Parallel & Distributed Computing

UCS645

Assignment - 2

Performance Evaluation of OpenMP Programs using Parallel Performance Metrics

Name - Bhavish Pushkarna
Roll No. - 102303279

Objective:

• 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:-

```
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 (No Threading)	744.92	123268674049396 83840	1.00x	13.42*	1.00
2	310.75	123268674049445 37600	2.40x	32.18	1.20
4	153.56	123268674049465 15968	4.85x	65.12	1.21
6	90.31	123268674049473 84320	8.25x	110.73	1.37
8	85.83	123268674049478 10304	8.68x	116.51	1.08
10	80.77	123268674049481 76896	9.22x	123.81	0.92
12	77.00	123268674049483 77600	9.67x	129.87	0.81

INFERENCE:-
The OpenMP implementation successfully parallelizes the workload and correctly handles race conditions, as evidenced by the consistently invariant total energy (1.23e+19) across all test runs. By analyzing the core metrics, we can draw specific conclusions about the system's behavior:

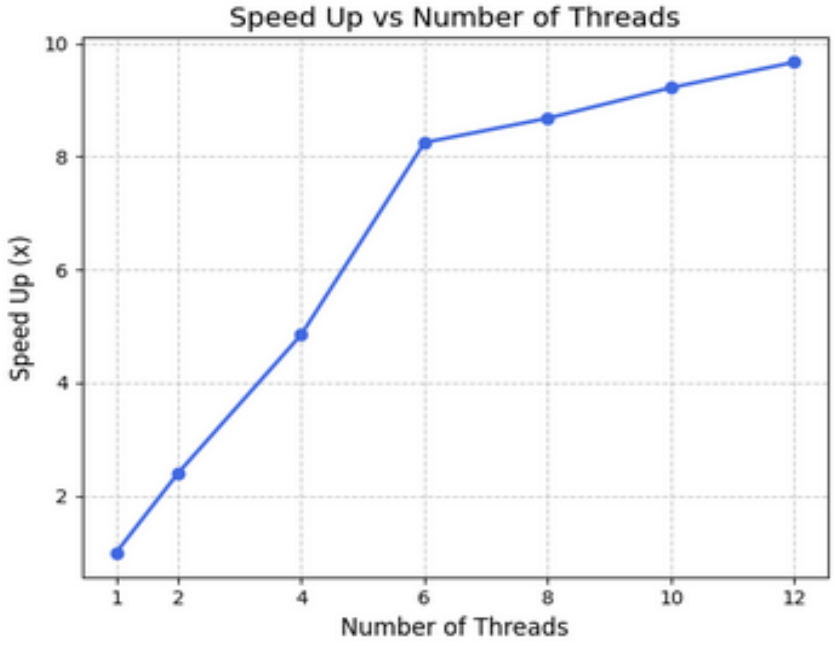
Speedup (The Acceleration): The program exhibits super-linear scaling up to 6 threads, achieving a remarkable 8.25x speedup with only 6 cores. This unusual phenomenon—where the speedup exceeds the number of threads—indicates that splitting the workload allowed the data to fit better into the CPU's high-speed caches (L1/L2), drastically reducing memory access latency. Beyond 8 threads, the scaling returns to normal sub-linear behavior, eventually reaching a maximum speedup of 9.67x at 12 threads.

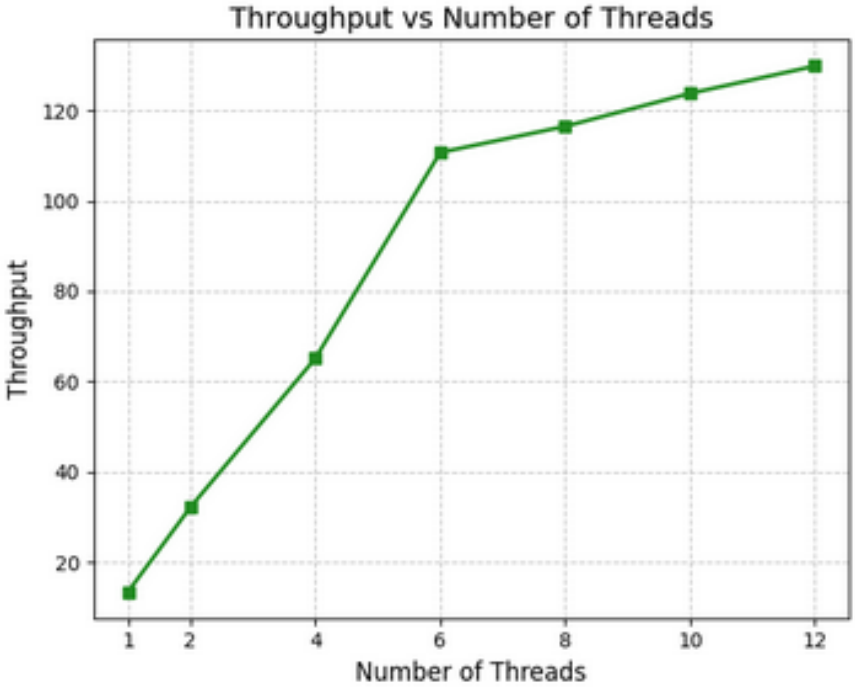
Throughput (The Processing Capacity): Mirroring speedup, the system throughput climbs from a single-threaded baseline (relative 13.42) to a peak of 129.87 at 12 threads. The sharp jump in throughput between 2 and 6 threads confirms that the parallelization strategy effectively utilized the memory hierarchy. However, the gains begin to plateau after 8 threads, suggesting that the system is approaching the limits of its memory bandwidth or thread management capabilities.

