

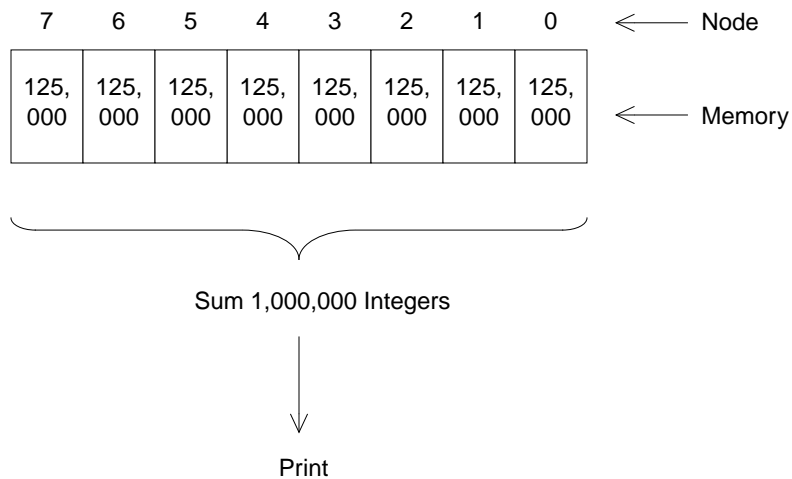
## An Example Parallel Program

To illustrate the way in which parallel programs are conceived and implemented, consider the problem of computing and printing the sum of 1,000,000 integers.

We wish to find an efficient algorithm for solving this problem. We start by observing that the dominant activity, addition, is computation-intensive (rather than memory-intensive, communication-intensive, or I/O-intensive). With this in mind, we allocate the available computation resources to our problem. For example, if we have eight nodes on our nCUBE 2 supercomputer and adequate memory at each node, 125,000 integers can be summed on each node, in parallel. Figure 1-5 shows this data-partition and resource-allocation method in the context of the problem to be solved:

---

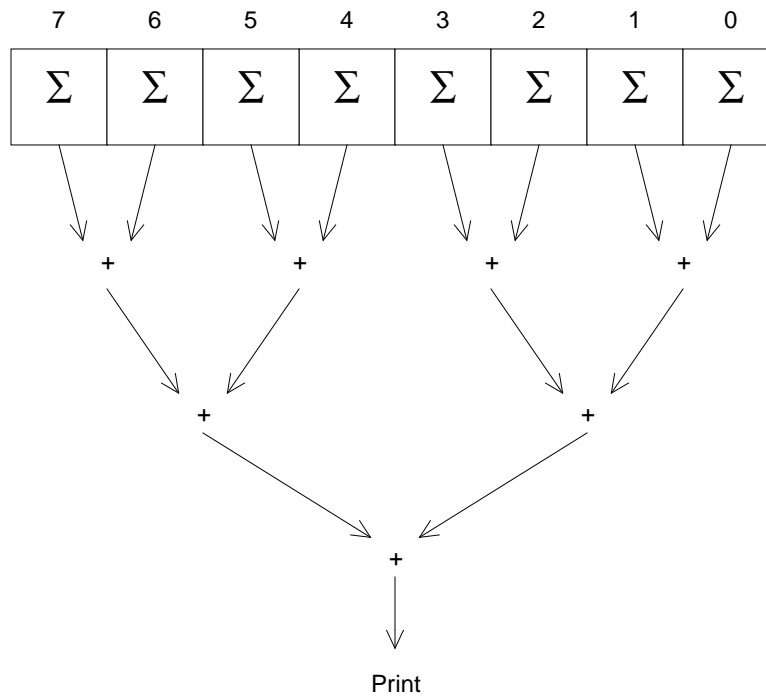
Figure 1-1: Adding 1,000,000 Integers



With our data partitioned equally among the nodes, we next need a method of collecting the results from each node and adding them to the results of all other nodes. One way to do this is with a tree structure that gathers data and computes partial sums in successive stages. Figure 1-6 illustrates such a tree:

---

Figure 1-2: Addition Tree



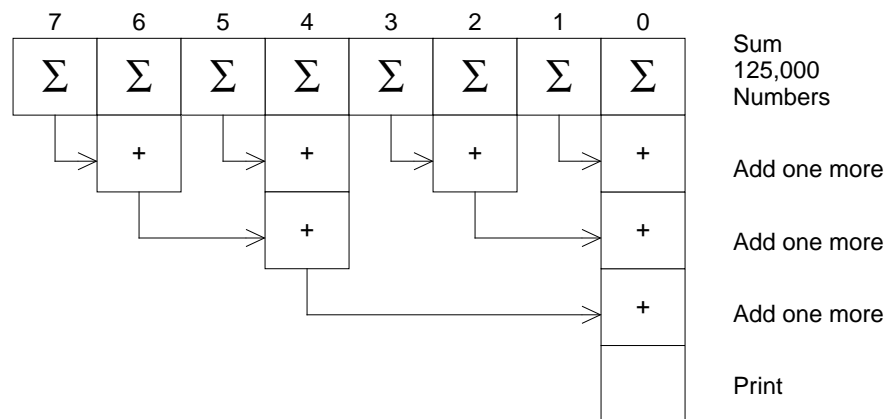
---

Tree structures are often used in parallel programs that, like our example, are based on the principle of divide-and-conquer. Trees are efficient for several reasons. First, they typically have logarithmic depth. In our example, three stages of partial summing are required to gather up the results from eight nodes; for a problem eight times as large, using 64 nodes, only six such stages would be required. Second, trees are easily mapped into a parallel computing environment. Because integer addition is associative—the order of summing does not affect the result—the tree structure will work correctly for our example.

Now that we have chosen an algorithm—the tree structure—for our problem, we look at its performance implications. To do this, we need to understand how the algorithm can be implemented in the parallel environment.

Figure 1-7 restates the problem in a way that corresponds to the activity of eight processing nodes, numbered 0 through 7.

Figure 1-3: Node Activity



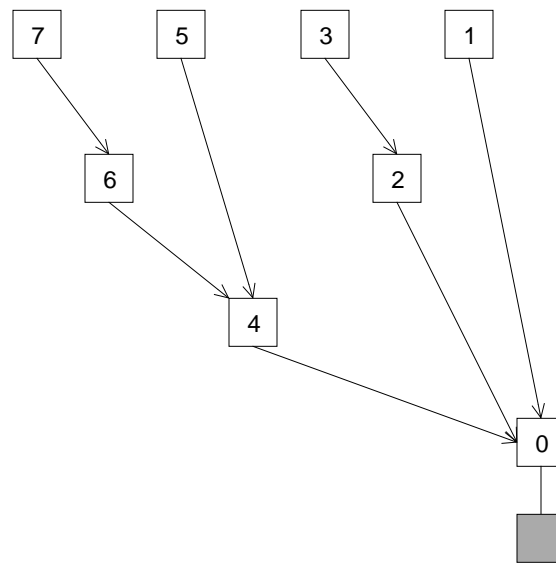
Considering the activity only at node 0 in the above diagram, we see the following order of actions:

- Node 0 sums its 125,000 integers (one-eighth of the 1,000,000 integers). It does this in parallel with all other nodes.
- Node 0 receives Node 1's result and adds it to its own. It does this in parallel with similar summing by Nodes 2, 4, and 6.
- Node 0 receives Node 2's result and adds it to its own. It does this in parallel with similar summing by Node 4.
- Node 0 receives Node 4's result and adds it to its own. It now has the answer.

Figure 1-8 shows this tree structure in the form of a computation-and-communication network. The boxes represent nodes where program elements compute results; the links between nodes represent the communication along the tree branches to the root node:

---

Figure 1-4: Communication Tree



---

In each communication link (represented in the diagram by an arrow), the sending node is a *child* of the receiving node. The core of the parallel program that implements the tree structure and runs, as a compiled program element on each node, looks like this:

```
add 125,000 numbers
for each child node
    read inputs from child node
    add inputs to my current sum
if a parent node exists
    write sum to parent node
else
    print answer
```

This parallel program results in the following flow of activity over time:

1. 125,000 additions at each node, in parallel
2. A four-byte sum transferred in parallel between four pairs of nodes
3. An addition at each of four nodes
4. A four-byte sum transferred in parallel between two pairs of nodes
5. An addition at each of two nodes
6. A four-byte sum transferred between one pair of nodes
7. An addition at one node
8. A print routine

Figure 1-9, called a *time-line diagram*, illustrates this program. The activity of each node (0 through 7) is represented by a vertical time line labelled with the node number. The heavy vertical arrows indicate periods of time during which a node is busy. The oblique arrows indicate the passing of a message from one node to another. Dotted lines indicate idle time, during which a node is waiting for a message.

---

Figure 1-5: Time-Line Diagram

