University of Amsterdam

Lab 1

Multithreading with OpenMP and pthreads

November 12, 2018

Students:
S.R.W. van Kampen
11874716

R.A.J. Wacanno
11741163

Group:
B

Course:
Concurrency en Parallel Programmeren

1 Introduction

2 Method

2.1 Wave equation simulation

2.1.1 Implementation using pthreads

In the `simulate` function, we split the amount of amplitude points of the wave equation simulation into n parts. Then, we create n worker threads, which each compute one of those parts. The thread creation code is shown in listing 1, and the worker computation code is shown in listing 2.

Listing 1: Thread creation code

```
pthread_t thread_ids[num_threads];
struct worker_data d[num_threads];
data = d;

// First, slice up the input data into num_threads pieces.
int chunk_size = ceil((double)i_max / num_threads);
int items_left = i_max;

for (int i = 0; i < num_threads; ++i)
{
    data[i].i_min   = (i * chunk_size) + 1;
    data[i].size    = (i + 1 == num_threads) ? items_left - 2 :
    chunk_size;
    items_left -= chunk_size;

    // Then, create the worker threads
    pthread_create(&thread_ids[i], NULL, worker, &data[i]);
}
```

Listing 2: Worker computation code

```
void *worker(void *ptr)
{
    struct worker_data *my_data = (struct worker_data *)ptr;
    while (running)
    {
        /* Compute values for the next array. */
        for (int i = my_data->i_min;
             i < (my_data->i_min + my_data->size);
             ++i)
        {
            double left    = current_array[i - 1];
            double right   = current_array[i + 1];
            double current = current_array[i];
            double old     = old_array[i];
            next_array[i]  = 2 * current - old +
                             WAVE_CONSTANT *
                             (left - (2 * current - right));
        }

        /* Synchronisation and simulation ending code */
    }

    pthread_exit(NULL);
}
```

As can be seen in the thread creation code, each worker receives a data pointer containing information about the subset of the array indices the worker is responsible for. While running, the worker does the computation described above. When the computation is finished, the worker takes a lock and increments the done counter. When this done counter reaches the number of threads, the last worker knows that the rest of the workers are finished, and the next timestep can begin. The last worker rotates the old, current and new arrays, and notifies the other threads that they can continue. Hereafter, the lock is released, and the threads continue. This is illustrated in listing 3.

Listing 3: Worker synchronisation code

```
/* Take a lock, which protects both n_done
 * and the old, current and next pointers. */
pthread_mutex_lock(&mutex);
n_done++;
if (n_done == num_threads)
{
    // We're the last worker, so we have to rotate the arrays
   and go to the next timestep.
    n_done = 0;
    rotate(); /* old, current, next = current, next, old */
    t++;
    if (t != t_max)
    {
        /* This is not the end of the simulation.
         * Notify the other threads that they can
         * continue. */
        pthread_cond_broadcast(&rotated);
    }
```

```
18      else
19      {
20          /* We've arrived at the end of the simulation.
21           * Tell the threads to stop running and exit.
22           * The results are returned from simulate() */
23          end_simulation();
24      }
25  }
26  else
27  {
28      // If we're not the last worker, wait until the last worker
        has completed the rotation.
29      pthread_cond_wait(&rotated, &mutex);
30  }
31  pthread_mutex_unlock(&mutex);
```

2.1.2 Implementation using OpenMP

The OpenMP implementation is extremely simple - the algorithm is written sequentially, OpenMP is told how many threads to use, and a pragma is added which tells OpenMP to parallelize the inner loop. The code inside `simulate` is as follows:

Listing 4: OpenMP wave simulation code

```
1  omp_set_num_threads(num_threads);
2
3  for (int t = 0; t < t_max; ++t)
4  {
5      #pragma omp parallel for
6      for (int i = 1; i < (i_max - 1); ++i)
7      {
8          double left    = current_array[i-1];
9          double right   = current_array[i+1];
10         double current = current_array[i];
11         double old     = old_array[i];
12         next_array[i]  = 2 * current - old + WAVE_CONSTANT *
13                          (left - (2 * current - right));
14     }
15
16     double* tmp    = old_array;
17     old_array      = current_array;
18     current_array = next_array;
19     next_array     = tmp;
20 }
21
22 /* Return the final results. */
23 return current_array;
```

2.2 Sieve of Eratosthenes

The implementation of the Sieve of Eratosthenes algorithm sought to exploit the multi-threading possibilities of pthreads.

Listing 5: Put function

```
1  int put(queue_t *q, int n)
```

```
{
    pthread_mutex_lock(&q->mut);
    if (q->size == q->cap)
    {
        pthread_cond_wait(&q->con, &q->mut);
    }

    // Insertion logic

    if ((q->size - 1) == 0)
    {
        pthread_cond_signal(&q->con);
    }
    pthread_mutex_unlock(&q->mut);
    return 0;
}
```

Listing 6: Get function

```
int get(queue_t *q)
{
    pthread_mutex_lock(&q->mut);
    if (q->size == 0)
    {
        pthread_cond_wait(&q->con, &q->mut);
    }

    // Retrieval logic

    if ((q->size + 1) == q->cap)
    {
        pthread_cond_signal(&q->con);
    }
    pthread_mutex_unlock(&q->mut);
    return n;
}
```

3 Results

n primes	Time in s
100	0.010
500	0.063
1000	0.158
2500	1.011
3966	2.318

Table 1: Run time of prime calculation for n primes.

4 Discussion

4.1 Conclusion