Algorithms for Collective Communication Preliminaries, Basics:
Version 1 (HPC Lecture WS 2023)

Jesper Larsson Träff
Faculty of Informatics
TU Wien, Austria

October 5, 2023

Contents

Chapter 1

Introduction

Parallel computing is the discipline of getting some finite set of more or less independent processors that together comprise a parallel computer to collaborate closely in order to solve some given computational problem in an efficient way. Parallel computers abound, in theory and in reality, and come in many different flavors, counting from a few to thousands, ten thousands, hundred thousands to even millions of processors. A particular kind of parallel computer system and theoretical abstraction thereof is the distributed memory computer where otherwise independent, only loosely synchronized processors with their own local data and programs collaborate by explicitly exchanging data through some communication medium. The exchange of data involving all or at least a large subset of the processors available at some given time with the purpose of achieving a desired organization of the data local to the processors, possibly combined with the application of some operation on the data, is inevitable in any non-trivial, parallel, distributed memory algorithm. It is therefore useful and fruitful to study such collective data exchange and reduction operations in isolation, in order to distill out patterns that are useful for algorithms and concrete applications, and in order to develop good algorithms and implementations for such communication patterns, taking into account as accurately as possible the fundamental characteristics of different distributed memory machines and communication networks.

Premises: The fundamental premises and abstractions of *parallel computing* thus are as follows. Some given, *well-defined, computational problem* is to be solved on a *parallel computer* consisting of a *finite number of processors* capable of executing any desired program. The parallel program may consist of programs for each processor or a single, same program executed by all processors. The processors are assumed to be *dedicated* to and under full control of the parallel program solving the problem, and not doing anything else during that time. The computational problem is assumed to be large, either in the time required to solve the problem, or in the memory space needed to hold the problem, the result, and the data structures used in the program, or in both time and space. In *distributed memory parallel computing*, the processors do not have a shared memory through which they can exchange data by simply writing to and reading from this memory. Instead, all or most data exchange has to be effected by communication through a *communication network*. Both processors and network are *reliable*, and no failures will happen during the execution of the program. The aim is to solve the given problem as

efficiently under given metrics as possible under the given characteristics of processors and network. Typical metrics are the maximum time any one processor is kept occupied (the *time*), the total time the dedicated processors are kept occupied with the problem (the *work*), the total memory resources used, the maximum amount of memory used by any one processor, the energy used, and others.

Commentary: All these assumptions are quite crucial and noteworthy. There is usually little point in solving small (in time and memory space) problems in parallel. First, such problems can readily be solved sequentially, second, there is almost always a non-negligible overhead in solving a problem by a parallel algorithm on a parallel computer. Parallel computing, in the *High-Performance Computing* (HPC) variation, is about large problems on large systems. The parallel computing system is assumed to be dedicated, and parallel computing is not about provisioning of such resources. Eventually, there is a price to be paid for occupying such a system, and minimizing this cost is most often a primary objective. Conversely, the time for a parallel program to complete is defined and measured as the time in which the parallel computer is needed which is the same as the time for the overall last, slowest processor to finish its part of the computation, assuming that all processors start at the same time (as far as this notion makes sense). Another crucial, parallel computing assumption is that the system is reliable in all respects as long as the parallel program is otherwise correct. For the exchange of messages, reliability means that messages are neither lost, nor corrupted. Also, message order is respected. A sequence of messages sent from one processor to another will be received in that order. The prime concern of parallel computing is with *efficiency*, that is with minimizing the amount of resources needed to solve the given problem. The goal is to find efficient algorithms that are good or even optimal in the algorithmic metrics considered, and efficient programs that utilize the processors and network of the parallel computing system well. When designing algorithms, the processors can be assumed to have extensive, prior *knowledge* about the system, like the number of processors in total to be involved in the computation, the structure and properties of the communication network, the programs executed by the other processors, and possibly also about the distribution of data over the processors.

Consequences: With the focus on problems and efficiency, and the premises of reliability and full knowledge, parallel computing is quite different from *distributed computing* [Ray13, Ray18] which is concerned with cooperation under partial knowledge with sometimes faulty media, and with more focus on correctness properties than efficiency. The two fields are related, and techniques and results from one is often relevant for the other. Likewise, parallel computing is different from all sorts of cloud and grid computing where the focus is typically more on provisioning of resources. Certain models of computation like cellular automata [Cod68, TM87] and systolic arrays of finite state automata are not considered parallel computers (here).

Aims: Any parallel algorithm for solving a large, algorithmically non-trivial problem will need communication between the processors. In some cases this communication may be restricted to distributing parts of the input data in some pattern over the processors, and in collecting parts of the result from the processors. More often, repeated communication between all, or subsets of the processors will be required for bringing data distributed

over the processors into some other desired distribution. Such data exchange patterns where more than just a pair of processors interact are often called *collective operations* and collective operations are the focus of study in the following. Collective patterns and operations are useful for expressing algorithms and when properly implemented can serve as useful building blocks for programming at a higher level, relieving the programmer from much complicated work.

Chapter notes

Parallel (distributed memory) computing has been central in computer science since the formation of the discipline, and proliferated in the 1970ties to 1990ties, both theoretically and practically, see for instance [Akl89, Qui87, Kun80, AG94, KGGK94, GR88, JáJ92]. More recent, general texts on parallel computing basics for distributed memory and high-performance systems are the books by Rauber and Rünger [RR10], the book by Schmidt et al. [SGDHS18], and also the book by Lin and Snyder [LS08]. Specifically High-Performance Computing (HPC) oriented in the sense of achieving high processor efficiency in practice on given general-purpose processor architectures is the book by Hager and Wellein [HW11], but also the much less detailed book by Levesque [Lev11]. A superficial tour of High-Performance Computing with some interesting historical detail can be found in the book by Sterling et al. [SAB18]. More concretely useful is the "Sourcebook" by Dongarra et al. [DFF+03]. An early, forgotten but nevertheless influential reference with much early information on collective operations was the book by Fox and coworkers [FJL+88].

Chapter 2

Communication, Algorithms and Performance

This chapter introduces directed graphs as a tool for designing and analyzing algorithms for message passing parallel computers. It introduces graphs and other considerations to describe significant characteristics of the communication architecture of distributed memory parallel computers. It discusses various cost models for the analysis of algorithms designed using specific communication structures when mapped to specific hardware architectures. Finally, it introduces notation for describing collective communication problems to be solved by algorithms eventually running on concrete hardware architectures. It recapitulates essential aspects and functionality of one specific programming model and concrete framework for implementing (collective) algorithms for concrete systems, namely the Message-Passing Interface MPI [MPI15, MPI21].

2.1 Graphs and Communication Structures

Parallel computers and communication structures: An idealized, distributed memory parallel computer consists of a finite number of processors with sufficient (unbounded) local memory and own possibly different programs, each with a unique identification. Let the number of processors be p. Each processor will be assumed to have a unique *rank* in the range $0, 1, \ldots, p-1$ by which it can be identified and identify itself.

A *communication structure* describes possible communication between pairs of processors, either as required by an algorithm, or as allowed by a communication network [FL94]. A communication structure is modeled as a *finite, directed graph* $G = (V, E)$ with vertex set V representing the processors, thus $p = |V|$, and set of directed edges $E, E \subseteq V \times V$ representing communication that can and possibly will take place at points or internals in time from one processor to another or between some pair of processors u and v in V. Processor $u \in V$ is allowed to send data as a *message* to processor $v \in V$ only if there is a directed edge $(u, v) \in E$ in the communication structure. A processor may be required and allowed to send a message to itself, so (u, u) may be an edge in a communication structure. Such edges are called *(self)loops*. The term *arc* for a directed edge will rarely be used. For any vertex $u \in V$, all vertices $v \in V$ for which there is an edge $(u, v) \in E$ are said to be *adjacent* to u, as are the defining edges (u, v) that are also called *incident* to (on) u (and v). Edges $(u, v) \in E$ incident to $u \in V$ are also said to be the *outgoing*

edges from u. Conversely, the edges of the form $(v, u) \in E$ for vertices $v \in V$ are the *incoming* edges of u. The notation $u \to v$ is often suggestive for the directed edge (or arc) $(u, v) \in E$ from processor u to processor v, and will mostly be used. A sequence of d edges, or $d + 1$ vertices, $d > 0$, $u_0 \to u_1 \to \cdots \to u_d$ where $(u_i, u_{i+1}) \in E$ (equivalently written $u_i \to u_{i+1} \in E$) for $i = 0, \ldots, d - 1$ is called a (directed) *path* from u_0 to u_d of *length* d. If all vertices u_i and u_j are distinct, the path is said to be *simple*. If the last and first vertices are (allowed to be) equal, $u_d = u_0$, the path is a (simple) *cycle* of length d. The *empty path* consisting of no edges is sometimes denoted ϵ. For any two processors $u, v \in V$ there may be multiple paths between u and v in a communication structure. The length of a path with the smallest possible number of edges (a *shortest path*) is called the *distance* from u to v, and is denoted by $\text{dist}(u, v)$. If there is no directed path from u to v, then $\text{dist}(u, v) = \infty$. Communication structures for which there exist processors u and v for which both $\text{dist}(u, v) = \infty$ and $\text{dist}(v, u) = \infty$ are usually not interesting, since communication problems where information (messages) has to be transferred from processor u to another processor v cannot be solved if there is no path from u to v in the communication structure. A directed graph is said to be *strongly connected* if for any two vertices u and v, there is a directed path $u \to \ldots \to v$. The existence of such a path will be denoted by $u \rightsquigarrow v$.

For an unspecified graph G, the vertex set V of G is denoted and can be found as $V(G)$ and the edge set is denoted and can be found as $E(G)$. For any two directed graphs G' and G'', the graph union $G' \cup G''$ is the graph $G' \cup G'' = (V(G') \cup V(G''), E(G') \cup E(G''))$. The *transpose* or *reverse* of G is the graph $(V(G), \{(v, u) \mid (u, v) \in E(G)\})$ formed by reversing the direction of all edges of G. The number of vertices in a graph $G, |V(G)|$, is sometimes called the *order* of G, and the number of (outgoing) edges, $|E(G)|$, the *size* of G.

A graph H is a *subgraph* of a graph G if $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. We say that a communication structure or graph H can be *embedded* into or *mapped* onto a communication structure or graph G if there is an injective function $\Pi : V(H) \to V(G)$ (that is, $\Pi(u) = \Pi(v)$ only if $u = v$) such that if $(u, v) \in E(H)$ then $(\Pi(u), \Pi(v)) \in E(G)$. An algorithm that can be executed on the communication structure H can therefore also be executed on the communication structure G, since all processors and communication edges of H map to processors and edges of G; but processor u (in G) will change "role" or "name" to $\Pi(u)$ when the algorithm is executed on H. We do not want to consider embeddings where more than one processor of H is mapped to the same processor in G, and therefore mappings are required to be injective. Given a subset of vertices $U \subseteq V(G)$, the *induced subgraph* is the graph $(U, \{(u, v)\} \mid u \in U, v \in U, (u, v) \in E(G))$.

A graph $G = (V, E)$ is *bidirected* if for each $(u, v) \in E$ also the *reverse edge* $(v, u) \in E$. The *bidirected completion* of a graph G is the union of G and its transpose, that is the graph $(V(G), \{(u, v), (v, u) \mid ((u, v) \in E(G)\})$. The undirected completion of a directed graph G is a graph where all edges (arcs) are treated as unordered pairs. The terms bidirected and undirected will be used interchangeably when the difference is not important. For edges in undirected graphs, it is sometimes suggestive to write $u - v$ for $(u, v) \in E$. For bidirected graphs, the shorthand $u \leftrightarrow v$ denotes the two directed edges $u \to v$ and $v \to u$. The definitions of path and cycle and distance extend to undirected graphs.

For strongly connected, directed graphs, or undirected graphs $G = (V, E)$, the *diameter*,

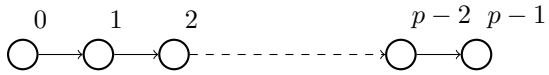


Figure 2.1: A linear processor array L_p for some number of processors p in the canonical representation.

denoted by $\mathrm{Diam}(G)$, is the largest distance between any two vertices in V, that is

$$\mathrm{Diam}(G) \quad = \quad \max\{\mathrm{dist}(u,v) \mid u,v \in V\}$$

The diameter of such graphs is finite. For a directed or undirected graph G, $\mathrm{Diam}(G) = \infty$ if there are vertices u and v between which no (directed) path exists. Again, such *disconnected* or not strongly connected graphs are uninteresting as communication structures for many of the problems we will consider. For directed graphs, this means that there are processors that can only in at most one direction communicate with each other. Communication problems, where information flows from one or a few vertices to (all) other vertices, may not require strongly connected, finite diameter communication structures.

The *(out)degree* of a vertex $u \in V$ is the number of vertices (or outgoing edges) that are adjacent (incident) to u, that is

$$\deg(u) \quad = \quad |\{v \in V \mid (u,v) \in E\}| \quad .$$

Since any arc $(u,v) \in E$ is incident to two vertices as outgoing from one vertex and incoming to the other, it obviously holds that $\sum_{u \in V} \mathrm{outdeg}(u) = \sum_{v \in V} \mathrm{indeg}(v)$. For undirected graphs, it holds that $\sum_{u \in V} \deg(u) = 2|E|$ since each edge is counted twice, once for each of its incident vertices.

The degree $\Delta(G)$ of a directed or undirected graph $G = (V,E)$ is the maximum over the degrees of all vertices $u \in V$, that is

$$\Delta(G) \quad = \quad \max_{u \in V} \deg(u) \quad .$$

A graph is called *k-regular* if all vertices $u \in V$ have the same (out)degree $\deg(u) = k$. Regularity or almost regularity (in the form of some maximum degree upper bound) is sometimes an important property for certain (pipelined) algorithms that will be considered.

Common communication structures: Most of the algorithms that will be introduced and discussed in the following chapters follow simple, highly structured communication patterns that can be fitted into corresponding communication structures. We now define and describe a number of common communication structures, which also in part describe important aspects of the communication structures of existing high-performance systems, or systems that have existed in the (recent) past.

Recall that a graph or communication structure $G = (V,E)$ is *isomorphic* to a graph or communication structure $G' = (V',E')$ if G can be embedded into G', and G' can be embedded into G, that is, if there is a bijective mapping (*isomorphism*) $\Pi : V \to V'$ such that $(u,v) \in E$ if and only if $(\Pi(u), \Pi(v)) \in E'$). Graph isomorphism is an equivalence relation, and isomorphic graphs can often be treated as equal, which is then expressed by

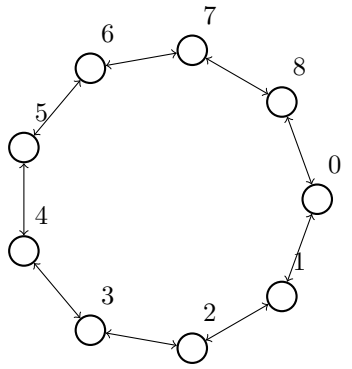


Figure 2.2: The bidirected ring R_9 in canonical representation.

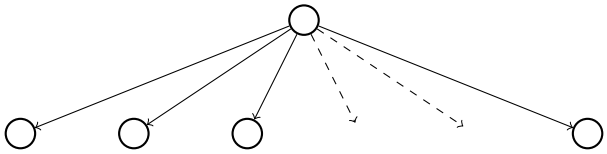


Figure 2.3: An unnumbered, rooted star network S_p directed from the root for some number of processors p.

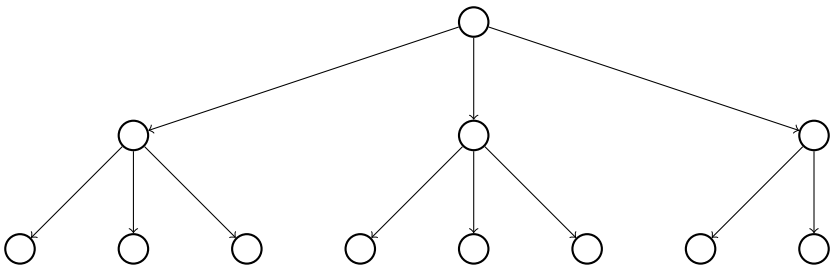


Figure 2.4: An unnumbered 3-ary height 2 tree T_3^2 with a single leaf removed.

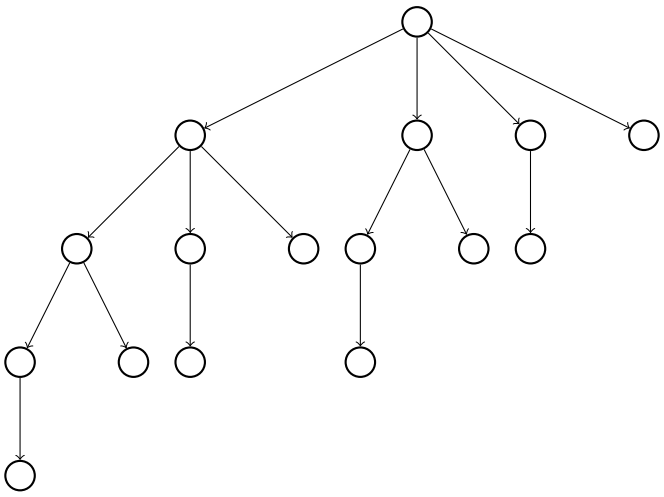


Figure 2.5: An unnumbered, complete binomial tree (k-nomial tree with $k = 2$) of height 4, B_2^4.

$G = G'$. We describe the communication structures by giving a *canonical representation* or *numbering* of the processors in the communication structure, with $V = \{0, \ldots, p-1\}$ for some given number of processors p, and allow for isomorphic graphs to be considered as the same communication structure. As used here, Π will mostly map $V = \{0, 1, \ldots, p-1\}$ onto V itself, $\Pi(V) = V$, and thus be *permutations* (or reorderings) of the processor ranks in the set of processors V. Recall that in general deciding whether two graphs G and G' are isomorphic is a difficult problem, whose complexity is still not known (that is polynomial or outside the class of deterministic polynomial time algorithms) [GJ79, Bab16, Bab19].

Communication algorithms will (in the following chapters) be described assuming some specific communication structure, such that each processor (with unique *rank* in the set $0, 1, \ldots, p-1$) can determine where it is located in the communication structure. Processors are allowed to communicate with other processors by sending and receiving *messages* along the (directed) edges of the assumed communication structure. It is therefore essential that each processor with some rank $r \in \{0, 1, \ldots, p-1\}$ can know, or efficiently compute its incoming and outgoing edges (v, r) and (r, u). To accomplish this, the numbering or canonical representation of the communication structure is important, as are mappings between communication structures. Some representations may be more efficient than others for determining the communication edges fast enough as required by some algorithm, and such considerations will play an important role in the following chapters.

Definition 2.1 (Linear array)**.** *A directed* linear processor array *communication structure is any directed graph that is isomorphic to $L_p = (V, E)$ with $V = \{0, 1, \ldots, p-1\}$ and*

$$E = \{i \to (i+1) \mid 0 \leq i < p-1\} \quad .$$

For processor $i, i < p-1$, the adjacent processor $(i+1)$ is said to be the successor *of i. For processor $i, 0 < i$, the adjacent processor $i-1$ is said to be the* predecessor *of i.*

A linear processor array is a simple path from a vertex u without an incoming edge to a vertex v without an outgoing edge, and is not strongly connected. Each vertex

$u \in V$ except one has outdegree $\deg(u) = 1$ and indegree one as well. For the undirected completion of L_p, all vertices u except two have $\deg(u) = 2$. The length of a longest path is $p-1$, which is also the diameter of the (undirected completion of the) linear array. The number of edges (size) is likewise $p-1$. In the bidirected (strongly connected) completion of a linear array, the distance between any two processors i and j is $\text{dist}(i,j) = |i-j|$. A linear array is shown in Figure 2.1.

Definition 2.2 (Ring)**.** *A* directed ring *communication structure is any directed graph that is isomorphic to* $R_p = (V, E)$ *with* $V = \{0, 1, \ldots, p-1\}$ *and*

$$E = \{i \to (i+1) \bmod p \mid 0 \leq i < p\} \quad .$$

A directed ring is a simple cycle over the processors, and therefore strongly connected. A longest shortest path (from processor i to processor $(i-1) \bmod p$) has length $p-1$. Each processor i has indegree and outdegree $\deg(i) = 1$. The number of edges is p. In the bidirected or undirected completion of a ring, all processors have $\deg(i) = 2$. The diameter of the undirected completion of a directed ring is $\lfloor p/2 \rfloor$. This can be seen as follows. Any processor i can be reached from processor 0 by either the path $0 \to 1 \to \ldots \to i$ of length i, or the path (in the "other direction") $0 \to (p-1) \to (p-2) \to \ldots \to i$ of length $p-i$. Thus, for the distance from processor 0 to processor i it holds that $\text{dist}(0,i) = \min(i, p-i)$. The processor which is farthest away from processor 0 is the processor i where this minimum is the largest possible. This holds for the largest integer i such that $i = p - i \Leftrightarrow 2i = p$, that is $i = \lfloor p/2 \rfloor$. Since the same distance consideration holds for all other processors $j \neq 0$, it follows that $\text{Diam}(R_p) = \lfloor p/2 \rfloor$ for the bi- or undirected directed completion of the ring. The distance between any two processors i and j where $i < j$ is $\min(j-i, p-j+i)$. A bidirected ring is shown in Figure 2.2.

Definition 2.3 (Star)**.** *A* rooted star *communication structure directed from the* root *is any directed graph that is isomorphic to* $S_p = (V, E)$ *with* $V = \{0, 1, \ldots, p-1\}$ *and*

$$E = \{0 \to i \mid 1 \leq i < p\} \quad .$$

The root *is the vertex with outgoing edges to all other vertices.*

A rooted star *directed towards the* root *is any directed graph that is isomorphic to the transpose (reverse) of* S_p*.*

The number of edges in a rooted star is $p-1$. The root processor r has out- or indegree $\deg(r) = p-1$, and all other vertices in- or outdegree $\deg(i) = 1, i \neq r$. The diameter of an undirected star is $\text{Diam}(S_p) = 2$.

Definition 2.4 (Fixed-degree k-ary height d Tree)**.** *A* directed, rooted, fixed-degree k-ary, height d tree *communication structure directed from the* root *(for* $k > 0$*) is any directed graph that is isomorphic to* $T_k^d = (V, E)$ *with* $p = \frac{k^{d+1}-1}{k-1}$ *processors (and for the degenerate case* $k = 1$*, with* $p = d$ *processors),* $V = \{0, 1, \ldots, p-1\}$ *and*

$$E = \{i \to (ik+j) \mid j = 1, 2, \ldots, k \wedge ik + j < p\} \quad .$$

A rooted, k-ary tree *directed towards the* root *is any directed graph that is isomorphic to the transpose of* T_k^d*.*

By definition, the order of T_k^d is $p = \frac{k^{d+1}-1}{k-1}$, and the size (number of edges) $p-1$. Vertex (processor) 0 has no incoming edge and is called the *root* (processor). Vertices without outgoing edges are called *leaves*. Vertices that are not leaves are called *interior*. For each interior vertex i, the adjacent vertices $ik+j$ are called the *children* of i. For any processor i' that is not the root, $i' > 0$, there are unique i and $j, 1 \leq j \leq k$, such that $i' = ik + j$, namely $i = \lfloor (i'-1)/k \rfloor$ and $j = ((i'-1) \bmod k) + 1$. The unique vertex i is called the *parent* of i'. The processors $ik + j'$ with $1 \leq j' \leq k, j' \neq j$ are called the *siblings* of i'; sometimes, the siblings with $j' < j$ are called the *left siblings*, and the siblings with $j' > j$ the *right siblings*. The distance from the root vertex 0 to a vertex i is called the *depth* of i; when the tree is directed towards the root, the depth is the distance from i to the root. Conversely, The *height* of a vertex i is the longest distance to a leaf that can be reached from i. The height of a tree is the height of the root.

Lemma 2.1. *Let T_k^d be a fixed-degree rooted k-ary tree. The number of vertices at depth d is k^d, and are the vertices $\frac{k^d-1}{k-1}, \frac{k^d-1}{k-1}+1, \ldots, \frac{k^d-1}{k-1}+k^d-1$.*

Proof. The proof is by induction on d. The base case is clear since the tree T_k^0 consists of the single root vertex. Assume the claim holds for $d-1$. There are k^{d-1} vertices at depth $d-1$, each of which potentially has k children, which would lead to k^d vertices at depth d. By the induction hypothesis, the lowest numbered vertex at depth $d-1$ is $\frac{k^{d-1}-1}{k-1}$, and the first vertex at depth d is therefore

$$k\frac{k^{d-1}-1}{k-1} + 1 = \frac{k^d - k + (k-1)}{k-1} = \frac{k^d-1}{k-1} \quad .$$

Again by the induction hypothesis, the highest numbered vertex at depth $d-1$ is $\frac{k^{d-1}-1}{k-1} + k^{d-1} - 1$, and the last vertex at level d is therefore

$$k\left(\frac{k^{d-1}-1}{k-1} + k^{d-1} - 1\right) + k = \frac{k^d-k}{k-1} + k^d - k + k = \frac{k^d-1}{k-1} + k^d - 1 \quad .$$

Since both first and last vertex at depth d are less than $\frac{k^{d+1}-1}{k-1}$ (in particular, $\frac{k^d-1}{k-1} + k^d = \frac{k^{d+1}-k^d+k^d-1}{k-1} = \frac{k^{d+1}-1}{k-1}$) as required in the definition of T_k^d, the claim follows. □

As Lemma 2.1 shows, all leaves of a tree T_k^d have depth d (distance d from the root processor), independently of k, so the height of T_k^d is likewise d. In the undirected version of T_k^d, the maximum distance between any two vertices is thus $\mathrm{Diam}(T_k^d) = 2d$ (which is attained by two leaves). A tree (not necessarily fixed-degree) with this property is said to be *balanced*. It can also be seen that all processors in a T_k^d communication structure have either k or no children. A fixed-degree tree with this property is said to be *complete*. Trees that are not complete are called *incomplete*, and trees that are not balanced, un- or *imbalanced*. Fixed-degree, k-ary height d communication structures are thus balanced, complete trees; sometimes called *full* trees [CLRS22]. All interior vertices $u \in V(T_k^d)$ including the root have a fixed outdegree $\deg(u)$ (independent of the order of the tree) with $\deg(u) = k$; the leaves all have $\deg(u) = 0$. A particularly common rooted k-ary tree is the *binary tree* with $k = 2$. For the undirected completion of a k-ary rooted tree, the root has degree k, interior vertices degree $k+1$, and leaves degree 1.

A incomplete, unbalanced, rooted 3-ary tree where one leaf and its incident edge has been removed is shown in Figure 2.4, see Definition 2.6.

A number of alternative definitions, properties and refinements of fixed or bounded degree k-ary trees will be introduced and used in later chapters.

Observation 2.1. *A linear processor array is isomorphic to a rooted,* 1*-ary tree. A rooted star is isomorphic to a rooted,* $(p-1)$*-ary tree.*

Definition 2.5 (Height d k-nomial Tree)**.** *A* rooted, height d k-nomial tree *communication structure for* $k > 1$ *directed from the* root *is any graph that is isomorphic to* $B_k^d = (V, E)$ *with* $p = k^d$ *processors* $V = \{0, \ldots, p-1\}$ *and*

$$E \;=\; \{i \to (i + jk^e) \mid j = 1, 2, \ldots, k-1 \wedge k^e > i \wedge e = 0, 1, \ldots, i + jk^e < p\}$$

A rooted, k-nomial tree *directed towards the* root *is any directed graph that is isomorphic to the transpose of* B_k^d*.*

By definition, the order of a k-nomial tree B_k^d is k^d, and the size $k^d - 1$. Processor 0 in the canonical numbering has no incoming edge and is the root of the k-nomial tree. Interior and leaf vertices are defined as for rooted, k-ary trees. In general, the *height* of a tree communication structure (and not only a B_k^d) is the largest distance from the root to a leaf vertex of the tree, and the height of an interior vertex the largest distance to a leaf..

Lemma 2.2. *Let* B_k^d *be a rooted,* k*-nomial tree. The height of* B_k^d *is* d*. The number of leaves is* $(k-1)k^{d-1}$*, and the number of interior vertices is* k^{d-1}*.*

Proof. The proof is by induction on d. For $d = 0$, $k^0 = 1$ and B_k^0 consists of the single root processor. Assume the claim holds for $d-1$. For each of the tree vertices i in B_k^{d-1}, $k-1$ new leaves are added by choosing the largest e such that $k^d > k^e > i$ in the definition of the edge set. By this, all vertices in B_k^{d-1} remain or become interior vertices, of which there are then k^{d-1}, and $(k-1)k^{d-1}$ new leaves are added. The total number of vertices in B_k^d is $k^{d-1} + (k-1)k^{d-1} = k^d$. Since all vertices beget $k-1$ new children, the length of the longest path from the root likewise increases by one, and the height of B_k^d therefore becomes d. □

With k^d vertices in total, each $i \in V(B_k^d)$ can be viewed as a d-digit number in base k of the form $\sum_{e=0}^{d-1} j_e k^e$ with $0 \leq j_e < k$, and the edges are of the form

$$\begin{gathered}
j_{e-1}j_{e-2}\cdots j_0 \to j j_{e-1}j_{e-2}\cdots j_0 \\
j_{e-1}j_{e-2}\cdots j_0 \to j 0 j_{e-1}j_{e-2}\cdots j_0 \\
j_{e-1}j_{e-2}\cdots j_0 \to j 00 j_{e-1}j_{e-2}\cdots j_0 \\
\cdots \\
j_{e-1}j_{e-2}\cdots j_0 \to j 0\ldots 0 j_{e-1}j_{e-2}\cdots j_0
\end{gathered}$$

where $j_{e-1} \neq 0$ is the most significant digit of i (except for the root $i = 0$ where we take $e = 0$), $j_0, j_1, \ldots, j_{e-2} \in \{0, \ldots k-1\}$ and $j = 1, \ldots, k-1$ a new, more significant digit, and the number of 0-digits between j and j_{e-1} is at most $d - e - 1$. Because all k^d k-ary, d digit numbers are vertices in B_k^d, rooted, k-nomial trees as defined here are called *complete*. The depth of vertex i is exactly the number of non-zero k-ary digits in i. By this observation, two other important properties of k-nomial trees follow.

Lemma 2.3. *The number of children and thus the (out)degree of a vertex* $i = j_{e-1}j_{e-2}\dots j_0$ *is* $\deg(i) = (k-1)^{(d-e)}$. *The total number of vertices at depth* e *in a* k*-nomial tree* B_k^d *is* $(k-1)^e\binom{d}{e}$.

In particular, the vertices of k-nomial trees do neither have fixed, nor bounded degree. The maximum degree is attained by the root 0 and is $\deg(0) = (k-1)^d$. Summing over the number the vertices at all depths $e, 0 \leq e \leq d$ gives $\sum_{e=0}^{d}\binom{d}{e}(k-1)^e = \sum_{e=0}^{d}\binom{d}{e}(k-1)^e 1^{(d-e)} = ((k-1)+1)^d = k^d$ by the binomial theorem, as required by Definition 2.5.

The particular, canonical ordering in Definition 2.5 makes it easy for a processor i to find its parent and leaves in B_k^d. The parent of processor i is found by removing the most significant k-ary digit of i, and the leaves can be enumerated by the observation above.

A particularly common k-nomial tree is the *binomial tree* with $k = 2$. The name is motivated by the observation in Lemma 2.3 where $\binom{d}{e} = \frac{d!}{e!(d-e)!}$ is the *binomial coefficient* (which is the number of ways to choose a subset of e elements from a larger set of d elements). A complete, rooted binomial tree is shown in Figure 2.5. The construction of the edges for a 4-nomial tree is illustrated in Table 2.1.

A number of alternative definitions, numberings, properties and refinements of k-nomial trees will be introduced and used later.

Both k-ary and k-nomial trees have a prescribed number of vertices that make them complete in their respective senses. Trees are often used as communication structures for other processor counts p, and are for such purposes adopted in various ways. Such adopted trees can be derived from the defined trees by repeatedly eliminating leaves together with their incident edges.

