Exploring Optimisations for the Serial Jacobi Method

Anthony Wharton
aw15885@bristol.ac.uk

Abstract

After being given code which sequentially performs the Jacobi method, this report explores various possible optimisations and discusses the results of these.

1 Choice of Compiler

The original code[1] contains a Makefile which compiles the program with the `CC` compiler. In order for greater experimentation breadth, the code will also be compiled through `GCC` and `ICC`.

1.1 Basic Compiler Flags

The simplest *go-to* optimisation flags are the infamous `O0`, `O1`, `O2` & `O3` flags. These are shorthand for a whole host of other flags[2], ranging from no optimisations in the case of `O0`, to most available optimisations with `O3`. Optimisations available include fusing loops/inlining code to reduce branch instructions and hoisting conditionals out of loops to improve CPU pipelining - to name a couple. The results of these `O`-flags are given in Table 1.

	CC	GCC	ICC
O0	10.895/10.878	10.894/10.878	11.15111.135
O1	3.496/3.484	3.497/3.484	3.270/3.259
O2	3.286/3.275	3.291/3.279	3.266/3.255
O3	3.285/3.274	3.286/3.274	3.569/3.558

Table 1: Total Run Time/Jacobi Solve time for 1000×1000 (sec)

It is clear to see that across different compilers, code runs with largely the same times. In addition, as expected, the higher `O`-level flags are generally quickest. Oddly though, `ICC` performs slower with `O3` than `O2`. This is due to the fact one of the optimisations enabled collapsing `IF` statements, which typically on aggregate is better for performance. However, due to the fact we have a very predictable conditional statement on line 61 of the original code[1], this optimisation is likely to be slower than the system's hardware branch-prediction used formerly. This can even add extra unecesary instructions, slowing performance.

1.2 Compiler Reports

After manually trying small optimisations, such as loop fusion and conditional hoisting, it was found that in most cases there was negligible difference ($\pm$0.01sec) between manual and compiled `O2`/`O3` optimisations. This is within the error bounds of run time, due to varying background system load. Compiler reports are vital in order to bring light to why these changes were not performing as well as hoped, as they inform *what* the compiler is doing.

2 Profiling

Before delving deeper into optimisations, it is important to pay close attention as to what is causing slow downs in the program. Profiling will be done in `gprof` which requires extra flags at compilation (`-pg`, along with `-g` if line-specific profiling is required). Benchmarking the `gcc -O2` compiled code shows where the time is being consumed in a report, of which a snippet is included in Table 2.

Line 59 and 62 are using around 70% of the run time, and after inspecting the code[1], it is clear to see why. These lines are responsible for the majority of the 'work', as they perform the (partial) dot product in the Jacobi method. The first optimisation was made in light of this fact. It should be noted that profiling motivated many of the future optimisations as well, *even* if not explicitly mentioned in this report.

% time	cumulative seconds	name
43.57	1.43	run (jacobi.c:62 ..)
27.67	2.34	run (jacobi.c:59 ..)

Table 2: Extract from `gprof` profile report with default code compiled with `gcc` and `-O2`

3 Optimisations

As seen in the previous section, the lines responsible for loop logic and dot product in the Jacobi method were using the majority of CPU time. The simplest way to go about fixing this, and other issues, is to take into consideration how the system is *actually* working; specifically around how the CPU, Cache and Main Memory function.

3.1 'Cache Thrashing'

The Jacobi method deals with a 2 dimensional array for the coefficients of the linear system. Accesses to this two dimensional array are prevalent throughout the Jacobi method code, so care must be taken in how to store/access this data. An abstraction is made when using multi-dimensional data, due to memory being stored in one dimension. With 2 dimensional data, *Row Major Order* is the notion of storing each row one after the other in memory, whereas *Column Major Order* is the opposite notion where the columns dictate storage order. As a programmer it is important to note how the language being used stores multidimensional arrays, as this will drastically impact performance. For example C uses row major order, whereas Fortran uses column major order. Large performance impacts come from loading data to/from the cache.

Retrieving data from main memory can take an order of magnitude more clock cycles to complete than lower level caches. As data is loaded from memory into the much faster cache on a line-by-line basis, helping the compiler load as much *processable* data at once from memory, is hugely beneficial for the performance of the code. To elaborate, in the loops on line 56/59 in the original code[1], the data is being accessed in a column major manner. This will result in *'cache thrashing'*; the event where different lines constantly need to be loaded into the cache, causing high proportions of cache invalidation. In this code, accessing columns require a new cache line every time, resulting in lots of cache load requests. With minor alterations to the loops, as well as the initialisation and convergence checking code, it is possible to greatly mitigate this by making loops execute in row major order. After recompiling and testing these changes, there is an approximate $5\times$ speedup in the best case (`icc -O3`), show in Table 3.

