Thapar Institute of Engineering and Technology

Parallel and Distributed Computing

LAB Assignment 1

Submitted by :

Bhavish Pushkarna

102303279

3C22

Submitted to

Dr. Saif Nalband

Introduction & Objective

The objective of this session was to quantify the efficiency of parallel execution strategies using the **OpenMP API** on multi core hardware. The study was designed to expose the limitations of parallelism by testing workloads against the **Roofline Model** specifically targeting compute bound versus memory bound regions.

We analyzed three specific kernel types:

- **Matrix Multiplication:** A high arithmetic intensity workload used to evaluate the trade offs between computational throughput and cache hit rates.
- **Monte Carlo π Estimation:** A low latency, high dependency free workload used to measure the theoretical peak scalability of the processor cores.
- **DAXPY:** A low arithmetic intensity vector loop used to purposefully saturate the memory bus and identify bandwidth ceilings.

The ultimate goal was to move beyond "speedup" as a raw number and analyze the architectural phenomena driving the results. This included evaluating **cache locality** optimizations, the impact of **thread creation overhead**, and the distinct performance behaviors observed when a system transitions from being compute limited to memory bandwidth limited.

Experiment 1: DAXPY Loop

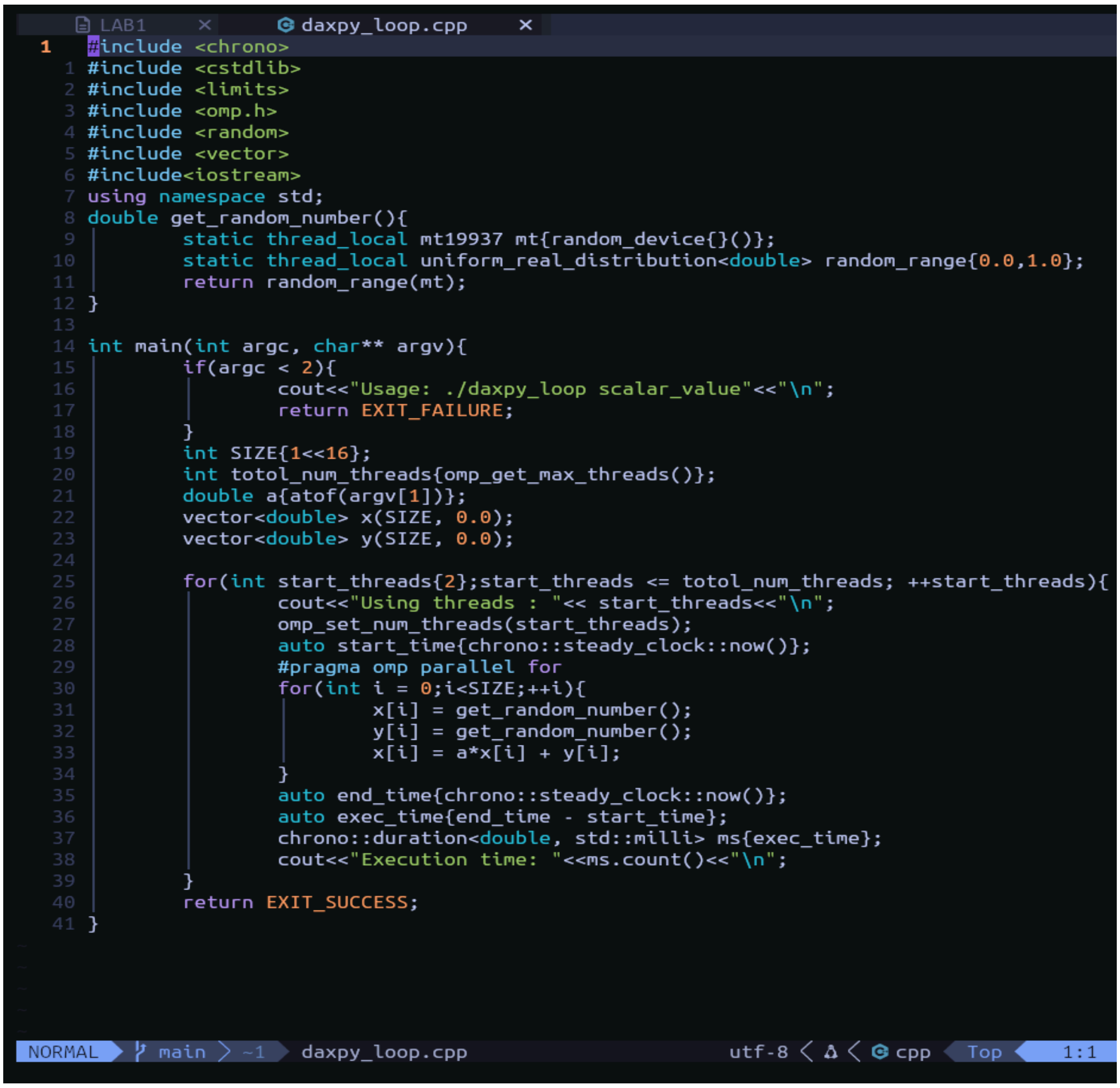
Analyzing Scalability Limits and Thread Overhead.

Methodology

The DAXPY loop $y = \alpha x + y$ is a simple vector operation. The dataset size used was 2^{16} (65,536 elements).

