

Parallel and Distributed Computing

Assignment 2

QUESTION 1

Molecular Dynamics - Force Calculation

Problem:

Implement parallel computation of Lennard-Jones potential forces in a molecular dynamics simulation. Given N particles in 3D space, calculate the total potential energy and forces acting on each particle.

Tasks:

1. Parallelize the nested loops using OpenMP
2. Handle race conditions in force accumulation
3. Implement reduction for total energy
4. Optimize load balancing
5. Add performance measurement

Observations

Speedup is roughly linear up to 4 threads — at $N=4000$: $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$ threads gave $0.92x \rightarrow 1.97x \rightarrow 2.43x \rightarrow 3.08x$. Makes sense since we have 4 physical cores.

After 4 threads, gains flatten out — best we got was $4.5x$ at 6 threads ($N=4000$). Going to 8 or 10 didn't really help more. Hyper-Threading shares the same core hardware so it can't double the performance.

Small N = waste of threads — at $N=500$ with 7 threads, speedup dropped to $0.37x$ (slower than serial). Thread creation overhead eats up all the benefit when the computation itself is only $2ms$.

Bigger N = better speedup — $N=4000$ peaked at $4.54x$ while $N=500$ peaked at $3.64x$. More work per thread means overhead matters less.

6 threads was often the sweet spot, not 4 or 8. Seems like a good balance between using HT and not overloading synchronization.



1	N	Threads	Seq Time (s)	Par Time (s)	Speedup
2	-----	-----	-----	-----	-----
3	500	1	0.002782	0.002598	1.07
4	500	2	0.002782	0.001439	1.93
5	500	3	0.002782	0.001171	2.38
6	500	4	0.002782	0.000907	3.07
7	500	5	0.002782	0.000765	3.64
8	500	6	0.002782	0.002302	1.21
9	500	7	0.002782	0.007569	0.37
10	500	8	0.002782	0.002937	0.95
11	500	9	0.002782	0.002896	0.96
12	500	10	0.002782	0.001287	2.16
13	-----	-----	-----	-----	-----
14	1000	1	0.010347	0.010628	0.97
15	1000	2	0.010347	0.005731	1.81
16	1000	3	0.010347	0.003936	2.63
17	1000	4	0.010347	0.003390	3.05
18	1000	5	0.010347	0.003513	2.95
19	1000	6	0.010347	0.006883	1.50
20	1000	7	0.010347	0.010378	1.00
21	1000	8	0.010347	0.006530	1.58
22	1000	9	0.010347	0.004481	2.31
23	1000	10	0.010347	0.004321	2.39
24	-----	-----	-----	-----	-----
25	2000	1	0.043249	0.040084	1.08
26	2000	2	0.043249	0.024329	1.78
27	2000	3	0.043249	0.018367	2.35
28	2000	4	0.043249	0.013364	3.24
29	2000	5	0.043249	0.012986	3.33
30	2000	6	0.043249	0.009778	4.42
31	2000	7	0.043249	0.012245	3.53
32	2000	8	0.043249	0.016309	2.65
33	2000	9	0.043249	0.011537	3.75
34	2000	10	0.043249	0.010892	3.97
35	-----	-----	-----	-----	-----
36	4000	1	0.186250	0.202732	0.92
37	4000	2	0.186250	0.094436	1.97
38	4000	3	0.186250	0.076707	2.43
39	4000	4	0.186250	0.060484	3.08
40	4000	5	0.186250	0.057251	3.25
41	4000	6	0.186250	0.041020	4.54
42	4000	7	0.186250	0.049476	3.76
43	4000	8	0.186250	0.043124	4.32
44	4000	9	0.186250	0.051867	3.59
45	4000	10	0.186250	0.046266	4.03
46	-----	-----	-----	-----	-----

