CITS3402 High Performance Computing

Project 2 – Hybird Parallelisation with OpenMP and MPI

Thomas Ankers - 21490093
Jason Ankers – 21493118

A number of techniques have been utilised in the processing of column-wise matrix collisions. These techniques have been thoroughly tested against other parallelisation techniques to ensure the fastest results. The final processing relies on a combination of OpenMP for shared-memory multi-processing and MPI for distributed memory processing over the cluster. All MPI testing was done under the same conditions using the UWA cluster during off-peak hours. Tests were queued up simultaneously to ensure a consistent environment.
The steps involved, which must be completed in order, are: finding neighbourhoods within each column, processing each neighbourhood into blocks of four elements and finding collisions between the blocks from different columns. The techniques we used to parallelise these steps are as follows:

Finding Neighbourhoods

Neighbourhoods can be found by sorting the column by value and iterating over the sorted elements. If elements are within the same neighbourhood, they will be consecutive in the sorted list and are therefore easier to find. This process is repeated for all 500 columns, with column 500 containing a particularly large number of neighbourhoods. The nature of neighbourhood finding means that it can only be parallelised at the column level. Usage of OpenMP to find neighbourhoods results in a significant increase in speed.

Parallelisation across columns								
getNeighbourhoods() on each column								
0	1	2	4	…	497	498	499	500
Total neighbourhoods found: 5170								

OpenMP	Time
No	3 seconds 424 milliseconds
Yes	0 seconds 801 milliseconds (77% decrease)

The following averages were calculated from running the functions 100 times with the same data on a quad-core (8 thread) machine.

The significant reduction in processing time achieved through the use of OpenMP and the relatively quick total runtime made neighbourhood finding a poor candidate for MPI.

Finding Blocks

Blocks are found by processing the neighbourhoods found in the previous step. A block contains four elements and is identified by its signature. A block's signature is found by summing the keys of the elements contained in the block. The first step to finding a block is to calculate the total possible combinations of 4 that can be created using a neighbourhood's elements. This is calculated using the following formula: $\frac{n!}{k!(n-k)!}$

This formula allows us to know how long a neighbourhood will take to process based on the number of blocks it will produce. For example, neighbourhoods of 4 elements will produce 1 block while neighbourhoods of 5 elements will produce 5 blocks. The maximum sized neighbourhood is found in column 500, it is 79 elements long and produces 1,502,501 blocks. The vast differences in neighbourhoods make block finding difficult to parallelise effectively. Initially in our first project, we only saw a small increase in speed from incorporating parallelisation.

Smart neighbourhood division

The strategy we developed divides neighbourhoods into groups that produce roughly the same number of blocks. These groups can then be processed in parallel on different cores using OpenMP. The steps involved in this process are as follows:

1. Divide the number of neighbourhoods (5170) into groups equal to the number of available threads. On an 8 core machine this is roughly 650 neighbourhoods per core.
2. Sort the neighbourhoods by their calculated block count
3. Allocate 650 neighbourhoods to each core so that each core will calculate roughly the same number of blocks. This was achieved by pairing large neighbourhoods from the bottom of the sorted list with small neighbourhoods from the top.
4. Recursively calculate the blocks for each core's neighbourhoods in parallel.

This strategy proved to be extremely effective in reducing the running time of block finding, the results can be seen below.

Parallelisation by neighbourhood division									
Total neighbourhoods: 5170									
Thread	1	2	3	4	5	6	7	8	Total
Neighbour-hoods	646	646	646	646	646	646	647	647	5170
Blocks	2,856,420	3,719,994	4,123,187	4,181,776	3,945,826	3,709,040	2,876,391	3,940,514	29,353,148
Total blocks found: 29,353,148									

OpenMP	Time
No	7 seconds 944 milliseconds
Yes (No smart neighbourhood division)	6 seconds 389 milliseconds (20% decrease)
Yes (With smart neighbourhood division)	2 seconds 145 milliseconds (73% decrease)

The following averages were calculated from running the functions 100 times with the same data on a quad-core (8 thread) machine.

The space complexity involved with storing blocks made them a poor candidate for *parallelisation* using MPI. The bulk of the memory used by the program comes from storing

29 million blocks, each containing 4 elements and a signature. The time used in transferring the neighbourhoods to the nodes and transferring the blocks back would significantly slow down the block finding process. Couple this with the complex structure of blocks (a struct with a pointer which points to more structs) made them more suited to the shared-memory environment of OpenMP.

