CS 554 / CSE 512: Parallel Numerical Algorithms
Lecture Notes
Chapter 1: Parallel Computing

Michael T. Heath and Edgar Solomonik
Department of Computer Science
University of Illinois at Urbana-Champaign

September 4, 2019

1 Motivation

Computational science has driven demands for large-scale machine resources since the early days of computing. **Supercomputers**—collections of machines connected by a network—are used to solve the most computationally expensive problems and drive the frontier capabilities of scientific computing. Leveraging such large machines effectively is a challenge, however, for both algorithms and software systems. **Numerical algorithms**—computational methods for finding approximate solutions to mathematical problems—serve as the basis for computational science. Parallelizing these algorithms is a necessary step for deploying them on larger problems and accelerating the time to solution. Consequently, the interdependence of parallel and numerical computing has shaped the theory and practice of both fields. To study parallel numerical algorithms, we will first aim to establish a basic understanding of parallel computers and formulate a theoretical model for the scalability of parallel algorithms.

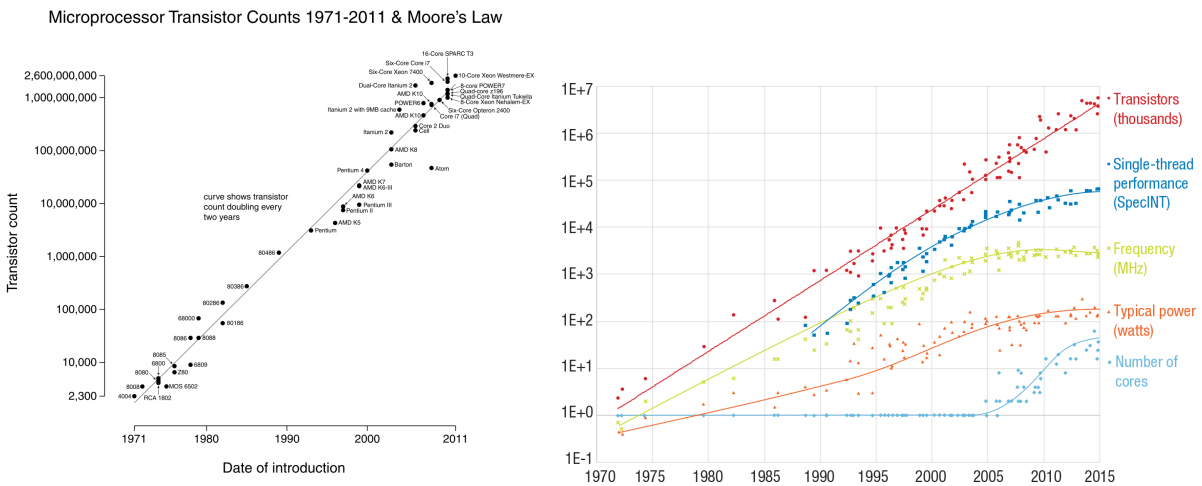
1.1 Need for Parallelism

Connecting the latest technology chips via networks so that they can work in concert to solve larger problems has long been the method of choice for solving problems beyond the capabilities of a single computer. Improvements in microprocessor technology, hardware logic, and algorithms have expanded the scope of problems that can be solved effectively on a sequential computer. These improvements face fundamental physical barriers, however, so that parallelism has prominently made its way on-chip. Increasing the number of processing cores on a chip is now the dominant source of hardware-level performance improvement.

The fundamental engineering challenge in increasing chip clock speed is to reduce both

- line delays – signal transmission time between gates, and
- gate delays – settling time before state can be reliably read.

Both of these can be improved by reducing the device size, but this is in turn ultimately limited by heat dissipation and thermal noise (degradation of signal-to-noise ratio). Moreover, transistors and gate logic cannot shrink indefinitely due to the finite granularity of atoms and quantum uncertainty in the state of physical microstructures.



(a) Moore's law [60]

(b) Architectural scaling trends [11]

Figure 1: Trends in change in numbers of transistors, sequential performance, frequency (clock rate), power consumption, and the number of cores on a chip, have varied over time. Note that single core ('sequential') execution still involves some parallelism, as a core generally has multiple FPUs, and the number of these has grown in recent years.

1.2 Moore's Law

The rapid spread and disruptive emergence of computers in the modern world is due to the exponential growth in their capabilities and corresponding reduction in relative cost per component. This phenomenon is a manifestation of a famous prediction made by Gordon Moore in 1965, known as **Moore's law** [27,39,51], that the density of transistors on a chip would double every two years[1]. For more than 40 years, transistor size of state-of-the-art chips has indeed shrunk by a factor of two every two years (see Figure 1(a)).

The commonplace interpretation of Moore's law has been that the clock speed of microprocessors would also double every two years, and the frequency of chips did in fact almost double every two years from 1975 to 2005. The connection between Moore's law and clock frequency can be interpreted via the notion of **Dennard scaling**. Based on the fundamental equations of electrodynamics, a 1974 paper by Robert Dennard and coauthors [10,18] postulated that as the density of transistors increases, current density (electric current for a given cross-section of the chip) can be kept constant, thereby increasing frequency without necessitating an increase in power consumption. In practice, however, increasing clock speed does require some increase in power due to current leakage and heat dissipation [36]. This growth in power consumption of chips is demonstrated historically in Figure 1(b). Consequently, Dennard scaling inevitably ceased around 2005, again due to limits imposed by physical laws. Cooling systems that can handle the heat dissipation of chips running at an operating frequency above 3 GHz are expensive to build and power, and at a high-enough frequency, they are impossible to engineer [45].

Fortunately, clock frequency is far from the only mechanism for achieving higher performance in hardware. More sophisticated hardware functionality, such as branch predictors and caching subsystems serve invaluable roles in accelerating execution. However, additional parallelism, either via an increase in cores per chip or floating-point processing units (FPUs) per core, provides the primary way to increase the maximum achievable performance of a single chip. The replacement of Dennard scaling (clock frequency growth) by increasing core counts since 2005 can also be observed in Figure 1(b).

[1] Actually, in 1965 Gordon Moore predicted that the density of transistors would double every year, but in 1975 revised this to two years.

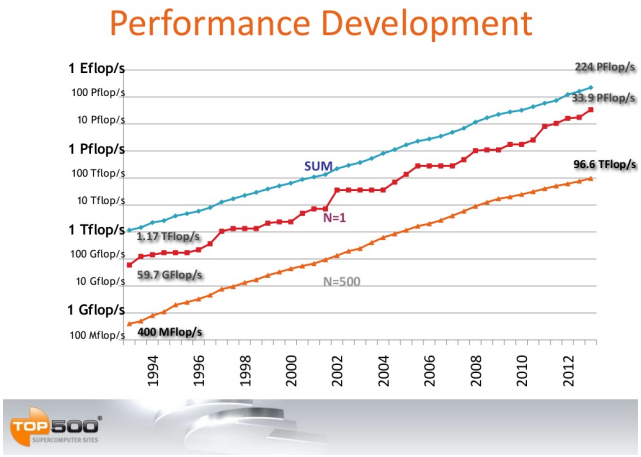


Figure 2: Peak performance rate achieved for LINPACK [19] by supercomputers participating in the Top 500 list [54]. $N = 1$ corresponds to the top computer on the list and $N = 500$ to the last, while 'SUM' is the aggregate performance of all 500 supercomputers on the list.

1.3 Supercomputing

The emergence of multicore chips is a relatively recent development in the greater picture of parallel computing. Supercomputers—high-end cluster of chips—have driven the development of parallel algorithms for computational science for many decades. The capabilities of supercomputing resources have reaped the benefits from improvements in on-chip performance driven by Moore's law. In fact, the peak performance of top supercomputers has outpaced Moore's law (see Figure 2), as the number of chips in the largest supercomputers has grown and accelerators such as GPUs have been deployed.

Thus, the performance and capabilities offered by supercomputers comes almost entirely from parallelism. Today's top machines employ up to 100,000 nodes, typically employing millions of hardware threads overall. Leveraging these machines effectively has become more and more challenging, especially as improvements in memory subsystem performance and network performance have been exponentially outpaced by the growth in aggregate floating-point performance [52]. This increase in parallelism and in the overhead of data movement now dictates the development methodology for computational science applications that strive to reach new frontiers in problem scale and fidelity.

These scientific applications almost universally depend on similar numerical kernels (Phillip Collela provides a succinct classification of these, which has become known as the "Seven Dwarfs" [6]). We will study algorithms for these numerical problems, using parallelism as an analytical lens. Before diving into the details of parallel computer architectures, we should ask whether radically different approaches to computer architecture design provide a way around the problems of parallelization. Specialized computers, such as the Anton molecular dynamics supercomputer [53] and the Tensor Processing Unit (TPU) [32], have proven to be effective for some computational science problems in recent years, but generally are massively parallel architectures themselves. Quantum computers [21, 42] have perhaps the most radical and theoretically significant potential of any computer architecture concept, and could in the future serve to supplant classical massively parallel architectures. Current quantum computing technology is far from becoming of practical interest to applications, however, and parallel quantum architectures have themselves become a topic of study [41].

2 Parallel Computer Architectures

We now provide an overview of the basic concepts of parallel computer architecture. The reader is expected to have some familiarity with computer architecture basics. As our focus is on the development of algorithms, we cover only essential aspects of parallel architectures.

A parallel computer consists of a collection of processors and memory banks together with an interconnection network to enable processors to work collectively in concert. Major architectural issues in the design of parallel computer systems include the following

- *processor coordination*: do processors need to synchronize (halt execution) to exchange data and status information?
- *memory organization*: is the memory distributed among processors or do all processors own a local chunk?
- *address space*: is the memory referenced globally or locally with respect to each processor?
- *memory access*: does accessing a given memory address take equal time for all processors?
- *granularity*: how much resources does the hardware provide to each execution stream (how fine or coarse of a task size does the hardware consider)?
- *scalability*: how does the amount of available resources scale with the size of a subdivision of a system (number of processors)?
- *interconnection network*: what connectivity pattern defines the network topology and how does the network route messages?

In the context of parallel algorithms, the organization of memory and the interconnection network are the most significant architectural features.

Our motivating discussion drew attention to the presence of parallelism within each core, across cores in a multicore chip, and on networked systems of chips. Often, several chips are combined in a single processing **node**, with all having direct access to a shared memory within that node. The following terminology is commonly used to refer to these sources of parallelism:

- **instruction-level parallelism (ILP)** refers to concurrent execution of multiple machine instructions within a single core,
- **shared-memory parallelism** refers to execution on a single-node multicore (and often multi-chip) system, where all threads/cores have access to a common memory,
- **distributed-memory parallelism** refers to execution on a cluster of nodes connected by a network, typically with a single processor associated with every node.

2.1 Parallel Taxonomy

Parallelism in computer systems can take various forms, which can be classified by **Flynn's taxonomy** in terms of the numbers of **instruction streams** and **data streams** [22], where "instructions" can refer to low-level machine instructions or statements in a high-level programming language, and "data" can correspondingly refer to individual operands or higher-level data structures such as arrays.