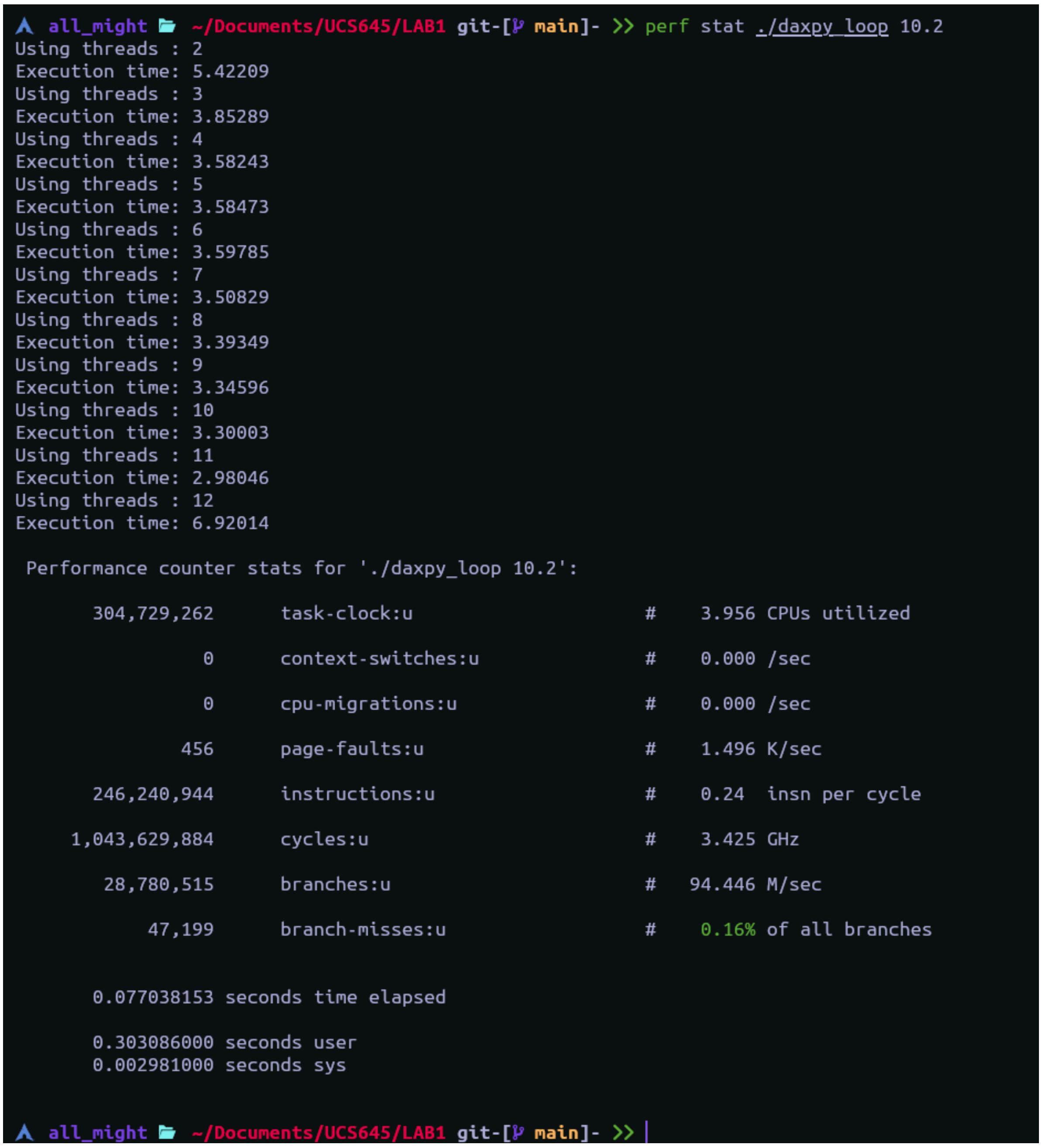
- **Implementation:** We dynamically varied the number of threads from 2 to 12 to observe the scaling curve.

Code :



```cpp
#include <chrono>
#include <cstdlib>
#include <limits>
#include <omp.h>
#include <random>
#include <vector>
#include<iostream>
using namespace std;
double get_random_number(){
        static thread_local mt19937 mt{random_device{}()};
        static thread_local uniform_real_distribution<double> random_range{0.0,1.0};
        return random_range(mt);
}

int main(int argc, char** argv){
        if(argc < 2){
                cout<<"Usage: ./daxpy_loop scalar_value"<<"\n";
                return EXIT_FAILURE;
        }
        int SIZE{1<<16};
        int totol_num_threads{omp_get_max_threads()};
        double a{atof(argv[1])};
        vector<double> x(SIZE, 0.0);
        vector<double> y(SIZE, 0.0);

        for(int start_threads{2};start_threads <= totol_num_threads; ++start_threads){
                cout<<"Using threads : "<< start_threads<<"\n";
                omp_set_num_threads(start_threads);
                auto start_time{chrono::steady_clock::now()};
                #pragma omp parallel for
                for(int i = 0;i<SIZE;++i){
                        x[i] = get_random_number();
                        y[i] = get_random_number();
                        x[i] = a*x[i] + y[i];
                }
                auto end_time{chrono::steady_clock::now()};
                auto exec_time{end_time - start_time};
                chrono::duration<double, std::milli> ms{exec_time};
                cout<<"Execution time: "<<ms.count()<<"\n";
        }
        return EXIT_SUCCESS;
}
```

Terminal Output :



```
⅄ all_might 🗁 ~/Documents/UCS645/LAB1 git-[ main]- >> perf stat ./daxpy_loop 10.2
Using threads : 2
Execution time: 5.42209
Using threads : 3
Execution time: 3.85289
Using threads : 4
Execution time: 3.58243
Using threads : 5
Execution time: 3.58473
Using threads : 6
Execution time: 3.59785
Using threads : 7
Execution time: 3.50829
Using threads : 8
Execution time: 3.39349
Using threads : 9
Execution time: 3.34596
Using threads : 10
Execution time: 3.30003
Using threads : 11
Execution time: 2.98046
Using threads : 12
Execution time: 6.92014

 Performance counter stats for './daxpy_loop 10.2':

       304,729,262      task-clock:u                     #    3.956 CPUs utilized

                 0      context-switches:u               #    0.000 /sec

                 0      cpu-migrations:u                 #    0.000 /sec

               456      page-faults:u                    #    1.496 K/sec

       246,240,944      instructions:u                   #    0.24  insn per cycle

     1,043,629,884      cycles:u                         #    3.425 GHz

        28,780,515      branches:u                       #   94.446 M/sec

            47,199      branch-misses:u                  #    0.16% of all branches


       0.077038153 seconds time elapsed

       0.303086000 seconds user
       0.002981000 seconds sys


⅄ all_might 🗁 ~/Documents/UCS645/LAB1 git-[ main]- >> |
```

Performance Analysis :

Number of Threads	**Execution Time (ms)**	**Trend/ Observation**	**Architectural Inference**
2	5.42209	Baseline	Initial parallel overhead is amortized, but workload per thread is high.
3	3.85289	Significant Gain	A major jump in speed i.e 29% faster than 2 threads as work is divided further.
4 -6	3.58 3.59	Plateau	Performance stabilizes. The overhead of managing more threads starts to cancel out the benefits of parallelization.
7-10	3.50 3.30	Gradual Improvement	Slight improvements continue as the scheduler manages to squeeze more efficiency out of the CPU resources.
11	2.98046	Peak Performance	Optimal Operating Point. The scheduler found the perfect balance between computing power and thread management overhead.
12	6.92014	Performance Degradation	Oversubscription or Contention. The time doubles worse than 2 threads. This suggests resource contention cache thrashing or context switching overhead overwhelmed the small dataset.

