UNIVERSITY OF
BATH

CM30225: Parallel Programming

Coursework 1: Shared Memory Programming

MA2451

The University of Bath
Department of Computer Science

12th November 2022

Introduction

This assignment sought to provide experience with the use of primitives for parallelism, namely pthreads, mutexes and semaphores, on a large scale computer cluster with a batch processing system. Parallel programs were tested on the Azure HPC, utilizing a single node with 44 individual processors and a shared memory architecture.

The problem involved creating a script in C to perform differential iterative relaxation on a matrix of doubles until the difference between one iteration and the next settled to a precision value. Matrix boundary values remained fixed. The approach to how parallelism should be implemented was entirely dependent on me.

Approach To Parallelism

My first instinct was to partition the matrix into n sub-matrices for n processors and run each partition on a separate thread. However, my relative lack of experience in C made this infeasible given the time frame of the coursework so I opted for a compromise, making the approach to parallelism easier to implement and conceptualize at the cost of sacrificing performance.

During my research, I came across a Stack Overflow post[1] about the use of mutex locks in a matrix as a method of locking the cells in a matrix of data. Given the information in lectures about the cost of initializing and using locks, I was intrigued by the possible trade-offs between increasing the number of processors and threads versus using so many locks, so I decided to go ahead with this approach.

Starting with a sequential approach to the problem, I created a script that would allocate the memory required for the matrix, fill it with values and solve the relaxation problem row-wise, iterating through the matrix from element [1] [1] to element [matrix size - 1] [matrix size - 1]

Fig.1 (Sequential.c relaxation algorithm extract)

```
int precision_counter = 0;
for (int i = 1; i < matrix_size - 1; i++) {
	for (int j = 1; j < matrix_size - 1; j++) {
		// Start of critical Section Grab the current value at [i][j]
		double current = relax_matrix[i][j];
		// Calculate the average of the 4 adjacent cells
		relax_matrix[i][j] = (relax_matrix[i - 1][j] + relax_matrix[i + 1][j]
			+ relax_matrix[i][j - 1] + relax_matrix[i][j + 1]) / 4;
		// End of critical section Find difference between old and new values
		double delta = (double)fabs(current - relax_matrix[i][j]);
		if (delta > precision) {
			precision_counter++;
		}
	}
}
if (precision_counter == 0) break;
```

The above code exists inside an infinite while loop, so the program continues to iterate through the matrix, calculating the average of the 4 adjacent values around a cell. If the difference between the value of this cell and the new calculated value is less than the precision given, that cell can be considered “solved.” Otherwise, another iteration is needed and the value of ‘precision_counter’ is incremented. The precision counter must be zero at the end of a complete iteration in order for the matrix to be considered solved, implying all of the differences between cells were less than the precision.

I chose this particular approach because it allowed me to keep the ‘precision_counter’ variable outside of the critical section, since any races to increase the value will be irrelevant as having any value other than 0 will cause another iteration to happen regardless. The value at each cell in the matrix would need to be locked to prevent two or more threads from trying to calculate the average and modify it at the same time.

Using the same memory allocation and cell assignment approach as with the main data matrix, I created a separate set of functions to generate a matrix of mutex locks of size (relax matrix - 2). This was done to slightly ease the computational burden of creating so many locks, especially for matrix sizes greater than 1000x1000, by avoiding the creation of locks for cells whose value will never change. The number of mutex creations prevented by this diminishes as the matrix size increases, but can certainly be valuable for smaller matrices where the size is less than 10 as edge cells make up >36% of the matrix.

		1.000	1.000	1.000	1.000	1.000	1.000	1.000
Data Matrix		1.000	0.000	0.000	0.000	0.000	0.000	1.000
		1.000	0.000	0.000	0.000	0.000	0.000	1.000
Fixed Value		1.000	0.000	0.000	0.000	0.000	0.000	1.000
Variable Value		1.000	0.000	0.000	0.000	0.000	0.000	1.000
		1.000	0.000	0.000	0.000	0.000	0.000	1.000
		1.000	1.000	1.000	1.000	1.000	1.000	1.000
		Lock 1	Lock 2	Lock 3	Lock 4	Lock 5		
Mutex Matrix		Lock 6	Lock 7	Lock 8	Lock 9	Lock 10		
		Lock 11	Lock 12	Lock 13	Lock 14	Lock 15		
Mutex Lock		Lock 16	Lock 17	Lock 18	Lock 19	Lock 20		
		Lock 21	Lock 22	Lock 23	Lock 24	Lock 25		

This choice creates a discontinuity when iterating through the main data matrix, as the lock for cell [i] [j] in the data matrix exists at [i - 1] [j - 1] in the mutex matrix. This had to be accounted for when locking and unlocking.

When applying the locking mechanisms around the critical section of code, I opted for *pthread_mutex_trylock* which means that a thread will attempt to lock a mutex but will not wait around for the mutex to unlock and instead move on to an unlocked cell, thus *in theory* increasing the rate at which cell averages are calculated.

