Data

In single-program multiple-data (SPMD) parallel programs, global data is partitioned, with a portion of the data assigned to each processing node. Issues relevant to choosing a partitioning strategy include:

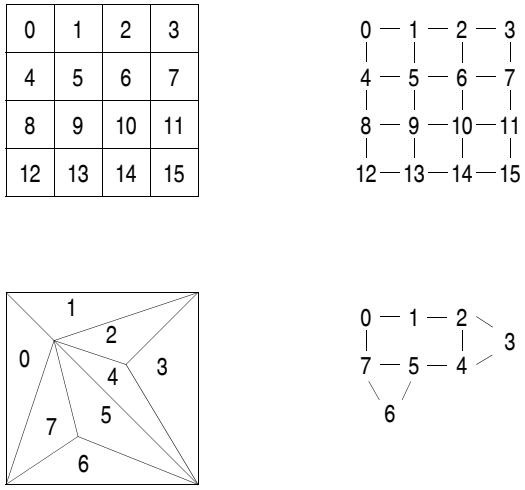
- *Load Imbalance Overhead*—A time cost is incurred whenever some nodes are kept idle while they wait for others to finish a task. Therefore, try to distribute the computational work evenly among the nodes.
- *Communication Overhead*—Internode communication is often required for carrying out computations on partitioned data. Each internode message requires, at both the sending and the receiving nodes, a certain startup time plus a time proportional to the length of the message. In addition, each message incurs a latency which can exacerbate load imbalance. Therefore, try to minimize the communication-to-computation ratio.
- *Programming Complexity*—The algorithms used to compute data partitions vary in the amount of programmer effort they require. Library routines are available for partitioning regular data structures. Custom routines may be required for irregular structures. In general, the more complex the partitioning algorithm, the better the resulting load balance and communication efficiency.

In certain applications—tracking or simulating complex systems, for example—the distribution of computational work among the nodes can evolve, resulting in load imbalance. In some cases, adaptive repartitioning of the data may be required.

In many algorithms, only elements near each other in the data structure enter into any single computational step. Schemes for the numerical solution of differential equations are often of this type. At every iteration, each entry in the data matrix is updated, based only on its own current value and the values of neighboring entries.

In programs that perform computations only on neighboring data, internode communication typically occurs across the common boundaries of adjacent regions of the partitioned data. The distributed data structure is then characterized not only by the amount and internal organization of the data in each region, but also by the topology of the communication network that links adjacent regions across their common boundaries. Figure 1-31 shows two ways to partition a two-dimensional region, and the resulting communication topologies.

Figure 1-31: Communication Topologies



Global computations on distributed data often exploit a particular topology of message-passing links. In particular, sequential algorithms based on the principle of *divide-and-conquer* can often be made parallel using a tree topology. Common tree structures and algorithms based on them are discussed in the *Trees* section, below.

Sometimes, partitioning can be avoided altogether. For instance, an array that is read often but seldom written can be copied onto each node. Any time a write is performed, all the nodes must be notified, in order to maintain data coherency.

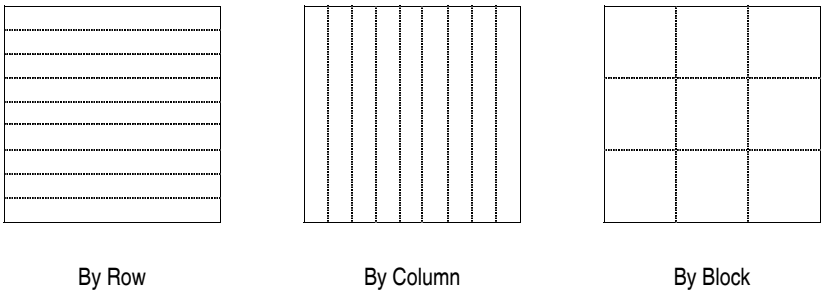
Data Partitioning

Several approaches are available for partitioning a global data set among nodes. An appropriate choice in any particular case depends on the nature of the data and the computations to be performed on it. In general, simpler partitioning strategies apply to well-defined problem types, while more complex strategies can apply to a wider range of problems. The next few sections discuss a variety of strategies, in order of increasing complexity.

