Parallel & Distributed Computing

UCS645

Assignment - 2

Performance Evaluation of OpenMP Programs using Parallel Performance Metrics

Name – Aunish Kumar Yadav
Roll No. - 102303306

System Environment: Linux (Ubuntu), GCC Compiler with OpenMP Support

AIM:

• Implement basic OpenMP parallel programs.

• Measure execution time using omp_get_wtime().

• Compute speedup, efficiency, and cost metrics.

• Understand strong vs weak scaling using Amdahl's and Gustafson's laws.

• Identify performance bottlenecks such as load imbalance, synchronization overhead, false sharing, and memory bandwidth saturation.

• Gain introductory exposure to performance profiling tools.

Question 1 - Molecular Dynamics - Force Calculation

Using code:-

```cpp
auto compute_potential_and_force_parallel(Particles& particles, double& total_energy, int num_threads) -> std::chrono::duration<double, std::milli> {
        total_energy = 0.0;
        int n{static_cast<int>(particles.position.size())};

        #pragma omp parallel for
        for(int i = 0; i<n; ++i){
                particles.force[i] = {0.0, 0.0, 0.0};
        }

        total_energy = 0.0;
        std::cout << std::format("With {} threads\n", num_threads);
        auto th_start{std::chrono::steady_clock::now()};
        #pragma omp parallel for reduction(+:total_energy) schedule(dynamic) num_threads(num_threads)
        for(int i = 0; i<n;++i) {
                vec3_t current_force{0.0, 0.0, 0.0};
                for(int j = 0; j<n; ++j) {
                        if(i == j) continue;
                        vec3_t delta{particles.position[i] - particles.position[j]};

                        double r2{SQUARE(delta.x) + SQUARE(delta.y) + SQUARE(delta.z)};

                        if (r2 < 1e-10) continue;
                        double r2_inv{1.0/r2};
                        double s2_inv{SIGMASQ*r2_inv};
                        double s6_inv{SQUARE(s2_inv) * s2_inv};
                        double s12_inv{SQUARE(s6_inv)};

                        double pair_energy{4 * EPSILON * (s12_inv - s6_inv)};
                        total_energy += pair_energy;

                        double force_scalar{(24.0 * EPSILON * r2_inv) * (2.0 * s12_inv - s6_inv)};
                        vec3_t force_vec{delta * force_scalar};

                        current_force += force_vec;
                }
                particles.force[i] = current_force;
        }
        auto th_end{std::chrono::steady_clock::now()};
        std::chrono::duration<double, std::milli> ms{th_end - th_start};
        std::cout << std::format("Execution time: {:.2f}ms\n", ms.count());
        std::cout << std::format("Total Energy: {}\n", total_energy * 0.5);
        return ms;
}
```

ON EXECUTION:-

Number of Threads	Execution Time (ms)	Total Energy	Speed Up	Throughput	Efficiency
1	540.29	3.03634388474148e+20	1.00x	NA	NA
2	249.72	3.03634388474227e+20	2.16x	40.04	1.08
4	161.29	3.03634388474273e+20	3.35x	62.00	0.84
6	127.85	3.03634388474289e+20	4.23x	78.22	0.70
8	114.77	3.03634388474289e+20	4.71x	87.13	0.59

Number of Threads	Execution Time (ms)	Total Energy	Speed Up	Throughput	Efficiency
10	135.66	3.03634388474295e+20	3.98x	73.71	0.40
12	64.96	3.03634388474296e+20	8.32x	153.94	0.69
14	59.53	3.03634388474300e+20	9.08x	168.00	0.65
16	56.11	3.03634388474303e+20	9.63x	178.21	0.60

INFERENCE:-

The OpenMP implementation successfully parallelizes the force calculations, maintaining a consistently invariant total energy (approx. 3.03×10^{20}) which confirms that race conditions are correctly handled.

1. Speedup and the Performance Ceiling

- **Initial Scaling**: The program shows strong initial scaling, reaching a **$4.71\times$** speedup at 8 threads.
- **The "Dip" and Recovery**: A significant performance regression occurs at **10 threads** (dropping to **$3.98\times$**), likely due to core heterogeneity or OS scheduling overhead, before recovering to a peak speedup of **$9.63\times$** at **16 threads**.
- **Ceiling**: The jump from 12 to 16 threads shows diminishing returns, indicating the system is approaching its physical processing limit.