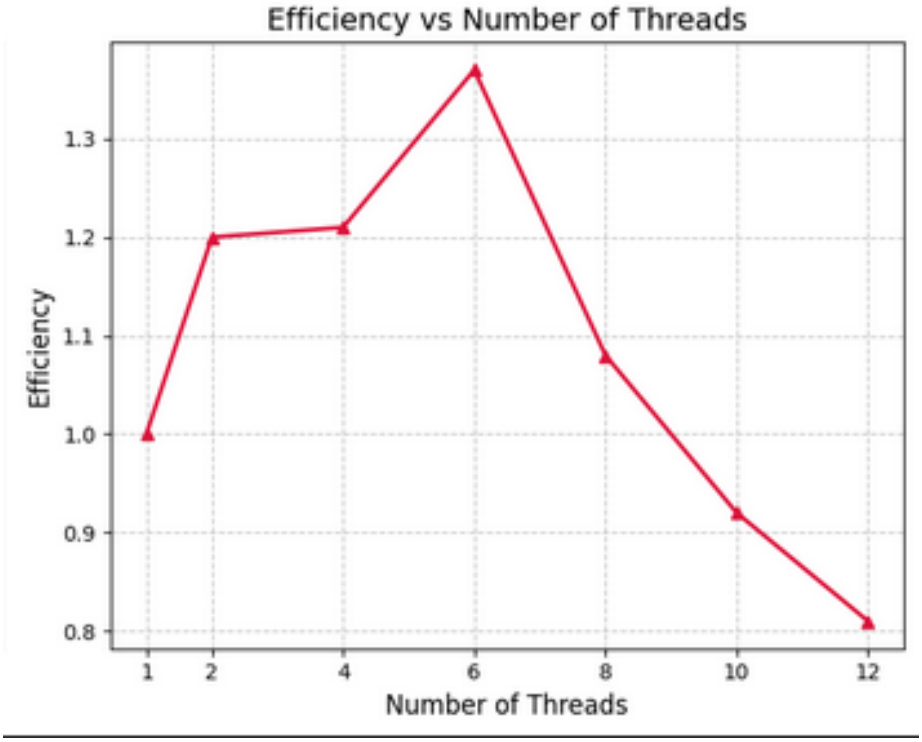
Efficiency (The Resource Utilization): This is the most revealing metric for this specific run. Unlike typical scaling where efficiency immediately degrades, your efficiency actually peaks at 1.37 with 6 threads. This confirms valid cache optimization. However, efficiency drops below 1.0 starting at 10 threads (0.92) and falls to 0.81 at 12 threads. This degradation is driven by:

- **Memory Saturation:** As more threads compete for data, the benefits of cache locality diminish, and the main memory bus becomes a bottleneck.
- **Parallel Overhead:** Managing 10+ threads introduces scheduling and synchronization costs that begin to outweigh the benefits of adding more computational power.

While 12 threads yield the absolute fastest execution time (77.00ms), the data suggests the optimal spot is at 6 threads. At this point, the program delivers massive speedup (8.25x) with peak efficiency (1.37), maximizing both performance and hardware utilization without wasting CPU cycles on overhead.


Speed Up vs Number of Threads
Speed Up (x)
Number of Threads
10
8
6
4
2
1
2
4
6
8
10
12


Throughput vs Number of Threads
Throughput
Number of Threads
120
100
80
60
40
20
1
2
4
6
8
10
12



OBSERVATION FROM PERF STAT:

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

    3,781,291,633      task-clock                       #    3.519 CPUs utilized
              172      context-switches                 #   45.487 /sec
               10      cpu-migrations                   #    2.645 /sec
              367      page-faults                      #   97.057 /sec
   22,681,779,994      instructions                     #    1.54  insn per cycle
   14,681,982,577      cycles                           #    3.883 GHz
    2,118,657,377      branches                         #  560.300 M/sec
          346,784      branch-misses                    #    0.02% of all branches

      1.074610852 seconds time elapsed

      3.759102000 seconds user
      0.023924000 seconds sys
```