Definition 2.6 (Pruned Tree)**.** *A* pruned tree *communication structure* T *is any directed graph that can be obtained from a (pruned) tree by removal of one or more leaves together with each incident edge.*

Pruned trees remain connected, that is for any two vertices u and v in the bidirected completion of such a T, either $u \leftrightarrow \dots \leftrightarrow v$. No tree vertex will become isolated by pruning. A pruned, T_3^2 tree where one leaf and its incident edge has been removed is shown in Figure 2.4.

Remark 2.1. *Given a suitably represented graph* $G = (V, E)$ *(by a collection of adjacency lists, by an adjacency matrix, or similar). Then it is an easy (polynomial time solvable) problem to detect whether* G *is a linear array, a ring, a* k*-ary or* k*-nomial tree (for given parameter* k*). This can for instance be detected from a* breadth first search *(BFS) numbering from a vertex* $u \in G(V)$ *that may be a root vertex.*

Observation 2.2. *In a rooted,* k*-ary tree in the canonical representation, the processor ranks correspond to a breadth first search numbering starting from rank* 0 *for the root processor.*

Definition 2.7 (d-dimensional Torus)**.** *A* d-dimensional torus *communication structure with dimension orders* $D[j], 0 < D[j], 0 \leq j < d$ *is any directed graph that is isomorphic to* $M_d^D = (V, E)$ *with*

$$V = \{(c_0, \dots, c_j, \dots, c_{d-1}) \mid 0 \leq c_j < D[j], 0 \leq j < d\},$$

Table 2.1: Edges of a complete, depth 3 4-nomial tree B_4^3 in canonical order.

Processor:	Children								
0:	1	2	3	4	8	12	16	32	48
1:				5	9	13	17	33	49
2:				6	10	14	18	34	50
3:				7	11	15	19	35	51
4:							20	36	52
5:							21	37	53
6:							22	38	54
7:							23	39	55
8:							24	40	56
9:							25	41	57
10:							26	42	58
11:							27	43	59
12:							28	44	60
13:							29	45	61
14:							30	46	62
15:							31	47	63
16:									
17:									
⋮:									
62:									
63:									

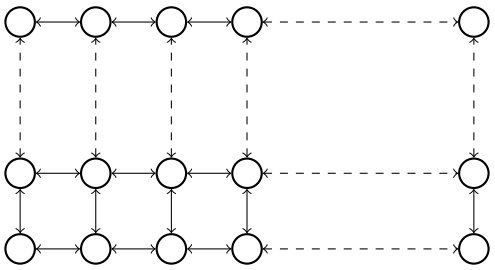


Figure 2.6: A 2-dimensional mesh M_2^D for some number of processors p and dimension orders $D[0]D[1] = p$.

Algorithm 1 One-to-one mapping between vector and processor rank in d-dimensional tori and meshes, M_d^D. The vector $(c_0, \ldots, c_j, \ldots, c_{d-1})$ is stored in the array C with $C[j] = c_j$, $0 \leq j < d$.

```
function RANK-TO-VECTOR(i, d, D)
    n ← ∏_{j=0}^{d-1} D[j]
    for j = d − 1, d − 2, ..., 0 do
        n ← n/D[j]
        C[j] ← i/n
        i ← i mod n
    end for
    return C
end function

function VECTOR-TO-RANK(C, d, D)
    i ← 0
    n ← 1
    for j = 0, 1, ..., d − 1 do
        i ← i + C[j] · n
        n ← n · D[j]
    end for
    return i
end function
```

and

$$E = \bigcup_{j=0}^{d-1} \{(c_0, \ldots, c_j, \ldots, c_{d-1}) \leftrightarrow (c_0, \ldots, (c_j + 1) \bmod D[j], \ldots, c_{d-1})\}$$

The edges of the form $(c_0, \ldots, D[j] - 1, \ldots, c_{d-1}) \leftrightarrow (c_0, \ldots, 0, \ldots, c_{d-1})$ *are called* wrap-around edges.

The canonical numbering is given by the row order *mapping from vectors* $(c_0, \ldots, c_j, \ldots, c_{d-1}) \in V$ *to processor ranks* $i \in \{0, \ldots, p-1\}$ *by* $i = \sum_{k=0}^{d-1} c_k \prod_{j=0}^{k-1} D[j]$ *where the empty product* $\prod_{j=0}^{-1} D[j]$ *is per definition* 1*, and* $\prod_{j=0}^{0} D[j] = D[j]$.

Since the processors in a d-dimensional torus can be viewed as d digit, mixed-radix numbers, the one-to-one mapping between processor ranks i in the canonical numbering and the corresponding coordinates $(c_0, \ldots, c_j, \ldots, c_{d-1})$ can be computed in $O(d)$ steps as shown by the two functions implemented in Algorithm 1.

The torus is a bidirected communication structure. The vertex set of a d-dimensional torus is the *Cartesian product* of the d sets $\{c_j \mid 0 \leq c_j < D[j]\}$ for $j = 0, \ldots, d-1$. The number of vertices in a d-dimensional torus is $p = \prod_{j=0}^{d-1} D[j]$, in other words, the order of the torus graph M_d^D is the product of the dimension orders. For each dimension j, the processors form a bidirected ring $R_{D[j]}$.

A shortest path between any two processors i and i' identified by their coordinates $(c_0, \ldots, c_j, \ldots, c_{d-1})$ and $(c'_0, \ldots, c'_j, \ldots, c'_{d-1})$ follow the ring in each dimension j for which $c_j \neq c'_j$. The distance between i and i' is therefore $\text{dist}(i, i') = \sum_{j=0}^{d-1} \min(|c_j - c'_j|, D[j] -$

$|c_j - c'_j|$). The diameter is achieved between two processors that are farthest away from each other in each dimension, and is therefore $\mathrm{Diam}(M_d^D) = \sum_{j=0}^{d-1} \lfloor D[j]/2 \rfloor$.

The torus is a regular graph. If for all dimension orders $D[j], j = 0, \ldots d-1, D[j] > 2$, the degree of a d-dimensional torus communication structure G is $\Delta(G) = 2d$. If for some dimension j, $D[j] = 2$, the two edges in dimension j coincide, and the degree decreases by one. If $D[j] = 1$, there is no edge (no selfloop) in this dimension, and the degree decreases by two.

If the dimension order $D[j] = k$ is the same for all d coordinates $j, 0 \leq j < d$, the torus is called *regular*. The vertices $i, 0 \leq i < k^d$ of a regular torus can be written as d-digit, base k (k-ary) numbers. The two neighbors for digit (dimension) j are found by incrementing and decrementing digit j modulo k, leaving the other digits unchanged.

Definition 2.8 (d-dimensional Mesh)**.** *A d-dimensional mesh communication structure N_d^D with dimension orders $D[j], 0 \leq j < d$ is a d-dimensional torus M_d^D with the same dimension orders with all wrap-around edges removed.*

The number of wrap-around edges in a torus (with each $D[j] > 2$) is

$$2 \sum_{i=0}^{d-1} \prod_{0 \leq j < d, j \neq i} D[j] \quad .$$

Therefore, the size of an N_d^D mesh is $2d \prod_{j=0}^{d-1} D[j] - \sum_{i=0}^{d-1} \prod_{0 \leq j < d, j \neq i} D[j]$.

A torus is isomorphic to a graph that is the Cartesian graph product of d bidirected $D[j]$-processor rings $R_{D[j]}$,

$$M_d^D \quad = \quad R_{D[0]} \Box R_{D[1]} \Box \ldots R_{D[d-1]}$$

Recall that the Cartesian (or box) product of two graphs G_0 and G_1 is the graph $G_0 \Box G_1 = (V, E)$ with $V = V(G_0) \times V(G_1)$ and

$$E \quad = \quad \{(u_0, u_1) \to (v_0, v_1) \mid ((u_0, v_0) \in E(G_0) \wedge u_1 = v_1) \vee (u_0 = v_0 \wedge (u_1, v_1) \in E(G_1))\} \quad .$$

The $\Box$ product is associative, so products of d graphs $G_j, j = 0, \ldots d-1$ can readily be formed [Har69, HIK11].

A mesh is isomorphic to the Cartesian product of d $D[j]$-dimensional bidirected linear arrays $L_{D[j]}$. An *incomplete torus* is the Cartesian product of a sequence of d rings or linear arrays. The incomplete torus is said to be *non-periodic* in the dimensions j for which the corresponding graph in the product is a linear array. A torus is per definition *periodic* in all d dimensions. The Message-Passing Interface (MPI) has unusual functionality to organize processes into such (in)complete tori [MPI15, MPI21, Chapter 7]. A 2-dimensional mesh is illustrated in Figure 2.6

Linear arrays and rings can, under certain constraints, be embedded into meshes and tori. This means that communication algorithms for linear arrays and tori can then be executed on meshes and tori, bar a possible renumbering or remapping of the processors.

Lemma 2.4. *A p-processor linear array L_p can be mapped into any d-dimensional mesh (or torus) with at least p processors.*

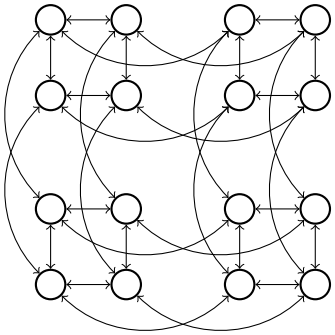


Figure 2.7: An unnumbered 4-dimensional hypercube H_4.

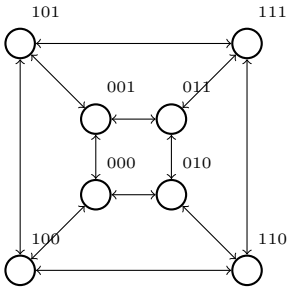


Figure 2.8: A planar embedding of a canonically numbered 3-dimensional hypercube H_3.

Proof. Constructively follow the processors in L_p in the canonical order. Processor 0 is mapped to mesh processor $(0, 0, \ldots, 0)$ (d-digit, mixed radix 0). The next processor 1 is mapped to mesh processor $(0, 0, \ldots, 1)$, and so on, until the last processor $(0, 0, \ldots, D-1)$ along dimension 0. Processor $D+1$ is then mapped to mesh processor $(0, 0, \ldots 1, D-1)$ which is connected by a dimension 0 edge. Then move backwards until mesh processor $(0, 0, \ldots, 1, 0)$ is reached, and map the next processor $2D+1$ to mesh processor $(0, 0, \ldots, 2, 0)$, etc.. That is, the linear array is embedded in the mesh by counting through the mixed radix d-ary number corresponding to the mesh processors, switching direction at each digit overflow, until all p processors have been mapped. □

This construction will not work for processor rings R_p in general, since there may be no mesh (or torus) edge between the first and the last mesh processor in the mapping.

Definition 2.9 (d-dimensional Hypercube)**.** *A d-dimensional hypercube communication structure $H_d = (V, E)$ for $d > 0$ is a d-dimensional regular mesh with $D[j] = 2$ for all dimensions $0 \leq j < d$. A 0-dimensional hypercube communication structure H_0 is a graph consisting of a single vertex with no edges.*

With dimension order $D[j] = 2$ for all d coordinates, the processors of a hypercube can be written as d-digit binary numbers $i_2 = b_{d-1}b_{d-2}\ldots b_1 b_0$ for bits $b_j \in \{0, 1\}, 0 \leq j < d$. Bit b_0 is the least significant bit, that is $i = \sum_{j=0}^{d-1} b_j 2^j$. With this representation, the d adjacent processors of i are found by flipping exactly one of the d bits. Therefore, processor $i_2 = b_{d-1}\ldots b_j \ldots b_0$ is a *neighbor* to the d processors $b_{d-1}\ldots \overline{b_j} \ldots b_0$ where bit j

has been flipped, $0 \leq j < d$ ($\overline{0} = 1, \overline{1} = 0$). The processors all have exactly one neighbor in each dimension and therefore only one outgoing and one incoming edge for each dimension (the special case for $D[j] = 2$ where mesh and torus coincide).

Lemma 2.5. *The order and size of a d-dimensional, undirected hypercube H_d is $p = 2^d$ and $d2^d/2 = d2^{d-1}$, respectively. The dimension of a hypercube with p processors is $\log_2 p$. The degree and the diameter is $\Delta(H) = d$ and $\mathrm{Diam}(H) = d$, respectively.*

Proof. Since the vertex set of the d-dimensional hypercube is the set of all d-digit numbers, the number of vertices is 2^d. Each vertex is adjacent to a different vertex by flipping exactly one bit (out- and incoming edge to and from the same vertex), therefore the degree is d, and the number of directed edges $d2^d$. A simple path from vertex i to j is a sequence of vertices whose binary representation differ in exactly one bit. The length of a shortest path between i and j is therefore the smallest number of bits to be flipped in order to change i into j (or the other way round). The largest possible number of bits to flip (for instance between i and its bitwise complement) is d, which is therefore the diameter of H. □

Observation 2.3. *If p, for instance the number of processors to be used in a communication structure, is viewed as a binary number, then it holds that p is a power of two, that is $p = 2^d$ for some $d \geq 0$, if and only if*

$$p_2 \wedge (p-1)_2 \quad = \quad 0_2$$

where $\wedge$ denotes the bitwise conjunction (and) of its two operands, and the subscript 2 emphasizes that the number is represented in binary [Knu11, War13]. If indeed $p = 2^d$ (for $d > 0$), only the dth bit is set, and $p - 1$ is a binary number $01 \ldots 1$ of $d - 1$ one bits. The bitwise conjunction of p and $p - 1$ is $00 \ldots 0$. If p is not a power of two, there is a least significant one bit at position $0 \leq e < d$. Subtracting one from p will "borrow" this bit (giving $e - 1$ one bits), but leave the bits after e unchanged, including at least bit d. Therefore $p \wedge (p - 1)$ will be non-zero.

From Lemma 2.5, it follows that the length of a shortest path between any two processors i and j in a hypercube H_d is $\texttt{popcount}(u \bigtriangledown v)$, where $\bigtriangledown$ denotes the bitwise exclusive or of its two operands ($0 \bigtriangledown 0 = 0, 0 \bigtriangledown 1 = 1, 1 \bigtriangledown 0 = 1, 1 \bigtriangledown 1 = 0$), and `popcount` (for "population count") is the number of one bits in the binary representation of its operand. By this, the number of different shortest paths between u and v is $\texttt{popcount}(u \bigtriangledown v)!$ (where ! denotes the factorial).

Definition 2.10 (Extended d-dimensional Hypercube)**.** *An* extended d-dimensional hypercube *communication structure is any directed graph isomorphic to an H_d with $O(2^d)$ additional edges between vertices u and v that are not adjacent in H_d.*

A $d = 4$-dimensional hypercube H_4 is illustrated in Figure 2.7. A $d = 3$-dimensional hypercube H_3 can be drawn in the plane (and is thus a planar graph) as shown in Figure 2.8.

Definition 2.11 (d dimensional, k-ary Butterfly)**.** *A d-dimensional, k-ary* butterfly *communication structure is any directed graph isomorphic to $Q_d^k = (V, E)$ with V being the set of d-digit, k-ary numbers $V = \{a_{d-1} \ldots a_j \ldots a_0 \mid a_j \in \{0, \ldots, k-1\}, 0 \leq j < d\}$, and*

$$E \quad = \quad \{a_{d-1} \ldots a_j \ldots a_0 \to a_{d-1} \ldots a'_j \ldots a_0 \mid a_j \neq a'_j\}$$

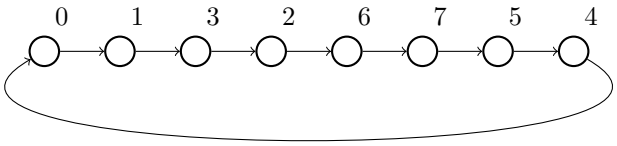


Figure 2.9: The ring R_8 in Gray code numbering.

The k-ary butterfly is a bidirected graph, and each of the k^d processors is connected to $k-1$ other processors for each of the d digits. The k-ary butterfly is a $d(k-1)$-regular graph with out- and indegrees both $d(k-1)$. The hypercube is a binary (2-ary) butterfly. The order of the (bi)directed, k-ary butterfly is k^d and the size is $d(k-1)k^d$. For undirected butterflies, the size is $d(k-1)k^d/2$.

An interesting observation is that the k-ary butterfly is isomorphic to the Cartesian graph product of d *fully connected graphs* K_k of k vertices (see Definition 2.14),

$$\begin{aligned} Q_d^k &= \underbrace{K_k \square K_k \square \ldots K_k}_{d \text{ copies}} \\ &= K_k \square Q_{d-1}^k \end{aligned}$$

More general flies can be defined by allowing the complete graphs K_k to have different number of vertices. For instance, the k's could be the prime factors of the number of processors p. This observation gives rise to interesting algorithms for certain operations, for cases where the number available of processors is not a power of k.

A number of alternative definitions, properties and refinements of hypercubes and butterflies will be introduced and used later.

Proposition 2.1. *Any linear array of p processors L_p can be embedded into a d-dimensional hypercube H_d for $p \leq 2^d$. A ring R_p of $p = 2^d$ processors can be embedded into a d-dimensional hypercube H_d.*

For the embeddings, a mapping is constructed that maps the ith processor in the linear array or ring onto a processor on the hypercube. This mapping must fulfill that flipping one bit of the hypercube processor rank will give the hypercube processor rank of either $i+1$ (the successor of i) or $i-1$ (the predecessor of i).

Such a mapping, in particular for the $p = 2^d$ processor ring R_p, is called a d-digit (binary) *Gray code* (after Frank Gray, Bell Telephone Laboratories, ca. 1934) or *reflected binary code* [Knu11, Lei92, War13]. A Gray code Γ_d for a d-dimensional hypercube lists the hypercube processors in the ring or linear array order, such that each hypercube processor rank in the list Γ_d differs by exactly one bit from the predecessor and successor in Γ_d. We let $\texttt{gray}(i) = \Gamma_d(i)$ denote the hypercube processor rank corresponding to ring or array processor i, and $\texttt{yarg}(h)$ the inverse, such that $\texttt{yarg}(\texttt{gray}(i)) = i$. Since we can start with any of the $p = 2^d$ processors on the ring, and since the order of bits of the processors ranks can be permuted in $d!$ ways, there are at least $d!2^d$ different such Gray codes. Figure 2.9 shows a ring R_8 with $p = 2^3 = 8$ processors in Gray code order.

The following recursive definition obviously defines a Gray code:

- $\Gamma_0 = \epsilon$
- $\Gamma_{d+1} = 0\Gamma_n \mid 1\Gamma_d^R$.

where $\mid$ denotes the concatenation of the two lists Γ_d, Γ_d^R, and Γ_d^R is the reverse of the list Γ_d, and the prefixes 0 and 1 are taken as new, most significant (position $d-1$) bits of all processor ranks in Γ_d and Γ_d^R, respectively. If Γ_d is indeed a Gray code for a d-dimensional hypercube, so are both $0\Gamma_d$ and likewise $1\Gamma_d^R$ Gray codes for two subcubes of H_{d+1}, and the last hypercube rank in $0\Gamma_d$ indeed differs by one bit, namely the most significant from the first processor in $1\Gamma_d^R$.

From the recursive definition, we observe that

$$\Gamma_{d+1}(i) \quad = \quad \begin{cases} \Gamma_d(i) & \text{for } 0 \leq i < 2^d \\ 2^d + \Gamma_d(2^d - 1 - i) & \text{for } 2^d \leq i < 2^{d+1} \end{cases}$$

since $\Gamma_d^R(i) = \Gamma_d(2^d - 1 - i)$. We note that $2^d - 1 - i$ is the *ones-complement* of i, if i is thought of as a binary number, which can be computed as the bitwise complement of i, $2^d - 1 - i = \overline{i_2}$.

We can use these observations to compute the Gray code embedding in constant time.

Lemma 2.6. *Let $i_2 = a_{d-1}a_{d-2}\ldots a_0$ be the d-bit binary representation of i, $a_k \in \{0,1\}, 0 \leq k < d$. The kth bit b_k of the binary representation of gray(i) is $b_{d-1} = a_{d-1}$, and $b_k = a_{k+1} \bigtriangledown a_k$ for $0 \leq k < d-1$.*

Proof. The proof is by induction on the dimension d of the hypercube. For $d = 1$, for $a_0 = 0$, $\texttt{gray}(0) = 0 = a_0$, and for $a_0 = 1$, $\texttt{gray}(1) = 1 = a_0$. Now consider a $d+1$-digit rank $i = a_d a_{d-1}\ldots a_0$. From the observation, it is clear that $b_d = a_d$ for the most significant dth bit of i. Assume the claim holds for a d-dimension Gray code. For $i < 2^d$, $a_d = 0$, and $b_{d-1} = a_{d-1}$ (and $b_k = a_{k+1} \bigtriangledown a_k$) by the induction hypotheses, therefore $b_{d-1} = a_d \bigtriangledown a_{d-1}$ as required. For $i \geq 2^d$, $a_d = 1$, and by the induction hypothesis $b_{d-1} = \overline{a}_{d-1}$ and $b_k = \overline{a}_{k+1} \bigtriangledown \overline{a}_k = a_{k+1} \bigtriangledown a_k$ by property of the exclusive or operation $\bigtriangledown$. Since $b_{d-1} = \overline{a}_{d-1} = 1_2 \bigtriangledown a_{d-1} = a_d \bigtriangledown a_{d-1}$ the claim follows. □

The discussion and Lemma 2.6 together proves Proposition 2.1.

Since the bits b_k of $\texttt{gray}(i)$ are determined independently of each other, all bits of $\texttt{gray}(i)$ can be computed in parallel by a bitwise exclusive or operation, and we have

$$\texttt{gray}(i) \quad = \quad i \bigtriangledown (i \searrow 1)$$

where $i \searrow 1$ denotes a rightmost (non-cyclic), or down shift of i by 1 position.

For the inverse operation, $\texttt{yarg}(h)$, for mapping a hypercube processor rank $h = b_{d-1}\ldots b_0$ into its position in the Gray code list, we see from Lemma 2.6 that $a_k = a_{k+1} \bigtriangledown b_k$ (since in general $a \bigtriangledown a \bigtriangledown b = b$), and therefore that:

$$\begin{aligned} a_{d-1} &= b_{d-1} \\ a_{d-2} &= b_{d-1} \bigtriangledown b_{d-2} \\ a_{d-3} &= b_{d-1} \bigtriangledown b_{d-2} \bigtriangledown b_{d-3} \\ &\cdots \\ a_k &= \bigtriangledown_{d-1}^{k} b_k \end{aligned}$$

Algorithm 2 Inverse Gray code, $i = \texttt{yarg}(h)$ for hypercube processor rank h. The loop requires $\lfloor \log_2 h \rfloor$ iterations (for $h > 0$ and no iterations for $h = 0$).

```
function yarg(h)
    i ← h
    h ← h ↘ 1                          ▷ Shift right (downwards) by one position
    while h ≠ 0 do
        i ← i ▽ h
        h ← h ↘ 1                      ▷ Shift right (downwards) by one position
    end while
    return i
end function
```

Algorithm 3 Faster inverse Gray code, $i = \texttt{yarg}(h)$ for hypercube processor rank h when h is represented as a 32-bit word.

```
function yarg(h)
    i ← h
    i ← i ▽ (i ↘ 16)                   ▷ Shift down by 16 positions (half-word)
    i ← i ▽ (i ↘ 8)
    i ← i ▽ (i ↘ 4)
    i ← i ▽ (i ↘ 2)
    i ← i ▽ (i ↘ 1)
    return i
end function
```

As seen, the kth bit of $i = \texttt{yarg}(h)$ can be computed as the kth *suffix sum* of the bits of h. The computation is shown in Algorithm 2 and takes $O(\log h)$ time steps. We notice that both $\texttt{gray}(i)$ and $\texttt{yarg}(h)$ can be computed from i and h alone without knowing the dimension d of the hypercube.

There are well-known, parallel algorithms for computing all prefix and suffix sums in parallel of d element inputs under any associative operator [JáJ92], for instance by a recursive doubling scheme. We can employ such an algorithm to compute the inverse Gray code faster as shown in Algorithm 3. When the dimension d of the hypercube is known and fixed, this algorithm takes a constant number of operations, and $O(\log d) = O(\log \log h)$ steps for hypercubes of dimension d.

Remark 2.2. *Similar to the word- or bit-parallel algorithm for computing the inverse Gray code shown as Algorithm 2, there are a great many (folklore) algorithms and tricks for doing useful computations fast on word of bits by exploiting the inherent parallelism in constant time computer arithmetic and bitwise logical operations working on multi-bit words [Knu11, War13]. Algorithms 4 and 5 show how to compute the "population count"* `popcount` *of a word w (here of 32 bits), find the integer base 2 logarithm, and the nearest power-of-two floor and ceiling of w. For a fixed, constant number of bits, the functions take constant time, as a function of the number of bits, they are all obviously $O(\log_2 w)$, which is $O(\log \log n)$ in the magnitude of the non-negative integer $n, 0 \leq n < 2^w$ that can be represented by w.*

It is instructive to ponder how these algorithms work. The $\texttt{popcount}(w)$ *algorithm*

Algorithm 4 Useful, word-parallel algorithms for computing $\texttt{popcount}(w)$, smallest power of 2 larger than or equal to w, and largest power of 2 smaller than or equal to w for 32-bit words w.

function $\texttt{popcount}(w)$
  $w \leftarrow (w \wedge \texttt{0x55555555}) + ((w \searrow 1) \wedge \texttt{0x55555555})$
  $w \leftarrow (w \wedge \texttt{0x33333333}) + ((w \searrow 2) \wedge \texttt{0x33333333})$
  $w \leftarrow (w \wedge \texttt{0x0F0F0F0F}) + ((w \searrow 4) \wedge \texttt{0x0F0F0F0F})$
  $w \leftarrow (w \wedge \texttt{0x00FF00FF}) + ((w \searrow 8) \wedge \texttt{0x0F0F0F0F})$
  $w \leftarrow (w \wedge \texttt{0x0000FFFF}) + ((w \searrow 16) \wedge \texttt{0x0000FFFF})$
  return w
end function

function $\textsc{floor2}(w)$
  $w \leftarrow w \vee (w \searrow 1)$
  $w \leftarrow w \vee (w \searrow 2)$
  $w \leftarrow w \vee (w \searrow 4)$
  $w \leftarrow w \vee (w \searrow 8)$
  $w \leftarrow w \vee (w \searrow 16)$
  return $w - (w \searrow 1)$
end function

function $\textsc{ceiling2}(w)$
  $w \leftarrow w - 1$
  $w \leftarrow w \vee (w \searrow 1)$
  $w \leftarrow w \vee (w \searrow 2)$
  $w \leftarrow w \vee (w \searrow 4)$
  $w \leftarrow w \vee (w \searrow 8)$
  $w \leftarrow w \vee (w \searrow 16)$
  return $w + 1$
end function

is based on the observation that the sum of two w'-bit words can always be represented in $2w'$ bits. The algorithm divides w first into blocks of one bit by masking out every second bit and in parallel by wordwise summation sums all such adjacent blocks. This is done by a mask $0101\ldots0101_2$ which is $5\ldots5_{16}$; in C hexadecimal constants are written by prefixing with `0x`*. Next, adjacent blocks of two bits are summed, then of four bits and so on, finally giving the population count sum of all bits in w. There are variations of this idea using even fewer instructions [War13]. The floor and ceiling functions both expand the leading bit downwards by repeated shifting downwards by doubling shift distances. As can be seen, all these algorithms are straight-line code, that is, branch-free. The integer logarithm function checks for a set bit in the upper half word of w; if a bit is set, the logarithm is at least the size of the half word (here 16), and in order to find which bit, w is shifted downwards by 16. Now the idea is repeated on a word of half the number of bits (here 16), and so on, again for a logarithmic number of steps in the word length of w. The code for this function is not branch-free.*

Algorithm 5 A word-parallel algorithm for computing the integer base logarithm of 32-bit word w.

```
function ILOG2(w)
    d ← 0
    if (w ∧ 0xFFFF0000) ≠ 0x0 then d, w ← d + 16, w ↘ 16
    end if
    if (w ∧ 0x0000FF00) ≠ 0x0 then d, w ← d + 8, w ↘ 8
    end if
    if (w ∧ 0x000000F0) ≠ 0x0 then d, w ← d + 4, w ↘ 4
    end if
    if (w ∧ 0x0000000C) ≠ 0x0 then d, w ← d + 2, w ↘ 2
    end if
    if (w ∧ 0x00000002) ≠ 0x0 then d ← d + 1
    end if
    return d
end function
```

Definition 2.12 (Cube Connected Cycles)**.** *The order d* cube connected cycles *communication structure for $d > 0$ is any graph isomorphic to $Y_d = (V, E)$ with*

$$V = \{b_{d-1}b_{d-2}\dots b_j\dots b_0 a \mid b_j \in \{0,1\}, a \in \{0,\dots,d-1\}\}$$

and

$$\begin{aligned} E &= \{b_{d-1}b_{d-2}\dots b_j\dots b_0 a \to b_{d-1}b_{d-2}\dots \overline{b_j}\dots b_0 a \mid 0 \le j < d\} \\ &\cup \{b_{d-1}b_{d-2}\dots b_j\dots b_0 a \leftrightarrow b_{d-1}b_{d-2}\dots b_j\dots b_0((a+1) \bmod d) \mid a \in \{0,\dots,d-1\}\} \quad . \end{aligned}$$

The cube connected cycles replaces each vertex in a hypercube with a bidirected ring of processors, each of which with an edge to the (replaced) neighbor in the hypercube. For $d > 2$ (cube connected cycles for $d = 1$ and $d = 2$ are special, degenerate cases), the (out)degree is therefore $\deg(Y_d) = 3$, and the cube connected cycles is a 3-regular graph. The number of vertices is $d2^d$ and the number of directed edges $3d2^d$. A processor $i \in V$ is said to have *ring index* $i \bmod d$ and *cube rank* $\lfloor i/d \rfloor$. A path from processor i to processor j in the cube connected cycles Y_d can be found by determining the bits that have to be flipped between the cube ranks $\lfloor i/d \rfloor$ and $\lfloor j/d \rfloor$, and for each bit flip move along the corresponding ring to get to the processor connected with the corresponding cube neighbor. To complete, this has to be followed by a move along the ring in which processor j is located along a path of length at most $\lfloor d/2 \rfloor - 1$ edges, and indicates that the diameter of the cube connected cycles should be at most $d + (d-1) + \lfloor d/2 \rfloor = 2d - 1 + \lfloor d/2 \rfloor$. Actually, it holds that $\mathrm{Diam}(Y_3) = 6$, and for $d > 3$ that the diameter is $\mathrm{Diam}(Y_d) = 2d + \lfloor d/2 \rfloor - 2 = 2(d-1) + \lfloor d/2 \rfloor$ as shown in [MC93, FHL97]. To see this, consider two processors in the cube connected cycle i and j with cube ranks $0\dots0$ and $1\dots1$, respectively. To get from i to j at least d bits have to be flipped, and for each bit flip except the first, a move along a ring is required, for a path of length $2d - 1$. Any bit can be chosen as the starting point for the first flip. The longest distance to be traveled along the ring of j is therefore the diameter of the ring minus one, $\lfloor d/2 \rfloor - 1$ by choosing the first bit such that the distance between this index and the ring index of j is at most

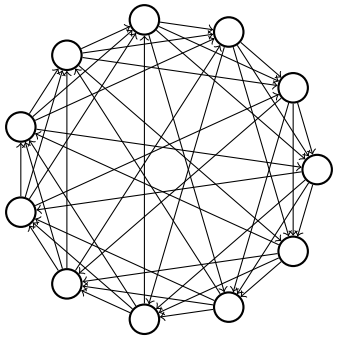


Figure 2.10: A circulant graph C_p^s with $p = 11$ and skips $s = 1, 2, 3, 6$.

$\lfloor d/2 \rfloor$, giving a total, longest distance between two appropriately chosen processors i and j of $2d + \lfloor d/2 \rfloor - 2$ (for $d > 3$). This argument gives rise to an algorithm for computing, given d, i and j alone, a path of length at most $2d + \lfloor d/2 \rfloor - 2$ edges in Y_d.

