SOFT354 Parallel Computation and Distributed Systems Coursework

Lukas Ngo, BSc (Hons) Computing & Games Development Final Year Undergraduate, School of Computing and Mathematics, University of Plymouth

Introduction

This project examines the effectiveness of using parallelisation to solve a mathematical problem, how well it scales across multiple threads and processes and what limitations there may be. For the purpose of this experiment this project tries to calculate PI π using CUDA and MPI. Number π is a mathematical constant that can be calculated by many different algorithms. The project uses different approaches / algorithms to calculate π both in series and parallel. There may be more efficient and accurate algorithms to calculate π, however the purpose of this project is not to find the most accurate or fastest algorithm but to determine how well the parallel implementation performs against the serial algorithm and how well the algorithm scales with multiple threads and processes.

PI calculation was chosen for this project because it's a simple mathematical problem that can be parallelised across as many threads or processes as we want with consideration to limits of the hardware used. The idea behind the project is that similar implementation could be used as a benchmark to test CPU and GPU computing performance. Serial implementation of π calculation algorithm with high number of iterations can be used to calculate the single core performance of a processing unit which then can be compared with other processing units to determine which one has higher single core performance. MPI parallel implementation could be used in similar way to determine multi thread performance of different processors.

Implementation

Both CUDA and MPI implementations were designed using Foster's Design Methodology [1]. The implementation meets most but not all the criteria of Foster's design methodology. Following section will describe how the implementations comply with this design methodology.

- Partitioning

Through decomposition the problem of π calculation was reduced into smaller tasks which can be computed on their own. This allows for parallelisation of the algorithm as each thread can deal with a single primitive task - calculate a portion of the problem.

Table 1 Breakdown of Foster's partitioning criteria

Criteria	Criteria met?

The partition defines at least an order of magnitude more tasks than there are processors in your target computer.	Yes – the number of tasks to calculate a partial result can be set arbitrarily independent of how many processors we have available.
Redundant computations and data storage are avoided as much as possible.	Yes – each thread only computes a part of the problem which has been reduced to a simple mathematical operation. No additional data is required to be stored.
Primitive tasks are roughly the same size.	Yes – Each primitive task size is calculated based on the amount of threads or processes available and the workload is distributed evenly among all.
The number of tasks is an increasing function of the problem size.	No – number of tasks is dependant on how many threads or processes are available and increasing the problem size doesn't increase the number of tasks but only increases the size of individual tasks.
Identified several alternative partitions.	Yes – Problem could be divided into a predefined number of tasks which could be distributed among threads / processes, however this is unlikely to offer any increase in performance and therefore it was chosen to divide the problem based on the amount of threads / processes available.

Each task computes a part of the problem and completes the following steps:

1. Enter a loop
2. Calculate partial sum by performing a summing operation
3. Store the calculated partial result into a vector
4. Iterate through the loop until finished

- Communication

Tasks don't perform any local communication as they don't need values from other tasks, instead all tasks perform global communication where they contribute the computed data into one vector. After all computation threads are finished, the data from the vector are summed outside the kernel on the host. This applies for CUDA implementation only. MPI implementation is similar but instead of saving partial results into a vector, the result is added to a double value and the result is calculated by calling MPI_Reduce() function.

Table 2 Breakdown of Foster's communication criteria

Criteria	Criteria met?
All tasks perform the same number of communication operations	Yes – each task performs the same amount of communication – storing a partial result into a vector
Each task should communicate only with a small number of neighbours	N/A - tasks don't need to communicate with other tasks, they compute the data independently
The communication operations should be able to proceed concurrently	Yes – tasks don't need to wait for other tasks as they write into a different part of the vector and therefore can proceed concurrently.
Tasks can perform their computations concurrently	Yes – each task is working on its own part of the problem and can compute independently of other tasks

- Agglomeration

The algorithm implementation both in CUDA and MPI has not been agglomerated to reduce the number of tasks, as the primitive tasks are well suited for parallelisation and each thread (CUDA) or process (MPI) works with the same problem size. Tasks don't exchange data with each other therefore grouping them together would not lower communication overhead.