Resource Utilization:

- **Parallel Efficiency:** With **3.519 CPUs utilized**, the system is effectively using parallel resources, distributing approximately **3.78 seconds** of task-clock time across cores during the **1.07-second** elapsed physical time.
- **Clock Frequency:** The active cores are running at a high frequency of **3.883 GHz**, indicating the workload is running in a high-performance state (likely boosting) during execution.

Instruction Efficiency:

- **Throughput (IPC):** The system achieves an **IPC (Instructions Per Cycle) of 1.54**. While decent, this is lower than the theoretical maximum (4+ for modern CPUs), suggesting that while the CPU is churning through **22.6 billion instructions**, it is frequently stalling.
- **Branch Prediction:** The branch prediction is phenomenal. With **2.1 billion branches** checked, only **0.02%** (346,784) resulted in misses. This confirms the CPU executes the loop logic almost perfectly without needing to flush the pipeline due to bad guesses.

Primary Bottlenecks (Inferred):

- **Memory Latency (Backend Bound):** Since branch prediction is near-perfect (0.02% miss rate), the CPU is *not* stalling because of logic decisions. The moderate IPC of 1.54 strongly suggests the system is **Backend Bound**—specifically, the execution units are likely waiting on data to arrive from memory (cache misses) to calculate the molecular forces, rather than being limited by calculation speed itself.

