

ENCE464

Embedded Software

Computer Architecture

Assignment 2

Group 13

James Howe - 53290287

Joe Mulder - 63079212

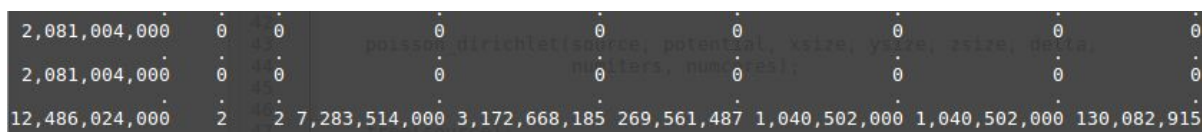
16/10/2018

Introduction

This report describes the implementation of Poisson's equation using the Jacobi relaxation algorithm to have an optimal use of the cache. The algorithm involves iterating through multiple large loops in order to calculate the electric potential in a cube for a given source.

Initial Implementation

To start the assignment, the equation had to be implemented using the given inputs from the poisson test function provided. The source charge or heat source was set as the middle value of the input dimensions, spreading out over the number of iterations. Once implemented with the four for loops, for number of iterations, x, y, and z, the calculated values were output to show the function was working as expected. The print function greatly increased the operation time of the function, so when the values were shown to be correct, the print function was removed. With the help of cachegrind, the number of instructions and the cache information from each line was visible.



2,081,004,000	0	0	0	0	0	0	0	0	0
2,081,004,000	0	0	0	0	0	0	0	0	0
12,486,024,000	2	2	7,283,514,000	3,172,668,185	269,561,487	1,040,502,000	1,040,502,000	130,082,915	

Figure 1: Cachegrind output for initial code implementation.

Figure 1 shows the number of read and write instructions used by the cache in the program when calculating the potential for a 101 size cube. The equation was split up into 6 separate temporary variables that were then added together and multiplied by delta squared over 6. The figure above shows that all of the cache data reads were completed in the final line, which was the poisson equation. The first column is the number of instruction reads, with the fourth column through to the sixth showing, in order, the data reads, the level 1 cache data misses and the last level data misses. This gives a in depth understanding of how efficient the initial implementation is, as the number of cache hits versus the number of misses is low, with almost half of all data reads being missed, even for a small sized cube.

Code Optimisations

The first optimisation made was to change the order of the loops so the value iterated over the most corresponded to the smallest change in the array index. This greatly reduced the number of cache misses as values were more likely to be in the cache during the loop execution due to the temporal locality concept being relied on by the cache when it loads data. The array is initiated to have the x dimension values first, the y dimension second and the z last. When iterated over, the z value was changed to be the value that is in the outer loop, so that the array is now accessed in order rather than skipping large block of values every loop. This means that the data surrounding these read values are also cached, and as they will then be required to be read, the read time will be greatly decreased. With the loops in the wrong order there was an average data cache miss rate of about 50% for a size 101 cube. This miss rate will only increase as the cube size is increased as the locations of the data will be further away from the cache line as the cube grows larger. With the loops in the better order, this miss rate was reduced to 15% for the same sized cube, which gave a 4x speed increase to the program execution.

The equation provided uses a division by six that would need to be calculated for every single iteration. Due to the increased number of instructions that are required for a division of a number that is not a power of two, this means that the whole function is slowed as a result. Instead, the equation is multiplied by the approximate reciprocal value to lower the instruction count. This increased the speed of the program by 15% when testing with a 101 size cube and 1000 iterations.

Every iteration, the potential is accessed multiple different times, each with a changing index. This index value is calculated every time, with slightly different input arguments. Changing the calculation from being hard coded to being an inline function that takes in the input arguments i , j , and k , as well as the box dimensions helped to speed up the program. Due to the way the compiler optimises recurring blocks of code, the use of the function with slightly changing arguments increased the speed of the program by a consistent 20%. The input arguments are only changed slightly with each loop, allowing the function to be optimised. Using this function halved the total number of instructions required, which also helped to reduce the cache miss rate by a factor of 3.7. One thing noticed from this function was that using unsigned integers as the loop iterators increased the program execution time by almost 50% compared with using a signed integer. This was caused by a doubling of the number of instructions, as every loop the values had to be converted to signed integers due to the function implementation. This was fixed by changing the function return value into an unsigned integer rather than a signed integer.

The boundary conditions of the rectangular box meant that for each iteration, each value would have to be checked if it was on the boundary. The initial solution to this issue was to have an if statement checking every temporary equation to see if it needs to be set to zero. With this setup, every single time the equation was calculated, there were 6 if statements executed. This greatly decreased the efficiency of the pipeline as it introduced many control hazards, due to having to stall and undo previous instructions in the branches. To combat this issue, when the initial dimensions were set they had an extra two added to them, in order to pad what would be both sides of a cube along each axis with extra zeros. This means that what used to be boundary values are now one value further in and the loops no longer access those zero values, removing the issue with non-zero boundary conditions.

