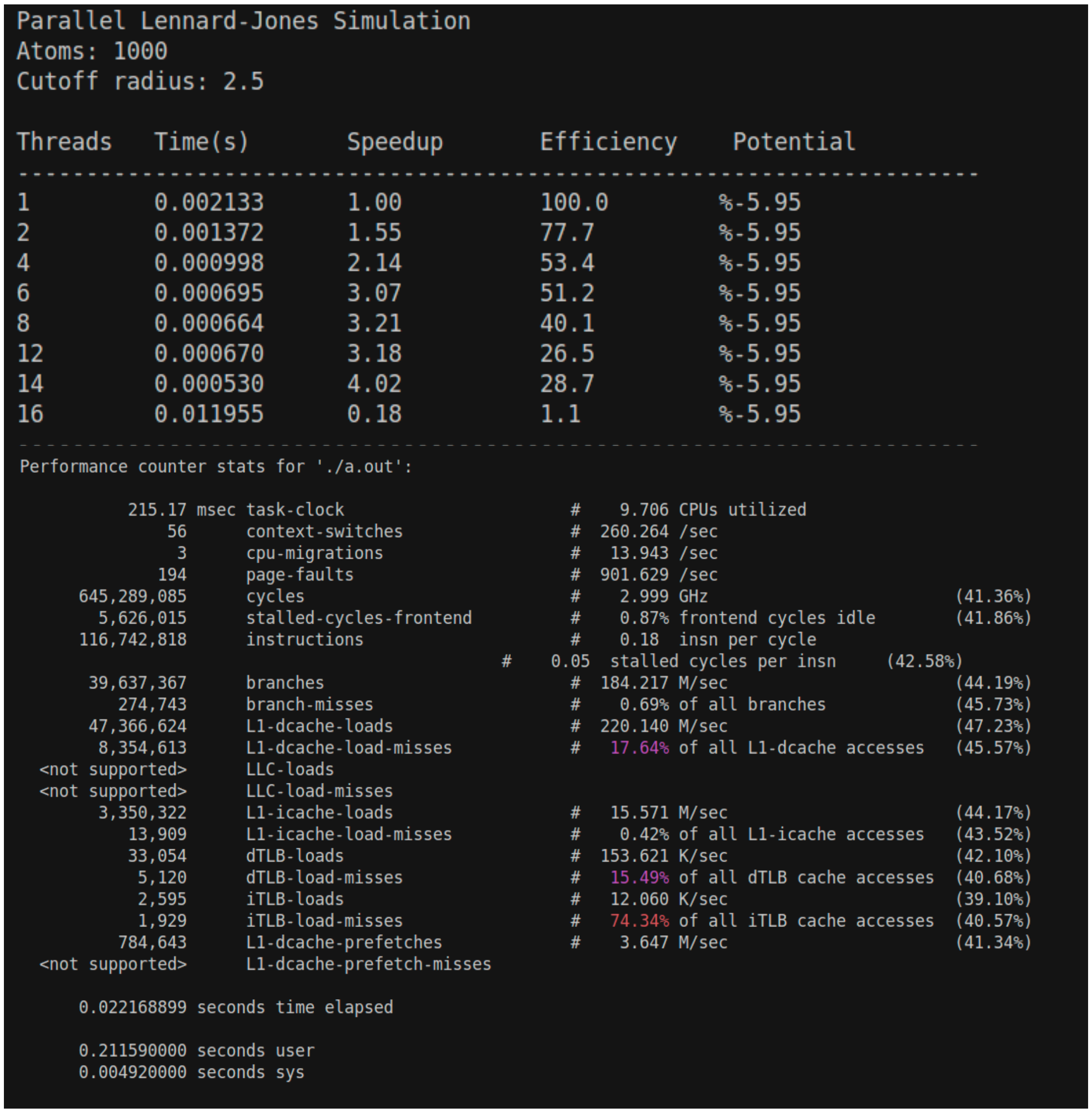
ASSIGNMENT-2
GAURANG GARG [1023031334]

QUESTION-1 :



