Coursework 2: OpenMP

High Performance Computing COMS30005

Goce Dimitrov (gd14470)

For the previous coursework we were asked to optimise the Jacobi solver as much as possible in serial, but there is so much that only one core can do and this is where the need for parallelisation comes in.

In this report I will talk about how I used my optimised version of the Jacobi solver and parallelised it using the OpenMP application programming interface (API). This is done by adding a set of compiler directives called pragmas that are incorporated at compile-time to generate a multi-threaded version of my code.

1. Parallelising the Jacobi solver

1.1. Preparing for parallel optimisations

In order to activate the OpenMP extensions, the compile-time flag "-qopenmp" needed to be specified. This arranges for automatic linking of the OpenMP runtime library. Thus, enabling the paralleliser to generate multi-threaded code based on OpenMP directives. I specified the number of threads in my job submission script and set it to 16 for the duration of my code optimisation in order to utilise the maximum that the node can offer. Also, the matrix size used for testing out my optimisations and speedups for the report was 4000x4000.

1.2. Creating parallel threads

After enabling OpenMP I needed to tell the compiler to parallelise the two for loops in the solver since they were the most time consuming in the program. This is done by adding the "omp parallelise" pragma in the outer loop which tells the compiler that the structured block should be run in parallel by a team of threads.

By doing this my runtime got around 2.1 times slower. It went from 38.993s to 84.473s. This is because I am only having my newly created threads executing the exact same steps in that parallel region which is not what I wanted. Normally I would want each thread to execute a different task, or work on a separate piece of the problem.

1.3. Making each thread to execute a different task

In order to fix this, I added the "for" clause to the directive mentioned above. This divided the loop iterations between the already created team of threads. When running my code, I got a completely different result which was incorrect. This was because of synchronisation bug (race conditions). Meaning variables were getting updated by different threads at the same time and because of this I ended up with a completely different result.

1.4. Race conditions

The variable causing race conditions was "sqdiff" it was being updated by 2 or more threads every time and giving a wrong answer. By using the reduction clause each thread receives its own local copy of the reduction variable and the thread modifies only the local copy of this variable. Therefore, there is no data race. At the end the threads join together and all the local copies of the reduction variable are combined to the global variable.

This fixed the race conditions and it reduced my runtime to 18.073s or 2.15 times faster than my serial implementation without changes to my iterations number or solution error, but there was also another problem in the two parallel for loops called false sharing.

1.5. False sharing

The index variables in the for loops and "dot" were being shared by all threads and causing false sharing. Intel processors follow the MESI (Modified/Exclusive/Shared/ Invalid) protocol. So, when a variable first reads from a cache line, the cache line gets marked as 'Exclusive' access. As long as it is marked exclusive, subsequent loads are free to use the existing data in cache. If that cache line gets loaded by another processor on the bus, it gets marked as 'Shared' access. If the cache line marked as 'Shared' gets "Modified" all other processors are sent an 'Invalid' cache line message. If the processor sees the same cache line which is now marked 'Modified' being accessed by another processor, the processor stores the cache line back to memory and marks its cache line as 'Shared'. The other processor that is accessing the same cache line incurs a cache miss. The frequent coordination required between processors when cache lines are marked 'Invalid' requires cache lines to be written to memory and subsequently loaded can significantly degrade performance.

This was fixed by declaring the variables inside the parallel region or by declaring them as private in my loop pragma. I used both ways to give each thread their own local copy of the private variables. This reduced runtime to 17.774s without any change to my solution error or iterations number which gave me a speed increase by 1.01 times. This was not as much as I expected but it was still an improvement.

1.6. Interesting case of false sharing

After playing around with the "dot" variable like changing it to shared nothing happened. This piqued my interest so I did a little bit of research and came to the conclusion that this happened by chance on this particular code. So, when "dot" gets updated it gets saved in L1 or L2 cache it then updates a variable and gets reset immediately after this. Because of that there is no time for the variable to go to L3 or DRAM to get updated from there with an invalid value. If there were more operations using "dot" I would have had problems with it being shared for sure and because of this I decided not to leave it to chance, but to set it as private.

2. Load Imbalance

After successfully parallelising the loops in the Jacobi solver I wanted to see if there is anything else I can do to my code to improve my timings. The most helpful was the VTune Amplifier. Its report showed me that my Jacobi solver had load imbalance.