- **SISD** – single instruction stream, single data stream

  - used in traditional serial computers (Turing machines)

- **SIMD** – single global instruction stream, multiple data streams
  - critical in special purpose "data parallel" (vector) computers, present also in many modern computers

- **MISD** – multiple instruction streams, single data stream
  - not particularly useful, except if interpreted as a method of "pipelining"

- **MIMD** – multiple instruction streams, multiple data streams
  - corresponds to general purpose parallel (multicore or multinode) computers.

SIMD execution is the backbone of vector-based architectures [48], which operate directly on vectors of floating-point values rather than on individual values. While restrictive in that the same operation must be applied to each value or pairs of values of vectors, they can often be used effectively by data parallel applications. Dense matrix computations are one domain where this type of parallelism is particularly prevalent and easy to exploit, while algorithms that access data in irregular (unstructured) patterns oftentimes cannot.

Modern architectures employ SIMD vector instruction units, but also support multiple (vector) instructions streams (MIMD), especially in the presence of many cores. Typical numerical algorithms and applications leverage MIMD architectures not by explicitly coding different instruction streams, but by running the same program on different portions of data. The instruction stream created by programs is often data-dependent (e.g., contains branches based on state of data), resulting in MIMD execution. This single-program multiple-data (**SPMD**) programming model is widely prevalent and will be our principal mode for specifying parallel algorithms.

2.2 Memory Organization

In an SPMD style program, we must generally be able to move data so that different instances of the program can exchange information and coordinate. The semantics of data movement depend on the underlying memory organization of the parallel computer system. The most basic distinction is between shared-memory and distributed-memory organizations (see Figure 3), although modern clusters are almost always a mix of the two.

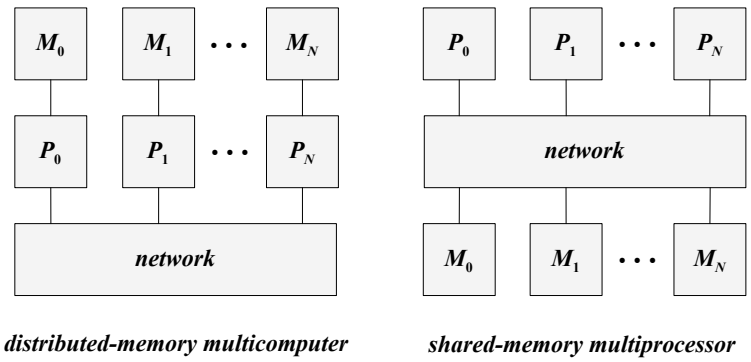


Figure 3: Shared-memory and distributed-memory systems differ with respect to how processors ($P_1, \ldots P_N$) are connected to memory (here partitioned in banks $M_1, \ldots, M_N$).

These two memory organizations are associated with different programming models. As different threads access different parts of a shared memory, the hardware caching protocols implicitly orchestrate data movement between local caches and main memory. Threads typically coordinate their accesses via

- **locks**, which restrict access to some data to a specific thread, and
- **atomic operations**, which allow threads to perform basic manipulations of data items without interference from other threads during that operation.

By contrast, on distributed-memory systems, data movement is typically done explicitly by passing messages between processors.

In terms of their corresponding programming models, shared and distributed memory organizations have advantages and disadvantages with respect to one another. For instance, incrementally parallelizing a sequential code is usually easiest with shared-memory systems, but achieving high performance using threads usually requires the data access pattern to be carefully optimized. Distributed-memory algorithms often require the bulk of the computation to be done in a data parallel manner, but they provide a clear performance profile, due to data movement being done explicitly. Table 1 provides a simplified overview of typical differences and similarities.

task	distributed memory	shared memory
scalability	easier	harder
data mapping	harder	easier
data integrity	easier	harder
performance optimization	easier	harder
incremental parallelization	harder	easier
automatic parallelization	harder	easier

Table 1: Typical programmability advantages and disadvantages of shared and distributed memory programming models and architectures.

An important commonality of the two programming models is that ultimately the performance of a program often depends primarily on load balance and data movement. Software systems have also been developed to permit the use of either programming model on either type of architecture. All widely used distributed-memory programming languages and libraries are capable of execution on shared-memory systems simply by partitioning the common memory. The other direction is addressed by partitioned global address space (**PGAS**) languages, which provide a shared memory abstraction for distributed memory systems by orchestrating data movement in software. Examples of PGAS languages include Global Arrays [43], UPC [12], and Co-array Fortran [44]. One of the most common distributed data structures used in these languages and in other libraries for parallel programming is the multidimensional array, which conveniently matches the vector and matrix abstractions prevalent in the mathematical specification of most numerical methods.

Our exposition of parallel numerical algorithms will be based on a distributed-memory execution model, which will allow us to reason quantitatively about data movement on supercomputing architectures. We note, however, that much of the literature in the theory of parallel algorithms is based on the parallel random access memory (PRAM) model, which is a shared-memory execution model. Data movement can be quantified in this setting by augmentations of the PRAM model, e.g., LPRAM [3] and BSPRAM [56]. Before specifying our formal model for parallel execution and data movement, we consider in more detail how data movement is supported in distributed-memory hardware systems.

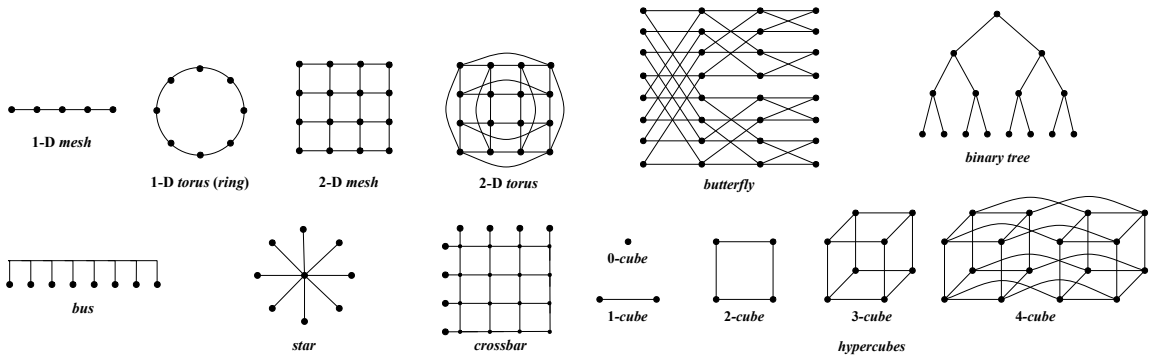


Figure 4: Depiction of several types of classical network topologies.

3 Interconnection Networks

Connecting each pair of nodes directly via network wires would require $O(p^2)$ wires and communication ports on a system with p processors, which is not scalable. Practical networks are much more sparsely connected, which implies that messages between processors must somehow be routed through the network. And yet some modern network designs are so effective that a **fully-connected** theoretical network model often provides reasonable predictions about algorithmic performance in relative terms. In this section we review the fundamentals of network topologies and routing, as well as discussing how parallel algorithms should view the network. In subsequent algorithmic analysis sections we will consider various network topologies, mostly as virtual abstractions for the execution of communication protocols for a given decomposition of data.

3.1 Network Topologies

Perhaps the most fundamental issue in the design of networks for distributed-memory computers is the **network topology** – how nodes in a network are connected. The two basic types of network topologies are:

- **direct** – each network link connects some pair of nodes,
- **indirect (switched)** – each network link connects a node to a network switch or connects two switches.

Figure 4 depicts some classical direct and indirect network topologies. The crossbar topology is an example of an indirect network containing $p^2/4$ switches (smaller dots in Figure 4). The most important of these classical topologies are **tori** and **hypercubes**. Torus networks are still used in modern supercomputers and are often convenient as model virtual topologies. Hypercubes are generally no longer built, but remain of significant theoretical importance. To discuss these and other networks more formally, we study their properties as graphs.

3.1.1 Graph Representation

We first review some basic graph terminology, which will form a basis for our representation and analysis of network topologies. The general definition of undirected graphs (sufficient for our purposes) and some important special cases thereof are

- **Graph** – pair (V, E), where V is a set of vertices or nodes connected by a set E of edges,
- **Complete graph** – (fully-connected) graph in which any two nodes are connected by an edge,

Network	Nodes	Degree	Diameter	Bisection Bandwidth	Edge Length
bus/star	$k+1$	k	2	1	var
crossbar	k^2+2k	4	$2(k+1)$	k	var
1-D mesh	k	2	$k-1$	1	const
2-D mesh	k^2	4	$2(k-1)$	k	const
3-D mesh	k^3	6	$3(k-1)$	k^2	const
n-D mesh	k^n	$2n$	$n(k-1)$	k^{n-1}	var
1-D torus	k	2	$k/2$	2	const
2-D torus	k^2	4	k	$2k$	const
3-D torus	k^3	6	$3k/2$	$2k^2$	const
n-D torus	k^n	$2n$	$nk/2$	$2k^{n-1}$	var
binary tree	2^k-1	3	$2(k-1)$	1	var
hypercube	2^k	k	k	2^{k-1}	var
butterfly	$(k+1)2^k$	4	$2k$	2^k	var

Table 2: Basic properties of classical network topologies.

- **Path** – sequence of contiguous edges in graph,
- **Connected graph** – graph in which any two nodes are connected by a path,
- **Cycle** – path of length greater than one that connects a node to itself,
- **Tree** – connected graph containing no cycles,
- **Spanning tree** – subgraph that includes all nodes of a given graph and is also a tree.

Critical to understanding how nodes can communicate with each other will be the notion of the **distance** between a pair of nodes – the number of edges (**hops**) in **shortest** path between them.

The following important properties of network topologies can also be modelled conveniently using graphs:

- **degree** – maximum number of edges incident on any node (determines number of communication ports per processor),
- **diameter** – maximum distance between any pair of nodes (determines maximum communication delay between processors),
- **bisection bandwidth** – smallest number of edges whose removal splits the graph into two subgraphs of approximately equal size (determines ability to support simultaneous global communication),
- **edge length** – maximum physical length of any wire (may be constant or variable as the number of processors varies).

The (generalized versions of the) topologies in Figure 4 are characterized in terms of these properties in Table 2. Note the tradeoffs between properties associated with construction cost (degree and edge length) and properties associated with the performance of the network (diameter and bisection bandwidth).

3.1.2 Graph Embedding and Emulation

Given model graphs for network topologies, two important questions are

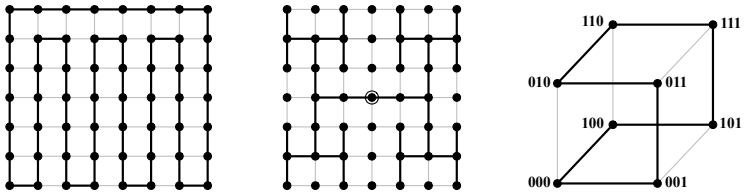


Figure 5: Ring embedded in 2-D mesh with dilation 1 (left), binary tree embedded in 2-D mesh with dilation $\lceil (k-1)/2 \rceil$ (middle), and ring embedded in hypercube with dilation 1 (right).