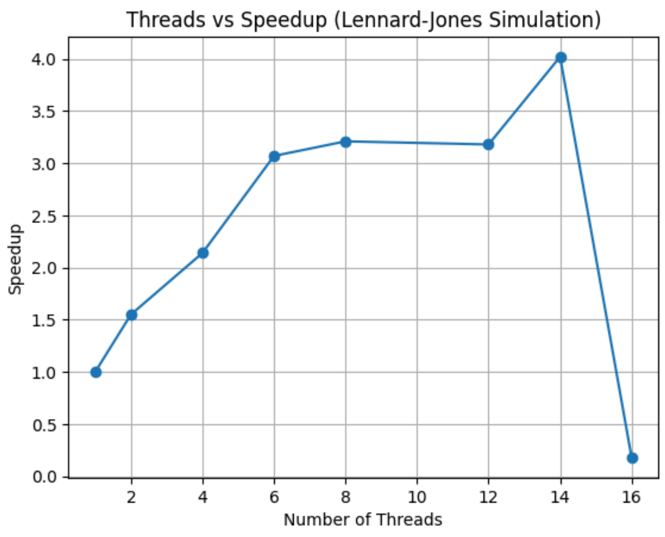
```
Parallel Lennard-Jones Simulation
Atoms: 1000
Cutoff radius: 2.5

Threads    Time(s)       Speedup       Efficiency    Potential
----------------------------------------------------------------------
1          0.002133      1.00          100.0        %-5.95
2          0.001372      1.55          77.7         %-5.95
4          0.000998      2.14          53.4         %-5.95
6          0.000695      3.07          51.2         %-5.95
8          0.000664      3.21          40.1         %-5.95
12         0.000670      3.18          26.5         %-5.95
14         0.000530      4.02          28.7         %-5.95
16         0.011955      0.18          1.1          %-5.95
----------------------------------------------------------------------
Performance counter stats for './a.out':

           215.17 msec task-clock                       #    9.706 CPUs utilized
               56      context-switches                 #  260.264 /sec
                3      cpu-migrations                   #   13.943 /sec
              194      page-faults                      #  901.629 /sec
      645,289,085      cycles                           #    2.999 GHz                      (41.36%)
        5,626,015      stalled-cycles-frontend          #    0.87% frontend cycles idle     (41.86%)
      116,742,818      instructions                     #    0.18  insn per cycle
                                                 #    0.05  stalled cycles per insn     (42.58%)
       39,637,367      branches                         #  184.217 M/sec                    (44.19%)
          274,743      branch-misses                    #    0.69% of all branches          (45.73%)
       47,366,624      L1-dcache-loads                  #  220.140 M/sec                    (47.23%)
        8,354,613      L1-dcache-load-misses            #   17.64% of all L1-dcache accesses (45.57%)
  <not supported>      LLC-loads
  <not supported>      LLC-load-misses
        3,350,322      L1-icache-loads                  #   15.571 M/sec                    (44.17%)
           13,909      L1-icache-load-misses            #    0.42% of all L1-icache accesses (43.52%)
           33,054      dTLB-loads                       #  153.621 K/sec                    (42.10%)
            5,120      dTLB-load-misses                 #   15.49% of all dTLB cache accesses (40.68%)
            2,595      iTLB-loads                       #   12.060 K/sec                    (39.10%)
            1,929      iTLB-load-misses                 #   74.34% of all iTLB cache accesses (40.57%)
          784,643      L1-dcache-prefetches             #    3.647 M/sec                    (41.34%)
  <not supported>      L1-dcache-prefetch-misses

      0.022168899 seconds time elapsed

      0.211590000 seconds user
      0.004920000 seconds sys
```



Observations :

Starting with 1 thread (baseline time: 2.13ms), the program gets faster as we add more threads but only up to a point. With 2-6 threads, we see decent improvements: 6 threads runs about 3x faster with 51% efficiency. This is pretty good.