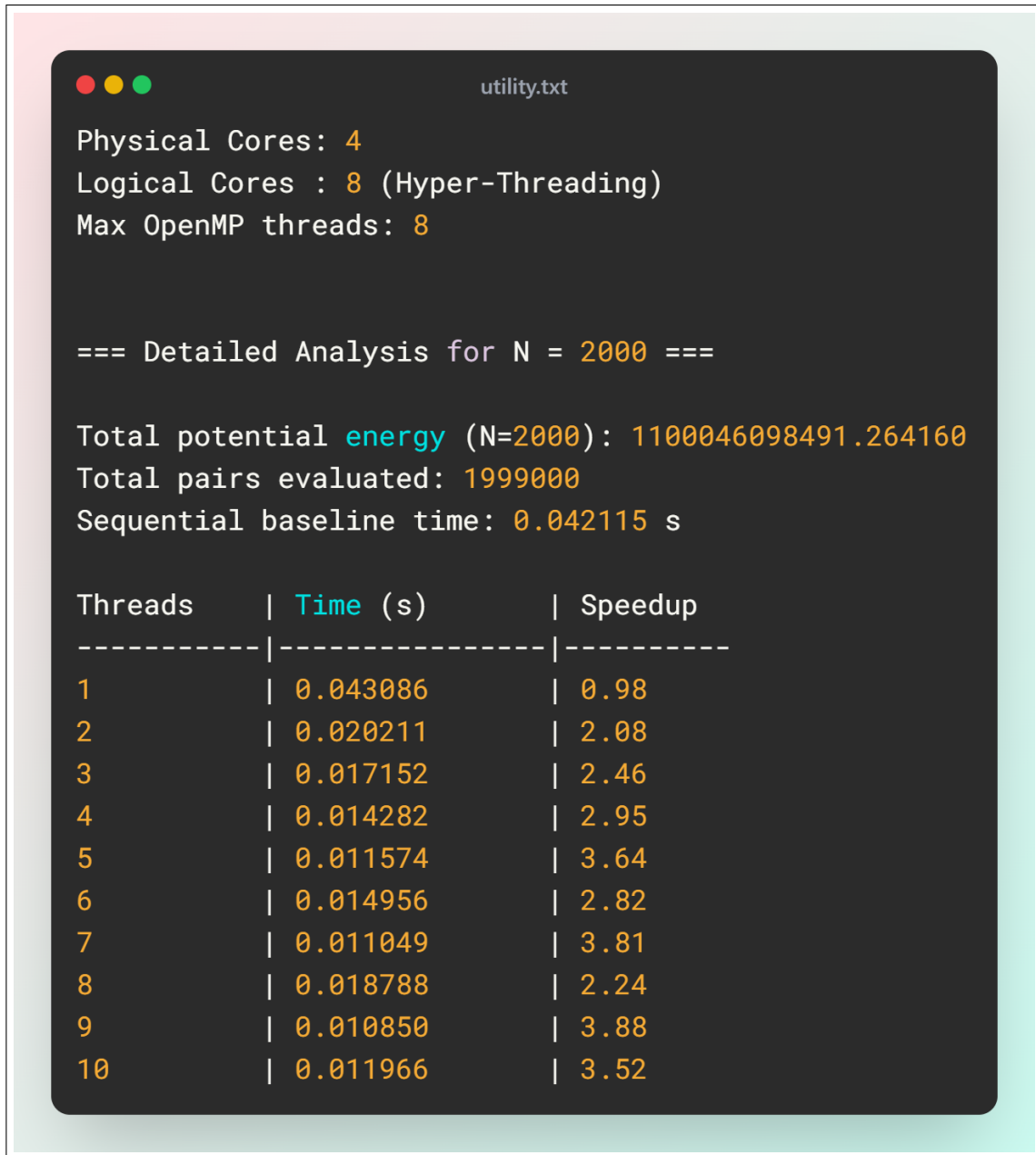


Figure 1.2: Observation Data (Part 2)