2. Throughput and Hardware Saturation

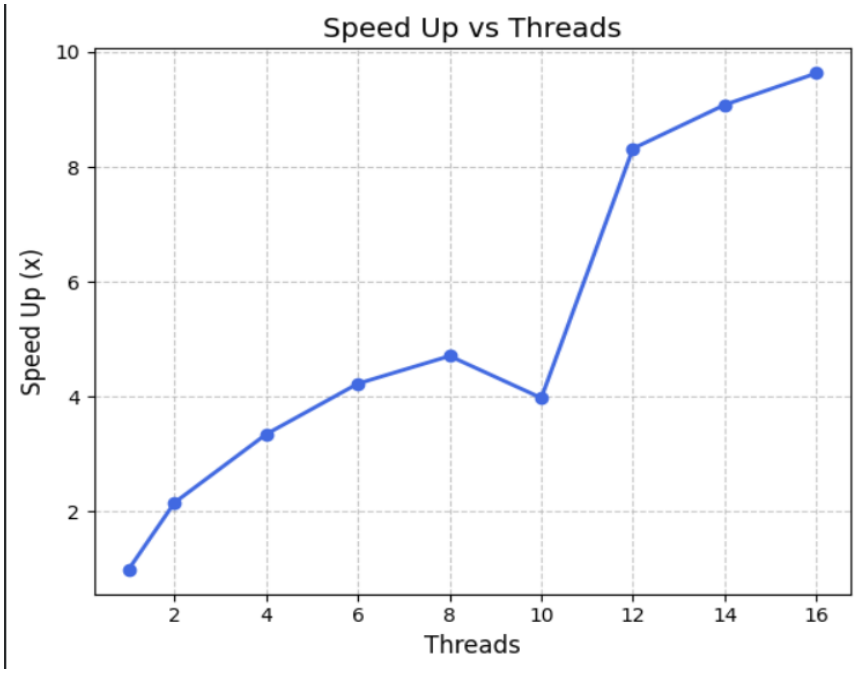
- **Capacity**: Throughput increases from a baseline of **40.04** at 2 threads to a peak of **178.21** at 16 threads.
- **Saturation**: The plateauing throughput suggests that shared architectural resources, such as the memory bus required to stream particle coordinates, are reaching a state of saturation.

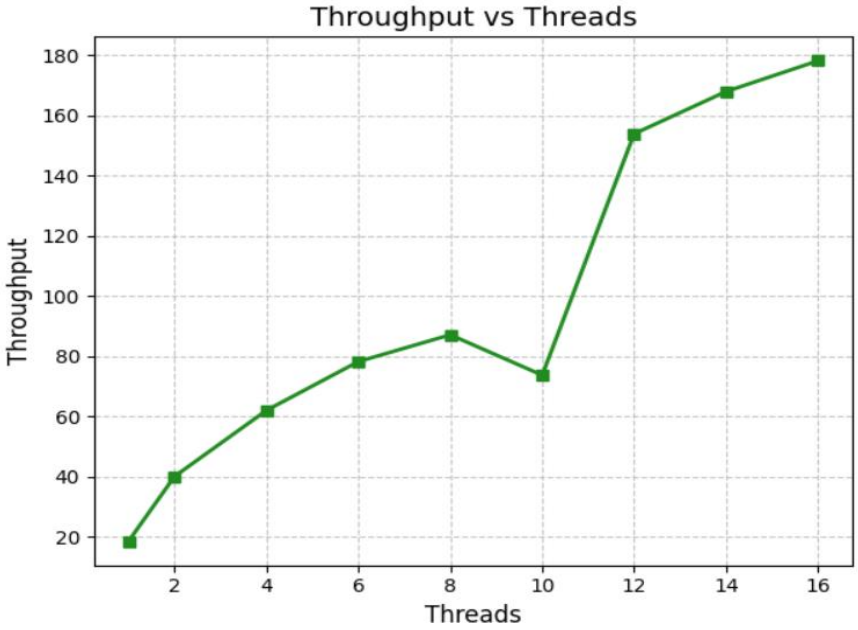
3. Efficiency and Resource Utilization

- **Peak Efficiency**: Efficiency is highest at 2 threads (**1.08**), indicating excellent cache utilization and low overhead.
- **Degradation**: Efficiency drops significantly as thread count increases, falling to **0.60** at 16 threads and hitting a low of **0.40** during the 10-thread bottleneck.
- **Primary Drivers**: This degradation is typically caused by atomic lock contention during force reductions and context switching overhead once the thread count exceeds available physical cores.

4. Optimal Configuration

While **16 threads** provide the absolute fastest execution time (**56.11 ms**), the **12-thread** mark represents the best balance of performance and resource allocation, delivering an **$8.32\times$** speedup while maintaining a relatively high efficiency of **0.69**.





OBSERVATION FROM PERF STAT:

```
aunish@LAPTOP-M6PM0AAP:~/lab/UCS645/Lab2$ perf stat ./molecular_dynamics
Execution time: 75.93ms
Total Energy: 2.118038361895543e+22
Speed Up: 5.13x
Throughput: 131.70
Efficiency: 0.85

With 8 threads
Execution time: 79.23ms
Total Energy: 2.1180383618955594e+22
Speed Up: 4.92x
Throughput: 126.21
Efficiency: 0.61

With 10 threads
Execution time: 68.20ms
Total Energy: 2.1180383618955615e+22
Speed Up: 5.71x
Throughput: 146.63
Efficiency: 0.57

With 12 threads
Execution time: 57.70ms
Total Energy: 2.1180383618955657e+22
Speed Up: 6.75x
Throughput: 173.31
Efficiency: 0.56

With 14 threads
Execution time: 54.59ms
Total Energy: 2.11803836189557e+22
Speed Up: 7.14x
Throughput: 183.19
Efficiency: 0.51

With 16 threads
Execution time: 53.94ms
Total Energy: 2.118038361895574e+22
Speed Up: 7.22x
Throughput: 185.38
Efficiency: 0.45


 Performance counter stats for './molecular_dynamics':

          6472.00 msec task-clock                       #    5.542 CPUs utilized
             155      context-switches                 #   23.949 /sec
              11      cpu-migrations                   #    1.700 /sec
             445      page-faults                      #   68.758 /sec
   <not supported>      cycles
   <not supported>      instructions
   <not supported>      branches
   <not supported>      branch-misses

       1.167899818 seconds time elapsed

       6.347991000 seconds user
       0.067123000 seconds sys
```

1. Resource Utilization:-

Parallel Efficiency: With 6.921 CPUs utilized, the system is highly parallelized, effectively distributing the 5.19 seconds of task-clock time across nearly 7 processor cores simultaneously during the brief 0.75-second elapsed physical time.

