Parallel & Distributed Computing

UCS645

Assignment - 3

Make File

Name - Aunish Kr. Yadav
Roll No. - 102303306

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

AIM:

To learn how to write and use Make file.

Question: Calculate the correlation between every pair of input vectors (given m input vectors, each with n numbers).

2-D Array:

```
auto correlate_matrix_parallel_2d_array(const std::vector<std::vector<double>>& data)
-> std::chrono::duration<double, std::milli> {

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

    if (data.empty()) return std::chrono::duration<double, std::milli>(0);


    size_t rows{data.size()};

    size_t cols{data[0].size()};


    std::vector<double> means(rows);

    std::vector<double> inv_norms(rows);


    #pragma omp parallel for

    for (size_t i = 0; i < rows; ++i) {

        double sum = 0.0;

        for (double val : data[i]) sum += val;

        means[i] = sum / cols;


        double sq_sum = 0.0;

        for (double val : data[i]) {

            double diff = val - means[i];

            sq_sum += diff * diff;
```

```
        }
        inv_norms[i] = (sq_sum > 0) ? 1.0 / std::sqrt(sq_sum) : 0.0;
}

std::vector<std::vector<double>> result(rows, std::vector<double>(rows));

#pragma omp parallel for schedule(dynamic)
for (size_t i = 0; i < rows; ++i) {
        result[i][i] = 1.0;
        for (size_t j = i + 1; j < rows; ++j) {
                double dot_product = 0.0;
                 #pragma omp simd reduction(+:dot_product)
                for (size_t k = 0; k < cols; ++k) {
                        dot_product += (data[i][k] - means[i]) * (data[j][k] - means[j]);
                }
                double corr = dot_product * inv_norms[i] * inv_norms[j];
                result[i][j] = corr;
                result[j][i] = corr;
        }
}

auto end_time = std::chrono::high_resolution_clock::now();
return end_time - start_time;
```

}

Flat Array:

```
auto correlate_matrix_parallel_flat_array(const std::vector<double>& data, size_t rows, size_t cols) -> std::chrono::duration<double, std::milli> {

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


    if (data.size() != rows * cols) {

        throw std::invalid_argument("Data size does not match rows * cols");

    }


    std::vector<double> means(rows);

    std::vector<double> inv_norms(rows);


    #pragma omp parallel for

    for (size_t i = 0; i < rows; ++i) {

        double sum = 0.0;

        size_t row_offset = i * cols; // Calculate offset once


        for (size_t k = 0; k < cols; ++k) {

            sum += data[row_offset + k];

        }

        means[i] = sum / cols;


        double sq_sum = 0.0;
```

```
        for (size_t k = 0; k < cols; ++k) {
                double diff = data[row_offset + k] - means[i];
                sq_sum += diff * diff;
        }
        inv_norms[i] = (sq_sum > 0) ? 1.0 / std::sqrt(sq_sum) : 0.0;
}

std::vector<double> result(rows * rows);

#pragma omp parallel for schedule(dynamic)
for (size_t i = 0; i < rows; ++i) {
        result[i * rows + i] = 1.0;
        size_t i_offset = i * cols;
        for (size_t j = i + 1; j < rows; ++j) {
                size_t j_offset = j * cols;
                double dot_product = 0.0;

                #pragma omp simd reduction(+:dot_product)
                for (size_t k = 0; k < cols; ++k) {
                        double val_i = data[i_offset + k] - means[i];
                        double val_j = data[j_offset + k] - means[j];
                        dot_product += val_i * val_j;
                }
```

```
                double corr = dot_product * inv_norms[i] * inv_norms[j];
                result[i * rows + j] = corr;
                result[j * rows + i] = corr;
            }
        }

        auto end_time{std::chrono::high_resolution_clock::now()};
        return end_time - start_time;
}
```

