

CM30225 Parallel Computing

Assessed Courseork Assignment 1

November 15, 2016

1 Approach

The relaxation problem involves repetadly replacing each cell in a matrix with the average of its four neighbours until a given precision is reached. Firstly to solve the probelm using a serial approach each cell is processed row by row. After each cell has its value replaced the diffrence between the previous and new value is calculated and checked to see if it is less than the precision value. When every cell is replaced by a number smaller than the precision the computation stops.

In order to store the matrix two double**'s are created one to record the result of the last iteration and one to store all the new values in, after each iteration the pointers for the two matrix's are swapped.

2 Parralisation Technique

In order to parallelise the problem it was decided to spilt the matrix up into rows then give each thread a number of these rows. More specifically each thread is given a starting row, which is applies relaxation too, then adds the number of threads to the starting row to get the next row to compute on. So if 4 threads are used on a 14, the two end rows are fixed and don't require relaxation, row matrix the rows are split up like so:

rowNumber	1	2	3	4	5	6	7	8	9	10	11	12	13	14
thread		1	2	3	4	1	2	3	4	1	2	3	4	

This splits the rows evenly between the given number of threads. If the number of number of threads is more than the size of the matrix then the number of threads is reset to the size of the matrix.

Each thread is allowed to run at its own speed while it computes the relaxation for its set of rows. The threads are then all synchronised using a barrier before they can move on to the next iteration. After all rows have been

computed the read and write matrixes must be swapped this is a serial operation so only one thread performs is while the rest wait. To make sure only one thread performs this the C function `pthread_barrier_wait` returns the value `PTHREAD_BARRIER_SERIAL_THREAD` to one thread, the thread that gets given this is the one which will perform the serial step. The pseudocode below shows how this is implemented.

```
while (precision_not_met)
{
    doRelaxation();

    serial = barrier_wait();

    if (serial == PTHREAD_BARRIER_SERIAL_THREAD)
    {
        doSerialWork();
    }

    barrier_wait();
}
```

A quick note is that the program will actual use the number of thread plus the main thread, it was decided not to change this as the main thread just will be de-scheduled while it waits for all the other threads to return so won't effect the investigation significantly.

Finally to check if the precision has been made a global variable is used by all threads to show whether the program should continue. This global value is set to 0 before each iteration then if any of the threads do not meet the precision they set it to 1. This means that if by the end of the iteration the value is 1 then the computation needs to continue.

3 Avoiding Race Conditions

There are a few places where race conditions are possible:

1. While thinking about this problem there were two options firstly to read and write to the same matrix as this would probably tend towards the same answer. However as multiple threads could be trying to read and write to the same cell at once the program could yield different results each time unless the matrix was protected by a lock, which would add overhead. Therefore two matrices are used so one can be used to read from and then one to write to. This avoids threads trying to write and read to the same cell simultaneously, simultaneous reads are possible and not a problem.
2. Once a thread has finished it sets a global variable “cont” to 1 if the precision was not met. this variable does not need a lock as if one thread resets it after it doesn’t matter as it’s still 1.
3. After all threads have finished they check if the continue value is 0 and if so return. As each thread can update this value then all threads must have finished the iteration before any one can check this value, so a barrier is required before. Also as the next step is to set the value to 0 and continue the next iteration a barrier is required after the check as well to make sure that all threads have checked the value before resetting it.
4. The swapping of the matrix’s and resetting the continue back to 0 needs to be done by a single thread after all threads have finished relaxing. therefore before a `pthread_barrier_wait` appears before and after this serial code (lines 135 and 142).

With this in mind the pseudocode will actually look like this.

```
while (true)
{
    doRelaxation();

    barrier_wait();
    if (precision_met)
    {
        return;
    }

    serial = barrier_wait();

    if (serial == PTHREAD_BARRIER_SERIAL_THREAD)
    {
```

```
        swapMatrixs ();  
        precision_met = 0;  
    }  
  
    barrier_wait ();  
}
```

4 Correctness Testing

In order to test the program is correctly computing the answer the problem was split into three separate sections which can be tested separately. Firstly the relaxation, in order to test this the first three iterations of a 5 x 5 matrix bounded by all 1's were hand calculated then compared to the output from my program running with one thread. For example in the first iteration you would expect 0.5 at the corners $(1 + 1 + 0 + 0 / 4)$, 0.25 on the edges $(1 + 0 + 0 + 0 / 4)$ and zero in the middle. As you can see below the program outputs the correct value.

1	1	1	1	1
1	0.5	0.25	0.5	1
1	0.25	0.0	0.25	1
1	0.5	0.25	0.5	1
1	1	1	1	1

This was continued for the next two iterations, after this it was concluded that the program can correctly calculate each relaxation iteration.

The second step to the correctness testing is that the relaxation stops when a given precision has been reached. To test this a precision is picked, 0.01 for example, then two 3 x 3 matrix was then chosen with bounding values 0.011 and 0.01 when relaxation is performed the first matrix should iterate twice and the second only once as the precision is immediately met.