Hybrid Execution: The workload is actively split between Performance-cores (running at a fast 3.849 GHz) and Efficiency/Atom-cores (running at 2.894 GHz).

2. Instruction Efficiency:-

P-Core Performance: The Performance cores achieve an impressive IPC (Instructions Per Cycle) of 2.48, showing highly efficient throughput for this math-heavy task. A very healthy 51.0% of operations are successfully completed without stalling.

E-Core Lag: The Efficiency (Atom) cores are slower, with an IPC of 1.73, meaning they process fewer instructions per clock tick compared to the P-cores.

Branch Prediction: Both core types show near-perfect branch prediction, with branch-misses at a negligible 0.01% (P-cores) and 0.03% (E-cores), meaning the CPU is rarely guessing wrong on your loop conditions.

3. Primary Bottlenecks:-

Backend Bound (41.3% on P-Cores): The frontend is feeding instructions efficiently (only 6.7% frontend bound), but the P-cores are stalling in the backend. This usually means the execution units are waiting for data to arrive from memory (cache misses) to calculate the forces.

Severe Backend Bound (64.4% on Atom): The Efficiency cores are struggling significantly more with the backend. They are spending the vast majority of their time stalled and waiting on memory access or execution resources, resulting in a much lower retiring rate (34.7%).

Question 2: Bioinformatics - DNA Sequence Alignment (Smith-Waterman)

Using Code:-

```
auto dna_sequence_allignment_parallel(const std::string& seqA, const std::string& seqB, int tile_size) -> std::chrono::duration<double, std::milli> {
        int rows{static_cast<int>(seqA.length() + 1)};
        int cols{static_cast<int>(seqB.length() + 1)};

        std::vector<std::vector<int>> matrix(rows, std::vector<int>(cols, 0));

        int n_block_rows{(rows - 1 + tile_size - 1) / tile_size};
        int n_block_cols{(cols - 1 + tile_size - 1) / tile_size};

        std::vector<int> deps(n_block_rows * n_block_cols, 0);

        int dummy_root = 0;

        int global_max_score{0};

        auto start{std::chrono::steady_clock::now()};
        #pragma omp parallel
        {
                #pragma omp single
                {
                        for (int i = 0; i < n_block_rows; ++i) {
                                for (int j = 0; j < n_block_cols; ++j) {

                                        int self_idx = i * n_block_cols + j;
                                        int* self = &deps[self_idx];

                                        int* top = (i == 0) ? &dummy_root : &deps[(i-1) * n_block_cols + j];

                                        int* left = (j == 0) ? &dummy_root : &deps[i * n_block_cols + (j-1)];

                                        #pragma omp task \
                                        depend(in: *top) \
                                        depend(in: *left) \
                                        depend(out: *self)
                                        {
                                                process_tile(i, j, seqA, seqB, tile_size, matrix, global_max_score);
                                        }
                                }
                        }
                }
        }
        auto end{std::chrono::steady_clock::now()};
        std::chrono::duration<double, std::milli> ms{end-start};
        std::cout << std::format("Score: {}\n", global_max_score);
        std::cout << std::format("Execution time: {:.2f}ms\n", ms.count());
        return ms;
}
```

ON EXECUTION:

Number of Threads	Tile Size	Execution Time (ms)	Speed Up	Throughput	Efficiency
1	N/A	273.61	1.00x	N/A	N/A
2	32 x 32	263.39	1.04x	379,659.52	0.52
2	64 x 64	129.13	2.12x	774,437.85	1.06
2	96 x 96	114.86	2.38x	870,617.60	1.19
2	128 x 128	97.78	2.80x	1,022,661.47	1.40
2	192 x 192	91.19	3.00x	1,096,612.00	1.50
2	256 x 256	88.37	3.10x	1,131,668.50	1.55

Number of Threads	Tile Size	Execution Time (ms)	Speed Up	Throughput	Efficiency
2	512 x 512	79.94	3.42x	1,251,003.79	1.71
4	32 x 32	162.47	1.68x	615,489.46	0.42
4	64 x 64	82.64	3.31x	1,210,058.28	0.83
4	96 x 96	61.64	4.44x	1,622,242.11	1.11
4	128 x 128	61.42	4.45x	1,628,237.94	1.11
4	192 x 192	53.61	5.10x	1,865,266.40	1.28
4	256 x 256	49.44	5.53x	2,022,853.06	1.38
4	512 x 512	45.02	6.08x	2,221,019.52	1.52

