CM30225 – Parallel Computing

Memory Sharing in C

Abstract

An example of parallelism and shared memory realised using C programming and POSIX threads. This document demonstrates the effects of parallelism on a variation of the Jacobi relaxation method and tries to identify its efficiency, scalability and shortfalls.

Philo Waddell

University of Bath

Section 1 – Problem Analysis

With a large amount of previous experience with sequential programming in C, it took very little study of parallelism with POSIX threads to decide that the optimal solution would contain minimal parallel aspects. Communicating between threads often requires a large amount of processing overhead which could be considered as 'wasted' given its not directly aiding computation. Other parallel aspects that add to slowdown are thread synchronisation and awaiting read/write access, as these can leave threads idle for hundreds if not thousands of clock cycles.

By heavily relying on constraint conditions to avoid these pitfalls of parallel software design, an optimal solution could be realised with minimal inter-thread communication. Figure 1 shows the concept for load distribution on a 10x10 matrix with 3 threads.

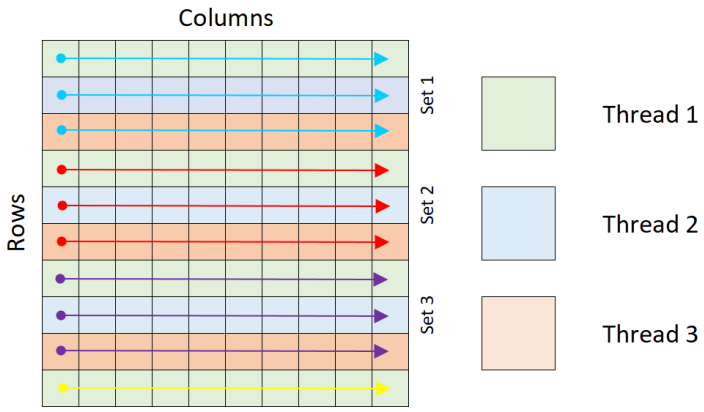

Fig 1 – Load Distribution Diagram

It is apparent from figure 1 that 'Thread 2' and 'Thread 3' are required to wait for 'Thread 1' while it finishes computation on the final row of the matrix. However, the impact of this diminishes as the size of the matrix gets larger.

For this concept to work, a dummy array is required to maintain the original values while the algorithm updates the first array. Once this is complete, the new values can be copied to the dummy array in a similar manner before the process begins again. Each thread must be aware of the cells it has been implicitly allocated to modify.

Section 2 – Software Design

General Structure

The algorithm is split into two functions: relax, which acts as an argument handler and initialises the child threads; and manipulate, the 'multi-threaded' function which manages all computational aspects. Relax generates shared features, such as mutexes, barriers and the dummy array; before packing them into a structure along with the function arguments. The child threads are then generated, passing a pointer to this structure as the function argument. The structure is then unpacked by each thread before any computation is performed.

Thread Identification

As mentioned earlier, it is important for each thread to be aware of the set of data it has been allocated to modify to prevent race conditions from occurring when preforming computation. To solve this, each thread provides itself with a unique identification number separate to its thread ID. This new number is the threads private ID and directly corresponds to the row numbers that this thread will be operating on. The code which allows each thread to do this is shown below:

```
/* Register each thread with a private ID */
pthread_mutex_lock(&locks[0]);
      pid = *active;
      *active += 1;
pthread_mutex_unlock(&locks[0]);
```

Fig 2 – PID Code Snippet

This code runs immediately after the argument structure has been unpacked, theoretically while the next thread is still being created or unpacking the arguments itself. This reduces the impact of using a mutex around these two lines of code. In this snippet, variable 'pid' stores the current value of the shared variable 'active' before incrementing it by one. 'Active' is initialised at zero so that each thread is given a 'pid' from 0 to N, where N is the total number of threads.

Parallel Data Manipulation

Due to the constraints put in place in the form of the threads private id, communication between threads is kept absolutely minimal during the main computational process. A single shared flag indicates whether or not the required level of precision has been achieved by all cells. Read/write access is separated with barriers so that all threads are either writing or reading at any one time. Simultaneous writing is safe in this instance as the previous value of the flag is considered irrelevant, unless it is being reset in which case all threads do so immediately after synchronisation.

