Assignment 2 - OpenMP Parallelism with Jacobi Code

Luke Storry (LS14172)

December 2017

Introduction

This report describes a series of experiments and parallel optimisations using OpenMP that were applied to a `jacobi.c` code file (with previous serial optimisations), to make it run efficiently across the 16 cores of a single Intel SandyBridge blade on the BlueCrystalPhase3 supercomputer.

This code performs the iterative Jacobi method[1] of solving a diagonally-dominant system of linear equations, which includes repeated independent iterations over the rows of a matrix, which make it a great candidate for parallelisation.

Open MP Pragmas

The OpenMP `#pragma` compiler directives can be used to both fork threads and assign work to threads, as well as describe various execution-control and memory-assignment actions.[2]

Parallel Rows

The obvious starting point was to wrap the row-iterating `for-loop` with both a `parallel` pragma, which forks a specified number of threads, and a worksharing `for` pragma, which specifies that each iteration of the following loop with be executed by a different thread in the team. The shortcut to combine both of these together is `#pragma parallel for`, with a list of clauses, to define how many threads to use and which variables should be private to the threads.

This worked surprisingly nicely, with the simple nature of the Jacobi algorithm implying that no mid-loop synchronisation or execution control was required, each matrix row can be handled cleanly and separately as the cells contribute to thread-local results, which are then collated into exclusive cells in the shared `x` array, causing none of the usual parallel memory issues.

For a `4000x4000` matrix, just this one line of code resulted in a 2.6x speedup, from 46.7s in serial, to 17.6s when run across 16 cores.

Inner loop

Next was an attempt to parallelise the loops that iterate through the cells on each row. However, the Jacobi algorithm specifies a simple multiplication per cell, to be accumulated across each row, there isn't much time-increase to be gained from running the inner loop in parallel, but doing so could give easily rise to memory issues and race conditions.

One potentially useful pragma command was `omp simd`, which can be used to vectorise loops to be run concurrently, however I had already utilised vectorisation as one of my serial optimisations through the use of the `-O3` compiler flag, so `simd` led to either erroneous results, or slower runtimes.

[1] `https://en.wikipedia.org/wiki/Jacobi_method`
[2] `https://software.intel.com/en-us/node/522685`

Scheduling

OpenMP gives us four different options for scheduling the these `parallel for` loops.[3]

First, an overview: The `static` option naively divides the loop into equally-sized chunks, giving the same number of iterations to each thread. Alternatively, `dynamic` gives a small chunk of iterations to each thread, which then request successive chunks upon completion. Choosing `guided` scheduling is similar, but with gradually decreasing chunk sizes to help load-balancing. To simply give the compiler the choice of mapping iterations to threads, the `auto` option can be chosen.

The extra overhead required for `dynamic` and `guided` seemed a good trade-off compared to the naivety of the `static` approach, giving 1.07x and 1.15x speedups respectively, but `auto` resulted in a good middle ground with good scheduling but lower overheads to come out in the lead with a speedup of 1.17x.

However, this is only the case with the original badly-initialised input matrix: once later optimisations spread the data initialisation across the NUMA memory nodes, the `static` scheduling was the fastest, with a speedup of 1.1x over all the other options. This is due to the Jacobi algorithm having the same number of operations for every loop, so as long as every thread has the same memory access time, the load is balanced, and the overhead of load-balancing is wasted.

Combining for Loops

At the end of every full iteration, the rows are again looped through to calculate their squared difference, as a check for convergence. Although it resulted in no measurable speed increase, it still made sense to combine these two identical for loops into one, and calculate the square difference on the fly.

Reduction

This created some errors with different threads having different values for `sqdiff`, so to prevent race conditions and/ false sharing, I utilised OpenMP's reduction operator over the `pragma`, which keeps track of each thread's square-differences, then automatically combines them upon completion.

Data Initialisation across Nodes

The main load-imbalance in the system, as mentioned earlier, was due to the data being ineffectively initialised across the Non-Uniform Memory Access (NUMA) architecture.[4]

In the original code, after memory is allocated, the input matrix is initialised with random numbers. However, due to Linux's default first-touch memory allocation policy, the memory isn't physically allocated at the point of `malloc`, but only after data is written to it by a process.[5] This default usually facilitates good memory-process affinity: by waiting to see what process will be using the data before physically allocating it on the disk, it supports NUMA architectures by placing data close to where it is needed to reduce networking overhead.

However, in the given jacobi code with its serial initialisation, this causes the whole matrix to be associated with the socket of the thread that runs the initialisation, and physically allocated near the NUMA node of the core that thread was run on. This not only causes massive memory-access load-inbalances between the cores, but also a lot of overhead, cache-trashing and potential bottlenecks, as this one source of data is repeatedly accessed from different NUMA nodes.

To fix this, I needed to initialise the array in parallel, splitting the data placement across as many cores as are being used for the calculations, balancing the memory load. `C`'s randomisation is not thread safe, so after the `malloc`, the ma-

[3] `https://software.intel.com/en-us/articles/openmp-loop-scheduling`
[4] `https://software.intel.com/en-us/articles/optimizing-applications-for-numa`
[5] `http://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-affinity.html#First-touch`

trix is pre-initialised to zero using the same `omp pragma parallel for` loop as above, giving the relevant cores their required first-touch to place the memory, before then initialising the matrix with random numbers with the original serial for loop.

This resulted in a total speedup of 1.42x for a 4000x4000 matrix across 16 cores, bringing the overall time down from 17.6s to 12.4s.

Thread Affinity

Leading on from the now efficient data placement upon initialisation, I looked into preventing threads from migrating, and keeping a consistent memory affinity.