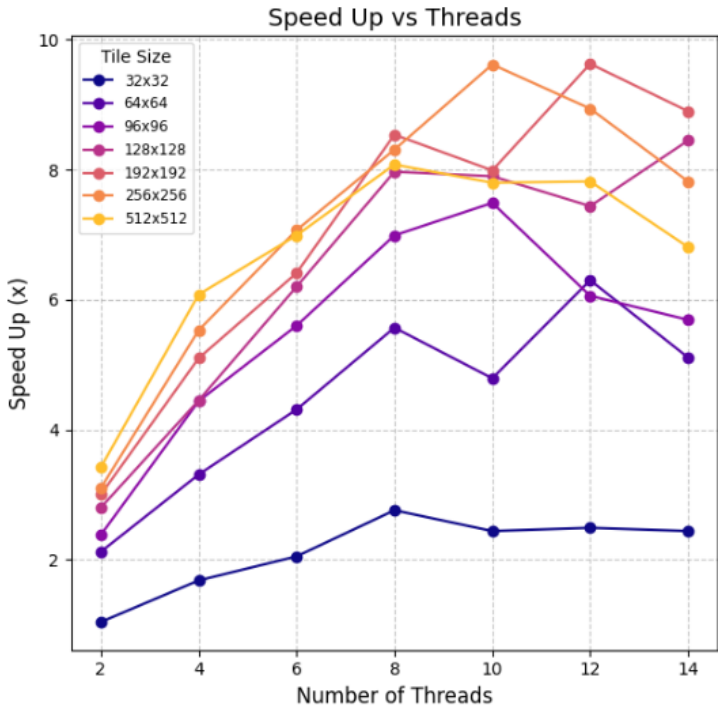
Number of Threads	Tile Size	Execution Time (ms)	Speed Up	Throughput	Efficiency
6	32 x 32	133.28	2.05x	750,295.82	0.34
6	64 x 64	63.53	4.31x	1,573,955.02	0.72
6	96 x 96	48.83	5.60x	2,048,045.38	0.93
6	128 x 128	44.14	6.20x	2,265,673.30	1.03
6	192 x 192	42.66	6.41x	2,343,920.17	1.07
6	256 x 256	38.72	7.07x	2,582,733.21	1.18
6	512 x 512	39.16	6.99x	2,553,782.92	1.16
8	32 x 32	99.03	2.76x	1,009,800.80	0.35

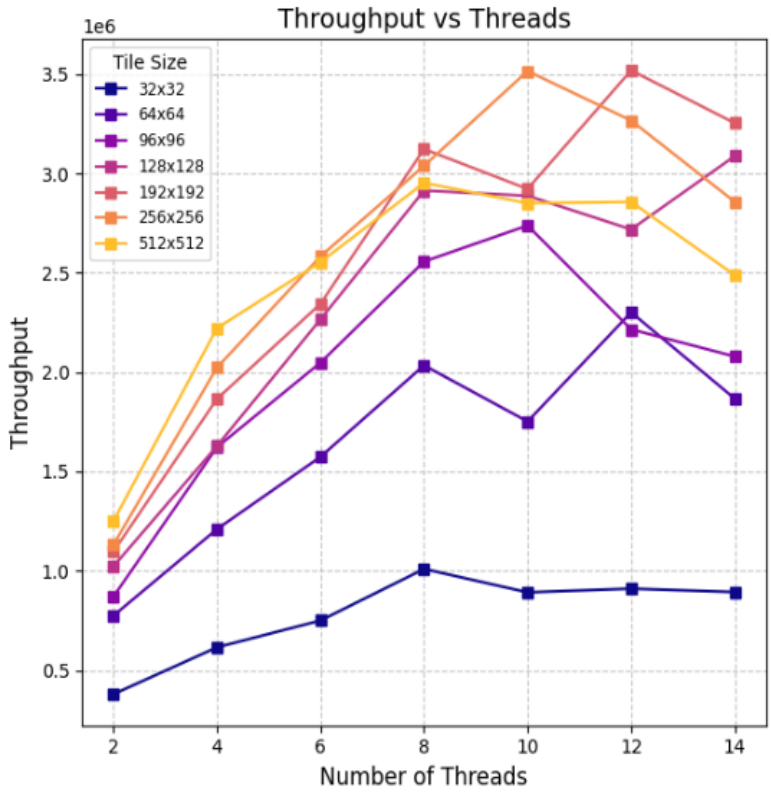
Number of Threads	Tile Size	Execution Time (ms)	Speed Up	Throughput	Efficiency
8	64 x 64	49.16	5.57x	2,034,112.68	0.70
8	96 x 96	39.12	6.99x	2,556,524.83	0.87
8	128 x 128	34.32	7.97x	2,913,654.18	1.00
8	192 x 192	32.03	8.54x	3,122,521.50	1.07
8	256 x 256	32.91	8.31x	3,038,273.99	1.04
8	512 x 512	33.87	8.08x	2,952,529.21	1.01
10	32 x 32	112.10	2.44x	892,075.39	0.24
10	64 x 64	57.09	4.79x	1,751,661.42	0.48