By the discussion above and Proposition 2.1, it is clear that a processor ring R_{d2^d} can be embedded into a cube connected cycles structure Y_d.

The cube connected cycles of order d can also be defined as the Cartesian product between a hypercube of order d and the bidirected completion of a ring R_d, $Y_d = H_d \Box R_d$.

Definition 2.13 (Circulant graph)**.** *A* circulant graph *communication structure with* $d, d > 0$ jumps *(or* skips*)* $s_0, s_1, \ldots s_{d-1}$ *and* $s_j \geq 0$ *is any graph that is isomorphic to* $C_p^s = (V, E)$ *with* $V = \{0, \ldots, p-1\}$ *and*

$$E \quad = \quad \{i \to (i + s_j) \bmod p \mid 0 \leq i < p, 0 \leq j < d\} \quad .$$

The circulant graph is d-regular, and the number of directed edges in is therefore dp. The diameter depends on the actual jumps. In many of the circulant graph algorithms that will be described in the following, the jumps will be the powers of two, $s_i = 1, 2, 4, 8, 16, \ldots$. A circulant graph with $p = 11$ and skips $s_j = 1, 2, 3, 6$ is shown in Figure 2.10. As can be extrapolated from the drawing, any graph that can be drawn as a regular p-gon (polygon) with the p corners representing the vertices and is rotation symmetric is a circulant graph. The name may have been derived from this property. It is an easy observation that a circulant graph with skips $s_0, \ldots s_{d-1}$ is connected if and only if $\gcd(p, s_0, \ldots, s_{d-1}) = 1$. As communication structures we mostly (only) consider connected circulant graphs.

Definition 2.14 (Fully Connected Graph)**.** *A* fully connected *communication structure is any bidirected graph that is isomorphic to* $K_p = (V, E)$ *with* $V = \{0, 1, \ldots, p-1\}$ *and*

$$E \quad = \quad \{i \to j \mid 0 \leq i < p, 0 \leq j < p, i \neq j\} \quad .$$

The diameter of a fully connected graph is $\mathrm{Diam}(K_p) = 1$, at the cost of a large size, namely $p(p-1)$ directed edges (p^2 edges if selfloops are included). The fully connected graphs is the best possible (but most expensive in terms of size) communication structure: All other communication structures over p processors can be embedded into K_p (with or without selfloops, depending).

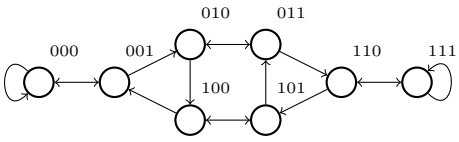


Figure 2.11: An order 3 shuffle-exchange communication structure X_3 with the canonical (binary) numbering.

Observation 2.4. *A ring is a circulant graph with jump* 1*. A fully connected graph is a circulant graph with jumps* $1, 2, \ldots, p-1$*.*

Definition 2.15 (k-partite Graph)**.** *A fully connected,* k-partite *communication structure with parts* $P_0, \ldots, P_{k-1}$ *is any bidirected graph that is isomorphic to* $K_{p_0,\ldots,p_{k-1}} = (V, E)$ *with* $V = \{0, 1, \ldots, p-1\}$ *where the parts* $P_i, P_i \subseteq V$ *partition* V*,* $p_i = |P_i|$*, and*

$$E \quad = \quad \{i \to j \mid i \in P_{i'}, j \in P_{j'}, 0 \leq i' < k, 0 \leq j' < k, i' \neq j'\} \quad .$$

The k-partite graph with $k > 1$ has $\mathrm{Diam}(K_{p_0,\ldots,p_{k-1}}) = 2$ since any path between processors in the same part has to be via some other part. The size of the graph depends on the sizes of the parts in the partition and is $\sum_{i=0}^{k-1} |P_i|(p - |P_i|)$.

An important special case of the k-partite communication structure, used for socalled *inter-communicators* in MPI [MPI15, Chapter 6], are fully connected, *bipartite graphs* where $k = 2$. A p-partite, fully connected graph is a fully connected network. A 1-partite graph is a graph without edges, and thus not a meaningful communication structure.

Definition 2.16 (Shuffle-exchange graph)**.** *A* shuffle-exchange *communication structure of order* d *is any directed graph isomorphic to* $X_d = (V, E)$ *with vertices* $V = \{0, 1, \ldots, 2^d - 1\}$ *written as* d*-digit binary numbers* $i_2 = b_{d-1}b_{d-2}\ldots b_1b_0 \in V$ *and edges*

$$\begin{aligned} E \quad &= \quad \{b_{d-1}b_{d-2}\ldots b_1b_0 \to b_{d-1}b_{d-2}\ldots b_1\overline{b_0}\} \\ &\cup \quad \{b_{d-1}b_{d-2}\ldots b_1b_0 \to b_{d-2}\ldots b_1b_0b_{d-1}\} \end{aligned}$$

The first part of the set of edges connecting vertices differing only in their least significant bit b_0 *are called* exchange edges (arcs) *and are bidirected. The edges that connect two vertices where the second vertex is found by shifting the bits of the first vertex cyclically upwards (to the next most significant position) by one position are called* shuffle edges (arcs)*.*

The order of the shuffle-exchange communication structure is as for the hypercube 2^d. In contrast, the shuffle-exchange is 2-regular only: an outgoing exchange arc, an outgoing shuffle arc, and likewise two incoming arcs. For the vertices 0 and $2^d - 1$, the shuffle arcs are selfloops. A shuffle-exchange structure for $d = 3$ is shown in Figure 2.11. The size of the communication structure is therefore 2^{d+1}. The diameter is $2d - 1$, which can be seen as follows. Choose processors $i = 0$ and $j = 2^d - 1$ whose bit patterns obviously differ in all bits. A path from i to j can be constructed by alternatingly following exchange and

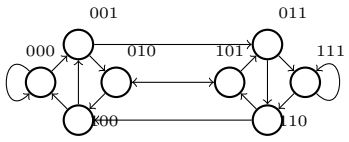


Figure 2.12: A canonically numbered, (binary) degree 2, order 3 de Bruijn communication structure J_2^3.

shuffle edges, as in

$$\begin{aligned} 00\ldots00 &\rightarrow 00\ldots01 \\ &\rightarrow 00\ldots10 \\ &\rightarrow 00\ldots11 \\ &\rightarrow \ldots \\ &\rightarrow 01\ldots10 \\ &\rightarrow 01\ldots11 \\ &\rightarrow 11\ldots10 \\ &\rightarrow 11\ldots11 \end{aligned}$$

The number of edges on this path is clearly $2d-1$, namely d exchanges (bit flips) and $d-1$ shuffles. This path is also shortest. Any other path must have immediately successive shuffle edges which means that a 0 is shifted into (the binary representation of) a position which can only be flipped by some later exchange from bit position 0. This will lead to a longer path. As can be seen, other vertex pairs have shorter distance than $2d-1$. The outlined construction can be used to find a path between any two vertices of length at most $2d-1$, showing that the shuffle-exchange graph is strongly connected; but this will generally not be a shortest path. Here is the algorithm with a few obvious improvements.

Algorithm X: Computing a path from processor i to processor i' in a shuffle-exchange communication structure X_d. Let $i'_2 = b'_{d-1}b'_{d-2}\ldots b'_0$ be the binary representation of i'.

X1. [Initialization.] Set $u \leftarrow i$. Set $e \leftarrow d-1$.

X2. Let b be the least significant (last) bit of u.

X3. [Exchange.] If $b \neq b'_e$ set $v \leftarrow u \bigtriangledown 1_2$ (where as above $\bigtriangledown$ denotes the bitwise exclusive or operation), output edge $u \rightarrow v$, and set $u \leftarrow v$ (now the last bit of u is equal to b'_e). If $u = i'$ done, else (if $b = b'_e$) set $v \leftarrow u$.

X4. [Shuffle.] Shift v cyclically upwards by one bit. If $v \neq u$, output edge $u \rightarrow v$, and set $u \leftarrow v$. Set $e \leftarrow e-1$.

X5. If $u = i'$ done, else continue with X2.

Definition 2.17 (de Bruijn graph)**.** *A* de Bruijn graph *communication structure of degree k and order d is any directed graph isomorphic to $J_k^d = (V, E)$ with vertices $V = \{0, 1, \ldots, k^d - 1\}$ written as d-digit, k-ary numbers $i_k = a_{d-1}a_{d-2}\ldots a_1a_0 \in V$ and edges*

$$E \;=\; \{a_{d-1}a_{d-2}\ldots a_1a_0 \to a_{d-2}\ldots a_1a_0a \mid a \in \{0, 1, \ldots, k-1\}\}$$

The order (number of vertices) of a de Bruijn graph J_k^d is k^d. de Bruijn graphs are k-regular, so the size is k^{d+1}. A binary (degree 2) de Bruijn structure for $d = 3$ is shown in Figure 2.12. The diameter of a de Bruijn graph is d, independently of k. This can be seen as follows. Let i and j be processors that when written as d-digit, k-ary number, differ in all positions. To get from i to j follow a path where each edge "shifts in" the next lower digit of j, starting with the most significant digit.

Definition 2.18 (Kautz graph)**.** *A* Kautz graph *communication structure of degree k and order d is any directed graph isomorphic to Z_d^k with*

$$V \;=\; \{a_{d-1}a_{d-2}\ldots a_j\ldots a_1a_0 \mid a_j \in \{0,\ldots k\}, a_j \neq a_{j-1}, j = d-1, d-2, \ldots, 1\}$$

written as d-digit, $(k+1)$-ary numbers subject to the condition that successive digits differ, and

$$E \;=\; \{a_{d-1}a_{d-2}\ldots a_j\ldots a_1a_0 \to a_{d-2}\ldots a_j\ldots a_1a_0a \mid a \in \{0,\ldots,k\}, a \neq a_0\} \quad .$$

Since there are k possibilities for a in $a_{d-1}a_{d-2}\ldots a_j\ldots a_1a_0 \to a_{d-2}\ldots a_j\ldots a_1a_0a$, the degree of each vertex is k so the Kautz graph is regular. For a vertex in the graph, there are $k+1$ possibilities for the first digit and k possibilities for each successive digits, so the order of the Kautz graph (number of vertices) is $(k+1)k^{d-1} = k^d + k^{d-1}$. Since the graph is regular, the size of the graph is $k^{d+1} + k^d$. Like for the de Bruijn graph, the diameter is d, independently of k, which can be seen by a similar path construction. Note that the construction of the Kautz graph given in the definition is not a canonical numbering, since not all d-digit, base $(k+1)$ numbers are in V.

The Kautz graph has a strong, extremal property. The socalled *degree diameter problem* asks for the largest number of vertices that a graph G with a given maximum degree $k = \Delta$ and diameter $d = D$ can have, socalled (Δ, D)-graphs [BDQ86] (also often denoted (d, k)-graphs with k and d exchanged such that d denotes the maximum degree and k the diameter; k and d is used here as in the definitions of de Bruijn and Kautz graphs). The following simple argument gives an upper bound on the number of vertices. Pick a vertex. In the case of an undirected graph, this vertex can be connected to at most k different other vertices. Each of these k vertices can be connected to at most $k-1$ other, new vertices (since there is already an edge to the parent), which can again be connected to at most $k-1$ new, other vertices, etc., up to a distance of d from the initial vertex.

Table 2.2: Order of some directed communication structures compared to the directed Moore bound.

k	d	H_d	J_k^d	Z_d^k	Moore bound
5	5	32	3125	3750	3906
10	10	1024	10^{10}	$11 \cdot 10^9$	11 111 111 111
15	15	32768	15^{15}	$16 \cdot 10^{14}$	$469 \cdot 10^{15}$

This relates the order p of a communication structure G in (k, d) to k and d as follows.

$$\begin{aligned} p &\leq 1 + k + k(k-1) + \ldots k(k-1)^{d-1} \\ &= 1 + k \sum_{i=0}^{d-1} (k-1) \\ &= 1 + k \frac{(k-1)^d - 1}{(k-1) - 1} \\ &= \frac{k - 2 + k(k-1)^d - k}{k-2} \\ &= \frac{k(k-1)^d - 2}{k-2} \end{aligned}$$

for $k > 2$ (and $p \leq 1 + 2d$ for $k = 2$). This bound is called the *Moore bound* for undirected graphs, and undirected, k-regular graphs G that achieve this bound ($p = \frac{k(k-1)^d-2}{k-2}$) are called *Moore graphs* (after E. F. Moore, ca. 1958). For $d = 1$, the Moore graphs are the fully connected graphs. For $k = 2$, the Moore graphs are rings of order $1 + 2d$. Other than that, Moore graphs can exist only for $d = 2$ and $k = 2, 3, 7, 57$ [MŠ13]. For directed graphs with maximum in- and out-degree k, the same argument gives

$$p \leq 1 + k + k^2 + \ldots + k^d = \frac{k^{d+1} - 1}{k - 1}$$

There are no directed (k-regular) Moore graphs unless $k = 1$ or $d = 1$ [BT80].

An asymptotic Moore bound bound for (un)directed graphs is $p = k^d + O(k^{d-1})$. The Kautz graph of order $p = k^d + k^{d-1}$ is among the graphs closest to this bound; the de Bruijn graph achieves only k^d vertices within the same diameter d.

The number of vertices in hypercubes of given order is compared against the order of de Bruijn and Kautz graphs with the same degree and diameter, and against the directed Moore bound in Table 2.2.

Summary: The order, size, degree and diameter properties of the communication structures introduced here are summarized in Table 2.3. The literature, as is natural, sometimes use different naming schemes and abbreviations; $Q(n)$ is sometimes used for the n-dimensional hypercube [MS90]; CCC_d for the cube connected cycles of order d [FL94], etc..

Table 2.3: Basic properties of the simple, directed communication structures.

Structure	Shorthand	Order	Size	Degree	Diameter (undirected)
Linear array	L_p	p	$p-1$	≤ 1	$p-1$
Ring	R_p	p	p	1	$\lfloor p/2 \rfloor$
k-ary Tree	T_k^d	$\frac{k^{d+1}-1}{k-1}$	$\frac{k^{d+1}-1}{k-1}-1$	k	$2d$
k-nomial Tree	B_k^d	k^d	k^d-1	$\leq (k-1)d$	$d+(d-1)$
Torus ($D[j]>2$) with $p=\prod_{i=0}^{d-1} D[i]$	M_d^D	p	$2dp$	$2d$	$\sum_{i=0}^{d-1}\lfloor D[i]/2 \rfloor$
Mesh ($D[j]>2$) with $p=\prod_{i=0}^{d-1} D[i]$ and $e=\sum_{i=0}^{d-1}\prod_{0\leq j<d, j\neq i} D[j]$	N_d^D	p	$2dp-2e$	$\leq 2d$	$\sum_{i=0}^{d-1}(D[i]-1)$
Hypercube	H_d	2^d	$d2^d$	d	d
Butterfly	Q_d^k	k^d	$d(k-1)k^d$	$(k-1)^d$	d
Cube Connected Cycles	Y_d	$d2^d$	$3d2^d$	3	$2d+\lfloor d/2 \rfloor -2$
Circulant graph	C_p^s	p	sp	s	
Fully connected	K_p	p	$p(p-1)$	$p-1$	1
k-partite	$K_{p_0,\ldots,p_{k-1}}$	p			2
Shuffle-Exchange	X_d	2^d	2^{d+1}	d	$2d-1$
de Bruijn	J_k^d	k^d	k^{d+1}	k	d
Kautz	Z_d^k	k^d+k^{d-1}	$k^{d+1}+k^d$	k	d

2.2 Message Passing Algorithms

A message passing algorithm consists in a set of sequential algorithms for each of the p processors in a distributed computing system with a given, directed communication structure $G = (V, E)$. Processors can communicate with each other by sending messages along the directed edges in the communication structure, but otherwise cannot not share any information (through shared variables, data structures, memory or other). A sequential algorithm is a sequence of instructions to be carried out by a processor, which will lead to a desired, specified solution of the given problem after a finite number of steps [Tar83,Knu73]. Processors are full-fledged, stored-program computers with access to a local memory only. A program is an incarnation of an algorithm in a specific programming language. A program running on a processor will be referred to as a *process*. As the processors, processes will likewise be identified by their *rank* $i, i \in \{0, \ldots, p-1\}$ (position in G), where p is the total number of running processes. Algorithms are described semi-formally by their sequence of steps as in Algorithm X, and more formally in pseudo-code as in Algorithm 1, and Algorithm 2 and 3. Concrete program code in C [KR88] will sometimes be used.

For the pseudo-code, more or less "standard" algorithm notation will be used with $a, b \leftarrow x, y$ to denote (multiple, simultaneous) assignment of the expression values x and y to the variables a and b, with x and y evaluated independently and with the old values of a and b, bounded **for**-loops, **forall**-loops where the order of the loop iterations do not matter (and could be done in parallel), unbounded **while**-loops, procedures and functions, etc. Data structures will be used freely with various notation to denote arrays and array slices, e.g. $A[i : j]$ for the consecutive slice of A from index i up to and including index j, and will be explained in context. Arrays are, however, always indexed from 0 (C-like convention). Array and data structure assignment ("memory copy") operations will likewise be allowed and used freely, e.g., $A \leftarrow B$ for the assignment (copy) of the elements of the array or linearly ordered data structure B into array or linearly ordered data structure A. The size of the arrays, that is, the number of elements to be copied will follow implicitly from the size of the corresponding data structures. The notation $\|$ is used to denote explicitly the possibility of parallel execution of two or more statements.

Division and modulus and cyclic indexing As in C, division of an integer a by another b yields an integer result a/b that is truncated towards zero (rounded to the largest, smaller integer for positive a/b and the smallest, larger integer for a/b negative); for clarity here most often written explicitly as $\lfloor a/b \rfloor$. The modulus operation $a \bmod b$ is defined such that $b\lfloor a/b \rfloor + a \bmod b = b$, and $a \bmod b$ is therefore integer as well. This means that if $\lfloor a/b \rfloor < 0$ (either $a < 0$ or $b < 0$ but not both), then also $a \bmod b < 0$.

We often deal with cyclic data and communication structures, e.g., rings R_p and circulant graphs C_p^s, and arrays where index $p-1$ is followed by index 0, and index 0 preceded by $p-1$.

For convenience for such uses, we define for index $i, 0 \leq i < p$ (in general, just $i \geq 0$), and skip or offset $s, -p \leq s \leq p$.

$$(i+s)_p \quad \equiv \quad (i+s+p) \bmod p$$

such that it holds that $0 \leq (i+s)_p < p$.

Synchronous message passing: Two processes can communicate by sending information as *messages* from one process the other, if there is a corresponding arc in the underlying communication structure, or between the two processes if there is a bidirected edge connecting the two. Communication operations are denoted by unidirectional *send* and *receive* operations, and are written as follows:

Process $i \in V(G)$:	
$\mathsf{Send}(A, j, G)$	Send a message consisting of data read sequentially from array A (or other linearly ordered data structure in memory) to process j in communication structure G.
$\mathsf{Recv}(B, j, G)$	Receive data from process j in communication structure G into array B.

Send and receive operations in a communication structure G can be carried out by process $i \in V(G)$, if there are edges $i \to j \in E(G)$ (for sending) and $j \to i \in E(G)$ (for receiving). We tacitly assume at least partially synchronized communication, not unlike (but also not quite like) MPI [MPI21]. By this, we mean that when process $i \in V(G)$ encounters a send operation $\mathsf{Send}(A, j, G)$, the process will be blocked and waiting until process $j \in G$ encounters a corresponding $\mathsf{Recv}(B, i, G)$ operation; likewise, process j is blocked on the $\mathsf{Recv}(B, i, G)$ operation until process i encounters its corresponding $\mathsf{Send}(A, j, G)$ operation. At this (*rendezvous*) point, the communication of the message from process i, the *source process*, to process j, the *destination process*, takes place. Message passing communication as described here *synchronizes* the two participating processes, and is therefore called *synchronous message passing*. This is the form of message passing that will be used for most of the algorithms that will follow. It follows that a process performing a send operation will be blocked forever and not able to proceed if there is no corresponding receive operation. This situation is called a *deadlock*, and is to be avoided at all costs. A (correct) message passing algorithm for some possibly arbitrary (but finite) number of processes will terminate in a finite number of steps, and in particular will not contain a deadlock for its allowed inputs; only (deadlock-free) algorithms are of interest here.

A process running on a communication structure G will be identified by its rank $i \in V(G)$ which we will assume is known in the algorithms we will present for different communication structures. The following program for process $i \in V(G)$, for instance, assuming that $0, 1 \in V(G)$ and $0 \to 1 \in E(G)$ for a given communication structure G, in particular will always deadlock when executed on at least the two processes 0 and 1:

if $i = 0$ **then**
  $\mathsf{Send}(A, 1, G)$
  $\mathsf{Recv}(B, 1, G)$
else if $i = 1$ **then**
  $\mathsf{Send}(A, 0, G)$
  $\mathsf{Recv}(B, 0, G)$
end if

Note that this is a different semantics from that of the potentially blocking send operation in MPI, `MPI_Send()`, where the corresponding program may not deadlock (depending for instance on the size of the message, and in general on the implementation

of the used MPI library); but alike the synchronous send operation, `MPI_Ssend()`. The same holds for the program where the send and receive operations of each of the two processes are interchanged (receive first, send second).

Such deadlocks can be avoided by either sequencing the operations differently in the two processes, and thus breaking the symmetry, as for instance in the following algorithm

if $i = 0$ **then**
  $\mathsf{Recv}(B, 1, G)$
  $\mathsf{Send}(A, 1, G)$
else if $i = 1$ **then**
  $\mathsf{Send}(A, 0, G)$
  $\mathsf{Recv}(B, 0, G)$
end if

or by allowing send and receive operations to take place (virtually) in parallel, which will be written

if $i = 0$ **then**
  $\mathsf{Send}(A, 1, G) \parallel \mathsf{Recv}(B, 1, G)$
else if $i = 1$ **then**
  $\mathsf{Send}(A, 0, G) \parallel \mathsf{Recv}(B, 0, G)$
end if

This notation denotes *bidirectional communication* and is allowed if there is indeed a bidirected edge $i \leftrightarrow j \in E(G)$ in the communication structure G. If G contains a selfloop, a process having such an edge can send a message to itself, and sometimes this is even useful. Bidirectional communication correspond to the MPI operation `MPI_Sendrecv()`. The operator $\parallel$ is used here to denote that two algorithmic activities take place concurrently, and can even be done in parallel, if the processor resources for doing so are available.

Data for send and receive operations are supposed to be in local arrays or other, linearly ordered data structures $A, B, \ldots$ (linearly ordered means that it makes sense to talk about the ith element, the next element, the previous element, and so on), and the data elements of a message will be sent and received successively, one after the other, in the linear order of the data structure. For instance, if A and B are arrays of n elements, the array elements will be send as $A[0], A[1], \ldots, A[n-1]$ and received likewise as $B[0], B[1], \ldots, B[n-1]$. In the algorithms, it will be clear from context or by special notation, how many data elements will be sent in a $\mathsf{Send}(A, j, G)$ operation. For instance, $A[s:e]$ indicates an array slice of array A starting from index s up to and including index e and therefore consisting of $e-s+1$ elements (of the type of the array), and it will indeed be allowed to write $\mathsf{Send}(A[s:e], j, G)$ to send this slice of elements. We will sometimes use the notation $[m]$ to indicate an anonymous buffer of m elements. In a $\mathsf{Recv}(B, j, G)$ operation, the array or data structure B must be large enough to store the data being sent by the corresponding send operation, and in our algorithms it will be known exactly how many data elements are being received. In that sense, by properties of the way our algorithms are conceived, pairs of send and receive operations will always *match*: The number and data type of the elements sent and received are exactly the same, but we will allow the data structures A and B to be different. For instance A could be a row of a local matrix of the sending process, and B a column of another matrix of the receiving process; the communication will be legal and can succeed if the number of elements in the

sent row and the received column are the same. This form of conversion between data structures (but not individual element data types) can also be realized in MPI by the use of socalled *user-defined* or *derived data types* and can sometimes be convenient (and improve performance) in concrete programs. For bidirectional communication, and in general, for activities that may take place concurrently, in parallel, it is important that the data structures (arrays) for messages to be sent and messages to be received are distinct and not in any way overlapping in memory. If not, the outcome would not be well-defined (without restrictions on the concurrency allowed), and such usage would constitute a classical *race condition*. Because of the matching conditions that are fulfilled by correct message passing algorithms, it is not necessary to send any meta-information with the data elements: data type and number of elements are known by both processes partaking in the rendezvous. MPI benefits from that, and MPI communication can therefore ideally have very low extra overhead by data in addition to the actual "message payload".

A pair of send and receive operations are said to correspond or *match* if (a) the destination process that is specified in the send operation eventually performs a receive operation with the sending process as source, (b) the number of elements specified (implicitly) in the send operation equal to (in MPI: is no larger than) the number of elements specified (implicitly) in the receive operation, and (c) both processes specify or tacitly expect the sequences of elements to have the same data types. Note that this does not mean that all elements in a message must be of the same datatype, but only that the sth element in the message sent must be of the same type as the sth element expected to be received. As mentioned MPI supports typed communication in this way via the derived datatypes [MPI15, Chapter 4].

A message passing algorithm is correct if it obeys these rules on message passing communication, and only correct algorithms are given and of interest in the following (the exception is the deadlock situations explained above). It is an interface and implementation issue, whether such rules are checked and enforced for the practical programmer. For instance, MPI implementations do, for performance reasons, usually not check such rules. It is likewise an implementation and partly a hardware system issue whether communication is actually synchronous as described above, or less strict. The point is that algorithms are described and analyzed as if communication can only take place synchronously.

Communication capabilities: For a communication structure with a bidirected edge $i \leftrightarrow j$, simultaneous communication from processor i to j and from processor j to i was allowed and implemented or supported by parallel send and received operations. For this to actually be possible and take place, the underlying hardware on which the message-passing program is running must be able to support this. This part of the communication system hardware is often abstractly called a *port*. We use the term *communication capabilities* to classify to what extent simultaneous communication can take place, and can classify either algorithms or actual processor and communication hardware according to such capabilities.

A system can have one or more communication ports, each of which can be engaged in a communication operation. With a single port, at most one communication operation can take place at a time, with k ports up to k communication operations. A system with a single port is, surprisingly, said to be *single-ported* (or *one-ported*), a system with more

than one port, *multi-ported* or k-ported, to emphasize the concrete number of ports. In a multi-ported system, it is assumed that k is independent of p (constant), the number of processing units, and tacitly that all processing units have the same number of ports. Real hardware may be more heterogeneous than that.

Communication on a port is

- *uni-directional* (or *half-duplex*), if only either a single send or a single receive operation is taking place at any one time, and
- *bidirectional* (or *full duplex*), if both a send and a receive operation is taking place at the same time. Bidirectional communication can be either
  - *telephone* like and take place between a pair of processes i and j, where both processes send and receive along a bidirected edge $i \leftrightarrow j$ in the corresponding communication structure, or
  - *fully bidirectional* (or *send-receive* bidirectional) where a process i receives from a process l and sends to a process j, both $i \rightarrow j$ and $l \rightarrow i$ are edges in the corresponding communication structure.

A port allowing only uni-directional communication is *weaker* than a port with bidirectional communication in the sense that an algorithm for the former will work without modification on the latter; but not vice versa. A uni-directional port may be less costly (in some measure of cost) than a bidirectional one, and in that sense desirable. The telephone model of bidirectional communication is a special, restricted case of the fully bidirectional model; but again, may be easier to realize in concrete hardware and circumstances. One-ported communication requires less resources (ports) than k-ported communication, but it is conceivable that problems exist that could be solved faster with more communication ports. It is therefore interesting to study and classify problems and algorithms with respect to their required communication capabilities, and to study trade-offs between resources (ports) and for instance time bounds of algorithms. It is therefore also important that system and model assumptions for concrete algorithms are always (explicitly or implicitly) clear.

As described above, in our algorithmic notation we use for processor i

$\mathsf{Send}(S, j, G) \parallel \mathsf{Recv}(R, l, G)$

to denote bidirectional, concurrent, parallel, but one-ported communication along the edges $l \rightarrow i$ and $i \rightarrow j$ in the communication structure G. It is important that the linearly ordered communication data structures S and R are entirely disjoint; concurrently sending and receiving from partly overlapping structures would lead to race conditional and non-deterministic or undefined results. The number of elements sent from S and received into R may be different. For k-ported communication, up to k such concurrent send and receive operations can take place simultaneously. Also here, no intersection between the data structures will be tolerated.

2.3 Message Passing Implementations with MPI

All message passing algorithms that are developed in the following many chapters are intended to and can be implemented concretely (in C) with the Message Passing Interface

(MPI) [MPI15, MPI21]. MPI is designed and implemented as a library with a large number of operations for facilitating communication between processes in various ways. We briefly recapitulate some of the important features of MPI.

A running MPI program consists in a set of processes that is started externally and, unless dynamic process creation is used in the program, not changing throughout the execution. The MPI processes are *mapped* to the processor resources of the parallel computer on which the program is started, and also this mapping is normally not changing. Requesting system resources, starting and mapping processes is not treated here, so the practical programmer need to consult resources and documentation for the system at hand.

A set of processes that can communicate with each other, via send and receive operations as in the algorithm and deadlock examples above, via socalled one-sided communication (which we will rarely use), or via collective communication operations, is represented in MPI as a socalled *communicator*. A communicator is an object with properties and operations that is distributed across and shared by the processes. Only processes in (having access) to the same communicator can communicate. Any process in an MPI program can be part of multiple communicators, though. A fundamental principle of MPI is that communication is always relative to a communicator (which is given as an argument to the communication operations, much as the communication structure is mentioned explicitly in the algorithms notation for sending and receiving data), and that communication within a communicator is shielded from communication in other communicators. This design principle makes it possible to implement *safe parallel libraries*. When an MPI program is started over a set of requested processors or processor-cores (externally, by a scheduling system or command) as many processes are started with an initial, global communicator called `MPI_COMM_WORLD`. For this communicator to come into existence, the MPI library must be initialized by the MPI operation `MPI_Init()`. This is shown in the MPI program in Listing 2.1. All MPI objects are cleaned up properly by the `MPI_Finalize()` call. The running processes can be identified by their *rank* in the `MPI_COMM_WORLD` communicator. The rank of a process in a communicator is always an integer $i, 0 \leq i < p$ where p is the number of processes belonging to the communicator, for instance as started initially and belonging to `MPI_COMM_WORLD`. Both the *rank* and the *size*, which is the number of processes in a communicator can be looked up by a process by calling the functions `MPI_Comm_rank(comm,...)` and `MPI_Comm_size(comm,...)`, respectively. In general, a process in some communicator `comm` has its rank in that communicator, and the same process can have different ranks in different communicators, and different communicators can likewise have different sizes.

Listing 2.1: A first MPI program: Initialization, rank and size and communication with a potential deadlock.

```
#include <stdio.h>
#include <stdlib.h>

#include <assert.h>

#include <mpi.h>

#define TAG0 1000
#define TAG1 1001
```

```
int main(int argc, char *argv[])
{
  MPI_Init(&argc,&argv);

  int rank;
  int size;

  MPI_Comm comm = MPI_COMM_WORLD;

  MPI_Comm_rank(comm,&rank);
  MPI_Comm_size(comm,&size);
  assert(size>1);

  int A[10];
  int B[12];

  int i;
  for (i=0; i<10; i++) A[i] = i+1;

  MPI_Status status;

  if (rank==0) {
    int elements;

    MPI_Send(A,10,MPI_INT,1,TAG0,comm);
    MPI_Recv(B,12,MPI_INT,1,TAG1,comm,&status);

    MPI_Get_count(&status,MPI_INT,&elements);
    printf("Rank␣%d␣received␣%d␣integer␣elements\n",rank,elements);
  } else if (rank==1) {
    MPI_Send(A,8,MPI_INT,0,TAG1,comm);
    MPI_Recv(B,10,MPI_INT,0,TAG0,comm,&status);
  }

  MPI_Finalize();

  return 0;
}
```

Based on the process ranks in `MPI_COMM_WORLD`, the program in Listing 2.1 attempts to exchange information between the processes with ranks 0 and 1. All processes have the statically allocated arrays `A` and `B` of 10 and 12 C integers, respectively. The process with rank 0 (from now on, we will just say "process i" for the process with rank i in the communicator that is used and referred to implicitly) sends 10 integer elements as indicated by the `MPI_INT` datatype argument to process 1, and the message that will consist of the 10 elements in the order $A[0], A[1], A[2], \ldots, A[9]$ is *tagged* with the integer label `TAG0`. As discussed, the type information (integer elements) is not sent as part of the message that is transferred towards process 1, but information on the number of elements and the tag are. For (high-)performance reasons, MPI is designed so as to require only little meta-information in the messages that are communicated, and a good MPI implementation conforms to this design goal. The third, datatype argument in `MPI_Send(buffer,count,datatype,...)` can describe complex layouts of data in the

memory and the second, count argument give the number of elements of this datatype to send. Process 1 on the other hand sends "only" 8 elements out of its local `A` array to process 0. Data to be sent in MPI are always specified by the local memory address to where the data are stored, an element count, and the datatype and structure of the elements. Data buffers are assumed to have been allocated properly before the `MPI_Send()` call, and at most as many elements shall be sent as are actually present in the data buffer before the call. This means, for instance, that data buffers should not be updated (or deallocated) as long as a sending communication operation using the buffer may be going on.

