Optimising the Serial Jacobi Code

Adam Matheson
Student Number: P 1757290

October 19, 2017

1 Introduction

As parallelization continues to increase in the world of high-performance computing (HPC), optimising the serial code to begin with is just as important (if not more important) than ever. Parallelizing one's code may seem like a tempting choice when it has the potential to offer such large performance increases without the tedium of profiling and analyzing code as required with optimisation, but it would be remiss to leave such performance on the table. What follows is a walkthrough of an attempt to optimise serial code running the Jacobi algorithm for solving a set of linear equations.

2 Optimisations

2.1 Baseline

In order to make meaningful assessments of the performance gains made whilst optimising, baseline performance is required beforehand. From here on unless stated otherwise, performance will be tested on matrix orders 1000, 2000, and 4000 to give a range of figures, so places where Xs/Yz/Zs is seen refers to run time for matrix order 1000 = X (in seconds), 2000 = Y (in seconds), and 4000 = Z (in seconds). Run times are averaged and rounded.

Running the code as-is on BlueCrystal Phase 3, compiled with GNU compiler without any optimisation flags or debugging or any improvements to the stock code, gave times of 10.91/130.75/1180.81s.

2.2 Compiler

The first crucial step to optimisation is of course profiling. Learning where the slow part of your code is is essential for efficient use of optimisation time; even eliminating 99.999% of the run time of a code block that only takes up 1% of the overall program run time is still only eliminating 1% of the program run time. By contrast, reducing a code block that comprises 90% of the total run time by 10% eliminates 9% of the total run time.

However, before beginning profiling, I switched compilers from GNU to Intel C Compiler (icc). Whilst the GNU compiler may indeed be better for some things, I didn't need profiling to know the Intel compiler offers better optimisations than GNU. Besides being the general concensus, in speed tests such as **X AND Y** prove it compiles to faster executables. Indeed, after this change, the timings were 3.27s/53.35s, nearly 3 times as fast without changing a single line of code myself.

2.3 First profile and offending code identification

Now the preliminary steps had been taken care of it was time to profile. After a short comparison of profiling tools including gperf, tau, valgrind, gperftools, and Intel VTune, I settled on gperftools. It struck a balance between detail and complexity. Gprof seemed too simple from initial tests, although it is installed by default on Linux systems. At the other end of the scale VTune seemed to offer incredible flexibility, but was more difficult to manipulate in order to get the exact desired output.

Whilst gperftools was not installed on BlueCrystal, I could profile locally since the processing power required was not high and I didn't need like-for-like run times; I just wanted to identify which code blocks were taking up *percentages* of run time. After installing libunwind, dot, dv and EPEL, gperftools could be used. This simply involved including the header file and invoking "ProfilerStart(¡outputfile¿)" and "ProfilerStop()" functions, then recompiling with -g debug flag and linking to the profiler with -lmprofiler. It is worth mentioning that debugging increased the run time significantly to around 10 seconds for the 1000 matrix, but once again, I am simply trying to identify percentage of run times. The profile is accessed with pprof –text –lines ./¡program¿ ¡profilefile¿.

Immediately gperftools identified line 64 of the default code as an offender occupying 85% of the total run time. I identified this as a loop in the run() function where the Jacobi algorithm is executed, and the dot product of the inputs and the x values is calculated.

2.4 Investigation and cachegrind cache analysis

Which line of code was the offending line may have identified by the profiling tool but it was still up to human logic to figure out exactly why this line was offending. To figure this out, I relied on the trusty pen and paper - empirically the best debugger around (reference). I reduced the matrix order to 3 and drew out step by step which element of the array is being checked and assigned for each value of row and col (9 values in total). When I visually represented the single dimension array as a 2D array on paper it became clear the data was being accessed column-wise. Since in C arrays are stored row-wise, this means that the memory was being accessed in an inefficient manner - jumping around incontiguously - which involves memory being loaded in and out throughout the memory heirarchy and wasting CPU cycles.