Once the critical section is complete, the difference between the new and old values are calculated in the same manner as the sequential program.

```
int precision_counter = 0;
for (int i = 1; i < matrix_size - 1; i++) {
	for (int j = 1; j < matrix_size - 1; j++) {
		// Mutex Trylock: Attempt to lock the mutex, return 0 if
		// successful and -1 if unsuccessful
		if (pthread_mutex_trylock(&mutex_matrix[i - 1][j - 1]) != 0) {
			continue;
		}

		// Start of critical Section
		double current = relax_matrix[i][j];
		// Calculate the average of the 4 adjacent cells
		relax_matrix[i][j] = (relax_matrix[i - 1][j] + relax_matrix[i + 1][j]
			+ relax_matrix[i][j - 1] + relax_matrix[i][j + 1]) / 4;
		// End of critical section

		// Unlock the mutex for the cell this thread has worked on
		pthread_mutex_unlock(&mutex_matrix[i - 1][j - 1]);

		double delta = (double)fabs(current - relax_matrix[i][j]);
		if (delta > precision) {
			precision_counter++;
		}
	}
}
if (precision_counter == 0) break;
```

Fig. 2 (Shared_memory.c relaxation algorithm extract)

Locking and unlocking is expensive, and therefore the obvious outcome of this approach will be a significant slow-down for smaller matrices, and perhaps larger ones when using a lower thread count, but ideally the benefit of having a large number of threads and processors should outweigh this overhead for large matrices and outpace the sequential implementation.

Creating Threads

Given that n threads can be used in this problem, it was important to avoid race conditions when providing threads with identifiers. As discussed in lectures for this course, using a single variable and modifying it between thread creations can cause issues. I instead opted for a struct of n integers that could be individually assigned and act as a catalog of threads, each being given to a thread during a loop iteration. The identifiers were equivalent to i in the range 0 to N.

Threads were given a NULL attribute and no modification to the threads was necessary during the running of the ‘relaxation’ function, as it is effectively sequential.

```
// Create a structure to catalogue thread IDs and prevent races
struct thread_ids {int thread_identity};
// Initialise the threads we need in an array
pthread_t threads[thread_count];
// Create the thread ID catalogue, size n for n threads
struct thread_ids identifiers[thread_count];

for (int i = 0; i < thread_count; i++) {
      // Assign each thread a unique identity based on i
      identifiers[i].thread_identity = i;
      // Create a NULL attribute thread i, with ID i, and run relaxation
      pthread_create(&threads[i], NULL, relaxation, &identifiers[i]);
}
```

Fig. 3 (Shared_memory.c creating threads & identifiers)

Using Azure

I accessed the Azure cluster by first SSH-ing into the University's Linux service before the HPC itself. Creating a shell script to run my compiled code, I ensured all recommended warnings (-Wall, -Wextra, -Wconversion) were active for the gcc compiler. The SLURM job submission script that ran this shell was set to allocate one node and a variety of CPU cores depending on a given test case. I created N threads in the C code for N CPU cores that I assigned in this file, ideally meaning 1 thread per processor.

Rather than automating the process, out of mild paranoia I decided to run each test case individually and manually adjust the core count / matrix size every time, ensuring that I could quickly address any problems that could come up and better prepare myself for subsequent use of the cluster after the completion of this coursework. My chosen test-case matrix sizes were *10x10*, *25x25*, *50x50*, *100x100*, *250x250*, *500x500*, *1000x1000*, *2500x2500* and *5000x5000* to spread the problem size diversely enough that a trend may appear and conclusions could be made. I tested across the entire span of 44 cores in increments of 4, thus testing 11 configurations (not including the sequential implementation) with 9 test cases each.

Correctness Testing

To keep the timing consistent and allow accurate comparisons, the matrix that relaxation was performed on was made up entirely of 0's, with the edge cells containing 1's as per the coursework specification example. Using random values could cause the number of relax iterations for a given matrix size and precision to vary.

I first tested my sequential implementation so that I could use it as a baseline. I used a 10x10 matrix with a precision of 0.01 as my test case. By creating a copy of the sequential program in Python (a language I am much more familiar with), and performing the calculation by hand over the course of several hours, I could conclude that my C implementation was accurate.

// Sequential Result in C

1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000
1.000000	0.988164	0.979039	0.973383	0.971474	0.973125	0.977746	0.984456	0.992222	1.000000
1.000000	0.979039	0.962895	0.952900	0.949541	0.952476	0.960658	0.972526	0.986254	1.000000
1.000000	0.973383	0.952900	0.940236	0.935997	0.939741	0.950129	0.965182	0.982582	1.000000
1.000000	0.971474	0.949541	0.935997	0.931482	0.935510	0.946642	0.962755	0.981371	1.000000
1.000000	0.973125	0.952476	0.939741	0.935510	0.939319	0.949805	0.964969	0.982481	1.000000
1.000000	0.977746	0.960658	0.950129	0.946642	0.949805	0.958488	0.971033	0.985515	1.000000
1.000000	0.984456	0.972526	0.965182	0.962755	0.964969	0.971033	0.979789	0.989894	1.000000
1.000000	0.992222	0.986254	0.982582	0.981371	0.982481	0.985515	0.989894	0.994947	1.000000
1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000

My sequential result was consistent with Python and my own handwritten values - in both settled values and iteration counts. I thus considered my sequential C program to be an acceptable benchmark for the parallel program.