Both `MPI_Send(...,comm)` and `MPI_Recv(...,comm,status)` are socalled *blocking* operations in MPI terms. This means that upon return from such a call, the specified operation, either send or receive, is complete from the issuing process' point of view. For the `MPI_Send()` calls, this means that the data elements from the `A` arrays have been sent (somewhere), and that the arrays can now be used for other things by the processes. There is in general no implication on where the data are, in particular, it cannot be inferred that the receiving process has even started on its `MPI_Recv(o)` peration. This semantics give MPI implementers some freedom in how to implement `MPI_Send()`: Sometimes it is beneficial to buffer (small) data at the sending process, sometimes at the receiving process, and sometimes it is better to wait for the receiving process to start its `MPI_Recv()` operation. Therefore, the program in Listing 2.1 may indeed not deadlock, in stark contrast to the synchronizing assumptions made for message-passing algorithms as discussed above. Nevertheless, the program is potentially deadlocking, dependent on message sizes and MPI implementation, and therefore *unsafe*. Such programs should be avoided at all cost. For this, MPI programs with send and receives should be written as if the operations would be synchronous, such that each send operation by any process is matched by a receive operation by the target process.

A blocking receive operation can of course not complete (the process not return from the call) before data have been sent by the sending process. After the `MPI_Recv()` operation, data are stored in the buffer, here array `B`, which must have been allocated large enough to store the number of elements specified by the second count argument. The actual number of elements that is received, and which may be smaller than the specified count, is stored as part of the last argument which is an MPI *status object*. The allocated `status` object is queried for the number of received elements by the `MPI_Get_count()` call.

This first example program is written to work with any number of started processes. With only one process, the communication would have immediately and deterministically deadlocked, and this exceptional case is caught by an assertion (see the C assert library).

Communicators and communication structures An MPI communicator can be thought of as representing a fully connected communication structure (Definition 2.14) over p processes, since any MPI process can communicate with any other MPI process in the communicator, either by collective operations, or process-wise using the rank in the communicator by send and receive operations or one-sided communication.

MPI makes it possible to impose some structure on communicators via socalled Cartesian and distributed graph communicators. A suggestion to extend this further can be found in [THMH21]:

Process mappings in MPI MPI is a static interface in the sense that new MPI object can only be created out of old ones; existing objects cannot be modified. This holds also for communicators: `MPI_COMM_WORLD` cannot change (by removing already started or adding new processes), and a started process cannot change its rank in a communicator. To achieve any of this (adding new processes is a special situation not covered here, refer to the process creation and management facilities [MPI15, Chapter 10]), instead new communicators with the desired properties must be created out of old ones. Such is done by special, *collective* operations which have to be called by all processes in some old communicator and return to (a subset of) the calling processes communicator(s) with the desired properties.

We discuss three such collective operations for creating new communicators out of old ones.

`MPI_Comm_split(comm,...,&newcomm)` `MPI_Cart_create(comm,...,&newcomm)` `MPI_Dist_graph_create(comm,...,&newcomm)`

`MPI_Comm_dup(comm,&newcomm)` is useful for library building.

2.4 Global Specifications for Global Operations

The problems that will be considered in the following chapters are all of the sort that require participation of a (sub)set of processors, each of which will have a (possibly empty) part of the input and a part of the output. A distributed, parallel algorithm for solving such a problem will correctly compute the specified output from the given, required input. This input-output relationship between the parts of the input and the parts of the output, as well as the distribution over the participating processors will be described by specifying for each process or processor a pre-condition (requirement) that this process must satisfy before the algorithm is started, and a post-condition (ensured result) that this process will satisfy after the algorithm has terminated for the process. Since the focus will be on collective communication and reduction problems, a special notation will be used for describing pre- and post-conditions by the structure of the data that will be communicated.

A communication operation is described by specifying for each processor a local block of data to be distributed in some way to the other processors, and a local block of data that is received or computed from the blocks of the other processors, possibly including the processor itself. Data blocks and receive buffers always have an implicit or explicit *linear order* as discussed in Section 2.2, so that it makes sense to talk about the sth element of some block.

Let m be the number of elements (e.g., bytes, integers, complex numbers, ...) of some data block that is stored in the memory of some processor. We will use the notation $[m]$ to both *name* and *denote* (the actual elements of) such a linearly ordered data block with m elements to indicate that the block can have some particular structure: our operations are intended not only for simple consecutive buffers or arrays. Let $[m_0], [m_1], \ldots, [m_{p-1}]$ be data blocks with $m_0, m_1, \ldots, m_{p-1}$ elements, respectively. By $[m] = [m_0 m_1 \cdots m_{p-1}]$ we will denote a block of $m = \sum_{i=0}^{p-1} m_i$ elements formed by concatenating the elements of the named blocks $[m_0], [m_1], \ldots, [m_{p-1}]$. The block $[m]$ is again linearly ordered by the elements of the $[m_0]$ block, followed by the elements of the $[m_1]$ block, etc. up to

the elements of the $[m_{p-1}]$ block. Concatenation may not have to be done explicitly by copying data around, since the block notation does not describe exactly where the data are located in memory, but only assumes that they can be accessed in a convenient and fast way. A data block is to be thought of as something that can be sent (streamed) from one processor to another. A block can even be virtual and used for the purpose of referring abstractly to some sequence of data elements. If needed to refer to a specific element or sequence of elements in a linearly ordered block $[m]$ we can again use array notation like $[m][s:e]$ for the elements from s up to and including e, where for both s, e, it holds that $0 \leq s \leq e < m$.

Operations on data blocks by a distributed memory parallel algorithm are specified by giving pre- and post-conditions on data blocks at the processors. More formally, an operation X involving all p processors, $0, 1, \ldots, p-1$ will be described by the required distribution of the input data blocks before the operation (above the bar), followed by the ensured distribution of the output blocks after the operation (below the bar).

$$\begin{array}{rccccccc} \text{Proc}: & 0 & 1 & \ldots & i & \ldots & p-1 \\ & [m_0] & [m_1] & \ldots & [m_i] & \ldots & [m_{p-1}] \\ \mathsf{X} & \hline & & & & & & \\ & [m'_0] & [m'_1] & \ldots & [m'_i] & \ldots & [m'_{p-1}] \end{array} \tag{2.1}$$

The input blocks $[m_i]$ name and denote the data stored at the respective processors $i, 0 \leq i < p$ before the operation. For each processor i, both the size of the block m_i as well as the actual data represented by some linearly ordered data structure $[m_i]$ are given and known. The output blocks $[m'_i]$ consist of data collated or computed from the input blocks. Each data element in an output block $[m'_i]$ must come from or depend on some or more data elements from some input block(s) $[m_j]$. The sizes of the output blocks m'_i must be either known by the processors or computed from the known input sizes m_j.

An algorithm that is claimed to implement operation X must be shown to achieve the ensured output for each of the involved processors, and the specification of the operation X clearly describe which output sizes will be known and given as part of the input and which will be have to be computed by the algorithm.

2.5 Parallel, Distributed Memory Systems

The intention here is to exemplify the discussed communication structures by actual systems that play(ed) a role in parallel, high-performance computing. We discuss hybrid clusters versus purely networked systems. Different communication media. Deep(er) hierarchies. Communication processor. Communication ports. "Network Interface Card" (NIC).

Hypercube systems: Intel IPSC Hypercube, Cosmic cube [Sei85], Intel Paragon.

Fat tree systems: Connection Machine CM5. Earth Simulator 2. [Lei85].

Torus systems: IBM Blue Gene systems. K-Computer. Fugaku.

Fully connected systems: Earth Simulator [ZS04].

Kautz network systems: SciCortex.

Dragon Fly: Cray
Projective networks, Slim fly.

2.5.1 Direct and indirect networks

Multi-stage networks: InfiniBand

2.5.2 Deep(er) hierarchies

2.5.3 Routing

2.6 Communication Costs and Algorithm Performance

To analyze, judge and compare message passing algorithms, adequate performance models are needed. First, models for the cost of the (synchronous) communication operations used between processors (and processes) in the given communication structure. Second, models for the cost of entire, distributed memory parallel algorithms.

2.6.1 Communication costs

Assume that a communication structure G is given, permitting direct communication between processors u and v that are connected by an edge $(u, v) \in E(G)$. A cost model should assign cost (in time, or other resource) to both individual communication operations (performed in isolation), and to simultaneous communication between many pairs of adjacent processors, and be useful to assign cost to the whole execution of an algorithm. We first address the problem of costs of individual, isolated communication operations.

The standard, linear transmission cost model: Following the synchronous model of pairwise communication between adjacent processors, both processors will be involved during the whole message transmission. It is reasonable to assume a certain overhead by both processors for initiating and completing the communication which is independent of the size (number of elements) of the message data to be transferred. This overhead may be dependent on the number of processors p, in general on the communication structure G of the system. In the *standard model*, we assume this overhead, the *start-up latency*, denoted by α, to be independent of p and constant, so $\alpha = O(1)$. For the message of size m (be this elements or Bytes, or some other unit), the standard model assumes a transfer cost per unit, β that is likewise independent of both p and m. The total time cost of transferring a message of m data elements is in the standard, linear (better: affine) transmission cost model as follows:

$$T(m) \quad = \quad \alpha + \beta m$$

The unit will typically be *time* (cost), that is seconds (micro-seconds, milli-seconds, ...). The cost per unit β is sometimes referred to as the *inverse bandwidth* and the βm term as the (inverse) *bandwidth term*. The linear transmission cost model is sometimes called the *Hockney model* [Hoc87, Hoc91, AGH+94, Hoc94]; but misleadingly, since the model has (tacitly) been used elsewhere and long before. For the analysis of algorithms, a first approximation is to assume the same (maximum) latency α and cost per unit β between all pairs of processors, and that these can indeed communicate independently without affecting the costs of each other.

On any modern, clustered, distributed-memory parallel system, benchmarking (for point-to-point communication completion time as a function of message size between a pair of processes in isolation, over all or many possible pairs of processes: not an easy or trivial task at all [HCA16]!) will reveal that the standard model is not corresponding all that well to what is measured. The simple, linear dependence on message size over a large range of message sizes will not hold (different costs per unit in different ranges, discontinuities), and different pairs of processors will have different communication characteristics.

A first refinement of the standard model is to model transmission time by a piecewise linear (affine) function with message thresholds $m_0, m_2, \ldots m_k$ and corresponding latencies and unit costs $\alpha_0, \alpha_1, \ldots, \alpha_k$ and $\beta_0, \beta_1, \ldots, \beta_k$.

$$T(m) \quad = \quad \begin{cases} \alpha_0 + \beta_0 m & m < m_0 \\ \alpha_1 + \beta_1 m & m_0 \leq m < m_1 \\ \ldots & \\ \alpha_k + \beta_k m & m_{k-1} \leq m < m_k \end{cases} \tag{2.2}$$

A somewhat extreme variant of the piecewise linear transmission cost model would be to model communication time by staircase function of the form

$$T(m) \quad = \quad \alpha + \beta b \lfloor m/b \rfloor \tag{2.3}$$

with floor (or range or block or packet) size b. This model would correspond to communication hardware and protocols where messages are sent as packets of some size b with a cost of β per packet. Standard and staircase models with the same α and β parameters are illustrated in Figure 2.13 where the time cost for given message sizes is plotted as a function of the message size.

A second refinement of the standard model is to assign different latencies and costs per unit to different pairs (i, j) of processes

$$T(m) \quad = \quad \alpha_{i,j} + \beta_{i,j} m \tag{2.4}$$

such that the communication time $T(m)$ depends on the identity of both sending and receiving process. On a realistic system, the two processors (cores) could be located inside a shared-memory node, or on different shared-memory nodes and use different communication media in the two cases with different characteristics.

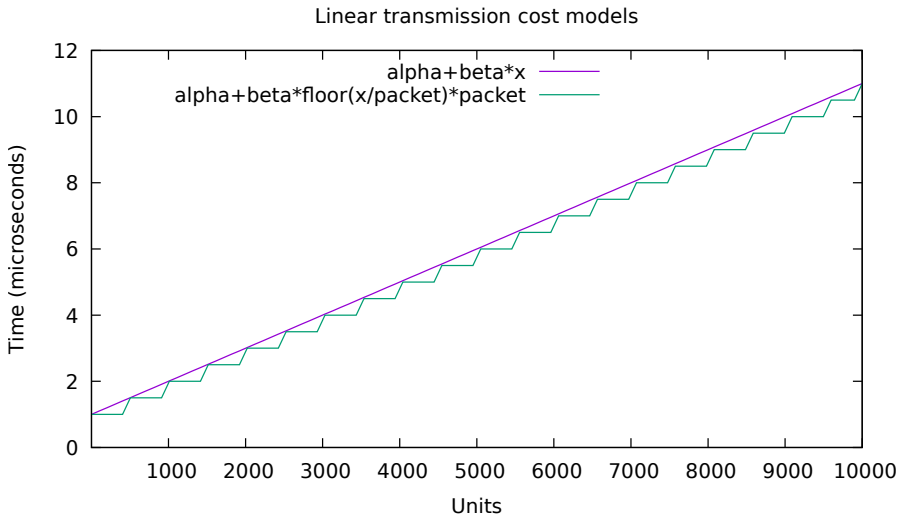


Figure 2.13: Transmission times of data m in range $[0, 10\,000]$ units in the linear and staircase transmission cost models with somewhat realistic values of $\alpha = 0.5\mu s$ and $\beta = 0.001\mu s$/unit.

Communication costs and capabilities: The different (piecewise) linear transmission cost assumptions model uni-directed communication from a sending to a receiving processor. As the communication capabilities allow, and on the assumption of synchronous communication, if both send and receive communication is taking place at the same time, or several send or receive operations, as under multi-ported assumptions, the cost assigned to a processor is the maximum of the costs of these individual operations.

The *LogGP*-family of models: This model claims a more realistic, more global view of communication, and departs from the synchronous transmission model. The models assume a communication medium which makes it possible for P processes to communicate. When a message is to be sent by some process(or) there is first an *overhead* of cost o (time) incurred for preprocessing the message (whatever may be needed) and initiating transmission. A sufficiently short message (corresponding to some packet size) is then transferred to the receiving process without further involvement of the sending process which takes time L, and in the *LogGP* models is referred to as the message *latency*. When the receiver is ready to receive the message, again an overhead o is incurred for making the message available for use. At the sending side, a next message can be sent after a certain (short message) *gap* g. This is the basic *LogP* model introduced in [CKP+93, CKP+96].

2.6.2 Algorithms

With a given communication structure G the assumption is that processors i and j that are connected by an edge $(i, j) \in E(G)$ can communicate as soon as both processors are ready to engage in communication, that is, if there are no delays on either side by a processor having to do other things, be it either computation on local data or other communication along some other edge in G. The costs depend on the message size and

possibly other factors as discussed in Section 2.6.1. Communication between i and j is independent of communication between other pairs of processors. In that sense, despite individual communication operations being synchronous with both processors involved for the full message transmission, the system is asynchronous in that different communication operations with different duration can take place concurrently and overlap each other in "real" time.

Asynchronous, critical path models: A processor in a message-passing algorithm which per definition has no (potential) deadlocks comes to an end since all communication with other processors will eventually take place. It makes sense to define the time complexity of (or time spent by) a processor as the sum over all communication operations, including possible, but necessary delays incurred by communication partners having to do something else before a communication operation takes place, and the sum over all computations that the processor is doing outside of communication operations. The time complexity of a message-passing algorithm is the time complexity of the, in that sense, slowest processor.

More formally, we execute the message-passing algorithm on an input (of size) m on the set of processors of the intended communication structure G and collect a *trace* consisting for each processor of a sequence of weighted communication and computation *activities* with communication edges between matching communication activities (on different processors). Compute activities correspond to local computation for the processors and are assigned the (asymptotic, worst-case) cost of the computation modeled by the activity (say, preprocessing some data for the next communication operation). Communication activities correspond to communication operations (adhering to the communication capabilities of the system) and are assigned the cost for the processor of the corresponding communication operation. The total computational or communication *work* done by a processor, is the sum of the weights of all computation or communication activities for the processor. The total (time) cost for completing an activity at a processor is defined inductively by following the trace of the processor backwards as follows. The time cost of a compute activity at a processor is the cost of the preceding (communication or compute) activity plus the cost assigned with the compute activity. The time cost of a communication activity at a processor is the maximum of the costs of the preceding activities at all processors connected to the activity plus the cost assigned for the communication activity at the processor. The *total (time) cost* for a processor is the total time cost for the last activity in the trace for that processor. We note that we account for communication times locally, per processor.

Definition 2.19 (Message-passing algorithm (time) complexity)**.** *The (worst-case)* time complexity *of a message passing algorithm consisting of a set of sequential algorithms for the $p = |V(G)|$ processors of a communication structure G on input (of size) m appropriately distributed across the processors is the total time cost of the most costly processor in the trace of the algorithm. The* total work *of the algorithm is the the sum of the work of all p processors.*

A trace of an algorithm on communication structure G with p processors on some input m is illustrated in 2.14.

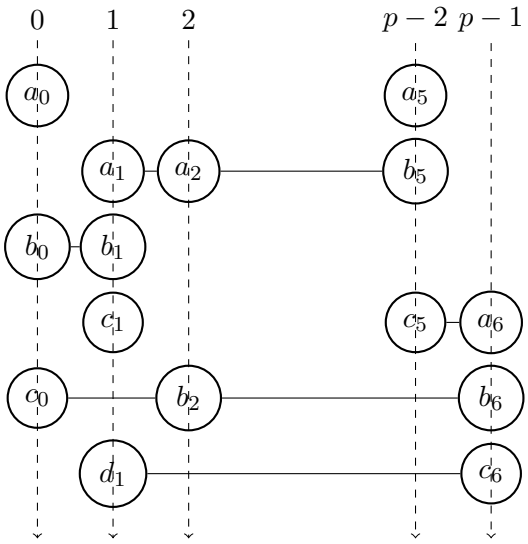


Figure 2.14: Trace of an algorithm on p processors on some input of size m. The activities for processor i are shown with their costs $a_i, b_i, c_i, \ldots$. The activities a_0, a_5, c_1 are compute activities; if a_5 is more than $O(1)$ it constitutes harmful algorithmic latency for communication activity b_5. Likewise, the compute activities a_0, c_0 may or may not be harmful algorithm latency, depending on the communication costs. The communication activates a_1, a_2, b_5, and c_0, b_2, b_6 arise from fully bidirectional communication, where processor 2 simultaneously sends and receives to and from processors 1 and 2, and to and from processors 0 and 6, respectively. In the linear transmission cost model these activities have costs $\alpha + \beta m_i$. Depending on the costs of the communication activities, determined by the amount of data m_i communicated, critical paths may be a_5, a_2, b_2, c_6, or a_0, b_1, c_1, d_1, as well as many others.

By Definition 2.19, the time complexity of a message-passing algorithm is determined by a single, most costly (slowest) processor, and the time for that processor depends, due to communication, on the time at which preceding activities at other processors can be carried out. By following the activities backwards from the slowest processor, a chain of dependent activities can be found with total cost equal to the time cost of this slowest processor. We can call this a most expensive or longest chain of dependent (communication and computation) activities, or a *critical path*. Alternatively, we could have defined the time complexity of a message-passing algorithm as the total cost of the activities along a critical path or most expensive chain of dependent activities. Such a critical path has a number of communication activities, and a number of compute activities, in which we are sometimes interested, and can refer to as the path length. The difference between the total time and the work for a specific processor is the *idle time* of that processor. A critical path has no idle times. Algorithm analysis is thus simply critical path analysis: finding a critical path, and computing the costs by summing the activities on the path.

For algorithms for collective communication operations as will be considered in the following chapters, the compute activities are mostly pre- or postprocessing operations (copying of data, reduction of data) required for the next communication activity. In that sense, these compute activities constitute *overhead* which is harmful if it dominates the total cost of a critical path. We will refer to all such preprocessing computations as the *algorithmic latency* of an algorithm (for some communication operation). Non-constant, algorithmic latency is *harmful*, if the cost of all critical path can be reduced by reducing this overhead.

Synchronous, round based models: Alternatively, a computation can be divided into synchronized *rounds*. The *round complexity* is the number of rounds required in the worst case for inputs of size m..

Chapter notes

An excellent survey on communication structures much in the sense introduced here was given by Fraigniaud and Lazard [FL94]. An early survey of embedddings of communication networks was given by Monien and Sudborough [MS90]. Some excellent, modern (mathematical) graph theory textbooks are [BJG09, Bol98, Die17]. An early graph theory text, much read by computer scientists is the book by Harary [Har69], and also the book by [Ber73], in the same vein see also [Eve12]. See also the compact introduction for computer scientists in [Knu11].

Much algorithm detail on array, tree and hypercubic networks can be found in the book by Leighton [Lei92], of which unfortunately Volume 2 never materialized. A mathematically oriented summery of hypercube graphs can found in [HHW88].

The cube connected cycles were introduced by Preparata and Vuillemin [PV81]. A proof for the diameter claim can be found in [FHL97], and earlier, by way of an optimal routing algorithm, in [MC93]. The former paper does not cite the latter.

The shuffle-exchange network was introduced by Stone [Sto71, LS76] with algorithms for Fast Fourier Transform (FFT), polynomial evaluation and bitonic sorting. The de Bruijn graphs were introduced in [dB46], see for instance [SP89] for use as interconnection

network. The Kautz graph is due to W. F. Kautz [Kau68, Kau69], see for instance [BP89] for use as interconnection network. A summary of graph theoretic methods for the construction of interconnection network with fixed degree and small diameter can be found in [BDQ86].

Standard, recent textbooks on lower level aspects of interconnection networks for parallel and high performance computing are for instance [DT04, DYN03].

The message-passing model itself has many roots, significantly in the idea (in the Dijkstra tradition) of *Communicating Sequential Processes* (CSP) as proposed by Hoare [Hoa78, Hoa85].

Elementary MPI (and OpenMP) oriented textbooks include the books by Pacheco and Quinn [Pac97, Pac11, Qui03]. Specifically for MPI, the "Using MPI" series of books can be useful [GLS94, GLT99, GHTL14]. The indispensable reference for MPI is the standard itself, which any MPI programmer need to consult (better:read) [MPI15, MPI21]. The book by Fox *et al.* arguably significantly influenced the design of MPI [FJL+88]. For historical overviews of MPI design issues, see [HHMW94, HW99].

A discussion of communication structures (topological structure) with MPI communicators can be found in [THMH21].

The *LogP* model was originally introduced by David Culler *et al.* in a series of papers from the mid-1990ties [CKP+93, CKP+96] The necessity of the extension from *LogP* to *LogGP* (and beyond) was argued in [AISS97]. An overview of communication performance models for HPC can be found in [RMML19].

2.7 Exercises

1. Prove that the Cartesian graph product is associative, that is $(F \Box G) \Box H = F \Box (G \Box H)$ for graphs F, G, H.

2. Show that the Cartesian graph product is not commutative, that is $G \Box H \neq H \Box G$ for some G and H.

3. Develop and implement an explicit algorithm for embedding a linear processor array L_p in an arbitrary d-dimensional mesh (or torus) with given dimension orders $D[0], \ldots D[d-1]$ with $p \leq \Pi_{i=0}^{d-1} D[i]$.

4. Implement the algorithms for computing Gray codes and their inverses, that is $\texttt{gray}(i)$ and $\texttt{yarg}(h)$. Construct and conduct an experiment to determine when the fast $\texttt{yarg}(h)$ $O(\log \log h)$ step algorithm is faster than the straightforward $O(\log h)$ algorithm.

5. When a ring is embedded into a d-dimensional hypercube, the successor processor for any hypercube processor h can be computed as $\texttt{gray}((\texttt{yarg}(h) + 1) \mod 2^d)$, and the predecessor by $\texttt{gray}((\texttt{yarg}(h) - 1 + 2^d) \mod 2^d)$. Find and implement a more efficient way of determining which bit of h to flip in order to find either the successor or the predecessor processor of h.

6. Prove that a p-vertex circulant graph C_p^s with d skips $s_0, s_1, \ldots, s_{d-1}$ is connected if and only if $\gcd(p, s_0, s_1, \ldots, s_{d-1}) = 1$.

7. Show by giving a corresponding algorithm that it can be recognized in polynomial time whether a given directed graph $G = (V, E)$ is a circulant graph. The algorithm should compute d and the jumps (skips) $s_0, \ldots, s_{d-1}$ for $C_p^s = G$.

8. Formulate and implement the algorithm sketched for determining a path of length at most $2d + \lfloor d/2 \rfloor - 2$ between any pair of processors i and j in the cube connected cycles Y_d. Prove or disprove that this algorithm always finds a shortest path between i and j.

9. Implement Algorithm X. Prove or disprove that the algorithm always finds a shortest path between i and i'.

Chapter 3

Common Collective Communication and Reduction Operations

In this chapter common, "standard" collective operations, as found in MPI and to some extent in other interfaces and languages for distributed memory parallel programming [EGCSY05] are classified and specified more precisely using the notation previously introduced in Section 2.4. The names for the operations that are introduced are commonplace, and will also be used as names for specific *collective communication problems*.

3.1 Classifying Collectives

In message-passing terminology, communication between processes or processors can be classified as *one-to-one* point-to-point communication from one processor to another or between two processors, as *one-to-many*, where one processor has message(s) to many (or all) other processors in the communication structure, or as *many-to-one*, and finally as *many-to-many*, where many (or all processors) communicate with many other processors. Message passing communication of the one-to-many, many-to-one, and many-to-many form is often referred to as *collective communication*. The characteristic is that many (or all) processors or processes are involved in the communication operation, and not only a pair of processors as in point-to-point communication.

Algorithms or operations where many processors have to participate in order to accomplish a result can conveniently be called *collective* when the emphasis is on a particular subpart of a larger algorithm or application that has to effect a reorganization of distributed data (any non-trivial parallel algorithm is collective, but that is a trivial misuse of terminology and intention). Particular operations implemented by such algorithms for reorganizing data are termed *collective operations*. Collective operations are always of the one-to-many, many-to-one, or many-to-many sort. Collective operations are either purely *data exchange operations*, or can in addition entail operations on the data, and are as such called *reduction operations*. A special collective operation that neither exchanges nor reduces data, but has the purpose of synchronizing the participating processes is the *barrier*.

Operations of the one-to-may or many-to-one sort, where one (or a few processes) are privileged and play a special role are thus *asymmetric* and often called *rooted* to emphasize the special *root* process (or processes). Operations of the many-to-many sort are

Table 3.1: The "standard" collective operations as found in MPI [MPI21].

Synchronization (dense)

	`MPI_Barrier(comm)`

Data exchange (dense)

	Regular	Irregular

Asymmetic (rooted)	`MPI_Bcast(...,root,comm)`
	`MPI_Gather(...,root,comm)`
	`MPI_Scatter(...,root,comm)`
		`MPI_Gatherv(...,root,comm)`
		`MPI_Scatterv(...,root,comm)`

Symmetric (non-rooted)	`MPI_Allgather(...,comm)`
	`MPI_Alltoall(...,comm)`
		`MPI_Allgatherv(...,comm)`
		`MPI_Alltoallv(...,comm)`
		`MPI_Alltoallw(...,comm)`

Reduction (dense)

Asymmetric (rooted)	`MPI_Reduce(...,op,root,comm)`	

Symmetric	`MPI_Allreduce(...,op,comm)`
	`MPI_Reduce_scatter_block(...,op,comm)`
	`MPI_Scan(...,op,comm)`
	`MPI_Exscan(...,op,comm)`
		`MPI_Reduce_scatter(...,op,comm)`

Data exchange (sparse)

Symmetric	`MPI_Neighbor_allgather(...,comm)`
	`MPI_Neighbor_alltoall(...,comm)`
		`MPI_Neighbor_allgatherv(...,comm)`
		`MPI_Neighbor_alltoallv(...,comm)`
		`MPI_Neighbor_alltoallw(...,comm)`

symmetric. Collective operations describe data exchanges and possibly operations on the data between processes. If the amount of data exchanged between any pair of processes is the same for all pairs of processes, the operation is called *regular*, and otherwise *irregular*.

Collective operations where "many" means "all", that is operations of the sort *one-to-all*, *all-to-one*, or *all-to-all* will be called *dense*, and operations where "many" means "few", will in contrast be called "sparse".

The collective operations in MPI The MPI collectives classified according to the discussion above are listed in Table 3.1. MPI provides the barrier operation, a set of data exchange, and a set of reduction operations, both asymmetric (rooted) and symmetric, in regular and irregular variants. There is also a smaller collection of symmetric, sparse data exchange operations, the socalled neighborhood collectives.

3.2 Specifying Collectives

Let $V = \{0, 1, \ldots, p-1\}$ be a set of p processors (in a communication structure $G = (V, E)$). Using the notation introduced in Section 2.4, we specify collective operations Broadcast, Gather, Scatter, Allgather, Alltoall, Shift, Permute, Reduce, ReduceAll, ReduceScatter, ScanExclusive, and ScanInclusive that cover and generalize the MPI collectives. The specifications immediately give rise to a number of useful observations. The names will also be used as names and precise specifications of the communication problems that will be studied in the following chapters.

The Broadcast operation is defined in (3.1). A designated *root processor* (or just *root*) $r, 0 \leq r < p$ has a data block $[m]$ of m elements. This block has to be distributed to all other processors in V, such that after the operation, each processor has a data block $[m]$ with the elements from the data block of the root in the same linear order, but possibly with a different structure (which is ignored by the block $[m]$ notation).

$$\begin{array}{lccccccc} & 0 & 1 & \ldots & r & \ldots & p-1 \\ & [] & [] & \ldots & [m] & \ldots & [] \\ \hline \text{Broadcast} & [m] & [m] & \ldots & [m] & \ldots & [m] \end{array} \tag{3.1}$$

The Broadcast(r, m, p) problem on p processors is classified by the size of the data to be broadcast m and the root r. It is assumed that all processors know the identity of the single root r, as well as the problem size m of data elements to be broadcast. The corresponding MPI operation is `MPI_Bcast()`.