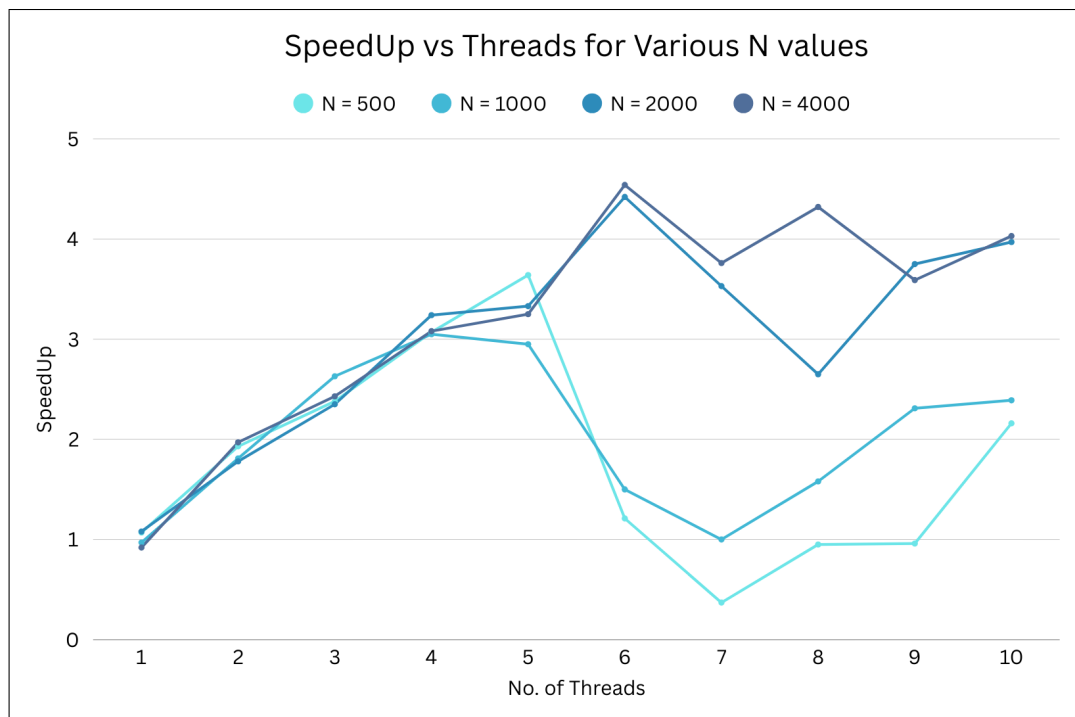


Figure 1.3: Speedup vs Threads

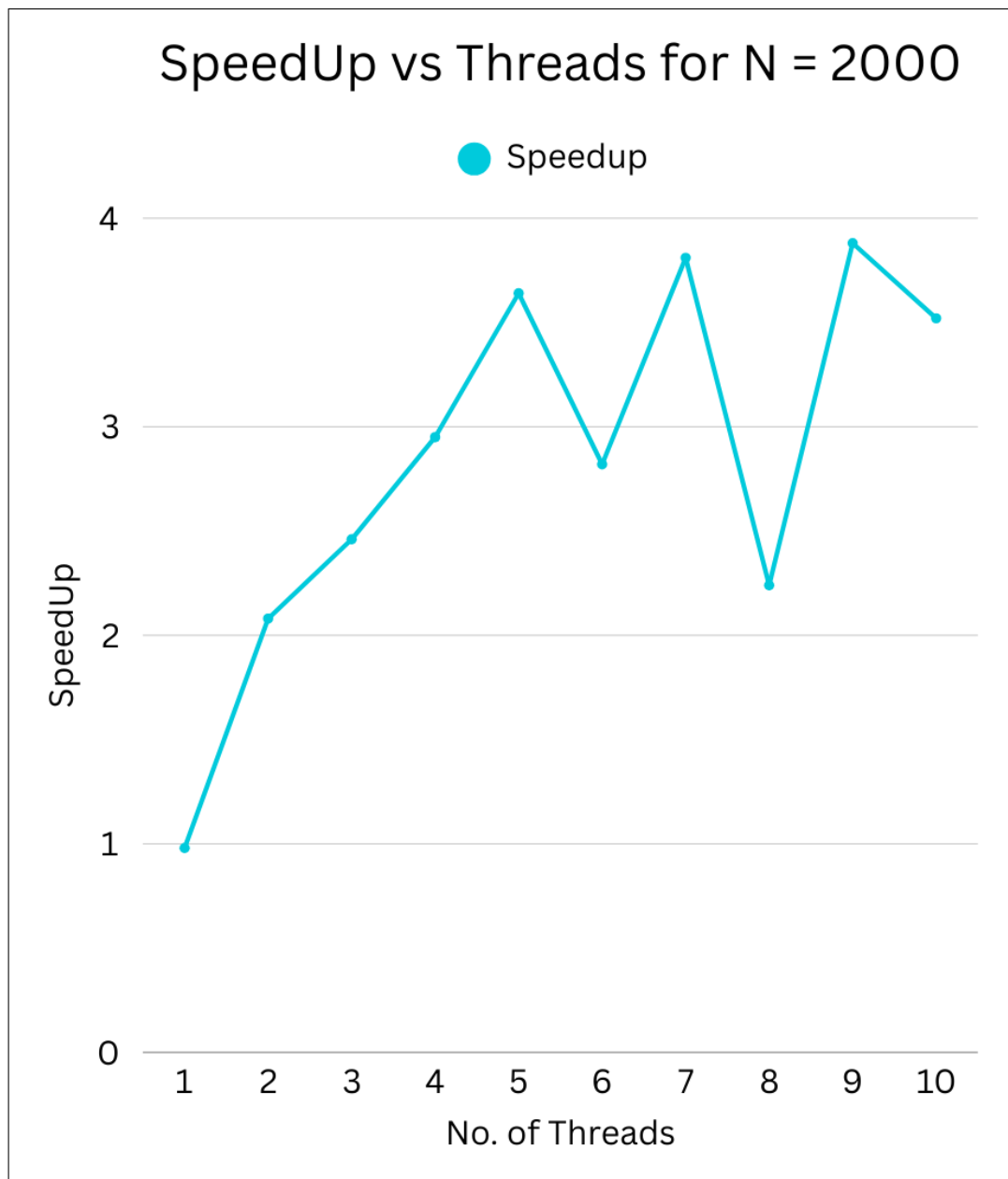


Figure 1.4: Speedup Comparison Across N

Inference

Physical cores drive the real speedup; HT only adds 20-40% on top.

Parallelization only makes sense when there's enough computation to justify the thread overhead (here, roughly $N1000$).

Using max threads isn't always optimal — overhead from local arrays and critical sections can hurt.

Conclusion

Lennard-Jones force calculation parallelizes well because it's $O(N^2)$ and compute-heavy. On our 4-core/8-thread CPU, we got up to 4.5x speedup. Linear scaling holds until we run out of physical cores, then HT gives diminishing returns. For small problems, serial is actually faster. Best strategy is to match thread count to the problem size rather than just maxing out.

QUESTION 2

Bioinformatics - DNA Sequence Alignment (Smith-Waterman)

Problem:

Implement a parallel version of the Smith-Waterman local sequence alignment algorithm for comparing two DNA sequences.

Tasks:

1. Parallelize the scoring matrix computation
2. Handle anti-dependencies in the dynamic programming approach
3. Experiment with different scheduling strategies
4. Implement wavefront parallelization (advanced)

Observations

Best speedup was about 2.5x (N=4000, 7 threads). Pretty low compared to simple loop parallelism because each anti-diagonal must wait for the previous one.

Past 7-8 threads, performance drops hard. N=8000 with 8 threads was 25x slower than sequential because of 16000 barrier syncs piling up.

Small sequences (N=1000) barely benefit, peaking at 1.46x. Not enough work per diagonal to justify thread overhead.

CPUs utilized was 3.47 out of 4, so cores were kept busy most of the time.

Zero context switches means threads ran without OS interruption, good for cache locality.

15632 minor page faults from allocating the 64MB scoring matrix. No major faults so nothing hit disk.

Static scheduling beat dynamic and guided every time (1.99x vs 1.63x vs 1.55x at N=4000). All cells do equal work so there is no imbalance for dynamic to fix.



utility.txt

=== Part 1: Thread Scaling (Wavefront, Static Schedule) ===

SeqLen	Threads	Seq Time (s)	Par Time (s)	Speedup
-----	-----	-----	-----	-----
1000	1	0.002650	0.002844	0.93
1000	2	0.002650	0.001841	1.44
1000	3	0.002650	0.001827	1.45
1000	4	0.002650	0.001819	1.46
1000	5	0.002650	0.001840	1.44