Table 3 Breakdown of Foster's agglomeration criteria

Criteria	Criteria met?
Agglomeration should reduce communication costs by increasing locality	N/A – agglomeration would not affect communication costs because threads don't communicate with each other
If agglomeration has replicated computation, the benefits of this replication should outweigh its costs	N/A – agglomeration has not replicated any computation
If agglomeration replicates data, it should not compromise the scalability of the algorithm	N/A – agglomeration does not replicate any data. Any problem size or processor count can be used.
Agglomeration should produce tasks with similar computation and communication costs	Yes – all tasks have the same computation and communication costs as they work on an equal part of the problem
The number of tasks can still scale with problem size	Yes – number of tasks depends on number of threads and processes and can be scaled independently of the problem size
There is sufficient concurrency for current and future target computers	Yes – CUDA implementation of the algorithm uses a set number of blocks and blocks per thread which may not work on older graphics cards, however

	the implementation should allow for sufficient concurrency for the target graphics card (GTX 1080ti) and any newer cards with same or higher CUDA capabilities. MPI implementation can use any number of processes.
The number of tasks cannot be made any smaller without introducing load imbalances, increasing software engineering costs, or reducing scalability	Yes – current number of tasks matches the number of threads or processes available and therefore reducing the number of tasks would result in decreased performance
The trade-of between the chosen agglomeration and the cost of modification to existing sequential code is reasonable	N/A

- Mapping

Both CUDA and MPI implementation of the parallel algorithm are based on the same premise. Each process in MPI or thread in CUDA computes a small part of the problem in parallel.

  - CUDA implementation
    - Predefined number of blocks and threads per block is used to calculate the number of parallel tasks [5].
    - Problem size N is a constant – however this can be changed by editing the source code
    - Problem size is divided by the number of threads and each thread will perform almost the same amount of computation
    - Each thread will perform a simple math operation and store the result in a vector, this will loop until the computation is finished
    - Once all threads are finished with the computation the data from the device is copied to the host and then the host computes the result
  - MPI implementation
    - Arbitrary number of processes can be chosen before the program is run, the number of processes is equal to the number of parallel tasks
    - Same as in CUDA implementation, problem size N is a constant but can be changed by editing the source code
    - Problem size is divided by the number of processes and each process will perform almost the same amount of computation
    - Each process will perform a simple math operation and store the result in partial sum double value
    - Once each process has finished its computation the MPI_Reduce() method is called to sum the partial results from each thread

- Implementation CUDA

  CUDA parallel algorithm calculates π as described in previous section (Mapping – CUDA implementation)

  This implementation is a reworked and modified version of algorithm described by Nakano A. [2]

```
//kernel to run pi calculation in parallel
__global__ void pi_calc_kernel(double *sum, int n, double step_size, int num_thread, int num_block) {
  int threadId = blockIdx.x * blockDim.x + threadIdx.x; //calculate thread id
  double partSum; //double value to store partial result
  for (int i = threadId; i< n; i += num_thread*num_block) { //each thread will calculate a part of the problem
    partSum = (i + 0.5)*step_size;
    sum[threadId] += 4.0 / (1.0 + partSum*partSum); //store the calculated partial value into sum vector
  }
}
```

```
pi_calc_kernel << <blocksPerGrid, threadsPerBlock >> > (sum_Device, N, step_size, NUM_THREAD, NUM_BLOCK);

cudaMemcpy(sum_Host, sum_Device, memSize, cudaMemcpyDeviceToHost); //copy data from device to host
for (int i = 0; i < NUM_THREAD*NUM_BLOCK; i++) //calculate the sum of all elements in sumhost vector
{
  gpu_result += sum_Host[i];
}
gpu_result *= step_size; //part of PI calculaton equation
```