The Gather collective operation is defined in (3.2). Each processor i has an individual block of data $[m_i]$, and all p data blocks are to be concatenated into a larger block $[m_0 m_1 \cdots m_r \cdots m_{p-1}]$ at the given root processor $r, 0 \leq r < p$. Also the root r has an input data block $[m_r]$, that could consist of $m_r = 0$ elements.

$$\begin{array}{lccccccc} & 0 & 1 & \ldots & r & \ldots & p-1 \\ & [m_0] & [m_1] & \ldots & [m_r] & \ldots & [m_{p-1}] \\ \hline \text{Gather} & [] & [] & \ldots & [m_0 m_1 \cdots m_r \cdots m_{p-1}] & \ldots & [] \end{array} \tag{3.2}$$

An alternative definition of Gather is given in (3.3). Here $[m] = [m_0 m_1 \cdots m_r \cdots m_{p-1}]$ describes a (virtual) block of $m = \sum_{i=0}^{p-1} m_i$ data elements that is to be gathered at the root process r from disjoint parts of the block that are distributed across the p processors, with processor i having m_i data elements of $[m]$ from (for $i > 0$) $\sum_{j=0}^{i-1} m_j$ to $(\sum_{j=0}^{i} m_j) - 1$ (for processor $i = 0$, from 0 to $m_0 - 1$) which is denoted by $[m][\sum_{j=0}^{i-1} m_j : (\sum_{j=0}^{i} m_j) - 1]$ (and $[m][0 : m_0 - 1]$).

$$\begin{array}{lccccc} & 0 & \ldots & r & \ldots & p-1 \\ & [m][0 : m_0 - 1] & \ldots & [m][\sum_{j=0}^{r-1} m_j : (\sum_{j=0}^{r} m_j) - 1] & \ldots & [m][\sum_{j=0}^{p-2} m_j : (\sum_{j=0}^{p-1} m_j) - 1] \\ \hline \text{Gather} & [] & \ldots & [m] & \ldots & [] \end{array} \tag{3.3}$$

In the Gather(r, m, p) problem, the identity of the single root processor r is known to all p processors. The total problem size is $m = \sum_{i=0}^{p-1} m_i$, the amount of data to be

gathered at the root. Each processor i knows the size of its own data block m_i, and the root knows the size of the data blocks from all processors. In the specification in terms of a virtual block $[m]$ known to all processors, the non-root processors i will in addition be assumed to know the total size of the data blocks of "preceding" processors $j < i$. The corresponding MPI operations are the regular `MPI_Gather()`, in which all data blocks m_j have the same size, and the irregular `MPI_Gatherv()`, where indeed different processors may contribute blocks of different sizes. The root in this case has a vector giving the sizes of the blocks for the processors, and a vector specifying the displacement of each block in the linearly ordered block $[m]$.

The Scatter collective operation is defined in (3.4). A designated root processor $r, 0 \leq r < p$ has a data block $[m_0 m_1 \cdots m_r \cdots m_{p-1}]$ consisting of p individual, disjoint blocks $[m_i]$ for each of the p processors (including the root itself). The individual subblocks are to be distributed to the p processors with block $[m_i]$ for processor i.

$$\begin{array}{c|cccccc} & 0 & 1 & \ldots & r & \ldots & p-1 \\ & [] & [] & \ldots & [m_0 m_1 \cdots m_r \cdots m_{p-1}] & \ldots & [] \\ \hline \text{Scatter} & [m_0] & [m_1] & \ldots & [m_r] & \ldots & [m_{p-1}] \end{array} \tag{3.4}$$

An alternative definition, similar to (3.3) can likewise be given. For the Scatter(r, m, p) problem, again the root r, which is assumed to be known to all p processors, must know the sizes of all data blocks m_i and can easily compute also the total problem size $m = \sum_{i=0}^{p-1} m_i$. The corresponding MPI operations are the regular `MPI_Scatter()` and the irregular `MPI_Scatterv()`.

The Allgather collective operation is defined in (3.5). Each processor has a block of data $[m_i]$, and the p blocks are to be concatenated into a larger block $[m_0 m_1 \cdots m_r \cdots m_{p-1}]$ for all processors.

$$\begin{array}{c|cccccc} & 0 & 1 & \ldots & i & \ldots & p-1 \\ & [m_0] & [m_1] & \ldots & [m_i] & \ldots & [m_{p-1}] \\ \hline \text{Allgather} & \begin{bmatrix} m_0 \\ m_1 \\ \vdots \\ m_i \\ \vdots \\ m_{p-1} \end{bmatrix} & \begin{bmatrix} m_0 \\ m_1 \\ \vdots \\ m_i \\ \vdots \\ m_{p-1} \end{bmatrix} & \ldots & \begin{bmatrix} m_0 \\ m_1 \\ \vdots \\ m_i \\ \vdots \\ m_{p-1} \end{bmatrix} & \ldots & \begin{bmatrix} m_0 \\ m_1 \\ \vdots \\ m_i \\ \vdots \\ m_{p-1} \end{bmatrix} \end{array} \tag{3.5}$$

In the Allgather(m, p) problem, all processors $i, 0 \leq i < p$ contribute a data block $[m_i]$, and these blocks are concatenated at all processors. The problem is therefore sometimes called the *concatenation problem* [BHK+97, BBC+95]. All processors are assumed to know all block sizes m_i, and can easily compute the total problem size $m = \sum_{i=0}^{p-1} m_i$ as well. The corresponding MPI operations are the regular `MPI_Allgather()`, in which all blocks have the same size, given as part of the input, and the irregular `MPI_Allgatherv()`, in which blocks from different processors may have different sizes, described by input vectors of block counts and displacements.

The Alltoall collective operation is defined in (3.6). Each processor i has an input block of data $[m_i] = [m_i^0 m_i^1 \cdots m_i^{p-1}]$ consisting of p individual blocks $[m_i^j]$ for each processor $j, 0 \leq j < p$. The jth individual block of processor i is to be communicated to processor j, such that each processor j ends with an output block of data consisting of the blocks $[m^j] = [m_0^j m_1^j \cdots m_{p-1}^j]$.

$$\text{Alltoall} \quad \frac{\begin{array}{cccccc} 0 & 1 & \dots & i & \dots & p-1 \\ \left[\begin{array}{c} m_0^0 \\ m_0^1 \\ \vdots \\ m_0^i \\ \vdots \\ m_0^{p-1} \end{array}\right] & \left[\begin{array}{c} m_1^0 \\ m_1^1 \\ \vdots \\ m_1^i \\ \vdots \\ m_1^{p-1} \end{array}\right] & \cdots & \left[\begin{array}{c} m_i^0 \\ m_i^1 \\ \vdots \\ m_i^i \\ \vdots \\ m_i^{p-1} \end{array}\right] & \cdots & \left[\begin{array}{c} m_{p-1}^0 \\ m_{p-1}^1 \\ \vdots \\ m_{p-1}^i \\ \vdots \\ m_{p-1}^{p-1} \end{array}\right] \end{array}}{\begin{array}{cccccc} \left[\begin{array}{c} m_0^0 \\ m_1^0 \\ \vdots \\ m_i^0 \\ \vdots \\ m_{p-1}^0 \end{array}\right] & \left[\begin{array}{c} m_0^1 \\ m_1^1 \\ \vdots \\ m_i^1 \\ \vdots \\ m_{p-1}^1 \end{array}\right] & \cdots & \left[\begin{array}{c} m_0^i \\ m_1^i \\ \vdots \\ m_i^i \\ \vdots \\ m_{p-1}^i \end{array}\right] & \cdots & \left[\begin{array}{c} m_0^{p-1} \\ m_1^{p-1} \\ \vdots \\ m_i^{p-1} \\ \vdots \\ m_{p-1}^{p-1} \end{array}\right] \end{array}} \tag{3.6}$$

In the Alltoall(m, p) problem, characterized by the total amount of data to be exchanged m between p processors, all processors have an individual block destined to each of the other processors, including the processor itself. The Alltoall operation is therefore sometimes called *personalized exchange*. Viewing the input as a matrix distributed column wise over the p processors, Definition (3.6) states that the matrix is being transposed, and the Alltoall operation is therefore sometimes referred to as *transpose*. Each processor i is assumed to know the sizes of the blocks that are sent m_i^j, and the sizes blocks that are received m_j^i. The amount of data to be sent and received by processor i is $m_i = \sum_{j=0}^{p-1} m_i^j$ and $m^i = \sum_{j=0}^{p-1} m_j^i$), respectively. It may, in the general case, well be that $m_i \neq m^i$ for any one processor i, but by definition of the problem, the total amount of data sent equals the total amount of data received, $m = \sum_{i=0}^{p-1} m_i = \sum_{i=0}^{p-1} m^i$, and is the total amount of data to be exchanged. This may not be known to any one processor. The corresponding MPI operations are the regular `MPI_Alltoall()`, in which all block sizes are the same, and given as a parameter by each calling process, and the irregular `MPI_Alltoallv()` and `MPI_Alltoallw()`, in which individual block sizes are described by vectors of counts and displacements, and element data structure descriptions (MPI datatypes) for `MPI_Alltoallw()`.

The Shift collective operation with shift (or offset) k (positive, or negative) is defined in (3.7). Each processor $i, 0 \leq i < p$ sends its data block $[m_i]$ to processor $(i + k)_p$.

$$\text{Shift} \quad \frac{\begin{array}{cccccc} 0 & 1 & \dots & r & \dots & p-1 \\ [m_0] & [m_1] & \dots & [m_i] & \dots & [m_{p-1}] \end{array}}{\begin{array}{cccccc} [m_{(0-k)_p}] & [m_{(1-k)_p}] & \dots & [m_{(i-k)_p}] & \dots & [m_{(p-1-k)_p}] \end{array}} \tag{3.7}$$

In the $\mathsf{Shift}(k, m, p)$ problem, the problem size is $\sum_{i=0}^{p-1} m_i$, the total size of all blocks that are sent and received, and (may) not (be) known to any one processor; but it is assumed that processor i knows both the size of the block it is sending (its own block) m_i, and the block that it will receive from processor $(i-k)_p$. In the regular Shift problem, all block sizes m_i would be the same, whereas they could be different in the irregular variant.

The $\mathsf{Permute}$ collective operation is defined in (3.8). Let $\pi : \{0, 1, \ldots, p-1\} \to \{0, 1, \ldots, p-1\}$ be a given permutation. Each processor $i, 0 \leq i < p$ sends its data block $[m_i]$ to processor $\pi(i)$.

$$\mathsf{Permute} \quad \begin{array}{cccccc} 0 & 1 & \ldots & r & \ldots & p-1 \\ \hline [m_0] & [m_1] & \ldots & [m_r] & \ldots & [m_{p-1}] \\ \hline [m_{\pi^{-1}(0)}] & [m_{\pi^{-1}(1)}] & \ldots & [m_{\pi^{-1}(r)}] & \ldots & [m_{\pi^{-1}(p-1)}] \end{array} \tag{3.8}$$

The $\mathsf{Permute}(\pi, m, p)$ problem is characterized by the total problem size $\sum_{i=0}^{p-1} m_i$ and the permutation π, and both π and its inverse π^{-1} are assumed to be known and/or hardwired into the algorithm that solves the $\mathsf{Permute}$ problem. Processor i likewise knows both the size of its own block m_i and the size of the block it will receive $m_{\pi^{-1}(i)}$.

The Shift operation is a special case of the $\mathsf{Permute}$ operation. There are no corresponding MPI collectives for the Shift and $\mathsf{Permute}$ operations, but these operations have been studied as fundamental, and are sometimes included as fundamental communication operations [BBC+95].

Reduction operators: Collective reduction operations apply some given operator on the input blocks from one or more processors in order to arrive at a combined result distributed over (some of) the processors. The combining operators will be taken as binary operators that combine data blocks pairwise. In order to make the specifications well-defined, the properties of the permitted binary operators must likewise be fixed precisely.

Let $\circ$ be a binary operator on data blocks $[m]$ that combines two data blocks $[m_i]$ and $[m_j]$ into a new data block

$$[m_k] \quad = \quad [m_i] \circ [m_j]$$

The operator $\circ$ is said to be *associative* if for any three data blocks $[m_i], [m_j], [m_k]$ it holds that

$$([m_i] \circ [m_j]) \circ [m_k] \quad = \quad [m_i] \circ ([m_j] \circ [m_k])$$

and we can therefore omit the brackets and just write $[m_k] \circ [m_i] \circ [m_j]$. By induction it follows that for any finite number of data blocks $[m_i], i = 0, \ldots, p-1$

$$\begin{array}{rcl} (\bigcirc_{i=0}^{p-2}[m_i]) \circ [m_{p-1}] & = & [m_0] \circ (\bigcirc_{i=1}^{p-1}[m_i]) \\ & = & \bigcirc_{i=0}^{p-1}[m_i] \end{array}$$

and so the same result will follow irregardless of where the brackets have been set. All binary operators to be considered from now on will, for this reason, be associative (and so the set of data blocks $[m]$ with binary operator $\circ$ will be a *semigroup*). Associativity is a

mathematical property, which does not always hold when objects (numbers) are represented in finite precision (32- or 64-bit floating point numbers, for instance); sometimes this is important to take into consideration, and perform the reductions with a particular bracketing. Mostly, however, algorithms for collective reduction operations will freely rely on associativity.

Mostly, the operator $\circ$ will operate on data blocks of the same size $m = m_i = m_j$ in a *conservative* way so as to produce also a data block $[m_k] = [m_i] \circ [m_j]$ of the same size $m = m_k$. Operators can for instance be thought of as element-wise (arithmetical) operations on n-element vectors like element-wise summation

$$\begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{pmatrix} + \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} x_0 + y_0 \\ x_1 + y_1 \\ \vdots \\ x_{n-1} + y_{n-1} \end{pmatrix} \tag{3.9}$$

and thus be conservative. Concatenation of two blocks $[m_i]$ and $[m_j]$ into a block $[m_i m_j]$ is an example of a non-conservative, albeit associative operator.

An operator $\circ$ on data blocks $[m]$ is said to be *idempotent* if $[m] \circ [m] = [m]$. An operator $\circ$ on data blocks $[m_i]$ and $[m_j]$ is said to be *commutative* if $[m_i] \circ [m_j] = [m_j] \circ [m_i]$ for any two blocks $[m_i]$ and $[m_j]$. An operator $\circ$ is said to have a *neutral element* $[e]$ if $[e] \circ [m] = [m] \circ [e] = [m]$. It is easy to see that such a neutral element is unique. For an operator with neutral element $[e]$, a data block $[m]$ is said to have an *inverse element*, here denoted by $-[m]$, if $[m] \circ -[m] = -[m] \circ [m] = [e]$. For a commutative operator $\circ$, it can easily be shown that

$$\bigcirc_{i=0}^{p-1} [m_i] \quad = \quad \bigcirc_{i=0}^{p-1} [m_{\pi(i)}]$$

for any permutation $\pi : \{0, 1, \ldots, p-1\} \to \{0, 1, \ldots, p-1\}$, and the $\circ$ operator can therefore be applied on the operands in any order and still give the desired result $\bigcirc_{i=0}^{p-1} [m_i]$.

Standard, mathematical operators like addition $+$ and multiplication $\cdot$ over natural, real or complex numbers, etc., are associative and commutative, and have both neutral and inverse elements. These, and similar operations, e.g., bitwise operations on words, will be extended to linearly ordered data blocks (vectors) by element-wise application as shown in Equation (3.9), and this extension will retain properties like associativity and commutativity. It is interesting to note that bitwise disjunction (or) and bitwise conjunction (and), are both idempotent (associative and commutative) operations, $x \wedge x = x, x \vee x = x$. The aritmetical operators are not idempotent, $x + x \neq x$ (unless $x = 0$). Bitwise exclusive or is not idempotent (but still associative and commutative), but has the property that $x \bigtriangledown x = \texttt{0x0}$, so any x is its own inverse. Also the associative, commutative minimum and maximum operators are idempotent, $\max(x, x) = \min(x, x) = x$.

Not all relevant binary operators are commutative. For instance, data blocks $[n \times n]$ could be (small) $x \times n$ matrices; matrix product $[n \times n] \times [n \times n]$ is not a commutative operation. A seemingly uninteresting non-commutative operator is the *replacement operator* $\lhd$ defined by

$$x \lhd y \quad = \quad x$$

which is associative since $(x \lhd y) \lhd z = x \lhd z = x = x \lhd y = x \lhd (y \lhd z)$.

An important class of non-commutative operators on "marked" pairs $\langle h, x\rangle$ where x is an object from a set with an associative operator $\circ$, and h a mark, either Boolean (**true**/**false**, with $h =$ **true** indicating that the corresponding x-element has been marked) or, say, $h \in \{\bot, 0, 1, \ldots, p-1\}$ (with $h = \bot$ indicating an unmarked x-element) is the derived, *segmented operators* defined by

$$\langle h_0, x_0\rangle \circ_{\text{seg}} \langle h_1, x_1\rangle \quad = \quad \begin{cases} \langle h_0, x_0 \circ x_1\rangle & \text{if } h_1 = \bot \text{ (unmarked)} \\ \langle h_1, x_1\rangle & \text{if } h_1 \neq \bot \text{ (marked)} \end{cases}$$

Segmented operators, including segmented replacement $\lhd_{\text{seg}}$, are surprisingly powerful when used with collective reduction operations, as will be seen often in the following chapters.

Similar, derived operators are the *location operators* for the minimum and maximum operators, defined by

$$\max_{\text{loc}}(\langle h_0, x_0\rangle, \langle h_1, x_1\rangle) \quad = \quad \begin{cases} \langle h_0, x_0\rangle & \text{if } x_0 > x_1 \\ \langle \min(h_0, h_1), x_0\rangle & \text{if } x_0 = x_1 \\ \langle h_1, x_1\rangle & \text{if } x_0 < x_1 \end{cases}$$

and

$$\min_{\text{loc}}(\langle h_0, x_0\rangle, \langle h_1, x_1\rangle) \quad = \quad \begin{cases} \langle h_0, x_0\rangle & \text{if } x_0 < x_1 \\ \langle \min(h_0, h_1), x_0\rangle & \text{if } x_0 = x_1 \\ \langle h_1, x_1\rangle & \text{if } x_0 > x_1 \end{cases}$$

where here the marks $h_0, h_1 \in \{0, 1, \ldots, p-1\}$. Note that in contrast to the segmented operators, the location operators are (obviously) commutative.

For operators $\circ$ with neutral elements, blocks of data $[m]$ can possibly be (and sometimes are) represented compactly in some compressed form $[m']$ by leaving out neutral elements, and/or by compressing sequences of identical elements (run-length encoding), meaning that block $[m]$ can be reconstructed uniquely (lossless compression) from the smaller block $[m']$, and the $\circ$ operator extended to work element-wise on such compressed blocks $[m'_i]$ and $[m'_j]$ by simply not performing the operation for the neutral elements (that were left out in $[m'_i]$ and $[m'_j]$). Such representations and reduction operators and collective operations are sometimes called *sparse reductions* [Trä10, LH22]. Sparse operator applications $[m'_i] \circ [m'_j] = [m'_k]$ are not conservative, and all three blocks $[m'_i]$, $[m'_j]$ and $[m'_k]$ may have different sizes and structures.

Like associativity, commutativity, idempotence, and the existence of neutral and inverse elements are properties that can be exploited in algorithms for collective reduction operations, and sometimes separate the complexity of variations of the reduction problems. It is therefore important to state which properties are assumed; most algorithms in the following chapter will rely solely on associativity, and sometimes go to certain lenghts to ensure this.

MPI defines a small set of associative, element-wise (and thus conservative) binary operators that can be used in the MPI reduction collectives. These are discussed in more detail in Section 4.2. It is likewise possible for a programmer to define own associative, and possibly commutative operators, likewise explained in Section 4.2. The function `MPI_Reduce_local()` makes it possible to apply any such pre- or user-defined operator on two input blocks (one of which is also the output, and thus desctructively updated).

Reduction collectives: The Reduce collective operation for an associative binary operator $\circ$ is defined in (3.10). Each processor contributes a block $[m_i]$, and the result of applying $\circ$ over the blocks $[m] = \bigcirc_{i=0}^{p-1}[m_i]$ is stored at the designated root processor $r, 0 \leq r < p$.

$$\begin{array}{lccccccc} & 0 & 1 & \dots & r & \dots & p-1 \\ & [m_0] & [m_1] & \dots & [m_r] & \dots & [m_{p-1}] \\ \textsf{Reduce} & \multicolumn{6}{c}{\text{---------------------------------------}} \\ & [] & [] & \dots & [m] & \dots & [] \end{array} \tag{3.10}$$

In the Reduce$(\circ, r, m)$ problem, all processors are assumed to know the identity r of the root processor, and are assumed to apply the same operator $\circ$. The size m of the output m is assumed to be known at (or computed by) the root. Normally, the operator $\circ$ is conservative, so all input blocks $[m_i]$ will have the same size m as the computed result data block $[m]$. The MPI operation implementing this specification is `MPI_Reduce()`.

The ReduceAll collective operation for an associative binary operator $\circ$ is defined in (3.11). Each processor contributes a block $[m_i]$, and the result of applying $\circ$ over the blocks $[m] = \bigcirc_{j=0}^{p-1}[m_j]$ is stored at all processors $i, 0 \leq i < p$.

$$\begin{array}{lccccccc} & 0 & 1 & \dots & i & \dots & p-1 \\ & [m_0] & [m_1] & \dots & [m_i] & \dots & [m_{p-1}] \\ \textsf{ReduceAll} & \multicolumn{6}{c}{\text{---------------------------------------}} \\ & [m] & [m] & \dots & [m] & \dots & [m] \end{array} \tag{3.11}$$

In the ReduceAll$(\circ, m)$ problem, the processors are all assumed to apply the same operator $\circ$. The size m of the output $[m]$ is likewise assumed to be known to all processors. Normally, the operator $\circ$ is conservative, so all input blocks $[m_i]$ will have the same size m as the computed result data block $[m]$. The MPI operation implementing this specification is `MPI_Allreduce()`.

The ReduceScatter collective operation for an associative binary operator $\circ$ is defined in (3.12). Each processor i has a block of data $[m] = [m_i^0 m_i^1 \cdots m_i^{p-1}]$ consisting of p individual blocks $[m_i^j]$ for $0 \leq j < p$. Let $[m^i] = \bigcirc_{j=0}^{p-1}[m_j^i]$ be the block computed for processor i from the ith input blocks $[m_j^i]$ from all processors $j, 0 \leq j < p$.

$$\begin{array}{lccccccc} & 0 & 1 & \dots & i & \dots & p-1 \\ & \begin{bmatrix} m_0^0 \\ m_0^1 \\ \vdots \\ m_0^i \\ \vdots \\ m_0^{p-1} \end{bmatrix} & \begin{bmatrix} m_1^0 \\ m_1^1 \\ \vdots \\ m_1^i \\ \vdots \\ m_1^{p-1} \end{bmatrix} & \dots & \begin{bmatrix} m_i^0 \\ m_i^1 \\ \vdots \\ m_i^i \\ \vdots \\ m_i^{p-1} \end{bmatrix} & \dots & \begin{bmatrix} m_{p-1}^0 \\ m_{p-1}^1 \\ \vdots \\ m_{p-1}^i \\ \vdots \\ m_{p-1}^{p-1} \end{bmatrix} \\ \textsf{ReduceScatter} & \multicolumn{6}{c}{\text{---------------------------------------}} \\ & [m^0] & [m^1] & \dots & [m^i] & \dots & [m^{p-1}] \end{array} \tag{3.12}$$

The ReduceScatter$(\circ, m)$ problem is characterized by the input size $m = \sum_{j=0}^{p-1} m_i^j$ which is assumed to be the same for all processors. Each processor is assumed to divide $[m] = [m^0 m^1 \cdots m^{p-1}]$ in the same way. Likewise, all processors are assumed to use the

same associative operation $\circ$. The MPI operations implementing this specification are `MPI_Reduce_scatter_block()` for the regular case where all m_i^j have the same size m/p, for each processor i, and each subblock j, and `MPI_Reduce_scatter()` for the irregular case where m_i^j may differ for different j.

The ScanExclusive collective operation for an associative binary operator $\circ$ is defined in (3.13). Each processor $i, 0 \leq i < p$ contributes an input block $[m_i]$, and computes, for $i > 0$, the output $\bigcirc_{j=0}^{i-1}[m_j]$.

$$\begin{array}{lcccccc} & 0 & 1 & \dots & i & \dots & p-1 \\ & [m_0] & [m_1] & \dots & [m_r] & \dots & [m_{p-1}] \\ \hline \mathsf{ScanExclusive} & [] & [m_0] & \dots & \bigcirc_{j=0}^{i-1}[m_j] & \dots & \bigcirc_{j=0}^{p-2}[m_j] \end{array} \tag{3.13}$$

The ScanExclusive$(\circ, m, p)$ problem is the message-passing *exclusive prefix-sums problem* over blocks of data. The MPI operation implementing this specification is `MPI_Exscan()`.

The ScanInclusive collective operation for an associative binary operator $\circ$ is defined in (3.14). Each processor $i, 0 \leq i < p$ contributes an input block $[m_i]$, and computes the output $\bigcirc_{j=0}^{i}[m_j]$.

$$\begin{array}{lcccccc} & 0 & 1 & \dots & i & \dots & p-1 \\ & [m_0] & [m_1] & \dots & [m_r] & \dots & [m_{p-1}] \\ \hline \mathsf{ScanInclusive} & [m_0] & [m_0] \circ [m_1] & \dots & \bigcirc_{j=0}^{i}[m_j] & \dots & \bigcirc_{j=0}^{p-1}[m_j] \end{array} \tag{3.14}$$

The ScanInclusive$(\circ, m, p)$ problem is the message-passing *inclusive prefix-sums problem*. The MPI counterpart is `MPI_Scan()`.

3.3 Relationships between Collective Operations

Many of the collective operations defined so far are related in intersting and sometimes non-trivial ways that can later be used for designing and judging algorithms that implement these operations.

Let $[m] = [m_0 m_1 \cdots m_{p-1}]$ be the block to be broadcast in a Broadcast operation from some root r. The input at the root processor r to the Scatter operation consists of a data block divided into p smaller blocks for each of the p processors, and results in a block $[m_i]$ for each processor i. This is the input for the Allgather collective, which collects at each processor a block $[m_0 m_1 \cdots m_{p-1}] = [m]$. This simple observation shows that any Broadcast operation can be accomplished by scattering the input into p smaller blocks, followed by an Allgather operation. Equation (3.15) summarizes the observation by stating that solving a Broadcast(r, m) problem is equivalent to solving a Scatter(r, m) problem, followed by an Allgather(m) problem, as denoted by the $|$ operator. By the same argument, ReduceScatter can be used with Allgather and Gather for the same effect as ReduceAll and Reduce, regardless of how the input $[m_i]$ is concatenated from smaller blocks, $[m_i] = [m_i^0 m_i^1 \cdots m_i^{p-1}]$. These oberservations are summarized in Equations (3.16) and (3.17). The last observations trivially expands the definitions of the ReduceScatter, ReduceAll and Allgather operations.

$$\mathsf{Broadcast}(r,m,p) = \mathsf{Scatter}(r,m,p) \mid \mathsf{Allgather}(m,p) \tag{3.15}$$
$$\mathsf{ReduceAll}(\circ,r,m,p) = \mathsf{ReduceScatter}(\circ,m,p) \mid \mathsf{Allgather}(m,p) \tag{3.16}$$
$$\mathsf{Reduce}(\circ,r,m,p) = \mathsf{ReduceScatter}(\circ,m,p) \mid \mathsf{Gather}(r,m,p) \tag{3.17}$$
$$\mathsf{ReduceScatter}(\circ,m,p) = \mathsf{Reduce}(\circ,r,m,p) \mid \mathsf{Scatter}(r,m,p) \tag{3.18}$$
$$\mathsf{ReduceAll}(\circ,m,p) = \mathsf{Reduce}(\circ,r,m,p) \mid \mathsf{Broadcast}(r,m,p) \tag{3.19}$$
$$\mathsf{Allgather}(m,p) = \mathsf{Gather}(r,m,p) \mid \mathsf{Broadcast}(r,m,p) \tag{3.20}$$

Equations (3.21) and (3.22) associate the input/output block of size $[m] = [m_r]$ with a single processor r ($m_i = 0$ for all other processors). With this input/output distribution, an Allgather has the effect of a Broadcast, and a ReduceScatter the effect of a Reduce.

$$\mathsf{Broadcast}(r,m) = \mathsf{Allgather}(m) \tag{3.21}$$
$$\mathsf{Reduce}(\circ,r,m) = \mathsf{ReduceScatter}(\circ,m) \tag{3.22}$$

The following observations solve collective communication problems (left hand side) by a sequence p independent, simpler operations, that can be carried out sequentially in any order, or even concurrently as is indicated by the $\|$ operator. Let for Allgather $[m] = [m_0 m_1 \cdots m_{p-1}]$. Let $[m_i^j]$ be the input data blocks for the Alltoall operations. Let for Equation (3.25) $m^k = \sum_{i=0}^{p-1} m_i^{(i+k)_p}$.

$$\begin{aligned}\mathsf{Allgather}(m,p) = {} & \mathsf{Broadcast}(0,m_0,p) \parallel \mathsf{Broadcast}(1,m_1,p) \parallel \ldots \parallel \\ & \mathsf{Broadcast}(p-1,m_{p-1},p) \end{aligned} \tag{3.23}$$
$$\begin{aligned}\mathsf{Allgather}(m,p) = {} & \mathsf{Gather}(0,m,p) \parallel \mathsf{Gather}(1,m,p) \parallel \ldots \parallel \\ & \mathsf{Gather}(p-1,m,p) \end{aligned} \tag{3.24}$$
$$\begin{aligned}\mathsf{Alltoall}(r,m,p) = {} & \mathsf{Shift}(0,m^0,p) \parallel \mathsf{Shift}(1,m^1,p) \parallel \ldots \parallel \\ & \mathsf{Shift}(p-1,m^{p-1},p) \end{aligned} \tag{3.25}$$
$$\begin{aligned}\mathsf{Alltoall}(r,m,p) = {} & \mathsf{Gather}(0,m^0,p) \parallel \mathsf{Gather}(1,m^1,p) \parallel \ldots \parallel \\ & \mathsf{Gather}(p-1,m^{p-1},p) \end{aligned} \tag{3.26}$$
$$\begin{aligned}\mathsf{Alltoall}(r,m,p) = {} & \mathsf{Scatter}(0,m_0,p) \parallel \mathsf{Scatter}(1,m_1,p) \parallel \ldots \parallel \\ & \mathsf{Scatter}(p-1,m_{p-1},p) \end{aligned} \tag{3.27}$$
$$\begin{aligned}\mathsf{ReduceScatter}(\circ,m,p) = {} & \mathsf{Reduce}(\circ,0,m^0,p) \parallel \mathsf{Reduce}(\circ,1,m^1,p) \parallel \ldots \parallel \\ & \mathsf{Reduce}(\circ,p-1,m^{p-1},p) \end{aligned} \tag{3.28}$$
$$\begin{aligned}\mathsf{ReduceAll}(\circ,m,p) = {} & \mathsf{Reduce}(\circ,0,m,p) \parallel \mathsf{Reduce}(\circ,1,m,p) \parallel \ldots \parallel \\ & \mathsf{Reduce}(\circ,p-1,m,p) \end{aligned} \tag{3.29}$$
$$\tag{3.30}$$

The final observations exploits properties of ScanInclusive, and the special, replacement operator $\triangleleft$.

$$\mathsf{Reduce}(\circ,p-1,m,p) = \mathsf{ScanInclusive}(\circ,m,p) \tag{3.31}$$
$$\mathsf{Broadcast}(0,m,p) = \mathsf{ScanInclusive}(\triangleleft,m,p) \tag{3.32}$$

3.4 Performance Guidelines for Collective Operations

The oberservations of the last section have algorithmic ramifications with respect to both model as well as practically measured performance. Let $\mathsf{X} \preceq \mathsf{Y}$ denote the expectation that operation X performs no worse that operation Y under the stated circumstances (total problem size, distribution, number of processors). We call such an expectation a *performance guideline*.

All equivalences stated above translate into performance guidelines. In addition, the following guidelines follow by specialization:

$$\mathsf{Gather}(r, m, p) \preceq \mathsf{Allgather}(m, p) \tag{3.33}$$

$$\mathsf{Scatter}(r, m, p) \preceq \mathsf{Broadcast}(m, p) \tag{3.34}$$

$$\mathsf{Reduce}(\circ, r, m, p) \preceq \mathsf{ReduceAll}(\circ, m, p) \tag{3.35}$$

$$\mathsf{Allgather}(m, p) \preceq \mathsf{Alltoall}(m, p) \tag{3.36}$$

3.5 Chapter notes

The notation used for specifying the collective operations is inspired by the notation used by van de Geijn in several papers dating back to the early 90ties [CHPvdG07]. The Shift and Permute operations are not in MPI, but have been discussed in the literature [BBC+95]. The idea of performance guidelines as a means to assess the quality of an MPI library implementations was introduced in [TGT10].