	GCC	ICC
1000×1000 - O2	2.721/2.710	1.109/1.099
1000×1000 - O3	2.720/2.709	1.104/1.095
2000×2000 - O2	23.121/23.074	12.621/12.578
2000×2000 - O3	23.117/23.071	12.607/12.566
4000×4000 - O2	169.462/169.276	115.041/114.868
4000×4000 - O3	168.709/168.524	96.906/96.736

Table 3: After Row/Column *'cache-thrash'* Optimisation Total Run Time/Jacobi Solve Time (sec)

3.2 Vectorisation

`ICC` has automatic vectorisation in the `O2` and `O3` optimisation flags, which is what caused the first notable dramatic speed difference between compilers (Table 3). Vectorisation is the process of converting single value instructions, into vector ones. Vector instructions use multiple sources of data - the *"single instruction, multiple data"* paradigm, compared to the traditional *"single instruction, single data"* one. To confirm this, the `ICC` compiler reports stated that the inner loop in the Jacobi method was fully vectorised. Recall this where the bulk of our processing time was spent! After reprofiling, the inner loop still shows this as taking around 70% of the computational time as previously, however, now the instruction processes far more data at once, resulting in a significantly lower run time.

There are a few other things to be noted here. The first of which is that `GCC 4.4.7` does support *some* auto vectorisation, with extra flags in addition to those of `O3`. However, it yielded far less impressive results of around 1.9 seconds. In order to get comparable results with `ICC`, the compiler would need extra aid such as refactoring code into an acceptable format for the auto vectorisation procedure, or trying other techniques such as memory alignment or even updating the compiler to a more recent version! Alternatively, auto vectorisation could be disabled and code could be rewritten using intrinsics - a low level abstraction similar to assembly, for specialised instructions at the hardware level. However, intrinsics are often not portable across platforms, and deemed out of scope for this report. Fortunately thanks to the luxary of compiler choice, there is no need these optimisations the report continues solely with `ICC` due to its superior auto vectorisation.

3.3 Further Memory Optimisations

The Intel™ Sandy Bridge Xeon E5's in Blue Crystal Phase 3, which were used for this report, have large specialised vector registers. The code used `doubles` for computational values, which are 8 bytes - compared to the 4 bytes of `floats`. After changing all values to `floats`, further speed increases are observed, illustrated in Table 4. This is due to the vector registers fitting twice as many values as before, which, paired with instruction set support facilitates *even* more data to be computed at once.

The use of `_mm_malloc()` instead of `malloc()` for initialising memory locations was also tried in hope of aiding `ICC` and/or the system. This proprietary function from Intel™ allows the alignment of memory to n bit intervals. In this case 64 bits is appropriate as the cache lines are 64 bits wide in this system, and accommodating this could help decrease cache misses. Sadly this had no effect on runtime, due to `ICC` already applying this optimisation implicitly.

	1000×1000	**2000×2000**	**4000×4000**
`icc -O2`	0.602/0.592	4.212/4.173	46.844/46.689
`icc -O3`	0.601/0.591	4.206/4.169	46.891/46.736

Table 4: After changing data type from `double` to `float` Total Run Time/Jacobi Solve Time (sec)

3.4 More Advanced Compiler Flags

As `ICC` was chosen for this project, experiments were run with a plethora of flags for the following improvements; non-precise & faster division, forced instruction prefetching, faster math/floating point libraries/models, harshest loop unrolling/function inlining settings and using native assembly. These can be found in `ICCFLAG` variable in the submitted `Makefile`. This resulted in another large improvement, giving `icc -O3` about a 25% improvement from the last optimisation to around 0.48 sec for 1000 × 1000.

3.5 Computing Multiple Rows

Throughout this project, two main areas of improvement have been targetted; the compiler and the main Jacobi loop. Due to the nature of the problem, even at this late stage when profiling the latest cod,e the majority of time is still spent with the dot operations. Thus, the last optimisation efforts will once again focus here.

Cache loading was helped with our data structure row-column 'swap', however now it is uncertain whether data in the cache *actually remains* there throughout execution. In order to help ensure that this occurs, the code was changed to explicitly execute multiple lines at once. The motivation is to ensure the compiler, CPU and memory management modules realise that the memory is needed, and mitigate any cache invalidation. This inspired from tiled matrix computation for optimal cache usage, with a *'manually unrolled'*, vectorisable loop. This can be found in the inline function `dotOperation` in the submitted code. This provides the final dramatic speedup, show in Table 5

4 Conclusion

With the exploration of optimisations wrapped up, a few things can be clearly concluded. Firstly, the process of using memory affects run time greatly, and optimisations to this help immensely. And finally, compilers are clever and will help as much as possible but often need prompts in the form or flags/pragmas/rearranged code for optimal results. Reports are especially useful for working *with* the compiler, and can help debug why manual optimisations are not effective - or worse than unoptimised code! With reasonably simple changes, it is clear to see from the results that large performance improved can be gained!

	1000×1000	**2000×2000**	**4000×4000**
`orig.`	10.895/10.878	131.856/131.760	1031.428/1030.911
`opt.`	0.365/0.355	2.632/2.593	38.655/38.500
`% inc.`	2985%	5001%	2668%

Table 5: Final Results from Optimisation Efforts, Run Time/Solve Time (sec)
Results from original[1] and submitted Makefile

Bibliography

[1] UoB-HPC. *COMS30005 - Introduction to High Performance Computing - Jacobi solver.* URL: `https://github.com/UoB-HPC/intro-hpc-jacobi/tree/8cf4c5` (visited on 10/17/2017).

[2] GNU Foundation. *3.10 Options That Control Optimization.* URL: `https://gcc.gnu.org/onlinedocs/gcc-4.4.7/gcc/Optimize-Options.html` (visited on 10/17/2017).