- whether a given network can **emulate** others effectively, and
- what types of communication patterns can be routed efficiently.

A useful tool for both of these analyses is **graph embedding**, defined formally via a map $\phi\colon V_s \to V_t$ from nodes in the source graph $G_s = (V_s, E_s)$ to nodes in target graph $G_t = (V_t, E_t)$, as well as a map of the edges in G_s to paths in G_t. The quality of a graph embedding can be measured in terms of

- **load** – maximum number of nodes in V_s mapped to the same node in V_t,
- **congestion** – maximum number of edges in E_s mapped to paths containing the same edge in E_t,
- **dilation** – maximum distance between any two nodes $\phi(u), \phi(v) \in V_t$ such that $(u, v) \in E_s$.

When considering the ability of networks to emulate one another, it makes sense to constrain the load and congestion to be no greater than one. Dilation and the size of V_t with respect to V_s characterize the overhead and slowdown associated with the emulation.

When considering the mapping of an algorithm onto a network, we can model the algorithm as a series of computations (nodes V_s) and messages between them (edges E_s). As we will see, the load of the graph embedding (with the network described by V_t) will be correlated with the load balance of the algorithm, i.e.,the amount of work performed by each processor. Congestion and dilation will describe the effectiveness of the network in executing the algorithm. For some algorithms, however, this graph embedding model does not account for concurrency, so the load and dilation measures alone are not sufficient to characterize the execution time of the algorithm. Graph embeddings will play multiple roles in how we think about parallel algorithms, their decomposition, and their mapping to network hardware.

For some important graphs, good or even optimal embeddings are known. Figure 5 demonstrates a few of these with examples of both unit and non-unit dilation. Network topologies that can emulate other networks are of both theoretical and practical interest. The network topologies we consider in the subsequent two sections (hypercubes and fat trees) came into prominence for precisely that reason.

3.1.3 Hypercubes

A hypercube of dimension k, or k-cube, is a graph with 2^k nodes numbered $0, \ldots, 2^k - 1$, with edges between all pairs of nodes whose binary numbers differ in exactly one bit position. A hypercube of dimension k can be created by recursively replicating a hypercube of dimension $k-1$ and connecting corresponding nodes between the two copies (see Figure 4, lower right). **Gray codes** – orderings of integers 0 to $2^k - 1$ such that consecutive members differ in exactly one bit position are convenient for thinking about routes in hypercubes. The following C-language function generates a Gray code, where the Gray code `gray(i)` for node `i` will be exactly one bit flip away from the Gray code `gray(i+1)` of node `i+1`.

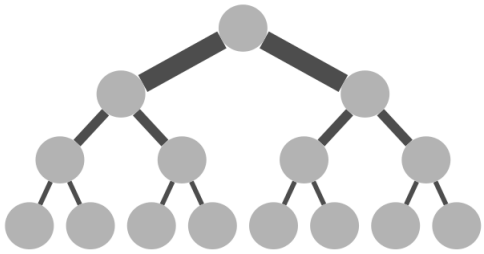


Figure 6: Depiction of fat-tree network [59].

```
/* Gray code */
int gray(int i){
  return((i>>1)^i); //shift i down by one bit and take XOR of result with i
}
```

Visiting the nodes of a hypercube in a Gray code order gives a **Hamiltonian cycle**, embedding a ring in the hypercube as in Figure 5. For a mesh or torus of higher dimension, concatenating Gray codes for each dimension gives an embedding in a hypercube of appropriate dimension.

An important property of hypercubes is their ability to perform collective communications effectively [49]. In Section 4, we will present near-optimal butterfly communication protocols, all of which can be executed effectively on hypercube networks. More generally, with the use of randomized mappings hypercube networks can route *any* parallel algorithm with logarithmic slow-down [58]. At a high level, this means that hypercubes (and many other sparsely connected networks) are no more than logarithmically worse in performance than a fully-connected network, yet are much more scalable in terms of the number of wires required.

3.1.4 Fat Trees

While a hypercube is sparsely connected, it is not necessarily cheap to build. In Table 2 this difficulty is reflected by the fact that the edge length of links in the hypercube network is not constant with the number of processors. The number of links also grows logarithmically with system size.

Ultimately, any physical network must be embedded in physical space, which is 3-dimensional. Quantitatively, we observe that the edge length of hypercubes necessarily grows as we construct larger networks, and the number of communication ports required per node also grows as well. Consequently, hypercubes are not used in modern systems with very large numbers of nodes.

The target question of network design in this view becomes [16, 37]

"*How good a network can be built within a given physical volume?*"

Fat trees provide a universal and near-optimal solution to this question [37]. Precisely stated, no network can be constructed with the same number of components that is faster by more than a polylogarithmic factor. Figure 6 displays a fat-tree layout. The network is an indirect binary tree, with each node being a leaf. In order to maintain good bisection bandwidth, fat trees have varying link bandwidth (bandwidth available to a given edge in the network), with links near the root of the tree being 'fatter'. For example, the fat tree will have bisection bandwidth proportional to p if the link bandwidth increases by a factor of 2 at each higher level in the binary tree.

By considering any network in physical space and embedding the space in a fat tree whose bisection bandwidth scales with $p^{2/3}$, it can be shown that the fat tree can emulate any exchange of messages on that

network with at most a factor of $O(\log(p)^3)$ more time. The generality of this result is surprising, as the only assumption necessary is that the emulated network can pass $O(a)$ data through an area of size a.

In practice, fat tree networks have been built with varying amounts of bisection bandwidth by choosing different factors of exponential growth through the levels of the tree. This makes sense, especially if many applications run only on a subset of nodes, corresponding to a subsection of the tree.

3.1.5 State of the Art: Constant-Diameter Networks

While low-diameter hypercube architectures have seen success historically [28], recent supercomputers have leveraged topologies such as low-dimensional tori [1, 34], with a larger diameter but constant degree. More recently, the trend is network topologies is toward lower diameter, which is achieved by higher-dimensional torus networks [4, 14] or alternative network topology constructions. **Dragonfly** networks [35] achieve a diameter of 3 topology that is featured in some current-generation Cray architectures. They are structured as follows

- define densely connected groups (cliques) of nodes,
- connect a single pair of nodes between each pair of groups.

The topology is diameter 3, since a route from a node in group A to a node in group B can be found by

1. routing to the node in group A that is connected to a node in group B,
2. routing through the link to the node in group B,
3. routing to the appropriate node in group B.

A major advantage of this type of low-diameter topology is low network congestion (short routes are less likely to all go through the same links). However, the network requires a degree of $\Omega(p^{1/3})$.

The success of the Dragonfly network topology has served to motivate other low-diameter networks, including the a peculiar diameter-2 network topology called **Slim Fly** [7]. The network uses a construction from graph theory that aims to minimize the degree-to-diameter ratio. In particular, the Moore bound (developed by Hoffman and Singleton) [30] states that any regular (uniform-degree) graph with degree d and diameter r, must satisfy

$$p \leq 1 + d\sum_{i=1}^{r-1}(d-1)^i \qquad \left(\text{asymptotically}, p = O(d^r)\right),$$

a bound attained by Moore graphs. However, these are not known for general degree and diameter. So the Slim Fly construction uses a class of graphs that obtains a very close approximation to the bound [25, 40] for diameter-2 graphs.

The Slim Fly is constructed using a Galois field, in particular modular arithmetic over a prime q. We use the notation $a \equiv_q b$ to mean $a \equiv b \mod q$. The Slim Fly is designed as follows.

- a network of size $2q^2$ is constructed where q is (almost any) prime,
- there are two $q \times q$ grids A, B, each node is connected to some nodes in its column and some nodes in the other grid,
- given a node $(x, y) \in A$ in the first grid and $(m, c) \in B$ in the second grid, they are connected if

$$mx + c \equiv_q y,$$

- the nodes $(x, y_1), (x, y_2) \in A$ are also connected if $y_1 \equiv_q y_2 + k$ where $k \in X$ (and similar for B) where X contains roughly $q/2$ primitive elements of the Galois field modulus q [7].

By leveraging the existence of the inverse of linear modular functions for a prime modulus, one can show that 2-hop routes exist between all pairs of nodes. For instance, to get from a node (x, y) in one column in A to any node (x', y') in a different column in A ($y \neq y'$), we can step through a single intermediate node (m, c) in B. In particular, we need

$$mx + c - y \equiv_q mx' + c - y' \equiv_q 0,$$

and can determine (m, c) directly provided the existence of the appropriate modular inverse (which slightly restricts possible choices of q),

$$\begin{aligned} mx + c - y &\equiv_q mx' + c - y' \\ m(x - x') &\equiv_q y - y' \\ m &\equiv_q (x - x')^{-1}(y - y') \quad \text{and} \quad c \equiv_q y - mx. \end{aligned}$$

Given p processors, the degree is then about $q = \sqrt{p}$, which comes to within a small factor of the Moore bound. One can compare the Slim Fly to a 2D grid network, where every node is connected to all nodes in their row and column (yielding a diameter-2 network), which requires $2(\sqrt{p} - 1)$ edges (worse by about a factor of 2). Generalizing the Slim Fly construction to higher diameter and more generally finding families of graphs that minimize the Moore bound remain open problems.

3.2 Topology Awareness of Algorithms

The contrast in the structural complexity between classical and modern networks must be recognized in the models for designing parallel algorithms. Tori and hypercubes provide multidimensional and bitwise representations of data to which structured algorithms are often easy to map. In contrast, the abstract structure of the Slim Fly network is significantly different from the structure provided by typical parallel decompositions of algorithms. But the advantage of modern low diameter networks is that the congestion should be low even if the communication pattern does not specifically exploit the structure of the topology.

Topology-aware algorithms aim to execute effectively on specific network topologies. If mapped ideally to a network topology, applications and algorithms often see significant performance gains, as network congestion can be controlled and often eliminated [2, 29]. In practice, unfortunately, applications are often executed on a subset of nodes of a distributed machine, which may not have the same connectivity structure as the overall machine. In an unstructured context, network congestion is inevitable, but could be alleviated with general measures.

While network-topology-specific optimizations are typically not performance portable, topologies provide a convenient visual model for designing parallel algorithms. We have seen that good network topologies have the capability to emulate one another for different communication patterns. This understanding suggests that if we design algorithms that avoid congestion on a minimal sparsely-connected network topology, then any topology that is more powerful would be able to execute the algorithm without congestion also. Provided an efficient mapping between network topologies, we can then determine how well the algorithm performs on a more powerful network topology. An even better algorithm might be possible on the better-connected network, however.

On the other hand, an algorithm designed for a more densely-connected network typically incurs a bounded amount of overhead on a more sparsely connected one, if it can be shown to emulate the protocols performed on a more powerful network. While topology awareness is an important area of study, ideally

we would like parallel algorithms to be **topology oblivious**, i.e., perform well on any reasonable network topology. Of course, one cannot reason about arbitrary topologies, but some abstractions (like collective communication) are sufficiently convenient to enable us to reason simultaneously about many.