3.6 Exercises

1. Show that the derived operator $\circ_{\text{seg}}$ is asociative for any associative operator $\circ$.

Chapter 4

MPI Compendium

In this chapter, we go through the MPI operations that will be used and implemented in this script. We explain the rationale for and use of the operations, point out limitations and peculiarities, and occasionally discuss things that might have been done better.

We discuss all relevant MPI operations by their C interface as defined in the `mpi.h` header file.

4.1 Point-to-point Message Passing

Listing 4.1: Blocking MPI send operations.

```
int MPI_Send(const void *buf, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm);
int MPI_Ssend(const void *buf, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm);
int MPI_Rsend(const void *buf, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm);
```

Listing 4.2: Non-blocking MPI send operations.

```
int MPI_ISend(const void *buf, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm, MPI_Request *request);
int MPI_ISsend(const void *buf, int count, MPI_Datatype datatype,
               int dest, int tag, MPI_Comm comm, MPI_Request *request);
int MPI_IRsend(const void *buf, int count, MPI_Datatype datatype,
               int dest, int tag, MPI_Comm comm, MPI_Request *request);
```

Listing 4.3: Blocking and non-blocking MPI receive operations.

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
             int source, int tag, MPI_Comm comm, MPI_Status *status);
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
              int source, int tag, MPI_Comm comm, MPI_Request *request);
```

Listing 4.4: Completing and testing non-blocking MPI operations.

```
int MPI_Wait(MPI_Request *request, MPI_Status *status);
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status);
```

```
int MPI_Waitany(int count, MPI_Request array_of_requests[],
                int *indx, MPI_Status *status);
int MPI_Testany(int count, MPI_Request array_of_requests[],
                int *indx, int *flag, MPI_Status *status);
int MPI_Waitall(int count, MPI_Request array_of_requests[],
                MPI_Status array_of_statuses[]);
int MPI_Testall(int count, MPI_Request array_of_requests[],
                int *flag, MPI_Status array_of_statuses[]);
int MPI_Waitsome(int incount, MPI_Request array_of_requests[],
                 int *outcount, int array_of_indices[],
                 MPI_Status array_of_statuses[]);
int MPI_Testsome(int incount, MPI_Request array_of_requests[],
                 int *outcount, int array_of_indices[],
                 MPI_Status array_of_statuses[]);

int MPI_Request_free(MPI_Request *request);
int MPI_Probe(int source, int tag, MPI_Comm comm,
              MPI_Status *status);
int MPI_Iprobe(int source, int tag, MPI_Comm comm,
               int *flag, MPI_Status *status);
```

Listing 4.5: Getting counts and elements.

```
int MPI_Get_count(const MPI_Status *status, MPI_Datatype datatype,
                  int *count);
int MPI_Get_elements(const MPI_Status *status, MPI_Datatype datatype,
                     int *count);
```

4.2 (Dense) Collective Operations

Listing 4.6: Standard, dense blocking collective operations.

```
int MPI_Barrier(MPI_Comm comm);
int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype,
              int root, MPI_Comm comm);
int MPI_Gather(const void *sendbuf,
               int sendcount, MPI_Datatype sendtype,
               void *recvbuf, int recvcount, MPI_Datatype recvtype,
               int root, MPI_Comm comm);
int MPI_Gatherv(const void *sendbuf,
                int sendcount, MPI_Datatype sendtype,
                void *recvbuf,
                const int *recvcounts, const int *displs,
                MPI_Datatype recvtype,
                int root, MPI_Comm comm);
int MPI_Scatter(const void *sendbuf,
                int sendcount, MPI_Datatype sendtype,
                void *recvbuf, int recvcount, MPI_Datatype recvtype,
                int root, MPI_Comm comm);
int MPI_Scatterv(const void *sendbuf,
                 const int *sendcounts, const int *displs,
                 MPI_Datatype sendtype,
                 void *recvbuf, int recvcount, MPI_Datatype recvtype,
```

```
                  int root, MPI_Comm comm);
int MPI_Allgather(const void *sendbuf,
                  int sendcount, MPI_Datatype sendtype,
                  void *recvbuf, int recvcount, MPI_Datatype recvtype,
                  MPI_Comm comm);
int MPI_Allgatherv(const void *sendbuf,
                   int sendcount, MPI_Datatype sendtype,
                   void *recvbuf,
                   const int *recvcounts, const int *displs,
                   MPI_Datatype recvtype, MPI_Comm comm);
int MPI_Alltoall(const void *sendbuf,
                 int sendcount, MPI_Datatype sendtype,
                 void *recvbuf, int recvcount, MPI_Datatype recvtype,
                 MPI_Comm comm);
int MPI_Alltoallv(const void *sendbuf,
                  const int *sendcounts, const int *sdispls,
                  MPI_Datatype sendtype,
                  void *recvbuf,
                  const int *recvcounts, const int *rdispls,
                  MPI_Datatype recvtype,
                  MPI_Comm comm);
int MPI_Alltoallw(const void *sendbuf,
                  const int sendcounts[], const int sdispls[],
                  const MPI_Datatype sendtypes[],
                  void *recvbuf,
                  const int recvcounts[], const int rdispls[],
                  const MPI_Datatype recvtypes[],
                  MPI_Comm comm);
int MPI_Reduce(const void *sendbuf, void *recvbuf,
               int count, MPI_Datatype datatype,
               MPI_Op op, int root, MPI_Comm comm);
int MPI_Allreduce(const void *sendbuf, void *recvbuf,
                  int count, MPI_Datatype datatype,
                  MPI_Op op, MPI_Comm comm);
int MPI_Reduce_scatter_block(const void *sendbuf, void *recvbuf,
                             int recvcount, MPI_Datatype datatype,
                             MPI_Op op, MPI_Comm comm);
int MPI_Reduce_scatter(const void *sendbuf, void *recvbuf,
                       const int recvcounts[], MPI_Datatype datatype,
                       MPI_Op op, MPI_Comm comm);
int MPI_Scan(const void *sendbuf, void *recvbuf,
             int count, MPI_Datatype datatype,
             MPI_Op op, MPI_Comm comm);
int MPI_Exscan(const void *sendbuf, void *recvbuf,
               int count, MPI_Datatype datatype,
               MPI_Op op, MPI_Comm comm);

int MPI_Op_create(MPI_User_function *user_fn, int commute, MPI_Op *op);
int MPI_Op_free(MPI_Op *op);
```

Listing 4.7: Standard, binary reduction operators with assignment.

```
MPI_MAX
MPI_MIN
MPI_SUM
MPI_PROD
```

```
MPI_LAND
MPI_BAND
MPI_LOR
MPI_BOR
MPI_LXOR
MPI_BXOR
MPI_MINLOC
MPI_MAXLOC
MPI_REPLACE
MPI_NO_OP
```

Non-blocking collectives.

Listing 4.8: The dense, non-blocking collective operations.

```
int MPI_Ibarrier(MPI_Comm comm, MPI_Request *request);
int MPI_Ibcast(void *buffer, int count, MPI_Datatype datatype, int root,
               MPI_Comm comm, MPI_Request *request);
int MPI_Igather(const void *sendbuf,
                int sendcount, MPI_Datatype sendtype,
                void *recvbuf,
                int recvcount, MPI_Datatype recvtype,
                int root, MPI_Comm comm, MPI_Request *request);
int MPI_Igatherv(const void *sendbuf,
                 int sendcount, MPI_Datatype sendtype,
                 void *recvbuf,
                 const int *recvcounts, const int *displs,
                 MPI_Datatype recvtype, int root,
                 MPI_Comm comm, MPI_Request *request);
int MPI_Iscatter(const void *sendbuf,
                 int sendcount, MPI_Datatype sendtype,
                 void *recvbuf, int recvcount, MPI_Datatype recvtype,
                 int root, MPI_Comm comm);
int MPI_Iscatterv(const void *sendbuf,
                  const int *sendcounts, const int *displs,
                  MPI_Datatype sendtype,
                  void *recvbuf, int recvcount, MPI_Datatype recvtype,
                  int root, MPI_Comm comm, MPI_Request *request);
int MPI_Iallgather(const void *sendbuf,
                   int sendcount, MPI_Datatype sendtype,
                   void *recvbuf,
                   int recvcount, MPI_Datatype recvtype,
                   MPI_Comm comm, MPI_Request *request);
int MPI_Iallgatherv(const void *sendbuf,
                    int sendcount, MPI_Datatype sendtype,
                    void *recvbuf,
                    const int *recvcounts, const int *displs,
                    MPI_Datatype recvtype,
                    MPI_Comm comm, MPI_Request *request);
int MPI_Ialltoall(const void *sendbuf,
                  int sendcount, MPI_Datatype sendtype,
                  void *recvbuf,
                  int recvcount, MPI_Datatype recvtype,
                  MPI_Comm comm, MPI_Request *request);
int MPI_Ialltoallv(const void *sendbuf,
                   const int *sendcounts, const int *sdispls,
                   MPI_Datatype sendtype,
```

```
                    void *recvbuf,
                    const int *recvcounts, const int *rdispls,
                    MPI_Datatype recvtype,
                    MPI_Comm comm, MPI_Request *request);
int MPI_Ialltoallw(const void *sendbuf,
                    const int sendcounts[], const int sdispls[],
                    const MPI_Datatype sendtypes[],
                    void *recvbuf,
                    const int recvcounts[], const int rdispls[],
                    const MPI_Datatype recvtypes[],
                    MPI_Comm comm, MPI_Request *request);
int MPI_Ireduce(const void *sendbuf,
                 void *recvbuf, int count, MPI_Datatype datatype,
                 MPI_Op op, int root,
                 MPI_Comm comm, MPI_Request *request);
int MPI_Iallreduce(const void *sendbuf,
                    void *recvbuf, int count, MPI_Datatype datatype,
                    MPI_Op op, MPI_Comm comm, MPI_Request *request);
int MPI_Ireduce_scatter_block(const void *sendbuf,
                              void *recvbuf, int recvcount,
                              MPI_Datatype datatype, MPI_Op op,
                              MPI_Comm comm, MPI_Request *request);
int MPI_Ireduce_scatter(const void *sendbuf, void *recvbuf,
                        const int recvcounts[],
                        MPI_Datatype datatype, MPI_Op op,
                        MPI_Comm comm, MPI_Request *request);
int MPI_Iscan(const void *sendbuf, void *recvbuf,
              int count, MPI_Datatype datatype, MPI_Op op,
              MPI_Comm comm, MPI_Request *request);
int MPI_Iexscan(const void *sendbuf, void *recvbuf,
                int count, MPI_Datatype datatype, MPI_Op op,
                MPI_Comm comm, MPI_Request *request);
```

4.3 Data layouts: MPI Derived Datatypes

4.4 Topologies and Communicators

Listing 4.9: Splitting communicators.

```
int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm);
int MPI_Comm_dup_with_info(MPI_Comm comm, MPI_Info info,
                           MPI_Comm *newcomm);
int MPI_Comm_create(MPI_Comm comm, MPI_Group group,
                    MPI_Comm *newcomm);
int MPI_Comm_split(MPI_Comm comm, int color, int key,
                   MPI_Comm *newcomm);
int MPI_Comm_split_type(MPI_Comm comm, int split_type, int key,
                        MPI_Info info, MPI_Comm *newcomm);

int MPI_Comm_free(MPI_Comm *comm);
```

4.5 Sparse Collective Operations

Listing 4.10: The sparse, blocking collective operations.

```
int MPI_Neighbor_allgather(const void *sendbuf,
                           int sendcount, MPI_Datatype sendtype,
                           void *recvbuf,
                           int recvcount, MPI_Datatype recvtype,
                           MPI_Comm comm);
int MPI_Neighbor_allgatherv(const void *sendbuf,
                            int sendcount, MPI_Datatype sendtype,
                            void *recvbuf,
                            const int recvcounts[], const int displs[],
                            MPI_Datatype recvtype, MPI_Comm comm);
int MPI_Neighbor_alltoall(const void *sendbuf,
                          int sendcount, MPI_Datatype sendtype,
                          void *recvbuf,
                          int recvcount, MPI_Datatype recvtype,
                          MPI_Comm comm);
int MPI_Neighbor_alltoallv(const void *sendbuf,
                           const int sendcounts[], const int sdispls[],
                           MPI_Datatype sendtype,
                           void *recvbuf,
                           const int recvcounts[], const int rdispls[],
                           MPI_Datatype recvtype,
                           MPI_Comm comm);
int MPI_Neighbor_alltoallw(const void *sendbuf,
                           const int sendcounts[],
                           const MPI_Aint sdispls[],
                           const MPI_Datatype sendtypes[],
                           void *recvbuf,
                           const int recvcounts[],
                           const MPI_Aint rdispls[],
                           const MPI_Datatype recvtypes[],
                           MPI_Comm comm);
```

4.6 Chapter notes

Forerunners: [BBC+95], [FJL+88] Using MPI books: [GLS94, GLT99, GHTL14]

Chapter 5

Graph Theoretic Lower Bounds

In this chapter, we examine some of the basic, well-known (*information theoretic* and *bisection-width*) bounds on the number of required dependent communication rounds for various collective operations for specific communication structures. Combining this with a communication cost model like the linear transmission cost model (Section 2.6.1), lower bounds on the time complexity for many operations can be given. With these bounds as baseline we can judge what has been achieved by the algorithms to be shown in the following many chapters. We also present and prove some of the basic hardness results for operations on arbitrary communication structures that are not know a priori, but given as part of the input.

5.1 Broadcast bounds

The information theoretic (information dissimination) lower bound.

5.2 Alltoall bounds

The al-to-all communication operations, all processes have to send a data block to all other processes. Let the processes be the nodes of a graph $G = (V, E)$. Each process $u \in V$ thus has a data block to each other $v \neq u$ in G (and possibly to itself). The total number of data blocks is thus $p(p-1)$.

Let (U, V) be a cut of G of roughly even size, $|U| = |V|$ or $|U| = |V| + 1 = \lceil p/2 \rceil$. The number of data blocks from processes $u \in U$ to processes $v \in V$ is

Bisection width bounds. Bisection width of the communication structures.

5.3 Tradeoffs

5.4 Hardness results

Optimal (smallest number of rounds) broadcast in arbitrary communication structures G with one-ported, uni-directional communication is NP-complete.

5.5 Chapter notes

[GJ79] [SCH81] [HHL88]

5.6 Exercises

More theoretically oriented exercises.

Chapter 6

Algorithms on Stars, Linear Arrays and Rings

This chapter presents algorithms that have linear-round communication complexity in the number of processors p due to the (restricted) structure and power of the underlying communication structures. We consider here stars, linear arrays, and rings. Nevertheless, since such structures can be seen (and will be used) as building blocks for larger, more powerful communication structures, like meshes, tori, hypercubes, and k-flies (defined as Cartesian graph products, for instance), studying algorithms on these structures is therefore valuable. In particular, we show how *linear pipelining* can be used to implement algorithms that are in some sense optimal for some of the standard collective operations. Since the (directed) degree of these networks is only one, communication with more than one communication port is not meaningful.

6.1 Star Networks

The simple star network (see Definition 2.3) is a useful (sometimes implicit) structure for giving operational definitions of rooted, asymmetric collectives operations like broadcast, gather/scatter, and reduction-to-root. This observation is often exploited in the MPI specification [MPI15, Chapter 5].

Algorithm 6 Algorithm for processor $i \in V(S_p)$ for broadcasting in a rooted star graph S_p of p processors from root processor $r \in V(S_p)$ with buffer $B = [m]$ of m elements.

```
1: if i = r then
2:     for all j = 0, 1, ..., p − 1, j ≠ r do Send(B, j, S_p)
3:     end for
4: else
5:     Recv(B, r, S_p)
6: end if
```

A linear-round algorithm for the Broadcast operation on a star graph S_p with the canonical numbering $V = \{0, 1, \ldots, r, \ldots, p-1\}$ rooted at processor $r \in V$ is shown as Algorithm 6. Initially, the root processor r has data, here in a buffer (array) $B = [m]$ of m elements, to be broadcast to the other processors $i \in V$. The root sends the

contents of B to the other $V \setminus \{r\}$ processors in $p-1$ communication rounds in some order (indicated by the **for all** iteration). Clearly, this satisfies the specification of the broadcast operation, and each processor will after the operation have the m data elements stored in its local buffer $B = [m]$. For the algorithm to work correctly, it is tacitly assumed that all processors know the number of elements m to be broadcast. This is the case with the `MPI_Bcast(buffer,count,datatype,root,comm)` interface, where all MPI processes are required to call with the same number elements (specified by the `count` and `datatype` arguments); likewise, all processes are required to give the same value for the `root` argument (a rank between 0 and the size of the communicator `comm`). Note that the results are placed in local data structures B at the processors with the only requirement that B is some linearly ordered data structure of m elements. In MPI, the `buffer` argument addresses the local buffer at each calling process, but the exact layout and access pattern can be different for different processes as specified locally by the `datatype` arguments.

Algorithm 7 Algorithm for processor $i \in V(S_p)$ for the Scatter operation in star graph S_p with root processor $r \in V(S_p)$ from a buffer $B = [m_0 m_1 \cdots m_r \cdots m_{p-1}]$ of $m_0 + m_1 + \ldots + m_r + \ldots m_{p-1}$ elements into processor local buffers $B = [m_i]$.

```
1: if i = r then
2:     for all j = 0, 1, ..., p − 1, j ≠ r do
3:         s ← Σ_{k=0}^{j−1} m_k
4:         Send(B[s : s + m_j − 1], j, S_p)
5:     end for
6: else
7:     Recv(B, r, S_p)
8: end if
```

A linear-round algorithm for the Scatter operation is shown as Algorithm 7. The input buffer $B = [m_0 m_1 \cdots m_r \cdots m_{p-1}]$ at the root $r \in V$ has a data block for all other processors $i \in V$, including the root itself. In the algorithm shown, the $[m_r]$ block for the root is left where it is, or in other words already assumed to be *in place*. Therefore, $p-1$ communication rounds suffice and are also required to send the data blocks $[m_j]$ to the other processors. The send operations at the root processor can be done in any oder, as indicated by the **for all** loop, and therefore the start index s of the jth block is recomputed at each send operation. In a real implementation, such recomputation should be avoided (although here of little practical import). In MPI, in-place, non-communication for the root can in the `MPI_Scatter(sendbuf,...,recvbuf,...,root,comm)` and `MPI_Scatterv(sendbuf,...,recvbuf,...,root,comm)` operations be effected by using the `MPI_IN_PLACE` argument for the receive buffer at the root. For the algorithm to be correct, it is required for the root to know all block sizes m_j for $j \in V$, and for each non-root processor i to know its own block size m_i; but block sizes can be different. This is guaranteed and required for correct use of the irregular `MPI_Scatterv()` operation. The regular case where all block sizes are the same, $m_j = b$ for all $j \in V$, is covered by the `MPI_Scatter()` operation. A star graph, linear-round algorithm for the Gather operation is analogous.

Finally, a linear-round algorithm for reduction with an associative, but not necessarily

Algorithm 8 Algorithm for processor $i \in V(S_p)$ for reduction-to-root, Reduce, on star graph S_p with root r, buffer $B = [m]$ of m elements and result at root in buffer $R = [m]$.

```
 1: if i = r then
 2:     R ← B
 3:     for j = r − 1, ..., 0 do                    ▷ Empty if r = 0
 4:         Recv(B, j, S_p)
 5:         R ← B ∘ R
 6:     end for
 7:     for j = r + 1, ..., p − 1 do                ▷ Empty if r = p − 1
 8:         Recv(B, j, S_p)
 9:         R ← R ∘ B
10:     end for
11: else
12:     Send(B, r, S_p)
13: end if
```

commutative operator $\circ$ is shown as Algorithm 8. The algorithm computes $B = ((B_0 \circ B_1 \circ \cdots \circ B_r) \circ \cdots \circ B_{p-1})$ in rank order, with the brackets indicating the order in which the $\circ$ operations are performed, regardless of which processor is root. This is enforced by the ordered **for**-loops that receive first from the processors ranked lower than the root r, then from the processors ranked higher than the root processor. An algorithm that do not rely on commutativity is preferable to an algorithm that needs this assumption, also in practical settings where commutativity may indeed not hold. Likewise, operators may not have an identity element (or the identity element may not be know to the implementation), therefore algorithms are always formulated such that the identity is not needed, even when this entails some cumbersome corner cases (handling the first received buffer in a special way, etc.). Algorithm 8 can easily be turned into an implementation for the `MPI_Reduce(...,comm)` collective.

MPI implementation concerns: In a concrete implementation in and for MPI, performing the operator $B_i \circ B_{i+1}$ on two MPI buffers (with the same `count` and `datatype` arguments) is done with the `MPI_Reduce_local(in,inout,count,datatype,op)` function. This function takes an operator argument for an operator that is applied element wise on the `count` elements in the two buffers, and stores the result in the second argument buffer `inout`. With this MPI function, the reduction and assignment "from the left"

$R \leftarrow B \circ R$

can readily be performed, but not the assignment "from the right"

$R \leftarrow R \circ B$

This problem has to be handled by "reducing into B"

$B \leftarrow R \circ B$
$R \leftarrow B$

and then copying the results from B into R. This is potentially wasteful, especially for large buffers B and R. A flexible, three-argument reduction function is so far missing from the MPI standard. The predefined operators for MPI reductions were listed in Table 4.7.

Note that for none of these, an explicit neutral or identity element is given, even though all these operators mathematically do possess an identity.

6.2 Collectives on Linear Arrays and Rings

Despite similarly weak properties, more interesting algorithms than on star networks exist on linear arrays (Definition 2.1) and on rings (Definition 2.2) for a larger selection of also symmetric, non-rooted, standard collective operations. Since some of the algorithms given here can actually be useful in practice, stand-alone or as parts of other algorithms, worst case performance analysis will be given and summarized using the pessimistic assumption of a worst case communication linear time of $\alpha + \beta m$ to transfer a block of m data elements, as discussed in Section 2.6.1.

6.2.1 A natural linear array operation: Prefix-sums

Algorithm 9 Algorithm for processor $i \in V(L_p)$ for computing the inclusive prefix-sums on the linear array L_p for an associative (commutative or not) operation $\circ$. Each processor has a data block $S = [m]$ of m elements. Block $R = [m]$ is used to store the result.

```
1:  t ← (i + 1)_p                          ▷ To-processor for i
2:  f ← (i − 1)_p                          ▷ From-processor for i
3:  if i > 0 then
4:      Recv(R, f, L_p)
5:      R ← R ∘ S
6:  else
7:      R ← S
8:  end if
9:  if i < p − 1 then
10:     Send(R, t, L_p)
11: end if
```

The message-passing prefix-sums or scan problem is to compute, in some linear order of the processors, for each processor i the "sum" of contributions S of all processors before (and in- or excluding) processor i under some associative operation on the S contributions (whether the operation is commutative or not is not important here). A linear array over the p processors is obviously suited to this problem, and a linear-round inclusive prefix-sums (scan) algorithm is shown as Algorithm 9. An associative operation $\circ$ is given which reduces (combines) two blocks of data A and B with the same number of elements into a block $A \circ B$. The algorithm correctly computes for processor $i \in V(L_p)$ the inclusive prefix-sum $R = \bigcirc_{j=0}^{i} S_i$ for per processor input data blocks S_i stored in processor local data structures (arrays, vectors) S of m elements. In order to avoid a deadlock situation, the first processor $i = 0$ in the linear array does not receive data (it cannot!), and the last processor $i = p - 1$ does not send data (it cannot). The algorithm can easily be adopted to the exclusive prefix-sums problem (exercise).

Per definition of the prefix-sums problem, the last processor $p - 1$ in the linear array computes the sum of the contributions from all processors, including $p - 1$ itself, in

processor order. Therefore, Algorithm 9 can be used for the reduction-to-root operation Reduce when $r = p - 1$.

Proposition 6.1. *On the linear processor array L_p with p processors, the inclusive and exclusive prefix-sums problems with an associative, binary operation $\circ$ on m-element input data blocks can be solved in $p - 1$ dependent communication operations. In the linear transmission cost model with latency α and time per unit β, and assuming cost γ per unit of reduced data, the total time for the last processor to finish is $(p - 1)(\alpha + \beta m + \gamma m)$. Uni-directional communication capabilities are sufficient.*

MPI implementation concerns: Algorithm 9 can readily be turned into an implementation for `MPI_Scan(...,op,comm)`. It is similarly straightforward to give a linear array implementation for the exclusive prefix-sums operation `MPI_Exscan(...,op,comm)`. To apply the MPI reduction operator `op`, the `MPI_Reduce_local()` function is used; but which unfortunately cannot do the

$R \leftarrow R \circ S$

reduction "from the right", as discussed above. An intermediate copy and/or extra buffer is needed.

6.2.2 Broadcast, reduction, scatter and gather

Rooted collectives with arbitrary root cannot be realized on the directed, linear array alone. But they can be nicely on the (uni-directional) ring. We first discuss algorithms for broadcast, reduction and scatter/gather operations. Recall the notation $(i + 1)_p$ and $(i - 1)_p$ introduce in Section 2.2 which is convenient here for computing the predecessor and successor processors of processor $i \in V(R_p)$.

Algorithm 10 Algorithm for processor $i \in V(R_p)$ for broadcasting on ring R_p from root processor $r \in V(S_p)$ with buffer $B = [m]$ of m elements.

```
1: t ← (i + 1)_p                          ▷ To-processor for i
2: if i = r then
3:     Send(B, t, R_p)
4: else
5:     f ← (i − 1)_p                      ▷ From-processor for non-root i
6:     Recv(B, f, R_p)
7:     if t ≠ r then
8:         Send(B, t, R_p)
9:     end if
10: end if
```

A straightforward algorithm for the Broadcast operation of a buffer $B = [m]$ from an arbitrary root $r \in V(R_p)$ is shown as Algorithm 10. The block B is sent through the processors of the ring, starting with the root processor and ending at the processor just before the root.

Proposition 6.2. *On the directed ring R_p with p processors, the broadcast problem for input of m elements can be solved in $p - 1$ dependent communication operations. In the*

linear transmission cost model with latency α and time per unit β, the total time for the last processor to finish is $(p-1)(\alpha+\beta m)$. Uni-directional communication capabilities are sufficient.

This is a most trivial result; it is perhaps surprising that even on the weak ring communication structure it is possible to do much better and actually achieve the lower bound for broadcast communication in R_p up to a factor of (at most) 2.

Algorithm 11 Algorithm for processor $i \in V(R_p)$ for scattering on ring R_p from root processor r. The data blocks $[m_j]$ at the root of input $B = [m_0 \cdots m_r \cdots m_{p-1}]$ are indexed as $B[j]$, and each of size m_j. For each non-root processor $i \in V(R_p)$ the resulting block is stored as $B = [m_i]$.

```
1: t ← (i + 1)_p                                          ▷ To-processor for i
2: if i = r then
3:     for j = 1, ..., p − 1 do
4:         Send(B[(r − j)_p], t, R_p)
5:     end for
6: else                          ▷ Buffer B used for intermediate blocks and final result
7:     f ← (i − 1)_p                                      ▷ From-processor for non-root i
8:     Recv(B, f, R_p)
9:     for j = 1, ..., dist(i, r − 1) do
10:        D ← B
11:        Send(D, t, R_p) ‖ Recv(B, f, R_p)
12:    end for
13: end if
```

A linear-round algorithm for the Scatter operation is shown as Algorithm 11. The root processor r sends the blocks one after the other to its to-processor $t = (r+1)_r$ in the ring in "downwards" order: block $(r-1)_p$ followed by block $(r-2)_p$, etc.. Non-root processor i receives block $(r-1)_p$ in communication round $\text{dist}(r,i)-1$, and $\text{dist}(i,r-1)$ blocks after the first block; here, $\text{dist}(r,i)$ denotes the distance in the ring R_p from processor r to processor i. The last block received by processor i is thus block $(r-1-\text{dist}(i,r-1))_p = i$, as required by the specification of the Scatter operation. For this to be possible, both the root and all other processors must know the sizes m_i of many blocks. More precisely, processor i receives all blocks between processors i and $r-1$, and must know the sizes of these blocks. This is the case with the regular `MPI_Scatter(...,recvbuf,recvcount,datatype,root,comm)` operation (where all blocks have the same number of elements), but not for the irregular `MPI_Scatterv(...,comm)` operation. The algorithm cannot readily be used for this case.

Non-root processor i receives into buffer B which is for the next iteration copied into the send buffer D. This incurs an unnecessary local copy cost proportional to m_j for each of the received blocks (except the last). In a concrete implementation, this can easily be avoided by a *double buffering* scheme, where send and receive buffers are swapped after each operation. The first receive buffer should for processor i be chosen such that the last receive buffer is B, the buffer for the final result. This observation will become relevant for the alltoall algorithm in Section 10.1.3.

An algorithm for the Gather collective is analogous, and the same considerations apply.

Proposition 6.3. *On the directed ring R_p with p processors, the scatter (and gather) problem(s) for input (and output) blocks of m elements in total can be solved in $p-1$ dependent communication operations. For the regular problem(s), where all sub-blocks have the same size $m_j = m/p$, the total time in the linear transmission cost model with latency α and time per unit β for the last processor to finish is $(p-1)(\alpha+\beta m/p) = (p-1)\alpha+\frac{p-1}{p}\beta m$. Fully bi-directional communication capabilities are needed.*

This will be a key component of a much better algorithm for the Broadcast operation.

6.2.3 The symmetric Allgather operation

Algorithm 12 Algorithm for processor $i \in V(R_p)$ for the Allgather operation on ring R_p. The result data blocks $[m_j]$ at each processor $B = [m_0 \cdots m_j \cdots m_{p-1}]$ are indexed as $B[j]$, and each of size m_j. The block that processor i has initially is $B[i]$.

1: $t \leftarrow (i+1)_p$ ▷ To- and from-processors
2: $f \leftarrow (i-1)_p$
3: **for** $j = 1, \ldots, p-1$ **do**
4: Send$(B[(i+1-j)_p], t, R_p)$ ‖ Recv$(B[(i-j)_p], f, R_p)$
5: **end for**

A linear-round allgather algorithm is shown as Algorithm 12. In each round j each processor i sends an old block $i+1-j$ from the array B of blocks, and receives a new block $i-j$; $p-1$ such rounds for $j = 1, \ldots, p-1$ are required. The algorithm uses a fully bidirectional communication model: Each processor can (and do) simultaneously receive from a from-processor f and send to a to-processor t and usually $f \neq t$. The algorithm assumes that all processors know all block sizes $m_i, 0 \leq i < p$. This is indeed the case with the `MPI_Allgather(...,comm)` collective, which defines a regular collective with all m_i blocks having the same size. It is also the case for the irregular `MPI_Allgatherv(...,comm)` collective.

Proposition 6.4. *On the directed ring R_p with p processors, the allgather problem can be solved in $p-1$ dependent communication operations. For the regular problem(s), where per processor input and all sub-blocks have the same size $m_j = m/p$, the total time in the linear transmission cost model with latency α and time per unit β for the last processor to finish is $(p-1)(\alpha+\beta m/p) = (p-1)\alpha+\frac{p-1}{p}\beta m$. Fully bi-directional communication capabilities are needed.*

It is worthwhile noting that the allgather algorithm can be specialized to an algorithm for the Gather operation with root processor r by omitting any communication that is not needed for gathering all the blocks at processor r. This would essentially lead to the algorithm analogous to Algorithm 11 as discussed in Section 6.2.2. It is worth pointing out that in our model assumptions, Proposition 6.3 and Proposition 6.4 state the same running time for scatter/gather and allgather operations on the ring R_p.

Algorithm 13 Algorithm for processor $i \in V(R_p)$ for the ReduceAll operation on ring R_p with an associative, but not necessarily commutative operator $\circ$ with buffer $R = [m]$ of m elements.