The program was compiled using the `-O3` flag to enable a large number of compiler optimisations in order to increase the performance of the code. Many of these flags seemed to only provide a small improvement to the code performance but some, especially from the `-O1` level, provided a large speed increase. The difference between no optimisations and the `-O1` level of optimisations reduces the execution time of the program by a factor of 10, while the difference between the `-O1` and `-O3` was only a factor of 1.2. Some of these optimisations performed tasks such as loop unrolling and loop reversal which help to reduce the cache miss rate from loops in the program.

Threading

As a way to speed up the execution time of the program, threads were used to split the work up between the multiple CPU cores. This helps to reduce the cache usage of each individual CPU, meaning that the average number of cache misses will be reduced, and will help to speed up the calculations themselves as each core is working concurrently. In order to implement this the cube is split up into smaller sections that each thread will work to calculate the potential of.

Figure 2 below shows the change in execution time as more threads are added to the program for varying sizes of cubes. These times were found when using 100 iterations as the overall effect of threading is independent on the number of iterations. Note that the y-axis scale of the second graph is much smaller than the scale of the first graph.

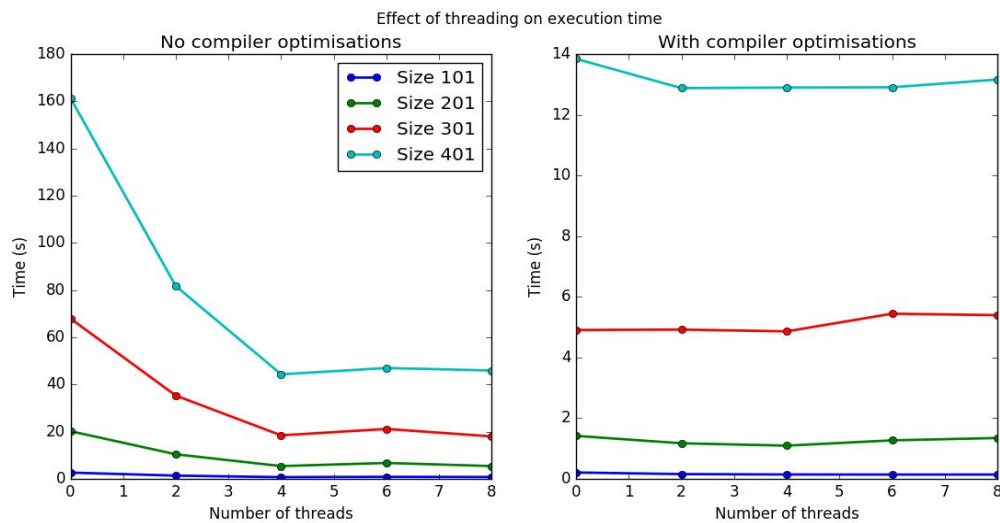


Figure 2: Graphs showing the effect of adding more threads to the program.

It can be seen that with no optimisations going up to 4 threads increases the speed by the number of threads used and there is no benefit from using more than 4 threads. This is due to the fact that once four threads are using each CPU, any additional CPU thread is not effective as the CPU resources, the functional units and the caches, are being used at full capacity. This means that any instructions being executed by extra threads are not able to run until these resources are freed. When compiler optimisations are enabled the effect of adding threads is almost negligible but the overall times are far lower than the times without optimisations. This is likely due to one or more of the compiler optimisation flags affecting how the threads operate or making their effect minimal due to how the loops are changed by the optimisation.

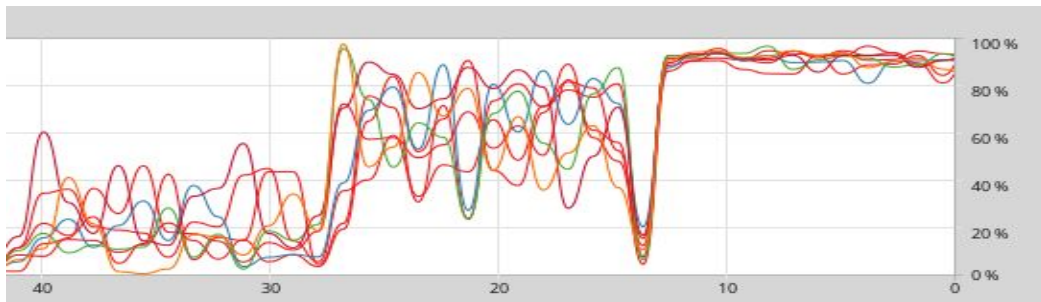


Figure 3: Resource usage for different thread counts.