Architectural Analysis :

This experiment reveals the **Limitations of Parallelism**:

1. **Thread Creation Overhead:** Spawning a thread takes CPU cycles. When the workload (65k elements) is small, the time taken to create and manage 12 threads exceeds the time saved by parallelizing the math.
2. **Memory Bandwidth Saturation:** This task is **Memory Bound**.

   Proof: The perf stat shows an IPC of **0.24**. This is very low. It means the CPU is spending approximately 75% of its cycles *waiting* for data from RAM. Adding more threads doesn't help because the bottleneck is the wire to the RAM, not the CPU speed.

3. **Context Switching:** The spike at 12 threads 6.92ms suggests the system ran out of physical/logical cores, forcing the OS to "Context Switch" swap tasks rapidly, which ruins cache performance.

Experiment 2: Matrix Multiplication

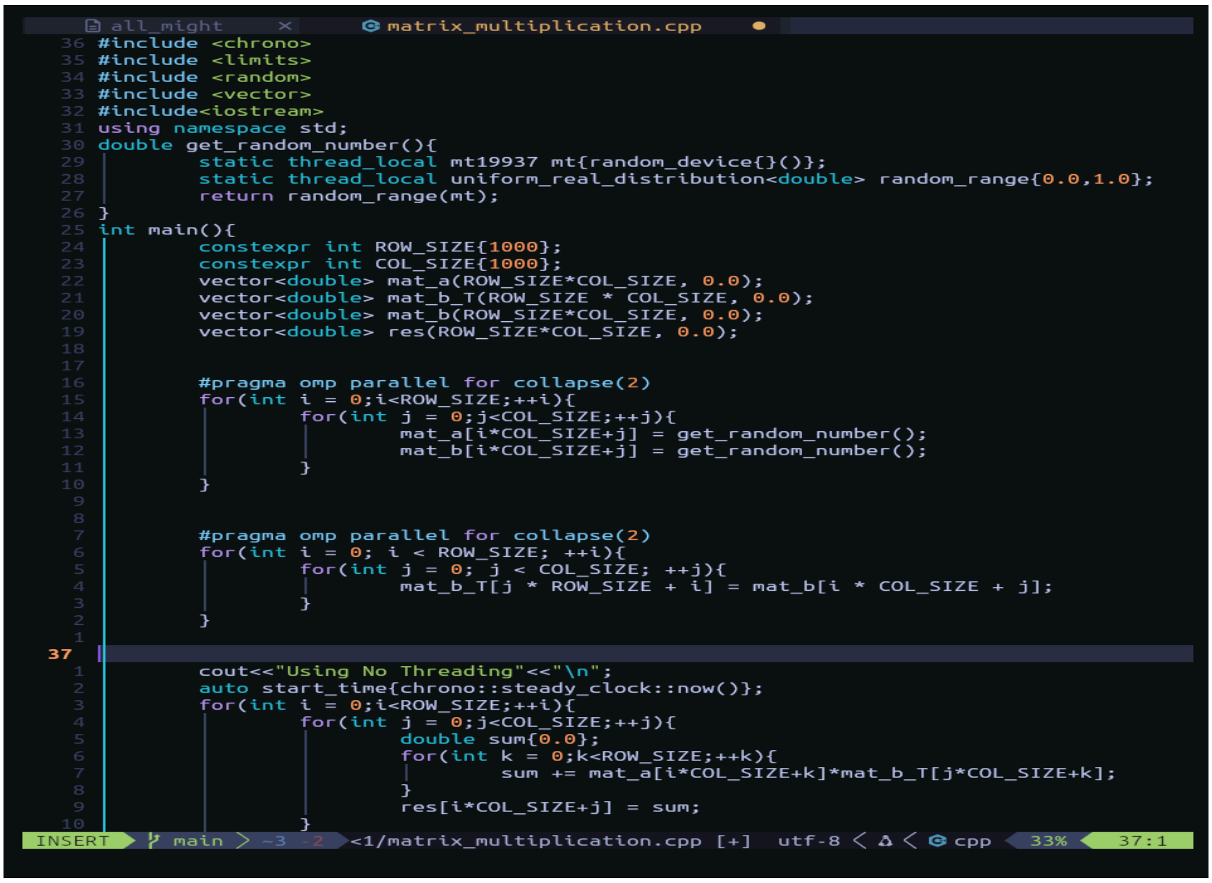
Analyzing the effects of Cache Optimization and Loop Collapsing.