While the best approach differs depending on context, our analytical methodology will be to

1. try to obtain cost optimality for a fully connected network, and then
2. organize the algorithm so that it achieves the same cost on some network topology that is as sparsely-connected as possible.

This methodology aligns with the idea that platform-specific optimizations should be of second-order importance, while maintaining awareness of how challenging the desired communication pattern is to map to a realistic network.

4 Communication

To understand and compare routing protocols for networks, we must begin to think about the **communication cost** (**communication complexity**) associated with movement of data messages. These measures will be defined to best correlation with the execution time. Specifically, we will work with **communication time** – the execution time of the communication protocol executed by algorithm. In general, analyzing communication complexity is intertwined with analyzing the complexity analysis of parallel algorithms. We introduce these formal concepts and first apply them to understand basic communication protocols in networks.

4.1 Message Passing

We use a simple standard model for the time taken to communicate a message of s words across a single link of a network,

$$T^{\text{msg}}(s) = \alpha + \beta\, s.$$

This model includes the following cost parameters:

- α = **startup time** = **latency** (i.e., time to send a message of zero length),
- β = incremental **transfer time** per word ($1/\beta$ = **bandwidth** in words per unit time).

For real parallel systems $\alpha \gg \beta$, so we often simplify $\alpha + \beta \approx \alpha$, meaning that we approximate the cost of sending a 1-byte (constant-sized) message by the latency cost. We will generally assume a processor can send or receive only one message at a time (but can send one and receive another simultaneously). On some systems it is possible to send many messages concurrently, the main constraint being how many network links can be saturated in bandwidth, which is given by the ratio of the **injection bandwidth** of the node to the **link bandwidth** of every processor within the node. Here and later in the analysis of communication in algorithms, we assume there is a single processor per node and the link bandwidth is the same as the injection bandwidth.

For low-level modelling of communication performance, this simple model is not always sufficient. The LogP [15] and LogGP [5] models provide a more refined parameterization of the latency overhead in messaging and the extent to which messages can overlap. However, it is difficult to work with a theory of algorithms that minimizes such small parameters (some known optimal algorithms in the LogP model are not described in closed form [33]).

In modern network hardware, three types of messaging protocols are typically implemented, which differ in the way they manage buffering,

- **eager** – sends a small message, writing to a dynamically allocated buffer then copying to destination buffer,
  - effective for small messages, where copy overhead is not an issue,
- **rendezvous** – first establishes a handshake via an additional small message, retrieves destination buffer information, and writes data directly to it,
  - incurs higher network latency but avoids copy overhead for large messages,
- **one-sided** – sends messages directly to destination buffer without synchronizing (performing a handshake) with the target processor, having pre-established buffer information,
  - requires special a priori buffer registration and remote destination memory access (RDMA) technology [38] but improves asynchrony and performance.

Our simple messaging model provides a reasonable cost representation for the first two types of messaging protocols. However, analyzing algorithms in terms of the one-sided communication model would suggest decoupling of synchronization and communication. The Bulk Synchronous Parallel (BSP) model [58] reflects this to an extent while maintaining a simple superstep-based parallel execution model. However, the complexity of algorithms under synchronization-based and messaging-based models are closely related [8, 47].

4.2 Algorithmic Communication Cost

To measure the execution time of a parallel program, neither local nor cumulative measures are sufficient. In general, we would like to understand the costs (of messaging and of local computation) incurred along the **critical path** of the execution of a parallel program. This measure is given by considering all dependent executions in an algorithm. Generally these are data-driven, so a precedence ordering between communication and computation tasks, as well as the order in which each processor executes them, provides a partial ordering ($\prec$) (see also Figure 7):

$$t_1 \preceq t_2 \preceq \cdots \preceq t_N.$$

If $T(t_i)$ is the time taken to execute task t_i, then the execution time is given by maximizing over all totally ordered subchains in the partial ordering,

$$T(t_1, \ldots t_N) = \max_{\{f_1, \ldots f_k\} \subseteq \{t_1, \ldots t_N\}, f_1 \prec f_2 \prec \cdots \prec f_k} \sum_{i=1}^{k} T(f_i).$$

We define the **communication cost** (**communication complexity**) in terms of S (the number of messages) and W (the number of words) incurred along the respective most expensive execution paths (totally ordered sequences in the partial ordering of tasks). Our communication cost/complexity bounds the **communication time**,

$$T^{\text{comm}}(c_1, \ldots c_M) = \max_{\{f_1, \ldots f_k\} \subseteq \{c_1, \ldots c_M\}, f_1 \prec f_2 \prec \cdots \prec f_k} \sum_{i=1}^{k} T(f_i) \leq \alpha S + \beta W,$$

where $c_1, \ldots c_M$ are all communication tasks. Communication complexity can be contrasted with **communication volume**, which is just the sum of the costs of all communication tasks.

We consider an example to illustrate the importance of this distinction. We study the time taken for p processors to pass a message of size s around a ring versus the time taken for each processor to pass a different message to the next processor in the ring. In both cases, the communication volume is the same

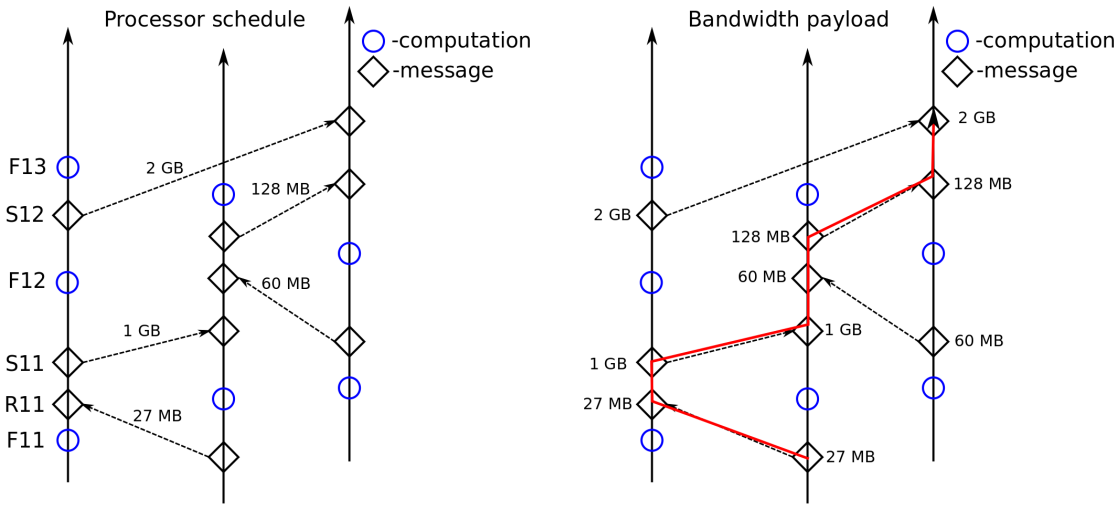


Figure 7: The critical path is given by the most expensive path in the depicted 3-processor execution. On the right, we highlight the path corresponding to the bandwidth cost component of the schedule.

(p, sp) if expressed as a pair (messages, words), while the communication complexity is either (p, sp) or $(1, s)$ in the same units, depending on whether a message is being passed around (communication tasks are dependent and so totally ordered) or all messages are independent (and can be sent concurrently). Our measure of communication time for the two cases is

- if the messages are sent *simultaneously*,

$$T_p^{\text{sim-ring}}(s) = T^{\text{msg}}(s) = \alpha + s \cdot \beta,$$

- if the messages are sent *in sequence*,

$$T_p^{\text{seq-ring}}(s) = p \cdot T^{\text{msg}}(s) = p \cdot (\alpha + s \cdot \beta).$$

Communication complexity is our measure of choice because it best reflects execution time, but communication volume may also be important for the energy consumption of an algorithm, as it reflects the total amount of network resources utilized over the entire execution of the algorithm.

4.2.1 Message Routing

Thus far our notion of communication complexity has been oblivious to network topology. To understand the cost and execution time of messaging on specific network topologies, we need to understand how small and large messages are **routed** in a given network. Message routing algorithms can be

- **minimal** or **nonminimal**, depending on whether the shortest path is always taken,
- **static** or **dynamic**, depending on whether the same path is always taken,
- **deterministic** or **randomized**, depending on whether the path is chosen systematically or randomly,
- **circuit switched** or **packet switched**, depending on whether entire the message goes along a reserved path or is transferred in segments that may not all take same path.

Most regular network topologies admit simple routing schemes that are static, deterministic, and minimal. For simple classical topologies such as meshes, tori, and hypercubes, such protocols are simple to derive. Figure 8 depicts some example routes.

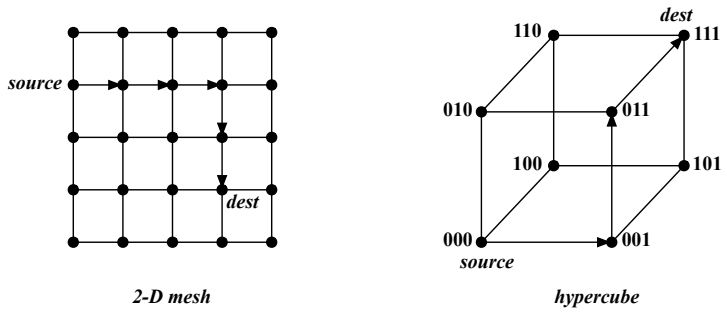


Figure 8: Minimal routes on a mesh and on a hypercube.

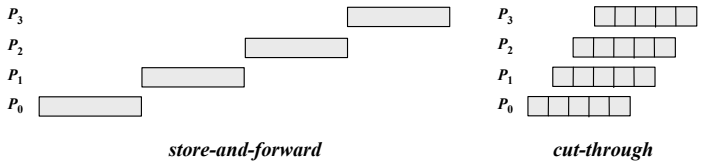


Figure 9: Comparison of store-and-forward and wormhole routing.

4.2.2 Wormhole Routing and Virtual Channels

When a long message needs to be routed along a network path, the use of pipelining can improve performance by a factor proportional to the length of the path. **Store-and-forward** routing handles the entire message atomically (without breaking it up). This means that the entire message traverses its route one network link at a time, so

$$T^{\text{route}}(s, D) = (\alpha + \beta s)D, \text{ where } D = \text{distance in hops.}$$

Store-and-forward routing corresponds to eager messaging protocols.

Cut-through or **wormhole** routing breaks the message into segments that are pipelined individually through the network, with each segment forwarded as soon as it is received, so

$$T^{\text{route}}(s, D) = \alpha + \beta s + t_h D, \text{ where } t_h = \text{incremental time per hop.}$$

Generally $t_h \leq \alpha$, so we can treat both as network latency,

$$T^{\text{route}}(s, D) = \alpha D + \beta s.$$

Figure 9 schematically depicts the two message routing techniques.