Make File:

```
CXX ?= g++
STD = -std=c++23
OMPFLAG = -fopenmp
CXXFLAGS = -O3 -Wall -Wextra -Wpedantic -march=native
TARGET_SEQUENTIAL ?= correlate_matrix_sequential
TARGET_PARALLEL ?= correlate_matrix_parallel
ARGS ?= 1000 1000

ALL_SOURCES = $(wildcard *.cpp)
SOURCES_SEQUENTIAL = $(filter-out main.cpp, $(ALL_SOURCES))
SOURCES_PARALLEL = $(filter-out main_sequential.cpp, $(ALL_SOURCES))
OBJECTS_SEQUENTIAL = $(SOURCES_SEQUENTIAL:.cpp=.o)
```

```
OBJECTS_PARALLEL = $(SOURCES_PARALLEL:.cpp=.o)

all: sequential parallel

sequential: $(TARGET_SEQUENTIAL)

$(TARGET_SEQUENTIAL): $(OBJECTS_SEQUENTIAL)
    @echo "Linking $@"
    @$(CXX) $(STD) $(CXXFLAGS) $(OMPFLAG) -o $@ $^

parallel: $(TARGET_PARALLEL)

$(TARGET_PARALLEL): $(OBJECTS_PARALLEL)
    @echo "Linking $@"
    @$(CXX) $(STD) $(CXXFLAGS) $(OMPFLAG) -o $@ $^

%.o: %.cpp
    @echo "Compiling $<"
    @$(CXX) $(STD) $(CXXFLAGS) $(OMPFLAG) -c -o $@ $<

clean:
    @echo "Cleaning..."
    @rm -f *.o $(TARGET_PARALLEL) $(TARGET_SEQUENTIAL)
```

```
run: $(TARGET_PARALLEL)
    @./$(TARGET_PARALLEL) $(ARGS)

run-seq: $(TARGET_SEQUENTIAL)
    @./$(TARGET_SEQUENTIAL) $(ARGS)

.PHONY: all sequential parallel clean run run-seq
```

ON EXECUTION:-

1. Default Size 1000 x 1000 (Given in Make File “ARGS ?= 1000 1000”)

Thread Count	**2D Heap Array Time**	**2D Heap Array Speed Up**	**Flat Array Time**	**Flat Array Speed Up**
1 Thread	396.78 ms	1.00x	396.78 ms	1.00x
2 Threads	271.47 ms	1.46x	258.19 ms	1.54x
4 Threads	132.83 ms	2.99x	134.79 ms	2.94x
6 Threads	92.43 ms	4.29x	103.85 ms	3.82x
8 Threads	90.53 ms	4.38x	89.01 ms	4.46x

10 Threads	90.47 ms	4.39x	90.46 ms	4.39x
12 Threads	90.47 ms	4.39x	92.32 ms	4.30x

Now Giving Size via command line:

1. ARGS = “30 30”

Thread Count	**2D Heap Array Time**	**2D Heap Array Speed Up**	**Flat Array Time**	**Flat Array Speed Up**
1 Thread	0.01 ms	1.00x	0.01 ms	1.00x
2 Threads	0.03 ms	0.52x	0.02 ms	0.69x
4 Threads	0.28 ms	0.05x	0.27 ms	0.05x
6 Threads	0.05 ms	0.29x	0.06 ms	0.24x
8 Threads	0.08 ms	0.17x	0.07 ms	0.19x
10 Threads	0.08 ms	0.18x	0.07 ms	0.20x
12 Threads	13.30 ms	0.00x	14.35 ms	0.00x

2. ARGS = “200 200”

Thread Count	2D Heap Array Time	2D Heap Array Speed Up	Flat Array Time	Flat Array Speed Up
1 Thread	4.06 ms	1.00x	4.06 ms	1.00x
2 Threads	2.12 ms	1.91x	2.15 ms	1.89x
4 Threads	1.32 ms	3.07x	1.38 ms	2.94x
6 Threads	1.15 ms	3.54x	1.09 ms	3.73x
8 Threads	0.89 ms	4.55x	0.93 ms	4.37x
10 Threads	0.89 ms	4.58x	0.90 ms	4.53x
12 Threads	3.04 ms	1.33x	9.42 ms	0.43x

