Tools for Parallel Programming

Parallel Programming can be a complex affair. Fortunately, we have some models which provide us the desired parallel functionality.

We'll cover the following

The **threads model** of parallel programming is one in which a single process (a single program) can spawn multiple, concurrent “threads” (sub-programs). Each thread runs independently of the others, although they can all access the same shared memory space (and hence they can communicate with each other if necessary). Threads can be spawned and killed as required, by the main program.

A challenge of using threads is the issue of collisions and race conditions, which can be addressed using synchronization. If multiple threads write to (and depend upon) a shared memory variable, then care must be taken to make sure that multiple threads don’t try to write to the same location simultaneously. The wikipedia page for race condition has a nice description (an an example) of how this can be a problem. There are mechanisms when using threads to implement synchronization, and to implement mutual exclusivity (mutex variables) so that shared variables can be locked by one thread and then released, preventing collisions by other threads. These mechanisms ensure threads must “take turns” when accessing protected data.

POSIX Threads (Pthreads)

POSIX Threads (Pthreads for short) is a standard for programming with threads, and defines a set of C types, functions and constants.

More generally, threads are a way that a program can spawn concurrent units of processing that can then be delegated by the operating system to multiple processing cores. Clearly the advantage of a multithreaded program (one that uses

multiple threads that are assigned to multiple processing cores) is that you can

achieve big speedups, as all cores of your CPU (and all CPUs if you have more than one) are used at the same time.

Here is a simple example program that spawns 5 threads, where each one runs the `myFun()` function:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define NTHREADS 5

void *myFun(void *x)
{
  int tid;
  tid = *((int *) x);
  printf("Hi from thread %d!\n", tid);
  return NULL;
}

int main(int argc, char *argv[])
{
  pthread_t threads[NTHREADS];
  int thread_args[NTHREADS];
  int rc, i;

  /* spawn the threads */
  for (i=0; i<NTHREADS; ++i)
    {
      thread_args[i] = i;
      printf("spawning thread %d\n", i);
      rc = pthread_create(&threads[i], NULL, myFun, (void *) &thread_args[i]);
    }

  /* wait for threads to finish */
  for (i=0; i<NTHREADS; ++i) {
    rc = pthread_join(threads[i], NULL);
  }

  return 1;
}
```

```
plg@wildebeest:~/Desktop$ gcc -o go go.c -lpthread
plg@wildebeest:~/Desktop$ ./go
spawning thread 0
spawning thread 1
Hi from thread 0!
spawning thread 2
Hi from thread 1!
spawning thread 3
Hi from thread 2!
spawning thread 4
Hi from thread 3!
Hi from thread 4!
```

Parallelizing for loops with OpenMP

Parallelizing for loops is really simple (see code below). By default, loop iteration counters in OpenMP loop constructs (in this case the `i` variable) in the for loop are set to `private` variables.

```
// gcc -fopenmp -o go go.c
// ./go

#include <stdio.h>
#include <omp.h>

int main(int argc, char **argv)
{
  int i, thread_id, nloops;

#pragma omp parallel private(thread_id, nloops)
  {
    nloops = 0;

#pragma omp for
    for (i=0; i<1000; ++i)
      {
    ++nloops;
      }

    thread_id = omp_get_thread_num();

    printf("Thread %d performed %d iterations of the loop.\n",
       thread_id, nloops );
  }

  return 0;
}
```

```
plg@wildebeest:~/Desktop$ gcc -fopenmp -o go go.c
plg@wildebeest:~/Desktop$ ./go
Thread 4 performed 125 iterations of the loop.
Thread 7 performed 125 iterations of the loop.
Thread 2 performed 125 iterations of the loop.
Thread 6 performed 125 iterations of the loop.
Thread 5 performed 125 iterations of the loop.
Thread 0 performed 125 iterations of the loop.
Thread 3 performed 125 iterations of the loop.
Thread 1 performed 125 iterations of the loop.
```

Critical Code

Using OpenMP you can specify something called a “critical” section of code. This is code that is performed by all threads, but is only performed **one thread at a time** (i.e. in serial). This provides a convenient way of letting you do things like updating a global variable with local results from each thread, and you don’t have to worry about things like other threads writing to that global variable at the same

time (a collision).

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char *argv[])
{
    int i, thread_id;
    int glob_nloops, priv_nloops;
    glob_nloops = 0;

    // parallelize this chunk of code
    #pragma omp parallel private(priv_nloops, thread_id)
    {
        priv_nloops = 0;
        thread_id = omp_get_thread_num();

        // parallelize this for loop
        #pragma omp for
        for (i=0; i<100000; ++i)
        {
            ++priv_nloops;
        }

        // make this a "critical" code section
        #pragma omp critical
        {
            printf("Thread %d is adding its iterations (%d) to sum (%d), ",
                   thread_id, priv_nloops, glob_nloops);
            glob_nloops += priv_nloops;
            printf(" total nloops is now %d.\n", glob_nloops);
        }
    }
    printf("Total # loop iterations is %d\n",
           glob_nloops);
    return 0;
}
```

For more information about collisions, synchronization, mutexes, etc, check out one of the many sources of documentation about Pthreads, e.g. here: Mutex Variables.

We do have several other platforms which make the parallel process more automated. One of them is OpenMP, which we will see shortly.