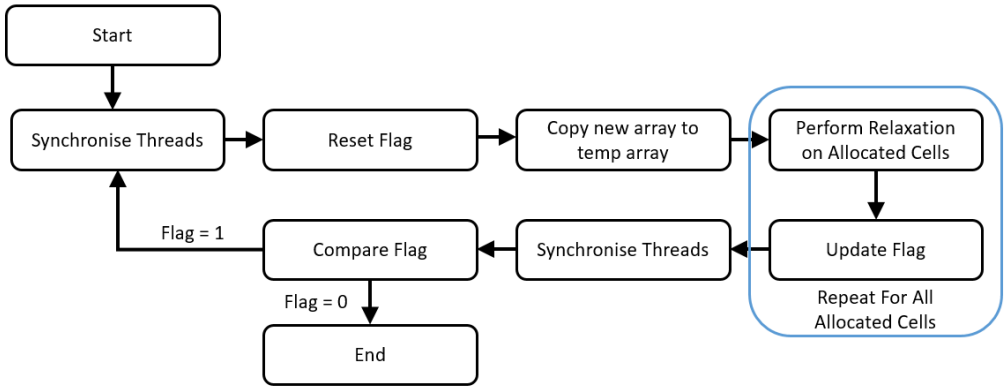

Fig 3 – Parallel Computation Flow Chart

The flowchart above shows the fundamental operation of the manipulation function once the argument structure has been unpacked and the private IDs assigned. The prevention of race conditions is underpinned by the synchronisation of the threads either side of the flag comparison and the private ID constraints.

The threads must be synchronised either side of the flag comparison to prevent an infinite wait occurring. If the threads are not synchronised before, a thread could exit before all threads have finished updating the flag. If the threads are no synchronised after, a thread could reset the flag before all threads have completed their comparison.

The private ID constraints are implemented within the for loops that increment the value of the current working row, these are used both during the copying and computation stages. The implementation of the parallel computation is shown in figure 4.

```
while(1)
{
	/* Wait for all threads to arrive */
	pthread_barrier_wait(barrier);
	*flag = 0;		// Lower flag

	/* Thread manages its allocated rows */
	for (i = 1 + pid; i < d - 1; i = i + threads){
		/* Copy all editable cells from that row */
		for (j = 1; j < d - 1; j++)
			tmp[i][j] = arr[i][j];	// Copy
	}

	/* Perform relaxation on allocated rows */
	for (i = 1 + pid; i < d - 1; i = i + threads){
		/* Manipulate all editable cells from that row */
		for (j = 1; j < d - 1; j++){
			/* Calculate from tmp array average and overwite arr */
			arr[i][j] = (tmp[i-1][j] + tmp[i+1][j] + tmp[i][j-1] + tmp[i][j+1]) / 4;
			/* Raise flag if precision not achieved on this cell */
			if(*flag == 0 && fabs(tmp[i][j] - arr[i][j]) > p)
				*flag = 1;
		}
	}

	/* Wait for all threads to complete computation */
	pthread_barrier_wait(barrier);
	if(*flag == 0)
		break;		// Break if flag is still low*/
}
```

Fig 4 – Parallel Computation Code

In figure 4, loops referring to the integer ‘i’ dictate the current working row for that given thread. These loops are initialised at 1 + the threads own pid and incremented in steps of ‘thread’, the total number of threads working on the computation. Figure 5 shows the ‘i’ values across 4 threads over 6 loops.

	Working Row					
	Loop 1	Loop 2	Loop 3	Loop 4	Loop 5	Loop 6
Thread 1 (PID: 0)	1	5	9	13	17	21
Thread 2 (PID: 1)	2	6	10	14	18	22
Thread 3 (PID: 2)	3	7	11	15	19	23
Thread 4 (PID: 3)	4	8	12	16	20	24

Fig 5 – i Values Example

Section 3 – Testing

Large Data Utilising Different Numbers of Threads

For in depth testing, the relaxation software was tasked with manipulating identical data utilising 1-16 threads over 10 runs. The data used was a randomly generated array of double values between 0 and 5 with the dimensions of the array being 150x150. The collected data was then used to find an average run time when utilising N threads which was in turn used to calculate the speedup and efficiency of the algorithm.