Below are two tables showing the two runs on the program, firstly with 0.011 bounding and secondly with 0.01.

iteration	difference	precision	continue
1	0.011	0.01	1
2	0.0	0.01	0

iteration	difference	precision	continue
1	0.01	0.01	0

This was tried with precisions 0.1, 0.01, 0.001 and according bounding values and then it was concluded that the relaxation correctly stops when the precision is met.

The Final thing to test is that all of the above still holds for multiple threads, to check this various number of threads were run using the same matrix size and the results were checked for equality. Once this was done for 4x4, 8x8, 10x10 and 20x20 with 2, 4, 8 and 16 threads it was concluded that the program was correctly calculating the relaxation of a matrix to a given precision with any number of threads.

5 Scalability Investigation

5.1 Speedup

Speed up is given as the speed taken to perform a computation on one processor divided by the speed on P processors.

$$S_p = \frac{\text{Time on 1 processor}}{\text{Time on } p \text{ processors}}$$

Given this the speedup was calculated for a 10000 by 10000 matrix for all number of threads between 1 and 16. The results of this are given below.

Threads	Time (S)	Speedup on P processors
1	48.39	1
2	24.67	1.96
3	16.63	2.91
4	12.68	3.82
5	10.26	4.72
6	8.69	5.57
7	7.54	6.42
8	6.71	7.21
9	6.11	7.91
10	5.58	8.66
11	5.34	9.07
12	4.94	9.8
13	4.81	10.07
14	4.65	10.41
15	4.19	11.55
16	4.15	11.65

As you can see from the results the speedup on P processors is very close to the number of processors until and trails off after 14 processors is reached. This is shown more easily by Figure 1.

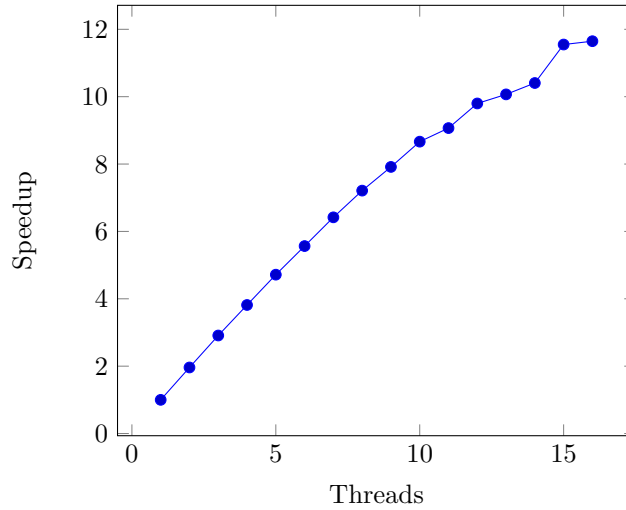


Figure 1: 10000 x 10000, 0.01 precision

5.1.1 Amdahls Law

One of the reasons for the tail off in speedup could be that it is approaching Amdahl's Limit. Amdahl's Law states that there is a limit on speedup because some part of the program must be done in serial. The serial part of this program is the swapping of matrixes and the resetting of the the continue variable.

The theoretical Amdahl Limit was calculated by running the serial code 10,000 times and getting the average length of time (0.00000472 seconds) the number of iterations performed on a 10000 by 10000 matrix with a precision of 0.01 is 37 this gives the serial portion of the computation a time of 0.00017 Seconds.

In theory this limit would be accesable with an infinate number of processors however there are extra overheads which are not taken into account, such as waiting on barriers and also createing and destroying the threads. These overheads will also increase as more threads are added causing the calculated Amdahl to be impossible to get near to with the current program structure.

5.1.2 Slowdown

Another thing noticed while doing the scaleability investigation was on smaller sized arrays, around less than 100 by 100, there was slowdown instead of speedup. This is shown by Figure 2 which was using a 100 by 100 matrix and 0.01 precision.

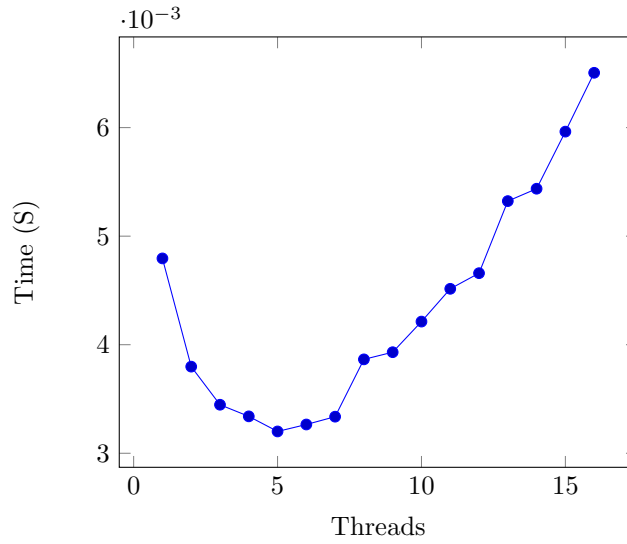


Figure 2: 100 x 100, 0.01 precision

As you can see from the graph initially adding more processors does speed the computation up as you would expect. However after 5 processes are added any more causes slowdown as the overhead of creating the extra thread outweighs the extra computing power gained.