When there is load imbalance, unequal amount of work is being assigned to the threads. Because of this there are idle threads which are wasted resources in the multithreaded executions.

I tried to fix this by using different scheduling like static, dynamic, guided, auto, runtime. All of them slowed down my program except static. But even static did not fix my load imbalance problems, instead runtime stayed the same. This was because static scheduling is defined by default in a parallel region. Why this one did not slow my program is because it divides the total number of tasks into (nearly) equally sized groups assigned to each thread. Because all iterations have the same computational cost this is an appropriate scheduling. For which in my case they do. After scheduling did not fix my load imbalance problem I found out about NUMA (Non-uniform memory access architecture) and its first-touch allocation.

3. First Touch Allocation (NUMA)

Computing systems that are composed of several nodes in such a way that the aggregate memory is shared between all processors, but for each processor access to the memory modules that are local to them is much faster than to access the memory of another processor.

The problem I encountered here is the first-touch allocation policy. Basically, what this means is that when calling "malloc" in Linux or Windows I am telling the operating system that I want to use a buffer of a certain size, however the operating system does not create virtual memory pages during "malloc". Pages are created and assigned to physical memory modules during first touch in other words when the program writes data to the memory for the first time it allocates it using the first touch policy. This means that the NUMA node that touched a memory page first will get it allocated in its local memory. This is a bad thing because the array "A" in my code gets processed in parallel using multiple threads, but it gets initialised in serial elsewhere. Because of this the entire array will be placed on the NUMA node that touched it. This means that the array will be placed in memory in that specific CPU that worked on its initialisation. This is bad since when the array gets processed by multiple threads by the other CPU it will have a slow access to the data.

In order to fix this, I found out that initialising the whole array as 0 in parallel before it starts getting its actual values was easiest, since I do not have to worry about parallelisation of random values. This way the parts that were touched by each CPU will go to their own local memory.

By doing this my program's performance increased quite a lot. It went from 17.774s to 7.389s and I got a speed up of 2.4 times.

This was my last code optimisation, so after this I tested my code a couple of times just to make sure it works. The interesting thing I noticed was that my runtime was fluctuating. In order to fix this, I found the environment variable extension from the Intel Compiler called "KPM_AFFINITY" that restricts execution of certain threads (virtual execution units) to a subset of the physical processing units in a multiprocessor computer.

4. Processor Affinity

The decision of choosing the thread granularity type was up to whether Hyper-threading is enabled or disabled on Blue Crystal 3. Since it is disabled both types would give me the same result. This is because in "core" granularity the threads can be run on any hardware threads in the same core, but with "fine/thread" it runs on a specific one. Because I said Hyper-threading is disabled this would not make a difference so I did not specify and left it to default. Now the decision came down to whether I should use "compact" or "scatter" affinity.

I made my decision by testing out both affinity types and looking at their timings for matrix sizes of (4000x4000). I used a scatter plot only for the (4000x4000) one, because there was no benefit from the smaller matrix sizes because their timings were not that different. The lines were overlapping when plotted.

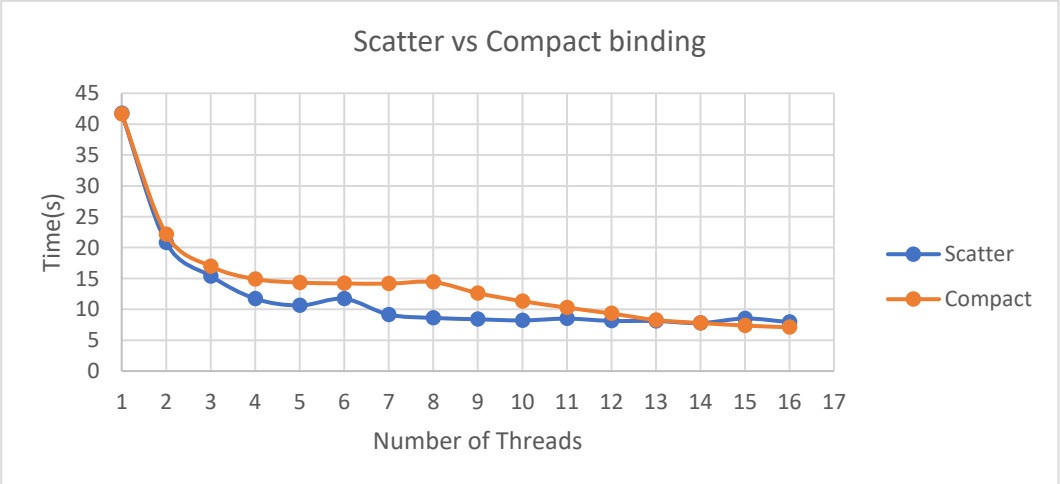

Figure 1: Timings of both affinity types using my final parallelised code on a 4000x4000 matrix