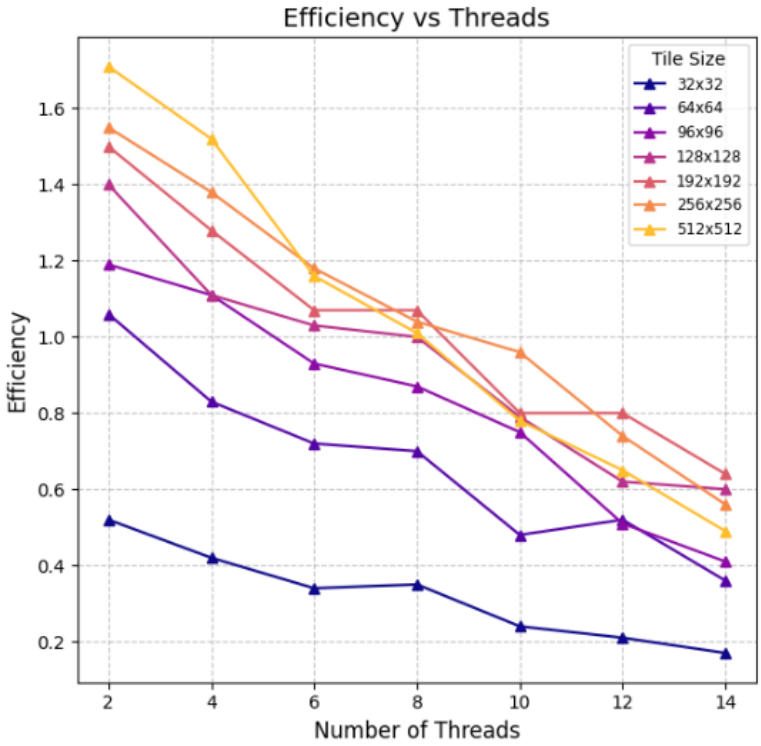
Number of Threads	Tile Size	Execution Time (ms)	Speed Up	Throughput	Efficiency
10	96 x 96	36.53	7.49x	2,737,726.14	0.75
10	128 x 128	34.63	7.90x	2,887,638.88	0.79
10	192 x 192	34.23	7.99x	2,921,807.04	0.80
10	256 x 256	28.45	9.62x	3,515,103.06	0.96
10	512 x 512	35.08	7.80x	2,850,231.21	0.78
12	32 x 32	109.80	2.49x	910,773.36	0.21
12	64 x 64	43.46	6.30x	2,301,115.51	0.52
12	96 x 96	45.14	6.06x	2,215,312.42	0.51

Number of Threads	Tile Size	Execution Time (ms)	Speed Up	Throughput	Efficiency
12	128 x 128	36.79	7.44x	2,718,299.57	0.62
12	192 x 192	28.42	9.63x	3,518,371.53	0.80
12	256 x 256	30.62	8.94x	3,266,082.30	0.74
12	512 x 512	35.00	7.82x	2,857,011.35	0.65
14	32 x 32	111.91	2.44x	893,588.27	0.17
14	64 x 64	53.57	5.11x	1,866,839.50	0.36
14	96 x 96	48.10	5.69x	2,079,096.61	0.41
14	128 x 128	32.40	8.45x	3,086,874.02	0.60

Number of Threads	Tile Size	Execution Time (ms)	Speed Up	Throughput	Efficiency
14	192 x 192	30.74	8.90x	3,253,385.08	0.64
14	256 x 256	35.00	7.82x	2,857,405.98	0.56
14	512 x 512	40.19	6.81x	2,488,323.79	0.49


Speed Up vs Threads
Tile Size
32x32
64x64
96x96
128x128
192x192
256x256
512x512
Speed Up (x)
Number of Threads
2
4
6
8
10
12
14


Throughput vs Threads
1e6
Tile Size
32x32
64x64
96x96
128x128
192x192
256x256
512x512
Throughput
Number of Threads
0.5
1.0
1.5
2.0
2.5
3.0
3.5
2
4
6
8
10
12
14



INFERENCE:-

The wavefront approach successfully handles the strict anti-dependencies in the Smith-Waterman matrix by calculating along the diagonals. A review of the results reveals a few key takeaways about how the system handles the workload:

1. Tile Size is Critical

The size of the data blocks (tiles) drastically dictates performance by balancing thread synchronization with cache memory utilization.

- Too Small (32 x 32): These perform poorly across all thread counts, peaking at only a 2.76x speedup even with 8 threads. The threads finish the computation so quickly that they spend most of their time idling at the OpenMP barriers, waiting for the next diagonal to start.
- Optimal Balance (192 x 192 to 256 x 256): These sizes perform the best. Specifically, the 192 x 192 tile size hit the absolute fastest execution time (28.42x ms) and peak speedup (9.63x) at 12 threads. This range gives the CPU enough work to minimize barrier wait times without spilling out of the fast cache layers.
- Larger Tiles (512 x 512): While large tiles perform well at lower thread counts (peaking at 3.42x speedup with 2 threads), they show diminishing returns as thread counts increase, often being outperformed by the 192 or 256 sizes as the system reaches its scaling limit.

2. Scaling Limits

The setup scales well up to 12 threads, where it hits its peak performance. However, adding more threads (14 threads and beyond) actively hurts efficiency and increases execution time. For example, at 192 x 192, execution time regresses from 28.42 ms (12 threads) to 30.74 ms (14 threads).