4.2.3 Network Congestion and Virtual Channels

Network congestion (**network contention**) is the competition for network resources due to multiple intersecting message routes desiring to use a given link at the same time. Especially in the presence of wormhole routing, any sparse interconnection network must be able to avoid deadlock and resolve network congestion for any possible routing of message interchanges. Most often, network congestion is resolved via the use of **virtual channels**, which are labels placed on message routes [17]. Each switch alternates servicing among the virtual channels. This means that if two long-message routes pass through the same network link concurrently, each will typically take twice as long, experiencing half of the nominal link bandwidth.

5 Collective Communication

Collective communication is a concurrent interchange of a set of messages among any set of pairs of processors. Basic structured collective communication operations deserve attention due to their prevalence in algorithms and applications. They serve as effective building blocks for both algorithms and software. We can classify some of the most common collective communication operations as follows,

- **One-to-All**: Broadcast, Scatter
- **All-to-One**: Reduce, Gather
- **All-to-One + One-to-All**: All-reduce (Reduce+Broadcast), All-gather (Gather+Broadcast), Reduce-Scatter (Reduce+Scatter), Scan
- **All-to-All**: All-to-all (also known as transpose and complete exchange)

The distinction between the last two types is made due to their different cost characteristics. The Message-Passing Interface (MPI) standard [24] provides all of these as well as more general versions that support variable message sizes.

We analyze the communication complexity of all of these algorithms, which are closely related to each other. Figure 10 provides a convenient visual matrix representation of some important collectives. The most common collective operation is broadcast, but when studying wormhole-routed collective protocols, we will see that scatter and gather are more primitive algorithmically.

5.1 Single-Item (Store-and-Forward) Collectives

We first consider collective communication protocols that function like store-and-forward routing, moving the whole message from processor to processor. In modern algorithm design, these are often used to generate small amoungs of global information, e.g.,computation of a vector norm. We subsequently study many-item (wormhole-routed) collectives, which communicate larger messages, exploiting additional parallelism to improve efficiency. The focus of this section, **single-item collectives** are protocols that communicate the message atomically (without subdividing it into chunks). In pratice, such protocols are also used when the messages can be subdivided but are too small for this to be beneficial.

Single-Item Broadcast – a root processor sends an indivisible message of $p-1$ other processors.

The most commonly used collective communication operation is *broadcast*. When executed on a subset of the processors of a system, a broadcast is also sometimes called a **multicast**. Every single-item broadcast protocol must propagate the message along a spanning tree of p processors. Here the specifics of our communication cost model come into play. If a processor can send and receive only one message concurrently, then a parent must choose an ordering of children to which they pass the message. This means that the first leaf of the tree will receive the message $\log_2(p)-1$ message passes before the last leaf does. Moreover, we can observe that after forwarding the message to both children, many parents in the tree could further assist algorithmic progress by passing the message to yet more processors. This intuition motivates the **binomial tree** construction. A binomial tree of height h is constructed by taking a binomial tree of height $h-1$ and appending a leaf to each of its processors. The broadcast protocol is similar, after h message-sends, it propagates the message from all processors in a binomial tree of size h to their respective neighbors in a larger binomial tree of size $h+1$, until reaching all processors. With p processors, the height of the binomial tree is as low as $\log_2(p+1)$, meaning that the communication time of a binomial tree broadcast is

$$T_p^{\text{bcast-binomial}}(s) = (\log_2(p+1)-1)T^{\text{msg}}(s) = (\log_2(p+1)-1)(\alpha+\beta\, s).$$

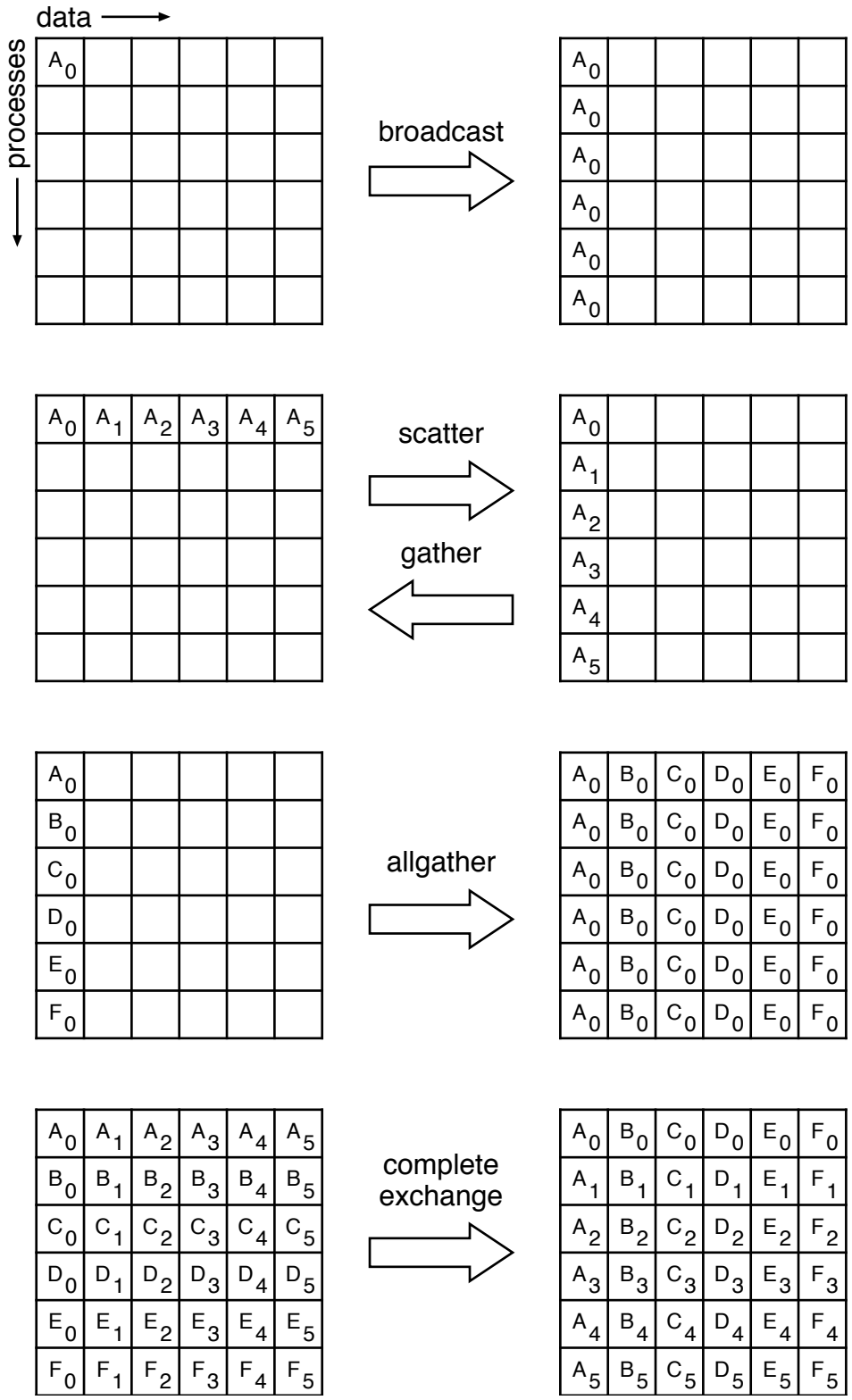


Figure 10: Depiction of collective communications using a messages-by-processors matrix [23].

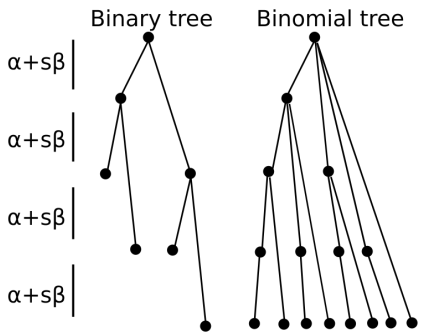


Figure 11: Comparison of binary and binomial trees for single-item broadcast. The binomial tree can be used to perform a single-item broadcast over 15 processors in the same communication time as a binary tree takes to do 7.

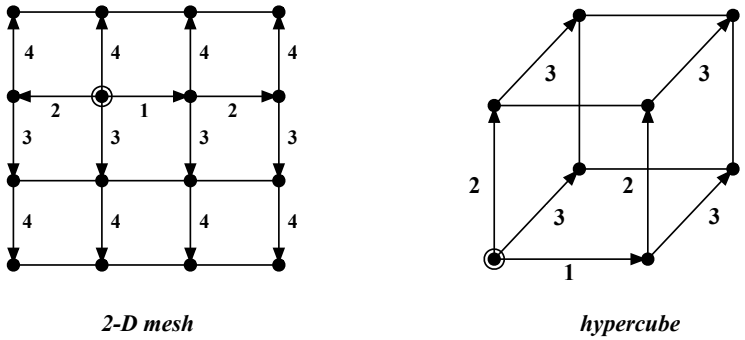


Figure 12: Example optimal broadcast trees on mesh and hypercube networks.

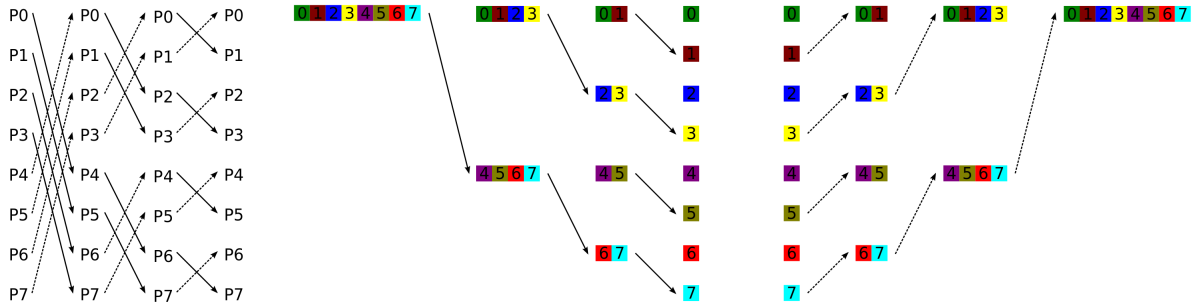
A binomial tree is optimal for broadcasting in synchronization cost (α prefactor in communication time) in our messaging model [20]. Figure 11 displays its advantage over a binary tree by dilating each edge in time. In the LogGP model, the best broadcast spanning tree depends nontrivially on the model parameters [33].

On a hypercube network topology, we can embed the binomial tree broadcast protocol and execute a multi-item broadcast optimally. On a mesh or torus topology of dimensions d, we can generally build a spanning tree of height proportional to the diameter $O(p^{1/d})$. Figure 12 demonstrates examples of how these broadcast protocols are realized on these two types of networks.

Single-Item Reduction – indivisble values from each of p processors are combined by applying a specified associative operation $\oplus$ (e.g.,sum, product, max, min, logical OR, logical AND) to produce a single value stored on the root processor.

The dual operation to broadcast is *reduction*, which combines values from all processors into a single value stored on one processor. Any broadcast protocol can be reversed to construct a reduction protocol, and vice versa. The cost of the computation in reductions is often disregarded, but presents technical challenges when implementing reductions directly into network hardware, causing broadcasts to be faster than reductions on typical systems.