An interesting pattern emerged when testing the parallel C program, however.

The **number of iterations remained the same**, but the **exact values in each cell were slightly different**

// Parallel result with 4 cores / 4 threads

1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000
1.000000	0.989113	0.980897	0.975874	0.974200	0.975696	0.979891	0.985937	0.992976	1.000000
1.000000	0.981454	0.967314	0.958542	0.955444	0.958029	0.965096	0.975373	0.987581	1.000000
1.000000	0.977240	0.959712	0.948567	0.944764	0.947725	0.956160	0.969009	0.984429	1.000000
1.000000	0.976259	0.957807	0.946179	0.942036	0.944931	0.953846	0.967440	0.983650	1.000000
1.000000	0.978180	0.961170	0.950452	0.946629	0.949302	0.957575	0.970099	0.984991	1.000000
1.000000	0.982334	0.968613	0.959968	0.956875	0.959060	0.965772	0.975886	0.987877	1.000000
1.000000	0.987931	0.978534	0.972603	0.970469	0.971965	0.976564	0.983480	0.991681	1.000000
1.000000	0.994026	0.989363	0.986417	0.985354	0.986095	0.988375	0.991806	0.995872	1.000000
1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000

The severity of the differences between values increases with a higher number of cores. If we extract one row of the completed matrix for visibility, the differences are more clear:

// Sequential (Baseline)

0.988164	0.979039	0.973383	0.971474	0.973125	0.977746	0.984456	0.992222

// 4 threads

0.989113	0.980897	0.975874	0.974200	0.975696	0.979891	0.985937	0.992976

// 20 threads

0.996744	0.994026	0.992780	0.991536	0.992171	0.993534	0.995451	0.997724

The differences between values is practically negligible for configurations at no more than 0.004 with less than 5 threads which is much less than the precision of 0.01, but slowly increases to around 0.015 at around 20 threads. Curiously, the number of iterations remains the same for the latter example, but this may have been luck.

Possible Solutions

I believe this problem is the result of a severe oversight on my part when coming up with my approach. When a cell is locked and modified as part of the averaging calculation, that middle cell is safe from race conditions, however the 4 adjacent cells used for the calculation are not. This can, and likely does, result in a race between a thread using the adjacent value for its calculation and a thread actually modifying this adjacent cell.

As well as this, it is possible that a cell could be locked, then unlocked, quickly enough that another thread could come along and perform a second averaging calculation in one iteration of the relaxation. This would not be such a severe issue as values begin to settle, but certainly during the first few iterations where neighboring values could be very different this could cause major reliability issues. This would be more likely to occur with high thread counts, and may be another reason why my results differ by so much with 20+ threads.

1.000	1.000	1.000	1.000	1.000	1.000	1.000
1.000	0.500	0.000	0.000	0.000	0.000	1.000
1.000	0.000	0.000	0.000	0.000	0.000	1.000
1.000	0.000	0.000	0.000	0.000	0.000	1.000
1.000	0.000	0.000	0.000	0.000	0.000	1.000
1.000	0.000	0.000	0.000	0.000	0.000	1.000
1.000	1.000	1.000	1.000	1.000	1.000	1.000

While I lack the time to do so and properly test the theory, I believe this could be fixed by locking the adjacent cells as well as the middle cell, while preventing the central cell from being locked in the first place until all of the adjacent cells are available to be locked as shown above (constant cells are shown being locked only for the sake of illustration). This would cause the runtime to suffer, needing 3-5x as many locks and unlocks, but could provide more stable and reliable results.

Regardless, given the fact that the number of relaxing iterations was constant for all core configurations and matrix sizes tested, I decided to press on with the scalability investigation.

Scalability

Since Linux and POSIX were available, it made sense to use clock_gettime()'s "MONOTONIC" clock to ensure any changes to the system time, despite being unlikely, were not a problem. The code used was derived from a GeeksforGeeks article[2]:

```
Struct timespec start, end;
clock_gettime(CLOCK_MONOTONIC, &start);
// Thread creation & relaxation
clock_gettime(CLOCK_MONOTONIC, &end);
double time_taken;
time_taken = (end.tv_sec - start.tv_sec) * 1e9;
time_taken = (time_taken + (end.tv_nsec - start.tv_nsec)) * 1e-9;
printf("%lf", time_taken);
```

Fig. 4 (Code used for timing the relaxation process in shared_memory.c)