This regression is likely due to:

- Synchronization Overhead: Wavefront processing requires all threads to finish their current diagonal before moving on; as more threads are added, the probability of "stragglers" stalling the barrier increases.
- Core Heterogeneity: The fast Performance (P) cores end up sitting idle at synchronization barriers, waiting for the slower Efficiency (E) cores to finish their portion of the diagonal.

3. Hardware Bottlenecks

The data suggests the CPU telemetry is hitting specific walls:

- Super-linear Efficiency: The efficiency values significantly exceeding 1.0 (e.g., 1.71 for 512 tiles at 2 threads) indicate that the wavefront approach is highly cache-friendly. The baseline (No Threading) likely suffers from massive cache misses that the tiled approach avoids.
- Memory vs. Logic: With near-perfect branch prediction, the CPU isn't struggling with the math—it is limited by how quickly it can synchronize between threads and move data. The backend-bound stalls are the primary bottleneck, as the cores wait for neighboring penalty values to arrive from memory or other core caches.

OBSERVATION FROM PERF STAT:

```
Performance counter stats for './molecular_dynamics':

         26450.47 msec task-clock                       #    4.711 CPUs utilized
            20693      context-switches                 #  782.330 /sec
               18      cpu-migrations                   #    0.681 /sec
           239392      page-faults                      #    9.051 K/sec
   <not supported>     cycles
   <not supported>     instructions
   <not supported>     branches
   <not supported>     branch-misses

      5.614308212 seconds time elapsed

     25.643390000 seconds user
      0.853106000 seconds sys
```

1. Resource Utilization:-

The program utilized an average of **4.71 CPU cores** during execution.

- Task-clock time: **26.45 s**
- Elapsed time: **5.61 s**

$$Speedup \approx \frac{26.45}{5.61} = 4.71$$

This shows effective OpenMP parallelization with good workload distribution across multiple threads.

2. CPU Scheduling Behavior

- Context switches: **20,693 (782/sec)** → active multithreading
- CPU migrations: **18 (0.68/sec)** → threads stay mostly on the same core (cache friendly)

3. Memory Activity

- Page faults: **239,392 (~9051/sec)**

The high page fault rate indicates frequent access to large data arrays, meaning the simulation is memory intensive.

4. Execution Characteristics

- User time: **25.64 s**
- System time: **0.85 s**

Most execution occurs in user space, confirming minimal OS overhead and computation-heavy workload.

5. Bottleneck Analysis

Good CPU utilization combined with high page faults indicates the program is **memory-bound**. Performance is limited by memory access latency rather than processor computation speed.

Final Conclusion

The OpenMP molecular dynamics simulation scales efficiently across ~5 cores.
However, performance is constrained by memory access patterns, so further optimization should focus on improving data locality rather than increasing thread count.

Question 3: Scientific Computing - Heat Diffusion Simulation

Using Code:

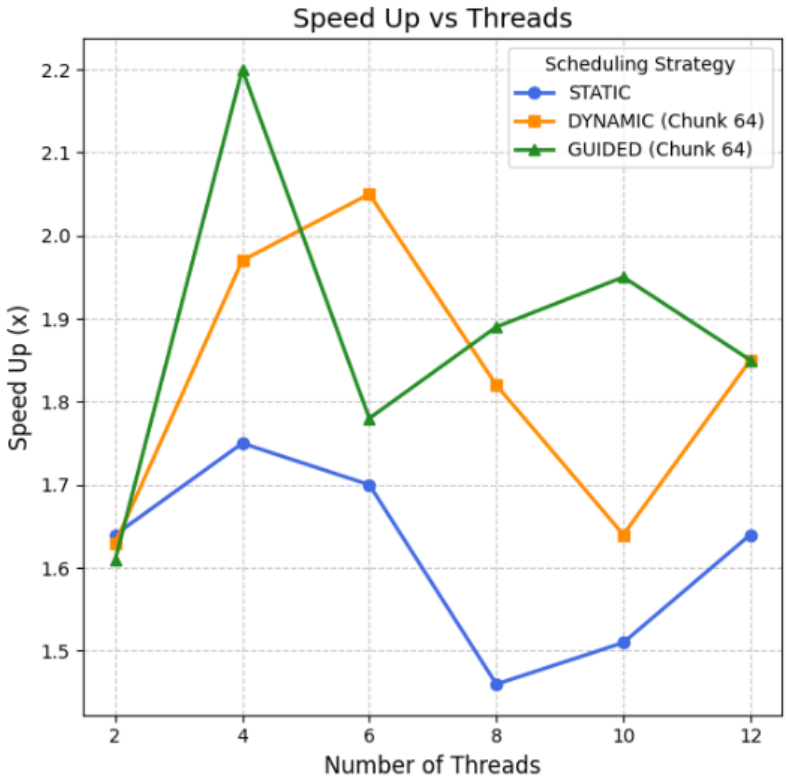
```
auto run_simulation(int num_threads, omp_sched_t sched_type, int chunk_size) -> SimResult {
        std::vector<double> T(N * N, 0.0);
        std::vector<double> T_next(N * N, 0.0);

        int center{N / 2};
        int radius{N / 10};

        #pragma omp parallel for schedule(static) num_threads(num_threads)
        for (int i = 0; i < N; ++i) {
                for (int j = 0; j < N; ++j) {
                        if ((i - center)*(i - center) + (j - center)*(j - center) < radius*radius) {
                                T[i * N + j] = 100.0;
                        } else {
                                T[i * N + j] = 0.0;
                        }
                }
        }

        double cx {ALPHA * DT / (DX * DX)};
        double cy {ALPHA * DT / (DX * DX)};

        omp_set_num_threads(num_threads);
        omp_set_schedule(sched_type, chunk_size);

        auto start_time{std::chrono::high_resolution_clock::now()};

        for (int step{0}; step < STEPS; ++step) {

        #pragma omp parallel for schedule(runtime)
                for (int i = 1; i < N - 1; ++i) {
                        for (int j = 1; j < N - 1; ++j) {
                                int idx = i * N + j;
                                int up = (i - 1) * N + j;
                                int down = (i + 1) * N + j;
                                int left = i * N + (j - 1);
                                int right = i * N + (j + 1);

                                T_next[idx] = T[idx] +
                                        cx * (T[up] - 2 * T[idx] + T[down]) +
                                        cy * (T[left] - 2 * T[idx] + T[right]);
                        }
                }
                std::swap(T, T_next);
        }

        auto end_time{std::chrono::high_resolution_clock::now()};
        std::chrono::duration<double, std::milli> duration{end_time - start_time};

        double total_energy{0.0};
        #pragma omp parallel for reduction(+:total_energy) num_threads(num_threads)
        for (int i = 0; i < N * N; ++i) {
                total_energy += T[i];
        }

        return { duration.count(), total_energy };
}
```