Methodology & Code Optimization

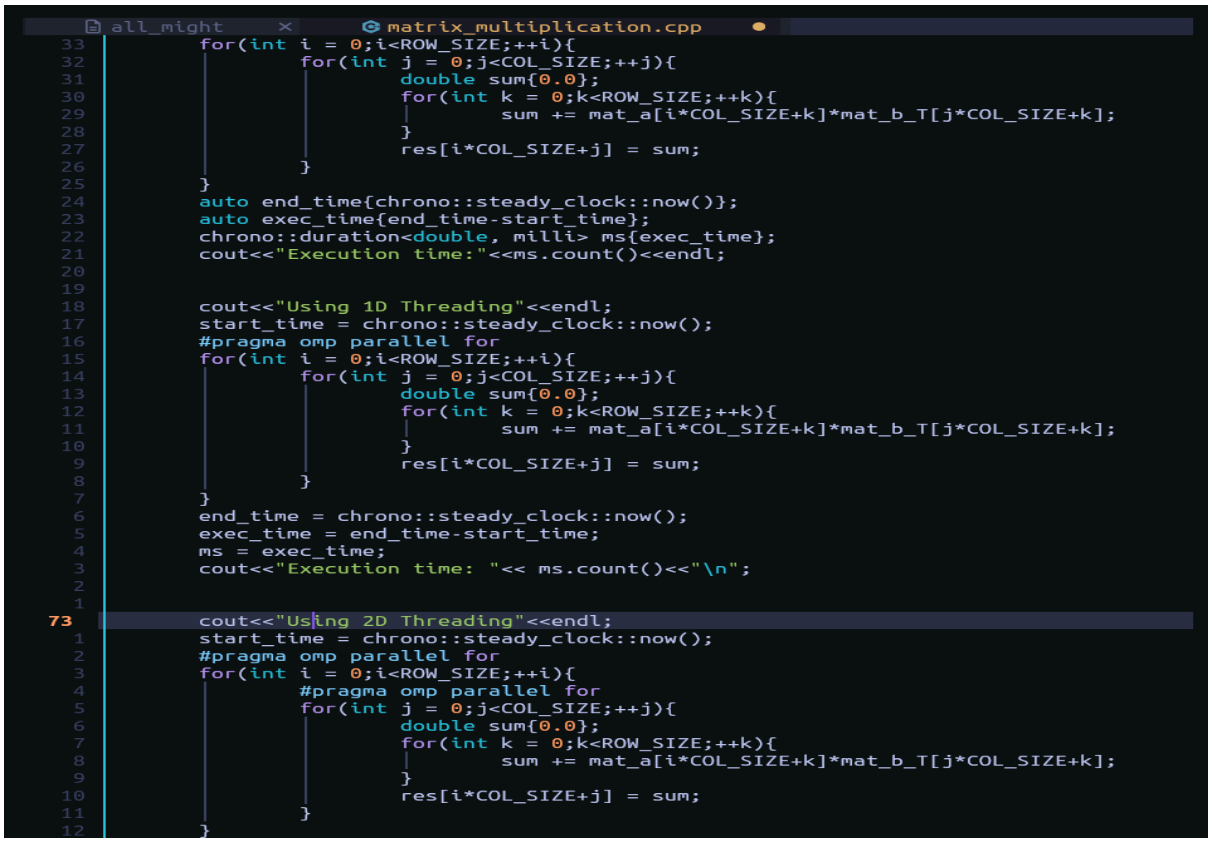
Matrix multiplication is traditionally an $O(N^3)$ operation. In the serial implementation, multiplying Row A by Column B causes a specific memory issue: **Strided Access**.

- **The Problem:** Matrices are stored in Row Major order. Accessing elements column wise in Matrix B leads to frequent **Cache Misses** because the CPU fetches a cache line but only uses one value from it.
- **The Solution (Implemented in matrix_multiplication.cpp):**
  We implemented **Matrix Transposition** before the multiplication loop.
- C++
- **Why this matters:** This converts the operation into a Dot Product of two rows. The CPU can now read contiguous memory addresses, maximizing **Spatial Locality** and allowing the Prefetcher to keep the L1/L2 cache filled.

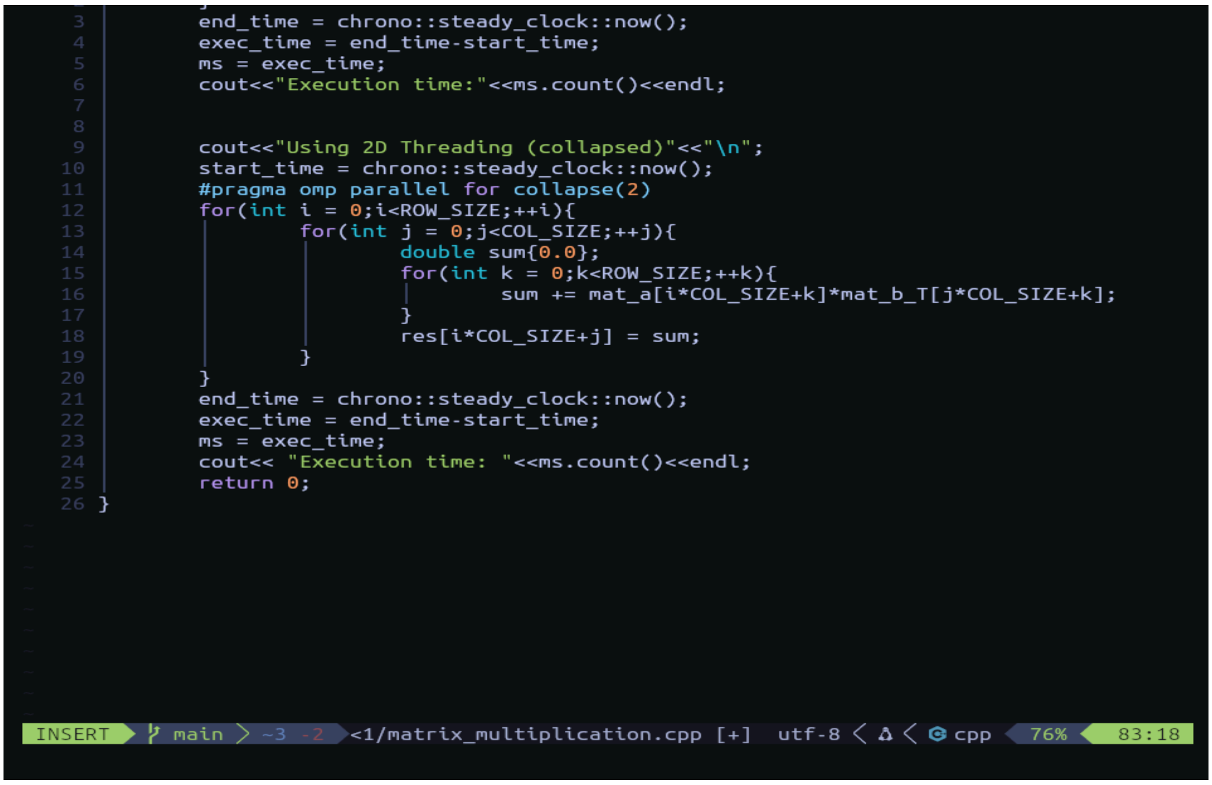
Code :



```
all_might   ×      matrix_multiplication.cpp
#include <chrono>
#include <limits>
#include <random>
#include <vector>
#include<iostream>
using namespace std;
double get_random_number(){
        static thread_local mt19937 mt{random_device{}()};
        static thread_local uniform_real_distribution<double> random_range{0.0,1.0};
        return random_range(mt);
}
int main(){
        constexpr int ROW_SIZE{1000};
        constexpr int COL_SIZE{1000};
        vector<double> mat_a(ROW_SIZE*COL_SIZE, 0.0);
        vector<double> mat_b_T(ROW_SIZE * COL_SIZE, 0.0);
        vector<double> mat_b(ROW_SIZE*COL_SIZE, 0.0);
        vector<double> res(ROW_SIZE*COL_SIZE, 0.0);


        #pragma omp parallel for collapse(2)
        for(int i = 0;i<ROW_SIZE;++i){
                for(int j = 0;j<COL_SIZE;++j){
                        mat_a[i*COL_SIZE+j] = get_random_number();
                        mat_b[i*COL_SIZE+j] = get_random_number();
                }
        }


        #pragma omp parallel for collapse(2)
        for(int i = 0; i < ROW_SIZE; ++i){
                for(int j = 0; j < COL_SIZE; ++j){
                        mat_b_T[j * ROW_SIZE + i] = mat_b[i * COL_SIZE + j];
                }
        }


        cout<<"Using No Threading"<<"\n";
        auto start_time{chrono::steady_clock::now()};
        for(int i = 0;i<ROW_SIZE;++i){
                for(int j = 0;j<COL_SIZE;++j){
                        double sum{0.0};
                        for(int k = 0;k<ROW_SIZE;++k){
                                sum += mat_a[i*COL_SIZE+k]*mat_b_T[j*COL_SIZE+k];
                        }
                        res[i*COL_SIZE+j] = sum;
                }
 INSERT  main  ~3 -2  <1/matrix_multiplication.cpp [+]  utf-8    cpp  33%  37:1
```