By looking at the graph above it can be clearly seen that the ”scatter” affinity type is a bit faster than the “compact” for lower number of threads, but when the thread numbers gets high enough ”compact” becomes slightly faster than “scatter”. This is because compact tries to place the threads as close as possible to one another, meaning that it tries to put as many as it can in one socket. Because of this the bandwidth in the other socket gets unused if all threads could fit the first socket. On the other hand, “scatter” scatters them between both sockets and thus giving them higher available bandwidth to use. I decided to use “compact” for my final implementation, because all threads are pinned one next to another and because all 16 threads were to be used thus removing the bandwidth issue as well as reduced the fluctuation. This got me a runtime of 7.053s an improvement of 1.04 times than before and without a change in solution error or iterations number.

5. Analysis of core scalability from 1 core up to 16 cores

As it can be seen from the figure below I am comparing three different matrix sizes. The most optimal one is the that is not too big, but also not too small. The 2000x2000 matrix is optimised the most. This is because it does not require as many computations as the 4000x4000 one and it is not wasting resources on too less computations as the 1000x1000. The rise and fall of speedup on the 1000x1000 matrix is due to me selecting the “compact” affinity type. An important thing to mention is that if I used the parallelised code on just one thread it would be slower than my original serial code. This is because all the extra stuff for parallelising is not meant for one thread.

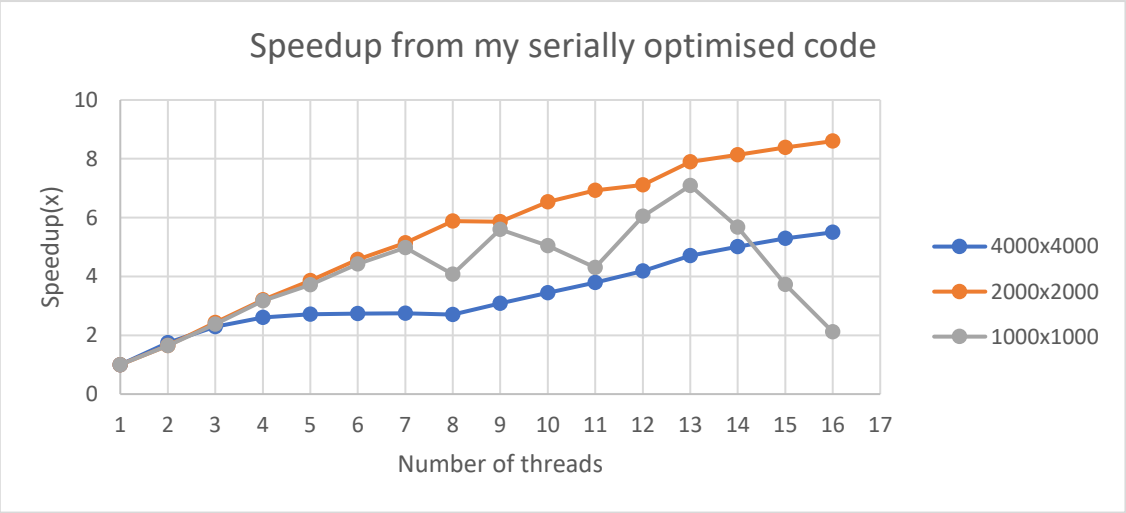

Figure 2: Speedup using 3 different matrix sizes

6. Conclusion

As it can be seen from the scatter plot in figure 2 and the table, parallelising code is quite useful and beneficial. It greatly reduces runtime of programs that need a lot of computations. So, all the extra code was really worth it.

Matrix size/Code	Serial	Parallelised
1000x1000	0.429s	0.218
2000x2000	3.115s	0.362
4000x4000	38.993s	7.053

References:
https://software.intel.com/en-us/articles/avoiding-and-identifying-false-sharing-among-threads