HPC Main Assignment

Angelo Yang (YNGANG003)

Julyan van der Westh (VSWJUL003)

Introduction

This assignment focuses on the parallelizing of an Abelian Sandpile - a well-known problem in High Performance Computing (HPC). This is to be achieved through OpenMP and MPI. This assignment has 2 primary aims: Firstly to achieve speedup over a serial version by implementing a parallelized version of the Abelian Sandpile, and secondly, to evaluate and report on the performance of such.

This report will discuss OpenMP, MPI and serial implementations of the Abelian Sandpile problem. In addition, validation methods, benchmark implementations, results in the form of speedup graphs and conclusions will be discussed.

The first parallelization methodology used in the assignment is OpenMP, a shared-memory programming model that enables the parallel execution of code by distributing work across multiple threads running on the same node. The second method, MPI, employs a distributed memory model, using multiple processes that communicate and synchronize through explicit message passing across different nodes.

Methods

Serial Implementation

The serial implementation uses a simple double for-loop approach where execution means iterating through the problem cell by cell - left to right, top to bottom. This has the cost of visiting stable cells unnecessarily, but has the advantage of being highly predictable and optimisable by the C compiler (Especially with the inclusion of the compilation flag: `-O2`). The code was optimised further by ensuring ‘good coding etiquette’ (not declaring variables in a loop, no unnecessary nested function calls, etc).

OpenMP Implementation

The OpenMP implementation of the Abelian Sandpile involved 2 separate phases, detection and toppling. The detection phase utilized multi-threading to identify all unstable cells without updating the grid itself. This ensured that simultaneous read/write conflicts or data races were avoided in the simulation. The second phase, toppling involved utilizing a single thread to serially apply all toppling updates to the grid. Although the serialization of this phase limited the scalability of the program itself, correctness was guaranteed as data races were avoided by ensuring that only 1 thread updates the grid.

Detection Phase:
The nested scan over the entire grid is parallelized using *omp for collapse(2)*, this compiler directive flattens the two loops into a single work-share of size rows x columns. Each thread executes a subset of the iterations, reading its assigned cells and, whenever it finds a cell value > 3, it appends its coordinates to a thread-private list. Additionally due to the fact that each thread only writes to its own list, there are no concurrent writes to shared memory and thus no risk of data races at this stage.

Once the parallel scan completes, a short *omp critical* section merges each thread's private list into a global array of unstable‑cell coordinates, and sets a shared unstable flag if any thread found work. OpenMP reduction was not used here because reductions only work on simple scalar values, for example sums or minimums. Merging each thread's list of cell coordinates and updating a shared index has to happen inside a critical section so that only one thread at a time appends its data.

Toppling Phase:
After gathering all unstable cell coordinates, a single thread iterates over this list and applies the toppling updates. Due to the fact that only 1 thread performs these updates/writes, data races are avoided, furthermore, atomics with greater synchronization overheads are also avoided.

This two phase implementation maximizes the parallelizable part (the nested scan of the entire grid) while safely serializing only the comparatively small set of actual toppling updates prioritizing the correctness of the program.

MPI Implementation

The MPI version was implemented using an asynchronous, halo-exchanging approach with spatial decomposition as a guiding parallelisation method. In broad terms, a problem area nxm is decomposed into *chunks (C)* by dividing the problem area according to the number of processors (p) available. This is optimised and handled with the use of an `MPI Topology` (see Figure 1).

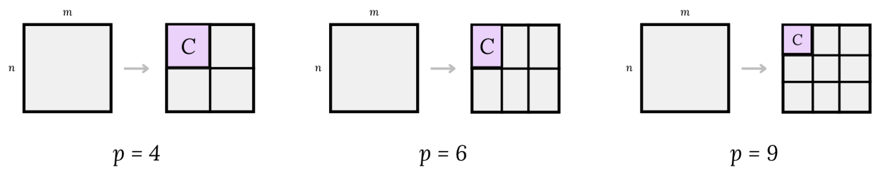


Figure 1: problem chunking for p, which represents processor count

Chunks are sent to processors where local computation can begin. During local computation, a chunk iteratively topples cells. When a topple occurs on the border of a chunk, residue topple that falls outside of the chunk is captured in a *halo* (see Figure 2). Each chunk has 4 halos for each corresponding side (Figure 3).