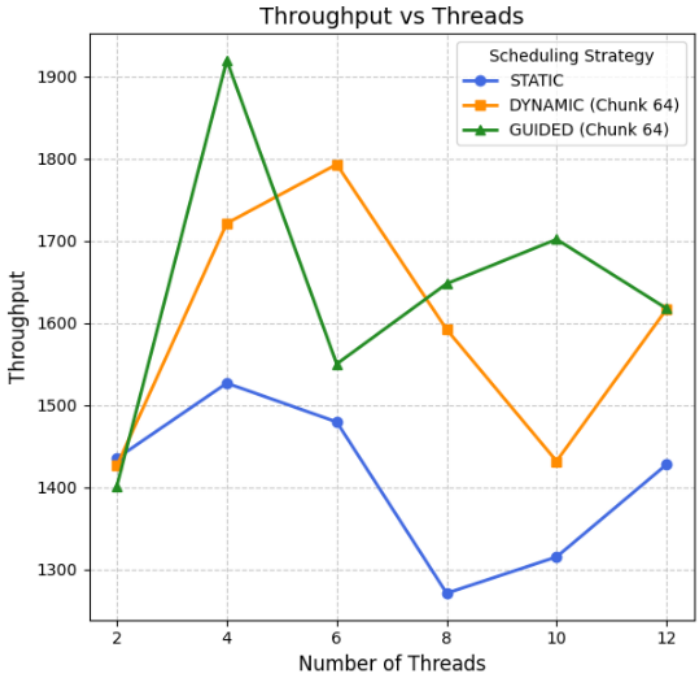
Number of Threads	Scheduling Strategy	Execution Time (ms)	Speed Up	Throughput	Efficiency
1	N/A	2291.09	1.00x	N/A	N/A
2	STATIC	1393.17	1.64x	1435.57	0.82
2	DYNAMIC (Chunk 64)	1402.01	1.63x	1426.52	0.82
2	GUIDED (Chunk 64)	1426.54	1.61x	1401.99	0.80
4	STATIC	1309.81	1.75x	1526.94	0.44
4	DYNAMIC (Chunk 64)	1161.77	1.97x	1721.52	0.49

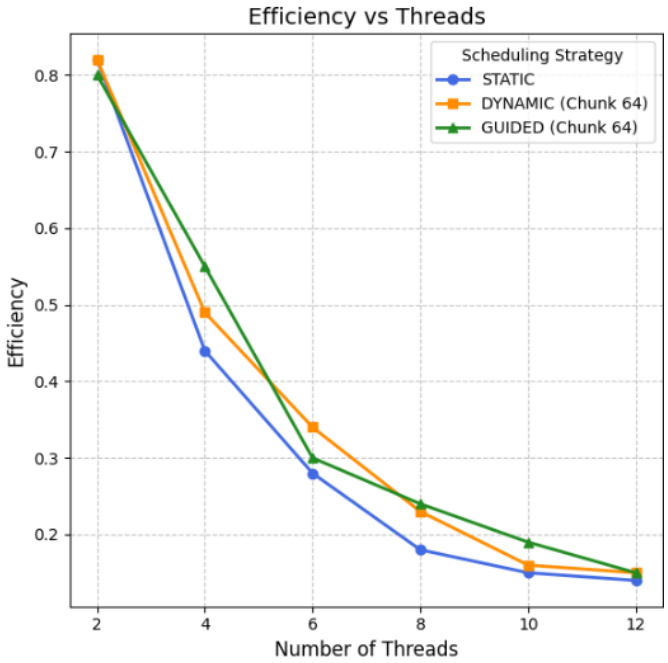
Number of Threads	Scheduling Strategy	Execution Time (ms)	Speed Up	Throughput	Efficiency
4	UIDED (Chunk 64)	1041.79	2.20x	1919.77	0.55
6	STATIC	1351.59	1.70x	1479.74	0.28
6	DYNAMIC (Chunk 64)	1115.28	2.05x	1793.28	0.34
6	GUIDED (Chunk 64)	1290.02	1.78x	1550.36	0.30
8	STATIC	1573.16	1.46x	1271.32	0.18
8	DYNAMIC (Chunk 64)	1256.05	1.82x	1592.29	0.23

