Tristan Saunders
ts16802

HPC CW1 Report

Introduction

For our first assignment using Bristol's Blue Crystal super computer we were given source code that implements the Jacobi method that determines the solutions of a diagonally dominant system of linear equations. The task consisted of implementing serial code optimizations in order to minimize the execution time of the Jacobi method on different matrix sizes.

Compiler and compiler flags

A first approach to starting the code optimizations consisted of looking at the compiler and compiler flags used. The correct choice of compiler and flags can result in significant increase in code performance. The original code provided uses a GCC compiler with no optimization flags specified.
And our target runtimes post-optimizations are the following:

1000 x 1000: <1 s
2000 x 2000: <8 s
4000 x 4000: <60 s

GCC's optimization flags produced numerous speedups, with the –O2 and –O3 flags reducing the runtime to approximately a third (3,268s) of the initial runtime. However considering we are working on Intel CPUs, and ICC is heavily optimized to run on Intel CPUs, simply switching to Intel's ICC compiler with no specified flags produced a runtime of 3.248s, 3.36 times the speed of the original 10,9s, and more efficient than GCC for the 1000x1000 matrix. GCC with the –O3 flag outperformed ICC with no flag when dealing with 2000x2000 and 4000x4000 matrices however, which is why the –O3 flag is later replaced with a more efficient flag, -fast.
The –O2 and -O3 flags optimize compilation time, program speed and performance. –O3 offers solutions such as loop and memory access transformation, maximizing speed, and is recommended for applications that have loops with heavy use of floating point calculations and process large data sets.
At this stage however, adding –O3 flag to the ICC compiler made the runtime slower, as the –O3 flag doesn't always allow for optimizations.

Profiling and loop merging

Profiling the program using gprof showed that the majority of the program's runtime was spent in the run() method, responsible for running the calculations behind the Jacobi method, making it clear that initial optimization efforts had to be focused there. Attempts at merging loops in the program to avoid revisiting memory when not necessary, such as the for-loops in the run() method, only slowed down the program. We can assume this is due to ICC's built in optimizations handling this by default in this case.

Memory Access Optimizations

Memory Access Pattern

C loads data in memory in row major order (RMO), however the initial code initializes and accesses the data in column major order, counter-intuitively to the way the data is stored in memory. This has a negative effect of causing Cache Thrashing. The non-contiguous data layout leads to conflict in the cashing system that causes cache lines to be loaded back in memory, causing performance to degrade.
A solution involved changing the initialization and access of array elements to row major order, enabling a contiguous access pattern in memory, significantly optimizing runtime (3-4x faster).

Type check

In the original code, array pointers and calculations results were 8-byte doubles taking up twice as much memory as 4-byte floats. Changing data types was therefore necessary to accelerate execution. However it only slowed down the runtime in the original solution.
Once the data layout is contiguous, swapping double to float allows twice the amount of data to be stored in caches using the same space, resulting in a speedup close to a factor of 2.

Vectorization

Using the -vec-report=2 flag produces a detailed report showing what loops in the code are vectorized and explains why other loops are not.
Vectorization is based on a parallel computing trait called Single Instruction Multiple Data (SIMD) that describes performing the same operation on multiple individual data points simultaneously.
The report stated that the for-loop responsible for performing the matrix multiplications in the run() method did not vectorize due to a dependency caused by the inner if statement. Re-writing the code to remove the inner if statement enabled vectorization of the for-loop resulting in a slight increase in performance.
Additionally, ICC's default –O2 flag automatically enables SSE (Streaming SIMD Extensions) optimization that allows for vectorization, but its –fast flag looks more aggressively for vectorization opportunities by enabling other optimization flags such as –O3 and –XHOST.

Extras

Aliasing can be a danger to optimizations when dealing with vectorization. In order for the compiler not to make memory-pointing mistakes the 'restrict' keyword is used in pointer declarations to avoid aliasing (~15% increase in speed in this case).
Further optimizations could involve using code blocking to avoid cache misses, minimizing cache thrashing, and increasing the width of the SIMD register file from 128 bits to 256 bits could enhance the number of operations ran simultaneously.

End Solution

The following table represents the different optimizations performed paired with test results to show the efficiency of each individual optimization, along with the efficiency of the end solution as a whole compared the original code provided. Overall, these optimizations have accelerated the solver runtime of the Jacobi method by (10.9/0.454) ~= 24 times for a 1000x1000 matrix, (130/3.277) ~= 40 times for a 2000x2000 matrix, and (1180/39.201) ~= 30 times for a 4000x4000 matrix.
A next step to optimize even further will be to parallelize the problem using openMP.

Size	GCC	- O3	ICC	RMO	Float	Vectorization	-fast	restrict	Error
1000	10.9s	3.268s	3.248s	1.049s	0.588s	0.558s	0.534s	0.454s	0.05004
2000	130s	52.051s	53.406s	12.528s	4.156s	4.056s	3.943s	3.277s	0.09999
4000	1180s	534,163s	555.268s	96.742s	46.456s	46.299s	44.752s	39.201s	0.1998

References

https://en.wikipedia.org/wiki/Advanced_Vector_Extensions
https://software.intel.com/sites/default/files/m/d/4/1/d/8/icc.txt
https://en.wikipedia.org/wiki/Thrashing_(computer_science)