A Prelude To:

Principles of Parallel Programming (Chap. 5)

I'm providing copies of chapter 5 of *Principles of Parallel Programming* by Calvin Lin, http://www.cs.utexas.edu/~lin/ and Larry Snyder, http://homes.cs.washington.edu/ snyder/ because I think it provides a great description of the `reduce` and `scan` algorithms. Later, we'll refer to it when we examine tiling matrices and work distribution.

A challenge of using multiple texts is coping with multiple notations. In *Principles of Parallel Programming*, Lin and Snyder use a notation that they call "Peril-L". Here's a quick introduction.

Variables

Peril-L is an imperative programming language. It provides three kinds of variables: local, global, and full-empty as described below:

Local variables: Local variables are written just like variables in C, for example, `x`, `i`, or `asparagus`. A Peril-L program consists of many processes, and each process has its own copy of the local variables. Thus, assigning a value to a local variable in one process has no effect on that variable (or any other variable) in any other process. Local variables can be stored in the local DRAM and caches of the processor that is executing the process. Thus, accesses to local variables are considered as being "fast".

Global variables: Global variables are underlined, for example, `data` or `biggest`. All processes see the same value for the variable; thus, assigning a value to the variable by one process will change the value seen by another process that reads the variable. Reading and writing global variables requires coordination between the processors in a parallel computer. Thus, accesses to global variables are considered as being "slow". Lin and Snyder write λ to denote the amount of time for a global operation. Typically, global operations are a hundred to a million times slower than local ones.

Full-empty variables: These are a special type of global variable. The name of a full-empty variables is underlined and followed by an apostrophe, for example, `buf`$'$. Full-empty variables combine a simple form of process coordination with reading and writing of the memory location. Full-empty variables are initially "empty".

- A write to an empty full-empty variable sets the value of the variable and marks the variable as full.
- A write to a full full-empty variable blocks until the variable is marked as empty (by some other process reading the current value).
- A read to a full full-empty gets the value of the variable and marks the variable as empty.
- A read to an empty full-empty blocks until the variable is marked as full (by some other process writing a value to the variable).

Full-empty variables are global variables. Thus, reads and writes of these variables are "slow" – they take λ time units plus any time stalling waiting for the variable to become full (or empty).

Processes and Synchronization

In Peril-L, processes are created by a parallel for loop construction called `forall`. Synchronization between processes is provided by `exclusive` blocks and `barrier` statement.

`forall` *`j`* `in` *`range`*: A new process is created for each value of j in *range*. The variable j must be a local variable – each process has its copy of j set to the value that it got from range. For example:

```
forall(j in range(0..3))
  x[j] = j*j;
```

Creates four processes, one with `j = 0`, one with `j = 1`, one with `j = 2`, and one with `j = 3`. Each processes sets one element of the global array, `x` to the square of the value of `j` for that process.

`exclusive { body }`: Only one process may execute the code in *body* at a time. If a process arrives at an `exclusive` block when another process is already executing the block, the newly arriving process waits. If multiple processes arrive at the same time, or multiple are waiting when a process leaves the exclusive block, a non-deterministic choice is made about which process enters the block next.

`barrier` All processes must reach the `barrier` before any continue beyond it.

Reduce and Scan

We'll talk about `reduce` and `scan` next week – that's why I handed out this chapter. Lin and Snyder include these operations in the Peril-L notation. For any associative operator (e.g. `+`, `max`, `*`, ...), *op*:

`op``/ `*`expr`*: the reduce version of *op*. *`op`*`/ `*`expr`* evaluates *expr* in each process. Note that this can produce a different value in each process because they have different values for their local variables. These values for *expr* are combined using the operation *op*. The result is the value of the reduce expression for every process – i.e. all processes get the same value. For example, if each process has computed a result in local variable `myTotal` (e.g. how many `3`s are in that process's part of a large array), then

```
forall(j in range(0..3))
  grandTotal = +/ myTotal;
```

computes the sum of all of these subtotals, and assigns it to the local variable `grandTotal` in each process.

`op``\ `*`expr`*: the scan version of *op*. *`op`*`\ `*`expr`* evaluates *expr* in each process and combines the local values with the operation *op*. Each process gets the the value produced by combining the results from all processes with indices less than or equal to its own. For example,

```
forall(j in range(0..3))
  foo = +\ j*j;
```

would result in `foo` in process 0 having the value 0 (i.e. `0*0`); `foo` in process 1 having the value 1 (i.e. `0*0 + 1*1`); `foo` in process 2 having the value 5 (i.e. `0*0 + 1*1 + 2*2`); and `foo` in process 3 having the value 14 (i.e. `0*0 + 1*1 + 2*2 + 3*3`).

Note that performing a reduce or scan operation also has the effect of acting like a barrier: all processes must participate in the reduce or scan before any can continue.

Everything Else

The Peril-L notation is based on C.