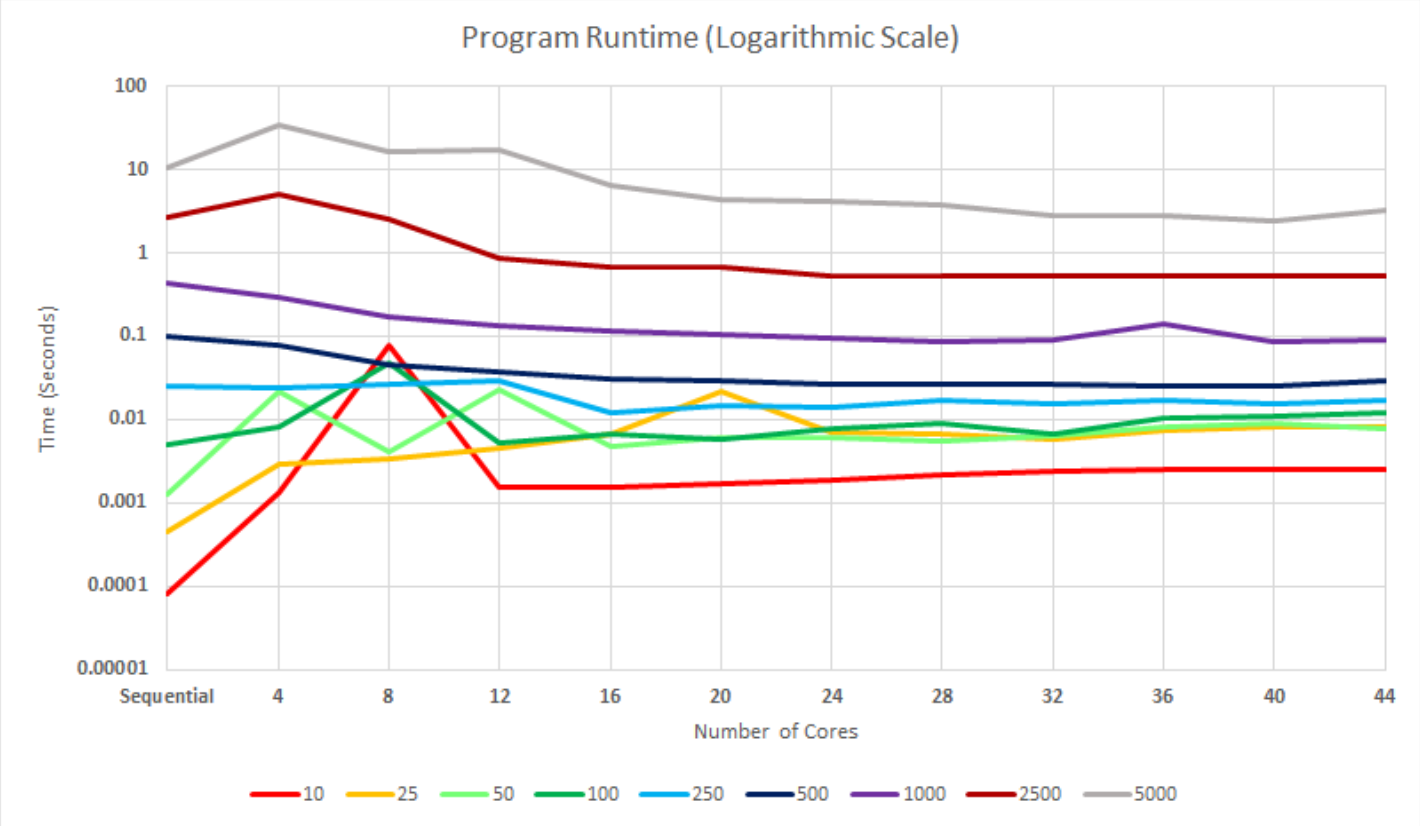


Fig. 5 (Execution time for various core configurations and matrix sizes, note the logarithmic time scale)