Finding Collisions

The process we used to find collisions in our first project involves sorting the 29 million blocks by signature and iterating over the sorted array. This allows us to easily identify blocks that have the same signature and requires less time than the n^2 method of iterating over each block and searching the whole array for a match. The sorting of the array was done using C's built-in quicksort method and was the single longest part of the whole program. We identified this step as something that could be parallelised using MPI and the cluster. We implemented the quick sort algorithm to scatter the array to the different processes, sort them individually in parallel and then recombine and sort the final array in the master process. This task was more suited to MPI than the block finding as the sorting of the array only requires a signature, meaning that the elements did not need to be transferred as well. This cut down on the space requirements and improved transfer speeds. The results of this implementations can be seen below:

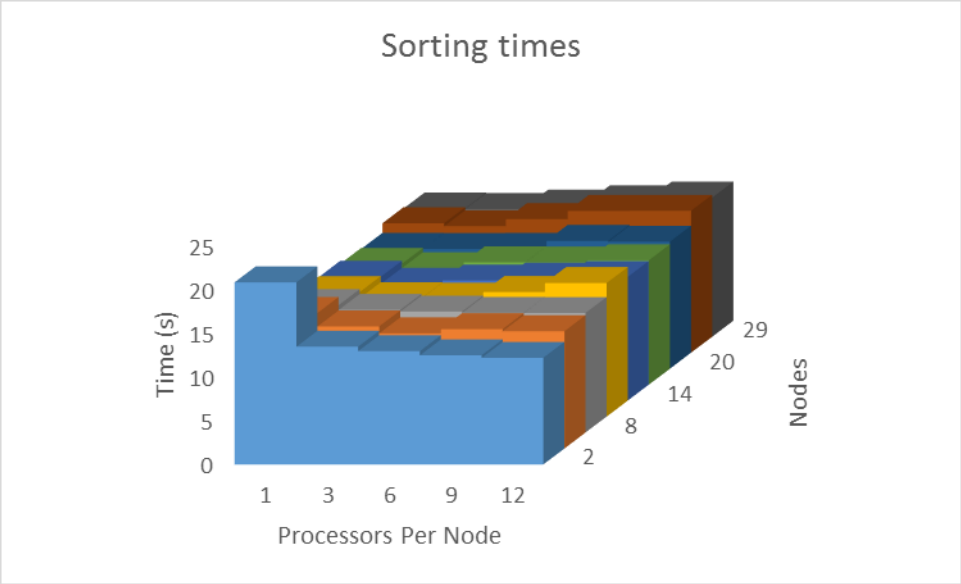
MPI	Time
No (Built in qsort)	14 seconds 593 milliseconds
Yes (Parallel quicksort)	12 seconds 217 milliseconds (16% decrease)

The following averages were calculated from running the functions 3 times with the same data on the cluster using 2 nodes and 12 PPN

The fastest configuration of MPI for our usage was found to be with 2 nodes and 12 processors per node. This was found by performing tests using a combination of different nodes and processors 3 times each and finding an average, the results of the tests are below:

	Processors Per Node				
Nodes	1	3	6	9	12
2	20.8704	13.49453	12.98073	12.50493	12.2173
5	15.5312	13.99145	13.2002	13.6832	13.43787
8	14.5979	14.04793	13.85953	13.7168	13.69663
11	14.26253	13.55903	13.5151	14.27413	15.2983
14	14.18363	13.37647	13.76523	13.97927	14.1667
17	13.8392	13.3474	14.01297	14.002	14.2542
20	13.72483	13.70863	13.72677	14.6421	14.54507
26	14.8343	14.49537	15.30375	16.23897	16.25607
29	14.7	14.63733	15.01473	15.51777	15.9376

Results are in seconds; fastest time is highlighted in green



Conclusion

In summary, we found that a hybrid implementation of OpenMP and MPI proved effective in achieving optimal running times of our program. The amount of data being processed and time taken to process this data made it challenging to achieve a major increase using MPI and the cluster. Ultimately we found that a smaller number of nodes resulted in a greater performance. This is likely due to the data transfer times between nodes, however had the data required more intensive processing, this overhead would likely have been less significant. OpenMP proved more suited to the smaller tasks for finding neighbourhoods and blocks, taking advantage of the shared memory between threads, while MPI was more suited to the large job of sorting.