■ Regular Partitions

A data set with a particularly simple inherent geometry—multidimensional arrays are a common example—can be partitioned among nodes in a regular, repeating pattern. Matrices, for instance, can be divided up by row, by column, or by block, as illustrated in Figure 1-32:

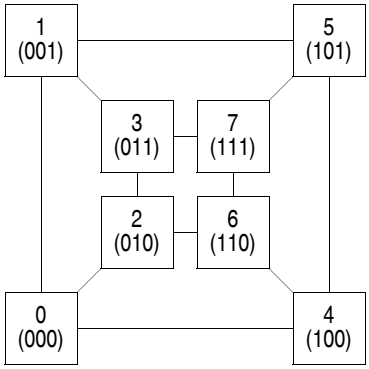
Figure 1-32: Regular Partitioning



Of these three approaches, partitioning by block is often preferred when only elements near each other in the data structure enter into any single computational step. Such computations require interprocessor communication only along partition boundaries. Since the block partition creates a smaller total boundary-length (4 units, in the figure) than the other approaches (8 units), it has the smallest communication-to-computation ratio.

The blocks of a partitioned array can be mapped onto the hypercube nodes simply and efficiently using *gray codes*. Each block (subarray) of data is assigned a code number that differs by only one bit from the code assigned to any of its neighbors, as shown in Figure 1-33:

Figure 1-33: Gray Code Mapping



The nodes of the nCUBE 2 hypercube have gray-code hardware addresses; nearest neighbors have processor ID (PID) numbers that differ only in a single bit. Once the data blocks have been assigned gray code numbers, each block is downloaded to the node whose PID matches (in its low-order bits) the code number of the block. In this way, the mesh communication topology

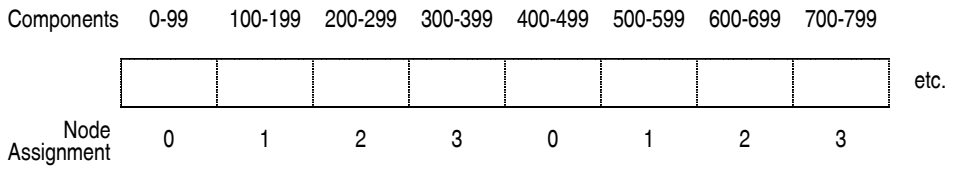
of the block-partitioned array is mapped into the hypercube topology so that neighboring data blocks always reside on neighboring nodes. This maximizes the data communication speed between nodes.

Setting up a regular partition requires very little computational effort. A regular partition establishes a simple, mesh-like communications topology which is easily mapped into the hypercube with library routines. In many applications, however, regular partitioning can result in a load imbalance. Regular partitioning is useful primarily for problems in which all the data is subject to the same (or similar) computations. For example, consider a scheme for approximating a solution to a differential equation using a simple recurrence relation. In any iteration, the amount of computational work required for each entry in the data matrix is the same.

In the case of an airflow simulation, a regular partition might divide the space into equal rectangular prisms. This is a good strategy for simulating a quiet volume, such as a wind-tunnel with uniform flow, where the air density and pressure are reasonably uniform. Each partition in this three-dimensional world would have only six neighbors. The small surface-area-to-volume ratio of the partitions would keep communication overhead low. If a jet engine were placed in the room, however, the resulting load imbalance would be intolerable. The air inside a rectangular prism close to the engine would require much more computation than the air in a prism of the same size in a corner of the room.

In some applications, the computational work required to process a global data object is concentrated in certain locations of the data, but it is not known in advance which locations those will be. In a numerical integration scheme, for example, the evaluation of the function to be integrated may be more difficult in some regions than in others. In such cases, a regular partition can be used to enhance load balance by maximizing the *dispersion* of data. The diagram below represents the partition of a long vector (with 12,000 components) among four nodes, with no node being required to process any more than 100 contiguous entries. In this way, computationally intensive clusters of data (on the order of a few hundred components in length) will be dispersed evenly among the processors. (See Figure 1-34.)