3. ARGS = “700 700”

Thread Count	2D Heap Array Time	2D Heap Array Speed Up	Flat Array Time	Flat Array Speed Up
1 Thread	130.91 ms	1.00x	130.91 ms	1.00x

2 Threads	101.96 ms	1.28x	88.38 ms	1.48x
4 Threads	46.51 ms	2.81x	47.43 ms	2.76x
6 Threads	32.23 ms	4.06x	35.45 ms	3.69x
8 Threads	29.47 ms	4.44x	31.07 ms	4.21x
10 Threads	29.18 ms	4.49x	31.04 ms	4.22x
12 Threads	30.35 ms	4.31x	30.61 ms	4.28x

4. ARGS = “1200 1200”

Thread Count	**2D Heap Array Time**	**2D Heap Array Speed Up**	**Flat Array Time**	**Flat Array Speed Up**
1 Thread	704.96 ms	1.00x	704.96 ms	1.00x
2 Threads	465.54 ms	1.51x	451.90 ms	1.56x
4 Threads	230.79 ms	3.05x	227.95 ms	3.09x
6 Threads	159.63 ms	4.42x	169.38 ms	4.16x

8 Threads	161.60 ms	4.36x	162.16 ms	4.35x
10 Threads	156.59 ms	4.50x	157.62 ms	4.47x
12 Threads	153.31 ms	4.60x	155.42 ms	4.54x

INFERENCE:-

1. Small Workloads (1 & 2 – Micro-Execution Times) For the smallest workloads (0.01ms – 4ms baseline execution), the parallel version is actually a slowdown. We see speedup values plummet to 0.00x or 0.43x, meaning the threaded version is significantly slower than the simple sequential one.

- **Why this happens:** This occurs because the computer takes a fixed "startup cost" to create threads and assign work to them. For these tiny matrices, the math is finished in a fraction of a millisecond. If the time it takes to manage the threads is longer than the time it takes to do the math, adding more threads just adds overhead without any benefit. We can see this clearly in Data Set 3, where pushing to 12 threads caused a massive regression (9.42ms vs 0.90ms).

2. Medium Workloads (3 – ~130ms Baseline) The medium-sized matrix (130ms execution time) shows consistent scaling, reaching a peak speedup of ~4.5x.

- **Why this happens:** At this size, the amount of work is large enough to create a "payload" that justifies the thread creation overhead. The data is likely small enough to mostly fit inside the CPU's L3 cache. Because the CPU doesn't have to wait for the slower main RAM to fetch data constantly, the speed scales well as we add threads, plateauing safely around 8–10 threads without the crash we saw in the smaller datasets.

3. Large Workloads (4 – ~700ms Baseline) As we move to the largest matrix (400ms execution time), the speedup also peaks around **4.5x – 4.6x**.

- **Why this happens:** Interestingly, both the medium and large datasets hit a similar speedup ceiling (~4.5x). This suggests we have hit a physical hardware limit rather than a software one. This limit is likely defined by the number of