Collectives such as scatter and (all)gather, as well as reduce-scatter and all-reduce, can be done in similar ways to broadcasts and reductions in the single-item case. The hierarchy among these becomes clearer if we consider them more generally in the multi-item case.



(a) Butterfly Schematic (b) Butterfly Scatter (Recursive Halving) (c) Butterfly Gather (Recursive Doubling)

Figure 13: 8-processor butterfly collective protocol schematics for recursive-halving to scatter messages and recursive-doubling to gather messages.

5.2 Multi-Item Collectives

Multi-item collectives can be thought of as concurrent executions of many independent single-item collectives. They are of independent interest, and are usually viewed as the basic general definition of collectives because they can be performed with more sophisticated and efficient protocols than simply a sequence of single-item collectives. In practice, they are also typically of greatest importance, as they can be used to achieve bulk movement of data.

There are numerous protocols to consider for each collective communication operation. We choose to focus on **butterfly collective communication protocols**, due to their generality and near-optimality. We refer the reader to the following alternate surveys of communication collectives [13, 31, 46, 55].

5.2.1 Gather and Scatter

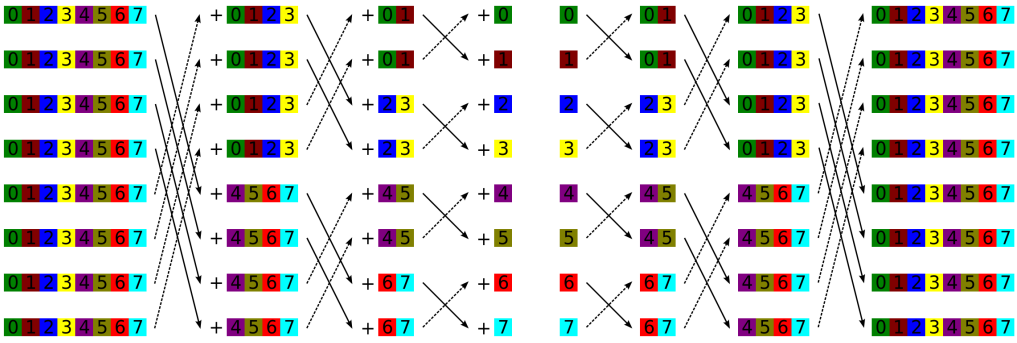
Scatter – a root processor sends a distinct message of size s/p to each of $p-1$ other processors.

One of the simplest multi-item collectives is the *scatter* operation. Efficient algorithms for scatter are achieved by disseminating messages down a spanning tree, but unlike broadcast, sending different data to each child in the tree. Simply pipelining the messages down any spanning tree is one way to achieve good efficiency, and is a generic strategy we can employ on many networks. But we can also achieve near-optimality via **recursive-halving**, which uses a binomial spanning tree, with each processor sending half as much data at each successive algorithmic step. This strategy is demonstrated in Figure 13(b), where the binomial tree is embedded in a butterfly. The butterfly depicts all processors at each level (column) as displayed in Figure 13(a), so it does not correspond to the network topology.

Gather – each processor sends a message of size s/p to the root processor.

The dual operation to scatter is *gather*. Reduce-scatter and all-gather are more balanced but more expensive versions of scatter and gather. They can be done with a butterfly protocol, with full utilization as demonstrated in Figure 14(a) and Figure 14(b). Such a protocol could be executed without slowdown on a hypercube network topology.

All of the butterfly collective protocols we have seen thus far increase or reduce the message size sent at each level of the butterfly by a factor of two. This characteristic enables them to obtain a bandwidth cost that is linear in the overall message size s. To analyze the cost of butterfly protocols, it helps to note their



(a) Butterfly Reduce-Scatter (b) Butterfly All-Gather

Figure 14: 8-processor butterfly collective protocol schematics for reduce-scatter and all-gather.

recursive structure. For example, after the first exchange of messages in the scatter-reduce protocol, two independent smaller butterfly protocols are executed. On the other hand, in the all-gather protocol the last step combines results computed by two independent butterfly protocols.

The butterfly network protocol for all-gather simply executes two independent butterfly all-gather protocols on each half of the processors, then combines the result by a pairwise exchange of messages of size $s/2$, which has cost $T^{\text{msg}}(s/2)$. Therefore, we can express the communication time complexity of the butterfly protocol for all-gather as a recurrence,

$$\begin{aligned} T_p^{\text{all-gather}}(s) &= \begin{cases} 0 & : p = 1 \\ T_{p/2}^{\text{all-gather}}(s/2) + \alpha + \beta(s/2) & : p > 1 \end{cases} \\ &\approx \alpha \log_2(p) + \beta \sum_{i=1}^{\log_2 p} s/2^i \\ &\approx \alpha \log_2(p) + \beta s. \end{aligned}$$

The communication time of scatter, gather, and reduce-scatter butterfly protocols are the same as that of all-gather,

$$T_p^{\text{all-gather}}(s) = T_p^{\text{scatter}}(s) = T_p^{\text{gather}}(s) = T_p^{\text{reduce-scatter}}(s).$$

5.2.2 Broadcast and Reduction

Now we can use these four basic one-to-all and all-to-one butterfly collective protocols to execute more complex operations efficiently, in particular broadcast and reduction.

Broadcast – a root processor sends same message of size s to each of $p-1$ other processors.

Broadcast can be done by performing a scatter followed by an all-gather (see Figure 15) in time

$$T^{\text{bcast}} = T^{\text{scatter}} + T^{\text{all-gather}} = 2T^{\text{all-gather}}.$$

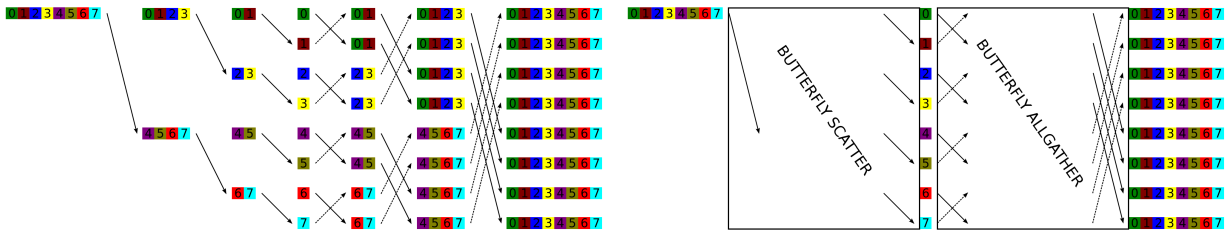


Figure 15: Multi-item broadcast done by butterfly scatter and all-gather protocols.

Reduction – s values for each of p processors are combined by applying a specified associative operation $\oplus$ (e.g.,sum, product, max, min, logical OR, logical AND) to produce a single value stored on the root processor.

A reduction can be done effectively by performing a reduce-scatter and a gather in time

$$T^{\text{reduce}} = T^{\text{reduce-scatter}} + T^{\text{gather}} = 2T^{\text{all-gather}}.$$

An all-reduction (where the result of the reduction is replicated on all processors) can be done effectively by performing a reduce-scatter followed by an all-gather (see Figure 16) in time

$$T^{\text{all-reduce}} = T^{\text{reduce-scatter}} + T^{\text{all-gather}} = 2T^{\text{all-gather}}.$$

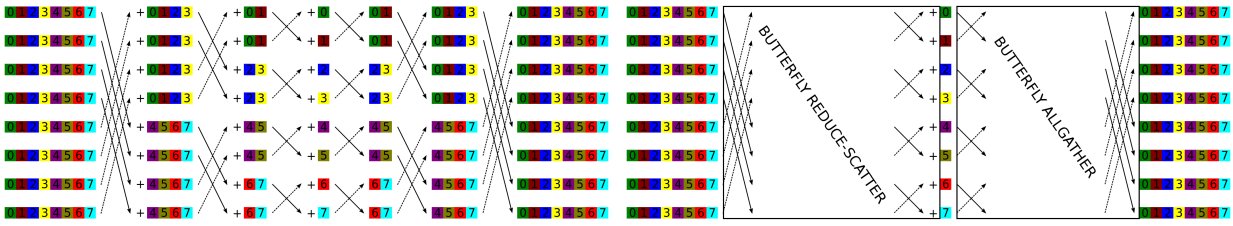


Figure 16: Multi-item all-reduce done by butterfly reduce-scatter and all-gather protocols. It still retains a recursive structure, i.e., a butterfly all-reduce will (1) sum different halves of the data owned by respective pairs of processors, (2) perform all-reduce recursively among the the two halves of the processors, (3) all-gather the resulting data among the same pairs and return.

Multi-item all-reduce done by butterfly reduce-scatter and all-gather protocols still retains a recursive structure as shown in Figure 17. If operating on k processors, the all-reduce pairs processor i with processor $i + k/2$ or $i - k/2$ and

1. has each processor interchange half of their items with their pairing neighbor and sums its respective half,
2. performs an all-reduce recursively on the two-way partition of processors induced by the pairings,
3. has each processor collect the results of the all-reduce obtained by its pairing neighbor.

This butterfly all-reduce protocol is specified in full detail in Algorithm 1.

We can compare these algorithms to the cost of single-item collectives executed with a larger message:

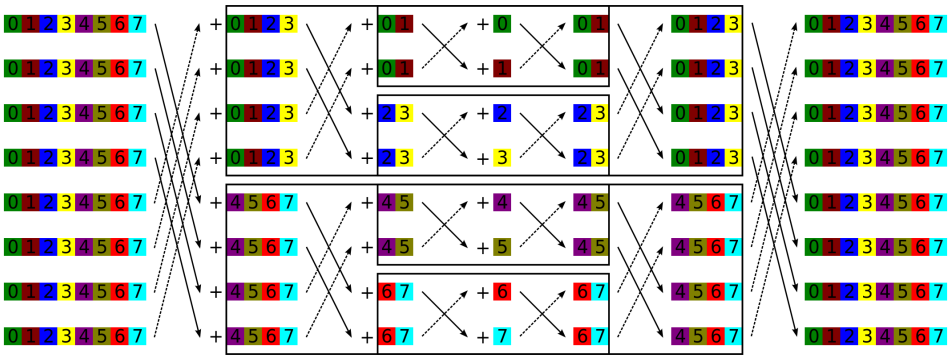


Figure 17: Nested/recursive structured of butterfly all-reduce protocol.

