F20DP Submission

Coursework 2

14th April 2022

Christopher Murdoch – H00272991

Authorship

My coursework partner did not contribute to the assignment as they're currently seeking temporary suspension of studies. As I was only informed of this two days prior to the deadline, I was unable to try and complete any of the OpenCL implementation myself.

As such, all code and all report material – MPI and OpenCL related – is written by me.

1. Introduction

The purpose of this project was to implement the Euler Totient Sum function in MPI and OpenCL, comparing the performance and design of both implementations. Unfortunately, my coursework partner dropped out of the course late in the semester, so the OpenCL part of the coursework could not be implemented.

In this report, I will detail the implementation of the MPI part, how the OpenCL part could have been implemented, and how the programming models compare.

The MPI application was developed on a personal windows machine, equipped with 6 cores and 12 threads. Using MPICC on WSL to compile the program, the application could be tested locally, even though the Robotarium was inaccessible.

The main learning objective of this coursework was to explore lower-level technologies as a method of parallelisation, in contrast to the higher-level OpenMP and Haskell used in coursework one.

4. Programming model comparison

4.1 MPI Initial Architecture

The MPI implementation initially allocated each process one contiguous segment of the Euler Totient range to calculate, reducing the results once finished. This worked fine but, like static scheduling in OpenMP, some processes were finishing long before others. To resolve this issue and use a system more similar to OpenMP's dynamic scheduling, I switched to an authoritative work distribution system.

4.2 MPI Revised Architecture

The final MPI implementation developed uses an authoritative process architecture. When running on two or more processes, one *controller process* is tasked with the exclusive responsibility of dealing out *tasks* – Euler Totient calculations – to worker processes, finally reducing all computations into one consolidated output when no more tasks are available.

All processes but one are designated as *worker processes*, tasked with repeatedly requesting and completing work from the *controller process*. The workers continually sum the results of their calculations and, when signalled to stop, use a reduction function to return the total sum of all of their calculations.

This method of dealing out work, task by task, means that the work can be split up more evenly, and works in a very similar way to OpenMP's dynamic scheduling. Furthermore, an application parameter can be set, specifying the number of tasks allocated to a worker each request. This lessens the overhead of constant communication for every individual task.

4.3 MPI Drawback

Using this architecture has one primary drawback. Because the authoritative process is only distributing tasks and not calculating them, an application spread across N processes only has N-1 processes computing Euler Totients, reducing the overall performance.

This is not a huge issue, however. Although the authoritative process is not performing calculations, it's also using a tiny fraction of the system's processing time. Users can allocate one more process than the system *should* support, knowing that the authoritative process is sleeping for the majority of the time.

4.4 OpenCL Theory

GPUs are special-purpose processors designed to work on massively parallel problems. Although GPUs can have thousands of cores, they aren't as fast as CPU cores and must use a large degree of synchronisation to work with high efficiency.

Cores on the GPU don't have the same ability as those on the CPU to communicate. They can't talk to each other and redistribute tasks if one has too many. All synchronisation between GPU cores to efficiently split tasks must be defined statically – before runtime.

4.5 OpenCL Architecture Theory

We could calculate the Euler Totient Sum of a range by mapping each value in the range to a thread on the GPU, writing the result to an array, and finally summing the array in sequential C. However, the process of creating and destroying threads for every single value in the range introduces a lot of unnecessary overhead.

Instead, the number of threads can be tailored for the device they're running on. In order to map each of these threads to calculations in the Euler Totient Range, the threads can loop through the range in a predefined manner.

One way of having threads loop through the range is to allocate each one a contiguous chunk of the range, moving along one step at a time and writing the output to an array. This is often referred to as a *Monolithic Kernel*.

One issue with this is that the threads can't benefit from memory coalescence: *warps optimising threads writing to contiguous chunks of memory, grouping them all into one operation*. Another issue – specific to this problem – is that higher values of the range take longer to calculate. By allocating the highest X number of values of the range to one thread, that thread will far exceed the runtime of any other, bottlenecking the application.

The solution to both of these problems is to use a *Grid Stride kernel*. By having each thread move along the range by X places of the range each time, threads in warps are all accessing contiguous chunks of memory, and the higher values of the range are spread among multiple threads. This means that warps can run faster, and the high-computation-time tasks are spread more evenly among threads.

4.6 Technology Comparison

Implementing MPI and comparing it with the previously implemented OpenMP application, I am confident that MPI is a very suitable technology to calculate the Euler Totient Sum of a range. The MPI application performed marginally faster than the OpenMP implementation which itself was significantly faster than Haskell.

Although not implemented, I don't believe that OpenCL is a particularly suitable technology to apply to this problem for two reasons:

1. As far as I can tell, there is no way to calculate the Euler Totient Sum using SIMD. Below the point of parallelism (calculating the Euler Totient for one value), there is a high amount of branching and the same calculation for different values takes significantly different amounts of time. This branching renders efficient SIMD impossible, which is where the majority of GPU acceleration gets its performance.
2. Unlike MPI, OpenCL threads can't easily communicate with each other. This means that tasks can't be intelligently allocated based on load (dynamic scheduling), and one thread with high-duration tasks can bottleneck the whole application.