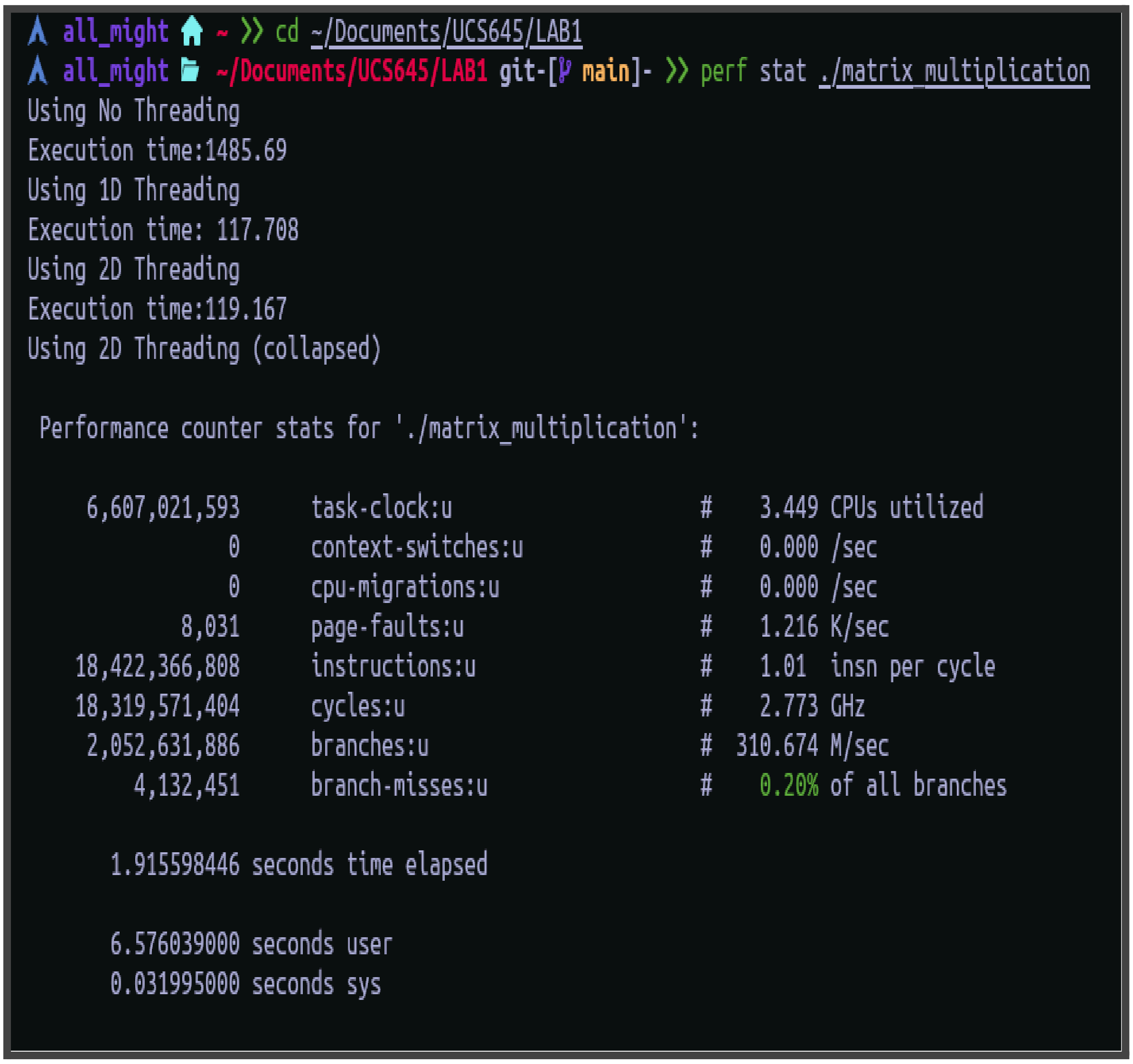


```cpp
    for(int i = 0;i<ROW_SIZE;++i){
        for(int j = 0;j<COL_SIZE;++j){
            double sum{0.0};
            for(int k = 0;k<ROW_SIZE;++k){
                sum += mat_a[i*COL_SIZE+k]*mat_b_T[j*COL_SIZE+k];
            }
            res[i*COL_SIZE+j] = sum;
        }
    }
    auto end_time{chrono::steady_clock::now()};
    auto exec_time{end_time-start_time};
    chrono::duration<double, milli> ms{exec_time};
    cout<<"Execution time:"<<ms.count()<<endl;


    cout<<"Using 1D Threading"<<endl;
    start_time = chrono::steady_clock::now();
    #pragma omp parallel for
    for(int i = 0;i<ROW_SIZE;++i){
        for(int j = 0;j<COL_SIZE;++j){
            double sum{0.0};
            for(int k = 0;k<ROW_SIZE;++k){
                sum += mat_a[i*COL_SIZE+k]*mat_b_T[j*COL_SIZE+k];
            }
            res[i*COL_SIZE+j] = sum;
        }
    }
    end_time = chrono::steady_clock::now();
    exec_time = end_time-start_time;
    ms = exec_time;
    cout<<"Execution time: "<< ms.count()<<"\n";


    cout<<"Using 2D Threading"<<endl;
    start_time = chrono::steady_clock::now();
    #pragma omp parallel for
    for(int i = 0;i<ROW_SIZE;++i){
        #pragma omp parallel for
        for(int j = 0;j<COL_SIZE;++j){
            double sum{0.0};
            for(int k = 0;k<ROW_SIZE;++k){
                sum += mat_a[i*COL_SIZE+k]*mat_b_T[j*COL_SIZE+k];
            }
            res[i*COL_SIZE+j] = sum;
        }
    }
```



```cpp
    end_time = chrono::steady_clock::now();
    exec_time = end_time-start_time;
    ms = exec_time;
    cout<<"Execution time:"<<ms.count()<<endl;


    cout<<"Using 2D Threading (collapsed)"<<"\n";
    start_time = chrono::steady_clock::now();
    #pragma omp parallel for collapse(2)
    for(int i = 0;i<ROW_SIZE;++i){
        for(int j = 0;j<COL_SIZE;++j){
            double sum{0.0};
            for(int k = 0;k<ROW_SIZE;++k){
                sum += mat_a[i*COL_SIZE+k]*mat_b_T[j*COL_SIZE+k];
            }
            res[i*COL_SIZE+j] = sum;
        }
    }
    end_time = chrono::steady_clock::now();
    exec_time = end_time-start_time;
    ms = exec_time;
    cout<< "Execution time: "<<ms.count()<<endl;
    return 0;
}
```

