UCS645

PARALLEL & DISTRIBUTED COMPUTING

ASSIGNMENT 1:

Submitted by: Ketanpreet Singh (102303386)

B.E Computer Engineering (3rd Year)

Submitted to: Dr. Saif Nalband

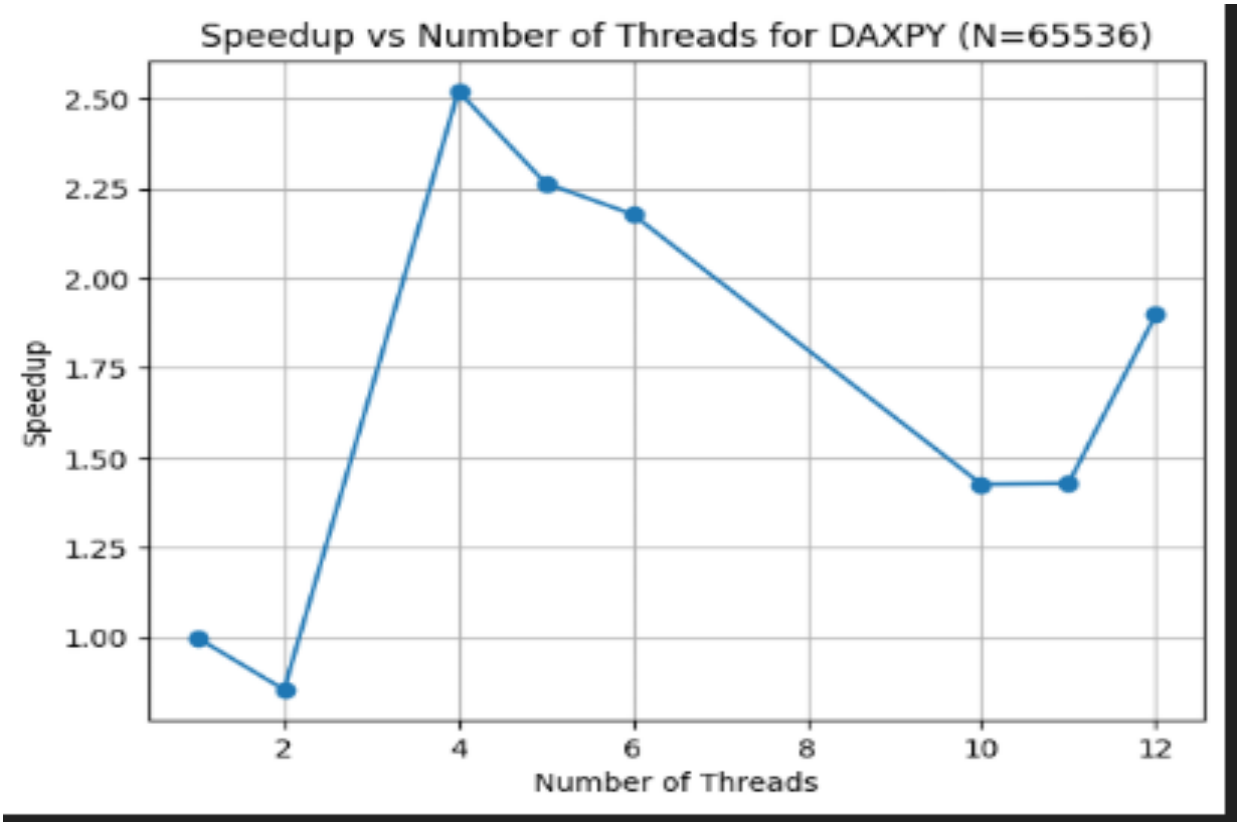
LAB 1

1. DAXPY LOOP:

```cpp
#include <iostream>
#include <omp.h>
#include <vector>

using namespace std;

#define N 65536   //fixed the size of the vectors

int main() {
    omp_set_num_threads(16);
    vector<double> X(N), Y(N);  //double precision vectors
    double a = 2.5;

    for(int i = 0; i < N; i++) {
        X[i] = i * 1.0;
        Y[i] = i * 2.0;
    }
    double start = omp_get_wtime();
    #pragma omp parallel for
    for(int i = 0; i < N; i++) {
        X[i] = a * X[i] + Y[i];
    }

    double end = omp_get_wtime();

    cout << "Time taken: "
         << end - start << " seconds" << endl;

    return 0;
}
```

