OpenMP Optimisations for the Jacobi Iterative Method

Ahmer Butt [ab15015]

1 Introduction

This report details porting an optimised serial implementation for the jacobi iterative method to OpenMP. The code is written in C and compiled using the Intel C/C++ compiler. The optimisations include reducing false sharing, tuning load balance and choosing thread affinity to target the Blue Crystal Phase 3 supercomputer. A description of the optimisations, comparison of optimised vs unoptimised runtimes and analysis of how the code scales from one to sixteen cores is included. A final runtime of 7.1 seconds for a 4000x4000 array using 16 cores is achieved.

2 Description of Optimisations

2.1 Worksharing, Load Balancing & Reductions

Firstly, a plan for parallelisation is devised. By using a profiler, one can identify that the for-loops in the `run` function are bottlenecks. OpenMP will be used to distribute iterations of the outer for-loop to different threads. Computation of the array `xtmp[row]` can now be done in parallel, where different threads will compute a different portion of the array. This is worksharing. Within each iteration of the outer for-loop, each thread will use vectorization to effectively parallelise computation of the inner for loop. This has far less overhead than parallelisation via threads. Parallelising the outer loop with SIMD and inner with threads would result in far more overhead from managing many more threads. There is also scope for parallelising the second for-loop. To do this, it is fused with the first for-loop. This reduces overhead compared to re-creating threads once execution reaches the second loop.

Parallelisation of the second for-loop requires accumulating the value of `sqdiff` across threads. A naive way to do this would be to use a shared variable and then have every thread write to it. To avoid a race condition on writing to the shared variable, the write would have to be enclosed within a critical region. This is extremely inefficient as it would require threads to become blocked within every iteration of the outer for-loop. A far better, but still not best, implementation would be to use an array of intermediate accumulators for each thread to use, and then have the master thread combine the intermediate results into an overall accumulated value. This approach dramatically improves load balancing across the threads. Previously, threads would be forced to wait idle at critical regions until the mutex around it became unlocked. Only one thread could write to `sqdiff` at a time, whereas now the time the other threads spend idle is alleviated.

However, whilst this approach far improves load balancing, it is still very cache inefficient. This is due to cache coherency causing false sharing. If two cores have a copy of a shared variable in their local memory, then a write to the variable from the first core will cause the value to become outdated in the second core's cache. To achieve cache coherence a multiprocessor system must ensure that, firstly, changes to data in any cache must be propagated to every other cache (write propagation), and secondly, read-then-writes to a single memory location must be seen by all other processing units in the same order (transaction serialisation). To achieve this a multiprocessor system may adopt a snooping mechanism: whenever a change is made to shared resource in local memory, it notifies all other caches which have the same copy of data. If another cache has the same copy, its coherency controller will mark the copy as invalid.

When threads are writing to an array of intermediate accumulators, cache coherency causes false sharing. This is when a processor attempts to periodically access data that will not be altered by another processing unit, but that data shares a cache line with data that is altered. The caching protocol then keeps marking the data as invalid and forcing reloads, even though there is no logical necessity. The shared `sqdiff[NTHREADS]` array can fit into a single cache line. Upon every iteration, a thread writes to `sqdiff[i]`, which causes the array to be marked as invalid in every other cache. This forces a reload, even though the element of the array that another thread wants to write to will not have been changed. The solution to this is to pad each element such that it uses an entire cache line. This could be done manually, but is instead done with an OpenMP reduction. The syntax is `reduction(+:sqdiff)`. OpenMP also adds the intermediate values in a tree structure, resulting in even faster accumulation. Using this now gives a solution which is both load balanced and cache efficient. Figure 1 gives a comparison of these different approaches, showing a time increase from 35s to 17s with 16 cores.