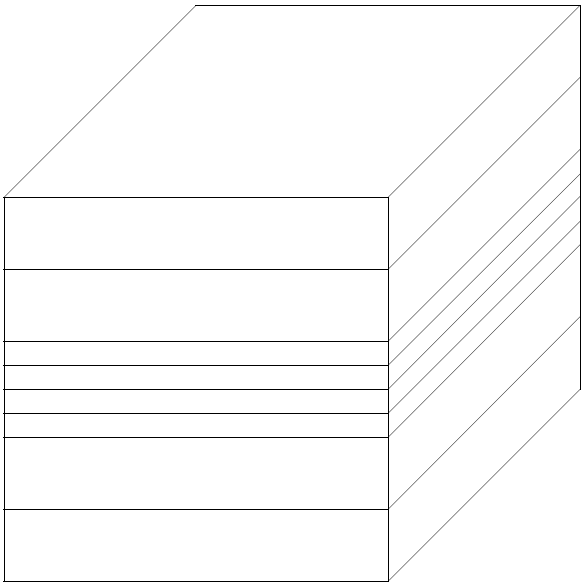
Figure 1-34: Maximum Dispersion



■ Irregular Partitions

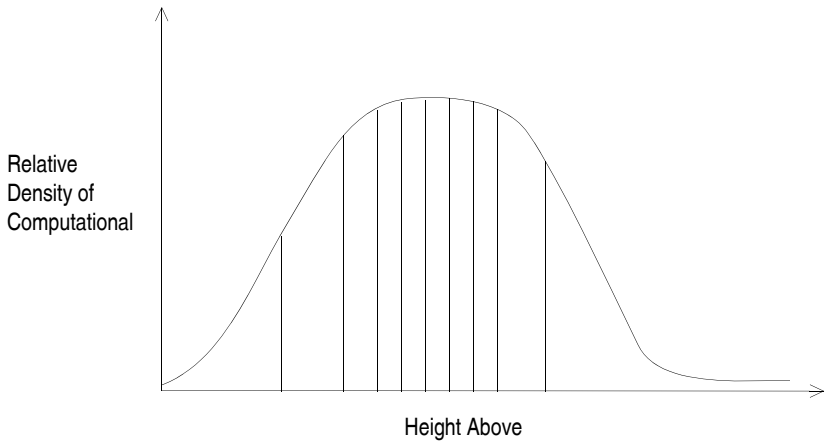
Simple irregular partitions can sometimes be used when a regular partition would cause too great a load imbalance. In the airflow simulation, for example, the room could be divided up with a collection of horizontal planes of varying vertical distances from each other. The data and computations for each of the resulting slices is assigned to a different node. The slices will be thin where they contain part of the jet and thick where there is relatively little air movement. (See Figure 1-35.)

Figure 1-35: Irregular Partition



To compute such a partition, you would first estimate the density of computational work as a function of height, using either past experience or the physics of the problem. The height axis is then subdivided into the same number of intervals as there are nodes, in such a way that the computational work (the area under the density graph) is the same in each interval. (See Figure 1-36.)

Figure 1-36: Computation vs. Height



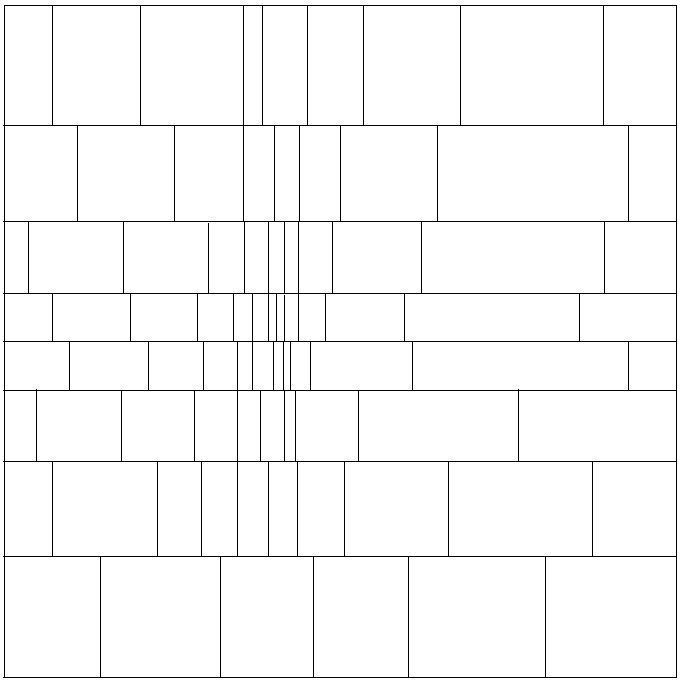
In the resulting partition, each region has only two neighbors. However, each of the boundaries has a very large surface area, creating communication overhead: the larger the surface area of a region, the greater the chance that a particle will cross the boundary and require internode communication.

