CITS3402 High Performance Computing

Mitchell Pomery
21130887

October 4, 2014

Introduction

To gain the ability to implement parallel processing the codes variable scopes needed to be fixed. The code was originally written in the C89 standard, which relies heavily on global variables, so it needed to be rewritten. This meant that every loop was using the same variable, and so attempting to run multiple loops at once would cause them to increment each others initialization variable.

Development

The code was placed into a github repository to make it easy to track changes and backtrack if issues arose. This came in handy when I accidentally changed a double to an integer and was suddenly seeing two files coming out differently than before. It was also useful when trying out different techniques for making the code use multiple threads. Some changes made decreased the speed of the parallel processing and so were not worth it.

To make sure that the code was compiled consistently, a makefile was used. The makefile was also used for testing that the input and output at each code iteration was the same. It did this by comparing the new programs output with the output from the original code base. This worked perfectly while refactoring the original code until the main while loop was modified. The main while loop was changed to a for loop, and the output ended up being ever so slightly different due to issues with floating point number precision.

Jenkins CI (Continuous Integration) was originally going to be used to automate the testing. However I was unable to get my Jenkins server to SSH to the High Performance Computing Server, despite both machines being located at UWA. Jenkins was eventually set up on the High Performance Computing Server, and a job was created to automate the building and testing. This allowed for a massive reduction time due to the fact that transferring files, compiling and then testing was all automated. After each job had been run, the output was logged, along with the git revisions SHA, meaning that past results could be viewed to debug problems and performance decreases that occurred.

Performance

Performance was tested by running the code on `hpc.csse.uwa.edu.au`. Unfortunately no system had been implemented to prevent multiple people from attempting to test at the same time, so in the time leading up to the submission deadline the server ended up being monopolized by one person. The original program was run to get a base timing, however due to how quickly it ran, it was modified so that the chain length was 500. This meant that the code ran slow enough to see if any improvements had been made, but not so slow as to hinder development. While testing, an apparent speedup was observed due to a difference in compile flags. For this reason, I tested with multiple compile flags to see how it altered performance.

Test Results

Name	Compile Flags	Run Time (seconds)
Supplied	-std=c99	77.989
	-std=c99 -O1	26.914
	-std=c99 -O2	25.748
	-std=c99 -O3	22.067
Refactored	-std=c99 -Wall -Werror	86.418
	-std=c99 -Wall -Werror -O1	24.303
	-std=c99 -Wall -Werror -O2	24.868
	-std=c99 -Wall -Werror -O3	22.110
Parallel	-std=c99 -Wall -Werror -fopenmp	35.938
	-std=c99 -Wall -Werror -fopenmp -O1	20.639
	-std=c99 -Wall -Werror -fopenmp -O2	18.023
	-std=c99 -Wall -Werror -fopenmp -O3	18.760

From the data in this table, there are only two sets of results we need to consider, no compiler optimization and maximum compiler optimization. These data sets are shown in the tables below.

No Compiler Optimization

Name	Compile Flags	Run Time (seconds)
Supplied	-std=c99	77.989
Refactored	-std=c99 -Wall -Werror	86.418
Parallel	-std=c99 -Wall -Werror -fopenmp	35.938

Looking at the raw speeds of each iteration allows us to see how the program performed without assistance. The supplied and refactored code bases ran in similar time frames, showing that the refactoring did very little to improve performance. The refactoring done was to reduce the codes memory footprint and remove unneeded operations, as well as rewriting several loops to remove global variables. Once parallel processing was added to the refactored code, a massive difference in run time was observed. The code that utilized multiple cores of the machine was running in 46% of the time of the original code.

Maximum Compiler Optimization

Name	Compile Flags	Run Time (seconds)
Supplied	-std=c99 -O3	22.067
Refactored	-std=c99 -Wall -Werror -O3	22.110
Parallel	-std=c99 -Wall -Werror -fopenmp -O3	18.760

Using the `-O3` compiler flag drastically reduced the run time for each code base. The supplied and refactored code bases again run in similar time frames. This time, the performance achieved by using multiple cores was nowhere near that of the non compiler optimized code. The parallel code only shaved of 15% of the original codes run time. This is due to the fact that the overhead introduced by using multiple cores is a large portion of the performance increase it brings. If the chain length was increased again, the parallel code would have a larger performance increase as the overheads introduced became less significant compared to the overall run time.