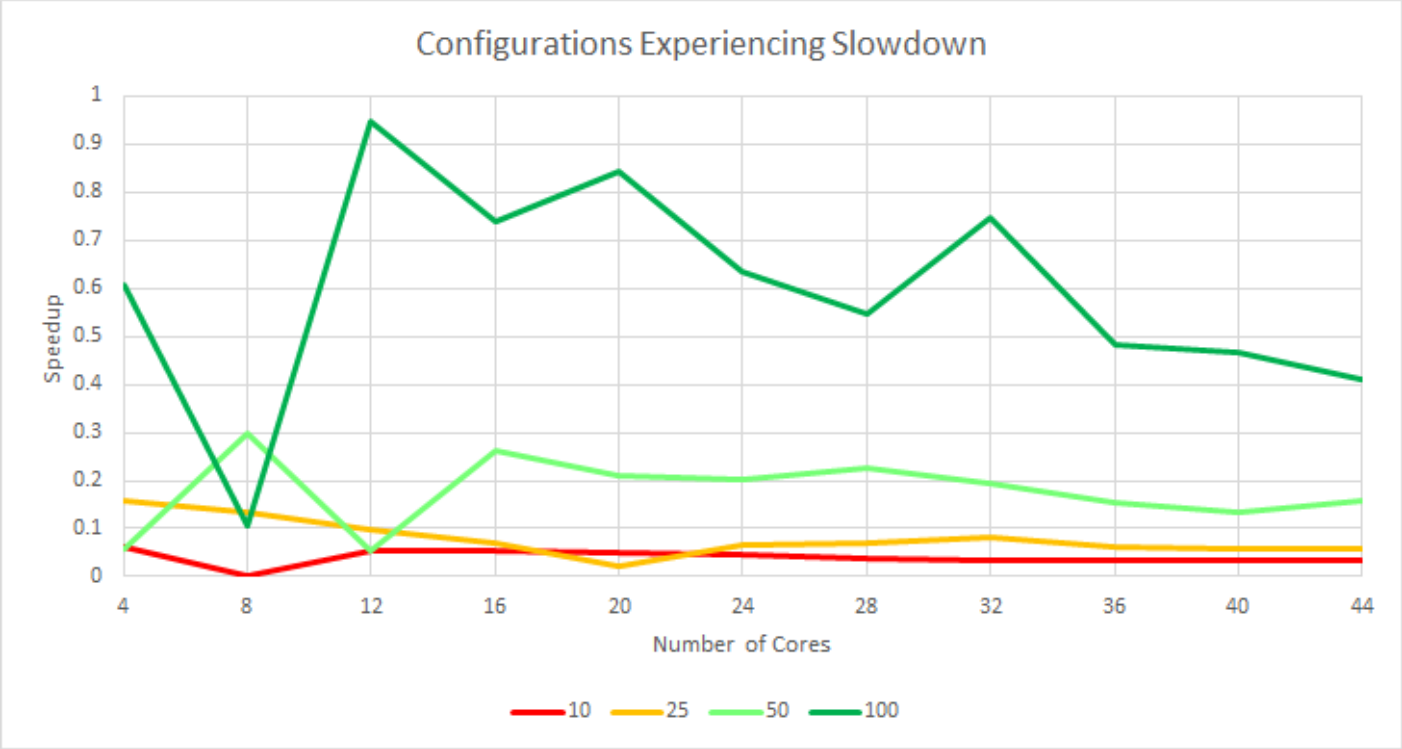
Raw data for all graphs in the scalability section can be found in the appendix.

Each core/thread configuration was tested 3 times, the timings were then averaged.
Smaller matrix sizes up to 250x250 had a tendency to slow down with a greater number of cores and were generally slower than sequential.c. The most extreme cases exhibit odd and trend-breaking behavior such as 10x10, which appears to slow down by almost 3 orders of magnitude at around 8 cores when compared to the sequential equivalent program before dropping back down by an order of magnitude. There is nothing in the Azure HPC documentation[3] that suggests there should be odd activity here, so there may be some interactions at this matrix size for 8 cores that I am presently unaware of.

Thankfully, the execution time sees significant improvement for larger matrix sizes, though all configurations appeared to plateau at between 24-32 cores. It is unlikely that the program was closing in on Amdahl's limit, and with further testing of extreme matrix sizes (10,000+) a down/uptrend may continue to occur, however it is more likely that the cost of lock/unlock time versus the benefits of adding new cores becomes so minimal that simply throwing more processors at the problem is effectively meaningless.