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;
}
```

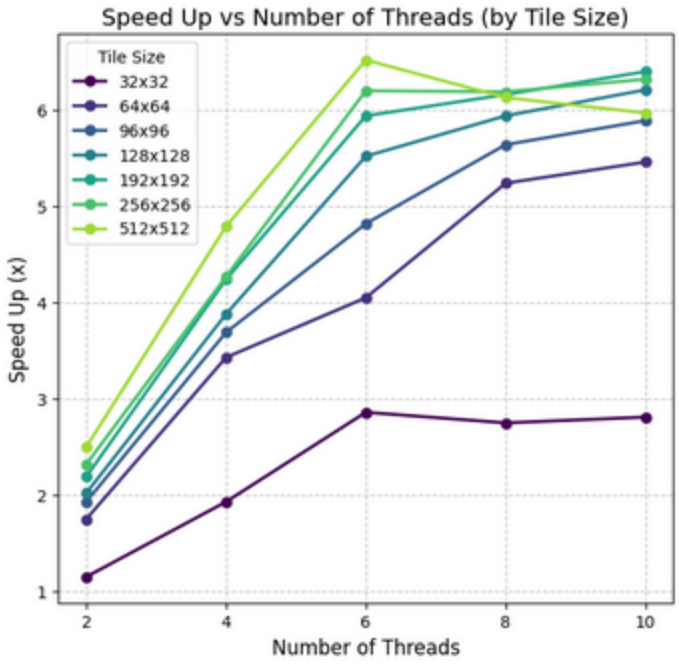
ON EXECUTION:

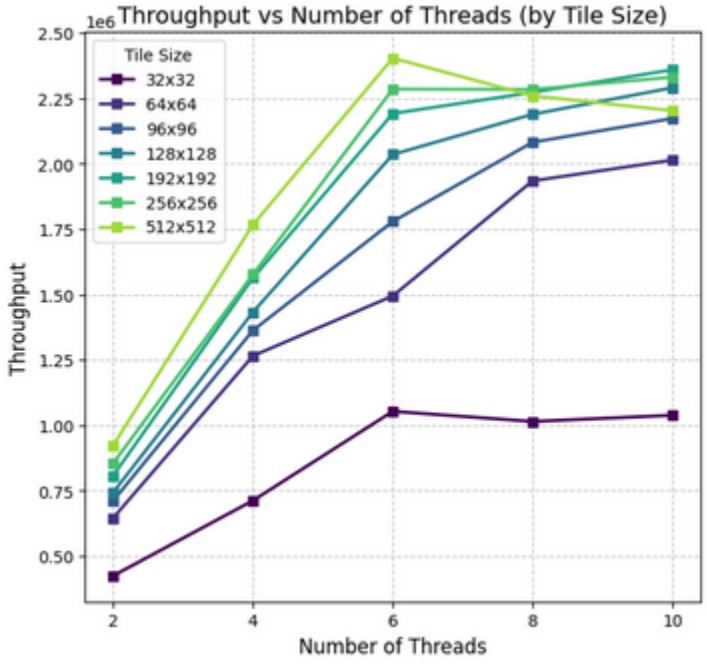
Threads	Tile Size	Execution Time (ms)	Speed Up	Throughput	Efficiency
1 (Baseline)	N/A	271.13	1.00x	-	1.00
2	32 x 32	235.49	1.15x	424,639.81	0.58
2	64 x 64	154.91	1.75x	645,535.46	0.88
2	96 x 96	140.52	1.93x	711,625.82	0.96
2	128 x 128	134.51	2.02x	743,452.70	1.01
2	192 x 192	123.86	2.19x	807,358.44	1.09
2	256 x 256	116.94	2.32x	855,152.52	1.16
2	512 x 512	108.26	2.50x	923,678.53	1.25
4	32 x 32	140.43	1.93x	712,076.58	0.48

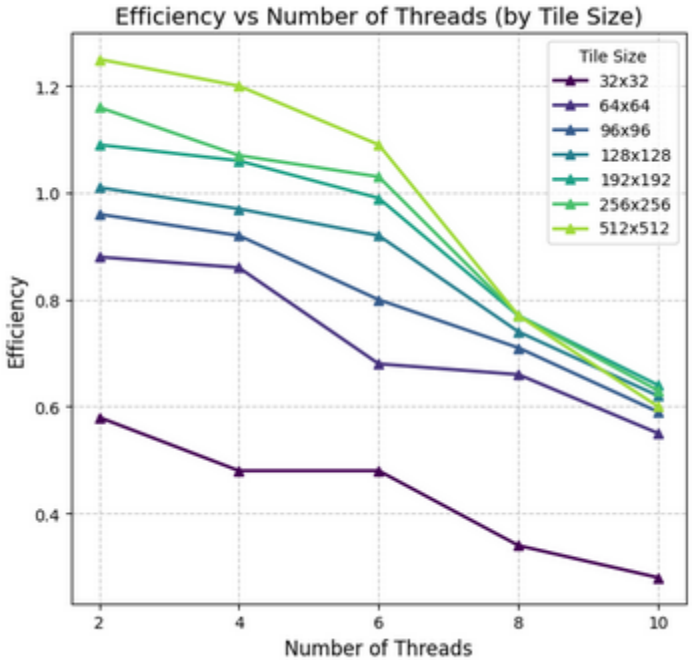
4	64 x 64	79.03	3.43x	1,265,380.05	0.86
4	96 x 96	73.38	3.69x	1,362,726.71	0.92
4	128 x 128	69.81	3.88x	1,432,535.83	0.97
4	192 x 192	63.97	4.24x	1,563,175.34	1.06
4	256 x 256	63.43	4.27x	1,576,617.68	1.07
4	512 x 512	56.60	4.79x	1,766,861.99	1.20
6	32 x 32	94.95	2.86x	1,053,199.56	0.48
6	64 x 64	66.92	4.05x	1,494,345.54	0.68
6	96 x 96	56.21	4.82x	1,778,943.91	0.80

6	128 x 128	49.13	5.52x	2,035,212.97	0.92
6	192 x 192	45.62	5.94x	2,191,813.97	0.99
6	256 x 256	43.76	6.20x	2,285,273.11	1.03
6	512 x 512	41.60	6.52x	2,403,759.94	1.09
8	32 x 32	98.57	2.75x	1,014,492.61	0.34
8	64 x 64	51.69	5.24x	1,934,438.36	0.66
8	96 x 96	48.03	5.64x	2,081,928.55	0.71
8	128 x 128	45.68	5.94x	2,188,989.66	0.74
8	192 x 192	44.00	6.16x	2,272,616.59	0.77

8	256 x 256	43.78	6.19x	2,284,055.15	0.77
8	512 x 512	44.25	6.13x	2,259,697.45	0.77
10	32 x 32	96.32	2.81x	1,038,181.41	0.28
10	64 x 64	49.66	5.46x	2,013,766.67	0.55
10	96 x 96	46.02	5.89x	2,172,733.20	0.59
10	128 x 128	43.66	6.21x	2,290,463.58	0.62
10	192 x 192	42.39	6.40x	2,359,152.19	0.64
10	256 x 256	42.90	6.32x	2,330,884.16	0.63
10	512 x 512	45.39	5.97x	2,202,907.13	0.60


Speed Up vs Number of Threads (by Tile Size)
Tile Size
32x32
64x64
96x96
128x128
192x192
256x256
512x512
Speed Up (x)
Number of Threads
1
2
3
4
5
6
2
4
6
8
10


Throughput vs Number of Threads (by Tile Size)
1e6
Tile Size
32x32
64x64
96x96
128x128
192x192
256x256
512x512
Throughput
Number of Threads
0.50
0.75
1.00
1.25
1.50
1.75
2.00
2.25
2.50
2
4
6
8
10



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:

Tile Size is Critical: The size of the data blocks (tiles) drastically dictates performance by balancing thread synchronization with cache memory.

- **Too Small (32x32):** These perform poorly, peaking at only a **2.86x speedup** (at 6 threads). The threads finish the math so quickly that they spend most of their time idling at the OpenMP barriers, waiting for the next diagonal to start.