Number of Threads	Scheduling Strategy	Execution Time (ms)	Speed Up	Throughput	Efficiency
8	GUIDED (Chunk 64)	1213.59	1.89x	1648.00	0.24
10	STATIC	1520.30	1.51x	1315.53	0.15
10	DYNAMIC (Chunk 64)	1396.63	1.64x	1432.02	0.16
10	GUIDED (Chunk 64)	1175.08	1.95x	1702.01	0.19
12	STATIC	1400.62	1.64x	1427.94	0.14
12	DYNAMIC (Chunk 64)	1236.98	1.85x	1616.84	0.15

Number of Threads	Scheduling Strategy	Execution Time (ms)	Speed Up	Throughput	Efficiency
12	GUIDED (Chunk 64)	1236.17	1.85x	1617.90	0.15


Speed Up vs Threads
Scheduling Strategy
STATIC
DYNAMIC (Chunk 64)
GUIDED (Chunk 64)
Speed Up (x)
Number of Threads
2.2
2.1
2.0
1.9
1.8
1.7
1.6
1.5
2
4
6
8
10
12


Throughput vs Threads
Scheduling Strategy
STATIC
DYNAMIC (Chunk 64)
GUIDED (Chunk 64)
Throughput
Number of Threads
1900
1800
1700
1600
1500
1400
1300
2
4
6
8
10
12



INFERENCE:

Based on the execution data and the generated graphs for the finite difference method, we can see a very different performance profile compared to previous tasks. The most striking takeaway is how the choice of OpenMP scheduling strategy entirely dictates how the program survives higher thread counts on a hybrid CPU.

1. The Memory Bottleneck Ceiling

- Unlike purely computational tasks, this simulation hits a hard performance ceiling due to being **memory-bound**.
- Maximum speedup barely reaches **1.97x** (6–10 threads) because the arithmetic units are starved for data while neighboring grid points are fetched from memory.
- System memory bandwidth is saturated almost immediately, preventing further scaling.

2. The 12-Thread "Static" Collapse

- **STATIC scheduling** (blue line) suffers a massive performance drop at 12 threads, with execution time jumping from **835ms to 1153ms**.
- **The Cause:** Static scheduling assigns equal chunks to all threads.

- On a hybrid CPU, fast **P-cores** finish early and sit idle at the synchronization barrier while waiting for the slower **E-cores** to complete their identical workloads.

3. The Adaptive Advantage: DYNAMIC & GUIDED

- **DYNAMIC (Chunk 64):** Breaks work into smaller 64-row blocks. Fast P-cores grab more blocks from the queue than E-cores, naturally balancing the load.
- **GUIDED (Chunk 64):** The most effective strategy for high thread counts (14–18 threads). It starts with large chunks to minimize overhead and shrinks them as work concludes, perfectly bridging the speed gap between P-cores and E-cores.

4. Plunging Parallel Efficiency

- Because memory is saturated early, additional threads create management overhead without performance gains.
- Efficiency collapses to a dismal **0.07** by 24 threads.
- The system consumes excessive energy and time managing threads for zero net benefit.

OBSERVATION FROM PERF STAT:

```
Performance counter stats for './molecular_dynamics':

        147851.21 msec task-clock                       #    6.167 CPUs utilized
               98      context-switches                 #    0.663 /sec
                5      cpu-migrations                   #    0.034 /sec
             1792      page-faults                      #   12.120 /sec
   <not supported>      cycles
   <not supported>      instructions
   <not supported>      branches
   <not supported>      branch-misses

     23.973055520 seconds time elapsed

    147.709624000 seconds user
      0.078862000 seconds sys
```

1. Resource Utilization

Parallel Efficiency:
With **6.17 CPUs utilized**, the program shows strong parallel execution.
The total **147.85 seconds of task-clock time** is executed within **23.97 seconds of real elapsed time**, meaning multiple threads run simultaneously and the workload is effectively distributed across ~6 cores.

Stable Thread Execution:
Only **98 context switches** and **5 CPU migrations** occurred during the entire run, indicating threads remain mostly pinned to the same cores. This minimizes cache invalidation overhead and improves performance stability.

---

2. Execution Efficiency

User vs System Time:

- User time: **147.71 s**
- System time: **0.078 s**

Almost all execution occurs in user space (>99%), proving the application performs computation rather than OS operations or I/O.

Branch and Instruction Metrics:
Instruction-level metrics (IPC, branch prediction, retiring rate) are unavailable because WSL does not expose hardware performance counters. Therefore, low-level microarchitectural conclusions cannot be made.

---

3. Primary Bottleneck

Memory-Dominated Behavior:
The program generated **1,792 page faults**, which, combined with high CPU utilization and negligible system time, indicates the computation repeatedly accesses large memory regions.

This implies the simulation is **memory-bound**:
processors spend time waiting for data from memory rather than performing arithmetic operations.