Parallel & distributed models

Multiprocessor systems come in many different flavors, but there are two basic categories: parallel computers and distributed computers. These two terms are used with some overlap, but usually a*parallel* system is one in which the processors are closely connected, while a *distributed* system has processors that are more independent of each other.

1.1. Parallel computing

In a **parallel computer**, the processors are closely connected. Frequently, all processors share the same memory, and the processors communicate by accessing this **shared memory**. Examples of parallel computers include the multicore processors found in many computers today (even cheap ones), as well as many graphics processing units (GPUs).

As an example of code for a shared-memory computer, below is a Java fragment intended to find the sum of all the elements in a long array. Variables whose name begin with `my_` are specific to each processor; this might be implemented by storing these variables in individual processors' registers. The code fragment assumes that a variable `array` has already been set up with the numbers we want to add and that there is a variable `procs` that indicates how many processors our system has. In addition, we assume each register has its own `my_pid` variable, which stores that processor's own**processor ID**, a unique number between 0 and `procs` – 1.

```
// 1. Determine where processor's segment is and add up numbers in segment.
count = array.length / procs;
my_start = my_pid * count;
my_total = array[my_start];
for(my_i = 1; my_i < count; my_i++) my_total += array[my_start + my_i];

// 2. Store subtotal into shared array, then add up the subtotals.
subtotals[my_pid] = my_total;                    // line A in remarks below
my_total = subtotals[0];                         // line B in remarks below
for(my_i = 1; my_i < procs; my_i++) {
    my_total += subtotals[my_i];                 // line C in remarks below
}

// 3. If array.length isn't a multiple of procs, then total will exclude some
// elements at the array's end. Add these last elements in now.
for(my_i = procs * count; my_i < array.length; my_i++) my_total += array[my_i
];
```

Here, we first divide the array into segments of length `count`, and each processor adds up the elements within its segment, placing that into its variable `my_total`. We write this variable into shared memory in line A so that all processors can read it; then we go through this shared array of subtotals to find the total of the subtotals. The last step is to take care of any numbers that may have been excluded by trying to divide the array into *p* equally-sized segments.

Synchronization

An important detail in the above shared-memory program is that each processor must complete line A before any other processor tries to use that saved value in line B or line C. One way of ensuring this is to build the computer so that all processors share the same program counter as they step through identical programs. In such a system, all processors would execute line A simultaneously. Though it works, this shared-program-counter approach is quite rare because it can be difficult to write a program so that all processors work identically, and because we often want different processors to perform different tasks.

The more common approach is to allow each processor to execute at its own pace, giving programmers the responsibility to include code enforcing dependencies between processors' work. In our example above, we would add code between line A and line B to enforce the restriction that all processors complete line A before any proceed to line B and line C. If we were using Java's built-in features for supporting such synchronization between threads, we could accomplish this by introducing a new shared variable `number_saved` whose value starts out at 0. The code following line A would be as follows.

```
synchronized(subtotals) {
    number_saved++;
    if(number_saved == procs) subtotals.notifyAll();
    while(number_saved < procs) {
        try { subtotals.wait(); } catch(InterruptedException e) { }
    }
}
```

Studying specific synchronization constructs such as those in Java is beyond this tutorial's scope. But even if you're not familiar with such constructs, you might be able to guess what the above represents: Each processor increments the shared counter and then waits until it receives a signal. The last processor to increment the counter sends a signal that awakens all the others to continue forward to line B.

Shared memory access

Another important design constraint in a shared-memory system is how programs are allowed to access the same memory address simultaneously. There are three basic approches.

- **CRCW: Concurrent Read, Concurrent Write.** Simultaneous reads and writes are allowed to a memory cell. The model must indicate how simultaneous writes are handled.
    - Common Write: If processors write simultaneously, they must write same value.
    - Priority Write: Processors have priority order, and the highest-priority processor's writewins in case of conflict.
    - Arbitrary Write: In case of conflict, one of the requested writes will succeed. But the outcome is not predictable, and the program must work regardless of which processorwins.
    - Combining Write: Simultaneous writes are combined with some function, such as adding values together.
- **CREW: Concurrent Read, Exclusive Write.** Here different processors are allowed to read the same memory cell simultaneously, but we must write our program so that only one processor can write to any memory cell at a time.
- **EREW: Exclusive Read, Exclusive Write.** The program must be written so that no memory cell is accessed simultaneously in any way.
- **ERCW: Exclusive Read, Concurrent Write.** There is no reason to consider this possibility.

In our array-totaling example, we used a Common-Write CRCW model. All processors write to the`count` variable, but they all write an identical value to it. This write, though, is the only concurrent write, and the program would work just as well with `count` changed to `my_count`. In this case, our program would fit into the more restrictive CREW model.

1.2. 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