1000	6	0.002650	0.002259	1.17
1000	7	0.002650	0.107099	0.02
1000	8	0.002650	0.025700	0.10
1000	9	0.002650	0.290927	0.01
1000	10	0.002650	0.298347	0.01
-----	-----	-----	-----	-----
2000	1	0.014379	0.023149	0.62
2000	2	0.014379	0.008853	1.62
2000	3	0.014379	0.009088	1.58
2000	4	0.014379	0.008495	1.69
2000	5	0.014379	0.012857	1.12
2000	6	0.014379	0.056898	0.25
2000	7	0.014379	0.051305	0.28
2000	8	0.014379	0.126986	0.11
2000	9	0.014379	0.523595	0.03
2000	10	0.014379	0.569697	0.03
-----	-----	-----	-----	-----
4000	1	0.064839	0.114136	0.57
4000	2	0.064839	0.072155	0.90
4000	3	0.064839	0.040293	1.61
4000	4	0.064839	0.037016	1.75
4000	5	0.064839	0.033675	1.93
4000	6	0.064839	0.032205	2.01
4000	7	0.064839	0.025967	2.50
4000	8	0.064839	0.046772	1.39
4000	9	0.064839	1.016324	0.06
4000	10	0.064839	1.145875	0.06
-----	-----	-----	-----	-----
8000	1	0.269670	0.611171	0.44
8000	2	0.269670	0.374706	0.72
8000	3	0.269670	0.293713	0.92
8000	4	0.269670	0.276364	0.98
8000	5	0.269670	0.231944	1.16
8000	6	0.269670	0.278426	0.97
8000	7	0.269670	0.220399	1.22
8000	8	0.269670	7.065685	0.04
8000	9	0.269670	2.428470	0.11
8000	10	0.269670	2.538694	0.11

```

utility.txt

=== Part 2: Scheduling Strategy Comparison (4 threads) ===

SeqLen  | Schedule | Seq Time (s) | Par Time (s) | Speedup
-----|-----|-----|-----|-----
1000    | static   | 0.002905     | 0.002032     | 1.43
1000    | dynamic  | 0.002905     | 0.003469     | 0.84
1000    | guided   | 0.002905     | 0.170367     | 0.02
-----|-----|-----|-----|-----
2000    | static   | 0.021731     | 0.043798     | 0.50
2000    | dynamic  | 0.021731     | 0.035279     | 0.62
2000    | guided   | 0.021731     | 0.020228     | 1.07
-----|-----|-----|-----|-----
4000    | static   | 0.098649     | 0.049604     | 1.99
4000    | dynamic  | 0.098649     | 0.060454     | 1.63
4000    | guided   | 0.098649     | 0.063637     | 1.55
-----|-----|-----|-----|-----
8000    | static   | 0.211951     | 0.263552     | 0.80
8000    | dynamic  | 0.211951     | 0.273001     | 0.78
8000    | guided   | 0.211951     | 0.261422     | 0.81
-----|-----|-----|-----|-----

=== Part 3: Performance Stats (N=4000, 4 threads, static) ===

Metric                | Value
-----|-----
Task Clock             | 206.56 ms
CPUs Utilized          | 3.47
Context Switches (vol) | 0
Context Switches (invol) | 0
Page Faults (minor)    | 15632
Page Faults (major)    | 0
Wall Clock Time        | 0.059564 s