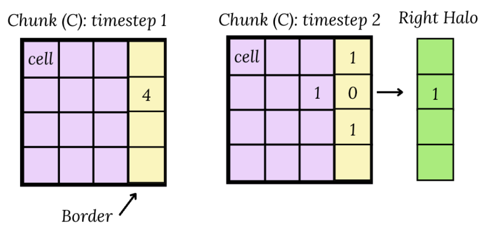


Figure 2: Chunk border toppling into Halo

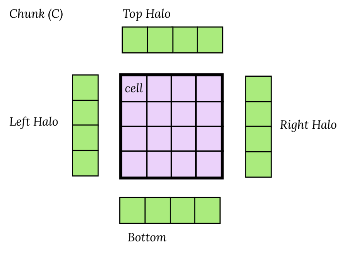


Figure 3: Chunk with 4 Halos

After an iteration of local computation, *halos* are sent to neighbouring chunks. When a chunk receives *halos*, the values stored inside the halos are applied to the chunk-in-question's border cells (see Figure 4). After this process completes, local computation can once again resume.

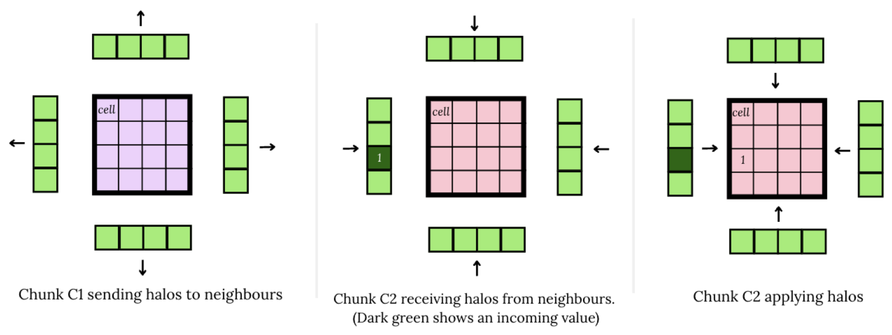

cell
Chunk C1 sending halos to neighbours
1
Chunk C2 receiving halos from neighbours. (Dark green shows an incoming value)
Chunk C2 applying halos

Figure 4: Chunks sending and receiving halos

This process occurs *asynchronously*. This means that chunk C_1 can perform local computation, send halos, receive and apply new halos and carry on with local computation again while ignoring where neighbouring chunks are in their *local* execution. Global convergence occurs when there are no *in-flight* messages (all sent messages have been received) and all chunks are locally stable.

At this point, each local chunk is gathered and stitched together again to present the final output of the simulation.

Validation

For the initial validation of our parallel implementations we informally, visually compare the .ppm output files with one another (taking the serial implementation as the source of truth). This, however, could be inaccurate and lead to false conclusions.

To *formally* validate our parallel implementations, we created a program to compare the .ppm output files generated by each version against that of the serial implementation. Using a custom Python script, we parsed and compared pixel values in the .pmm files to ensure that the final stable sandpile configurations were identical, confirming *correctness* across all implementations.

Benchmark Implementation

For the serial solution, the entire execution time is timed and recorded. This excludes other overhead which is non-essential to evaluating the execution time in isolation.

For the MPI solution, we record the execution time across all simulation-critical aspects. This includes chunking the problem area into chunks, executing the algorithm and stitching local chunks back together. We used the `MPI_Wtime()` method for this.

The benchmarking of the OpenMP solution involved utilizing `omp_get_wtime()` to obtain the start time and end time. Only the execution/simulation time is recorded. Table 1 and 2 below detail the parameters for the strong and weak scaling benchmarking.

For each solution, 3 sets of tests were taken and the average of the 3 are recorded within this report, ensuring correct measurements. Refer to the appendices below for elaboration on test configuration and results.

Problems and Difficulties

To implement the MPI solution, a major difficulty that arose was obeying the Abelian Sandpile automata rules when determining global convergence and simulation stability. This means that *race conditions* due to *asynchronous* message passing and receiving arose. These issues had to be addressed to ensure a *correct* output. Another difficulty was managing resources such that the C code was stylistically up to par (ie: no declaring variables inside loops, memory deallocation, etc). Finally, *asynchronous* execution routines within a chunk meant that *latency hiding* could also be achieved (a processor could post a non-blocking send and continue with its execution). This also caused race conditions which had to be solved.