```
1: t ← (i + 1)_p                                          ▷ To-processor
2: f ← (i − 1)_p                                          ▷ From-processor
3: T ← R
4: for j = 1, ..., i do
5:     Send(T, t, R_p) ‖ Recv(T', f, R_p)
6:     T ← T'
7:     R ← T ∘ R
8: end for
9: if i < p − 1 then
10:    Send(T, t, R_p) ‖ Recv(R', f, R_p)
11:    T ← R'
12:    for j = i + 2, ..., p − 1 do        ▷ One round already done, remaining p − i − 2
13:        Send(T, t, R_p) ‖ Recv(T', f, R_p)
14:        T ← T'
15:        R' ← T ∘ R'
16:    end for
17:    R ← R ∘ R'
18: end if
```

6.2.4 Ring algorithms for reduction

A linear-round algorithm for the reduction-to-all, ReduceAll, operation for non-commutative operators, similar to Algorithm 12 is shown as Algorithm 13. Each processor i contributes a buffer $B_i = [m]$ of the same number m of elements, which are reduced as $\bigcirc_{i=0}^{p-1} B_i$ in processor rank order. The algorithm works by sending the buffers around the ring in $p-1$ communication rounds in each of which each processor sends an old block from the previous round, and receives a new block. Each processor accumulates two partial results in buffers R and R'. In the first i communication rounds, processor i receives blocks from the processors $0, 1, \ldots i-1$. These blocks are accumulated in the buffer R, which after round i will store the sum $\bigcirc_{j=0}^{i} B_j$ (including the contribution from processor i itself). For the following $p-i$ rounds, the received blocks will be accumulated in the buffer R' which will after the last round store the sum $\bigcirc_{j=i+1}^{p-1} B_j$. At the end, the final result is computed as $R \leftarrow R \circ R'$.

The copying of the received block T' into T can be avoided by swapping the two intermediate buffers T and T' in each iteration, as described for Algorithm 11.

The algorithm for each processor computes the result by summing the blocks from the processors in rank order, and thus do not require or exploit commutativity of the binary $\circ$ operator. This is achieved at a high price, through: All processors compute the full sum $\bigcirc_{j=0}^{p-1} B_i$, and is in that sense maximally work-redundant. The work in computing the results of the $p-1$ applications of $\circ$ is in no way parallelized. For the better, faster reduction algorithms that will be discussed in the following chapters, this is improved, but often at the cost of requiring and exploiting commutativity of $\circ$. Ideally, the overhead for the reductions should amount to $O(\frac{p-1}{p}m)$ only, when reductions are fully parallelized.

As for Algorithm 12, also the reduction-to-all Algorithm 13 can easily be specialized to an algorithm for the rooted Reduce operation.

Proposition 6.5. *On the directed ring R_p with p processors, the reduction-to-all and the reduction-to-root problems on m element inputs can be solved in $p-1$ dependent communication operations. The total time in the linear transmission cost model with latency α and time per unit β and reduction time γ for the last processor to finish is $(p-1)(\alpha+\beta m+\gamma m)$. For the reduction-to-all problem, fully bi-directional communication capabilities are needed. For the reduction-to-root problem, uni-directional communication capabilities would suffice.*

6.2.5 An algorithm for the ReduceScatter collective

By dropping the non-commutativity condition that the sums have to be computed in rank order, interesting and better algorithms for various kinds of reductions can be given.

Algorithm 14 Algorithm for processor $i \in V(R_p)$ for reduce-scatter on the ring R_p. The input data blocks $[m_i]$ at each processor $S = [m_0 \cdots m_j \cdots m_{p-1}]$ are indexed as $S[i]$, each of size m_i. Processor i computes its result into R.

```
1: t ← (i + 1)_p                                   ▷ To- and from-processors
2: f ← (i − 1)_p
3: B ← S[(i − 1)_p]
4: for j = 1, ..., p − 1 do
5:     Send(B, t, R_p) ‖ Recv(R, f, R_p)
6:     B ← R ∘ S[(i − 1 − j)_p]
7: end for
8: R ← B
```

Algorithm 14 shows such an algorithm for the ReduceScatter collective operation for commutative operators. Per requirement, processor i computes $\bigcirc_{j=0}^{p-1} S_j[i]$ (in some, not necessarily rank order), where $S_j[i]$ denotes the ith input block $[m_i]$ for processor j. The invariant maintained by the algorithm is that after iteration $j, j = 1, \ldots, p-1$, processor i stores the sum $\bigcirc_{k=0}^{j} S_{(i-k)_p}[(i-1-j)_p]$ in B. This clearly holds after iteration $j = 1$, where processor $f = (i-1)_p$ has sent its input block $S_f[(f-1)_p]$ to processor i. Assuming the invariant holds, from-processor f will send in iteration $j > 1$ the sum $B = \bigcirc_{k=0}^{f} S_{(f-k)_p}[f-j]$ to processor i which can then reestablish the invariant. After the last iteration, B will store the full result for block $(i-1-(p-1))_p = i$. This is finally moved into the result block R. This could be avoided by letting the last iteration compute the result directly into R instead of into B.

In contrast to Algorithm 13, the computation of each partial result is distributed over the processors. The question is, whether it is possible to achieve this on a ring communication structure without exploiting commutativity? No such algorithm is known (and not possible?).

6.3 An unnatural Ring Algorithm: The **Alltoall** Collective

Linear arrays and rings are weak communication structures (in the sense of having low bisection width). They are therefore not efficient for communication intensive collective operations like alltoall. Nevertheless, it is instructive to see how alltoall can be done.

Algorithm 15 Algorithm for processor $i \in V(R_p)$ for alltoall on the ring R_p. Each processor i initially has $S = [m_i^0 \cdots m_i^j \cdots m_i^{p-1}]$ blocks that are indexed as $S[j]$, each of size m_i^j. The resulting blocks are going to be received in $R = [m_0^i \cdots m_j^i \cdots m_{p-1}^i]$ indexed as $R[j]$. For communication, arrays B^{in} and B^{out} are used with enough space to store the blocks that are received and sent.

```
1: t ← (i + 1)_p                                   ▷ To- and from-processors
2: f ← (i − 1)_p
3: R[i] ← S[i]                  ▷ Own block for processor i, copy from S to R buffer
4: for k = 1, ..., p − 1 do B^out[k − 1] ← S[(i + k)_p]
5: end for
6: for j = 1, ..., p − 1 do          ▷ Send and receive p − j blocks from B^out into B^in
7:     Send(B^out, t, R_p) || Recv(B^in, f, R_p)
8:     R[(i − j)_p] ← B^in[0]
9:     for k = 1, ..., p − 1 − j do B^out[k − 1] ← B^in[k]
10:    end for
11: end for
```

An algorithm for performing the alltoall operation an a ring R_p is given as Algorithm 15. Processor i first copies the block for itself from send buffer $S[i]$ to result buffer $R[i]$. The processors then symmetrically go through $p-1$ communication rounds.

In a real implementation with and for MPI, the shifting down of blocks and copying from in-buffer to out buffer does not necessarily have to be done, and should be avoided as far as possible.

The number of blocks sent and received per processor is $\sum_{k=1}^{p-1} p - k = p(p-1) - \frac{p(p-1)}{2} = \frac{p(p-1)}{2}$. The algorithm employs what is sometimes called *message combining*: Several blocks destined for different processors are collected and sent together. This idea saves communication operations and thus latency. We will see it in use many times later. The ring alltoall algorithm anticipates the algorithm using a circulant graph communication structure of Bruck *et al.* [BHK+97], see also [TRH14] that will be described in Section 10.1.3.

Proposition 6.6. *On the directed ring R_p with fully bidirectional communication capabilities, the alltoall problem can be solved in $p-1$ dependent communication rounds. In the linear transmission cost model with latency α and cost per unit β, the regular alltoall problem can be solved in $(p-1)\alpha + \frac{p(p-1)}{2}\beta m/p = (p-1)\alpha + \frac{(p-1)}{2}\beta m = (p-1)(\alpha + \beta m/2)$ time units for total problem size m and block size m/p units.*

6.4 Broadcast and Reduction with Pipelining

Rings and linear arrays can for the symmetric, rooted collectives, be used better, by which is meant that the processors can be kept busy with useful work for more of the time.

Algorithm 16 Pipelined algorithm for processor $i \in V(R_p)$ for broadcasting on ring R_p from root r. The m data elements are divided into n (roughly even sized) blocks of elements that are broadcast one after the other. The jth block is denoted $B[j]$.

```
1: t ← (i + 1)_p                                   ▷ To- and from-processor
2: f ← (i − 1)_p
3: if i = r then
4:     for j = 0, 1, ..., n − 1 do
5:         Send(B[j], t, R_p)
6:     end for
7: else if t = r then                              ▷ Last processor before root
8:     for j = 0, 1, ..., n − 1 do
9:         Recv(B[j], f, R_p)
10:    end for
11: else
12:    Recv(B[0], f, R_p)
13:    for j = 1, ..., n − 1 do
14:        Send(B[j − 1], t, R_p) || Recv(B[j], f, R_p)
15:    end for
16:    Send(B[n − 1], t, R_p)
17: end if
```

The linear array, pipelined broadcast algorithm is given more explicitly as Algorithm 16. The algorithm employs fully bidirectional communication, and it is difficult to do without this capability.

The following theorem is useful for analyzing pipelined algorithms of this kind.

Theorem 6.1 (Pipelining Lemma)**.** *Consider a processor in a pipelined algorithm in which data of m elements divided into blocks of roughly equal size are to be received. Let the algorithm for the processor have a* delay *(latency) of d dependent communication operations (each transmitting a block somewhere) before receiving the first block and a steady-state* rate *(frequency) of one new block for each sth communication operation (by doing something else or by waiting for the sending processor) with the delay d being larger than the rate s, $d > s$.*

Assuming a maximum time cost per communication operation of $\alpha + \beta b$ for a(ny) block of size b and thus a delay of $d(\alpha + \beta b)$, the time for processor i to complete the reception of the m elements and finish is

$$T_i(m) \quad = \quad (d-s)\alpha + 2\sqrt{s(d-s)\alpha\beta m} + s\beta m$$

Proof. Assume the m data elements are divided into n blocks of roughly $b = m/n$ elements. The first of the n blocks is received after the delay of d communication operations, and the remaining $n-1$ blocks at a rate of s communication operations per each new block. This gives a total time of

$$\begin{aligned} T_i(m) &= d(\alpha + \beta m/n) + (n-1)s(\alpha + \beta m/n) \\ &= d(\alpha + \beta m/n) - s(\alpha + \beta m/n) + sn\alpha + s\beta m \\ &= (d-s)(\alpha + \beta m/n) + sn\alpha + s\beta m \\ &= (d-s)\alpha + (d-s)\beta m/n + sn\alpha + s\beta m \quad . \end{aligned}$$

The first and the last term are independent of the number of blocks n. The term $sn\alpha$ is increasing with the number of blocks while the term $(d-s)\beta m/n$ is decreasing with the number of blocks (since per assumption $d > s$). The minimum time can be found now by balancing these two terms and solving for n, which gives a running time for n blocks of $T_i(m) = (d-s)\alpha + 2sn\alpha + s\beta m$.

$$\begin{aligned} (d-s)\beta m/n = sn\alpha &\Leftrightarrow \\ n^2 = \frac{(d-s)\beta m}{s\alpha} &\Leftrightarrow \\ n = \sqrt{\frac{(d-s)\beta m}{s\alpha}} \quad . \end{aligned}$$

Since now $2sn\alpha = 2\sqrt{s(d-s)\alpha\beta}$ the claim follows. □

The proof of the pipelining lemma tells that to yield the minimum running communication time,

- the best number of blocks is $n = \sqrt{\frac{(d-s)\beta m}{s\alpha}}$, and
- the best block size is $b = m/n = \sqrt{\frac{sm\alpha}{(d-s)\beta}}$.

The proof also stresses the following fact.

Alternatively, use calculus, differentiate the expression for the time, and solve $T_i'(m) = 0$.

The pipelining lemma is used here by looking at the slowest processor, the one with the largest delay. For the pipelined ring broadcast algorithm, and all the other ones, the delay is $d = p - 1$ and the rate $s = 1$.

Proposition 6.7. *On the directed ring R_p with p processors, the broadcast problem for input of m elements subdivided into n blocks can be solved in $p - 1 + n - 1$ dependent communication operations. In the linear transmission cost model with latency α and time per unit β, and assuming the m elements are divided into even-sized blocks of m/n elements, the total time for the last processor to finish is $(p - 1 + n - 1)(\alpha + \beta m/n)$. Uni-directional communication capabilities are sufficient.*

Using the pipelining lemma, gives the following strong result.

Lemma 6.1. *On the directed ring R_p with p processors, and assuming a linear transmission cost communication model with latency α and cost per unit β, the broadcast problem for input of m elements can be solved in time $(p-2)\alpha + 2\sqrt{(p-2)\alpha\beta m} + \beta m$.*

Algorithm 17 Pipelined algorithm for processor $i \in V(R_p)$ for the Reduce operation on ring R_p to root r with a binary, commutative, associative operator $\circ$. The m data elements are divided into n (roughly even sized) blocks of element that are reduced one after the other. The jth block is denoted $B[j]$.

1: $t \leftarrow (i+1)_p$ ▷ To- and from-processor
2: $f \leftarrow (i-1)_p$
3: **if** $i = r$ **then**
4: **for** $j = 0, 1, \ldots, n-1$ **do**
5: $\mathsf{Recv}(T, f, R_p)$
6: $B[j] \leftarrow T \circ B[j]$
7: **end for**
8: **else if** $f = r$ **then** ▷ First processor after root
9: **for** $j = 0, 1, \ldots, n-1$ **do**
10: $\mathsf{Send}(B[j], t, R_p)$
11: **end for**
12: **else**
13: $\mathsf{Recv}(T, f, R_p)$
14: $B[0] \leftarrow T \circ B[0]$
15: **for** $j = 1, \ldots, n-1$ **do**
16: $\mathsf{Send}(B[j-1], t, R_p) \parallel \mathsf{Recv}(T, f, R_p)$
17: $B[j] \leftarrow T \circ B[j]$
18: **end for**
19: $\mathsf{Send}(B[n-1], t, R_p)$
20: **end if**

The important property of the algorithm is that a strictly linear bandwidth term βm is achieved, meeting the lower bound for the broadcast problem.

Algorithm 17 modifies the pipelined broadcast Algorithm 16 to perform a reduction-to-root operation. This algorithm exploits commutativity, and for each block $B[j]$ being reduced computes the result as $\bigcirc_{i=r+1}^{p-1} B_i[j] \circ \bigcirc_{i=0}^{r} B_i[j]$. Algorithm 17 can easily be modified into algorithms for both the exclusive and inclusive scan operations.

6.4.1 A partially pipelined algorithm for the Allgather operation

Algorithm 18 shows how Algorithm 12 can be improved for the irregular case where the m_i block sizes may be different by introducing pipelining. This idea is from [TRS+10].

6.5 Chapter notes

The star algorithms are trivial, and folklore. They are used often in the MPI standard to explain the semantics of the corresponding collective operations, and sometimes also for concrete implementations.

Linear array and ring algorithms for some of the standard collectives are treated in [CHPvdG07] and used as building blocks.

The pipelined, irregular allgather algorithm is from [TRS+10].

6.6 Exercises

1. Implement a ring algorithm for the Gather operation similar to Algorithm 11. Implement the double buffering scheme. Does this give a measurable performance improvement?

Algorithm 18 Pipelined algorithm for processor $i \in V(R_p)$ for allgather on the ring R_p. The result data blocks $[m_j]$ at each processor $B = [m_0 \cdots m_j \cdots m_{p-1}]$ are indexed as $B[j] = [m_j]$, and each of size m_j. It is assumed that for at least one j, $m_j > 0$. The block that processor i has initially is $B[i]$, and a consecutive segment of c elements from offset o of this block is $B[i][o : o + c - 1]$. The precomputed, fixed pipeline block size is M.

1: $t, f \leftarrow (i+1)_p, (i-1)_p$ ▷ To- and from-processors
2: $r, s \leftarrow f, i$ ▷ Receive and send block indices
3: **if** $m_s = 0$ **then** ▷ Nothing to send initially, first receive a block
4: **repeat** $s \leftarrow (s-1)_p$
5: **until** $m_s > 0$
6: $o_s, c_s \leftarrow 0, \min(M, m_s)$
7: $\mathsf{Recv}(B[s][o_s : o_s + c_s - 1], f, R_p)$
8: $o_r \leftarrow o_s + c_s$
9: **else**
10: $o_s, o_r \leftarrow 0, 0$
11: **end if**
12: **while** $o_r = m_r$ **do** ▷ Next block to receive a segment from
13: $r, o_r \leftarrow (r-1)_p, 0$
14: **end while**
15: **while** $r \neq i \wedge s \neq t$ **do** ▷ Send and receive segments from blocks s and r
16: $c_s, c_r \leftarrow \min(M, m_s - o_s), \min(M, m_r - o_r)$
17: $\mathsf{Send}(B[s][o_s : o_s + c_s - 1], t, R_p) \parallel \mathsf{Recv}(B[r][o_r : o_r + c_r - 1], f, R_p)$
18: $o_s, o_r \leftarrow o_s + c_s, o_r + c_r$
19: **while** $o_s = m_s$ **do** ▷ Done with block s
20: $s, o_s \leftarrow (s-1)_p, 0$
21: **end while**
22: **while** $o_r = m_r$ **do** ▷ Done with block r
23: $r, o_r \leftarrow (r-1)_p, 0$
24: **end while**
25: **end while**
26: **while** $r \neq i$ **do**
27: $c_r \leftarrow \min(M, m_r - o_r)$
28: $\mathsf{Recv}(B[r][o_r : o_r + c_r - 1], f, R_p)$
29: $o_r \leftarrow o_r + c_r$
30: **while** $o_r = m_r$ **do**
31: $r, o_r \leftarrow (r-1)_p, 0$
32: **end while**
33: **end while**
34: **while** $s \neq t$ **do**
35: $c_s \leftarrow \min(M, m_s - o_s)$
36: $\mathsf{Send}(B[s][o_s : o_s + c_s - 1], t, R_p)$
37: $o_s \leftarrow o_s + c_s$
38: **while** $o_s = m_s$ **do**
39: $s, o_s \leftarrow (s-1)_p, 0$
40: **end while**
41: **end while**

Chapter 7

Linear-round Algorithms on Fully Connected Networks

This chapter answers some of the open questions of the last chapter by giving (straightforward) algorithms for some of the symmetric collective operations utilizing fully connected networks (Definition 2.14). It also contains a classic result for the existence of socalled one-factors in complete graphs (fully connected communication structures).

7.1 The ReduceScatter Operation

The simple (but still interesting) algorithm in Section 6.2.5 for the ReduceScatter collective on p processor directed ring communication structures relied on commutativity of the supplied associative operator $\circ$ on the data blocks. In order to perform the reductions always in linear (processor) order and thus not use commutativity, more power of the communication structure seems to be needed.

Algorithm 19 shows an algorithm for non-commutative operators on the fully connected graph K_p. The associative, non-commutative operator is $\circ$. For each processor i, it maintains two running "sums" that are computed in rank order, one for the input blocks from processors $0, \ldots, i$ (which is kept in the output buffer R), and one for the input blocks $i+1, \ldots p-1$ (which is kept in intermediate buffer B). In each round, each processor receives a new input block. For the first i rounds, these blocks are added to R, and for the remaining $p-1-i$ rounds, the blocks are added into B. Finally, the result R is formed as $R \circ B = (\bigcirc_{j=0}^{i} S_j[i]) \circ (\bigcirc_{j=i+1}^{p-1} S_j[i])$ where S_j denotes the input buffer for processor j. The same trick with two running sums of this form was used in Algorithm 13 for the ReduceAll operation on processor rings.

7.2 The Alltoall Operation

The Alltoall operation, in which each processor has an individual, "personalized" data block for each other processor (including, sometimes, the processor itself) is easy to explain and implement with a fully connected communication structure. The problem here is rather to schedule the communication operations and solve the problem in a small number of commutation rounds where all processors are actively communicating, and no processors

Algorithm 19 Algorithm for processor $i \in V(K_p)$ for ReduceScatter operation on the fully connected communication structure K_p. The input data blocks $[m_j]$ at each processor $S = [m_0 \cdots m_j \cdots m_{p-1}]$ are indexed as $S[j]$, each of size m_j. Processor i computes its result into R.

1: $R \leftarrow S[i]$
2: **for** $j = 1, \ldots, i$ **do**
3: $\quad t \leftarrow (i + j)_p$ ▷ To- and from-processors
4: $\quad f \leftarrow (i - j)_p$
5: $\quad \mathsf{Send}(S[t], t, R_p) \parallel \mathsf{Recv}(B, f, K_p)$
6: $\quad R \leftarrow B \circ R$
7: **end for**
8: **if** $i < p - 1$ **then**
9: $\quad \mathsf{Send}(S[t], t, R_p) \parallel \mathsf{Recv}(B, f, K_p)$
10: $\quad$ **for** $j = i + 2, \ldots, p - 1$ **do**
11: $\quad\quad t \leftarrow (i + j)_p$ ▷ To- and from-processors
12: $\quad\quad f \leftarrow (i - j)_p$
13: $\quad\quad \mathsf{Send}(S[t], t, R_p) \parallel \mathsf{Recv}(B', f, K_p)$
14: $\quad\quad B \leftarrow B' \circ B$
15: $\quad$ **end for**
16: $\quad R \leftarrow R \circ B$
17: **end if**

have idle rounds where they wait for other processors to complete other communication operations. We present the classic, linear communication round algorithms in this section.

7.2.1 Fully Bidirectional Communication

With fully bidirectional communication capabilities, each processor can trivially in a communication round send a block to a destination (to-)processor, and receive a result block from a source (from-)processor. A total of $p - 1$ communication rounds will be required, with possibly an extra round for copying (transmitting) an own block from each processor to itself.

Algorithm 20 Algorithm for processor $i \in V(K_p)$ for alltoall in a fully connected network K_p. Each processor i initially has $S = [m_i^0 \cdots m_i^j \cdots m_i^{p-1}]$ blocks that are indexed as $S[j]$, each of size m_i^j. The resulting blocks are going to be received in $R = [m_0^i \cdots m_j^i \cdots m_{p-1}^i]$ indexed as $R[j]$.

1: $R[i] \leftarrow S[i]$ ▷ Copy own block
2: **for** $j = 1, \ldots, p - 1$ **do**
3: $\quad t \leftarrow (i + j)_p$ ▷ To- and from-processors
4: $\quad f \leftarrow (i - j)_p$
5: $\quad \mathsf{Send}(S[t], t, K_p) \parallel \mathsf{Recv}(R[f], f, K_p)$
6: **end for**

Algorithm 20 is straight-forward, and operationally implements the definition of the Alltoall operation. The MPI standard specifies the `MPI_Alltoall(...,comm)` (and

`MPI_Alltoallv(...,comm)`) in this way [MPI15, Chapter 5.8].

Proposition 7.1. *In a fully connected communication structure K_p with one-ported, fully bidirectional communication capabilities, the alltoall problem can be solved in $p-1$ dependent communication rounds and one local copy operation. In the linear transmission cost model with latency α and cost per unit β the regular alltoall problem can be solved in $(p-1)(\alpha+\beta m/p)$ time units for total problem size m and block size m/p units.*

Algorithm 20 can easily be extended to k-ported, fully bidirectional capabilities with a corresponding reduction in the number of rounds and communication time.

Corollary 7.1. *In a fully connected communication structure K_p with k-ported, fully bidirectional communication capabilities, the alltoall problem can be solved in $\lceil \frac{p-1}{k} \rceil$ dependent communication rounds and one local copy operation. In the total time cost in the linear transmission cost model is $\lceil \frac{p-1}{k} \rceil (\alpha+\beta m/p)$*

7.2.2 Telephone Communication

Is it possible to implement the Alltoall collective without fully bidirectional communication capabilities? Answer is yes, as the following two algorithms will show. The algorithms in each communication round pairs up processors, and thus need only the weaker bidirectional, telephone communication capabilities, or can even do with only uni-directional communication without running into deadlocks.

Let u and v be processors in the fully connected communication structure K_p in a canonical numbering $0, 1, \ldots, p-1$. The basic paring idea is for a processor u to pair up with partner processor $v = p + i - u$ (modulo p, so $v = (i-u)_p$) in communication round i. This is indeed a pairing, since the partner of v is $p+i-v = p+i-(p+i-u) = u$, and if i takes all values $0 \leq i < p$ it will pair u with each of the other processors, including u itself. This observation is the basis for the simple algorithm shown as Algorithm 21.

Algorithm 21 Algorithm for processor $u \in V(K_p)$ for the Alltoall operation in a fully connected network K_p. Each processor u initially has $S = [m_u^0 \cdots m_u^v \cdots m_u^{p-1}]$ blocks that are indexed as $S[v]$, each of size m_u^v. The resulting blocks are going to be received in $R = [m_0^u \cdots m_v^u \cdots m_{p-1}^u]$ indexed as $R[v]$.

```
1: for i = 0, ..., p − 1 do
2:     v ← (i − u)_p                                  ▷ Partner for round i
3:     if u = v then
4:         R[u] ← S[u]                                ▷ Copy own block
5:     else
6:         Send(S[v], v, K_p) ‖ Recv(R[v], v, K_p)
7:     end if
8: end for
```

Algorithm 21 always goes through p communication rounds, and by construction uses only telephone-like bidirectional communication, and each processor u will in each round be paired with a different processor, including a round where it is paired with itself; in this round, the corresponding data block is copied locally from input buffer $S[u]$ to output buffer $R[u]$.

The algorithm can easily be adopted to the case where the communication system allows only uni-directional communication. Let processor u in some round be paired with processor v; if $u < v$ first send the block from u to v, and then the block from v to u; conversely, if $u > v$. This only doubles the number of communication rounds from p to $2p$.

Algorithm 22 Algorithm for processor $u \in V(K_p)$ for the Alltoall operation in a fully connected network K_p. Each processor u initially has $S = [m_u^0 \cdots m_u^v \cdots m_u^{p-1}]$ blocks that are indexed as $S[v]$, each of size m_u^v. The resulting blocks are going to be received in $R = [m_0^u \cdots m_v^u \cdots m_{p-1}^u]$ indexed as $R[v]$.

```
1: if p ∧ 0x1 = 0x0 then                                   ▷ Even p
2:     for i = 0, ..., p − 2 do
3:         if u = p − 1 then v ← ⌊i/2⌋ + (i mod 2)(p/2)
4:         else if i = (2u)_p then v ← p − 1
5:         else
6:             v ← (p − 1 + i − u)_{p−1}
7:         end if
8:         Send(S[v], v, K_p) ‖ Recv(R[v], v, K_p)
9:     end for
10:    R[u] ← S[u]                          ▷ Copy own block (first or last)
11: else                                    ▷ Odd p, same as Algorithm 21
12:    for i = 0, ..., p − 1 do
13:        v ← (i − u)_p                                ▷ Partner for round i
14:        if u = v then
15:            R[u] ← S[u]                                  ▷ Copy own block
16:        else
17:            Send(S[v], v, K_p) ‖ Recv(R[v], v, K_p)
18:        end if
19:    end for
20: end if
```

Algorithm 20 uses only $p - 1$ communication rounds, while Algorithm 21 always performs p rounds. Is it possible to get down to $p - 1$ rounds with only telephone like bidirectional communication? When p is even, it is possible to form $p/2$ pairs of processors, and since each processor needs to be paired with each other processor (excluding itself), it seems possible to do with $p - 1$ real communication rounds, plus an additional local copy operation per processor. When p is odd, every pairing will have to leave out one processor, and therefore p rounds are required in this case.

For even p, the required sequence of pairings can be achieved as will be explained below. What shall be done can be framed in graph theoretical terms. A (proper) *factor* of an undirected graph G is a subgraph of G whose set of edges $E \subset E(G)$ spans the vertices $V(G)$ of G, that is for each $(u, v) \in E$, both $u, v \in V(G)$. A *factorization* of G is a partition of G into proper factors. A *k-factor* of G is a factor that is a k-regular subgraph. In particular, a 1-factor (one-factor) is a *perfect matching* for G (the set of edges E is an independent set of size $p/2$). A *factorization* of G is a partition of the edges $E(G)$ of G into proper factors. For odd p, a *near 1-factor* is an almost perfect matching of G that excludes one vertex $V(G)$ of G.

A (near) 1-factorization of the fully connected communication structure K_p has $p-1$ (for even p) or p (for odd p) (near) 1-factors. Each factor can be used for telephone communication between the matched pairs; $p-1$ (or p) communication rounds are required, each using a different (near) 1-factor. In the ith round, the ith (near) 1-factor is used.

Theorem 7.1. *The fully connected graph k_p has a 1-factorization for p even, and a near 1-factorization for p odd.*

Proof. Let first p be odd, and assume that $V(K_p) = \{0, 1, \ldots, p-1\}$. For the ith near 1-factor, match vertex u with vertex $v = (i-u)_p$ (that this is indeed a matching or pairing was shown above). For each $i, 0 \leq i < p$, each vertex u is matched with a different vertex, therefore p factors suffice to match all edges $E(K_p)$. Vertex u is matched with itself in the $(2u)_p$th factor, since $u = (i-u)_p \Leftrightarrow i = (2u)_p$. For p odd, $(2u)_p$ takes on all values $0, 1, \ldots, p-1$ for $u \in \{0, 1, \ldots, p-1\}$.

Now let p be even. Treat vertex $u = p-1$ as special, and consider the near 1-factorization of the fully connected graph K_{p-1} where now $p-1$ is odd. In factor $i = (2u)_{p-1}$, match the special vertex of K_p with vertex u. That is, in round $i = (2u)_p$ processor u is matched with processor $p-1$, and processor $p-1$ is matched with processor $u = \lfloor i/2 \rfloor + (i \bmod 2)(p/2)$, since for this u, indeed $(2u)_p = i$. This gives a 1-factorization of K_p for p even. □

The construction is essentially, but not quite, the same as that given in [MR85].

Lemma 7.1. *On fully connected communication structures K_p with one-ported, telephone bidirectional communication, the alltoall problem can be solved in $p-1$ (for even p) and p (for odd p) communication rounds, respectively. With uni-directional communication capabilities, the number of communication rounds is $2(p-1)$ and $2p$, respectively.*

Algorithm 22 uses the optimal (near) 1-factorization of K_p of Theorem 7.1. It takes $p-1$ rounds for even p, and p rounds for odd p. As written, it relies on bidirectional telephone communication, but can easily be adopted to the more restricted uni-directional model. Algorithm 20 does not extend to uni-directional communication.

For $p = 2^d$ being a power of two, there is an easy construction of a 1-factorization of K_p. Each partner of a processor u can be found by flipping one or more bits of u, and all $p-1$ partners can be found by systematically going through all $p-1$ possible bit flips. Therefore, in the ith factor, processor u is matched with processor $v = u \bigtriangledown i$ for $i = 1, 2, \ldots, p-1$ (and this indeed a matching, since $(u \bigtriangledown i) \bigtriangledown i = u \bigtriangledown (i \bigtriangledown i) = u \bigtriangledown \texttt{0x0} = u$). This observation is folklore.

7.3 Irregular Alltoall Problems

are NP-hard [GPT06].

7.4 Chapter notes

The reduce-scatter algorithm for non-commutative operators can be found in [Ian97,BIL03]. For one-factorizations of complete graphs, see [MR85], and [Die17, Chapter 2].

7.5 Exercises

Chapter 8

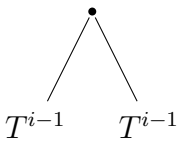
Collectives on Rooted Trees

Rooted communication trees are the easy, first choice for implementing many rooted collectives. We have already seen stars and linear arrays used as special case (degenerate) rooted trees. In this chapter, we will introduce a number of other trees with different properties that often lead to better algorithms with interesting properties.

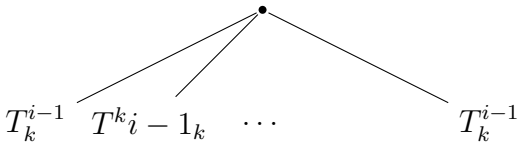
8.1 Structural Properties

Balanced binary and k-ary trees:

$T^0 = \bullet$, $T^i =$

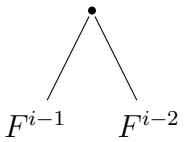


$T_k^0 = \bullet$, $T_k^i =$



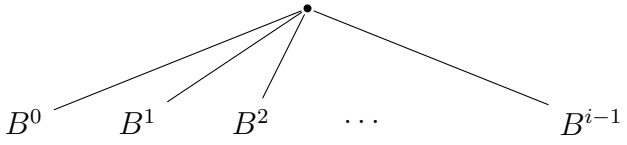
Fibonacci trees:

$F^0 = \bullet$, $F^1 =$, $F^i =$



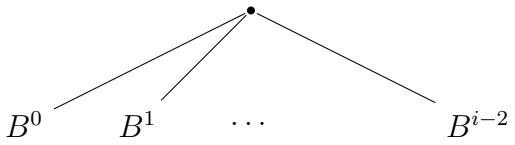
Binomial and k-nomial trees:

$B^0 = \bullet$, $B^i =$

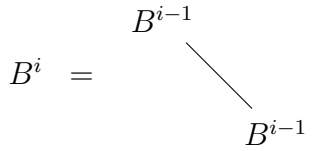


Since

$B^{i-1} =$



B^i is a B^{i-1} with one additional B^{i-1} subtree as last child, and we have



8.2 Algorithms on Fixed-degree Trees

Broadcast and reduction. See Algorithm 23, follows closely the linear ring algorithm, Algortihm 16, which can be seen as a special case.

Algorithm 23 Generic, pipelined algorithm for processor $i \in V(T^k)$ for broadcasting on fixed degree tree T^k from root processor r. The m data elements are divided into n (roughly even sized) blocks of elements that are broadcast one after the other. The jth block is denoted $B[j]$.