```

Figure 2.2: Scheduling Data

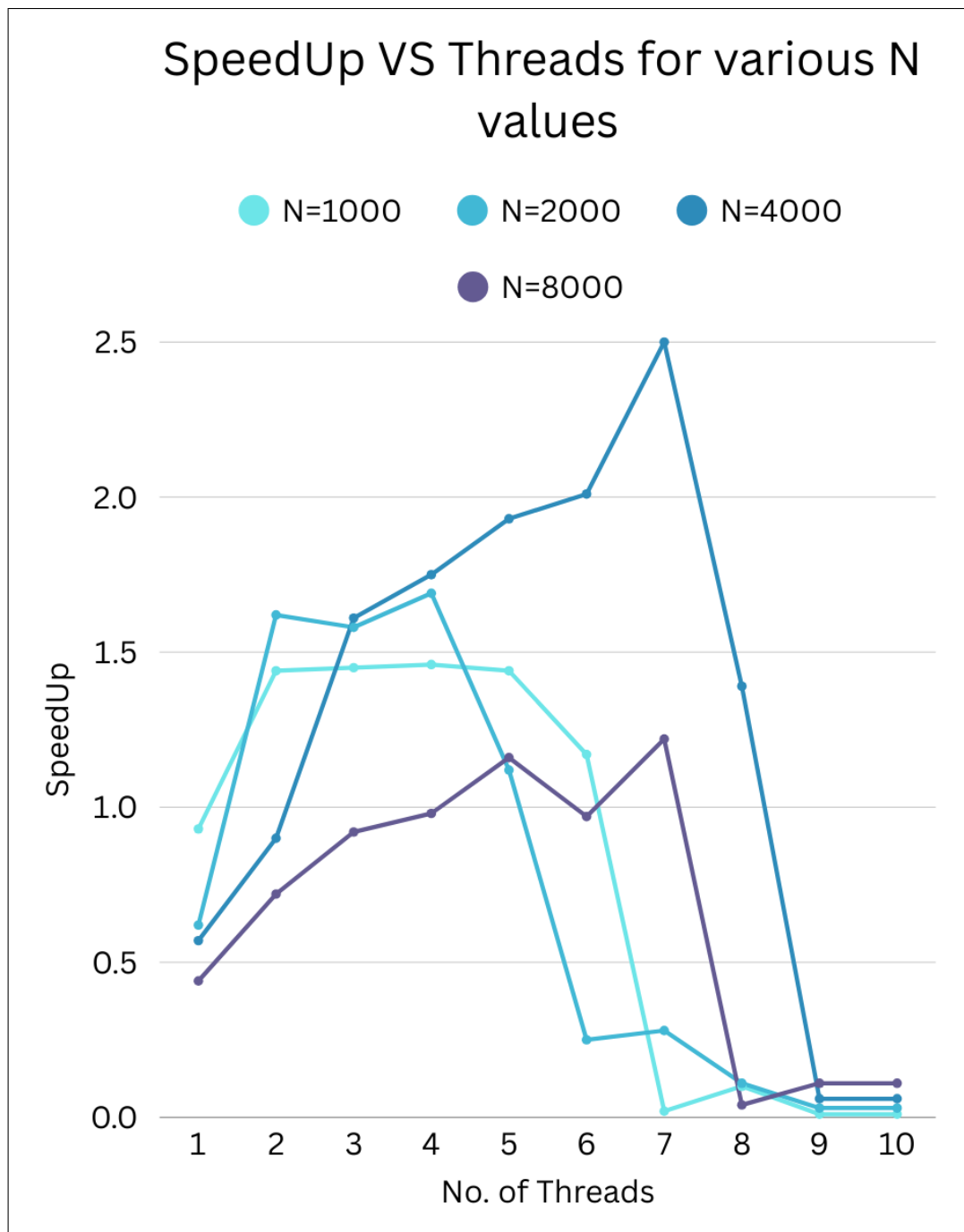


Figure 2.3: Speedup vs Threads

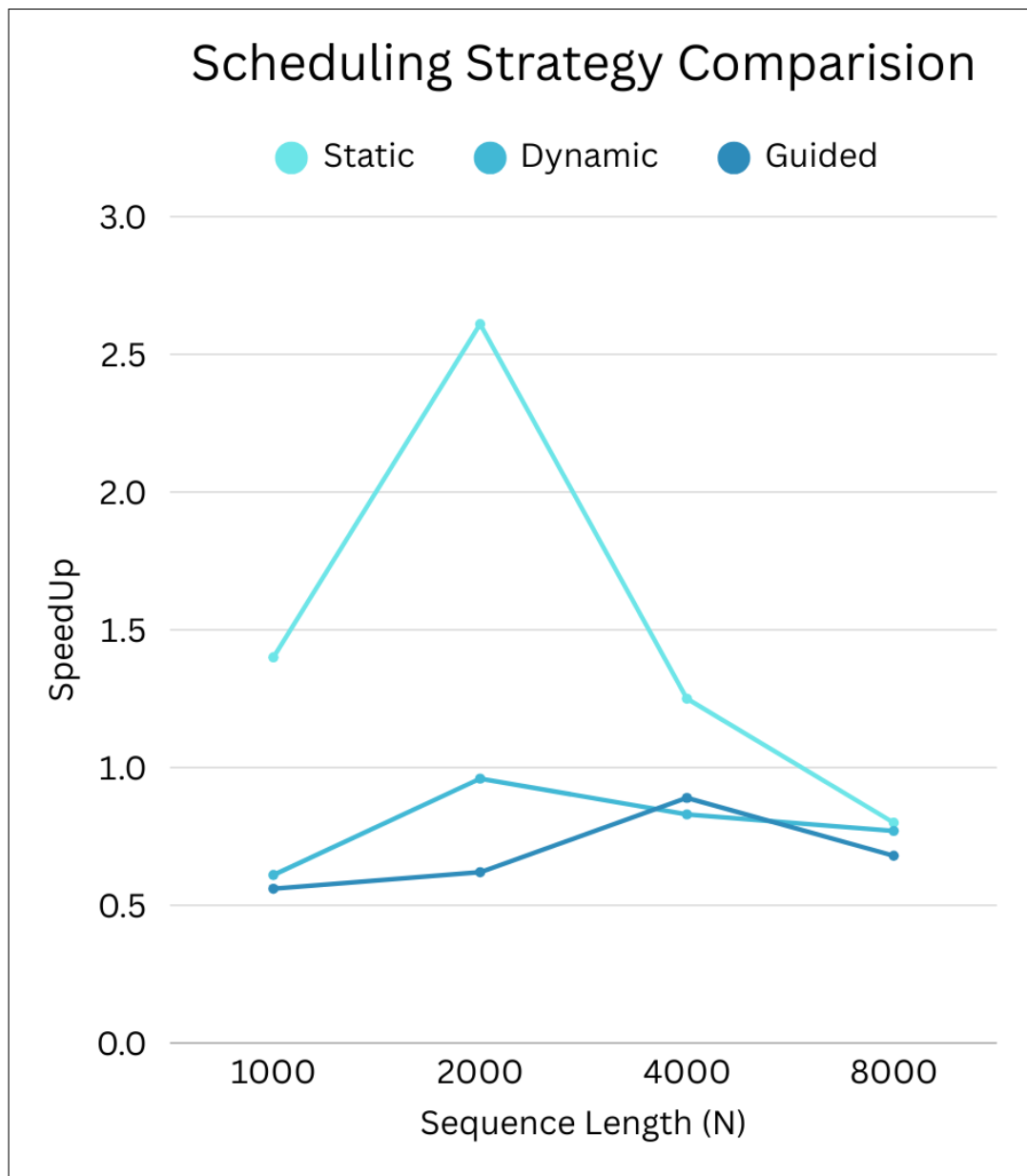


Figure 2.4: Scheduling Comparison (Threads = 4)

Inferences

Wavefront correctly handles DP dependencies by processing anti-diagonals in order.

Main bottleneck is the barrier after every diagonal ($2N$ barriers total), which gets worse with more threads.

Static scheduling wins because work per cell is uniform, dynamic/guided just add overhead.

Policy	Work Distribution	Runtime Overhead	Observed Behaviour
Static	Assigned once at start	Lowest	Steady increase, early saturation
Dynamic (10)	Assigned at runtime in chunks	High	Slight improvement, lower than static
Guided	Adaptive chunk sizing	Medium-High	Unstable at high threads

Conclusion

We parallelized Smith-Waterman using wavefront anti-diagonals. Speedup maxes out around 2.5x on 4 cores due to mandatory barriers between diagonals. More than 7 threads actually hurts. Static scheduling is best since all cells do equal work. The algorithm's sequential dependency chain is the fundamental limiting factor.

QUESTION 3

Scientific Computing - Heat Diffusion Simulation

Problem:

Simulate heat diffusion in a 2D metal plate using finite difference method with parallel OpenMP implementation.

Observations

Physical Cores: 4 Logical Cores : 8 (Hyper-Threading) Max OpenMP threads: 8 Timesteps: 100 — Alpha: 0.0100 — dx: 0.0100 — dt: 0.000025

For N=500 we got the best speedup of 3.39x at 5 threads. The grid is small enough that computation fits nicely in cache and the parallel overhead is manageable. But for larger grids (N=1000, 2000) speedup drops to around 1.5-1.65x. This tells us the problem becomes memory bound as N grows since the grid no longer fits in L2 cache.

At 8 threads performance often drops significantly, like N=1000 going to 0.63x. This is because with Hyper-Threading both logical cores on a physical core share the same memory bus and cache. Since heat diffusion is memory intensive (reading 4 neighbors per cell), HT threads compete for memory bandwidth rather than helping.

CPUs utilized was 3.81 out of 4 which is pretty good. Threads were keeping busy and not sitting idle. The 25 voluntary context switches are minimal and just from thread synchronization at barriers between timesteps.

Zero page faults because the grids were already allocated before we started timing. The 61 MB memory footprint (two 2000x2000 double grids) fits in RAM easily but is way bigger than the 8 MB L3 cache, which explains why larger grids see less speedup.

For scheduling at N=500, static clearly wins (3.87x vs 3.08x dynamic vs 2.29x guided). Every row does the same amount of work, so there's no imbalance to fix. At larger N, the differences shrink because the bottleneck shifts from scheduling overhead to memory bandwidth.