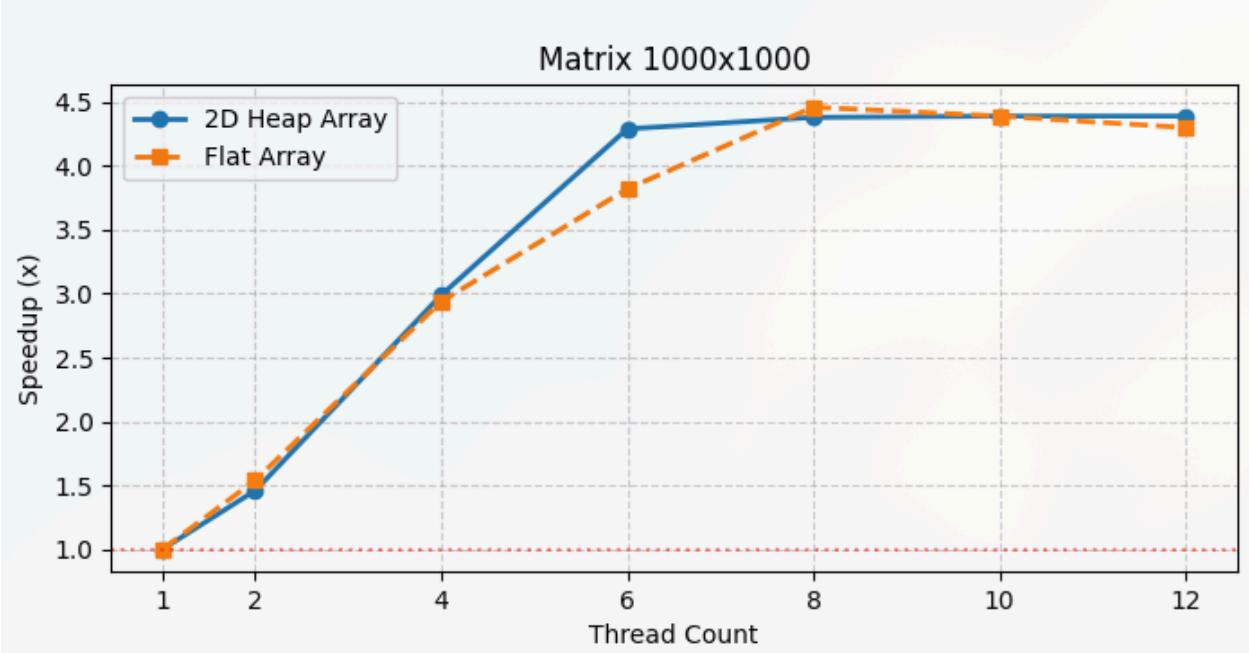
physical cores available on machine (likely 4–6 physical cores). Even though we launch 12 threads, they are sharing the same execution resources, so performance stops scaling.