Algorithm 1 $[v] \leftarrow$ All-reduce(A, Π)

Require: Π has P processors, A is a $s \times P$ matrix, $\Pi(j)$ owns $A(1:s, j)$
  if $P = 1$ **then** Return $v = A$
  end if
  Define ranges $L = 1 : \frac{s}{2}$ and $R = \frac{s}{2} + 1 : s$
  Let $x = j + \frac{P}{2}$ and $y = j - \frac{P}{2}$
  if $j \leq P/2$ **then**
    $\Pi(j)$ sends $A(R, j)$ to $\Pi(x)$ and receives $A(L, x)$ from $\Pi(x)$
    $[v(L)] \leftarrow$ All-reduce$\Big(A(L, j) + A(L, x), \Pi(1 : \frac{P}{2})\Big)$
    $\Pi(j)$ sends $v(L)$ to $\Pi(x)$ and receives $v(R)$ from $\Pi(x)$
  else
    $\Pi(j)$ sends $A(L, j)$ to $\Pi(y)$ and receives $A(R, y)$ from $\Pi(y)$
    $[v(R)] \leftarrow$ All-reduce$\Big(A(R, j) + A(R, y), \Pi(\frac{P}{2} + 1 : P)\Big)$
    $\Pi(j)$ sends $v(R)$ to $\Pi(y)$ and receives $v(L)$ from $\Pi(y)$
  end if
Ensure: Every processor owns $v = [v(L); v(R)]$, where $v(i) = \sum_{j=1}^{P} A(i, j)$

- 1-D mesh: $T_p^{\text{si-bcast-1D}}(s) = (p-1)\,(\alpha + \beta s)$,
- 2-D mesh: $T_p^{\text{si-bcast-2D}}(s) = 2(\sqrt{p}-1)\,(\alpha + \beta s)$,
- hypercube: $T_p^{\text{si-bcast-hcube}}(s) = \log_2 p\,(\alpha + \beta s)$.

The hypercube cost corresponds to a binomial tree collective that is optimal for the single-item case. The cost of a broadcast done via butterfly collective protocols for scatter and all-gather is

$$T_p^{\text{bcast}}(s) = 2T_p^{\text{all-gather}}(s) = 2\log_2 p\alpha + 2\beta s.$$

As we can see, this protocol can be better than the binomial tree on a hypercube, $T^{\text{si-bcast-hcube}}$, by a factor of $\log_2(p)/2$ for large messages, but can be slower by a factor of 2 for small messages. Furthermore, the butterfly collective protocol for broadcast is always within a factor of 2 of optimality.

To design bandwidth-efficient broadcasts for specific network topologies, such as meshes, we can take any spanning tree of the network and pipeline messages through it. For instance, if we pipeline messages using **packets** of size b in a 1-D mesh topology, the communication time becomes

$$T_p^{\text{1D-pipe}}(s, b) = (p + s/b - 2)(\alpha + b\beta s),$$

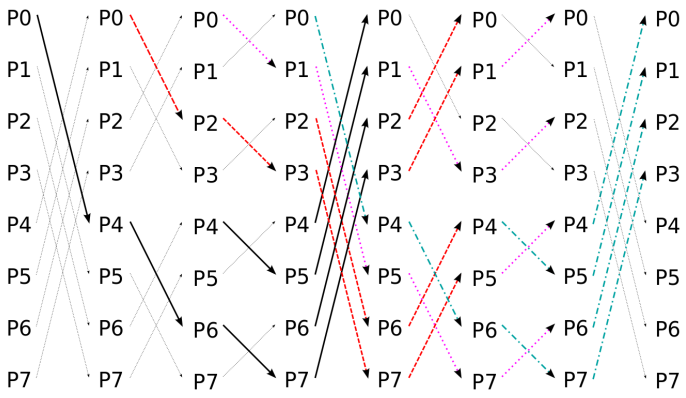


Figure 18: Depiction of pipelined wrapped-butterfly broadcast, which alternates among $\log_2(p)$ different binomial spanning trees of the hypercube to route each successive packet.

as each packet takes $\alpha + b\beta s$ time to travel across a link, the last packet will leave the root after $s/b - 1$ other sends and arrive at its destination $p - 1$ passes later. As we can see, a variable packet size allows us to balance latency and bandwidth costs associated with a message. A similar technique can be applied for higher-dimensional mesh topologies. An alternative approach is to build multiple edge-disjoint spanning trees in the network and pipeline different parts of the message along each of these. This approach can be better, as edge-disjoint spanning trees can operate concurrently, which allows us to reduce delays associated with each spanning tree. A particularly good example of this is the pipelined double binary tree, where delays associated with the need to route a packet to the left child before a right child are eliminated by concurrent operation of another binary tree [50].

5.2.3 State of the Art: Optimal Broadcasting

For any broadcasting protocol that uses a static packet size to communicate messages, a lower bound on the communication time is

$$T_p^{\text{bcast}}(s) \geq (\sqrt{\alpha \log_2(p-1)} + \sqrt{\beta s})^2.$$

This lower bound can be shown by taking into consideration (1) the time it takes for the root to send out all packets and (2) the minimum amount of time in which the last packet can reach all processors [31].

The butterfly-based broadcast protocol we introduced attains this bound when $\alpha \log_2(p-1) = \beta s$, and is otherwise always within a factor of 2. Interestingly, the protocol does not fall in the space of those considered by the lower bound, as it uses variable size messages. It is not known whether a variable-message broadcast protocol can achieve a time faster than the above lower bound.

However, there are known broadcast protocols [31,49,57] that use a static packet size and attain the bound for arbitrary p, s, α, β. These optimal broadcast protocols can also be implemented on hypercube network topologies. Butterflies again provide a convenient representation for the these broadcast protocols, but instead of using recursive halving and recursive doubling, which propagate information according to a butterfly network and a reverse butterfly network, we concatenate butterfly networks and consider various spanning trees from the root processor.

Figure 18 demonstrates an optimal packet-based broadcast protocol, which leverages different binomial spanning trees of the hypercube network. These are naturally enumerated by considering the binomial

trees induced by starting from processor 0 at successive levels of the butterfly network schematic. Picking the optimal packet size allows the protocol to attain the aforementioned lower bound.

5.2.4 Scan

The butterfly communication protocols we demonstrate for collective communication are also effective execution patterns for other algorithms.

An example of particular importance, is the **scan** or **prefix sum** operation.

Scan – each of p processors starts with s items, given by a column in $s \times p$ matrix A, and we compute prefix sums of each row, so that each processor owns a columns of $s \times p$ matrix B, where $B_{ij} = \sum_{k=1}^{j-1} A_{ik}$ and $\sum$ applies any specified associative operation $\oplus$ (e.g.,sum, product, max, min, logical OR, logical AND).

In the parallel algorithms literature, single-item scans ($s = 1$) are widely prevalent and regarded as fundamental primitives [9, 26]. The efficacy of prefix sum as a parallel primitive is its parallelizability. We can execute the scan operation in essentially the same communication time as all-reduce [50].

Define the segmented scan operation as $B = S(A)$. Separate A and B into odd and even columns,

$$A_{\text{odd}} = [A(:,1), A(:,3), \ldots A(:,P-1)],$$
$$A_{\text{even}} = [A(:,2), A(:,4), \ldots A(:,P)].$$

Now observe that $B_{\text{even}} = S(A_{\text{even}} + A_{\text{odd}})$ and that $B_{\text{odd}} = B_{\text{even}} + A_{\text{even}}$. Single-item prefix scan algorithms use this recursive structure to implement the algorithm in bottom-up and top-down traversal of a tree. Algorithm 2 provides a butterfly-based scan algorithm that is similar to all-reduce (Algorithm 1), except that some information must be stored before every recursive call and reused thereafter. The cost

Algorithm 2 $[B] \leftarrow \text{Scan}(A, \Pi)$

Require: Π has P processors, A is a $s \times p$ matrix, $\Pi(j)$ owns $A(1:s, j)$
  if $P = 1$ **then** Return $B = A$
  end if
  Define ranges $L = 1 : \frac{s}{2}$ and $R = \frac{s}{2} + 1 : s$
  Let $x = j + 1$ and $y = j - 1$
  if $j \equiv 0 \mod 2$ **then**
    $\Pi(j)$ sends $A(R, j)$ to $\Pi(x)$ and receives $A(L, x)$ from $\Pi(x)$
    $[W(L, j)] \leftarrow \text{Scan}\Big(A(L, j) + A(L, x), \Pi(1 : \frac{p}{2})\Big)$
    $\Pi(j)$ sends $W(L, j)$ to $\Pi(x)$ and receives $W(R, x)$ from $\Pi(x)$
    $B(:, j) = [W(L, j); W(R, j)]$
  else
    $\Pi(j)$ sends $A(L, j)$ to $\Pi(y)$ and receives $A(R, y)$ from $\Pi(y)$
    $[W(R, j)] \leftarrow \text{Scan}\Big(A(R, j) + A(R, y), \Pi(\frac{p}{2} + 1 : p)\Big)$
    $\Pi(j)$ sends $W(R, j)$ to $\Pi(y)$ and receives $W(L, y)$ from $\Pi(y)$
    $B(:, j) = [W(L, j) + A(L, j); W(R, j) + A(R, j)]$
  end if
Ensure: Processor j owns $B(:, j)$, such that $B(i, j) = \sum_{k=1}^{j-1} A(i, k)$

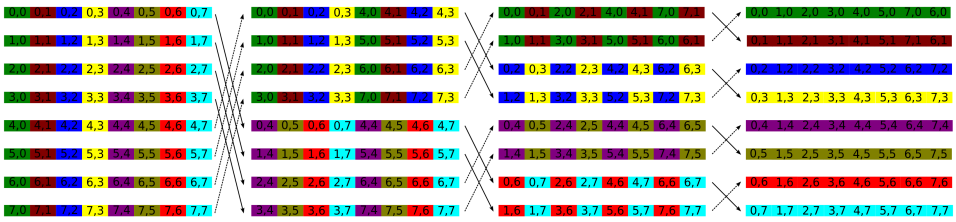


Figure 19: Butterfly collective protocol for all-to-all exchange.

recurrence for the above algorithm is

$$T_p^{\text{scan}}(s) = T^{\text{scan}}(s/2, p/2) + 2(\alpha + \beta s/2) = 2(\alpha \log_2 p + \beta s).$$

As we see, the scan operation (which is also provided in the MPI standard [24]) can be done in the same fashion and cost as a seemingly unrelated basic collective communication operation such as broadcast.

5.2.5 All-to-All

An important collective communication operation that has a somewhat greater cost than the collectives previously discussed is the *all-to-all*. A convenient representation of all-to-all is as a matrix transpose, going from a row-blocked to a column-blocked layout, or vice versa. The major distinction is that the overall input and output scales with the the number of processors, and unlike all-reduction or scan, is irreducible.

All-to-all – Every processor exchanges messages of size s/p with every other processor.

We can use a butterfly protocol to execute all-to-all, as demonstrated in Figure 19. The size of the messages sent is the same at each level, however, yielding an overall communication time of

$$T_p^{\text{all-to-all}}(s) = \alpha \log_2(p) + \beta s \log_2(p)/2,$$

which is a factor of $\log_2(p)$ higher in bandwidth cost than the collectives we have previously analyzed. It is possible to do all-to-all with less bandwidth cost (as low as βs by sending directly to targets) but at the cost of more messages. In particular, if all messages are sent directly, the cost of all-to-all is

$$T_p^{\text{all-to-all-direct}}(s) \approx \alpha p + \beta s.$$

We are not aware of a complete characterization of the latency-bandwidth tradeoff in all-to-all communication within a point-to-point messaging model.