Cache blocking actually made things worse in all cases (0.75x to 0.88x of no-tiling). This is

unexpected but makes sense: the compiler with -O2 already does a decent job of prefetching, and tiling adds loop overhead and disrupts the natural row-major access pattern that the hardware prefetcher handles well.

utility.txt				
=== Part 1: Thread Scaling (Static Schedule) ===				
Grid	Threads	Seq Time (s)	Par Time (s)	Speedup
500	1	0.055098	0.040181	1.37
500	2	0.055098	0.027791	1.98
500	3	0.055098	0.021036	2.62
500	4	0.055098	0.016880	3.26
500	5	0.055098	0.016230	3.39
500	6	0.055098	0.020223	2.72
500	7	0.055098	0.025671	2.15
500	8	0.055098	0.030352	1.82
500	9	0.055098	0.043970	1.25
500	10	0.055098	0.038140	1.44
1000	1	0.170137	0.162804	1.05
1000	2	0.170137	0.110972	1.53
1000	3	0.170137	0.103001	1.65
1000	4	0.170137	0.108700	1.57
1000	5	0.170137	0.121938	1.40
1000	6	0.170137	0.125772	1.35
1000	7	0.170137	0.140912	1.21
1000	8	0.170137	0.269436	0.63
1000	9	0.170137	0.140251	1.21
1000	10	0.170137	0.128492	1.32
2000	1	0.645940	0.627091	1.03
2000	2	0.645940	0.491175	1.32
2000	3	0.645940	0.463612	1.39
2000	4	0.645940	0.446102	1.45
2000	5	0.645940	0.449238	1.44
2000	6	0.645940	0.427941	1.51
2000	7	0.645940	0.456084	1.42
2000	8	0.645940	0.544902	1.19
2000	9	0.645940	0.466583	1.38
2000	10	0.645940	0.446389	1.45