- **Optimal Balance (512x512):** This size consistently performs the best, hitting the absolute fastest execution time (**41.60ms**) and peak speedup (**6.52x**) at 6 threads. Large tiles minimize the frequency of barrier synchronization, allowing the CPU to spend more time on actual compute work.
- **Consistency:** Unlike smaller tiles, the larger configurations (256x256 and 512x512) maintain high efficiency (>1.0) up to 6 threads, proving that reducing synchronization points is more valuable here than granular load balancing.

Scaling Limits: The setup scales super-linearly up to **6 threads**, but adding more actively hurts efficiency.

- **Peak Efficiency:** At 6 threads, the system achieves an efficiency of **1.09**, indicating optimal cache usage.

- **Diminishing Returns:** Pushing beyond this to 8 or 10 threads causes efficiency to plummet to **0.77** and **0.60** respectively. The execution time barely improves (and in some cases worsens) as the overhead of managing additional threads outweighs the computational gain.

Hardware Bottlenecks: The sharp drop in efficiency at higher thread counts signals a clear bottleneck:

- **Memory Saturation:** The dynamic programming algorithm constantly needs to fetch neighboring penalty values. With **10 threads** hammering the memory bus, the system likely hits a bandwidth wall.
- **Synchronization Overhead:** The slight regression in speedup at 10 threads (down to 5.97x for 512x512 tiles) confirms that the cost of barrier synchronization has become dominant, stalling cores as they wait for the slowest thread to complete its diagonal block.

OBSERVATION FROM PERF STAT:

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

   14,295,065,526      task-clock                       #    4.267 CPUs utilized

            8,080      context-switches                 #  565.230 /sec

                9      cpu-migrations                   #    0.630 /sec

          117,428      page-faults                      #    8.215 K/sec

   76,000,289,379      instructions                     #    1.52  insn per cycle

   50,030,237,184      cycles                           #    3.500 GHz

    4,345,991,929      branches                         #  304.020 M/sec

       20,320,530      branch-misses                    #    0.47% of all branches


      3.350060617 seconds time elapsed

     14.082569000 seconds user
      0.240573000 seconds sys
```

