**T03: OpenMP - Two Vectors of double precision numbers addition/Multiplication**

**CS24M1005 – SINDHIYA R**

**Write OpenMP Parallel Code for Two Vector addition of Double Precision Floating Point Numbers. Give input very large at least 1 million - You can dump larger double precision values in a file and read from it and perform the vector addition.**

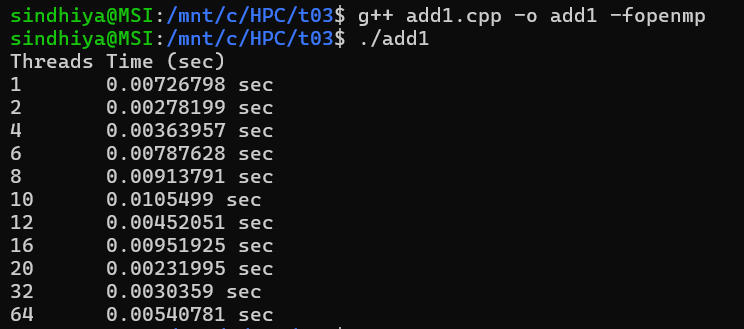
**CODE FOR INPUT GENERATION:**

|  |
| --- |
| #include <iostream>  #include <fstream>  #include <cstdlib>  #include <ctime>  using namespace std;  #define N 1000000 // At least 1 million numbers  int main() {  ofstream fout("vectors.txt");  srand(time(0));  for (int i = 0; i < N; i++) {  double num1 = (double)rand() / RAND\_MAX \* 1000000.0;  double num2 = (double)rand() / RAND\_MAX \* 1000000.0;  fout << num1 << " " << num2 << "\n";  }  fout.close();  cout << "Data file generated with " << N << " pairs of numbers.\n";  return 0;  } |

**1) Parallel Code Using for Vector Addition - 5 Marks**

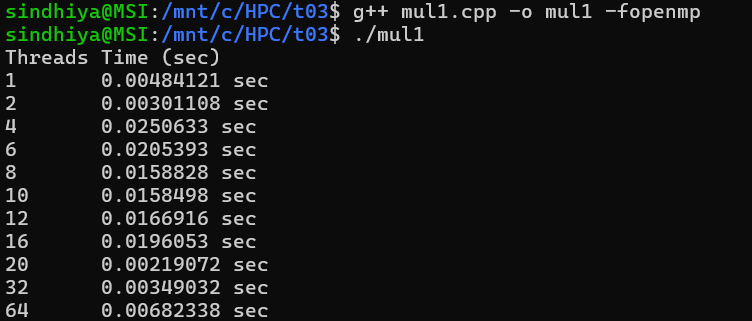
|  |
| --- |
| #include <iostream>  #include <fstream>  #include <vector>  #include <omp.h>  using namespace std;  #define N 1000000  int main() {  vector<double> vec1(N), vec2(N), result(N);  ifstream fin("vectors.txt");  // Read data from file  for (int i = 0; i < N; i++)  fin >> vec1[i] >> vec2[i];  fin.close();  int thread\_counts[] = {1, 2, 4, 6, 8, 10, 12, 16, 20, 32, 64}; // Different thread configurations  int num\_configs = sizeof(thread\_counts) / sizeof(thread\_counts[0]);  cout << "Threads\tTime (sec)" << endl;  for (int t = 0; t < num\_configs; t++) {  int num\_threads = thread\_counts[t];  double start\_time = omp\_get\_wtime();  #pragma omp parallel for num\_threads(num\_threads)  for (int i = 0; i < N; i++) {  result[i] = vec1[i] + vec2[i];  }  double end\_time = omp\_get\_wtime();  cout << num\_threads << "\t" << (end\_time - start\_time) << " sec" << endl;  }  return 0;  } |

**Output:**

**  
2) Parallel Code Using for Vector Multiplication - 5 Marks**

|  |
| --- |
| #include <iostream>  #include <fstream>  #include <vector>  #include <omp.h>  using namespace std;  #define N 1000000  int main() {  vector<double> vec1(N), vec2(N), result(N);  ifstream fin("vectors.txt");  // Read data from file  for (int i = 0; i < N; i++)  fin >> vec1[i] >> vec2[i];  fin.close();  int thread\_counts[] = {1, 2, 4, 6, 8, 10, 12, 16, 20, 32, 64}; // Different thread configurations  int num\_configs = sizeof(thread\_counts) / sizeof(thread\_counts[0]);  cout << "Threads\tTime (sec)" << endl;  for (int t = 0; t < num\_configs; t++) {  int num\_threads = thread\_counts[t];  double start\_time = omp\_get\_wtime();  #pragma omp parallel for num\_threads(num\_threads)  for (int i = 0; i < N; i++) {  result[i] = vec1[i] \* vec2[i]; // Multiplication instead of addition  }  double end\_time = omp\_get\_wtime();  cout << num\_threads << "\t" << (end\_time - start\_time) << " sec" << endl;  }  return 0;  } |