In implementing the OpenMP solution, the biggest challenge was in balancing thread synchronization overhead with achieving enough parallel work per iteration to obtain real speedup. This essentially meant minimizing the use of critical sections and atomics. Additionally, we also had to pre-allocate and reuse our thread-local buffers to prevent malloc/free overhead.

Results

OpenMP

The results for the OpenMP solution can be found in Appendix B - E.

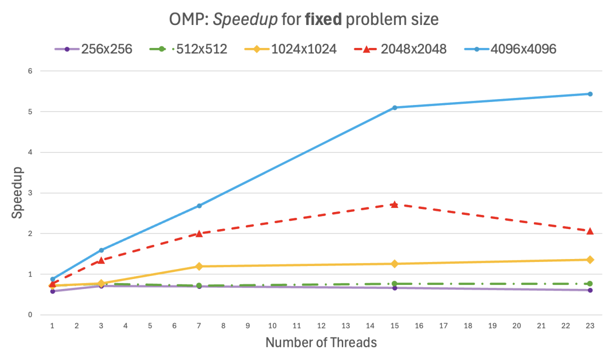


Figure 1: OMP Strong Scaling Speedup Graph

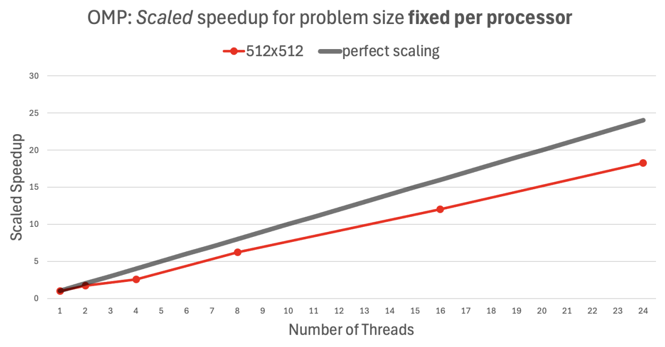


Figure 2: OMP Weak Scaling SpeedupGraph

For our strong-scaling benchmark we measured the wall-clock time of the OpenMP “two-phase” sandpile on grid sizes from 256x256 up to 4096x4096, varying thread counts from 2 to 24. Speedup was computed as: S(p) = T(1)/T(p) using the serial (1-thread) runtimes for each grid size.

From Figure 1, we can see that for small problem sizes, 256x256 and 512x512, the OpenMP solution performed poorly, with maximum speedups at 0.71x and 0.76x. This shows that for these small problem sizes, the fixed overheads - thread startup, barrier synchronization, and the critical-section merge - dominate the per-thread work and actually make the parallel version slower than serial. At moderate problem sizes, 1024x1024, we can start to see the benefits of OpenMP parallelization with a positive speedup with 8 threads. However, only a 1.36x speedup is achieved with 24 threads as the serial toppling and synchronization overhead still limits the efficiency and speedup. For large problem sizes, we can observe the best speedup achieved by the OpenMP solution with 2048x2048 peaking at 2.72× on 16 threads and 4096x4096 reaching a maximum of 5.44× at 24 threads. These larger problems provide enough parallel work to hide much of the barrier and merge overhead, giving clear strong-scaling behavior up to a machine-specific sweet spot (16 threads for 2048x2048, 24 threads for 4096x4096). However, an anomaly can be observed where the speedup dramatically drops for the problem size 2048x2048 with 24 threads. Usually, it is expected that speedup will increase to a certain point and then plateau. This anomaly can be attributed to barrier overheads and memory (DRAM) contention once the number of threads exceeds the machine's sweet spot for that problem size. Finally, the ideal conditions for the OpenMP solution to perform well is with large problem sizes and thread counts, thus within our benchmark the ideal condition is: 4096x4096 with 24 threads, outside of this benchmark, it can be expected that the program would perform well with even larger problem sizes and increasing number of threads up to the HPC system's sweet spot.