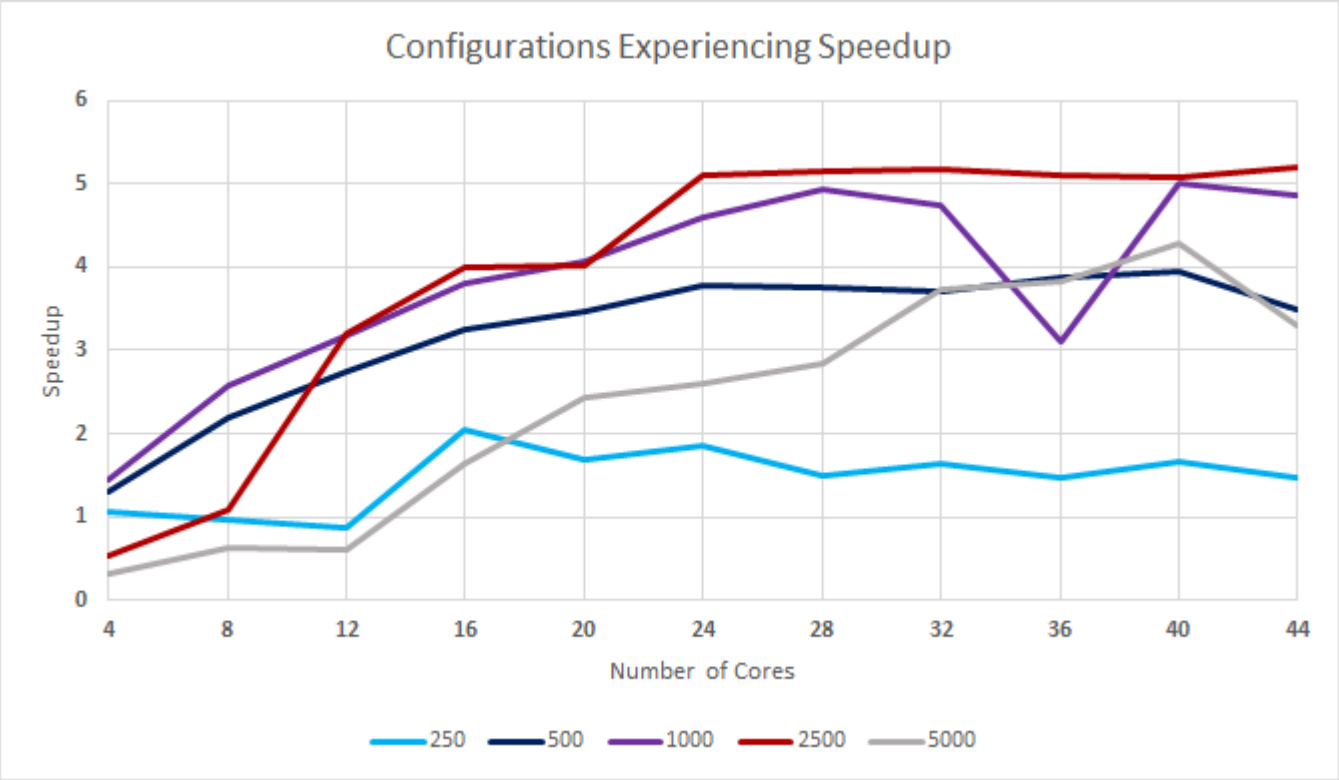
If we consider the data for each "number of cores" column individually, the time taken for the sequential implementation increases logarithmically. This is not strictly true for the parallel version. For instance the runtimes for 25, 50, 100 and 250 are very close together, especially for a larger number of processors.

The sequential code and parallel code I created are practically identical algorithmically, therefore I deemed it appropriate to use the average execution time for this sequential program to calculate the speedup, making use of the speedup equation, S_p = *Sequential Speed / Parallel Speed*



The only configurations to experience slowdown across the board were the smaller matrices of size 100x100 and below. I find it surprising that the slowdown was so dramatic, as I made specific use of the trylock function to ensure the threads would not be waiting around for locks, though this could be because the frequency at which a thread tries to access a value that is currently locked is far higher than would be the case with large matrices, leading to a lot of instances where a thread is doing very little.

The plots for the 10x10 and 100x100 matrix is the only configuration that resembles the typical speedup curve described in CM30225 lecture 6[4], whereas all other configurations either trend upward or downward, these examples slow down dramatically at low processor counts, spike back upwards then continue a somewhat linear down/uptrend.