In general, the more complex the partitioning algorithm, the better the resulting load balance and communication efficiency. For each program, you must decide where the optimal trade-off between complexity and performance lies.

Using arbitrarily shaped regions with low surface-area-to-volume ratios, you could in principle balance the computational load while minimizing communication overhead. This general optimization problem, however, is computationally intractable (NP-complete). Calculating even a near-optimal partition generally requires a great deal of computational effort. Another drawback of such a strategy is that the resulting communication topology may be highly irregular—each node could have any number of neighbors, for example—making communication routines difficult to program.

There are partitioning methods of intermediate complexity. After dividing an area into bands of varying thickness, for example, one could go further, and subdivide each slice. A two dimensional area partitioned in such a way would look like Figure 1-37:

Figure 1-37: Two-Dimensional Irregular Partition.



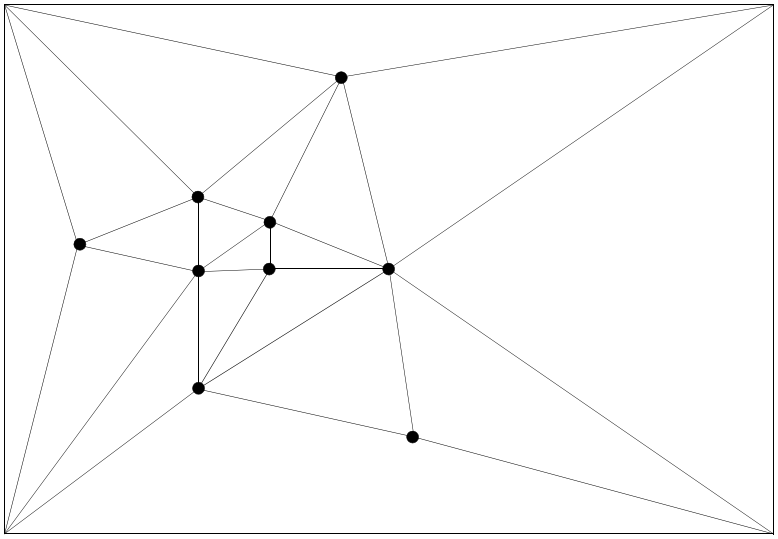
Such a partition is relatively easy to compute. It entails a slightly greater complexity in the communication network than is required for the simple partition into bands, however. Each region has exactly one right and one left neighbor, but a variable number of neighbors above and below. Each program element must therefore maintain a map of its upper and lower boundaries, showing which boundary communications must be directed to which neighbor.

Other partitioning methods can be found. There is, for example, nothing sacrosanct about rectangular subregions. Efficient algorithms exist for dividing a large rectangle into triangular regions in such a way that:

- The regions are smaller and more numerous where the computational load is greatest. You must specify the distribution of computational work throughout the original rectangle as input data for the algorithm.
- The regions are not long and thin. A reasonable boundary-to-interior ratio must be maintained, so as to minimize communication overhead.

Figure 1-38 illustrates an area partitioned using a method called Delaney triangulation. The computational load is concentrated in the middle left.