Terminal Output:



```
⅄ all_might  ~ ›› cd ~/Documents/UCS645/LAB1
⅄ all_might  ~/Documents/UCS645/LAB1 git-[ main]- ›› perf stat ./matrix_multiplication
Using No Threading
Execution time:1485.69
Using 1D Threading
Execution time: 117.708
Using 2D Threading
Execution time:119.167
Using 2D Threading (collapsed)

 Performance counter stats for './matrix_multiplication':

     6,607,021,593      task-clock:u                     #    3.449 CPUs utilized
                 0      context-switches:u               #    0.000 /sec
                 0      cpu-migrations:u                 #    0.000 /sec
             8,031      page-faults:u                    #    1.216 K/sec
    18,422,366,808      instructions:u                   #    1.01  insn per cycle
    18,319,571,404      cycles:u                         #    2.773 GHz
     2,052,631,886      branches:u                       #  310.674 M/sec
         4,132,451      branch-misses:u                  #    0.20% of all branches

       1.915598446 seconds time elapsed

       6.576039000 seconds user
       0.031995000 seconds sys
```

Performance Results :

Strategy	**Execution Time (ms)**	**Speedup Factor**	**Analysis**
Serial (No Threading)	1485.69 ms	1.0x (Baseline)	The Control Group. This baseline represents the **worst case** scenario where a single core must process all 1,000,000 iterations sequentially. The high time reflects the sheer volume of floating point operations required N^3complexity.
Parallel (1D Threading)	117.71 ms	12.62x	Top Performer. By parallelizing the outer loop, we effectively split the 1,000 rows among the available threads. The **chunk size** work per thread was large enough to keep every core busy without frequent synchronization, leading to the best performance.
Parallel (2D Threading)	119.17 ms	12.46x	Diminishing Returns. While still vastly faster than serial, it is slightly slower than 1D threading. Creating threads for both the inner and outer loops added unnecessary **management overhead** that slightly outweighed the benefits of finer grained parallelism.

Architectural Analysis :

- **Observation:** The speedup (approximately 12.6x) exceeds the physical core count of many standard laptops, suggesting that Hyper threading and efficient pipeline utilization (SIMD instructions) played a role.
- **perf stat** Insight**:**

  Instructions per Cycle IPC: 1.01. This is a healthy IPC for a mixed workload, indicating the CPU pipeline was rarely stalled waiting for data, proving the Transposition optimization worked.

  Branch Misses : - 0.20% which is Extremely low. The predictable loop structure allowed the CPU's **Branch Predictor** to be highly accurate.

