CITS3402 High Performance Computing

Project 1 – OpenMP Performance Document

Thomas Ankers – 21490093

Jason Ankers – 21493118

A series of experiments have been conducted using multithreading via OpenMP to improve the total run-time speed of the program. Data has been recorded both with and without multithreading in order to illustrate the overall improvements made. **Averages are taken from running any given function 100 times over the same data.**

To give context, a brief explanation of each function is as follows:

- **loadMatrix()** – A simple function to read in the external matrix data
- **loadKeys()** - A simple function to read in the external key data
- **getAllBlocks()** – Gathers all neighbourhoods and subsequent blocks from the matrix
  - **getBlocks()** – Calculates blocks from given neighbourhoods
  - **getNeighbourhoods()** – Calculates every neighbourhood from a given column
  - **qsort()** – Sorts the blocks by their signatures
- **getCollisions()** – Calculate every collision between the blocks

getAllBlocks() simply calls the three functions beneath it a certain number of times, and as such, cannot be completely parallelised. It constitutes the majority of the runtime of the program, and will be used to demonstrate the difference in speed as improvements are made to the functions it calls.

All experimental data was gathered on the same machine, under the same conditions. Specifications are listed below.

Processor: i7 4770K 3.5 GHz

Number of processors available: 4

Number of threads: 8

Compiler: GCC 6.2.0 compiled with –fopenmp

OS: Windows with MinGW for GCC

Without multithreading

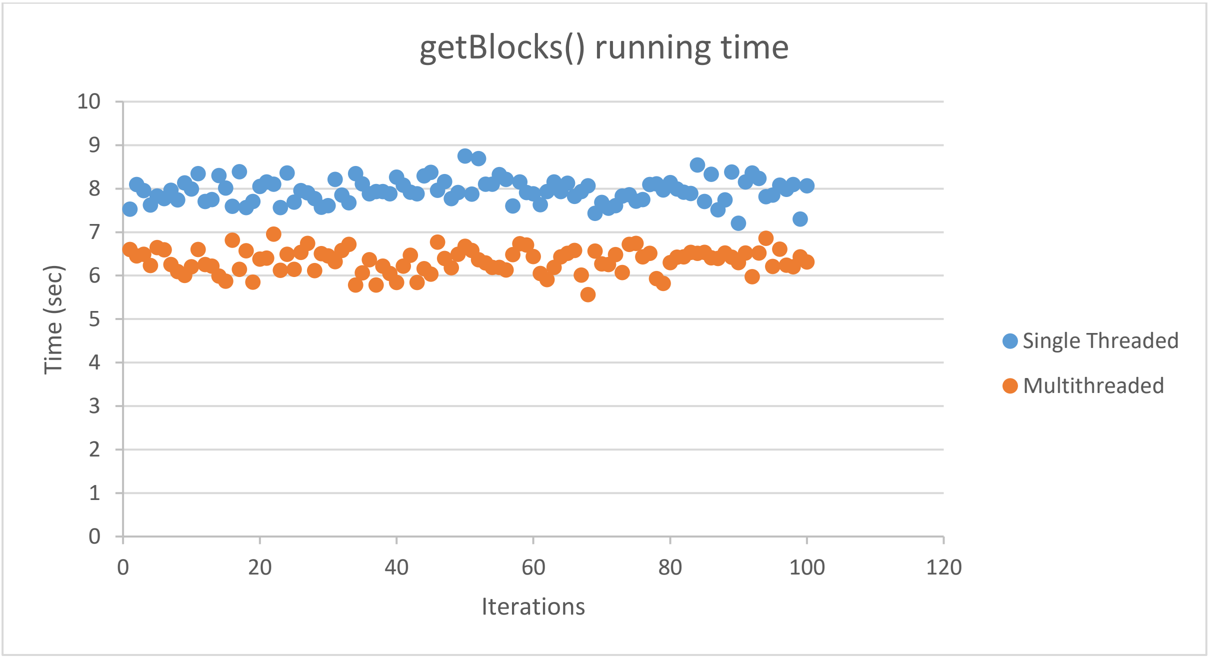
Average time taken for loadMatrix()	**0 seconds 642 milliseconds**
Average time taken for loadKeys()	**0 seconds 1 milliseconds**

Average time taken for getAllBlocks()	**20 seconds 243 milliseconds**
Average time taken for getBlocks()	**7 seconds 944 milliseconds**
Average time taken for getNeighbourhoods	**3 seconds 424 milliseconds**