Threads	Repeat Number										Average	Speedup	Efficiency
	1	2	3	4	5	6	7	8	9	10			
1	13.03003	13.03035	13.02962	13.03007	13.03015	13.02936	13.02939	13.02872	13.02985	13.02850	13.02960	1.00000	100.00000
2	10.50834	10.36951	10.22863	10.55315	10.84034	10.87150	10.63346	11.08138	10.51085	9.88241	10.54796	1.23527	61.76363
3	7.82014	7.80880	7.46311	7.45188	7.75411	7.73440	7.82162	7.76755	7.90346	7.87703	7.74021	1.68337	56.11218
4	6.18947	6.39492	6.67359	6.58843	6.69666	5.94577	6.33806	6.57843	5.92104	6.50092	6.38273	2.04138	51.03461
5	5.43022	4.74273	5.42859	4.74206	4.71822	5.60458	5.38358	5.76186	5.24985	4.98782	5.20495	2.50331	50.06619
6	5.25544	4.23146	4.60251	4.21377	4.31836	5.25953	5.18053	4.80761	4.74648	5.14046	4.77562	2.72836	45.47269
7	4.64489	4.55259	3.88574	4.82770	4.04678	4.33512	4.61613	4.34340	4.86736	3.19357	4.33133	3.00822	42.97462
8	4.95324	3.56896	4.40994	4.53018	4.18928	3.52979	3.58847	4.16734	3.76927	4.32820	4.10347	3.17527	39.69084
9	3.48625	3.76518	3.51753	3.60628	3.52601	4.43481	4.87229	3.54686	4.11137	4.53437	3.94009	3.30693	36.74363
10	4.35131	3.12812	2.94774	2.97111	3.22536	2.87455	2.95785	4.55470	3.06771	3.54556	3.36240	3.87509	38.75089
11	3.66484	3.59395	4.61397	3.93636	2.33290	2.94048	3.74800	4.36126	4.67986	4.62758	3.84992	3.38438	30.76710
12	2.36951	3.16725	2.69951	1.41599	3.12386	4.41262	3.14319	4.03872	4.89477	4.87439	3.41398	3.81654	31.80451
13	4.09518	1.51933	2.57956	2.46376	2.39060	2.49551	4.06758	4.46148	4.86603	4.81093	3.37500	3.86063	29.69714
14	1.53672	1.54405	3.35839	1.55882	1.53593	3.67676	3.87404	3.47258	4.18435	1.79634	2.65380	4.90979	35.06993
15	2.99112	1.60879	2.13819	1.60897	3.01306	1.59192	1.81012	2.00863	1.63003	1.69957	2.01004	6.48226	43.21505
16	1.95548	2.14306	1.67529	1.74542	2.14028	1.68161	2.12134	2.03725	2.14521	1.63959	1.92845	6.75650	42.22815

Fig 6 – Average test data for 150x150 matrix utilising different numbers of threads

The average data shows a decrease in runtime correlating with the increase in number of threads, something that is not entirely apparent from only one run. This trend is more blatant to see when plotted onto a graph like the one in figure 7.

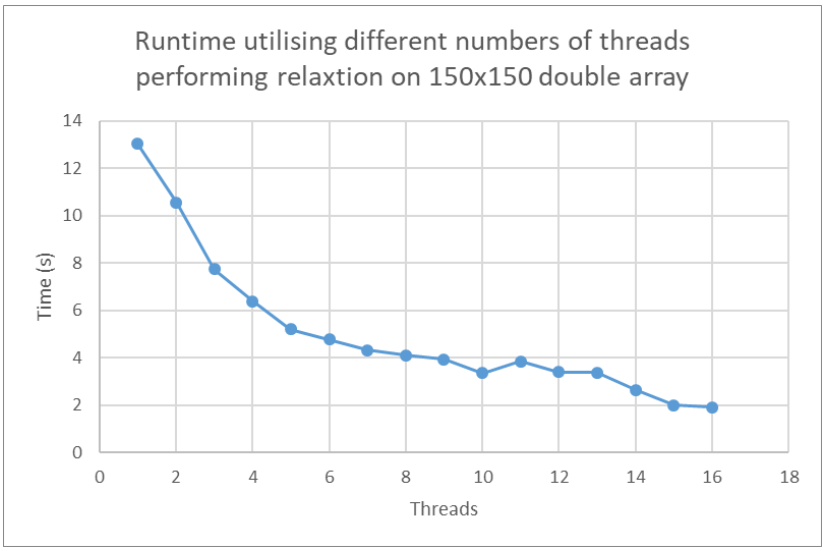

Fig 7 – Plot of average runtimes utilising different numbers of threads

The speedup and efficiency can be calculated using the equations below, the results of which are shown in figure 6 and the plots in figures 8 and 9.