The ideal and expected speedup is S = p and the best speedup achieved by our solution is S(24) = 5.44, which is only 23% of the ideal speedup for 24 cores. This shortcoming can be explained by 3 limitations of our OpenMP solution. Firstly, Amdahl's Law limits the program in that each iteration ends with a serial toppling pass and a critical-section merge of per-thread lists and buffers. Furthermore, even if the serial fraction is only 10–20 % of the total, it caps the maximum speedup to around 5x–10x no matter how many cores are added. Secondly, every barrier and critical section carries a synchronization cost that grows with the thread count, thus, beyond a sweet spot number of cores, threads spend more time idling than computing. Thirdly, OpenMP utilizes shared memory processing and thus we eventually saturate the machine's memory bandwidth with dozens of threads all streaming through the same large grid, memory (DRAM) channels become a bottleneck and per-core throughput falls off. Finally, towards the end of the simulation, work becomes unevenly distributed as some threads finish their local topplings quickly and idle at barriers while others lag behind.

In terms of weak scaling, we can see strong evidence of it in Figure 2 , where the actual speedup closely follows perfect weak scaling. Thus, our implementation is scalable for much larger core counts and data sizes without a significant drop in parallel efficiency.

Our measurements are accurate and the program's outputs are correct. The benchmarking utilized omp_get_wtime() to accurately time and record the execution time of the program. Furthermore, we ensured that initialization and output (.ppm files) creation were excluded from the benchmark such that only the simulation time was recorded. Additionally, the OpenMP program utilizes serial toppling instead of parallel to ensure that data races do not occur.

Our OpenMP solution showed evidence of weak scaling and strong scaling in that speedup raised with the number of numbers introduced until a plateau, however, the actual speedup did not come close to the ideal speedup.

MPI

The results for the MPI solution can be found in Appendix F - I.

The MPI implementation sees *reasonable* speedup - in line with what one might expect from a communication-heavy parallel problem. In general, we will discuss performance across varying configurations of problem size and / or number of processors.

*Speedup for **fixed** problem size - Strong Scaling(Figure 3).*

Our first set of tests included testing performance for **fixed** problem size and varying number of processors (see Figure 3) for that given size. We tested this configuration across a variety of sizes from `512x512` to `4096x4096` for Abelian Sandpile grids. From Figure 3, we observe *strong scaling* as the speedup increases as the number of cores are increased. Furthermore, we observe that larger problem sizes (`2048x2048, 4096x4096`) obtain larger speedup values. This could be due to the notion of 'optimal chunk size' where a balance between parallel gain and parallel overhead must be held. In addition, we see that scaleup tapers off as we increase the cores to higher numbers. This means that we don't see *perfect linear (ideal) strong scaling.* Again, we can look to parallel overhead due to chunk communication patterns as an explanation for this notion. It is important to note that *speedup per node* starts to decrease across all sizes after ≈ `4` processors. We could therefore assume that `4` processors are optimal for *efficiency.*

Figure 3: MPI Strong Scaling Speedup Graph

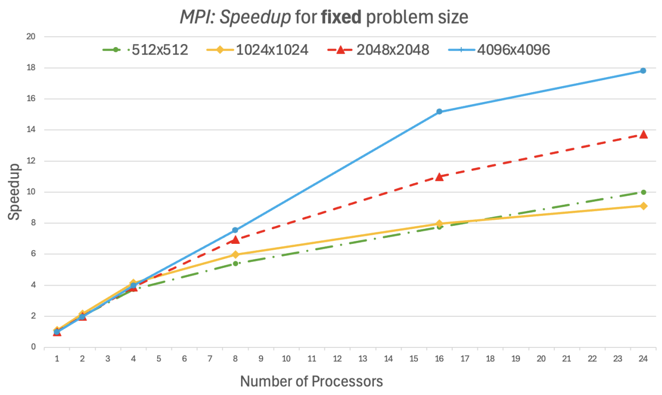
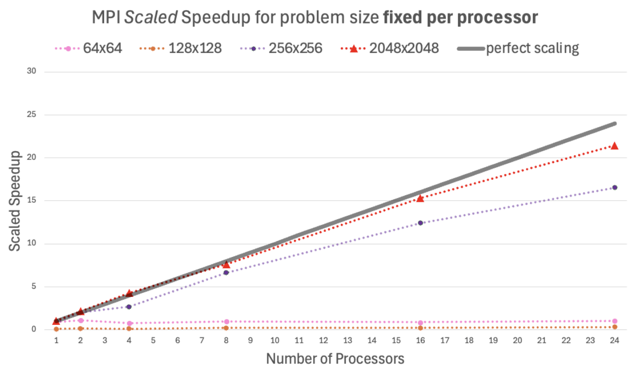