However, things stop improving after that. Using 8 or 12 threads barely runs any faster than 6 threads, and the efficiency drops to around 26-40%. The worst case is 16 threads, which actually makes the program *slower* than running on a single thread it takes 12ms instead of 2ms. This is a complete failure and probably happens because we're asking for more threads than the CPU can actually handle efficiently.

Why Isn't It Scaling Better?

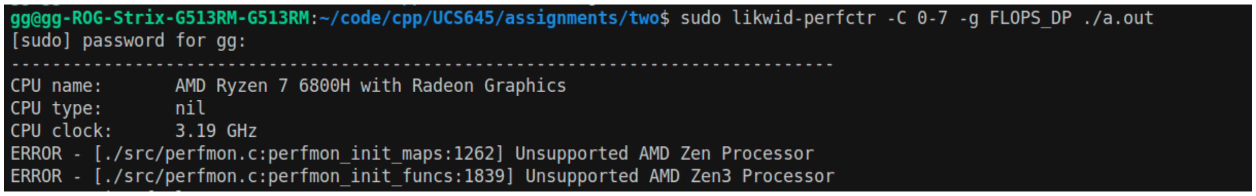
1.Memory bottleneck: The code spends about 5.5 CPU cycles for every instruction it executes. This means the processor is sitting idle most of the time, waiting for data from memory.

2.Cache misses: About 18% of memory accesses miss the fast L1 cache and have to go to slower memory.

3.Low CPU utilization: Even when we ask for 16 threads, only about 10 CPUs are actually doing useful work.

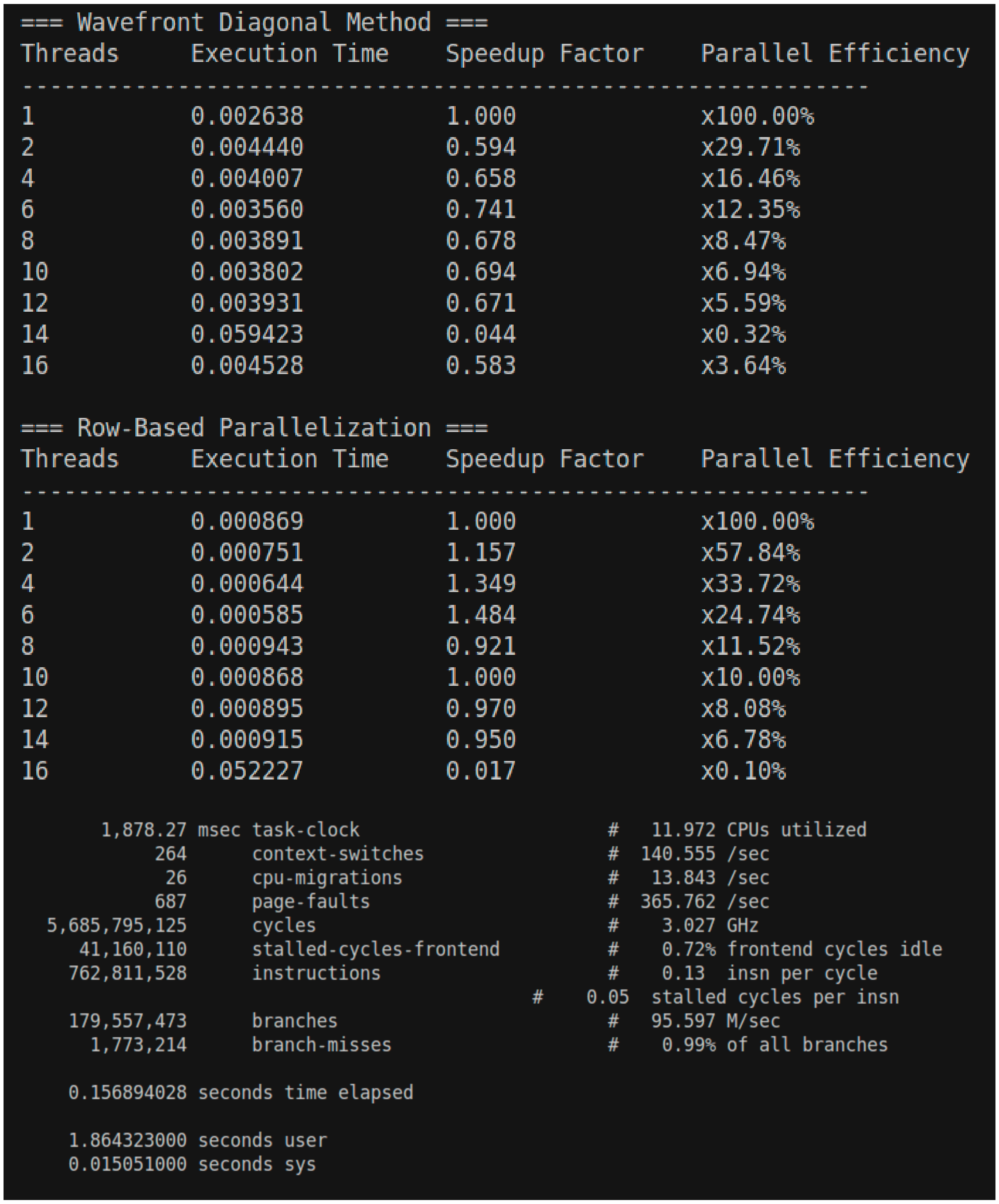
The code works correctly-the energy values stay consistent, which means the physics is right. But it doesn't scale well beyond 6 threads because the CPU spends too much time waiting for memory rather than doing calculations. The O(N^2) algorithm means every atom checks every other atom, which creates a lot of random memory access patterns that kill performance.