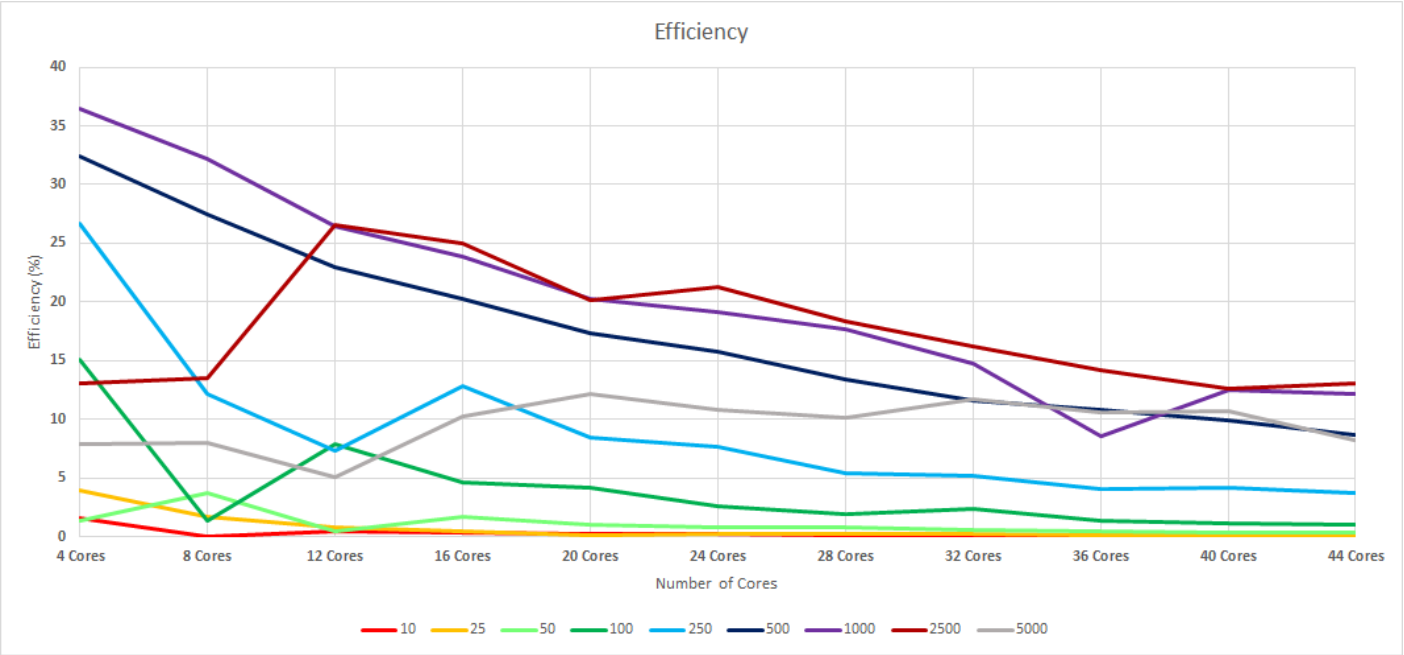


Configurations with matrix sizes greater than 100 experienced positive speedup, in that the parallel program was faster than the sequential program running with the same matrix size. This is to be expected, as the threads can quickly find new cells that are unlocked.

None of the speedup is superlinear, i.e. greater than the number of cores. This is at least a good indicator that the values obtained are legitimate, and can be attributed to the use of fixed starting matrix values to prevent the parallel program from completing the relaxation too quickly by random chance.

An intriguing loss in speedup occurs with the 5000x5000 matrix, which breaks the trend of speedup increasing with matrix size, but unlike the other configurations whose speedup appears to plateau at the aforementioned 24-32 core range, this configuration continuously increases up until the maximum number of available cores. If we look at the raw data, the time difference between the 2500 and 5000 setups for the sequential program is roughly 4x. For the parallel program this difference ranges from 6x for 44 cores to 20x for 12 cores. It is possible, then, that my previous suggestion that this parallelism approach would scale well with larger problems may be untrue.

The speedup can be a misleading indicator of the program's practical effectiveness, so to determine how well it performs while considering the cost of using more cores, we can calculate the efficiency for the results with the equation from lecture 7[5]: E_p = *Speedup value / Number of processors used*



The larger the matrix, the more efficient the use of parallel cores - with diminishing returns as more cores are added. As with the speedup graph, this does not apply to the 5000x5000 matrix for lower core counts so it would be interesting to see how extreme matrix sizes would fit into this efficiency plot. The plateau in execution time for higher core counts naturally contributes to a gradual decrease in efficiency from 20 cores onwards.

Putting these numbers into perspective, in the absolute best case of a 1000x1000 matrix running with 4 cores, we are putting 36.5% of the available processing power to use. The average efficiency for matrix sizes greater than 250x250 is 16.2%.

Conclusion

The compromise in execution time and efficiency that resulted from my decision to use a conceptually easier understood approach to parallelism leads me to conclude that this is not an ideal way of handling the problem. The cost of using so many mutex locks is such a substantial overhead that it completely undercuts the benefits of more processors.

I believe that an implementation that segments the data matrix into chunks of roughly equal size would be a much more effective solution and yield much greater speedup and efficiency results, especially for smaller matrix sizes. Mutexes should be used in moderation.

That being said, my results prove Gustafson's optimism correct in that increasing the problem size for a fixed number of cores will improve the efficiency, though this implementation's improvement is not linear. Therefore, we can conclude that increasing the problem size rather than throwing more computation at a fixed problem is the key to effective parallel programming.

UNIVERSITY OF BATH

Appendix

Results of execution time tests (seconds) - Number of processors against matrix size

	10	25	50	100	250	500	1000	2500	5000
Sequential	0.000083	0.000456	0.001237	0.004966	0.025371	0.100158	0.431973	2.70714	10.514288
4 Cores	0.00132	0.002881	0.021779	0.00821	0.023807	0.077162	0.296063	5.158367	33.394379
8 Cores	0.07721	0.003382	0.004132	0.047055	0.026107	0.045501	0.167493	2.50053	16.503455
12 Cores	0.00153	0.004594	0.022712	0.005238	0.028886	0.036422	0.136314	0.84835	17.396997
16 Cores	0.001526	0.006627	0.004691	0.006725	0.012325	0.030918	0.11326	0.678428	6.384726
20 Cores	0.00166	0.022156	0.005934	0.00589	0.015044	0.028939	0.10648	0.672165	4.308334
24 Cores	0.001879	0.006972	0.006147	0.00781	0.013724	0.026549	0.093921	0.53066	4.058553
28 Cores	0.002134	0.006654	0.005453	0.009072	0.016895	0.026682	0.087487	0.526055	3.696101
32 Cores	0.00238	0.005699	0.006413	0.006664	0.015362	0.027019	0.091304	0.523288	2.808776
36 Cores	0.002459	0.0074	0.008115	0.010313	0.017377	0.025782	0.13923	0.531373	2.751352
40 Cores	0.002508	0.008144	0.009103	0.010691	0.015172	0.025329	0.086486	0.534156	2.456452
44 Cores	0.002532	0.008052	0.007737	0.01209	0.017181	0.028684	0.088935	0.519641	3.189134