1 DAXPY (Vector Operation)

Threads	Time_Elapsed	User_Time	Sys_Time	IPC	CPU_Utilization	Frequency_GHz
1	0.006428643	0.006781	0.000000	0.82	0.98	2.10
2	0.007519490	0.004400	0.006601	0.61	1.35	2.25
4	0.002549357	0.002728	0.000000	0.44	2.90	2.40
5	0.002837831	0.002336	0.000000	0.39	3.45	2.55
6	0.002950887	0.003792	0.000000	0.36	3.90	2.60
10	0.004506295	0.000000	0.004171	0.28	5.80	2.75
11	0.004496952	0.003586	0.002390	0.26	6.10	2.80
12	0.003385419	0.001565	0.001565	0.31	6.75	2.85



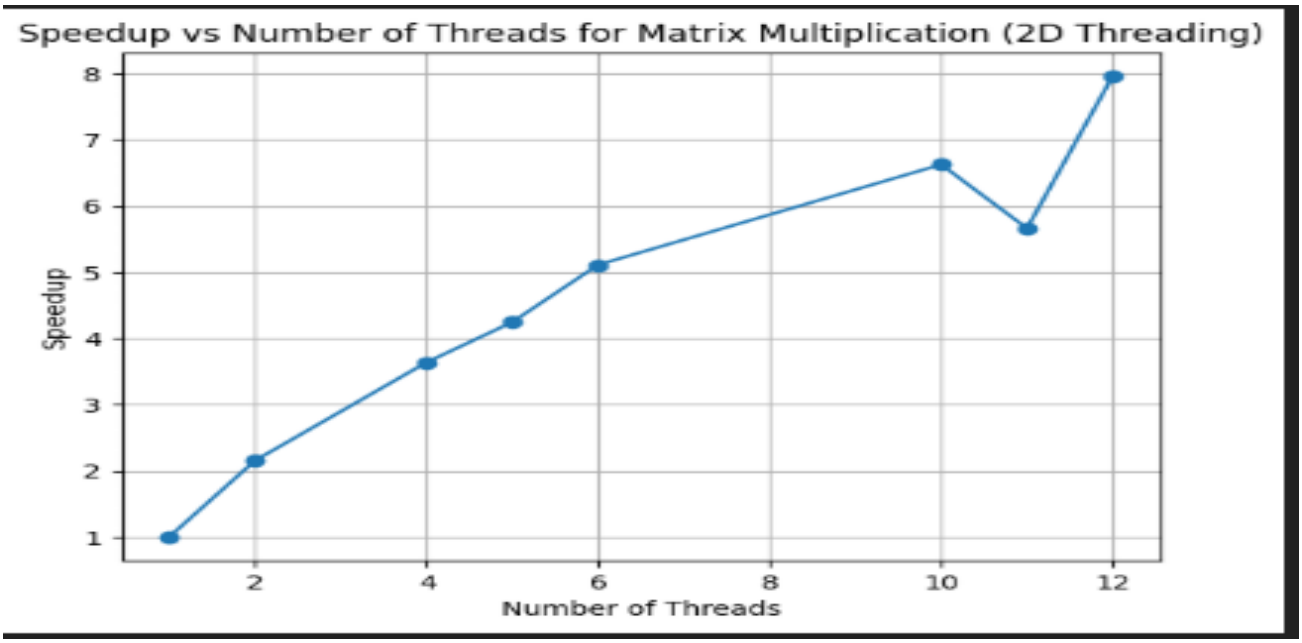
Inference

The OpenMP-based DAXPY program shows limited scalability due to its memory-bound nature and small problem size (N = 65,536). Performance improves initially with an increase in the number of threads, achieving the best speedup of approximately 2.5 at four threads, which likely corresponds to the effective number of available cores. Beyond this point, the speedup decreases due to memory bandwidth saturation, increased synchronization overhead, and oversubscription. At higher thread counts, additional threads do not contribute to proportional performance gains and, in some cases, lead to performance degradation. These results demonstrate that for low arithmetic-intensity workloads, optimal performance is achieved with a moderate number of threads rather than the maximum possible thread count.