Note : likwid command is running on my machine due to some incompatibility with the cpu.



```
gg@gg-ROG-Strix-G513RM-G513RM:~/code/cpp/UCS645/assignments/two$ sudo likwid-perfctr -C 0-7 -g FLOPS_DP ./a.out
[sudo] password for gg:
--------------------------------------------------------------------------------
CPU name:       AMD Ryzen 7 6800H with Radeon Graphics
CPU type:       nil
CPU clock:      3.19 GHz
ERROR - [./src/perfmon.c:perfmon_init_maps:1262] Unsupported AMD Zen Processor
ERROR - [./src/perfmon.c:perfmon_init_funcs:1839] Unsupported AMD Zen3 Processor
```

QUESTION-3 :



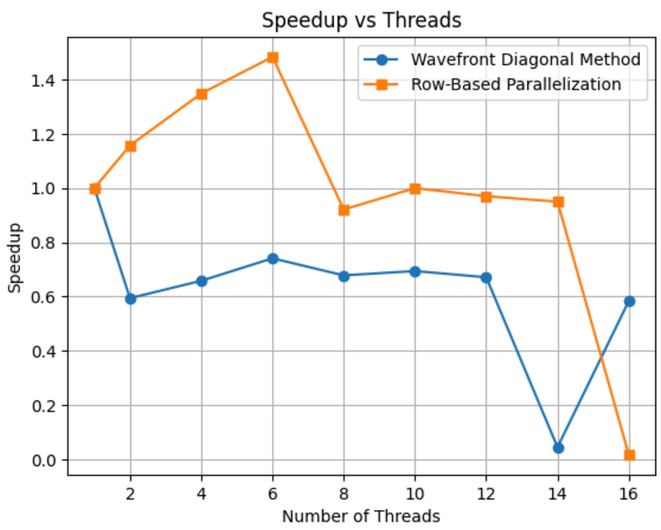
```
=== Wavefront Diagonal Method ===
Threads      Execution Time     Speedup Factor     Parallel Efficiency
------------------------------------------------------------
1            0.002638           1.000              x100.00%
2            0.004440           0.594              x29.71%
4            0.004007           0.658              x16.46%
6            0.003560           0.741              x12.35%
8            0.003891           0.678              x8.47%
10           0.003802           0.694              x6.94%
12           0.003931           0.671              x5.59%
14           0.059423           0.044              x0.32%
16           0.004528           0.583              x3.64%

=== Row-Based Parallelization ===
Threads      Execution Time     Speedup Factor     Parallel Efficiency
------------------------------------------------------------
1            0.000869           1.000              x100.00%
2            0.000751           1.157              x57.84%
4            0.000644           1.349              x33.72%
6            0.000585           1.484              x24.74%
8            0.000943           0.921              x11.52%
10           0.000868           1.000              x10.00%
12           0.000895           0.970              x8.08%
14           0.000915           0.950              x6.78%
16           0.052227           0.017              x0.10%

       1,878.27 msec task-clock                #   11.972 CPUs utilized
            264      context-switches          #  140.555 /sec
             26      cpu-migrations            #   13.843 /sec
            687      page-faults               #  365.762 /sec
  5,685,795,125      cycles                    #    3.027 GHz
     41,160,110      stalled-cycles-frontend   #    0.72% frontend cycles idle
    762,811,528      instructions              #    0.13  insn per cycle
                                        #    0.05  stalled cycles per insn
    179,557,473      branches                  #   95.597 M/sec
      1,773,214      branch-misses             #    0.99% of all branches

    0.156894028 seconds time elapsed

    1.864323000 seconds user
    0.015051000 seconds sys
```