To confirm this suspicion, I installed valgrind and utilised it's –cachegrind option which analyses how memory is being accessed, from instruction registers through L1 and L2 caches. The following results were produced:

```
[adstr123@adam2500k intro-hpc-jacobi]$ make
icc -std=c99 -g -o jacobi jacobi.c -lm -lprofiler
[adstr123@adam2500k intro-hpc-jacobi]$ valgrind --tool=cachegrind ./jacobi
==20497== Cachegrind, a cache and branch-prediction profiler
==20497== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==20497== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==20497== Command: ./jacobi
==20497==
--20497-- warning: L3 cache found, using its data for the LL simulation.
-----------------------------------
Matrix size:            1000x1000
Maximum iterations:     20000
Convergence threshold:  0.000100
-----------------------------------
Solution error = 0.050047
Iterations     = 2957
Total runtime  = 511.214059 seconds
Solver runtime = 510.494224 seconds
-----------------------------------
PROFILE: interrupts/evictions/bytes = 30271/0/336
==20497==
==20497== I   refs:      86,283,354,525
==20497== I1  misses:             2,664
==20497== LLi misses:             2,422
==20497== I1  miss rate:           0.00%
==20497== LLi miss rate:           0.00%
==20497==
==20497== D   refs:      47,571,465,605  (41,576,314,197 rd   + 5,995,151,408 wr)
==20497== D1  misses:     3,341,150,105  ( 3,336,103,128 rd   +     5,046,977 wr)
==20497== LLd misses:       370,490,427  (   370,222,813 rd   +       267,614 wr)
==20497== D1  miss rate:            7.0% (           8.0%     +           0.1%  )
==20497== LLd miss rate:            0.8% (           0.9%     +           0.0%  )
==20497==
==20497== LL refs:        3,341,152,769  ( 3,336,105,792 rd   +     5,046,977 wr)
==20497== LL misses:        370,492,849  (   370,225,235 rd   +       267,614 wr)
==20497== LL miss rate:             0.3% (           0.3%     +           0.0%  )
[adstr123@adam2500k intro-hpc-jacobi]$
```

This was actually a little better than I anticipated but 7% is still a very high percentage and would represent a large performance issue.

2.5 Manipulating the data structure

I had to make the algorithm access the array elements sequentially; so the "rows" of the 2D array had to be where the "columns" currently were. Altering the data initialisation by "transposing" the array using different indexing achieved this (proved with a debug loop to print the array before and after the code block). This resulted in a larger overall time due to the transposition but a lower solver run time, proving the concept. I then worked this code into the main initialisation block.

To match this, the Jacobi solver was also modified to index differently. This sent the time plummeting to 1.1s/y/90s.

2.6 Additional optimisations

Working through a structured checklist of other optimisations.

2.6.1 Precision

I decided that the precision was too high. I changed doubles to floats which resulted in an extremely small performance increase of 5%, which could be attributed to random changes. Timings after this were 1.04s/8.9s/84.7s.

2.6.2 Compiler Flags

As compiler optimisation through flags can have mixed and seemingly arbitrary effects, I decided to leave these til the end. Enabling -O3 aggressive optimisation gave times of 1.01s/7.1s/73.3s.

Enabling -no-prec-div gave times of 1.02/7.23s/73.37s, giving no benefit.

Enabling -Ofast gave 1.02s/7.18s/77.32s.

2.6.3 Loop Fusion

Combining loops gave 0.44s/2.11s/13.43s but increased the error to an inappropriate level. Given more time, loop fusion would have been worth pursuing.

2.6.4 Vectorisation

Enabled -qopt-report and viewed jacobi.optrpt. Vectorisation taking place mostly, some unable to vectorise.

3 Conclusion

Final times were 0.99/6.81/66.61. Overall this exercise proves there is massive value to be had in optimisation before parallelization. In a time when high performance computing resources are treated as lab tools, allocated by the second, every time saving operation counts. Whilst modern compilers can do a lot of work for you, there is still a lot to be said for the programmer optimising options and code herself.