

HPC Report - Gregory Sims - gs15687

Applying optimisation compiler flags

The first step I took was to apply optimisation compiler flags. These flags are just aliases of sets of other compiler flags. These other flags apply many optimisations to the generated machine code.

Matrix Size	No flags	O2	O3
1000	10.891718	3.287550	3.281046
2000	131.223217	51.716657	51.814691
4000	1026.215050	500.936584	535.832376

These tables clearly show that there is a significant increase in performance over the non-optimised code, with the -O2 being faster at larger matrices and -O3 being better at smaller matrices.

Using float instead of double

The next step I took was to change the data type of the matrices and vectors to floats instead of doubles. Since a float is half the size of a double this should make it possible to store double the number of matrix elements in higher frequency caches, reducing the memory fetch latency.

Matrix Size	Solution Error	Iterations	Total Runtime	Solver Runtime
1000	0.050054	2957	3.271599	3.259790
2000	0.099969	5479	22.110108	22.060969

4000	0.199903	10040	368.272344	368.272344
------	----------	-------	------------	------------

The results above show that this made the computation faster for the larger matrices. The small change of the 1000x1000 matrix could be because the smaller matrix could already fit in higher frequency caches when made of doubles.

Prevent cache thrashing

The provided algorithm accesses the data in matrix A in column major order, however the data is stored in row major order. To fix this, either the algorithm could be changed, or the form of data storage could be changed. I changed the structure of the matrix to store data in column major order. This prevented cache thrashing by making use of the spatial locality of the data in the cache to reduce the number of calls to lower frequency memory, in turn reducing latency.

Matrix Size	Solution Error	Iterations	Total Runtime	Solver Runtime
1000	0.050054	2957	2.722780	2.711216
2000	0.099969	5479	20.109965	20.064561
4000	0.199903	10040	157.603540	157.422528

The data here clearly shows significant improvement in the 4000x4000 matrix as this is the case where the most cache thrashing would have taken place.

Using ICC

In an attempt to optimise the code for the architecture of blue crystal, I switched to using the C compiler provided by Intel. The ICC is designed by Intel for Intel CPUs, where as CC and GCC are meant to be more general. This makes the code run faster as it is possible to

exploit optimisations in the specific CPU.

Matrix Size	Solution Error	Iterations	Total Runtime	Solver Runtime
1000	0.050048	2957	0.604577	0.594591
2000	0.099970	5479	4.233900	4.194686
4000	0.199879	10040	46.946463	46.789919

This compiler automatically applies vectorisation, where CC and GCC did not; which I imaging is the primary cause of the increase in performance.

Applying -fast flag

The -fast flag is another flag which applies a set of other flags, much like -O3, including O3. This optimises the code for the same reasons that O3 initially optimised the code.

Matrix Size	Solution Error	Iterations	Total Runtime	Solver Runtime
1000	0.050049	2957	0.596088	0.586650
2000	0.099972	5479	4.120078	4.083880
4000	0.199882	10040	45.606695	45.461055

Applying -axSSSE3 flag

The -axSSSE3 flag instructs the compiler to compile and optimise for the Xeon processor architecture. This allows the code to utilise more of the features of the processor.

Matrix Size	Solution Error	Iterations	Total Runtime	Solver Runtime
--------------------	-----------------------	-------------------	----------------------	-----------------------

1000	0.050049	2957	0.596737	0.594591
2000	0.099972	5479	4.092255	4.056034
4000	0.199882	10040	45.288872	45.143040