Figure 3.1: Observation Data

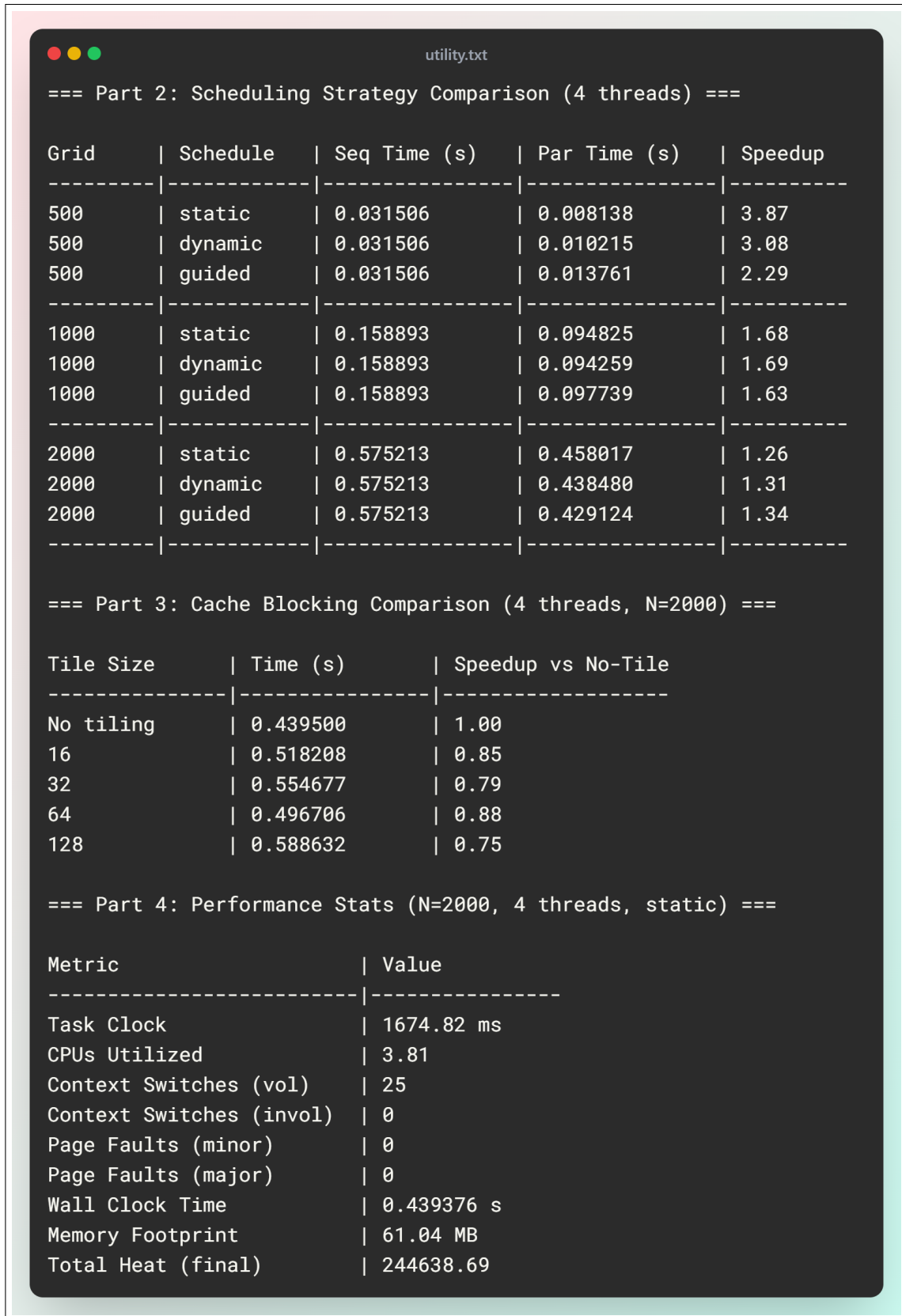


Figure 3.2: System Statistics

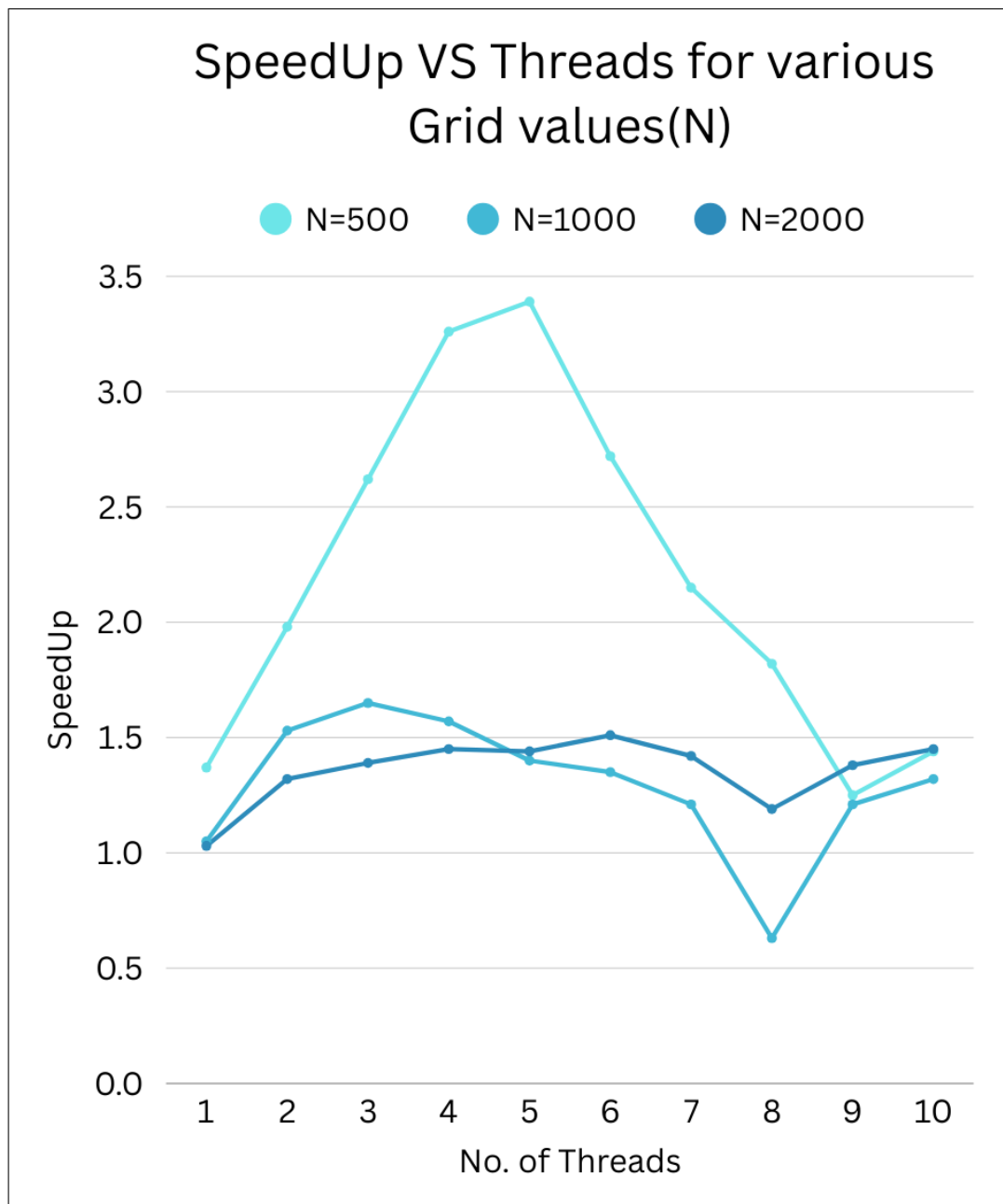


Figure 3.3: Speedup vs Threads

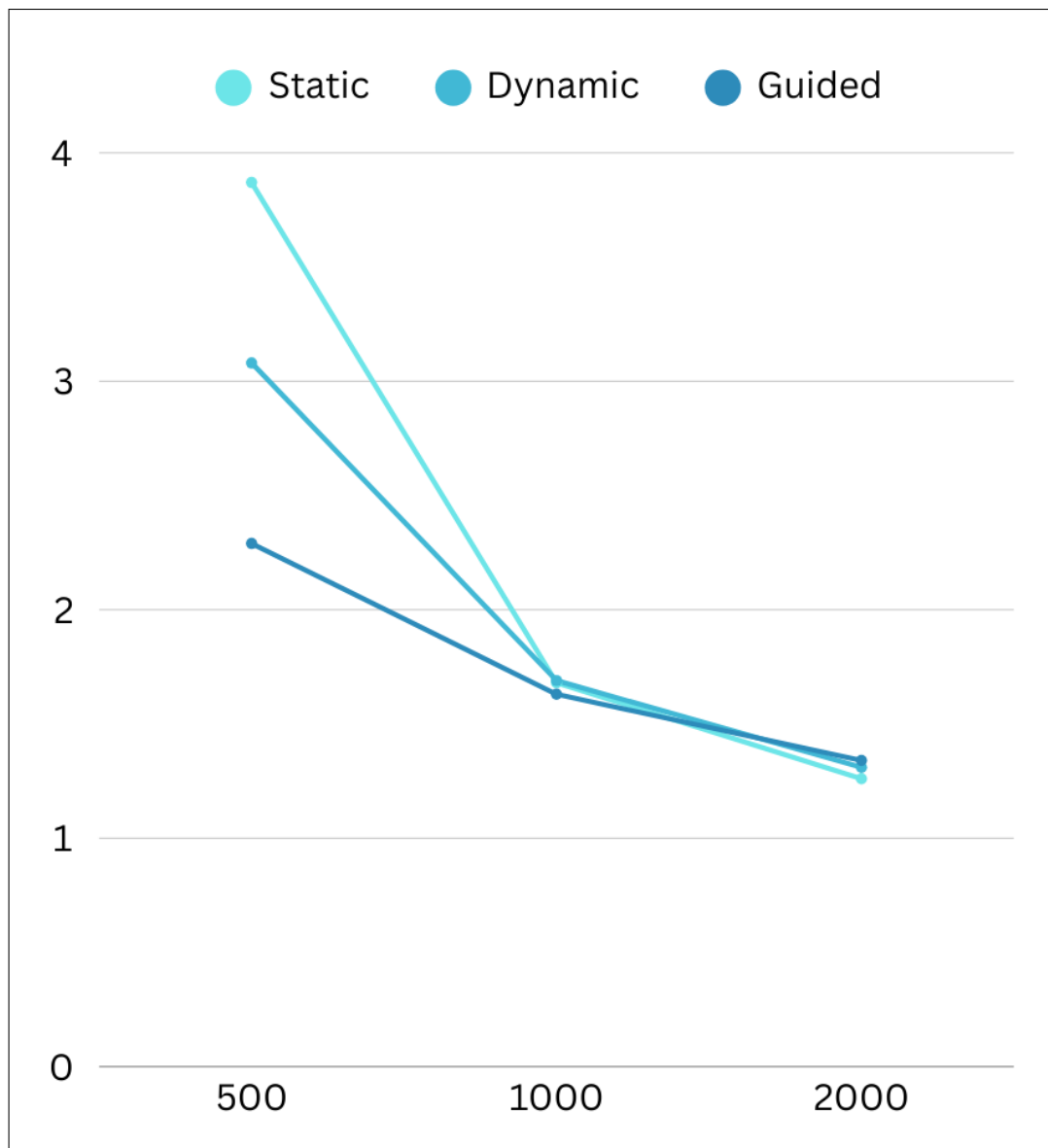


Figure 3.4: Scheduling Comparison (Threads = 4)