Resource Utilization:

- **Parallel Efficiency:** With **4.267 CPUs utilized**, the system is moderately parallelized, distributing **14.30 seconds** of task-clock time across roughly 4 processor cores during the **3.35-second** elapsed physical time.
- **Clock Speed:** The workload is running at a consistent **3.500 GHz**, indicating a solid performance state, though slightly lower than the peak turbo frequencies seen in lighter workloads.

Instruction Efficiency:

- **IPC (Instructions Per Cycle):** The system achieves an IPC of **1.52**. This is relatively low for modern hardware (which often targets 2.0+), suggesting that the cores are spending a significant amount of cycles stalled—waiting for resources rather than computing.
- **Branch Prediction:** Contrary to the typical struggles of dynamic programming, branch prediction here is actually quite good. The miss rate is only **0.47%** (20.3 million misses out of 4.3 billion branches). This indicates the CPU is correctly guessing the "maximum" path in the alignment matrix most of the time.

Primary Bottlenecks:

- **Memory & OS Overhead (High Page Faults):** A major visible bottleneck is the **117,428 page faults** (running at over 8,000 per second). This indicates the program is aggressively accessing or initializing large chunks of memory (the DP tables), forcing the operating system to constantly map virtual memory to physical RAM, which halts execution flow.
- **Backend Bound (Inferred):** Since branch prediction is reliable (0.47% misses) but IPC is still low (1.52), the bottleneck is almost certainly **Memory Latency**. The CPU knows *what* to calculate next, but it is waiting on data to be fetched from the large DNA matrices, saturating the cache hierarchy.
- **Context Switching:** With **8,080 context switches** (~565/sec), there is notable overhead from thread scheduling, suggesting potential contention or frequent synchronization points (barriers) in the wavefront algorithm.

Question 3: Scientific Computing - Heat Diffusion Simulation

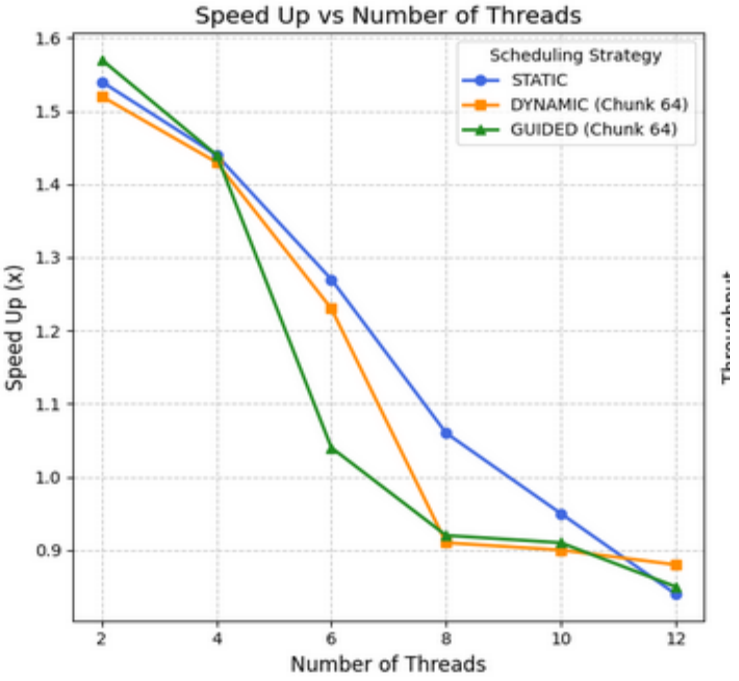
Using Code:

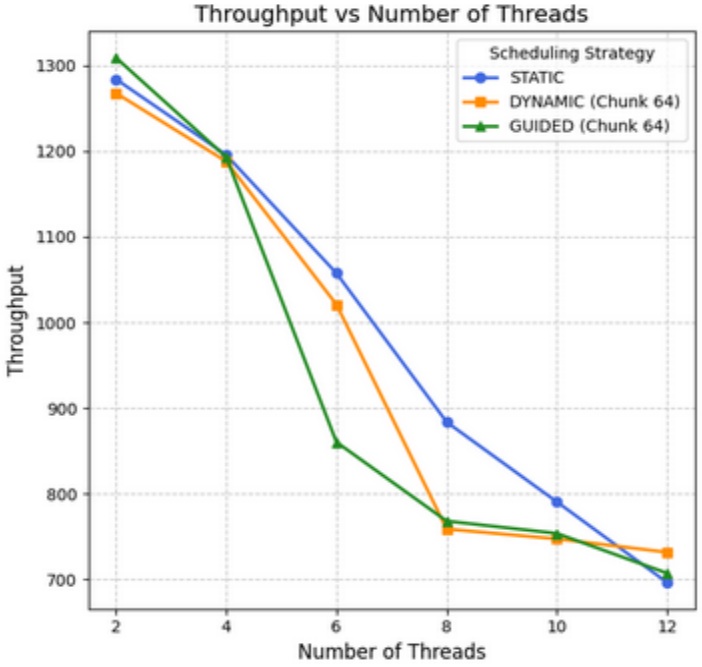
```
auto run_simulation(int num_threads, omp_sched_t sched_type, int chunk_size) -> SimResult {
        std::vector<double> T(N * N, 0.0);
        std::vector<double> T_next(N * N, 0.0);
        int center{N / 2};
        int radius{N / 10};
        #pragma omp parallel for schedule(static) num_threads(num_threads)
        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 };
}
```