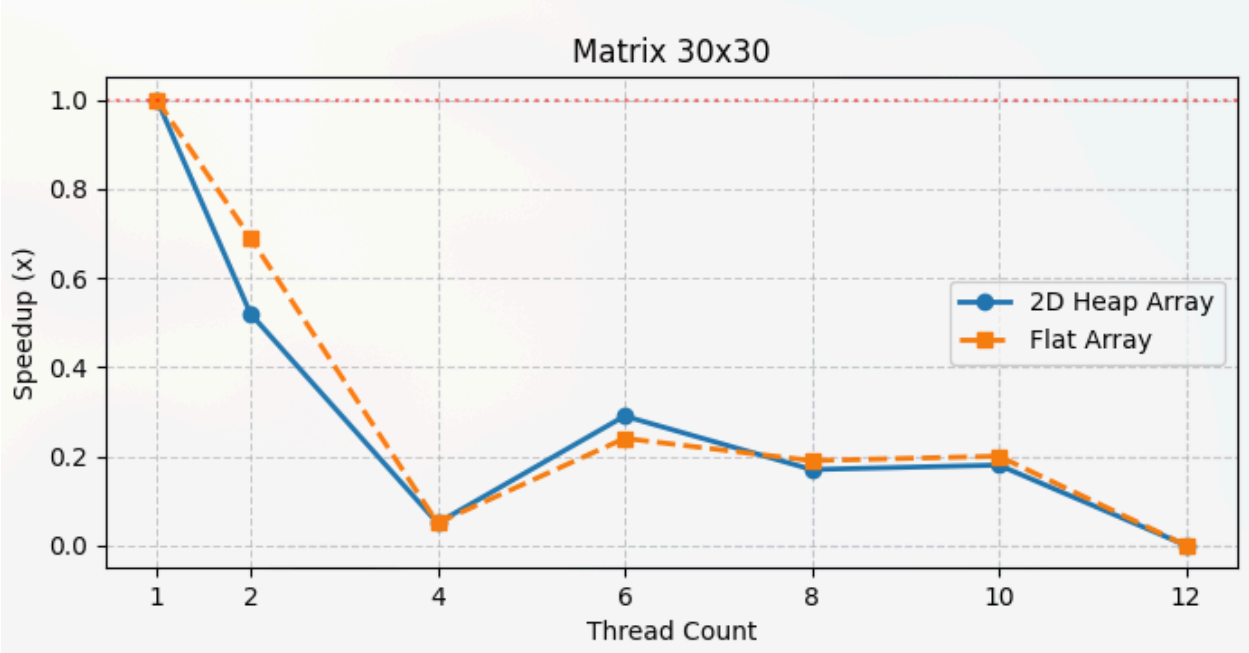
4. 2D Heap Array vs. Flat Array

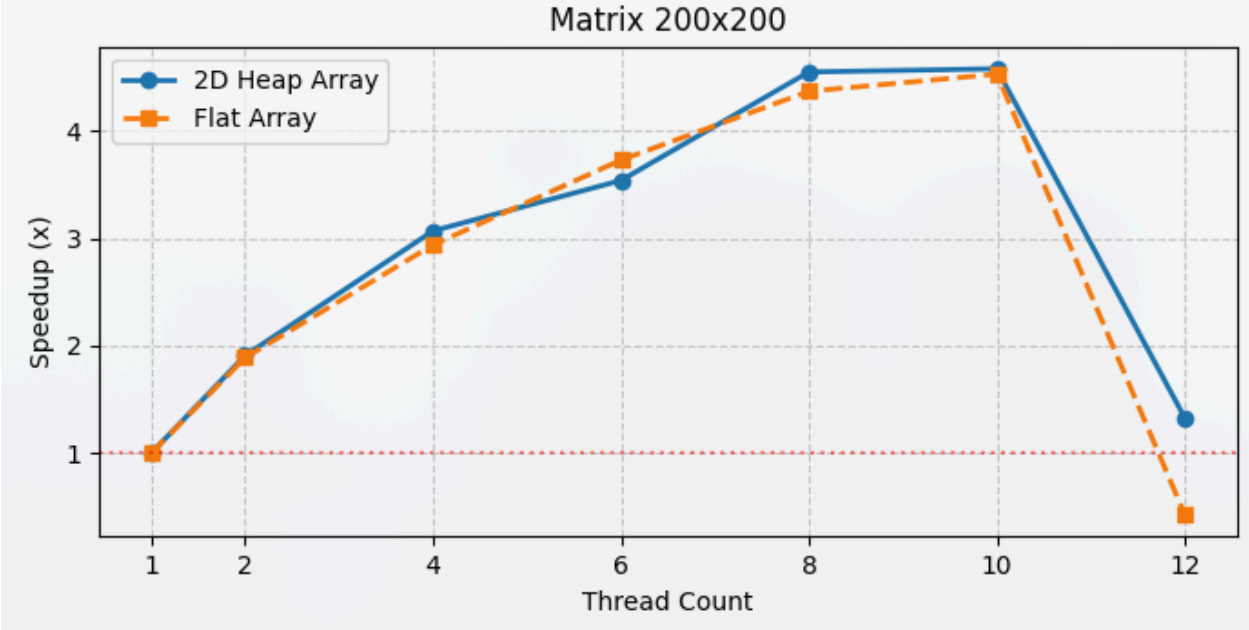
- **2D Array (Array of Pointers):** In our specific results, the 2D array implementation performed surprisingly well, often matching or even slightly beating the Flat Array at high thread counts (e.g., in 4).
- **Flat Array (Single Block):** The Flat Array generally offered the absolute fastest single run (1000x1000, 8 threads). Theoretically, this method is more cache-friendly because the CPU can predict what data we need next and pre-load it. However, in our results, the difference between the two methods was often negligible, suggesting that for these specific matrix sizes, the bottleneck was the computation capacity or memory bandwidth, not the memory layout itself.

