Porting the Jacobi Code to OpenMP

Adam Matheson
Student Number: P 1757290

November 23, 2017

1 Introduction

The trend in modern HPC is clear: parallelization is the way forward. In every aspect, performance is being found through concurrent execution; be that via vectorisation, multi-core CPUs or GPGPU computing. Despite new challenges arising from this new paradigm, the benefits on offer make this new way of programming more than worth the effort.

To illustrate the potential performance gains on offer, what follows is a walk-through of a successful attempt to parallelize optimised serial code running the Jacobi algorithm for solving a set of linear equations, by porting it to OpenMP.

1.1 Methodology

After making a change, the program was re-profiled and the new run-time for matrices of orders 500, 1000, 2000 and 4000 recorded, averaged and rounded to an appropriate accuracy across three runs on BlueCrystal. Times are given in the format W/X/Y/Z for each matrix order respectively and are in seconds. For convenience, where significant performance gains were made, improvements are described in terms of "*t*X times faster" (where *t* is the speedup); but smaller improvements are described in terms of percentages.

1.2 Serial Optimisations

Prior to the parallelization described in the following sections, a number of basic changes and serial optimisations were explored. Specifically, these were (in order, performance gain in reference to times after previous optimisation):

1. Change compiler from GNU (gcc 4.8.5) to Intel (icc 18.0.0): 3X faster
2. Ensure data order and data access pattern are the same (column vs. row traversal): 3.5X faster
3. Change data type from `double` to `float`: 5%-10% faster
4. Enable `-O3` compiler optimisation flag: 15% faster

5. Vectorisation check using `-qopt-report`
6. Attempted loop fusion (failed: introduced high error rate)

1.2.1 Further Serial Optimisations

In addition to the above, further potential optimisations became apparent after submission and were dutifully included in the serial code before embarking on this parallelization endeavour:

1. Move branching code `if (row != col)` in `run()` method to outer loop

2 Parallelisation

2.1 Baseline

To judge relative performance improvements, we require a baseline runtime. The serial code with the above optimisations and changes (including continuing with `icc` and `-O3` optimisation) ran in W/X/Y/Z.

2.2 Performance Analysis

Relative performance improvements are good, but how does a programmer know if he is really running fast code? What is "fast"? We require a way to measure absolute performance gains, against a theoretical maximum performance. We can use the concept of "operational intensity" to produce a few approximations of a theoretical maximum performance.

2.2.1 Computational Complexity

It is useful to first understand the computational complexity (CC) of the Jacobi code in terms of "big O" notation. The key part of the Jacobi algorithm processes a matrix, nestling a . . . therefore the CC is . . .

2.2.2 Operational Intensity

We can use the CC to help work out the operational intensity (OI). This is a measure that can be described as "operations per byte of memory traffic". Whereas CC only accounts for compute cost, OI can also describe the relationship between compute and memory cost.

Looking at the Jacobi code and its CC, we can also identify the memory operations required during matrix processing. Combining this with the CC gives OI of . . .

2.2.3 STREAM Benchmark

Another way of measuring absolute performance is with the STREAM benchmark. This measures the sustainable memory bandwidth for four long vector operations on specific hardware. Knowing the result of this benchmark for BlueCrystal can tell us how close we are to achieving peak memory bandwidth.

Running STREAM on BlueCrystal results in ...

2.2.4 Roofline Model

Now the OI and peak memory bandwidth can be used in a roofline model. This graph provides a visual means of identifying optimal performance in terms of a trade-off between compute and memory cost. We can also identify where the current code is under the "roof" of the graph to determine whether the code is compute-bound or memory-bound, helping to focus optimisation and parallelization efforts. Here is the roofline model for Jacobi running on BlueCrystal:

...

2.3 Adding Pragmas

parallel for loop look at toy code

2.4 Profiling

gperftools, tau

2.5 Cache coherency/false sharing

Reduction

2.6 Reprofiling

Overhead of parallel

2.7 Libraries, BLAS, NAG C

Seven dwarves

2.8 Re-testing compiler

Sometimes compiler optimisations can be arbitrary, worth checking flags again

3 Conclusion

3.1 Going Further - Scaling

One final useful thing to know is how well a piece of code will scale with further parallelization. Testing the Jacobi algorithm on a growing number of cores results in the following graph:

. . .

Which means the scaling is type. . .

So can be scaled more/cannot be scaled that well

Amdahl's Law Gustavson's Law