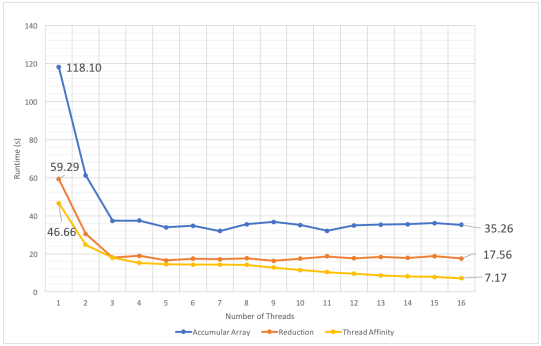

Figure 1: A comparison of runtimes from 1 to 16 cores for different implementation of the Jacobi iterative method. Times shown are for a matrix size of 4000x4000.

2.2 Scheduling and Affinity

The *schedule* of an OpenMP parallel loop determines how iterations are distributed between threads. In Jacobi, the body of the loop will take similar time to execute for each thread as they are identical and there are no conditionals. Choosing the `dynamic` option would result in unnecessary and significant overhead for distributing work dynamically to threads. Hence, the iterations are distributed uniformly by using the `static` option.

Thread Affinity is how threads are assigned to cores. BlueCrystal Phase 3 has a non-uniform memory access design, or NUMA. Each node has two 8-core CPUs. Each core has a 32KB L1 and 256KB L2 cache. Each CPU has a 20MB L3 cache. Memory that is in higher level caches is further away and takes more processor cycles to access. The L1 cache takes 1-4 cycles to access. An L3 cache miss will cause an access to DRAM. This takes 200-400 cycles to access. One must distribute loop iterations to threads and threads to cores such that memory is placed near the cores which require it and access times are minimised. However, the distribution of loop iterations must also still remain uniform otherwise load imbalance will be introduced.

To reduce memory access time, allocation of memory to a cache as close to the thread which is going to use it must be performed. This can be done using OpenMP. The for-loop which initialises the arrays is parallelised with exactly the same schedule and chunk size as the later for-loop. Now, each thread will make the first 'touch' to the segments of the arrays that they will actually use. Assignment in this parallel loop causes the OS to place the memory that each thread will use close to the core where the thread is executing. Figure 1 shows this technique decreases the runtime for 16 cores from 17.56s to 7.17s. Correspondingly, the L3 cache miss rate decreases dramatically from 35.5% to 3.9%.

The size of the chunks that the loop iterations are split up into must also be chosen carefully. Using a chunk size of 1, or round-robin assignment of iterations, is inefficient. This is because threads will access the array in non-contiguous order. As caches preload in neighbouring array elements to take advantage of spatial locality, and as arrays are stored contiguously in memory, this causes repeated cache misses. The effects are multiplied in a parallel environment due to false sharing. Instead, a chunk size of the loop trip count divided by the number of threads should be used. In this case, each thread will work on some contiguous section of the array and cache hit rate will be far higher for each cache. A comparison of runtimes and number of cache misses using these different chunk sizes is given in Figure 2. As can be seen, a large increase in runtime from 14.7s to 7.20s is achieved.

The allocation of threads themselves to different cores is also improved. If thread i operates on the xth segment of an array, then thread $i+1$ operates on the $x+1$th segment of the array. Threads that are allocated to cores close together will share the same L3 Cache. It is faster to distribute the array segments to cores that are next to each other, as they are closer together. The environment variable `KMP_AFFINITY=compact` is specified to instruct the compiler/kernel to allocate threads to cores in order and as compactly as possible. Figure 3 shows the decrease in runtime when setting `KMP_AFFINITY` to `compact` vs `scatter`, which spreads threads out as much as possible.

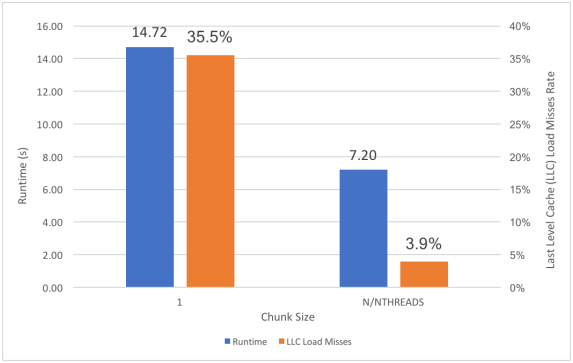

Figure 2: Runtimes and cache misses for using a static schedule with different chunk sizes. On the left, a chunk size of 1 is used. On the right, a chunk size of array size divided by the number of threads is used. Times for a matrix size of 4000x4000 is shown. The Last Level Cache is the L3 cache and the miss rate is taken by using the Intel vTune profiler.