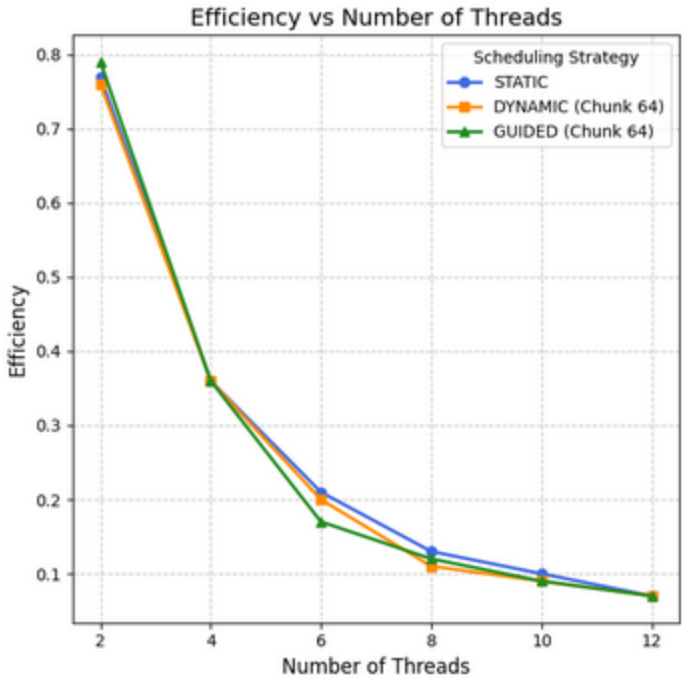
ON EXECUTION:

Threads	Schedule Type	Execution Time (ms)	Speed Up	Throughput	Efficiency
1	N/A (Baseline)	2405.87	1.00x	-	1.00
2	STATIC	1557.66	1.54x	1283.98	0.77
2	DYNAMIC (64)	1577.63	1.52x	1267.72	0.76
2	GUIDED (64)	1527.85	1.57x	1309.03	0.79
4	STATIC	1672.87	1.44x	1195.55	0.36
4	DYNAMIC (64)	1683.56	1.43x	1187.96	0.36
4	GUIDED (64)	1676.30	1.44x	1193.10	0.36
6	STATIC	1891.17	1.27x	1057.55	0.21
6	DYNAMIC (64)	1958.80	1.23x	1021.03	0.20
6	GUIDED (64)	2324.38	1.04x	860.44	0.17

8	STATIC	2262.93	1.06x	883.81	0.13
8	DYNAMIC (64)	2635.18	0.91x	758.96	0.11
8	GUIDED (64)	2603.21	0.92x	768.28	0.12
10	STATIC	2529.53	0.95x	790.66	0.10
10	DYNAMIC (64)	2675.50	0.90x	747.52	0.09
10	GUIDED (64)	2652.90	0.91x	753.89	0.09
12	STATIC	2871.29	0.84x	696.55	0.07
12	DYNAMIC (64)	2732.67	0.88x	731.89	0.07
12	GUIDED (64)	2827.16	0.85x	707.42	0.07


Speed Up vs Number of Threads
Scheduling Strategy
STATIC
DYNAMIC (Chunk 64)
GUIDED (Chunk 64)
Speed Up (x)
Number of Threads
1.6
1.5
1.4
1.3
1.2
1.1
1.0
0.9
2
4
6
8
10
12


Throughput vs Number of Threads
Scheduling Strategy
STATIC
DYNAMIC (Chunk 64)
GUIDED (Chunk 64)
Throughput
Number of Threads
1300
1200
1100
1000
900
800
700
2
4
6
8
10
12



INFERENCE:

Based on the execution data for the finite difference method, we observe a significantly restricted performance profile compared to computational-heavy tasks. The most critical takeaway is how quickly the memory bottleneck saturates, neutralizing the benefits of advanced OpenMP scheduling strategies beyond very low thread counts.

1. **The Overall Performance Ceiling (Severe Memory Bottleneck)**

   Unlike typical parallel tasks where speedup scales with thread count, this heat diffusion simulation hits a **hard brick wall almost immediately**. Across all strategies, the maximum speedup peaks at just **1.57x** with only **2 threads** (Guided Scheduling).