Observation :

Neither approach shows good parallel performance, but for different reasons.

Wavefront Diagonal Method actually gets *slower* with more threads. The single-threaded version runs in 2.64ms, but adding threads increases the time to around 3.5-4.5ms. The efficiency drops to just 6-12% with multiple threads. Thread 14 shows a catastrophic failure (59ms), and even 16 threads doesn't help (4.5ms, only 3.6% efficiency). This method is clearly not benefiting from parallelization at all.

Row-Based Parallelization performs much better overall. The single-threaded version is already 3x faster (0.87ms vs 2.64ms), showing it's a more efficient algorithm. It shows modest improvements up to 6 threads (1.48x speedup, 25% efficiency), which is decent. However, beyond that performance degrades, and at 16 threads it completely crashes (52ms-60x slower than baseline).

Why Isn't It Scaling?

1.Very low IPC (Instructions Per Cycle): Only 0.13, meaning the CPU executes just 1 instruction for every 8 cycles. This is extremely poor and indicates the processor is mostly waiting around doing nothing.

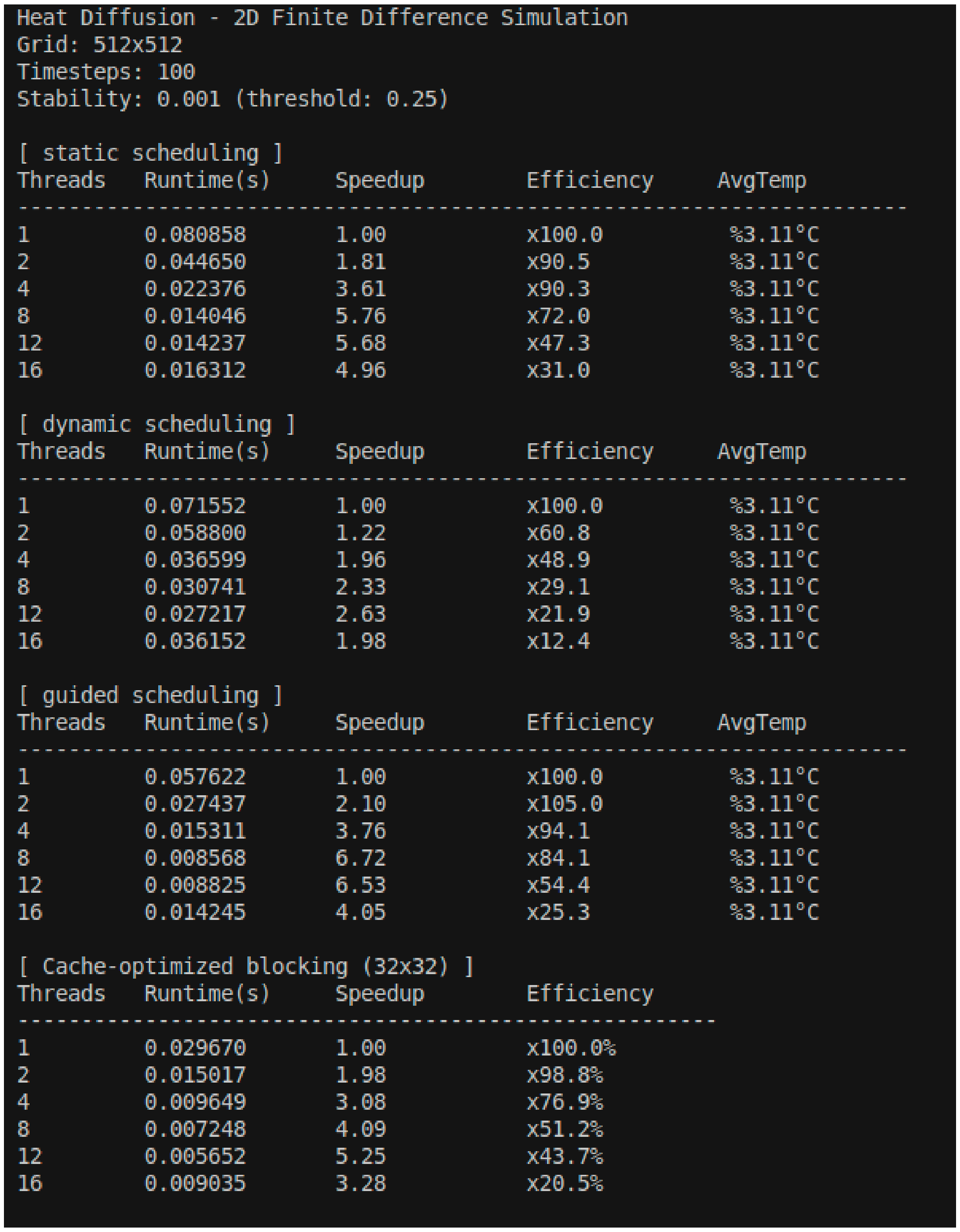
2.Memory-bound computation: The algorithm requires frequent access to the scoring matrix, and each cell depends on three previous cells. This creates memory access patterns that don't parallelize well.

3.Thread coordination overhead: With small sequences (500x500), the work per thread is tiny. The overhead of creating threads, synchronizing them, and managing parallel regions costs more than the actual computation.

3.Data dependencies: In the wavefront method, cells along a diagonal can be computed in parallel, but you must wait for the entire previous diagonal to finish. In the row-based method, each row depends on the previous row, limiting parallelism.

Both methods struggle with parallelization. The wavefront method has excessive synchronization overhead waiting after each diagonal, while the row-based method is inherently sequential (row-by-row) with insufficient work per row. The core issue is problem size is only 250,000 cells isn't enough work to justify parallel overhead. DNA alignment would need sequences of thousands of nucleotides, or parallel processing of multiple sequence pairs simultaneously, to see real speedup benefits. The catastrophic failures at 14-16 threads indicate system oversubscription beyond available hardware capacity.

QUESTION-3 :