Due to the static scheduling of my `pragma parallel for`, the jacobi for-loop's iterations will be chunked identically every iteration. Then by ensuring that the chunk-to-thread mapping remains constant for every iteration, each thread would be accessing the same chunk of rows (and thus the same memory locations) every iteration.

Enabling the environment variable `OMP_PROC_BIND`[6] prevents threads from migrating to a different place, and resulted in a 1.4x speedup in my code. As well as just `TRUE` or `FALSE`, it is also possible to use this variable to define where and how far apart the threads are bound to the core. Binding threads to the same place as the master, with the `master` command obviously worsening the load-balancing, with many threads sharing the same physical process, a 3x slowdown was observed. Setting the threads to successive places with `CLOSE`, or separating them with `SPREAD` had differing outcomes depending on the size or the input and the number of cores being used, so for my final submission I left it as `TRUE`, to let the compiler decide at runtime which method is best (icc usually sets `KMP_AFFINITY=scatter`). This marginally gave the best all-round results, with an average speedup of 1.01x.

To define how large the "places" are that the threads are bound to, there is another environment variable, `OMP_PLACES`. This can accept a explicit list of integers, but for better flexibility, more abstract names are also accepted: `threads`, `cores`, or `sockets`.[7]

Because both the L1 and L2 caches are shared amongst the whole core, limiting software threads to just one hardware `thread` offered no benefits, wheras repeated experiments found that assigning only one thread per core `core` gave a performance boost. With `OMP_PLACES=sockets`, the L3 cache is fast enough that even when threads can migrate between cores on the CPU, it is only marginally less performant than limiting to a single core, but may cause scaling problems because the likelihood of thread collisions is higher.[8]

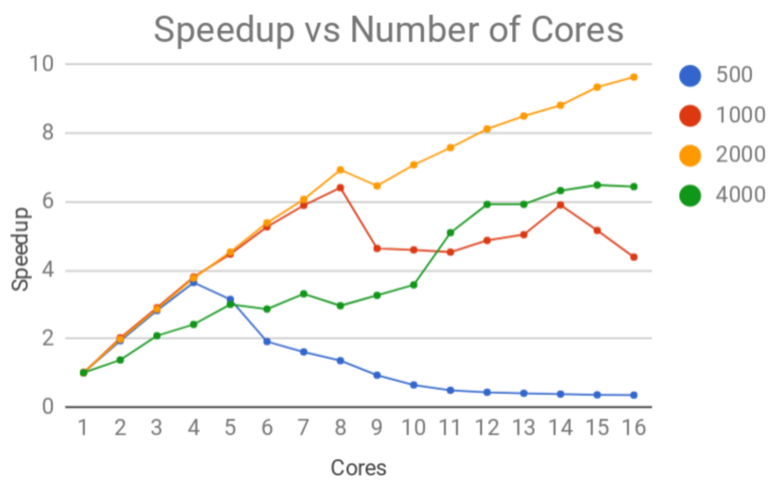
Final Analysis

Scalability

As can be seen from the below graph, although every size of input matrix begins with linearly positive scaling, the smaller matrices platau or start declining after a certain optimum number of cores. This is due to the tradeoff of getting extra computing power, but having the increased overhead of messaging, plus splitting the very fast caches up into smaller, less effective chunks.

For a matrix of size `500x500`, the speedup declines drastically with larger amounts of cores because the threads are spread evenly across all cores. When stored as floats, two full rows of this matrix are 32KB, which fits perfectly into the level one cache on Blue Crystal's Xeon chips, and the whole matrix, at 8MB, fits in the level three cache.

[6] `https://software.intel.com/en-us/node/695719#OMP_PROC_BIND`
[7] Using OpenMP–The Next Step, Ruud van der Pas, Eric Stotzer and Christian Terboven, p165
[8] `http://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-affinity.html`



Cores	Solve Time (s)
1	46.87
2	34.17
3	22.53
4	19.43
5	15.63
6	16.38
7	14.17
8	15.84
9	14.37
10	13.13
11	9.20
12	7.91
13	7.91
14	7.41
15	7.34
16	7.32

Table of the time taken to solve a `4000x4000` matrix with my jacobi implementation, from one to 16 cores, in increments of one core:

Evenly spreading the data out over the NOMA architecture may be good for larger datasets, but for this small matrix it prevents the cache from being utilised, whilst adding lots of overhead.

Similar drop-offs can be seen at 9 and 15 cores on the results of the `1000x1000` matrix. The sharp drop between 8 and 9 cores is due to passing over into a second CPU for the 9th core, which brings a lot of additional overhead without much extra computational power. Then a gradually increasing plateau for shows that extra cores do outweigh the overhead, at least until 15 are reached. At that point the bandwidth-limiting is too much.

The line of results from the `2000x2000` matrix follows a much more more linear scalability, with with only a small dip when the second CPU is added, but then continuing on upwards after that. The `4000x4000` has slightly more sublinear scaling than the `2000` because it doesn't fit into the cache as easily, so is a lot more memory-bandwidth-bound.

Operational Intensity

For each inner loop of the Jacobi iteration, there are two floating point operations: one addition and one multiplication, so the Workload is $2n^2$ for the whole loop. There are two memory reads and one memory write for each cell, so the entire memory access is $3n^2$.

$$\begin{aligned}\text{Operational Intensity}\\ &= \frac{\text{Floating Point Operations}}{\text{Bytes to DRAM}}\\ &= \frac{2}{8}\\ &= 0.25\end{aligned}$$

This implementation is usually memory-bandwidth-bound, unless the matrix size fits well into a cache, which speeds things up enough for the computation to become the limitation.