Figure 1-38: Delaney-Triangulated Rectangle



Communication is usually simple in such a partition, since each region has exactly three neighbors along its edges. On the other hand, any message that has to pass through a vertex of one of the triangles requires special handling, since the number of regions sharing a vertex can be arbitrarily large. Algorithms for decomposing a three dimensional space into tetrahedra also exist, but they are complicated.

■ Random-Access Data

In the previous examples of partitioning, only data from predictable locations in a given data structure entered into any one computational step. In many applications, however, computations require data from unpredictable locations in a global data structure. In these cases, each item is typically stored together with information about its position in the global data structure.

In sparse matrices, for example, the location of non-zero elements is unpredictable. A row index, a column index, or both, are therefore typically stored along with each non-zero value.

Often, the records in a database are sorted in alphabetical order using a particular field. The value of that field for a given record encodes the location of the record.

There are two approaches to partitioning such data:

- *Ad Hoc Optimization*—If possible, arrange the data so as to minimize the amount of data-movement required for the computations. Near-optimal partitions have been worked out in the research literature for many special cases.
- *Data Permutation*—Partition the data in a conceptually simple way, and make use of library routines for sorting or other data permutations before each global computation. nCUBE 2 libraries contain routines for common data permutations like sorting, matrix transposition, and index-shuffling for fast fourier transforms. Permutation routines typically work in two stages. First they tag each piece of data with the ID of its destination node, then they move the data.

Dynamic Repartitioning

For applications in which the computational work required by each block of data is both predictable and constant in time, a static data partitioning, computed at the beginning of program execution, is appropriate. Otherwise, some form of adaptive repartitioning may be required to avoid processing-load imbalances.

There are three issues involved in such adaptive partitioning:

- Deciding when to repartition.
- Computing the new partition.
- Redistributing the data in the new partition.

An airflow simulator used to predict weather would require dynamic repartitioning, because areas of computational intensiveness (e.g. storms) are in constant motion. You can compute a new partition for all the data after a predetermined number of simulation time steps, or you can set local criteria for the repartitioning of individual regions. In a particle simulation, you can detect shifts in the distribution of computational work by counting the particles each node is required to keep track of. For differential equations, mathematical methods exist for estimating the computational error in each region. In general, more computational effort should be devoted to regions where the error is greatest. In one common strategy, regions are partitioned more finely when the estimated error exceeds a predetermined value, and more coarsely when the estimated error is especially small. In this way, computational resources are adaptively deployed to maintain a predictable degree of accuracy throughout the computation.