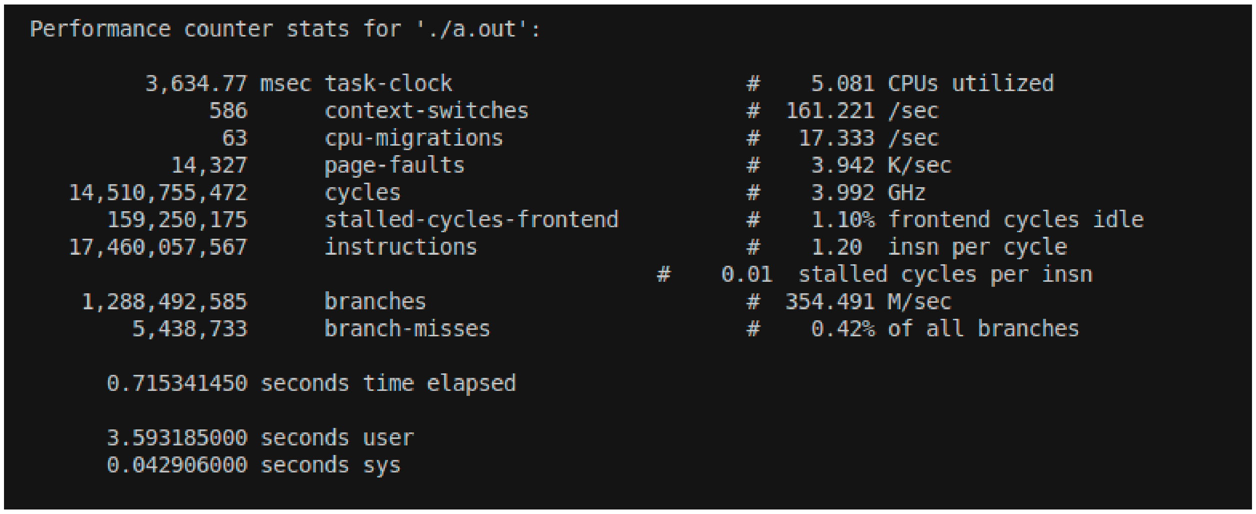
```
Heat Diffusion - 2D Finite Difference Simulation
Grid: 512x512
Timesteps: 100
Stability: 0.001 (threshold: 0.25)

[ static scheduling ]
Threads   Runtime(s)     Speedup        Efficiency     AvgTemp
----------------------------------------------------------------------
1         0.080858       1.00           x100.0          %3.11°C
2         0.044650       1.81           x90.5           %3.11°C
4         0.022376       3.61           x90.3           %3.11°C
8         0.014046       5.76           x72.0           %3.11°C
12        0.014237       5.68           x47.3           %3.11°C
16        0.016312       4.96           x31.0           %3.11°C

[ dynamic scheduling ]
Threads   Runtime(s)     Speedup        Efficiency     AvgTemp
----------------------------------------------------------------------
1         0.071552       1.00           x100.0          %3.11°C
2         0.058800       1.22           x60.8           %3.11°C
4         0.036599       1.96           x48.9           %3.11°C
8         0.030741       2.33           x29.1           %3.11°C
12        0.027217       2.63           x21.9           %3.11°C
16        0.036152       1.98           x12.4           %3.11°C

[ guided scheduling ]
Threads   Runtime(s)     Speedup        Efficiency     AvgTemp
----------------------------------------------------------------------
1         0.057622       1.00           x100.0          %3.11°C
2         0.027437       2.10           x105.0          %3.11°C
4         0.015311       3.76           x94.1           %3.11°C
8         0.008568       6.72           x84.1           %3.11°C
12        0.008825       6.53           x54.4           %3.11°C
16        0.014245       4.05           x25.3           %3.11°C

[ Cache-optimized blocking (32x32) ]
Threads   Runtime(s)     Speedup        Efficiency
-------------------------------------------------------
1         0.029670       1.00           x100.0%
2         0.015017       1.98           x98.8%
4         0.009649       3.08           x76.9%
8         0.007248       4.09           x51.2%
12        0.005652       5.25           x43.7%
16        0.009035       3.28           x20.5%
```