5.1.3 Gustafsons Law

Gustafson's law suggests that if the problem size is increased the percentage of time spent of the sequential part of a program will decrease, therefore giving a better speed up value. To try and show this in terms of the speedup was calculated for differing sizes of matrix, these results are shown in Figure 3.

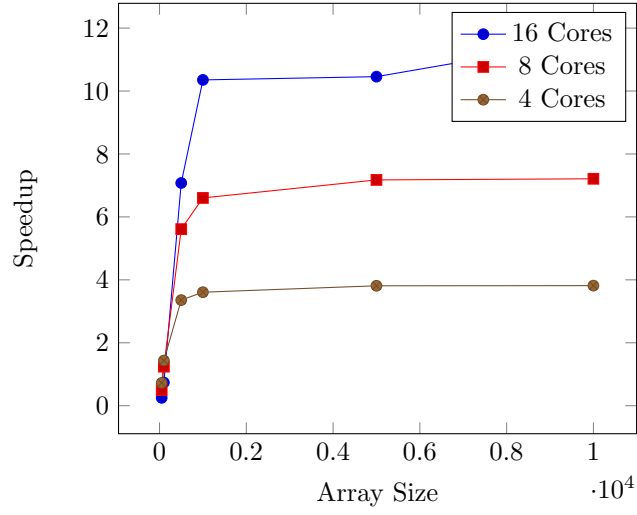


Figure 3: 0.01 precision

These results show that, as Gustafson suggests, as the problem size is increased the speedup on P processors tends towards the number of processors.

5.1.4 Efficiency

Efficiency, speed up per processor is given by the following equation.

$$E = \frac{\text{time on 1 processor}}{p \times \text{time on } P \text{ processors}}$$

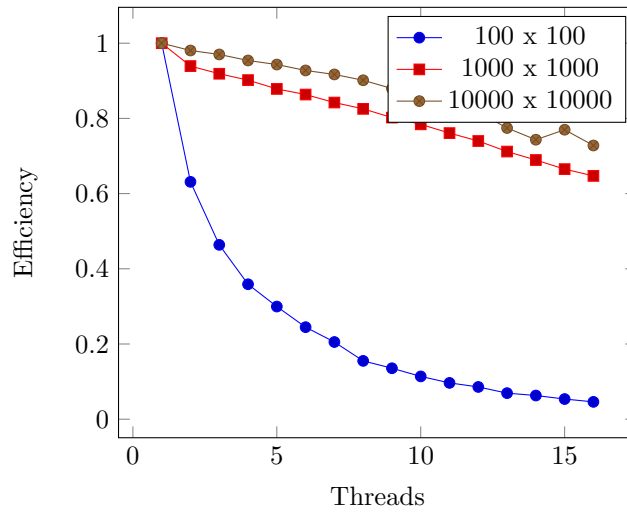


Figure 4: 0.01 precision

As shown in Figure ?? as more threads are added the efficiency drops, this is expected as the more threads there are the more processor time is spent waiting for other threads. How fast the efficiency drops is dependant on the problem size. Using a small problem 100 x 100 the efficiency drops very quickly as the percentage part of the problem which can be done in parallel is smaller so the threads spend a larger portion of time waiting for the serial part to complete, larger problem sizes drop off slower.

5.1.5 Karp-Flatt

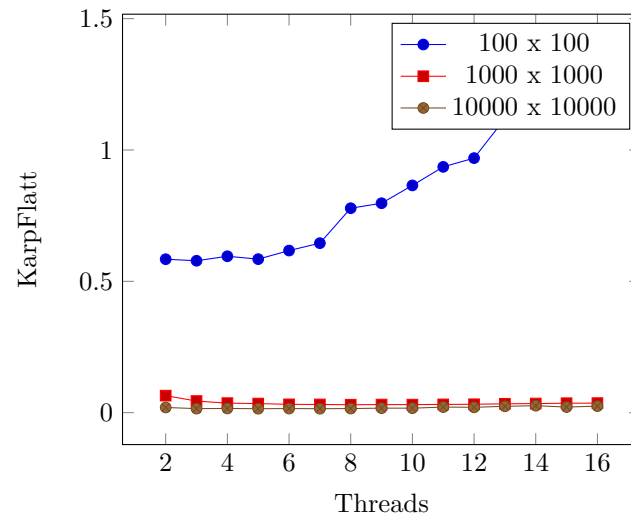


Figure 5: 0.01 precision