As the OpenCL wasn't implemented, I can't be certain that there is no way to achieve these speedups. With more time, I would be able to research and test these assumptions.

For these reasons, I believe that CPU-based technologies such as MPI are more appropriate for this problem than GPU-based ones like OpenCL. Although GPUs can achieve very high speed up for certain tasks, Euler Totient Sum does not seem to lend itself well to massively parallel, SIMD-accelerated architecture.

5. Appendix

MPI

Author: Christopher Murdoch

Repository: https://github.com/ChrisJMurdoch/F20DP-CW2

The code will also be submitted in a ZIP file with this submission.

There are instructions for building and running the application in the README.

MPI implementation appendices

As detailed in section 4.2, the MPI implementation uses one authoritative process to distribute tasks to worker processes as they become available. The timeline of events for each task (Euler Totient Computation) is as follows:

1. The controller process loops through each value of the range, each time waiting for a message from any worker process (Line 37). If tasks are chunked together, the loop strides using the chunk size.
2. A worker process sends its own rank to the controller, signalling it's ready for a task (Line 82).
3. The controller receives the worker's rank and sends back the starting value of the range to be calculated (Line 40).
4. The worker receives this start value of the range and sums the Euler Totient for each value in the chunk (Line 96). The chunk size determines how many values after the start the worker calculates.
5. This process continues until the controller has no more tasks. At that point, the controller will tell all workers to stop (Line 54) and reduce their sums into one consolidated sum (Lines 60 & 101), equal to the the Euler Totient Sum of the range.

As the point of parallelism for MPI was the same as OpenMP, the mathematical implementation of the Euler Totient function and relevant optimisations were reused from OpenMP in part one.

The following snippets show the code for the authoritative process and the code for the worker processes:

Authorititive *controller* process (rank zero):

```
/**
 * Main execution path for authoritative process
 * @param min minimum value for calculation range
 * @param max maximum value for calculation range
 */
long controller(long min, long max)
{
    // Distribute tasks until depleted
    for (int i=min; i<=max; i+=TASK_PACKET_SIZE)
    {
        // Get work request from any worker
        int workerRank;
        MPI_Status status;
        MPI_Recv(&workerRank, 1, MPI_INT, MPI_ANY_SOURCE, TAG_WORK_REQUEST, MPI_COMM_WORLD, &status);

        // Send work details back to worker
        MPI_Send(&i, 1, MPI_INT, workerRank, TAG_WORK_BRIEF, MPI_COMM_WORLD);
    }

    // Tell all workers to finish
    int processes;
    MPI_Comm_size(MPI_COMM_WORLD, &processes);
    for (int i=1; i<processes; i++)
    {
        // Get work request from any worker
        int workerRank;
        MPI_Status status;
        MPI_Recv(&workerRank, 1, MPI_INT, MPI_ANY_SOURCE, TAG_WORK_REQUEST, MPI_COMM_WORLD, &status);

        // Tell worker to finish
        MPI_Send(&SIGNAL_COMPLETE, 1, MPI_INT, workerRank, TAG_WORK_BRIEF, MPI_COMM_WORLD);
    }

    // Collect worker sums
    int const reductionInput = 0; // Required for reduction
    int sum;
    MPI_Reduce(&reductionInput, &sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    return sum;
}
```

Worker processes (rank one+):

```
64  /**
65   * Main execution path for worker processes
66   * @param min minimum value for calculation range
67   * @param max maximum value for calculation range
68   */
69  void worker(int min, int max)
70  {
71      // Get MPI rank
72      int rank;
73      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
74
75      // Sum up each calculation to reduce once finished
76      int sum = 0;
77
78      // Work until done
79      while (1)
80      {
81          // Send task request
82          MPI_Send(&rank, 1, MPI_INT, 0, TAG_WORK_REQUEST, MPI_COMM_WORLD);
83
84          // Recieve task details
85          int taskMin;
86          MPI_Status status;
87          MPI_Recv(&taskMin, 1, MPI_INT, 0, TAG_WORK_BRIEF, MPI_COMM_WORLD, &status);
88
89          // Check done
90          if (taskMin==SIGNAL_COMPLETE)
91              break;
92
93          // Perform work on task range
94          int taskMax = max<(taskMin+TASK_PACKET_SIZE-1) ? max : (taskMin+TASK_PACKET_SIZE-1);
95          for (int i=taskMin; i<=taskMax; i++)
96              sum += eulerTotient(i);
97      }
98
99      // Return calculation sum
100     int reductionOutput; // Required for reduction
101     MPI_Reduce(&sum, &reductionOutput, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
102 }
```

Additionally, this is the code run when only one process is allocated:

```
104 /**
105  * Main execution path when only one process has been allocated
106  * Controller-Worker method only works with a minimum of 2 processes
107  * @param min minimum value for calculation range
108  * @param max maximum value for calculation range
109  */
110 long sequential(long min, long max)
111 {
112     // Sum totient for each value in given range
113     int sum = 0;
114     for (int i=min; i<=max; i++)
115         sum += eulerTotient(i);
116     return sum;
117 }
```