```
 1: TREE(i, d, child, parent, T^k)          ▷ Determine position of processor i in T^k
 2: if i = r then                                                  ▷ Processor i is root
 3:     for j = 0, 1, ..., n − 1 do
 4:         for k = 0, 1, ..., d − 1 do
 5:             Send(B[j], child[k], T^k)
 6:         end for
 7:     end for
 8: else if d = 0 then                                  ▷ Processor i is leaf, no children
 9:     for j = 0, 1, ..., n − 1 do
10:         Recv(B[j], parent, T^k)
11:     end for
12: else                                                          ▷ Processor i is interior
13:     Recv(B[0], parent, T^k)                                          ▷ First block
14:     for j = 1, ..., n − 1 do
15:         Send(B[j − 1], child[0], T^k) ∥ Recv(B[j], parent, T^k)
16:         for k = 1, ..., d − 1 do
17:             Send(B[j − 1], child[k], T^k)
18:         end for
19:     end for
20:     for k = 0, 1, ..., d − 1 do                                      ▷ Last block
21:         Send(B[n − 1], child[k], T^k)
22:     end for
23: end if
```

Reduction, scan in fixed-degree trees.

Bidirectional communication in bidirected trees.

8.2.1 Closed trees

Keeping leaves busy by letting them communicate.

More symmetry by dual-root trees.

8.3 Multiple fixed-degree trees

[SST09]

8.4 Bi- and k-nomial trees

8.5 Adaptive trees

Trees that are constructed taking the amount of communication to be done into account.

8.5.1 Adaptive gather and scatter trees

8.5.2 Approximate, fast solutions

8.5.3 Optimal solutions, dynamic programming

8.5.4 Hardness of optimality

8.6 Chapter notes

For generating functions, see any textbook on the topic, e.g., [Wil94, GKP94, Bón16]

The idea of using two (or more) trees simultaneously has been pursued often. The particular use of two binary trees is from [SST09].

8.7 Exercises

Chapter 9

Hypercube Algorithms

Some classics on hypercube: Alltoall communication with message combining. As we have seen, binomial trees can be embedded into hypercubes, so rooted collective algorithms immediately carry over. The algorithms discussed in this chapter are therefore mostly for symmetric collectives like Allgather, ReduceAll, ReduceScatter, and Alltoall. Most of these algorithms are wonderful examples of bidirectional, telephone like communication.

9.1 Allgather

Algorithm 24 Algorithm for processor $i \in V(H_d)$ for the Allgather operation on hypercube H_d. The result data blocks $[m_j]$ at each processor $B = [m_0 \cdots m_j \cdots m_{p-1}]$ are indexed as $B[j]$, and each of size m_j. The block that processor i has initially is $B[i]$.

1: $b, h \leftarrow 1_2, 2^d - 1$ ▷ Bit 0 and all d bits set
2: **for** $k = 0, 1, \ldots d$ **do**
3: $\quad c \leftarrow i \bigtriangledown b$ ▷ Dimension k neighbor in H_d
4: $\quad h \leftarrow h \bigtriangledown b$
5: $\quad s \leftarrow i \wedge h$
6: $\quad$ **if** $i < c$ **then**
7: $\quad\quad$ Send$(B[s : s + (b-1)], c, H_d)$ ‖ Recv$(B[s + b : s + b + (b-1)], c, H_d)$
8: $\quad$ **else**
9: $\quad\quad$ Send$(B[s + b : s + b + (b-1)], c, H_d)$ ‖ Recv$(B[s : s + (b-1)], c, H_d)$
10: $\quad$ **end if**
11: $\quad b \leftarrow b \nwarrow 1$ ▷ Shift bit left (upwards)
12: **end for**

Lemma 9.1. *Algorithm 24 correctly gathers at each process i in hypercube H_d all data blocks $B[0], B[1], \ldots B[p-1]$.*

Proof. By induction on the hypercube dimension d. □

9.2 Alltoall

A classic instance of message-combining.

9.3 Reduction in Hypercubes

Algorithm 25 Algorithm for processor $i \in V(H_d)$ for the ReduceAll and ReduceScatter-like collective operations on hypercube H_d. The initial data of size m at each processor are divided into p blocks $B = [m_0 \cdots m_j \cdots m_{p-1}]$ that are indexed as $B[j]$, and each of size m_j. The resulting block(s) of processor i are likewise stored in $B[j]$ for some indices j. If the `halve` flag is set, the number of data blocks that are sent and received and reduced in each of the d communication rounds is halved. This flag can also be set and reset per iteration as discussed in the text.

1: $b \leftarrow 2^d$ ▷ Total number of blocks
2: $s \leftarrow 0$ ▷ First block index
3: **if** `halve` **then**
4: $h \leftarrow b$
5: **else**
6: $h \leftarrow 0$
7: **end if**
8: **for** $k = 0, 1, \ldots d$ **do**
9: $c \leftarrow i \bigtriangledown (1 \nwarrow k)$ ▷ Dimension k neighbor in H_d
10: $h \leftarrow h \searrow 1$ ▷ Halve
11: $b \leftarrow b - h$
12: **if** $i < c$ **then**
13: $\mathsf{Send}(B[s + h : s + h + (b - 1)], c, H_d) \parallel \mathsf{Recv}(R, c, H_d)$
14: $B[s : s + (b - 1)] \leftarrow B[s : s + (b - 1)] \circ R$
15: **else**
16: $\mathsf{Send}(B[s : s + (b - 1)], c, H_d) \parallel \mathsf{Recv}(R, c, H_d)$
17: $s \leftarrow s + h$
18: $B[s : s + (b - 1)] \leftarrow R \circ B[s : s + (b - 1)]$
19: **end if**
20: **end for**

Algorithm 25 shows how to perform reduction-to-all and also reduce-scatter in hypercubes.

9.3.1 Beyond hypercubes for reduction

[RT04]

9.4 Bin Jia's Broadcast Algorithm

The `nextbit` table can be computed in $O(\log p)$ steps.

[Jia09]

9.5 Edge-disjoint Binomial Trees

9.6 Chapter notes

[vdG94] [Lei92] [JH89]

Chapter 10

Collectives on Circulant Graphs

The circulant graph communication structure (Defintion 2.13) is very versatile. In some senses, it generalizes the ring, so many of the ideas for ring algorithms will come up again. In the algorithms described in this chapter, a logarithmic number (in the number of processors) of skips will be used, and the processors will all do a logarithmic number of communication operations, roughly one for each skip. The circulant graph in other senses generalizes hypercubes and butterflies, also a helpful observation for the algorithms to be developed now.

10.1 Straight Doubling Circulant Graphs

10.1.1 The Allgather Operation

The *straight doubling skip circulant graph* communication structure uses skips $s = 1, 2, 4, 8, \ldots$, that is skips that are powers of 2 and smaller than the number of processors p. Such circulant graphs C_p^s are well suited for the (regular) Allgather collective, and, by extension, to the Alltoall operation, too. This was discovered and made known in a paper by Bruck *et al.*. [BHK+97].

The Allgather algorithm is illustrated as Algorithm 26. Initially, the block of each processor i is locally copied into the temporary array of blocks B as block $B[0]$ (of size m_i). The circulant graph is used in reverse order, as discussed already for the ring Allgather algorithm. As is easy to see, after communication round j, processor i has received blocks $B[0 \ldots s-1]$, and after the last round, all p blocks will have been collected as $B[0 \ldots p-1]$. These blocks, except block $B[0]$ (which corresponds to $R[i]$ which is by assumption already in place) are copied into the resulting array R cyclically from position $R[i+1]$.

MPI implementation concerns Can the extra array and copying be avoided?

10.1.2 Asymmetric Uses of Circulant Graphs: Prefix-sums Algorithms

Algorithm 27 is the "Hillis-Steele" parallel prefix-sums algorithm [HGLS86].

Algorithm 26 Algorithm for processor $i \in V(C_p^s)$ for the Allgather operation on the circulant graph C_p^s with doubling skips s. The result data blocks $[m_i]$ at each process $R = [m_0 \cdots m_j \cdots m_{p-1}]$ are indexed as $R[j]$, and are each of size m_j. The block that processor i has initially is $R[i]$. The algorithm uses an auxiliary array of blocks B in which the result is collected and eventually copied into the right positions of R.

$B[0] \leftarrow R[i]$
$s \leftarrow 1$
while $s < p$ **do**
    $t, f \leftarrow (i-s)_p, (i+s)_p$ ▷ To- and from-processors
    $b \leftarrow s$ ▷ Number of blocks
    if $s + b > p$ **then** $b \leftarrow p - s$
    end if
    $\mathsf{Send}(B[0:b-1], t, C_p^s) \parallel \mathsf{Recv}(B[s:s+b-1], f, C_p^s)$
    $s \leftarrow s \nwarrow 1$ ▷ Double s by shifting left (upwards)
end while
for $j = 1, \ldots, p-1$ **do**
    $R[(i+j)_p] \leftarrow B[j]$
end for

Algorithm 27 Algorithm for processor $i \in V(C_p^s)$ for the ScanInclusive operation on the circulant graph C_p^s with doubling skips s. Both initial buffer and result is stored in $R = [m]$.

$s \leftarrow 1$
$t, f \leftarrow i+s, i-s$ ▷ To- and from-processors
while $0 \leq f \wedge t < p$ **do**
    $\mathsf{Send}(R, t, C_p^s) \parallel \mathsf{Recv}(T, f, C_p^s)$
    $R \leftarrow T \circ R$
    $s \leftarrow s \nwarrow 1$ ▷ Double s by shifting left (upwards)
    $t, f \leftarrow i+s, i-s$ ▷ To- and from-processors
end while
while $t < p$ **do**
    $\mathsf{Send}(R, t, C_p^s)$
    $s \leftarrow s \nwarrow 1$ ▷ Double s by shifting left (upwards)
    $t \leftarrow i+s$ ▷ To-processor
end while
while $0 \leq f$ **do**
    $\mathsf{Recv}(T, f, C_p^s)$
    $R \leftarrow T \circ R$
    $s \leftarrow s \nwarrow 1$ ▷ Double s by shifting left (upwards)
    $f \leftarrow i-s$ ▷ From-processor
end while

10.1.3 Alltoall

The doubling skip circulant graph algorithm for the Alltoall operation is a (the) classic message combining algorithm.

10.2 Roughly Halving Circulant Graphs

10.2.1 Allreduce

Algorithm 28 Allreduce for small problems. Circulant graph C_p^s.

10.2.2 Reduce-scatter

10.2.3 Allgather

MPI implementation concerns:

10.3 Specializing the Algorithms

The rooted collectives are special cases of the symmetric, non-root collectives. We can specialize the algorithms presented above.

10.3.1 Reduce

10.3.2 Gather and Scatter

10.4 Broadcast with Circulant Graphs

10.5 An application to irregular Allgather

10.6 Chapter notes

A most influential paper using the doubling skips circulant graph pattern is the work by Bruck *et al.* [BHK+97].

Allreduce for small problems from [BNKS93]. Essentially same algorithm, generalization to k-ported communication [BH93].

See also [BNBH+95].

The classic "Hillis-Steele" prefix-sums algorithm is folklore and can be found in many early papers, e.g., [KRS85, HGLS86].

10.7 Exercises

Chapter 11

Meshes and Tori

11.1 Chapter notes

11.2 Exercises

Chapter 12

Managing Linearly Ordered Data Layouts

In our algorithms we have used linearly ordered data structures freely. For data stored consecutively in (strided) arrays, this is entirely unproblematic. For non-consectutive, possibly irregularly stored data, this is non-trivial. This chapter deals with ways to describe and access lineaarly ordered, but non-consective data; mainly tree- or DAG (directed acyclic graph) descriptions.

MPI provides capabilities for describing and using such data by the user-defined, or derived datatype mechanism, see Section 4.3.

12.1 Description

12.2 Processing

12.3 Chapter notes

Lots

12.4 Exercises

Chapter 13

Algorithms for Hierarchical Systems

13.1 Chapter notes

[Lei92] [TH20]

13.2 Exercises

Chapter 14

Process to Processor Mapping

14.1 Chapter notes

14.2 Exercises

Chapter 15

Sparse Collective Communications

15.1 Patterns: Cartesian Collective Communication

15.2 Chapter notes

[TH19]

15.3 Exercises

Chapter 16

Other Collective Operations: Maintaining Bitmaps, Array Compaction, Changing Distributions

16.1 Non-oblivious collectives

Sometimes called "sparse collectives". Reduction operators have algebraic properties that can sometimes be used to reduce communication, e.g., neutral (zero-, identity-) elements may not have to be communicated. This is explored in this section.

16.2 Bitmaps and Sparse Reductions

An example of non-oblivious collectives.

16.3 Array Compation

We need bitonic merge, so there will be a subsection on that.

16.4 Changing distributions

This used to be hot. And it is still relevant.

16.5 Chapter notes

16.6 Exercises

Chapter 17

Benchmarking Collective Operations

Actual performance numbers will be of passing interest.

17.1 Chapter notes

17.2 Exercises

Bibliography

[AG94] G. S. Almasi and A. Gottlieb. *Highly Parallel Computing*. Benjamin/Cummings, second edition, 1994.

[AGH+94] Cliff Addison, Vladimir Getov, Anthony J. G. Hey, Roger W. Hockney, and I. C. Wolton. Benchmarking for distributed memory parallel systems: Gaining insight from numbers. *Parallel Computing*, 20(10-11):1653–1668, 1994.

[AISS97] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris J. Scheiman. LogGP: Incorporating long messages into the LogP model for parallel computation. *Journal of Parallel and Distributed Computing*, 44(1):71–79, 1997.

[Akl89] Selim G. Akl. *The Design and Analysis of Parallel Algorithms*. Prentice-Hall, 1989.

[Bab16] László Babai. Graph isomorphism in quasipolynomial time. In *48th Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 684–697, 2016.

[Bab19] László Babai. Canonical form for graphs in quasipolynomial time: preliminary report. In *51st Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 1237–1246, 2019.

[BBC+95] Vasanth Bala, Jehoshua Bruck, Robert Cypher, Pablo Elustondo, Alex Ho, Ching-Tien Ho, Shlomo Kipnis, and Marc Snir. CCL: A portable and tunable collective communications library for scalable parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):154–164, 1995.

[BDQ86] Jean-Claude Bermond, Charles Delorme, and Jean-Jacques Quisquater. Strategies for interconnection networks: Some methods from graph theory. *Journal of Parallel and Distributed Computing*, 3(4):433–449, 1986.

[Ber73] Claude Berge. *Graphs and Hypergraphs*, volume 6 of *North-Holland Mathematical Library*. North-Holland, third revised edition edition, 1973.

[BH93] Jehoshua Bruck and Ching-Tien Ho. Efficient global combine operations in multi-port message-passing systems. *Parallel Processing Letters*, 3(4):335–346, 1993.

[BHK+97] Jehoshua Bruck, Ching-Tien Ho, Schlomo Kipnis, Eli Upfal, and Derrick Weathersby. Efficient algorithms for all-to-all communications in multiport message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 8(11):1143–1156, 1997.

[BIL03] Massimo Bernaschi, Giulio Iannello, and Mario Lauria. Efficient implementation of reduce-scatter in MPI. *Journal of Systems Architecture*, 49(3):89–108, 2003.

[BJG09] Jørgen Bang-Jensen and Gregory Z. Gutin. *Digraphs – Theory, Algorithms and Applications*. Springer Monographs in Mathematics. Springer, second edition, 2009.

[BNBH+95] Amotz Bar-Noy, Jehoshua Bruck, Ching-Tien Ho, Schlomo Kipnis, and Baruch Schieber. Computing global combine operations in the multiport postal model. *IEEE Transactions on Parallel and Distributed Systems*, 6(8):896–900, 1995.

[BNKS93] Amotz Bar-Noy, Shlomo Kipnis, and Baruch Schieber. An optimal algorithm for computing census functions in message-passing systems. *Parallel Processing Letters*, 3(1):19–23, 1993.

[Bol98] Béla Bollobás. *Modern Graph Theory*, volume 184 of *Graduate Texts in Mathematics*. Springer, 1998.

[Bón16] Miklós Bóna. *Introduction to Enumerative and Analytic Combinatorics*. CRC Press, second edition, 2016.

[BP89] J.-C. Bermond and C. Peyrat. The de Bruijn and Kautz networks: A competition for the hypercube? In *Hypercube and Distributed Computers, Proceedings sur le 1er Colleque Europeen sur les Hypercubes*, pages 279–293, 1989.

[BT80] W. G. Bridges and Sam Toueg. On the impossibility of directed moore graphs. *Journal of Combinatorial Theory*, 29(3):339–341, 1980.

[CHPvdG07] Ernie Chan, Marcel Heimlich, Avi Purkayastha, and Robert A. van de Geijn. Collective communication: theory, practice, and experience. *Concurrency and Computation: Practice and Experience*, 19(13):1749–1783, 2007.

[CKP+93] David E. Culler, Richard M. Karp, David A. Patterson, Abhijit Sahay, Klaus E. Schauser, Eunice E. Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: Towards a realistic model of parallel computation. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 1993.

[CKP+96] David E. Culler, Richard M. Karp, David Patterson, Abhijit Sahay, Eunice E. Santos, Klaus Erik Schauser, Ramesh Subramonian, and Thorsten von Eicken. LogP: A practical model of parallel computation. *Communications of the ACM*, 39(11):78–85, 1996.

[CLRS22] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, fourth edition, 2022.

[Cod68] E. F. Codd. *Cellular Automata*. Academic Press, 1968.

[dB46] N. G. de Bruijn. A combinatorial problem. In *Proceedings of the Section of Sciences of the Koninklijke Nederlandse Akademie van Wetenschappen te Amsterdam*, volume 49, pages 758–764, 1946.

[DFF+03] Jack Dongarra, Ian Foster, Geoffrey Fox, William Gropp, Ken Kennedy, Linda Torczon, and Andy White, editors. *Sourcebook of Parallel Computing*. Morgan Kaufmann Publishers, 2003.

[Die17] Reinhard Diestel. *Graph Theory*. Springer, fifth edition, 2017.

[DT04] William James Dally and Brian Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers, 2004.

[DYN03] José Duato, Sudhakar Yalamanchili, and Lionel Ni. *Interconnection Networks*. Morgan Kaufmann Publishers, revised printing edition, 2003.

[EGCSY05] Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick. *UPC: Distributed Shared Memory Programming*. John Wiley & Sons, 2005.

[Eve12] Shimon Even. *Graph Algorithms*. Cambridge University Press, second edition, 2012. First edition 1976.

[FHL97] Ivan Friš, Ivan Havel, and Petr Liebl. The diameter of the cube-connected cycles. *Information Processing Letters*, 61(3):157–160, 1997.

[FJL+88] Geoffrey C. Fox, Mark Johnson, Gregory Lyzenga, Steve Otto, John Salmon, and David Walker. *Solving Problems on Concurrent Processors*, volume Volume 1: General Techniques and Regular Problems. Prentice-Hall, 1988.

[FL94] Pierre Fraigniaud and Emmanuel Lazard. Methods and problems of communication in usual networks. *Discrete Applied Mathematics*, 53(1–3):79–133, 1994.

[GHTL14] William Gropp, Torsten Hoefler, Rajeev Thakur, and Ewing Lusk. *Using Advanced MPI*. MIT Press, 2014.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, 1979. With an addendum, 1991.

[GKP94] Ronald Graham, Donald E. Knuth, and Oren Pataschnik. *Concrete Mathematics*. Addison-Wesley, second edition, 1994.

[GLS94] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 1994. Second printing, 1995.

[GLT99] William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press, 1999.

[GPT06] Alfredo Goldman, Joseph G. Peters, and Denis Trystram. Exchanging messages of different sizes. *Journal of Parallel and Distributed Computing*, 66(1):1–18, 2006.

[GR88] Alan Gibbons and Wojciech Rytter. *Efficient Parallel Algorithms*. Cambridge University Press, 1988.

[Har69] Frank Harary. *Graph Theory*. Addison-Wesley, 1969.

[HCA16] Sascha Hunold and Alexandra Carpen-Amarie. Reproducible MPI benchmarking is still not as easy as you think. *IEEE Transactions on Parallel and Distributed Systems*, 27(12):3617–3630, 2016.

[HGLS86] W. Daniel Hillis and Jr. Guy L. Steele. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[HHL88] Sandra M. Hedetniemi, T. Hedetniemi, and Arthur L. Liestman. A survey of gossiping and broadcasting in communication networks. *Networks*, 18:319–349, 1988.

[HHMW94] Rolf Hempel, Anthony J. G. Hey, Oliver McBryan, and David W. Walker. Special issue: Message passing interfaces. *Parallel Computing*, 20(4):415–678, 1994.

[HHW88] Frank Harary, John P. Hayes, and Horng-Jyh WU. A survey of the theory of hypercube graphs. *Computers& Mathematics with Applications*, 15(4):277–289, 1988.

[HIK11] Richard Hammack, Wilfried Imrich, and Sandi Klavžar. *Handbook of product graphs*. Discrete Mathematics and its Applications (Boca Raton). CRC Press, second edition, 2011. With a foreword by Peter Winkler.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Hoc87] Roger W. Hockney. Parametrization of computer performance. *Parallel Computing*, 5(1-2):97–103, 1987.

[Hoc91] Roger W. Hockney. Performance parameters and benchmarking of supercomputers. *Parallel Computing*, 17(10-11):1111–1130, 1991.

[Hoc94] Roger W. Hockney. The communication challenge for MPP: Intel Paragon and Meiko CS-2. *Parallel Computing*, 20(3):389–398, 1994.

[HW99] Rolf Hempel and David W. Walker. The emergence of the MPI message passing standard for parallel computing. *Computer Standards & Interfaces*, 21:51–62, 1999.

[HW11] Georg Hager and Gerhard Wellein. *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press. Taylor & Francis Group, 2011.

[Ian97] Giulio Iannello. Efficient algorithms for the reduce-scatter operation in LogGP. *IEEE Transactions on Parallel and Distributed Systems*, 8(9):970–982, 1997.

[JáJ92] Joseph JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[JH89] S. Lennart Johnsson and Ching-Tien Ho. Optimum broadcasting and personalized communication in hypercubes. *IEEE Transactions on Computers*, 38(9):1249–1268, 1989.

[Jia09] Bin Jia. Process cooperation in multiple message broadcast. *Parallel Computing*, 35(12):572–580, 2009.

[Kau68] William H. Kautz. Bounds on directed (d, k) graphs. Technical Report Theory of cellular logic networks and machines, AFCRL-68-0668 SRI Project 7258 Final report, Stanford Research Institute, 1968.

[Kau69] W. H. Kautz. Design of optimal interconnection networks for multiprocessors. In *Architecture and design of digital computers, Nato Advanced Summer Institute*, pages 249–272, 1969.

[KGGK94] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing*. Benjamin/Cummings, 1994.

[Knu73] Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, second edition, 1973.

[Knu11] Donald E. Knuth. *Conbinatorial Algorithms, Part1*, volume 4A of *The Art of Computer Programming*. Addison-Wesley, 2011.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, second edition, 1988.

[KRS85] Clyde P. Kruskal, Larry Rudolph, and Marc Snir. The power of parallel prefix. *IEEE Transactions on Computers*, C-34(10):965–968, 1985.

[Kun80] H. T. Kung. *The Structure of Parallel Algorithms*, volume 19, pages 65–112. Academic Press, 1980.

[Lei85] Charles E. Leiserson. Fat-trees: Universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers*, 34(10):892–901, 1985.

[Lei92] F. Thomson Leighton. *Introduction to Parallel Algorithms and Architechtures: Arrays, Trees, Hypercubes*. Morgan Kaufmann Publishers, 1992.

[Lev11] John M. Levesque. *High Performance Computing. Programming and Applications*. CRC Press. Taylor & Francis Group, 2011.

[LH22] Shigang Li and Torsten Hoefler. Near-optimal sparse allreduce for distributed deep learning. In *27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 135–149. ACM, 2022.

[LS76] Tomás Lang and Harold S. Stone. A shuffle-exchange network with simplified control. *IEEE Transactions on Computers*, 25(1):55–65, 1976.

[LS08] Calvin Lin and Lawrence Snyder. *Principles of Parallel Programming.* Pearson/Addison-Wesley, 2008.

[MC93] Dikran S. Meliksetian and C. Y. Roger Chen. Optimal routing algorithm and the diameter of the cube-connected cycles. *IEEE Transactions on Parallel and Distributed Systems*, 4(10):1172–1178, 1993.

[MPI15] MPI Forum. *MPI: A Message-Passing Interface Standard. Version 3.1*, June 4th 2015. `www.mpi-forum.org`.

[MPI21] MPI Forum. *MPI: A Message-Passing Interface Standard. Version 4.0*, June 9th 2021. `www.mpi-forum.org`.

[MR85] Eric Mendelsohn and Alexander Rosa. One-factorizations of the complete graph – a survey. *Journal of Graph Theory*, 9:43–65, 1985.

[MS90] B. Monien and H. Sudborough. Embedding one interconnection network into another. In G. Tinhofer, E. Mayr, H. Noltemeier, and M. M. Syslo, editors, *Computational Graph Theory*, volume 7 of *Computing Suppl.*, pages 257–282. Springer, 1990.

[MŠ13] Mirka Miller and Jozef Širáň. Moore graphs and beyond: A survey of the degree/diameter problem. *The Electronic Journal of Combinatorics*, 20(2), 2013. Dynamic survey.

[Pac97] Peter Pacheco. *Parallel Programming with MPI.* Morgan Kaufmann Publishers, 1997.

[Pac11] Peter S. Pacheco. *An Introduction to Parallel Programming.* Morgan Kaufmann Publishers, 2011.

[PV81] Franco P. Preparata and Jean Vuillemin. The cube-connected cycles: A versatile network for parallel computation. *Communications of the ACM*, 24(5):300–309, 1981.

[Qui87] Michael J. Quinn. *Designing Efficient Algorithms for Parallel Computers.* McGraw-Hill, 1987.

[Qui03] Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP.* McGraw-Hill, 2003.

[Ray13] Michel Raynal. *Distributed Algorithms for Message-Passing Systems.* Springer, 2013.

[Ray18] Michel Raynal. *Fault-tolerant Message-Passing Distributed Systems*. Springer, 2018.

[RMML19] Juan A. Rico-Gallego, Juan Carlos Díaz Martín, Ravi Reddy Manumachu, and Alexey L. Lastovetsky. A survey of communication performance models for high-performance computing. *ACM Computing Surveys*, 51(6):126:1–126:36, 2019.

[RR10] Thomas Rauber and Gudula Rünger. *Parallel Programming for Multicore and Cluster Systems*. Springer, second edition, 2010.

[RT04] Rolf Rabenseifner and Jesper Larsson Träff. More efficient reduction algorithms for message-passing parallel systems. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 11th European PVM/MPI Users' Group Meeting*, volume 3241 of *Lecture Notes in Computer Science*, pages 36–46. Springer, 2004.

[SAB18] Thomas Sterling, Matthew Anderson, and Maciej Brodowicz. *High Performance Computing. Modern Systems and Practices*. Morgan Kaufmann Publishers, 2018.

[SCH81] Peter J. Slater, Ernest J. Cockayne, and Stephen T. Hedetniemi. Information dissemination in trees. *SIAM Journal on Computing*, 10(4):692–701, 1981.

[Sei85] Charles L. Seitz. The cosmic cube. *Communications of the ACM*, 28(1):22–33, 1985.

[SGDHS18] Bertil Schmidt, Jorge Gonzaález-Domínguez, Christian Hundt, and Moritz Schlarb. *Parallel Programming. Concepts and Practice*. Morgan Kaufmann Publishers, 2018.

[SP89] Maheswara R. Samatham and Dhiraj K. Pradhan. The de Bruijn multiprocessor network: A versatile parallel processing and sorting network for VLSI. *IEEE Transactions on Computers*, 38(4):567–581, 1989.

[SST09] Peter Sanders, Jochen Speck, and Jesper Larsson Träff. Two-tree algorithms for full bandwidth broadcast, reduction and scan. *Parallel Computing*, 35(12):581–594, 2009.

[Sto71] Harold S. Stone. Parallel processing with the perfect shuffle. *IEEE Transactions on Computers*, 20(2):153–161, 1971.

[Tar83] Robert Endre Tarjan. *Data Structures and Network Algorithms*. Society of Industrial and Applied Mathematics (SIAM), 1983.

[TGT10] Jesper Larsson Träff, William D. Gropp, and Rajeev Thakur. Self-consistent MPI performance guidelines. *IEEE Transactions on Parallel and Distributed Systems*, 21(5):698–709, 2010.

[TH19] Jesper Larsson Träff and Sascha Hunold. Cartesian collective communication. In *48th International Conference on Parallel Processing (ICPP)*, pages 48:1–48:11, 2019.

[TH20] Jesper Larsson Träff and Sascha Hunold. Decomposing MPI collectives for exploiting multi-lane communication. In *IEEE International Conference on Cluster Computing (CLUSTER)*, pages 270–280. IEEE Computer Society, 2020.

[THMH21] Jesper Larsson Träff, Sascha Hunold, Guillaume Mercier, and Daniel J. Holmes. MPI collective communication through a single set of interfaces: A case for orthogonality. *Parallel Computing*, 107, 2021.

[TM87] Tommaso Toffoli and Norman Margolus. *Cellular Automata Machines: A New Environment for Modeling*. MIT Press, second edition, 1987.

[Trä10] Jesper Larsson Träff. Transparent neutral element elimination in MPI reduction operations. In *Recent Advances in Message Passing Interface. 17th European MPI Users' Group Meeting*, volume 6305 of *Lecture Notes in Computer Science*, pages 275–284. Springer, 2010.

[TRH14] Jesper Larsson Träff, Antoine Rougier, and Sascha Hunold. Implementing a classic: Zero-copy all-to-all communication with MPI datatypes. In *28th ACM International Conference on Supercomputing (ICS)*, pages 135–144. ACM, 2014.

[TRS+10] Jesper Larsson Träff, Andreas Ripke, Christian Siebert, Pavan Balaji, Rajeev Thakur, and William Gropp. A pipelined algorithm for large, irregular all-gather problems. *International Journal of High Performance Computing Applications*, 24(1):58–68, 2010.

[vdG94] Robert van de Geijn. On global combine operations. *Journal of Parallel and Distributed Computing*, 22:324–328, 1994.

[War13] Henry S. Warren. *Hacker's Delight*. Addison-Wesley, second edition, 2013.

[Wil94] Herbert S. Wilf. *Generatingfunctionology*. Academic Press, second edition, 1994.

[ZS04] H. P. Zima and M. Shimasaki. Special issue: The Earth Simulator. *Parallel Computing*, 30(12):1277–1343, 2004.