CITS3402 High Performance Computing

Mitchell Pomery
21130887

November 4, 2014

Having a program use multiple cores of a CPU and multiple machines is useful for resource intensive tasks such as hashing. Multiple CPU cores and Machines can improve the performance as long as the overheads of this are outweighed by the performance increase. The Fermi-Pasta-Ulam (FPU) problem from non-linear dynamics is simulated by `breather.c`. This simulation is very resource intensive so improving its runtime is of great interest. OpenMP is a library that can be used to spread computations over multiple cores in a computer. Open MPI is a library that allows you to further spread the work over multiple computers on a network. These two libraries were used to decrease the time it takes for `breather.c` to run. Unfortunately `breather.c` is not a very good candidate for parallelization due to the constant need to look at previous values in the array.

`breather.c` was originally written for the C89 standard, which relies heavily on global variables. These global variables needed to have their scope changed so multiple threads could work simultaneously. While modifying the variable scopes several variables could be converted from an array to a single variable, as only one variable from the array was ever read.

When using OpenMP in the program to attempt to increase speed, care had to be taken to ensure the program would output the same data to the files. This meant that anywhere that wrote to a file could not be run in parallel. The first inner loop in the single large while nested loop was seen as the best candidate for parallelization. Each iteration of the loop looked like it could be performed separately without modifying the programs outcome. The loop also required a significant amount of CPU time to run compared the the rest of the program. To make this loop run on multiple cores the outer loop needed to be changed to a for loop. This changed it to a triple nested loop, but did not modify the number of operations performed. Upon doing this the program hit issues with floating point precision and so started to return slightly different values.

After OpenMP was implemented and a decent performance increase was observed, Open MPI was added to the mix. This limited the areas where Open MPI could be used in the program. It was used in the same loop that was created during implementing OpenMP. After fully implementing Open MPI it was discovered that this was not a valid target to spread over multiple machines. The function `accel()` relies on previous values of `x`, which meant all values of `x` needed to be known to calculate the new accelerations.

Fortunately due to the choice to use Version Control, it is easy to return to a previous working state to modify the code to make it work correctly. Unfortunately due to time constraints this has not been done. 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 a double was accidentally changed to an integer and two files suddenly had different data. 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. When distributing the code to multiple machines to test the performance once Open MPI had been used, `git` made it easy to pull the latest version of the code and compile it on the system.

To make sure that the code was compiled the same, a makefile was used. The makefile was also used to automate testing and checking that the 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 during the OpenMP stage of development. 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. When developing with Open MPI, two containers were set up with 1GB of RAM and 2 CPUs for testing. They had a minimal install and were set up so it was possible to `ssh` from one to another without a password.

Test Results

OpenMP Implemented

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.

All Tests

Name	Compile Flags	Run Time (seconds)
Supplied	-std=c99	77.99
	-std=c99 -O1	26.91
	-std=c99 -O2	25.75
	-std=c99 -O3	22.07
Refactored	-std=c99 -Wall -Werror	86.42
	-std=c99 -Wall -Werror -O1	24.30
	-std=c99 -Wall -Werror -O2	24.87
	-std=c99 -Wall -Werror -O3	22.11
Parallel	-std=c99 -Wall -Werror -fopenmp	35.94
	-std=c99 -Wall -Werror -fopenmp -O1	20.64
	-std=c99 -Wall -Werror -fopenmp -O2	18.02
	-std=c99 -Wall -Werror -fopenmp -O3	18.76

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	77.99
Refactored	-std=c99 -Wall -Werror	86.42
Parallel	-std=c99 -Wall -Werror -fopenmp	35.94

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.

Open MPI Implemented

The `-O3` flag was used when testing the Open MP version of the program for a performance increase. `dt` was decreased to 0.000001 to increase the amount of processing that the modified loop had to perform to make a performance increase evident. `chainlngth` and `nprntstps` were decreased to 60 and 100 respectively to decrease the overall run time of the program so less time was spent waiting for tests to run. The program run time was measured using the program itself and the Linux `time` program.

Maximum Compiler Optimization

Name	Program Reported Run Time	Real Reported Run Time
Parallel (Single Machine)	38.12	38.05
Open MP (One Machines)	36.35	68.77
Open MP (Two Machines)	23.51	14.28

It is evident that using two machines gave the code a significant speed improvement over having the code. Looking at the real run time for the program on a single machine shows how much overhead the use of Open MPI added. The difference between the Program Reported Run Time and the Real Reported Run Time shows us how much time was spent waiting for data to be transfered between the two machines. Making the code utilize two machines drastically reduced the CPU time on the master machine, and made the program run in nearly half the time. If more time was spent to examine the program and determine a suitable place to use Open MPI, it would be possible to see a similar performance increase and the same output as the Parallel version of the code.

Bibliography

[1] Open MPI: Open Source High Performance Computing. 2014. Open MPI: Open Source High Performance Computing. [ONLINE] Available at: http://www.open-mpi.org/. [Accessed 04 November 2014].

[2] OpenMP.org . 2014. OpenMP.org . [ONLINE] Available at: http://openmp.org/. [Accessed 04 November 2014].

[3] mpomery/CITS3402 GitHub. 2014. mpomery/CITS3402 GitHub. [ONLINE] Available at: https://github.com/mpomery/CITS3402. [Accessed 04 November 2014].