Speedup calculations = Sequential time / Parallel time

	10	25	50	100	250	500	1000	2500	5000
4 Cores	0.0628787	0.1582783	0.0567978	0.6048721	1.0656949	1.2980223	1.4590577	0.5248056	0.3148520
8 Cores	0.0010749	0.1348314	0.2993707	0.1055360	0.9718083	2.2012263	2.5790510	1.0826264	0.6370961
12 Cores	0.0542483	0.0992599	0.0544646	0.9480717	0.8783147	2.7499313	3.1689555	3.1910650	0.6043737
16 Cores	0.0543905	0.0688094	0.2636964	0.7384386	2.0584989	3.2394721	3.8139943	3.9903129	1.6467876
20 Cores	0.05	0.0205813	0.2084597	0.8431239	1.6864530	3.4610041	4.0568463	4.0274932	2.4404533
24 Cores	0.0441724	0.0654044	0.2012363	0.6358514	1.8486592	3.7725714	4.5993228	5.1014585	2.5906494
28 Cores	0.0388940	0.0685302	0.2268476	0.5473985	1.5016868	3.7537665	4.9375678	5.1461159	2.8446971
32 Cores	0.0348739	0.0800140	0.1928894	0.7451980	1.6515427	3.7069469	4.7311508	5.1733271	3.7433700
36 Cores	0.0337535	0.0616216	0.1524337	0.4815281	1.4600333	3.8848033	3.1025856	5.0946133	3.8214986
40 Cores	0.0330940	0.0559921	0.1358892	0.4645028	1.6722251	3.9542816	4.9947159	5.0680700	4.2802741
44 Cores	0.0327804	0.0566318	0.1598810	0.4107526	1.4766893	3.4917724	4.8571765	5.2096351	3.2969100

Efficiency calculations = Sequential time / P processors * Parallel time

	10	25	50	100	250	500	1000	2500	5000
4 Cores	1.5719696	3.9569593	1.4199458	15.121802	26.642374	32.450558	36.476442	13.120140	7.8713007
8 Cores	0.0134373	1.6853932	3.7421345	1.3192009	12.147604	27.515329	32.238138	13.532831	7.9637021
12 Cores	0.4520697	0.8271658	0.4538716	7.9005981	7.3192896	22.916094	26.407962	26.592208	5.0364477
16 Cores	0.3399410	0.4300588	1.6481027	4.6152416	12.865618	20.246700	23.837464	24.939455	10.292422
20 Cores	0.25	0.1029066	1.0422986	4.2156196	8.4322653	17.305020	20.284231	20.137466	12.202266
24 Cores	0.1840518	0.2725186	0.8384848	2.6493811	7.7027470	15.719047	19.163845	21.256077	10.794372
28 Cores	0.1389074	0.2447507	0.8101700	1.9549949	5.3631674	13.406309	17.634170	18.378985	10.159632
32 Cores	0.1089810	0.2500438	0.6027795	2.3287439	5.1610711	11.584209	14.784846	16.166647	11.698031
36 Cores	0.0937598	0.1711711	0.4234271	1.3375782	4.0556482	10.791120	8.6182934	14.151703	10.615274
40 Cores	0.0827352	0.1399803	0.3397231	1.1612571	4.1805628	9.8857041	12.486789	12.670175	10.700685
44 Cores	0.0819510	0.1415797	0.3997027	1.0268817	3.6917234	8.7294310	12.142941	13.024087	8.2422751

References

[1] "att_guy," Stack Overflow, 2018, *How to lock a matrix cell in multithreading process* [Online] Available from: https://stackoverflow.com/questions/49966025/how-to-lock-a-matrix-cell-in-multithreading-process [Accessed 04/11/22]

[2] Kumar V., GeeksforGeeks, 2019, *Measure execution time with high precision in C/C++* [Online] Available from: https://www.geeksforgeeks.org/measure-execution-time-with-high-precision-in-c-c/ [Accessed 09/11/22]

[3] Confluence, 2020, *Research Computing Teaching Cluster Overview* [Online] Available from: https://wiki.bath.ac.uk/display/CloudHPC/Research+Computing+Teaching+Cluster+Home [Accessed 29/10/22]

[4] Bradford R., 2022, *CM30225 Lecture Notes 06* [Online] Available from: https://people.bath.ac.uk/masrjb/CourseNotes/Notes.bpo/CM30225/slides06.pdf [Accessed 27/10/22]

[5] Bradford R., 2022, *CM30225 Lecture Notes 07* [Online] Available from: https://people.bath.ac.uk/masrjb/CourseNotes/Notes.bpo/CM30225/slides07.pdf [Accessed 28/10/22]