Figure 1 CUDA Implementation

- Implementation MPI

  MPI parallel algorithm calculates π as described in previous section (Mapping – MPI implementation)

  Same as CUDA implementation, it is a reworked and modified version of the same Nakano's algorithm used in CUDA

```
for (int i = rank + 1; i <= n; i += size) { //equation to calculate PI in parallel
  x = step_size * ((double)i - 0.5);
  sum += 4.0 / (1.0 + x * x);
}
partSumParallel = step_size * sum;

MPI_Reduce(&partSumParallel, &parallel_result, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

Figure 2 MPI Implementation

- Implementation serial

  Serial implementation of PI calculation algorithm used in both the CUDA and MPI projects [3]

```
partSumSerial = 0;
for (int i = 0; i < n; i++)
  if (i % 2 == 0)
    partSumSerial += 1 / (2.0 * i + 1);
  else
    partSumSerial -= 1 / (2.0 * i + 1);
serial_result = 4 * partSumSerial;
```

Figure 3 Serial Implementation

Evaluation

CUDA

An experiment was conducted on CUDA implementation to evaluate the performance of the algorithm. The test measured how long it takes to calculate π using both parallel and serial algorithms. Time for parallel part of the code includes the time it takes to allocate the memory on the device. Each step in the graph (Figure 4) represents the problem size – the N value times 10^(6) on the horizontal axis and the computation time in seconds on the vertical axis. The test uses the same number of threads (same number of blocks * threads per block) in this case 8192 threads and the problem size increases.

The timing for parallel part of the code was performed to measure from the allocation of the memory on the device and invocation of the kernel to copying the data from device to host and calculating the result on the host. The timing for the serial part measured the time to calculate the solution on the host.

The results show that increasing the problem size affects the serial part in linear fashion where the problem size directly affects the time it takes to compute the result. If the problem size doubles, the time it takes to compute in serial doubles as well.

Parallel result only shows a very small increase in time it took to calculate the result. This shows how well the problem can be parallelised as each thread only need to calculate 1/8192 of the problem. The time to calculate the parallel part also includes the serial overhead of the final calculation on the host after the data was copied from the device back to the host.