Figure 4: MPI Weak Scaling Speedup Graph



*Speedup for **fixed per processor** problem size - Weak Scaling (Figure 4).*

Next, we tested performance for fixed problem sizes *per processor* across varying numbers of processors (see Figure 2) for that given size. Again, we tested this with multiple sizes for Abelian Sandpile grids. Here the results vary. Firstly we observe that smaller sizes (`64x64, 128x128`) don't scale well at all. This might be due to the high parallel overhead at smaller chunk sizes in relation to the possible parallel gain for those sizes. Next, we see that sizes

(`256x256, 2048x2048`) scales more or less linearly. At these sizes, we can assume that parallel overhead is better balanced with parallel gain so that these scaling results can be produced. In general, we observe *weak scaling* as speedup increases when the number of processors increases for fixed problem sizes *per processor*.

Optimal conditions for speedup

From the figures above, we can see that the MPI implementation is scalable, achieving increased speedup as cores are increased in the general case. Optimal speedup is achieved at around `4` cores across various problem sizes. Speedup then becomes increasingly *less* optimal as cores are added (non-ideal strong scaling). Larger problems are less susceptible to this attenuation (e.g, `4096x4096` achieves a speedup of 18 for `24` cores compared to a speedup of 12 for `24` cores by `2048x2048`). This is most likely due to a better balance between parallel overhead and parallel gain at larger sizes.

Improvements to the MPI implementation

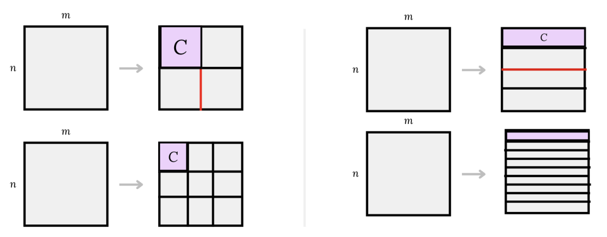
The MPI implementation performs *decently* and in accordance with expectation. However, some improvements can be made.

Firstly, all *chunks* have to converge globally. This is done through a series of `MPI_AllReduce()` calls.

```
MPI_Allreduce(&local_unstable, &global_unstable, 1, MPI_INT, MPI_LOR, cart_comm);
MPI_Allreduce(&total_sent, &global_sent, 1, MPI_INT, MPI_SUM, cart_comm);
MPI_Allreduce(&total_received, &global_received, 1, MPI_INT, MPI_SUM, cart_comm);
```

This forces synchronisation and to some degree negates the *asynchronous* operation of each chunk (depending on how chunks interleave with one another). An improvement would be to implement a better convergence-check strategy (eg: token-passing strategy). This would, therefore,improve chunk asynchronicity.

Another optimization that could be made is switching to *rectangular based chunking* instead of *quad-based chunking (see Figure 5)*. The current implementation divides the program up in 2D. This causes communication boundaries to grow exponentially. If a simpler 1D chunking method is used, communication boundaries would grow linearly. This would perhaps have an effect on