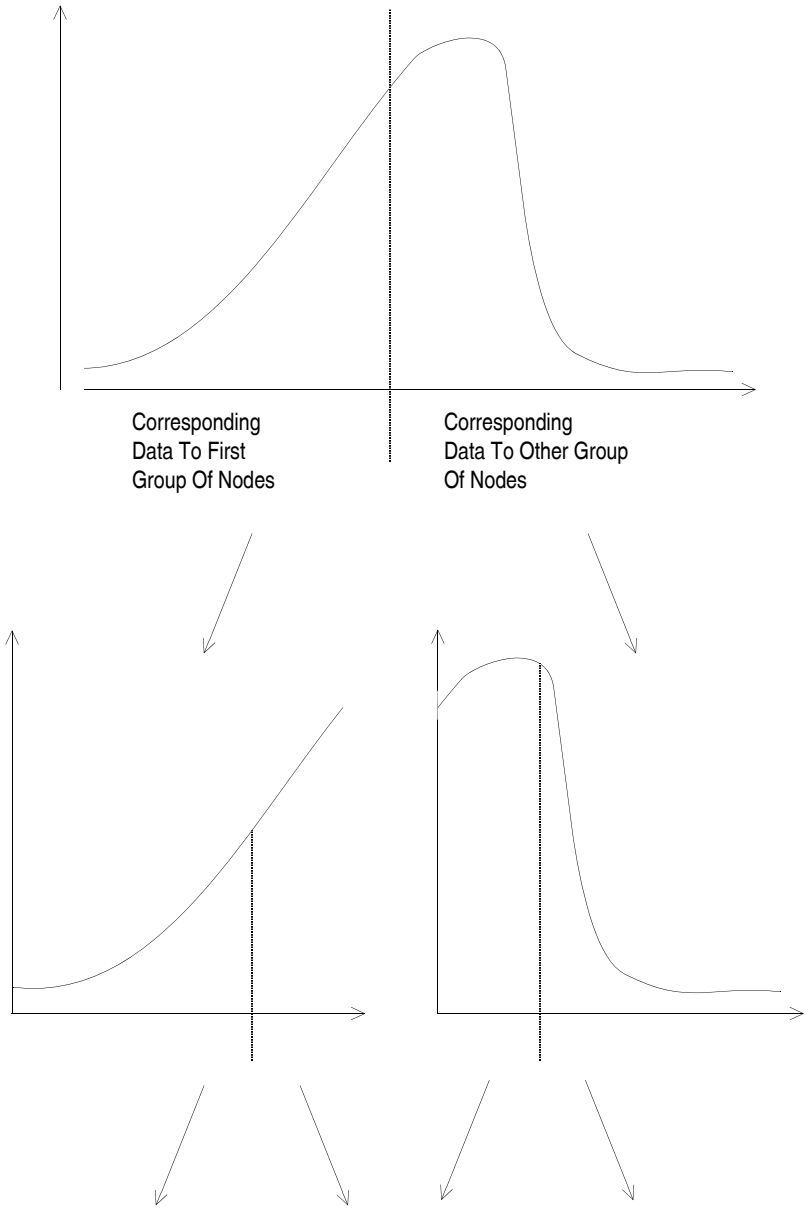
To compute a new partition, you can use the same algorithms that are available for computing a static partition. You simply provide the algorithm with information about the current distribution of computational work. Once the new partition has been computed, data needs to be redistributed among the nodes. The choice of algorithm for redistributing the data depends heavily on the nature of the partition.

Often the calculation and redistribution steps can be interleaved. Consider the airflow simulation example discussed above, in which a room is divided into horizontal slices of varying thickness. Suppose that after some iterations of the main program, the distribution of computational work has changed, and new information has been collected about the density of computational work as a function of height. From this point, there are many ways to proceed. One possibility is the following:

1. Each node is provided a copy of the load-distribution information; all nodes then compute the room height at which half the computational work lies above, and half below.
2. Throughout the simulation, half of the nodes are responsible for the top half of the room, half for the bottom. Data is shifted as necessary from the "top half" nodes to the "bottom half" nodes, or vice-versa, so that each group will have the same total amount of work to do when the main program resumes.
3. The partitioning then proceeds recursively, with each group of nodes—"top half" and "bottom half"—computing the midpoint of its own computational load, shifting data as needed, and then splitting into two subgroups.
4. The computation ends when only one node is left in each subgroup. At this point, the data have been distributed so as to balance the computational work among all the nodes.

This procedure is illustrated in Figure 1-39.

Figure 1-39: Dynamic Repartitioning

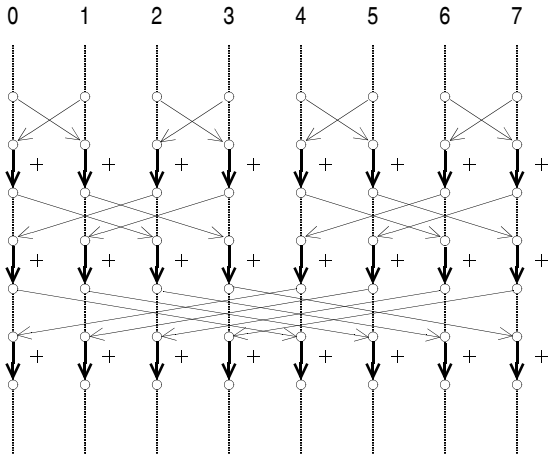


Trees

Using nCUBE 2 parallel library routines, you can easily construct algorithms that perform global computations on trees and related structures, particularly computations that require gathering or broadcasting information to all the nodes. Sequential algorithms based on a divide-and-conquer strategy can often be made parallel by using some kind of tree structure. Since the depth of a tree is typically related logarithmically to the number of nodes, the resulting parallel algorithms can be very efficient.