   Why it happens: This indicates the algorithm is severely memory-bound. Calculating the next temperature of a grid point requires fetching multiple neighboring points from memory. The CPU's arithmetic units are left idle, starved for data, because the system's memory bandwidth is fully saturated by just two cores. Adding more threads merely increases contention for the same limited memory bus.

2. The Rapid Collapse (4+ Threads)

If you look at the data for **4 threads and beyond**, there is a massive performance collapse across the board.

The Drop-off: Execution time jumps from ~1527ms (2 threads) to ~1676ms (4 threads) and continues to degrade.

Negative Scaling: By **8 threads**, the parallel execution is actually *slower* than the single-threaded baseline (Speedup < 1.0). For example, **dynamic** at 8 threads takes **2635.18ms**, which is slower than the **2405.87ms** baseline. This confirms that the overhead of managing threads and the intense memory contention completely outweighs any computational gain from extra cores.

3. The Scheduling Comparison (Guided vs. Static)

While all schedules suffer from the memory bottleneck, **Guided Scheduling** proved to be the optimal choice for the peak performance window.

Guided (Chunk 64): This achieved the absolute fastest time of **1527.85ms** at 2 threads. Its ability to start with larger chunks and progressively shrink them likely helped balance the initial workload better than Static or Dynamic.

Static Resilience: Interestingly, as the thread count increased to **6 and 8 threads**, **static** scheduling actually degraded *less* severely than Dynamic or Guided. At 8 threads, Static was ~340ms faster than Guided. This suggests that when memory is choked, the simpler overhead of Static scheduling is preferable to the runtime management required by Guided or Dynamic.

4. Plunging Efficiency

Because the memory bus is completely saturated by just 2 threads, adding more cores results in wasted energy. By **12 threads**, parallel efficiency collapses to a dismal **0.07**. The system is spending massive amounts of energy (and time managing threads) for absolutely no performance gain—in fact, it is actively hurting performance by over-saturating the memory controller.

OBSERVATION FROM PERF STAT:

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

  253,943,972,716      task-clock                       #    6.958 CPUs utilized

           24,977      context-switches                 #   98.356 /sec

              454      cpu-migrations                   #    1.788 /sec

           31,431      page-faults                      #  123.771 /sec

  120,394,475,337      instructions                     #    0.23  insn per cycle

  513,505,172,998      cycles                           #    2.022 GHz

   10,971,720,749      branches                         #   43.205 M/sec

       27,676,655      branch-misses                    #    0.25% of all branches


     36.498427403 seconds time elapsed

    253.296076000 seconds user
      0.517317000 seconds sys
```

Resource Utilization:

- **Parallel Efficiency:** With **6.958 CPUs utilized**, the system is effectively parallelized, distributing **253.94 seconds** of task-clock time across roughly 7 processor cores during the **36.50-second** elapsed physical time.
- **Clock Frequency:** The workload is running at an average of **2.022 GHz**. This relatively low frequency (compared to the 3-4 GHz seen in other tests) strongly suggests the cores are spending a significant amount of time idling or stalling while waiting for data, rather than boosting to their maximum computational potential.

Instruction Efficiency:

- **Catastrophic IPC:** The system shows a dismal **IPC (Instructions Per Cycle) of 0.23**. This is the defining characteristic of this heat simulation. It means that for every CPU clock cycle, less than a quarter of an instruction is completed. The execution units are effectively paralyzed.
- **Branch Prediction:** Unlike the low IPC, the branch prediction is stellar. Branch misses are practically non-existent at **0.25%** (only ~27 million misses out of nearly 11 billion branches). The finite difference loops are highly regular and predictable; the CPU knows exactly *where* to go next, it just can't get the data to do it.

Primary Bottlenecks:

- **Severe Memory Bottleneck (Backend Bound):** The combination of **0.23 IPC** and **99.75% branch accuracy** proves the application is completely **Memory Bound**. The CPU is not struggling with the math or the logic; it is starved for data. The processors are spending the vast majority of their time stalled, waiting to fetch the 2D grid neighbors from RAM to perform the stencil computation.
- **Context Switching:** With **24,977 context switches** (~98/sec), the overhead is moderate but not the primary cause of the slowdown. The main issue remains the saturation of the memory bandwidth, which prevents the 7 active cores from being fed data fast enough to stay busy.