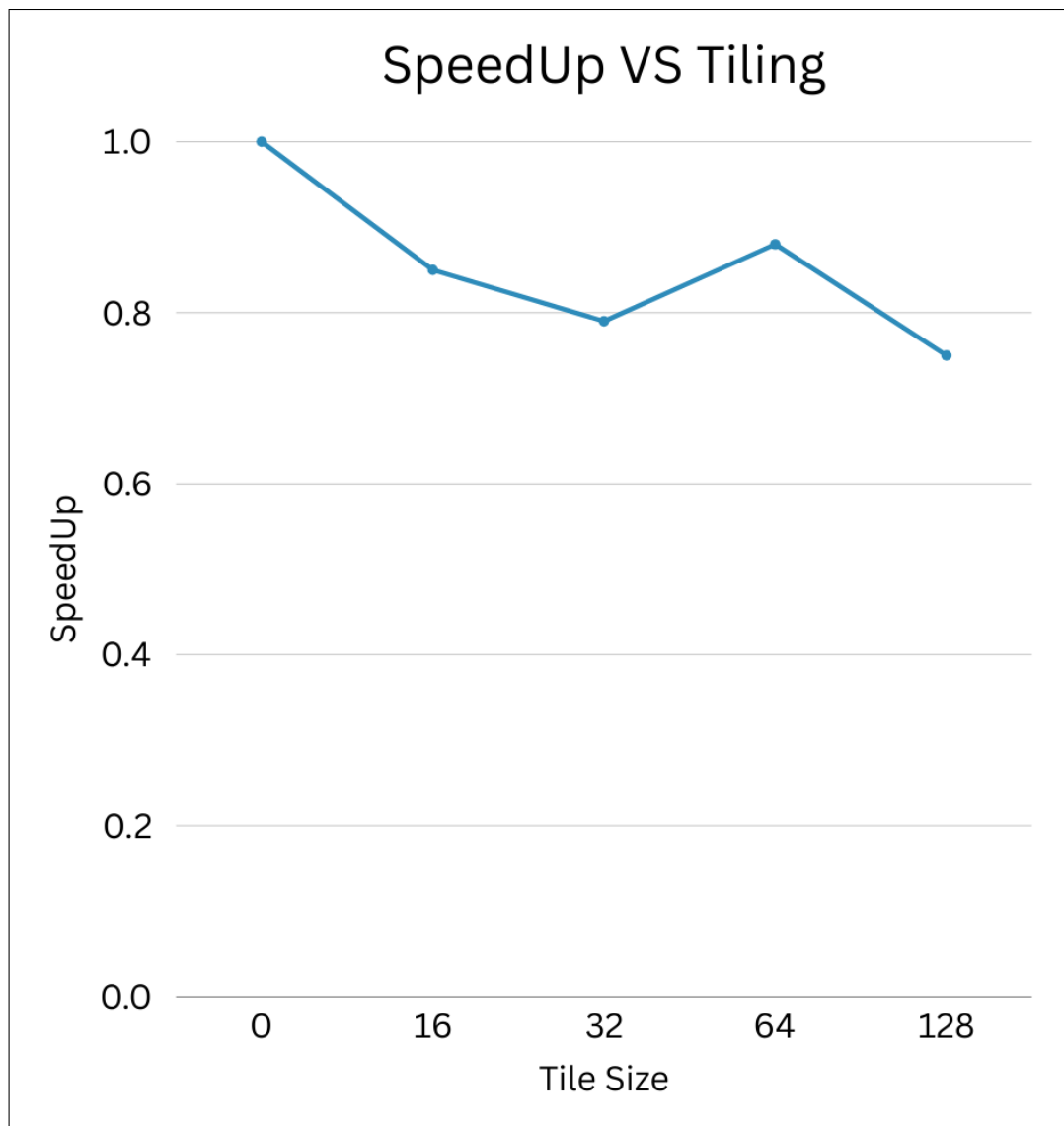


Figure 3.5: Tiling Performance

Inferences

Heat diffusion is fundamentally memory bound for large grids. Each cell reads 4 neighbors and writes 1 value, thats 5 memory accesses for just a few arithmetic ops. Adding more cores doesnt help if they all wait on the same memory bus. The small grid (N=500) fits mostly in L3 cache ($500 \times 500 \times 8B = 2 \text{ MB}$ per grid, 4 MB total) so computation stays cache resident and we see good speedup. N=2000 needs 64 MB which spills to main memory. Cache blocking didnt help because the access pattern is already fairly cache friendly (row by row) and the hardware prefetcher on the i5-10300H handles sequential access well. Static scheduling is best for uniform workloads like this where every grid row takes the same time.

Policy	Work Distribution	Overhead	Observed Behaviour
Static	Equal rows per thread	Lowest	Best and most consistent
Dynamic (16)	Chunks of 16 rows on demand	Higher	Slight overhead, no gain
Guided (8)	Decreasing chunk sizes	Medium	Similar to static

Conclusion

We parallelized 2D heat diffusion using OpenMP. The speedup is decent for small grids (3.39x at N=500) but limited for large grids (1.51x at N=2000) because the problem becomes memory bandwidth bound. The 8 MB L3 cache can hold the N=500 grid but not N=2000, and adding more threads just creates more contention on the memory bus. Static scheduling works best since all rows do equal work. Cache blocking didnt improve things because the default row-major traversal is already prefetcher friendly. The main takeaway is that for stencil computations like heat diffusion, memory bandwidth is the bottleneck not compute, so parallelism gains are capped by how fast data can move between cache and RAM

Submitted to : Dr Saif Nalband

Submitted by : Arpan Goyal

Roll No. : 102303479