DISTRIBUTED COMPUTING

A **distributed system** is one in which the processors are less strongly connected. A typical distributed system consists of many independent computers in the same room, attached via network connections. Such an arrangement is often called a **cluster**.

In a distributed system, each processor has its own independent memory. This precludes using shared memory for communicating. Processors instead communicate by sending messages. In a cluster, these messages are sent via the network. Though **message passing** is much slower than shared memory, it scales better for many processors, and it is cheaper. Plus programming such a system is arguably easier than programming for a shared-memory system, since the synchronization involved in waiting to receive a message is more intuitive. Thus, most large systems today use message passing for interprocessor communication.

From now on, we'll be working with a message-passing system implemented using the following two functions.

```
void send(int dst_pid, int data)
```

> Sends a message containing the integer `data` to the processor whose ID is `dst_pid`. Note that the function's return may be delayed until the receiving processor requests to receive the data — though the message might instead be buffered so that the function can return immediately.

```
int receive(int src_pid)
```

> Waits until the processor whose ID is `src_pid` sends a message and returns the integer in that message. This is called a **blocking** receive. Some systems also support a **non-blocking** receive, which returns immediately if the processor

hasn't yet sent a message. Another variation is a `receive` that allows a program to receive the first message sent by any processor. However, in our model, the call will always wait until it receives a message (unless there is already a message waiting to be sent), and the source processor's ID must always be specified.

To demonstrate how to program in this model, we return to our example of adding all the numbers in an array. We imagine that each processor already has its segment of the array in its memory, called `segment`. The variable`procs` holds the number of processors in the system, and `pid` holds the processor's ID (a unique integer between 0 and `procs` − 1, as before).

```
total = segment[0];
for(i = 1; i < segment.length; i++) total += segment[i];

if(pid > 0) { // each processor but 0 sends its total to processor 0
    send(0, total);
} else {      // processor 0 adds all these totals up
    for(int k = 1; k < procs; k++) total += receive(k);
}
```

This code says that each processor should first add the elements of its segment. Then each processor except processor 0 should send its total to processor 0. Processor 0 waits to receive each of these messages in succession, adding the total of that processor's segment into its total. By the end, processor 0 will have the total of all segments.

In a large distributed system, this approach would be flawed since inevitably some processors would break, often due to the failure of some equipment such as a hard disk or power supply. We'll ignore this issue here, but it is an important issue when writing programs for large distributed systems in real life.

Source : http://www.toves.org/books/distalg/index.html#1