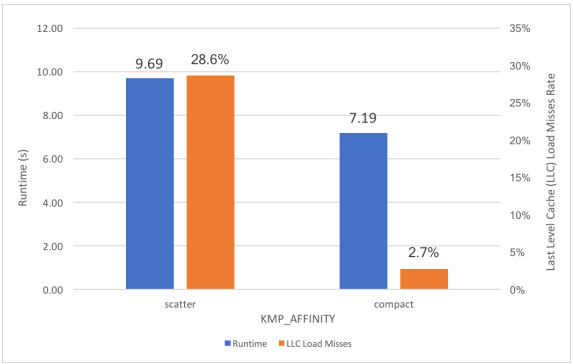

Figure 3: A comparison of runtimes and cache misses for using scatter vs compact thread affinity. A matrix size of 4000x4000 is used.

Parallelisation of the inner for-loop via SIMD instructions is also affected by the chunk size. Ideally, the chunk size should be a multiple of the vector length. In such a case, there will be no remainder loop and so likely better performance. However, loop trip counts are so large that the time spent by each thread in a remainder loop becomes very small relative to the kernel loop. Therefore, an attempt to achieve this is not discussed. Vectorisation is specified with the pragma `#pragma omp simd reduction(+:dot)` above the inner for-loop. Note that a reduction over `dot` is now also performed. Data alignment to assist vectorisation is also done by using calls to `_mm_malloc` and adding the clause `aligned(R:64, D:64, b:64, x:64, xtmp:64)` to the pragma. An in-depth discussion of vectorisation and data alignment is given in the 'Serial Code Optimisations' report.

3 Scalability of Code from 1 to 16 Cores

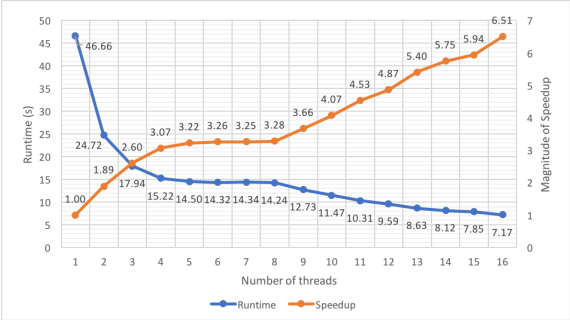

Figure 4: The runtimes and corresponding speedup achieved when using 1 to 16 cores for a 4000x4000 matrix.

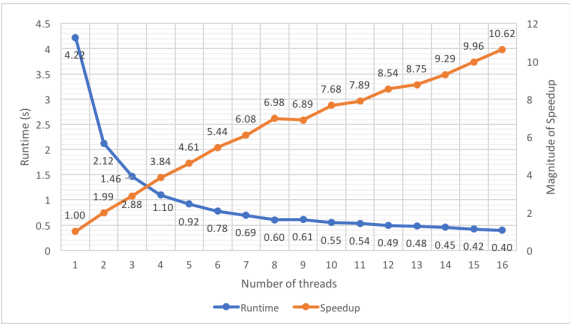

Figure 5: The runtimes and corresponding speedup achieved when using 1 to 16 cores for a 2000x2000 matrix.