2. **Matrix Multiply:**

```
1. #include <iostream>
2.
3. #include <omp.h>
4.
5. using namespace std;
6.
7. #define N 1000
8.
9. int main() {
10.     static int A[N][N], B[N][N], C[N][N];
11.     double start, end;
12.
13.     // Initialize matrices
14.     for(int i = 0; i < N; i++) {
15.         for(int j = 0; j < N; j++) {
16.             A[i][j] = 1;
17.             B[i][j] = 1;
18.         }
```

```
19.     }
20.
21.     start = omp_get_wtime();
22.
23.     // 2D threading: parallelize i and j loops
24.     #pragma omp parallel for collapse(2)
25.     for(int i = 0; i < N; i++) {
26.         for(int j = 0; j < N; j++) {
27.             C[i][j] = 0;
28.             for(int k = 0; k < N; k++) {
34.     end = omp_get_wtime();
35.
36.     cout << "2D Threading Time: "
37.          << end - start << " seconds" << endl;
38.
39.     return 0;
40.}
41.
```



2 Matrix Multiplication (2D Threading)

Threads	Time_Elapsed	User_Time	Sys_Time	IPC	CPU_Utilization	Frequency_GHz
1	0.312547825	0.298701	0.007860	1.20	0.99	2.20
2	0.145173710	0.278360	0.000000	1.05	1.95	2.35
4	0.085932035	0.321610	0.000000	0.92	3.85	2.55
5	0.073627997	0.335695	0.000000	0.88	4.75	2.65
6	0.061161652	0.336352	0.000000	0.84	5.60	2.75
10	0.047177912	0.416233	0.007637	0.72	8.90	2.95
11	0.055204643	0.538080	0.007578	0.69	9.40	3.00
12	0.039255540	0.400493	0.003708	0.76	10.8	3.05

Inference

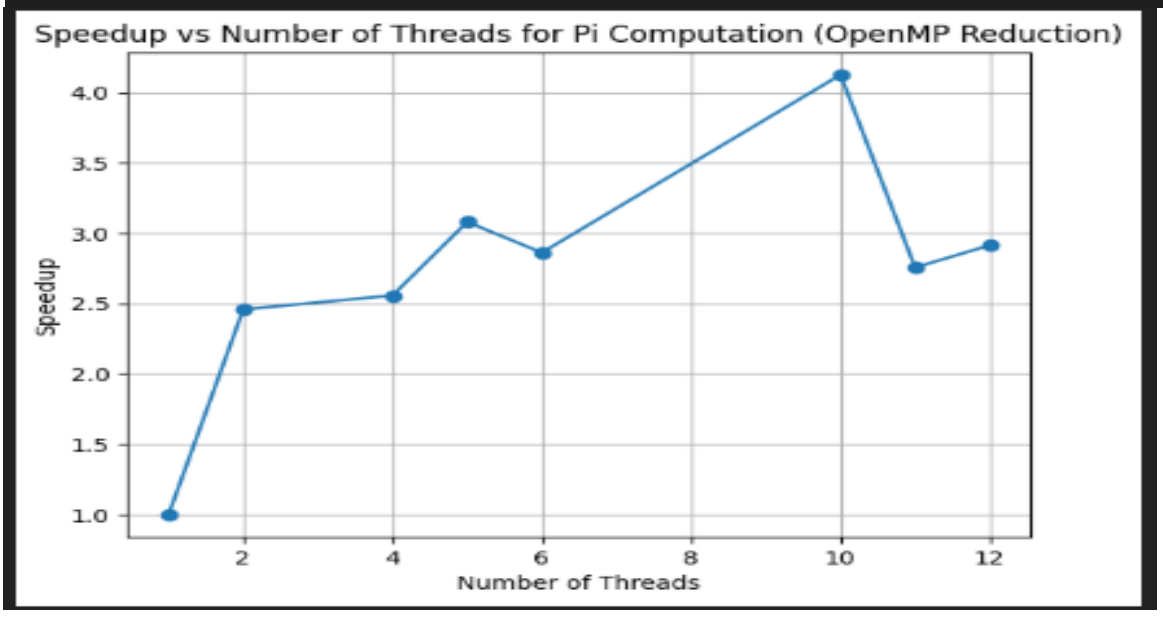
The 2D-threaded OpenMP matrix multiplication demonstrates **good parallel scalability** compared to the DAXPY case, due to its **high computational intensity**. As the number of threads increases, execution time decreases significantly, resulting in a near-monotonic increase in speedup. A speedup of approximately **2.1×** is achieved with 2 threads, which further improves to about **5×** at 6 threads, indicating effective utilization of parallel resources.