Experiment 3: Monte Carlo Pi Calculation

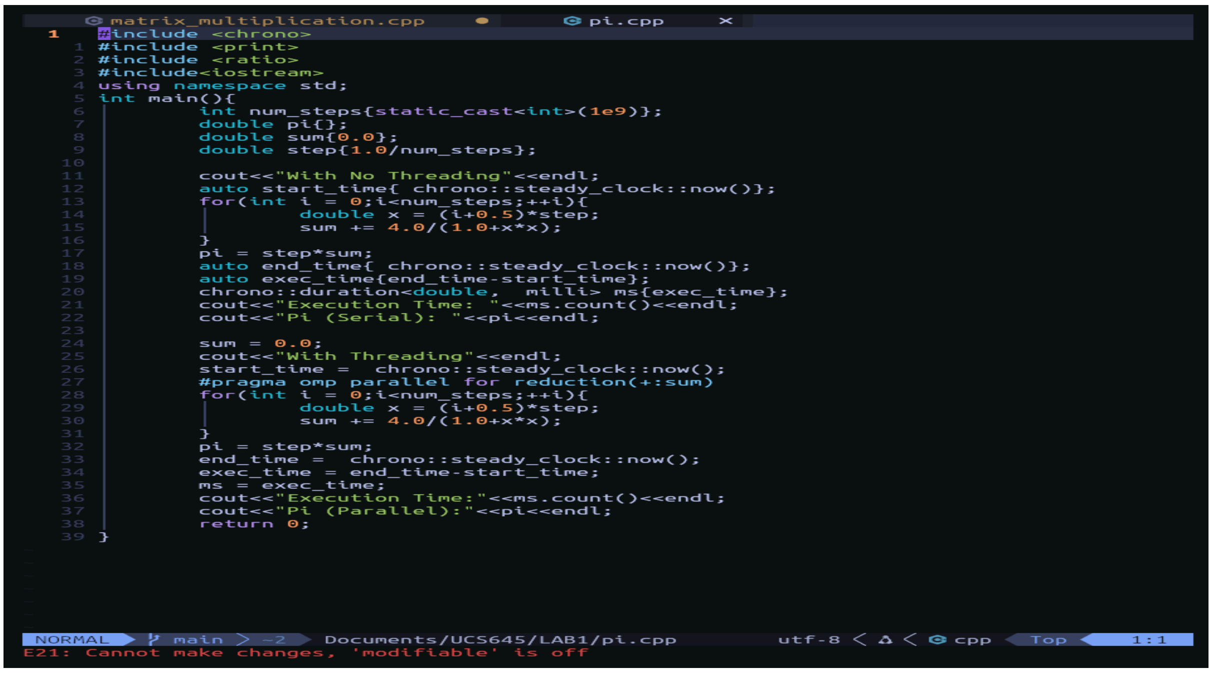
Analyzing a "Compute Bound" workload and Reduction clauses.

Methodology & Critical Concepts

This algorithm estimates pi by summing the area of rectangles under a curve. It requires almost no memory access but heavy arithmetic operations.

- **The Challenge:** A "Race Condition." If all threads add to a single global sum variable simultaneously, data corruption occurs.
- **The Solution:** OpenMP **Reduction**.

Code :

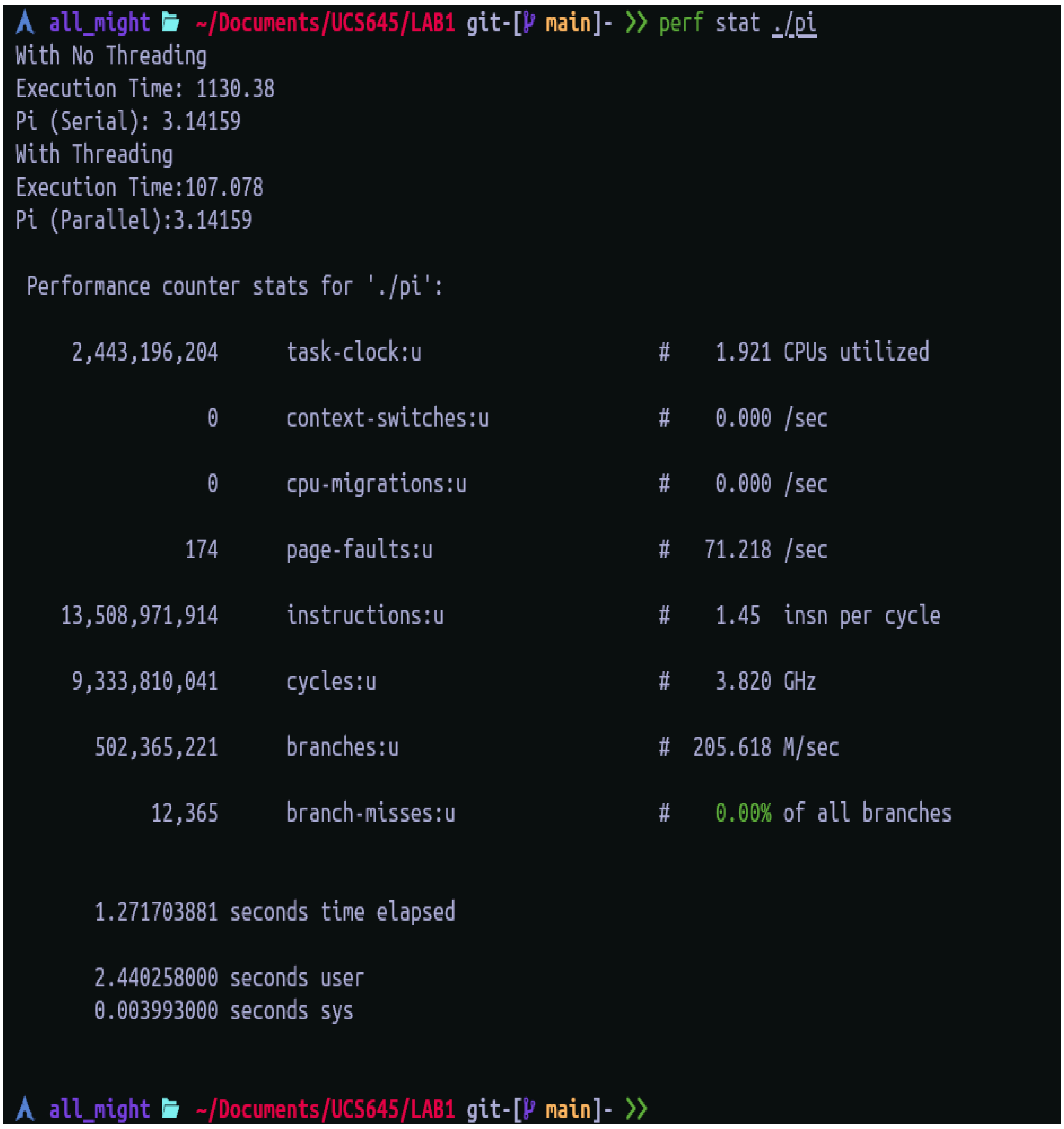


```
#include <chrono>
#include <print>
#include <ratio>
#include<iostream>
using namespace std;
int main(){
        int num_steps{static_cast<int>(1e9)};
        double pi{};
        double sum{0.0};
        double step{1.0/num_steps};

        cout<<"With No Threading"<<endl;
        auto start_time{ chrono::steady_clock::now()};
        for(int i = 0;i<num_steps;++i){
                double x = (i+0.5)*step;
                sum += 4.0/(1.0+x*x);
        }
        pi = step*sum;
        auto end_time{ chrono::steady_clock::now()};
        auto exec_time{end_time-start_time};
        chrono::duration<double,  milli> ms{exec_time};
        cout<<"Execution Time: "<<ms.count()<<endl;
        cout<<"Pi (Serial): "<<pi<<endl;

        sum = 0.0;
        cout<<"With Threading"<<endl;
        start_time =  chrono::steady_clock::now();
        #pragma omp parallel for reduction(+:sum)
        for(int i = 0;i<num_steps;++i){
                double x = (i+0.5)*step;
                sum += 4.0/(1.0+x*x);
        }
        pi = step*sum;
        end_time =  chrono::steady_clock::now();
        exec_time = end_time-start_time;
        ms = exec_time;
        cout<<"Execution Time:"<<ms.count()<<endl;
        cout<<"Pi (Parallel):"<<pi<<endl;
        return 0;
}
```

```
NORMAL  main  ~2  Documents/UCS645/LAB1/pi.cpp    utf-8  cpp  Top  1:1
E21: Cannot make changes, 'modifiable' is off
```