From this experiment we can conclude that the parallel algorithm performs very well in comparison to the serial implementation. To increase the accuracy of the test, the problem size could be increased even further to determine how well the algorithm handles even larger problem size, however it would take a very long time for the serial part to calculate the results, therefore for the purposes of this test only a smaller problem size was chosen. We can assume that increased problem size would increase the computation time the same

way it increases in the serial implementation minus the serial overhead which remains the same.

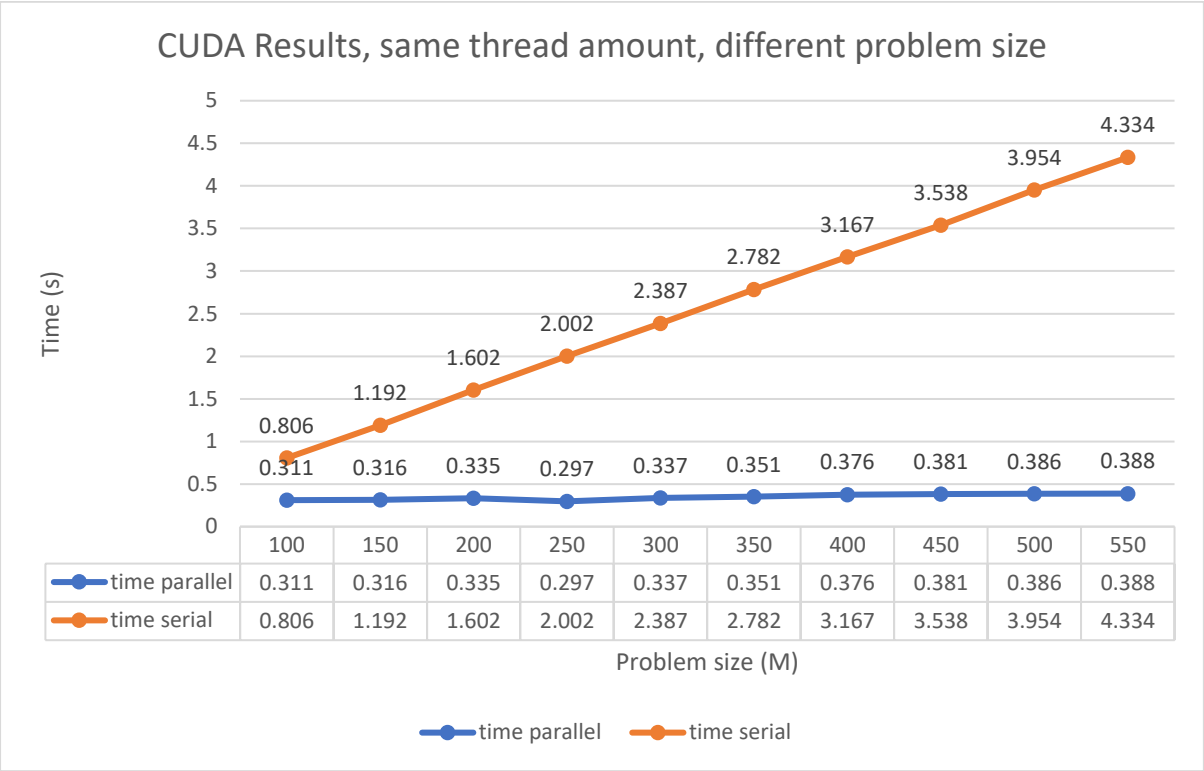

Figure 4 CUDA Results, constant thread amount, different problem size

MPI

Two experiments were conducted on MPI implementation. First test (Figure 5) is to determine how does the algorithm scale with different number of processes and second test is similar to the CUDA test to measure how long it takes to calculate the result using the same amount of processes with increasing problem size. Each step on the graph represents the number of processes on the horizontal axis and the time in seconds on the vertical axis. The test uses the same problem size of N = 500M and the number of processes increases in each step.

The timing was performed in similar way to the CUDA implementation. Parallel part of the algorithm measures how long it takes to compute the results until all processes are finished with the computation and includes the time to compute the result using the MPI_Reduce() function. Serial implementation is the same as in CUDA.

The results show that with only one process available the time it takes to compute the result is almost identical for both the serial and parallel algorithms, however the parallel algorithm with only one process is slightly slower then the serial implementation and this could be due to different approaches to calculate the π where both implementation use different equation and one may be more efficient than the other. Using more processes doesn't

affect the serial implementation as it only uses one process and the slight variances in the serial results could be affected by the processing power and utilisation of the CPU.