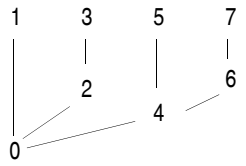
The *globalsum* library routine is an example of such an algorithm. It computes the sum of a collection of numbers, each of which resides on a different node. The routine must be called by the program elements on all nodes. It returns the value of the sum to each node. The operation on a hypercube of eight nodes is illustrated in the time-line diagram below (Figure 1-40):

Figure 1-40: Time-Line Diagram for *Globalsum*



Data is passed from node to node down the communication tree shown in Figure 1-41. At each node, the data from the child nodes is summed. After three communication-and-computation steps ($3=\log_2 8$), node 0 knows the sum of the numbers from all nodes.

Figure 1-41: Communication Tree for *Globalsum*



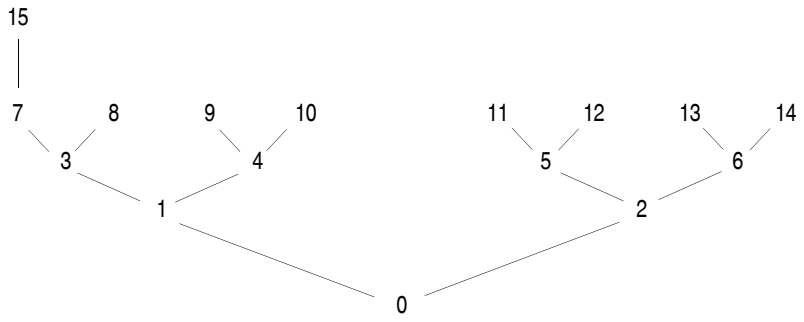
Eight such trees can be traced in the time-line diagram, each with a different node at its root. *Globalsum* therefore returns the same value—the sum of all the numbers—to every node. A collection of interlaced trees like the eight in our example is called a *Banyan net.* Banyan nets can be used—by calling a library routine similar to *globalsum*, or by writing a custom routine—to perform globally any associative operation that works on two numbers or other data items at a time.

The communication tree shown in the diagram above spans the hypercube in such a way that nodes connected in the tree are nearest neighbors in the hypercube. This reduces communication overhead, but not by very much. The message-routing hardware of the nCUBE 2 makes communication between distant nodes almost as fast as that between nearest neighbors. The main advantage of this tree is its conceptual simplicity and ease of programming.

In general, choose a tree topology based on the requirements of your problem and the availability of useful algorithms. Mapping your tree onto the hypercube should be your last step.

For instance, in the communication tree illustrated above, the number of children belonging to a node—the *fanout* of the node—varies from node to node. Node 0 has three children, while node 2 has only one child. Suppose a great deal of memory is required in each node, per child. Then to distribute the memory burden evenly would require a balanced tree structure, like the binary tree illustrated in Figure 1-42:

Figure 1-42: Binary Tree



Of the sixteen nodes in the binary tree, none has more than two children, so the memory burden in our hypothetical example is distributed fairly evenly. Binary trees, moreover, allow certain important functions to be computed easily. For example, the parent of node *N* is the node whose address is the largest integer which does not exceed *(N-1)/2.* On the other hand, a binary tree cannot be mapped into the hypercube in a way that preserves nearest neighbors.

Research literature on parallel programming abounds in special-purpose tree structures and algorithms exploiting them. For example, a near-binary tree has been found that can be mapped onto the hypercube so that neighbors are preserved. This tree controls fanout like a binary tree, but its use requires more complicated algorithms. The following table summarizes the main properties of the tree structures discussed in this section.

Tree Structure	**Maximum Fanout**	**Mapped as Neighbors?**	**Parent of Node N**
Hypercube	Dimension of Hypercube	yes	$N-2^k$, where 2^k is the largest power of 2 that divides N
Binary	2	no	Greatest integer not exceeding (N-1)/2
n-ary	n	no	Greatest integer not exceeding (N-1)/n
Near-Binary	2	yes	Complicated