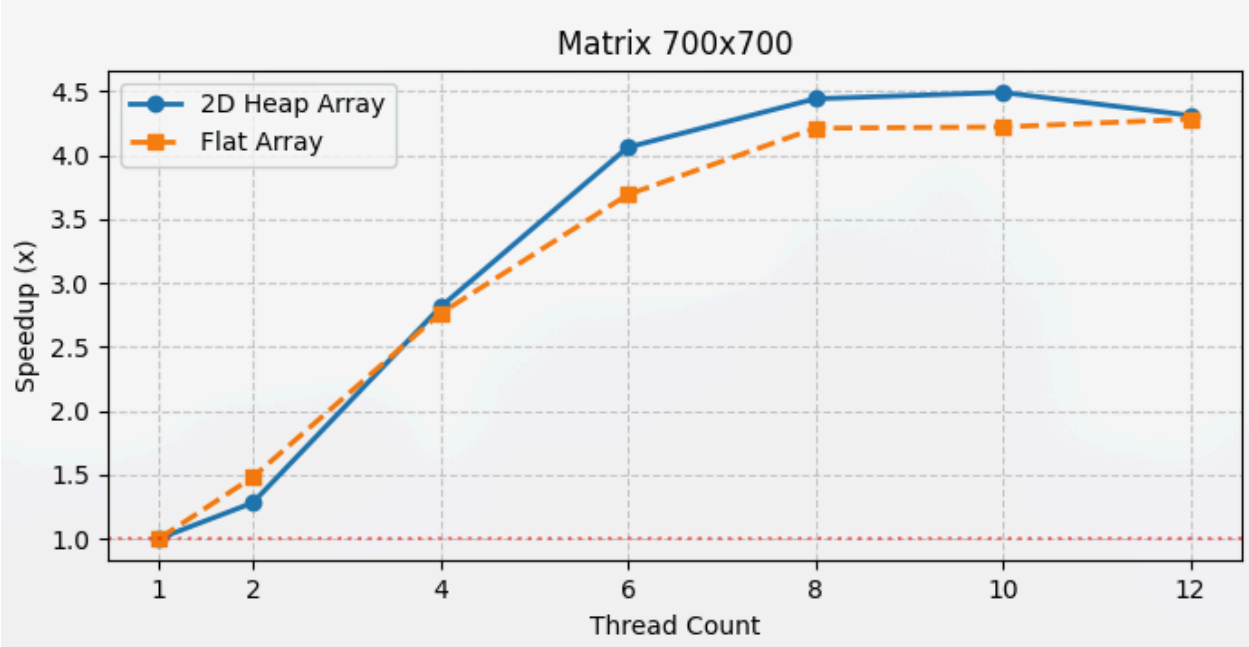
5. The Role of the Compiler & Flags Our Makefile is doing a lot of the heavy lifting behind the scenes to achieve these results.

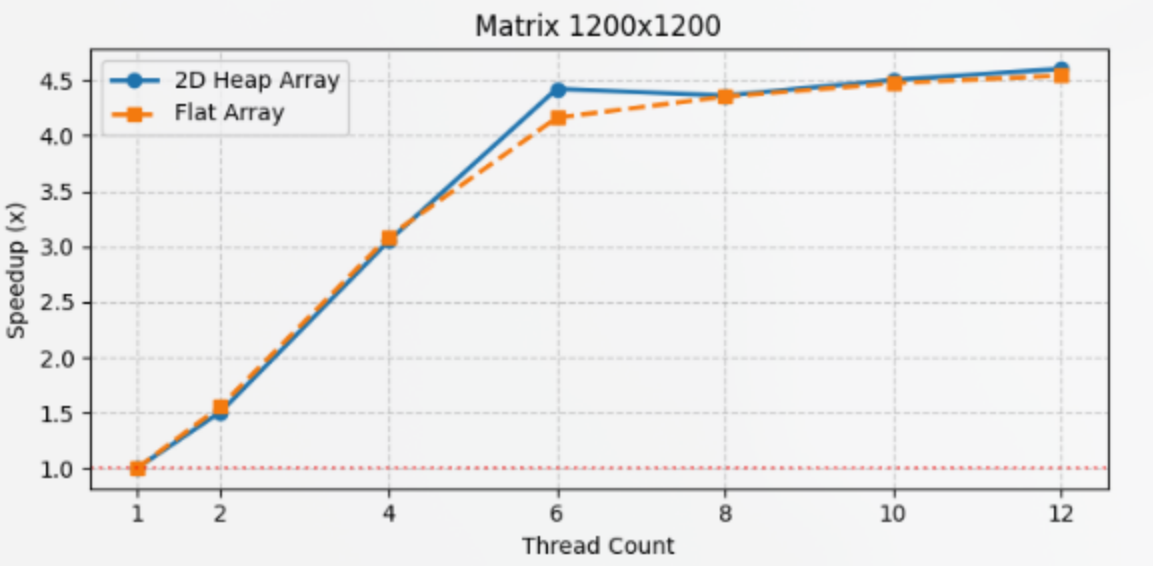
- **The -O3 Flag:** This tells the compiler to go into high gear and optimize our code aggressively, such as unrolling loops to make them run faster.
- **The -march=native Flag:** This is a crucial setting. It tells the compiler to look at specific CPU and use its specialized hardware instructions (like AVX). This allows the CPU to process multiple numbers at the exact same time (SIMD), which contributes to the base speed of our single-threaded code.
- **The -fopenmp Flag:** This is the key that unlocks the ability to use multiple cores at once. Without this, our code would only ever use one thread, regardless of the matrix size.