3.1 Parallelisation Overhead

Figures 4 and 5 show the run-times of the final solution using 1 to 16 cores for matrix sizes of $N=2000$ and $N=4000$. For $N=4000$ a total speedup of 6.5 times magnitude is observed with 16 cores. For $N=2000$ a total speedup of 10.62 times magnitude is observed with 16 cores. These both demonstrate sub linear scaling. One reason for this is because not all of the algorithm is inherently parallelisable. One can use Amdahl's Law to plot the maximum theoretical speedup for the fixed input size $N=4000$ given increasing number of threads. To do this the proportion of the algorithm that is parallelisable must be calculated. In the submitted Jacobi implementation, the proportion of the algorithm that can be parallelised has indeed been parallelised.

The Intel vTune profiler states, given the provided algorithm in *jacobi.c*, that (when running with the profiler) an estimated ideal time spent in the parallel region is 7.1 seconds. This is achieved if there is zero OpenMP overhead from parallelisation. vTune also shows the time spent in the sequential region as 0.79 seconds. This gives the sequential portion of computation as

$$f=\frac{t_s}{t_s+t_p}=\frac{0.79}{0.79+7.1}=10\%,$$

where t_s is the time spent in the sequential part and t_p is the time spent in the parallel region (on a sequential system). Then, for N cores, Amdahl's Law states we can write the potential speedup as

$$S=\frac{1}{f+\frac{1-f}{N}}.$$

Given $f=10\%$, or a proportion of time in parallel region on a sequential system of 90%, an Amdahl curve can be drawn plotting theoretical potential speedups given increasing values of N, the number of cores. This is shown in Figure 6. As can be observed, the speedup is less than the theoretical maximum. The actual time in a parallel region is 7.7 seconds, as given by vTune.

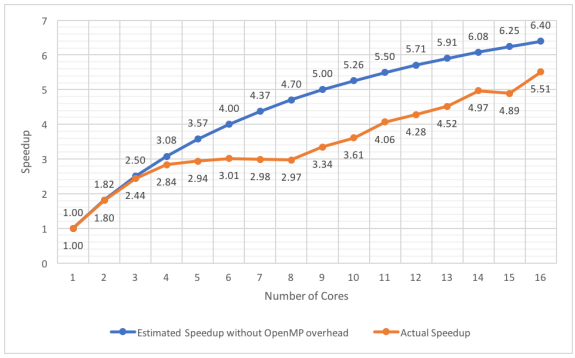

Figure 6: An estimated plot of Amdahl's Law for *jacobi.c* of the theoretical speedup achievable. It should be emphasised that the estimated ideal times given by Intel vTune *only* incorporates loss in time due to load imbalance and OpenMP runtime overhead. Therefore, the theoretical speedup curve should sit far higher. The gap between the theoretical curve and actual achieved speedups far greater; the reasons are discussed in the next section. Note, the runtimes used were measured when running with the vTune profiler.

Figure 7, generated with vTune, shows the time spent in each parallel region with 1 up to 16 cores running user code. It shows that often, less than all 16 cores are being utilised. This is because of load imbalance. vTune shows that execution took 131 seconds of CPU time, but only 106 seconds was spent executing user code. Even though an ideal static schedule has been set, there is still overhead from threads waiting idle for synchronisations. In summary, the sublinear speedup with respect to increasing number of cores shown in Figures 4 and 5 can be partly explained by the synchronisation and thread management overhead increasing as the number of threads increases. The next section discusses more contributing factors.

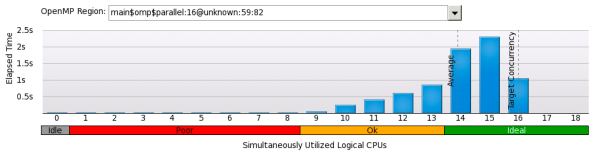

Figure 7: The elapsed time whilst $N = 1..16$ cores are simultaneously being utilised.

3.2 Memory Bandwidth

In this section, an anlysis of the memory bandwidth used by the optimised Jacobi is presented. A Roofline model of BlueCrystalp3 is considered. It is shown in Figure 8, which has been taken from the lecture slides. The Roofline model plots arithmetic intensity vs floating point operations per second. Arithmetic intensity, AI, is the total number of operations per byte of memory traffic. The value of the curve at some point represents the attainable FLOPS per second. It is calculated for a system as the minimum of the peak floating point performance and peak memory bandwidth multiplied by arithmetic intensity.