The CPU used to calculate the results is Intel i5-4210H [4] which is a laptop processor with 2 cores with hyper threading. With this in mind the results are not surprising. Difference between using one and two processes shows the biggest cut in computing time which is due to using a dual core CPU where the workload can be distributed evenly between the two cores. Increasing the process count to 3 and 4 decreases the computation time as well but not as drastically as going from one process to two. This is due to the CPU using hyper threading which enables multiple threads to run on each core which increases efficiency but not as much as having additional physical cores on the CPU. Increasing the number of processes even further doesn't affect computation time any more as they all need to be calculated by the same CPU.

From this test we can conclude that the parallel algorithm performs very well in comparison to the serial implementation and the ideal amount of processes for the target CPU (Intel i5-4210H) is 4 processes. We can assume that a processor with more cores available would compute the result faster as the algorithm scales well.

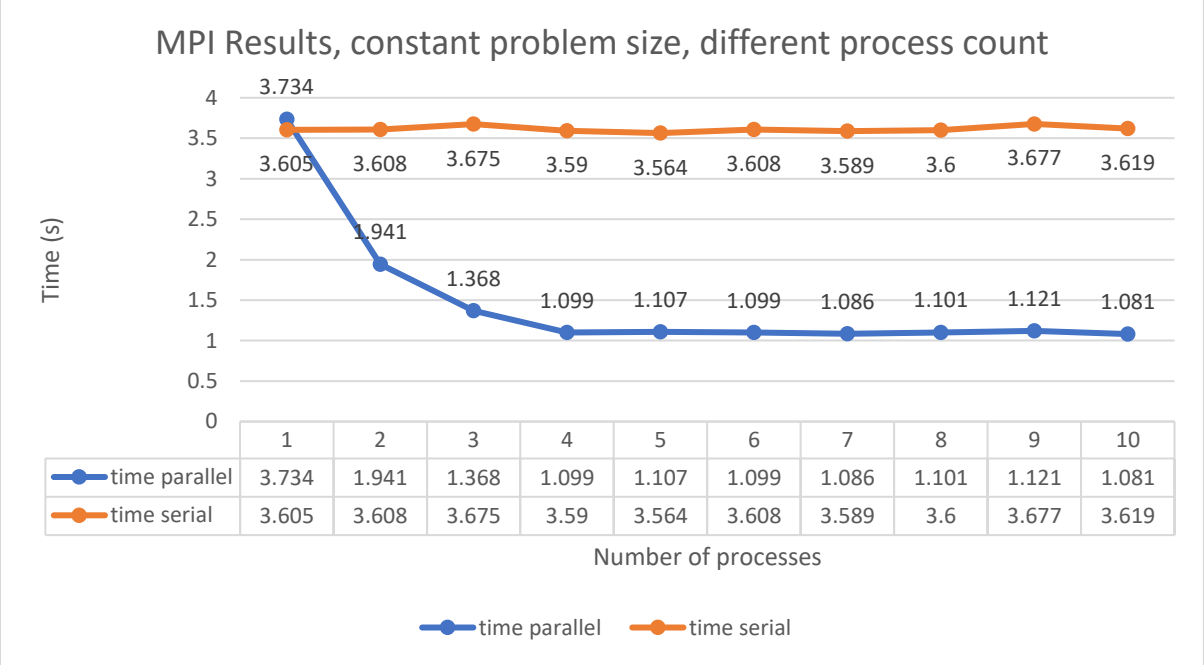

Figure 5 MPI Results, constant problem size, different process count

The second experiment on the MPI implementation (Figure 6) is to determine how well the algorithm scales with increasing problem size. The ideal number of processes was found out in the previous experiment and therefore the test is conducted using 4 processes. The problem size increases in each step.

The results for the serial implementation follow the same trend as the results in the CUDA experiment, however the time to compute the result is higher due to using a different (slower) CPU to calculate π.

The results for the parallel implementation increase the same way as the serial implementation (doubled problem size doubles the computation time) but at a slower rate due to having more resources available and each process only needs to calculate ¼ of the problem.

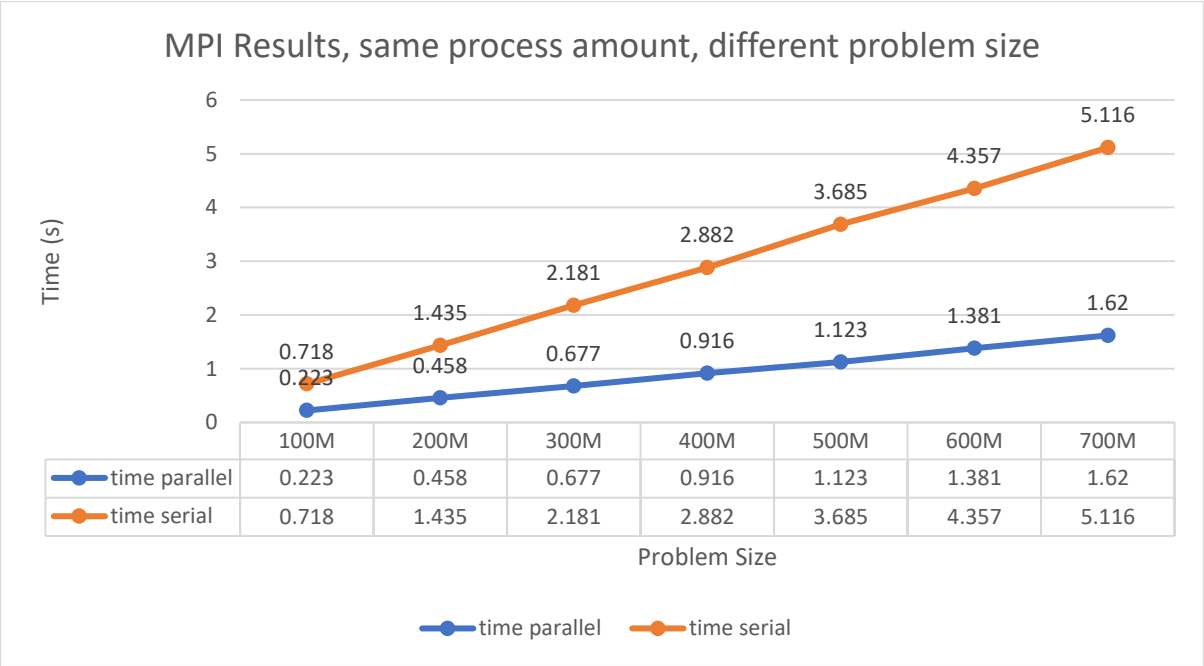

Figure 6 MPI Results, constant process amount, different problem size

Conclusion