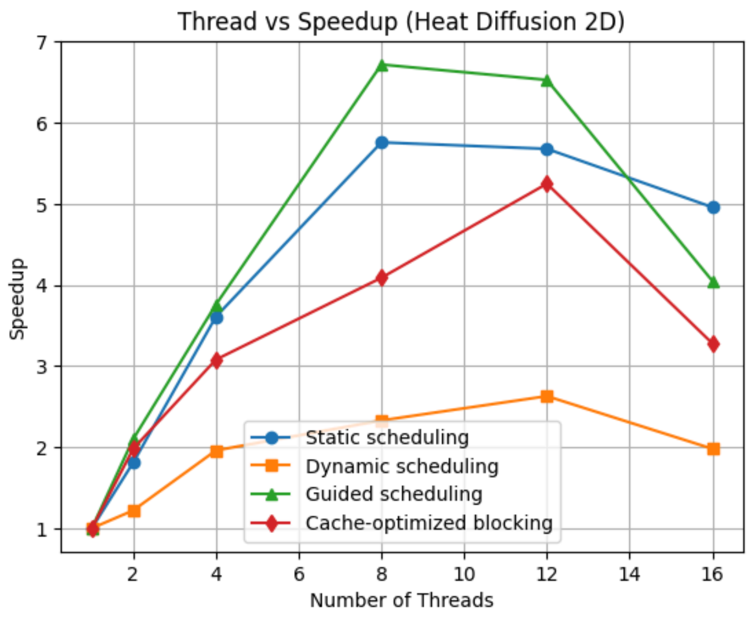


```
 Performance counter stats for './a.out':

          3,634.77 msec task-clock                       #    5.081 CPUs utilized
               586      context-switches                 #  161.221 /sec
                63      cpu-migrations                   #   17.333 /sec
            14,327      page-faults                      #    3.942 K/sec
    14,510,755,472      cycles                           #    3.992 GHz
       159,250,175      stalled-cycles-frontend          #    1.10% frontend cycles idle
    17,460,057,567      instructions                     #    1.20  insn per cycle
                                                  #    0.01  stalled cycles per insn
     1,288,492,585      branches                         #  354.491 M/sec
         5,438,733      branch-misses                    #    0.42% of all branches

       0.715341450 seconds time elapsed

       3.593185000 seconds user
       0.042906000 seconds sys
```



Observations :

Unlike the previous two programs, this one actually shows **good parallel performance** with the right approach:

Static Scheduling performs well up to 8 threads (5.76x speedup, 72% efficiency), then starts to plateau. At 12-16 threads, efficiency drops to 31-47% but still maintains reasonable speedup.

Dynamic Scheduling shows poor results across the board. Even with 12 threads, it only achieves 2.63x speedup (22% efficiency). The overhead of dynamically assigning work chunks clearly outweighs any load balancing benefits for this uniform workload.

Guided Scheduling delivers the best performance. It achieves impressive 6.72x speedup on 8 threads with 84% efficiency-nearly linear scaling. This is excellent. Performance remains strong at 12 threads (6.53x speedup, 54% efficiency) before degrading at 16 threads.

Cache-Optimized Blocking shows the most consistent scaling. While it doesn't reach the peak speedup of guided scheduling (max 5.25x at 12 threads), it maintains better efficiency across all thread counts and has the fastest absolute runtime for single-threaded execution (29ms vs 57-80ms for others).

Why Does This Scale Better?

The performance counters reveal why this code parallelizes much better than the previous examples:

1.Good IPC (Instructions Per Cycle): 1.20 is significantly better than the previous programs (0.13-0.18). The CPU is actually doing useful work most of the time.

2.Better memory access patterns: The heat equation has good spatial locality-each cell only needs its four neighbors. The blocking approach further improves cache utilization by processing 32x32 tiles that fit in cache.

3.Low branch miss rate: Only 0.42% branch misses indicates predictable control flow, which helps CPU pipeline efficiency.

4.Abundant parallelism: With a 512x512 grid and 100 iterations, there's enough work (26 million operations) to amortize thread overhead.

5.No data dependencies between cells: Unlike the DNA alignment problem, each grid cell can be computed independently based on the previous time step, allowing genuine parallel execution.

This heat diffusion simulation demonstrates effective parallelization. Guided scheduling achieves near-linear scaling up to 8 threads because it adapts chunk sizes dynamically-starting large to minimize overhead, then shrinking to balance load as work completes. Cache-blocking trades some peak speedup for better memory efficiency and more consistent performance. The key difference from previous programs is that this problem has sufficient computational intensity (good IPC), favorable memory access patterns, and true independence between parallel tasks. The decline at 16 threads still suggests hitting hardware limits, but overall this shows how parallel programming should work when problem characteristics align well with the approach.