$$S_p = \frac{\text{time on a sequential processor}}{\text{time on } p \text{ parallel processors}}$$

Eq. 1 – Calculation of Speedup

$$E_p = \frac{S_p}{p} = \frac{\text{time on a sequential processor}}{p \times \text{ time on } p \text{ parallel processors}}$$

Eq. 2 – Calculation of Efficiency

Speedup

Analysis of this data proved difficult due to the variation in runtimes when using identical data and identical numbers of threads. The key feature of this graph is the apparent loss in speed when increasing from 10 to 11 threads. However, this can be easily explained referring back to section 1 and the specific implementation of the relaxation algorithm. Figure 1 showed how threads would have to wait if the dimensions of the array were not fully divisible by the number of threads and this is the characteristic that is being displayed in figure 8. The algorithm does not display slowdown when increasing to 11 threads, but rather speedup when increasing to 10 threads. This is also apparent when increasing to 15 threads, the largest speedup in the whole data set.

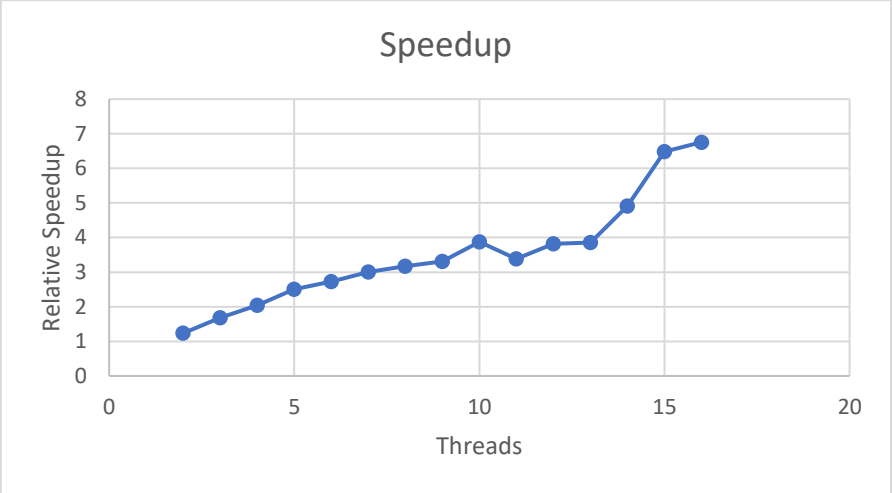

Fig 8 – Plot of relative speedup utilising different numbers of threads

Efficiency

Due to efficiencies direct relationship to speedup, similar 'anomalies' can be seen at 10 and 15 threads. Aside from these, a general decline in efficiency can be seen as the number of threads increases; this is to be expected. The relaxation algorithm includes the use of two barriers, threads have to wait while other threads complete computation; therefore it is safe to assume that more threads results in more time spent idle across all threads. It could be expected that a similar step to that seen at 10 and 15 threads would also occur for 20, 30 and 32 threads, had it been possible to analyse using that many cores. It would also be logical to assume that the increase seen at 14 threads is an anomalous result, unrelated to the implementation of the software.

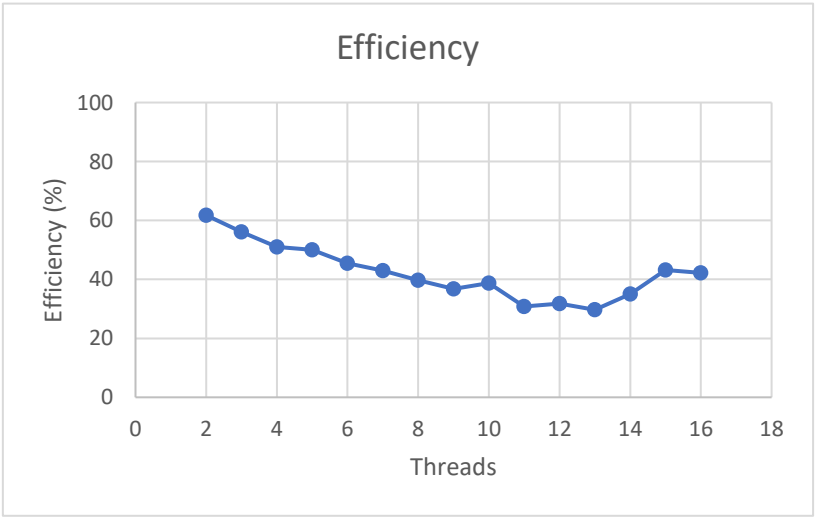

Fig 9 – Plot of relative speedup utilising different numbers of threads

Small Data Utilising Different Numbers of Threads

Subsequent tests were run on a much smaller sample data, this time an 18x18 array. The average of 5 runs is enough to demonstrate that the overhead required by thread generation outweighs the benefit of parallelisation. This data is also absent of the speedup that was apparent at 10 and 15 threads in the previous tests, which leads to confirm the theory from before.