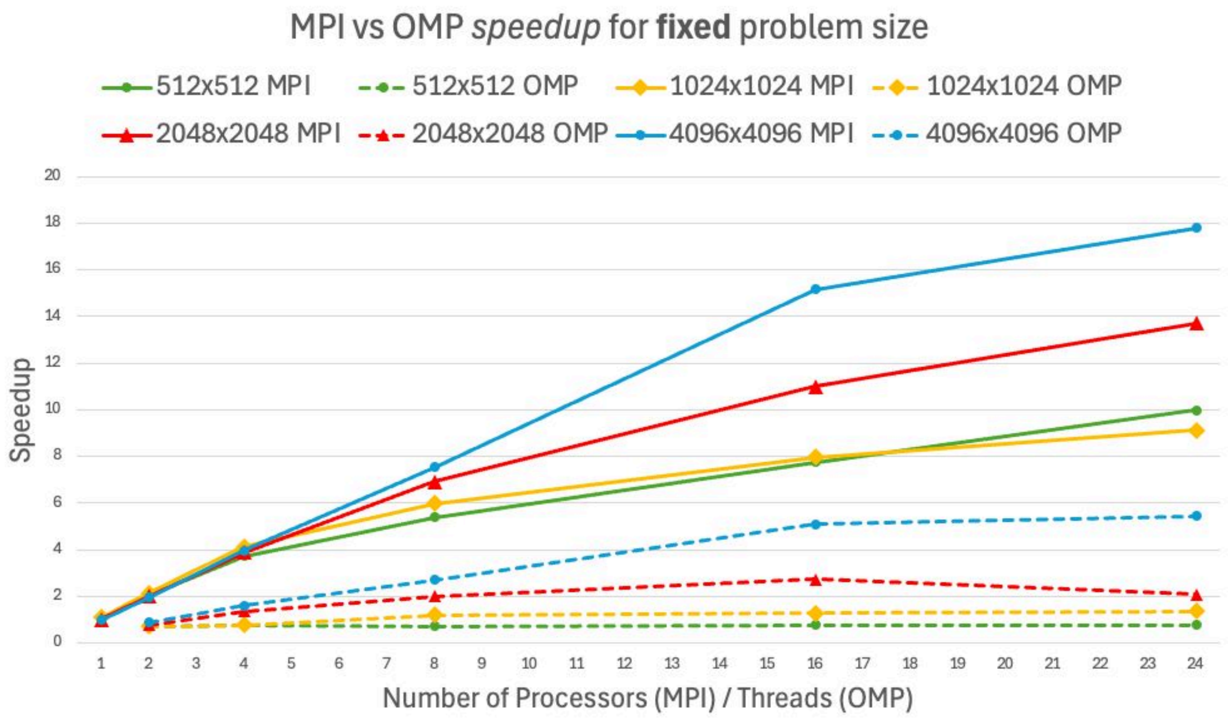
Smaller communication boundaries, but more

Larger communcation boundaries, but less performance.

Lastly, the inclusion of *dynamic load balancing*, through strategies such as *work stealing* could be implemented to better spread computational load from busy processors, to idle processors.

OpenMP vs MPI Comparison

Figure 5: MPI vs OpenMP Strong Scaling Speedup Graph



MPI's decomposition solution delivers far higher parallel performance than the two phase OpenMP code; this can be attributed to how each model handles data locality, synchronization, and serial work. In the OpenMP version, every iteration involves a full-team barrier and a critical-section merge of per-thread unstable-cell lists, followed by a single-thread "toppling" pass. This small but non-negligible fraction of the work is inherently serial and every thread must repeatedly rendezvous. This serial fraction and the growing cost of barriers throttles scalability on shared memory, limiting our best OpenMP speedup to only 5.4x on 24 cores for a 4096×4096 grid.

On the other hand, the MPI solution splits the grid into independent subdomains, each handled by one rank that performs local toppling in place and exchanges only narrow halo regions with its neighbors via nonblocking sends/receives. Thus, avoiding global critical section or single-thread update, as nearly the entire computation is parallel. While MPI does involve overhead for each exchange, when each rank handles a large subdomain that overhead is spread out over more computation, thus allowing us to achieve up to an 18× speedup on 24 ranks for a 4096×4096 grid

In summary, MPI outperforms OpenMP primarily because it maximizes the parallel fraction of the algorithm, minimizes fine-grained synchronization, and exploits distributed memory for better bandwidth. Whereas OpenMP's shared-memory two-phase design pays a heavy price in barriers, serial merges, and memory contention.

Conclusions

From the exploration above, we can draw a number of conclusions and findings.

1. Our MPI solution is relatively scalable (under both weak and strong scaling) for optimal conditions (large enough problem sizes). Our MPI implementation, therefore, performs in accordance with reasonable expectations for communication-heavy parallel algorithms.
2. OpenMP is easy to implement and performs well when each thread has a substantial workload, but its scalability is limited by synchronization and serial bottlenecks.
3. Our experiments show that parallelizing the Abelian Sandpile is clearly worthwhile for large problem sizes. The MPI solutions showed excellent scalability up to 18× speedup on 24 ranks for a 4096×4096 grid. The OpenMP version delivers more modest gains (peaking at 5.4x on 24 threads) and only for grids bigger or equal to 2048x2048.

