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 write wins 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 processor wins.
    - 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.

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