Threads	Repeat Number					Average
	1	2	3	4	5	
1	0.00366	0.00323	0.00315	0.00314	0.00324	0.00164
2	0.00739	0.00671	0.00693	0.00687	0.00675	0.00346
3	0.00987	0.00903	0.00896	0.00900	0.00879	0.00456
4	0.01466	0.01245	0.01632	0.01625	0.01181	0.00715
5	0.01505	0.01496	0.01518	0.01527	0.01475	0.00752
6	0.01950	0.01950	0.01950	0.02034	0.02173	0.01006
7	0.02386	0.02670	0.02538	0.02658	0.02504	0.01276
8	0.03216	0.03093	0.03112	0.03165	0.03142	0.01573
9	0.03430	0.03615	0.03524	0.03701	0.03434	0.01770
10	0.04233	0.04098	0.04069	0.04201	0.04205	0.02081
11	0.04522	0.04510	0.04958	0.04688	0.04515	0.02319
12	0.05024	0.05083	0.05290	0.04966	0.05158	0.02552
13	0.05415	0.05512	0.05511	0.05336	0.05432	0.02721
14	0.05918	0.06081	0.05884	0.05979	0.06169	0.03003
15	0.05782	0.06453	0.06106	0.06049	0.06238	0.03063
16	0.06359	0.06342	0.06289	0.06185	0.06437	0.03161

Fig 10 – Average test data for 18x18 matrix utilising different numbers of threads

The graph plot for this data shows linear slowdown which, while quite the opposite of the other runtime graph, is what should be expected as a result of the large time overheads required by thread creation.

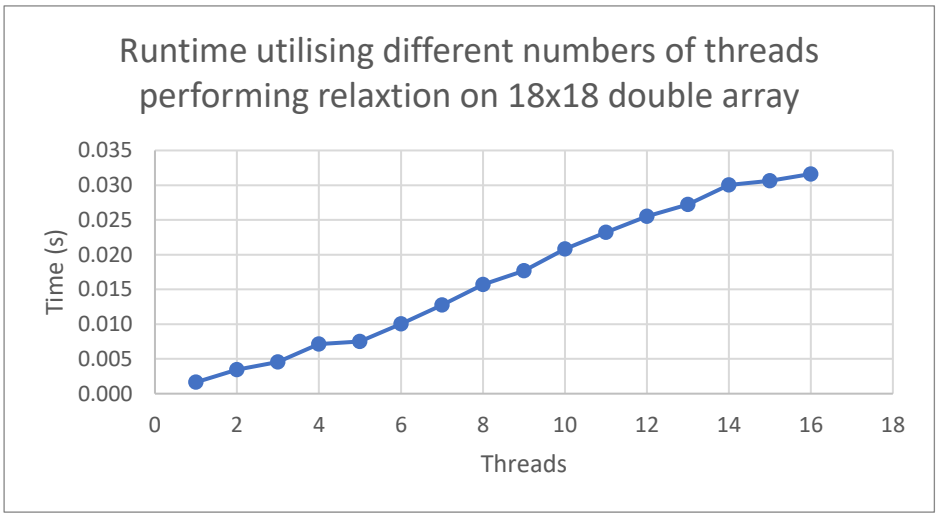

Fig 11 – Plot of average runtimes utilising different numbers of threads

Section 4 – Conclusion

To summarise, the parallel solution realised here is by no means the most efficient, yet it does display most of the characteristics that would be expected of shared memory parallel software. Perhaps a better solution would be for threads to select their working row from a shared access pool of row pointers, although there is no way of knowing the extra overheads that would be required to implement this. It is not uncommon in parallel computing to find the simplest solution is also the best.

Section 5 – References

CM30225 Parallel Computing, 2018. *Lecture Slides* [Online - Private]. people.bath.ac.uk. Available from: http://people.bath.ac.uk/masrjb/CourseNotes/cm30225.html [Accessed 22 November 2018]