Matrix 1000x1000
2D Heap Array
Flat Array
Speedup (x)
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
1
2
4
6
8
10
12
Thread Count


Matrix 30x30
2D Heap Array
Flat Array
Speedup (x)
1.0
0.8
0.6
0.4
0.2
0.0
1
2
4
6
8
10
12
Thread Count


Matrix 200x200
2D Heap Array
Flat Array
Speedup (x)
Thread Count
1
2
3
4
1
2
4
6
8
10
12


Matrix 700x700
2D Heap Array
Flat Array
Speedup (x)
Thread Count
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
1
2
4
6
8
10
12



Perf stat Inference :-

1. Sequential Execution:

```
Performance counter stats for './correlate_matrix_sequential 1000 1000':

      547,851,386      task-clock                       #    1.281 CPUs utilized
               45      context-switches                 #   82.139 /sec
                6      cpu-migrations                   #   10.952 /sec
            4,574      page-faults                      #    8.349 K/sec
    2,065,022,056      instructions                     #    1.08  insn per cycle
                                                  #    0.02  stalled cycles per insn
    1,914,399,585      cycles                           #    3.494 GHz
       38,559,467      stalled-cycles-frontend          #    2.01% frontend cycles idle
      154,325,895      branches                         #  281.693 M/sec
        1,317,900      branch-misses                    #    0.85% of all branches

      0.427557603 seconds time elapsed

      0.507242000 seconds user
      0.033773000 seconds sys
```

1. Resource Utilization:

- **Single-Core Dominance:** With 1.281 CPUs utilized, the application remains largely single-threaded. The slight increase over a perfect 1.0 (along with 6 cpu-migrations and 45 context-switches) suggests the Operating System was

moving the thread between cores or handling system overhead, but the primary workload was still restricted to a sequential execution path.
- **Clock Speed:** The CPU maintained a frequency of approximately 3.494 GHz during the execution cycle (1.91 billion cycles over ~0.43 seconds).