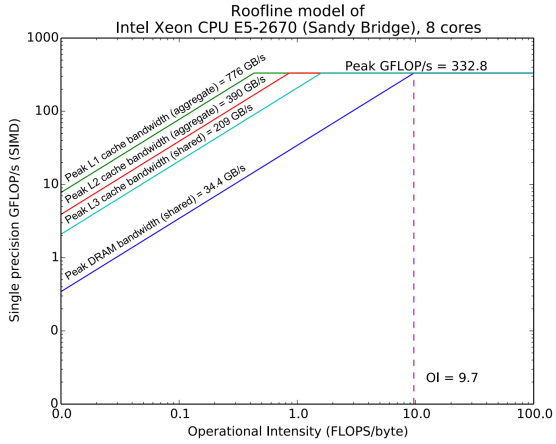

Figure 8: A roofline model for the BlueCrystal Phase 3 supercomputer, taken from the lecture slides. The ridge point is at an arithmetic intensity of 9.7. Jacobi is far less than this, at 0.22, which is derived in the text. So, Jacobi is memory bandwidth bound. Note that, this Roofline graph is for 1 socket (8 cores). The Jacobi code is run on up to 2 sockets, with a total of 16 cores. For a dual socket run, the peak GFLOP/s would increase, but a higher arithmetic intensity would be required to become compute bound.

AI can be estimated for *jacobi.c*. In this report, it is assumed that each simple arithmetic instruction in the source code counts as one operation and that SIMD instructions count as one instruction. Only the statements in the body of the for-loop in `run` will be considered. There are three lines to consider:

```
  73 | dot += R[row*N + col] * x[col];
  76 | xtmp[row] = (b[row] - dot) /
D[row];
  80 | sqdiff += (x[row] - xtmp[row]) *
(x[row] - xtmp[row]);
```

As single precision floating points are used, the total bytes loaded and stored is $4*9=36$. Load/stores to `dot` and `sqdiff` are not counted as they are local variables which will be in registers. The total number of arithmetic instructions is 8. Therefore, AI equals $8/36=0.22$ single precision floating point operations per byte. This is low and suggests Jacobi will be memory bandwidth bound. By observing the Roofline curve in Figure 8, it can be seen that this Jacobi implementation sits to the left of the ridge point and is hence memory bandwidth bound.

This implies running the Jacobi implementation on a node in BlueCrystalp3 requires enough accesses to some level of the memory hierarchy that execution becomes bottlenecked by the bandwidth of the data buses. With increasing number of threads, there are increasing number of cores executing the code. BlueCrystalp3 has a non-uniform memory architecture, where each core has its own cache. However, cache misses will definitely occur and threads will have to compete for access to higher levels of the memory hierarchy. (Note, if one thread maximised memory bandwidth on its own, then adding more threads operating on different data could not increase runtime). Therefore, as increasing number of threads results in more competition for bandwidth, and bandwidth does become limiting, this is indeed a contributing factor to the runtime of Jacobi scaling sublinearly with increasing cores.

4 Conclusion

In this report, it has been explained how to parallelise an optimised serial implementation of the Jacobi iterative method. Parallelisation was targeted for running on one node of the BlueCrystalp3 supercomputer and was realised with the shared memory paradigm, implemented with OpenMP by distributing loop iterations between threads. Load balancing with static scheduling, using reductions to avoid false sharing, performing SIMD on inner loops per thread, thread affinity and assigning memory close to cores that will use it to exploit the NUMA design of BlueCrystal contributed to a final runtime of 7.1 seconds for a 4000x4000 array with 16 cores. An analysis of the scalability of this solution from 1 to 16 cores was also presented. Discussions in the context of Amdahl's Law and the Roofline Model were given to explain sublinear scaling. It was found that this is due to load imbalance, overhead of creating, destroying and synchronising threads and becoming bottlenecked by memory bandwidth.