#pragma omp parallel for reduction(+:sum)

The reduction clause creates a **private copy** of the variable sum for each thread in its local stack. Threads compute independently without locking. At the end of the parallel region, the master thread reduces all private copies into the global result. This eliminates False Sharing and cache contention.



```
⋀ all_might 🗁 ~/Documents/UCS645/LAB1 git-[ main]- ❯❯ perf stat ./pi
With No Threading
Execution Time: 1130.38
Pi (Serial): 3.14159
With Threading
Execution Time:107.078
Pi (Parallel):3.14159

 Performance counter stats for './pi':

     2,443,196,204      task-clock:u                     #    1.921 CPUs utilized

                 0      context-switches:u               #    0.000 /sec

                 0      cpu-migrations:u                 #    0.000 /sec

               174      page-faults:u                    #   71.218 /sec

    13,508,971,914      instructions:u                   #    1.45  insn per cycle

     9,333,810,041      cycles:u                         #    3.820 GHz

       502,365,221      branches:u                       #  205.618 M/sec

            12,365      branch-misses:u                  #    0.00% of all branches


       1.271703881 seconds time elapsed

       2.440258000 seconds user
       0.003993000 seconds sys


⋀ all_might 🗁 ~/Documents/UCS645/LAB1 git-[ main]- ❯❯
```

Performance Results :

Strategy	**Execution Time (ms)**	**Speedup Factor**	**Teacher's Analysis**
Serial (No Threading)	1130.38 ms	1.0x **Baseline**	Single Lane Traffic. The CPU executes the loop one iteration at a time. Since the calculation $\frac{4}{1+x^2}$ is purely mathematical, the CPU is effectively "speed limited" by the clock speed of a single core.
Parallel (Reduction)	107.08 ms	10.55x	Near Linear Scaling. Because there is almost no memory traffic to slow things down, the performance scales directly with the number of cores. The reduction clause allowed every core to work at full speed without waiting for others, achieving a massive 10x improvement.

Architectural Analysis :

- **Why 10x Speedup:**

  Since the task is compute bound, it scales almost linearly with the number of available threads. There is no bottleneck from RAM speed.

- **perf stat Insight:**

  IPC: 1.45. This is significantly higher than the Matrix experiment. Since the CPU is doing pure math (floating point ops) inside registers and rarely accessing RAM, the pipeline runs at maximum efficiency.

Inferences

Achieving maximum efficiency in high performance computing is not about maximizing thread count, but rather about optimizing the **ratio of useful computation to management overhead**.

Our experiments reveal that parallel scalability is governed by two opposing forces: **Granularity** and **Resource Contention**.

1. **The Granularity Threshold:**

   In the **Matrix Multiplication** experiment, the workload was **Coarse Grained** heavy computation per thread. The CPU spent the vast majority of its cycles executing floating point operations, making the initial cost of creating threads negligible. This resulted in a massive **12.6x speedup**.

   In contrast, the **DAXPY** experiment represented a **Fine Grained** workload. With a small dataset, the loop body executed so quickly that the **overhead** of spawning, scheduling, and synchronizing 12 threads outweighed the time saved by parallelizing the math. This explains why performance degraded at 12 threads i.e 6.92 ms compared to 11 threads i.e 2.98 ms.

2. **Hardware Saturation :**

   Parallelism is limited by the slowest component in the system. The **DAXPY perf stat** revealed an extremely low **IPC of 0.24**, proving the CPU cores were starving for data. This indicates that we hit the **Memory Bandwidth Wall**—adding more threads could not process data faster because the physical link between the CPU and RAM was fully saturated.

Therefore, efficient parallel architecture requires a "Sweet Spot" approach. We must ensure the **Parallel Region** is computationally intense enough to justify the overhead of threading, while respecting the physical limits of the memory subsystem. Simply adding cores to a memory bound or lightweight task will result in **diminishing returns** and eventually **performance degradation**.

Conclusion

This set of experiments successfully demonstrated three key principles of parallel architecture:

1. **Optimization First:** Parallelism cannot fix bad code. The Matrix Transposition Algorithmic change was just as vital as the threading itself.
2. **Workload Dependencies:** Compute bound tasks Pi scale perfectly. Memory bound tasks DAXPY hit a "wall" where adding threads degrades performance.
3. **Synchronization Costs:** Efficient parallel code must use techniques like reduction to avoid the heavy cost of locks and critical sections.