Figure 3 above shows the different effect on the CPU that varying numbers of threads has. With the use of just one thread, the different CPUs are all being used, with the usage averaging around 20%. Changing the program to use 4 threads meant that the CPU usage jumped greatly to average around 60%, while using 8 threads pushes the CPU usage up close to 100%. This does not line up with the results from Figure 2, as the increased CPU usage across the cores would logically mean a low runtime. The results instead show that the time taken to execute is the same where the average CPU usage is 60% as nearly 100% usage.

The way that the threads were implemented meant that each iteration of the outer loop was split up into even sections for each thread to work on. Having each thread accessing data that was not close in proximity meant that each thread would be overwriting the other threads data that is stored in the lower level cache, as it is shared between the cores. This meant that having more threads was less effective than expected. Changing the way the outer loop was split up, so that each thread would alternate in order along the outer dimension would allow for temporal locality to be utilised more effectively. Each thread would only be slightly further along the array from the previous thread, allowing for a speed increase as the cache would no longer be overwritten with every thread execution.

Results

Figure 4 below shows two graphs demonstrating the relationships between the program execution times with the size of the cube and the number of iterations.

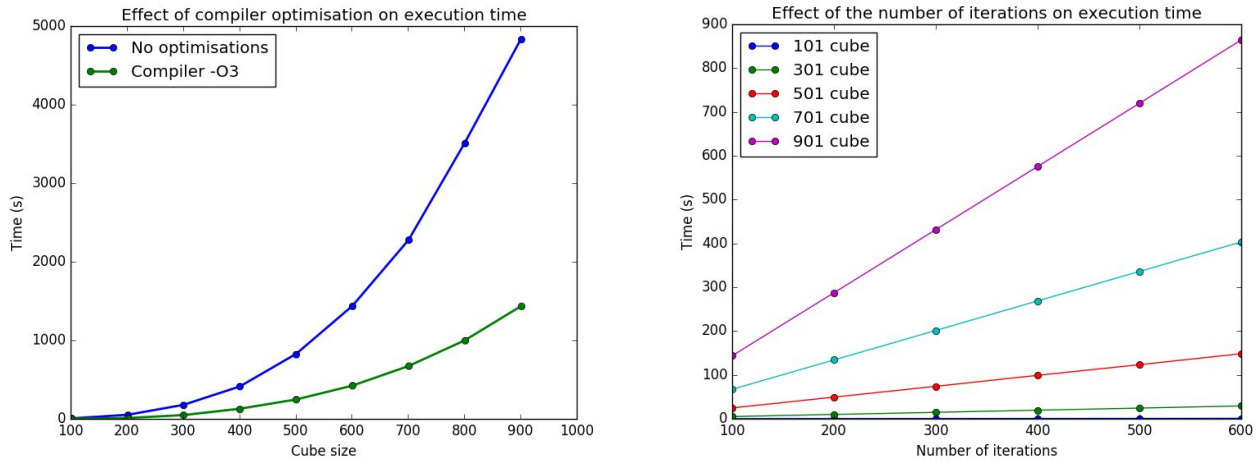


Figure 4: Graphs showing relationships between cube size and total iterations on execution time

The graph on the left shows the cubic relationship between the size of the cube and the time it takes for the program to execute. It also shows the difference between the time for the program to execute with the compiler optimisations turned on and with them turned off. The graph on the right shows the linear relationship between the number of iterations and the execution time of the program for various sized cubes.

Both of these trends were expected due to how the Jacobi Iteration problem is calculated, and can help to verify that the program was still properly calculating the potential values when the cubes that could not be visually checked due to the large size of them.

The means and standard deviations of the times to run 1000 iterations using the fastest version of the program with varying cube sizes is given in table 1 below. This was found using 4 threads and the compiler optimisations on as the threads started to have a slight impact with larger cube sizes.

Table 1: Means and standard deviation of the execution times for various sized cubes

Size	101	201	301	401	501	601	701	801	901
Mean	1.43	11.06	48.47	129.52	247.13	423.19	672.99	1000.00	1435.53
Std.dev	0.00694	0.05935	0.12527	0.35896	0.33633	0.79124	0.95783	1.19843	1.27752

The best time for the 901 size cube to execute with 1000 iterations was 1435 seconds when the optimisations were on. The standard deviations are not very large compared to the mean execution times, with the standard deviation of the 901 size cube being less than 0.1% of the total execution time.

Conclusion

Understanding of the cache and compiler optimisations allowed for a major speed increase in the execution of a program. Small changes in how a program is written, such as the order of loops, can make a huge difference in the overall efficiency as the compiler can often not make these changes itself. These changes helped increase the cache hit rate, improving the speed of the program as reads are no longer being missed and undergoing miss penalties.