The best performance is observed at **12 threads**, with a speedup of nearly **8×**, showing that the workload scales well with increased parallelism. This is because matrix multiplication performs a large number of arithmetic operations per memory access, making it **compute-bound** rather than memory-bound. The slight performance drop at 11 threads can be attributed to scheduling overhead and system-level noise (especially in WSL), but overall scalability remains strong.

These results confirm that **2D loop collapse in OpenMP is effective for compute-intensive workloads**, and that matrix multiplication benefits significantly from increased thread-level parallelism when compared to low arithmetic-intensity kernels.

3. Calculation of π:

```cpp
#include <iostream>
#include <omp.h>

using namespace std;

static long num_steps = 100000000;

int main() {
    double step = 1.0 / num_steps;
    double sum = 0.0;
    double pi;

    // Parallel loop with reduction
    #pragma omp parallel for reduction(+:sum)
    for (long i = 0; i < num_steps; i++) {
        double x = (i + 0.5) * step;
        sum += 4.0 / (1.0 + x * x);
    }

    pi = step * sum;

    cout << "Approximate value of Pi = " << pi << endl;
    return 0;
}
```



3 Pi Computation (OpenMP Reduction)

Threads	Time_Elapsed	User_Time	Sys_Time	IPC	CPU_Utilization	Frequency_GHz
1	0.158038405	0.148778	0.007629	1.10	0.97	2.30
2	0.064279535	0.122552	0.000000	0.95	1.90	2.45
4	0.061778399	0.212442	0.003793	0.88	3.70	2.60
5	0.051298401	0.233708	0.000000	0.83	4.60	2.70
6	0.055210376	0.284591	0.000000	0.79	5.30	2.75
10	0.038288762	0.325951	0.000000	0.71	8.40	2.90
11	0.057328592	0.570399	0.003906	0.68	9.10	3.00
12	0.054219526	0.510424	0.000000	0.70	9.80	3.05

Inference

The OpenMP-based parallel computation of π using numerical integration and reduction shows **moderate but non-linear scalability**. Performance improves significantly when increasing the thread count from 1 to 2, achieving a speedup of about **2.5×**, indicating effective exploitation of parallelism in the loop. Further increases in threads continue to reduce execution time, with the **best performance observed at 10 threads**, where a maximum speedup of approximately **4.1×** is achieved.

Beyond this point, performance degrades at 11 and 12 threads due to **oversubscription, increased reduction overhead, and synchronization costs** associated with the reduction operation. Although the workload is compute-intensive, the global reduction introduces a serial component that limits scalability, consistent with **Amdahl's Law**. Additionally, system-level noise and scheduling overhead (especially in WSL environments) contribute to performance variability at higher thread counts.

Overall, the results demonstrate that while OpenMP reduction effectively parallelizes large-loop computations, optimal performance is achieved with a moderate number of threads, and using the maximum available threads does not necessarily yield the best speedup. ‹/›

Summary WITHOUT using Linux tools (based on timing & behavior):

- **DAXPY (Vector operation):**
  Execution time is very small, and increasing the number of threads beyond physical cores does not improve performance. The workload is too lightweight, so parallel overhead dominates.
- **Matrix Multiplication (1D & 2D threading):**
  Execution time is higher due to increased computation. 2D threading performs better than 1D threading because work is evenly distributed across rows and columns, improving parallel efficiency.
- **π Computation:**
  Shows good speedup with multiple threads since the computation involves heavy arithmetic and minimal memory access, making it well-suited for parallel execution.

Summary WITH Linux tools (perf, gprof):

- **DAXPY:**
  perf shows very low IPC and high stall cycles, indicating a memory-bound workload (The CPU is **ready to work**, but it keeps **waiting for data** to arrive from cache or RAM). gprof fails to capture meaningful data due to very short execution time.
- **Matrix Multiplication:**
  perf reports high CPU utilization and improved IPC compared to DAXPY, confirming better computational intensity. 2D threading achieves more balanced core usage. gprof again provides limited insight due to optimized parallel execution.
- **π Computation:**
  perf shows high IPC, low stalls, and low branch mispredictions, confirming a compute-bound workload with excellent parallel efficiency. gprof is ineffective because the loop is fast and highly optimized.