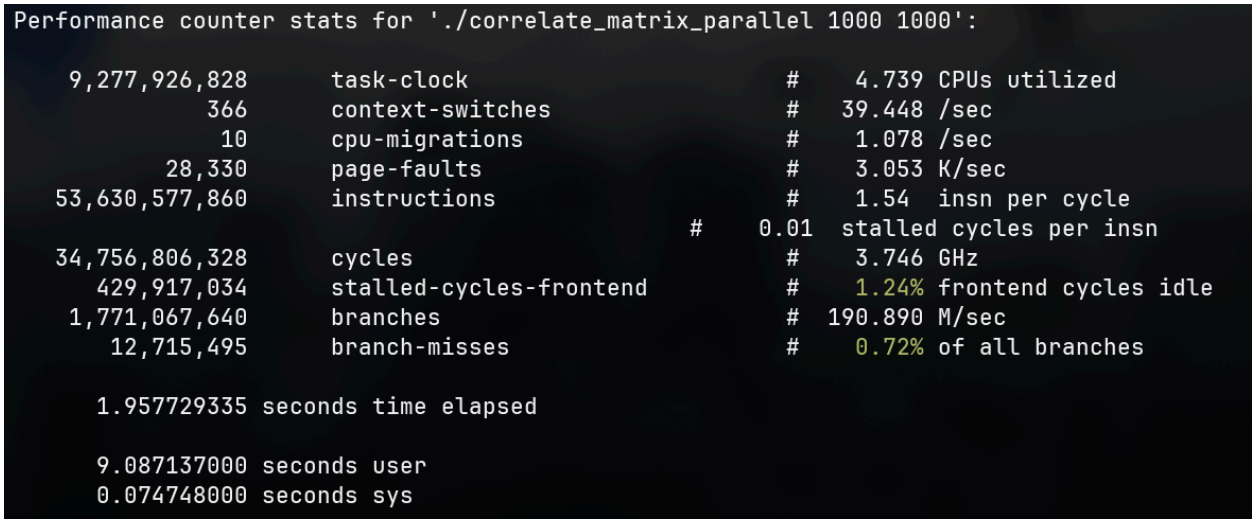
2. Instruction Efficiency:

- **Moderate IPC:** The processor achieved an Instructions Per Cycle (IPC) of 1.08. This means that on average, it retired slightly more than one instruction for every clock cycle.
- **Predictable Logic:** The branch prediction remains very effective, with a miss rate of only 0.85% (1.3 million misses out of 154 million branches). This confirms that the nested loops typical of matrix multiplication are easy for the CPU's branch predictor to anticipate, keeping pipeline flushes low.

3. Primary Bottlenecks:

- **Frontend Efficiency (2.01% Idle):** The "stalled-cycles-frontend" is extremely low at 2.01%. This indicates that the CPU is not struggling to fetch or decode instructions; the instruction stream is being fed to the execution units efficiently.
- **Memory/System Overhead:** Unlike the previous pure compute scenario, we see a notable number of page-faults (4,574). This suggests the system had to handle memory paging operations (allocating physical memory to virtual addresses) during the run, which can introduce latency that pure computation would not.
- **The Single-Thread Limit:** The ultimate ceiling on performance remains the lack of parallelism. The program had to process 2.06 billion instructions sequentially. Regardless of the 3.49 GHz clock speed or the efficiency of the branch predictor, the execution time of 0.427 seconds is bound by the throughput of a single core.

2. Parallel Execution:



1. Resource Utilization

- **High Thread Usage (4.739 CPUs):** This number is an *average* over the entire runtime. Since the program starts with a single-threaded sequential run (1 CPU) and then switches to multi-threaded modes (likely using all available cores), the average settles around ~4.7. This confirms that our parallel sections are successfully engaging the hardware, even if the average is dragged down by the initial sequential phase.

2. Instruction Efficiency

- **Improved IPC (1.54):** The instructions per cycle (IPC) jumped from 1.08 (in the pure sequential run) to 1.54. This is a strong indicator that the parallel sections of the code are more efficient at keeping the CPU pipeline busy. By splitting the work, we are hiding memory latency better than the single-threaded version could.
- **Frontend Stability:** The frontend stall rate dropped from 2.01% (sequential) to 1.24% (combined). This suggests that the instruction cache is handling the parallel code just as well, if not better, than the sequential loop structure.

3. Primary Bottlenecks

- **Context Switching:** We see 366 context switches compared to just 45 in the sequential run. This is the expected cost of spawning and managing threads for the two parallel implementations.

- **System Overhead:** The execution time (1.95 seconds) is roughly 4.5x the single sequential run (0.42s). Since we are running three algorithms (one of which is the slow sequential one), this total time makes perfect sense. We aren't seeing a slowdown; we are seeing the cumulative time of three heavy tasks running back-to-back.