6 General References

General references on parallel computing

- G. S. Almasi and A. Gottlieb, *Highly Parallel Computing*, 2nd ed., Benjamin/Cummings, 1994
- J. Dongarra, et al., eds., *Sourcebook of Parallel Computing*, Morgan Kaufmann, 2003
- A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing*, 2nd. ed., Addison-Wesley, 2003

- G. Hager and G. Wellein, *Introduction to High Performance Computing for Scientists and Engineers*, Chapman & Hall, 2011
- K. Hwang and Z. Xu, *Scalable Parallel Computing*, McGraw-Hill, 1998
- A. Y. Zomaya, ed., *Parallel and Distributed Computing Handbook*, McGraw-Hill, 1996

General references on parallel architectures

- W. C. Athas and C. L. Seitz, Multicomputers: message-passing concurrent computers, *IEEE Computer* 21(8):9-24, 1988
- D. E. Culler, J. P. Singh, and A. Gupta, *Parallel Computer Architecture*, Morgan Kaufmann, 1998
- M. Dubois, M. Annavaram, and P. Stenström, *Parallel Computer Organization and Design*, Cambridge University Press, 2012
- R. Duncan, A survey of parallel computer architectures, *IEEE Computer* 23(2):5-16, 1990
- F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufmann, 1992

General references on interconnection networks

- L. N. Bhuyan, Q. Yang, and D. P. Agarwal, Performance of multiprocessor interconnection networks, *IEEE Computer* 22(2):25-37, 1989
- W. J. Dally and B. P. Towles, *Principles and Practices of Interconnection Networks*, Morgan Kaufmann, 2004
- T. Y. Feng, A survey of interconnection networks, *IEEE Computer* 14(12):12-27, 1981
- I. D. Scherson and A. S. Youssef, eds., *Interconnection Networks for High-Performance Parallel Computers*, IEEE Computer Society Press, 1994
- H. J. Siegel, *Interconnection Networks for Large-Scale Parallel Processing*, D. C. Heath, 1985
- C.-L. Wu and T.-Y. Feng, eds., *Interconnection Networks for Parallel and Distributed Processing*, IEEE Computer Society Press, 1984

References

[1] N. R. Adiga, M. A. Blumrich, D. Chen, P. Coteus, A. Gara, M. E. Giampapa, P. Heidelberger, S. Singh, B. D. Steinmacher-Burow, T. Takken, et al. Blue Gene/L torus interconnection network. *IBM Journal of Research and Development*, 49(2.3):265–276, 2005.

[2] T. Agarwal, A. Sharma, A. Laxmikant, and L. V. Kalé. Topology-aware task mapping for reducing communication contention on large parallel machines. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 10–pp. IEEE, 2006.

[3] A. Aggarwal, A. K. Chandra, and M. Snir. Communication complexity of PRAMs. *Theoretical Computer Science*, 71(1):3 – 28, 1990.

[4] Y. Ajima, S. Sumimoto, and T. Shimizu. Tofu: A 6D mesh/torus interconnect for exascale computers. *Computer*, 42(11), 2009.

[5] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: incorporating long messages into the LogP model – one step closer towards a realistic model for parallel computation. In *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '95, pages 95–105, New York, NY, USA, 1995. ACM.

[6] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, et al. The landscape of parallel computing research: A view from Berkeley. Technical report, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[7] M. Besta and T. Hoefler. Slim Fly: A cost effective low-diameter network topology. In *High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for*, pages 348–359. IEEE, 2014.

[8] G. Bilardi, K. T. Herley, A. Pietracaprina, G. Pucci, and P. Spirakis. BSP vs LogP. In *Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*, pages 25–32. ACM, 1996.

[9] G. E. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on computers*, 38(11):1526–1538, 1989.

[10] M. Bohr. A 30 year retrospective on Dennard's MOSFET scaling paper. *IEEE Solid-State Circuits Society Newsletter*, 12(1):11–13, Winter 2007.

[11] K. M. Bresniker, S. Singhal, and R. S. Williams. Adapting to thrive in a new economy of memory abundance. *Computer*, 48(12):44–53, 2015.

[12] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to UPC and language specification. Technical report, Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, 1999.

[13] E. Chan, M. Heimlich, A. Purkayastha, and R. van de Geijn. Collective communication: theory, practice, and experience. *Concurrency and Computation: Practice and Experience*, 19(13):1749–1783, 2007.

[14] D. Chen, N. A. Eisley, P. Heidelberger, R. M. Senger, Y. Sugawara, S. Kumar, V. Salapura, D. L. Satterfield, B. Steinmacher-Burow, and J. J. Parker. The IBM Blue Gene/Q interconnection network and message unit. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 26:1–26:10, New York, NY, USA, 2011. ACM.

[15] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: towards a realistic model of parallel computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '93, pages 1–12, New York, NY, USA, 1993. ACM.

[16] W. J. Dally. Performance analysis of k-ary n-cube interconnection networks. *IEEE Transactions on Computers*, 39(6):775–785, Jun 1990.

[17] W. J. Dally. Virtual-channel flow control. *IEEE Transactions on Parallel and Distributed systems*, 3(2):194–205, 1992.

[18] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

[19] J. J. Dongarra, P. Luszczek, and A. Petitet. The LINPACK benchmark: past, present and future. *Concurrency and Computation: practice and experience*, 15(9):803–820, 2003.

[20] A. Farley. Broadcast time in communication networks. *SIAM Journal on Applied Mathematics*, 39(2):385–390, 1980.

[21] R. P. Feynman. Simulating physics with computers. *International journal of theoretical physics*, 21(6):467–488, 1982.

[22] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE transactions on computers*, 100(9):948–960, 1972.

[23] W. Gropp and E. Lusk. Using MPI-2. In *12th European PVM/MPI Users' Group Meeting-Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 2005.

[24] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable parallel programming with the message-passing interface*. MIT Press, Cambridge, MA, USA, 1994.

[25] P. R. Hafner. Geometric realisation of the graphs of McKay–Miller–Širáň. *Journal of Combinatorial Theory, Series B*, 90(2):223–232, 2004.

[26] M. Harris, S. Sengupta, and J. D. Owens. Parallel prefix sum (scan) with CUDA. *GPU gems*, 3(39):851–876, 2007.

[27] M. T. Heath. A tale of two laws. *The International Journal of High Performance Computing Applications*, 29(3):320–330, 2015.

[28] W. D. Hillis. *The connection machine*. MIT press, 1989.

[29] T. Hoefler and M. Snir. Generic topology mapping strategies for large-scale parallel architectures. In *Proceedings of the international conference on Supercomputing*, pages 75–84. ACM, 2011.

[30] A. J. Hoffman and R. R. Singleton. On Moore graphs with diameters 2 and 3. *IBM Journal of Research and Development*, 4(5):497–504, 1960.

[31] S. L. Johnsson and C.-T. Ho. Optimum broadcasting and personalized communication in hypercubes. *IEEE Trans. Comput.*, 38:1249–1268, September 1989.

[32] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, et al. In-datacenter performance analysis of a tensor processing unit. *arXiv preprint arXiv:1704.04760*, 2017.

[33] R. M. Karp, A. Sahay, E. E. Santos, and K. E. Schauser. Optimal broadcast and summation in the LogP model. In *Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures*, pages 142–153. ACM, 1993.

[34] R. E. Kessler and J. L. Schwarzmeier. CRAY T3D: A new dimension for Cray Research. In *Compcon Spring'93, Digest of Papers.*, pages 176–182. IEEE, 1993.

[35] J. Kim, W. J. Dally, S. Scott, and D. Abts. Technology-driven, highly-scalable Dragonfly topology. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 77–88, Washington, DC, USA, 2008. IEEE Computer Society.

[36] N. S. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, J. S. Hu, M. J. Irwin, M. Kandemir, and V. Narayanan. Leakage current: Moore's law meets static power. *Computer*, 36(12):68–75, Dec 2003.

[37] C. E. Leiserson. Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE transactions on Computers*, 100(10):892–901, 1985.

[38] J. Liu, J. Wu, and D. K. Panda. High performance RDMA-based MPI implementation over InfiniBand. *International Journal of Parallel Programming*, 32(3):167–198, 2004.

[39] C. A. Mack. Fifty years of Moore's law. *IEEE Transactions on semiconductor manufacturing*, 24(2):202–207, 2011.

[40] B. D. McKay, M. Miller, and J. Širáň. A note on large graphs of diameter two and given maximum degree. *Journal of Combinatorial Theory, Series B*, 74(1):110–118, 1998.

[41] C. Moore and M. Nilsson. Parallel quantum computation and quantum codes. *SIAM Journal on Computing*, 31(3):799–815, 2001.

[42] M. A. Nielsen and I. Chuang. Quantum computation and quantum information, 2002.

[43] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global Arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing*, 10:169–189, 1996.

[44] R. W. Numrich and J. Reid. Co-Array Fortran for parallel programming. In *ACM Sigplan Fortran Forum*, volume 17, pages 1–31. ACM, 1998.

[45] K. Olukotun and L. Hammond. The future of microprocessors. *Queue*, 3(7):26–29, 2005.

[46] J. Pješivac-Grbović, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra. Performance analysis of MPI collective operations. *Cluster Computing*, 10:127–143, June 2007.

[47] V. Ramachandran, B. Grayson, and M. Dahlin. Emulations between QSM, BSP, and LogP: a framework for general-purpose parallel algorithm design. In *Symposium on Discrete Algorithms: Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*, volume 17, pages 957–958, 1999.

[48] R. M. Russell. The CRAY-1 computer system. *Communications of the ACM*, 21(1):63–72, 1978.

[49] Y. Saad and M. H. Schultz. Data communication in hypercubes. *Journal of Parallel and Distributed Computing*, 6(1):115 – 135, 1989.

[50] P. Sanders, J. Speck, and J. L. Träff. Two-tree algorithms for full bandwidth broadcast, reduction and scan. *Parallel Computing*, 35(12):581–594, 2009.

[51] R. R. Schaller. Moore's law: past, present and future. *IEEE Spectrum*, 34(6):52–59, Jun 1997.

[52] J. Shalf, S. Dosanjh, and J. Morrison. *Exascale Computing Technology Challenges*, pages 1–25. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[53] D. E. Shaw, J. Grossman, J. A. Bank, B. Batson, J. A. Butts, J. C. Chao, M. M. Deneroff, R. O. Dror, A. Even, C. H. Fenton, et al. Anton 2: raising the bar for performance and programmability in a special-purpose molecular dynamics supercomputer. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 41–53. IEEE Press, 2014.

[54] E. Strohmaier. Highlights of the 45th TOP500 List. International Conference on Supercomputing (ICS), 2015.

[55] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of collective communication operations in MPICH. *International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.

[56] A. Tiskin. The bulk-synchronous parallel random access machine. *Theoretical Computer Science*, 196(1):109–130, 1998.

[57] J. L. Träff and A. Ripke. Optimal broadcast for fully connected networks. In *HPCC*, pages 45–56. Springer, 2005.

[58] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, Aug. 1990.

[59] Wikimedia Commons. Fat tree network. [Online: accessed on August 27, 2017].

[60] Wikimedia Commons. Transistor count and Moore's law - 2011. [Online: accessed on August 27, 2017].