unfortunately the case that many of the important practical parallel computations are communication-heavy. Examples include irregular sparse matrix and graph computations and iterative machine learning computations. For such computations, the BSP cost $W + g * H + L * S$ may have values of $H$ that are almost as large as $W$. We therefore require an architecture with a very low value of $g$ in order to achieve any significant degree of performance from the computation resources.

As the number of cores in modern parallel computing architectures keeps growing, the challenge of keeping the value of $g$ low becomes more and more difficult, and as a result, the architectures become increasingly unbalanced. The result of that imbalance is low utilization of the available compute resources. This problem manifests itself in many forms, but is often simplistically referred to as the "memory wall" problem.

On one level, the solution to the problem is equally simple. We need to spend more on communications and memory bandwidth, and spend relatively less on cores, aiming to produce more balanced architectures. However, without other changes in computer architecture it may be difficult to achieve the balance required in this way alone.

An unfortunate consequence of the success of the von Neumann model is that our global hardware industry is organized in a way that separates out the different parts of an architecture. We have some companies that focus on producing the compute elements, e.g. CPUs and GPUs. Other companies focus on producing the memory elements, while others still focus on storage systems. Finally, some focus only on interconnects and communications.

In order to build a successful, balanced architecture for communication-heavy applications, it is of course necessary to tightly aggregate all of these disjoint elements together in the best possible way.

Fortunately, in the next few years, each of these four sectors of our hardware industry will change in ways that will open up new ways of building such tightly aggregated architectures. In some areas, this is already underway.

- **Memory**. In the past this was simply to store data during a computation. In the future we will have In-Memory Computing (or Processing-in-Memory) that will allow computation within the memory. Possible functionality might include Non-Unit Strided Access, Pointer Chasing, Cache Hierarchy Collapse, Bulk Data Copy and Initialization, Bulk Bitwise Ops.
- **Interconnect**. In the past this was simply to move messages and data from one processor to another. In the future we will have In-Network Computing (Processing-in-Network) that will allow computation inside the network. Possible functionality might include Network Offloading, Combine/Reduce/ Aggregate, Arithmetic/Logical Ops, Consensus/Agreement, In-Network Caches.
- **Storage**. In the past this was simply to store data in a non-volatile, durable and persistent manner. In the future we will have In-Storage Computing (Processing-in-Storage) that will allow computation within the storage system. Possible functionality might include Deduplication, Compression, Near-Data Query Processing.
- **Compute**. In the past this was simply to perform arithmetic and logical computation. As the number of cores increases in CPUs, GPUs and accelerators, many compute elements increasingly now also have on-chip memory and on-chip networks.

So the four elements of a von Neumann computer architecture – Compute, Memory, Storage and Interconnect are all increasingly coming together and enabling a new future era of "Unified Computing" architectures.

These future unified non-von-Neumann architectures, in addition to tightly integrating the four types of elements, will also exploit advanced mathematics, algorithms, optimization and machine learning to ensure that they operate with high performance, and constantly adapt and improve wherever possible.

## References

Backus, J. (1978). Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, **21**(8) pp. 613–641.

Borodin, A. & Hopcroft, J. E. (1985). Routing, merging, and sorting on parallel models of computation. *Journal of Computer and System Sciences*, **30**(1) pp. 130–145.

Buluç, A., Mattson, T., McMillan, S., Moreira, J. & Yang, C. (2017). Design of the GraphBLAS API for C. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 643–652.

Church, A. (1941). *The Calculi of Lambda-Conversion*. Volume 6 of Annals of Mathematics Studies, Princeton University Press.

Curry, H. & Feys, R. (1958). *Combinatory Logic. I*, North-Holland Publishing Company.

Dean, J. & Ghemawat, S. (2004). MapReduce: Simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation*.

Dean, J., Monga, R. et al. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. *Google White Paper* .

Dennis, J. B. (1974). First version of a data flow procedure language. In *Programming Symposium, Proceedings Colloque sur la Programmation, Paris, France*, pp. 362–376.

Hill, J. M. D., McColl, W. F., Stefanescu, D. C., Goudreau, M. W., Lang, K., Rao, S. B., Suel, T., Tsantilas, T. & Bisseling, R. H. (1998). BSPlib: The BSP programming library. *Parallel Computing*, **24**(14) pp. 1947–1980.

Hoare, C. A. R. (1978). Communicating sequential processes. *Communications of the ACM*, **21**(8) pp. 666–677.

Irony, D., Toledo, S. & Tiskin, A. (2004). Communication lower bounds for distributed-memory matrix multiplication. *Journal of Parallel and Distributed Computing*, **64** pp. 1017–1026.

Jonas, E. et al. (2019). Cloud programming simplified: A Berkeley view on serverless computing. Technical Report No. UCB/EECS-2019-3, EE and CS Dept., University of California at Berkeley.

Leighton, F. T. & Maggs, B. M. (1989). Expanders might be practical: Fast algorithms for routing around faults on multibutterflies. In *Proc. 30th Annual IEEE Symposium on Foundations of Computer Science*, pp. 384–389.

Loomis, L. H. & Whitney, H. (1949). An inequality related to the isoperimetric inequality. *Bulletin of the American Mathematical Society*, **55** pp. 961–962.

Malewicz, G., Austern, M., Bik, A., Dehnert, J., Horn, I., Leiser, N. & Czajkowski, G. (2010). Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pp. 135–146.

McColl, W. F. (1995). Scalable computing. In *Computer Science Today: Recent Trends and Developments*, J. van Leeuwen, editor. LNCS Volume 1000. Springer-Verlag pp. 46–61.

McColl, W. F. (2018). A bridging model for high performance cloud computing. In *Proc. 18th SIAM Conference on Parallel Processing for Scientific Computing*.

McColl, W. F. & Tiskin, A. (1999). Memory-efficient matrix multiplication in the BSP model. *Algorithmica*, **24**(3) pp. 287–297.

Shazeer, N. et al. (2018). Mesh-TensorFlow: Deep learning for supercomputers. In *NIPS 2018 – Proc. 32nd Conference on Neural Information Processing Systems*.

Skillicorn, D. B., Hill, J. M. D. & McColl, W. F. (1997). Questions and answers about BSP. *Scientific Programming*, **6**(3) pp. 249–274.

Turing, A. M. (1937). On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society. Series 2*, **42** pp. 230–265. Corrections, ibid., **43** pp. 544–546.

Upfal, E. (1992). An $O(\log N)$ deterministic packet-routing scheme. *Journal of the ACM*, **39**(1) pp. 55–70.

Valiant, L. G. (1990*a*). A bridging model for parallel computation. *Communications of the ACM*, **33**(8) pp. 103–111.

Valiant, L. G. (1990*b*). General purpose parallel architectures. In *Handbook of Theoretical Computer Science, volume A, Algorithms and Complexity*, J. van Leeuwen, editor. Elsevier, pp. 943–971.

von Neumann, J. (1945). First draft of a report on the edvac. Moore School of Electrical Engineering, University of Pennsylvania. Contract No. W-670-ORD-4926 between the United States Army Ordnance Department and the University of Pennsylvania.

Yu, Q., Li, S., Raviv, N., Kalan, S. M. M., Soltanolkotabi, M. & Avestimehr, A. S. (2019). Lagrange coded computing – optimal design for resilience, security and privacy. In *Proceedings of the 22nd International Conference on Artificial Intelligence and Statistics (AISTATS), Naha, Okinawa, Japan*. PMLR: Volume 89.

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S. & Stoica, I. (2010). Spark: Cluster computing with working sets. In *HotCloud'10 – Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*.