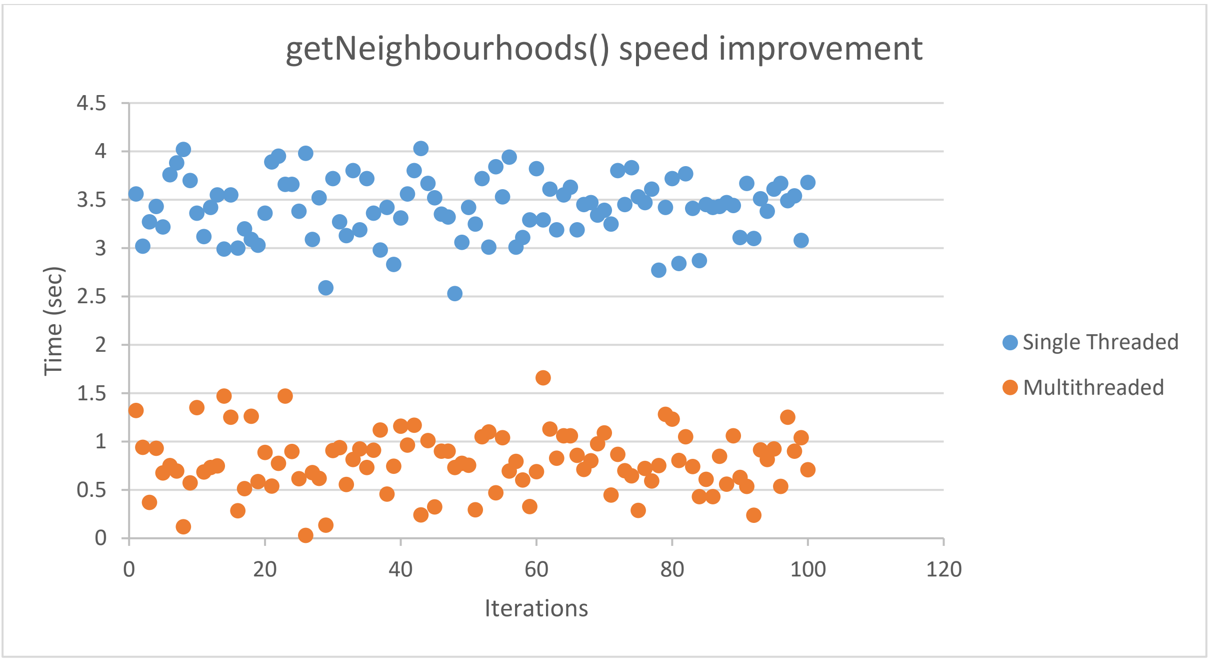
With multithreading

Average time taken for getAllBlocks()	**16 seconds 402 milliseconds**
Average time taken for getBlocks()	**6 seconds 389 milliseconds**
Average time taken for getNeighbourhoods	**0 seconds 801 milliseconds**

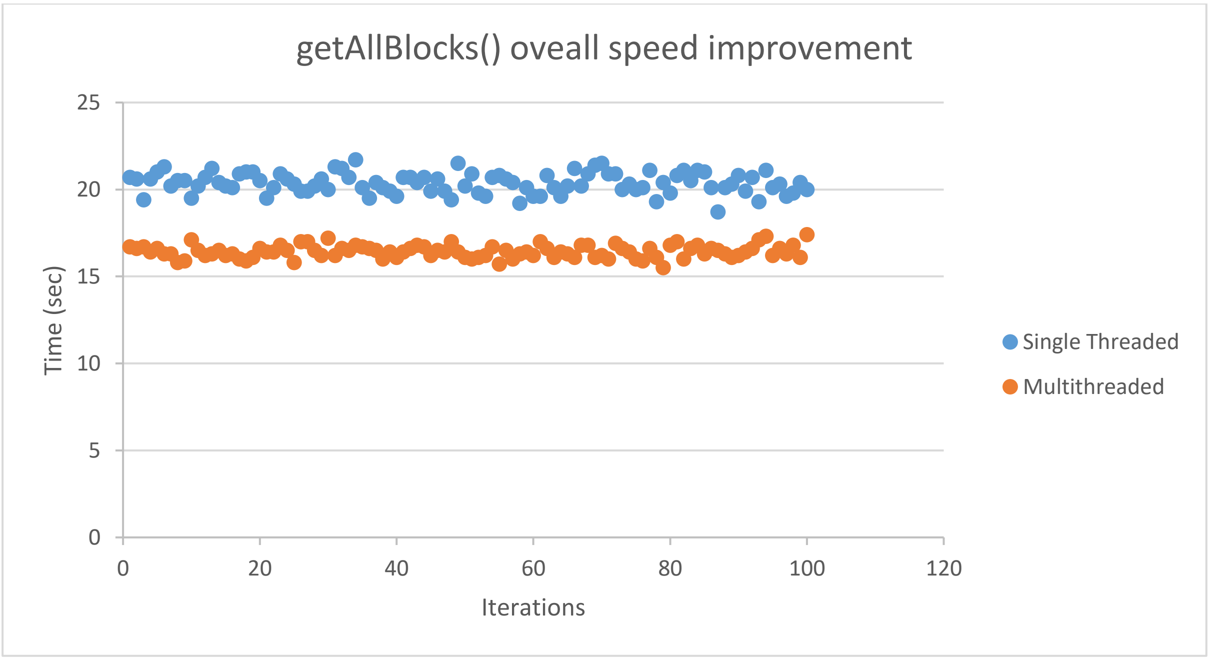
loadMatrix() and loadKeys() are left unchanged, both take less than 1 second on one thread. Parallelising the outer loop of getBlocks() required some modification to the way counters and variables are modified. getBlocks() calls the recursive function findCombinations() which is responsible for computing the blocks. A counter was used to track the number of blocks added to the array. When parallelised, this produced unexpected results due to sharing of the counter across multiple threads. By wrapping the reading and updating of the counter in an atomic capture block, the unexpected behaviour was prevented. After being successfully parallelised the runtime of getBlocks() was reduced to an average of 6 seconds 389 milliseconds. A 19.57% speed increase.



Parallelising the calls to getNeighbourhoods() did not require any modifications, as no external pointers, structs, or counters are modified within the function. The runtime of getNeighbourhoods was reduced to an average of 0 seconds 801 milliseconds, a 76.61% speed increase.



getAllBlocks() results in an average time of 16 seconds 402 milliseconds, an overall improvement of 18.97%. In summary, the implementation of multithreading proved successful in consistently reducing runtime speed of functions, cutting off almost 20% of the total running time of the program. The final scatter plot of getAllBlocks() can be seen below.


getAllBlocks() oveall speed improvement
Time (sec)
Iterations
Single Threaded
Multithreaded