In terms of reflections on the assignment, too much time and focus was put into ensuring correctness of the OpenMP solution thus resulting in parallel performance shortcomings. The OpenMP approach can be described as safe and being content with moderate speedups. As such, future implementations should explore more riskier OpenMP solutions while maintaining correctness.

Appendix

Appendix A: **Serial Execution Time (seconds)**

Grid Size	**Execution Time (seconds)**
256x256	5.36
512x512	28.09
1024x1024	60.98
2048x2048	164.80
4096x4096	629.20

Appendix B: **OpenMP Strong Scaling Execution Time (seconds)**

	2	**4**	**8**	**16**	**24**
256x256	9.19	7.52	7.66	8.14	8.77
512x512	38.92	37.16	38.75	36.80	37.12
1024x1024	84.26	79.43	51.22	48.36	44.76
2048x2048	215.13	122.17	82.36	60.52	79.45
4096x4096	716.15	396.22	233.62	123.45	115.60

Appendix C: **OpenMP Weak Scaling Execution Time (seconds)**

Number of Threads	1	2	4	8	16	24
Grid Size (nxm)	512x512	724x724	1024x1024	1448x1448	2048x2048	2506x2506
Execution TIme (seconds)	53.42	61.98	82.99	68.30	71.54	70.04

Appendix D: **OpenMP Strong Scaling Speedup**

	2	**4**	**8**	**16**	**24**
256x256	0.58	0.71	0.7	0.66	0.61
512x512	0.72	0.76	0.72	0.76	0.76
1024x1024	0.72	0.77	1.19	1.26	1.36
2048x2048	0.77	1.35	2.00	2.72	2.07
4096x4096	0.88	1.59	2.69	5.10	5.44

Appendix E: **OpenMP Weak Scaling Speedup**

Number of Threads	1	2	4	8	16	24
Grid Size (nxm)	512x512	724x724	1024x1024	1448x1448	2048x2048	2506x2506
Speedup	1	0.86	0.64	0.78	0.75	0.76

Appendix F: **MPI Strong Scaling Speedup**

	512x512	**1024x1024**	**2048x2048**	**4096x4096**
1	1.045068259	1.094975532	1.004798164	0.98638077
2	2.023939333	2.148571639	1.990005079	1.95239949
4	3.731255649	4.13712923	3.879203829	3.97122179
8	5.389764809	5.979776683	6.930535834	7.53325418
16	7.751791569	7.958232614	10.99284725	15.1663424
24	9.991719215	9.111391339	13.72352102	17.8044833

Appendix G: **MPI Strong Scaling Execution time**

	512x512	**1024x1024**	**2048x2048**	**4096x4096**
1	28.275665	55.86	163.306429	637.877401
2	14.60024	28.47	82.457076	322.264989
4	7.919586	14.79	42.299917	158.437386
8	5.482614	10.23	23.67638	83.521674
16	3.812022	7.686	14.926979	41.485942
24	2.957449	6.714	11.956844	35.338852

Appendix H: **MPI Weak Scaling Execution time**

	64x64	**128x128**	**256x256**	**2048x2048**
1	0.043451	0.507513	5.262383	163.306772
2	0.073365	0.712561	5.289826	161.398768
4	0.201219	1.581728	7.846805	161.37
8	0.336216	1.789627	6.382065	160.03
16	0.72909	3.46566	6.811823	159.89
24	0.910808	3.77247	7.669804	193.752359

Appendix I: **MPI Weak Scaling Speedup**

	64x64	**128x128**	**256x256**	**2048x2048**
1	0.9205772	0.098519644	1.005248003	1.0585599
2	0.54521911	0.070169431	1.000032893	1.07107385
4	0.19878838	0.031610998	0.674159738	1.0712648
8	0.11897114	0.027938783	0.828885322	1.08023496
16	0.05486291	0.014427266	0.776590936	1.08118081
24	0.04391705	0.013253916	0.689717755	0.89222139