Both CUDA and MPI implementations of the algorithm can successfully compute π in parallel with high efficiency. However, the CUDA solution performs much better than the MPI which is due to having an advantage in the processing power and number of threads available to run at the same time (8192 for CUDA compared to 4 for MPI).

Improvements could be made to the serial implementation of the code as it doesn't calculate π with very high accuracy, however the point of the project is not to find the most accurate solution but to determine how well can the problem be parallelised and what platform has the advantage. In this case of π calculation it's CUDA.

Experiment results between CUDA and MPI can't be directly comparable due to hardware differences. CUDA tests were run on university's computers and MPI test were run on a laptop which has a much slower CPU.

The MPI project can be used as a benchmark to test CPU processing power. If we use a larger problem size (so the result is not calculated too quickly) we can test multiple processors and determine which one has higher single core performance using the serial algorithm and also determine which one has higher multi core performance using the parallel algorithm.

The CUDA parallel implementation has an overhead which is limited by the host where the data at the end of the computation is copied from the device to the host and the host then calculates the final result. This overhead can not be reduced by adding any additional threads or processes.

References

[1] Gianni, M. (2018). *Foster's Design Methodology*. [online] Available at: https://dle.plymouth.ac.uk/mod/resource/view.php?id=669554 [Accessed 24 Jan. 2019].

[2] Nakano, A. (2017). *Calculating π in Parallel Using MPI*. [online] Available at: http://cacs.usc.edu/education/cs596/MPI-Pi.pdf [Accessed 24 Jan. 2019].

[3] Stack Overflow. (2015). *Calculating Pi with Taylor Method C++*. [online] Available at: https://stackoverflow.com/questions/32672693/calculating-pi-with-taylor-method-c [Accessed 24 Jan. 2019].

[4] Intel® ARK (Product Specs). (2014). *Intel® Core™ i5-4210H Processor (3M Cache, up to 3.50 GHz) Product Specifications*. [online] Available at: https://ark.intel.com/products/78929/Intel-Core-i5-4210H-Processor-3M-Cache-up-to-3-50-GHz- [Accessed 24 Jan. 2019].

[5] Devtalk.nvidia.com. (2016). [online] Available at: https://devtalk.nvidia.com/default/topic/978550/cuda-programming-and-performance/maximum-number-of-threads-on-thread-block/ [Accessed 24 Jan. 2019].