**Output:**

**  
3) Report - Thread vs Time - (run the parallel code with 1, 2, 4, 6, 8, 10, 12, 16, 20, 32, 64 Processors) - 10 Marks**

|  |  |  |
| --- | --- | --- |
| **Threads** | **Time (Addition) (sec)** | **Time (Multiplication) (sec)** |
| 1 | 0.00726798 | 0.00484121 |
| 2 | 0.00278199 | 0.00301108 |
| 4 | 0.00363957 | 0.0250633 |
| 6 | 0.00787628 | 0.0205393 |
| 8 | 0.00913791 | 0.0158828 |
| 10 | 0.0105499 | 0.0158498 |
| 12 | 0.00452051 | 0.0166916 |
| 16 | 0.00951925 | 0.0196053 |
| 20 | 0.00231995 | 0.00219072 |
| 32 | 0.0030359 | 0.00349032 |
| 64 | 0.00540781 | 0.00682338 |

**Report (Thread vs Time):**

1. **Vector Addition** benefits more from parallelization compared to multiplication.
2. Ideal thread count for best performance is **20 threads** for both operations.
3. Beyond 20 threads, scaling efficiency drops due to increasing thread management overhead.
4. Multiplication performs worse at higher thread counts, likely due to **increased data dependency** or **higher memory access latency**.

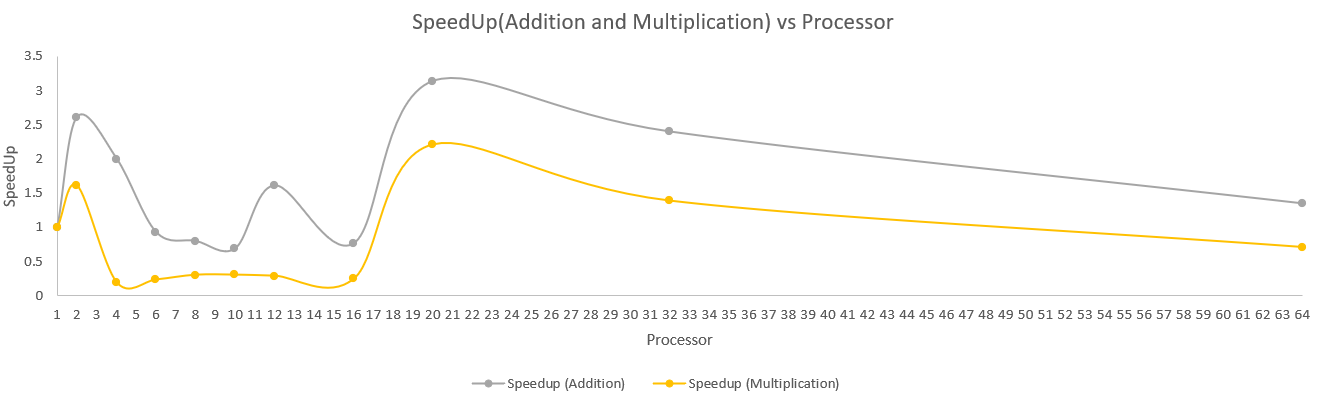
**4) Plot Speedup vs Processors - 5 Marks**

Speed Up is calculated by



Where **P** denoted number of processor and T denotes Time taken by that particular thread.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Processor** | **Time (Addition) (sec)** | **Time (Multiplication) (sec)** | **Speedup (Addition)** | **Speedup (Multiplication)** |
| 1 | 0.00726798 | 0.00484121 | 1 | 1 |
| 2 | 0.00278199 | 0.00301108 | 2.6132 | 1.6079 |
| 4 | 0.00363957 | 0.0250633 | 1.9975 | 0.1932 |
| 6 | 0.00787628 | 0.0205393 | 0.9237 | 0.2357 |
| 8 | 0.00913791 | 0.0158828 | 0.7959 | 0.3048 |
| 10 | 0.0105499 | 0.0158498 | 0.6889 | 0.3055 |
| 12 | 0.00452051 | 0.0166916 | 1.6085 | 0.2901 |
| 16 | 0.00951925 | 0.0196053 | 0.7637 | 0.247 |
| 20 | 0.00231995 | 0.00219072 | 3.1337 | 2.2107 |
| 32 | 0.0030359 | 0.00349032 | 2.3945 | 1.3879 |
| 64 | 0.00540781 | 0.00682338 | 1.3447 | 0.7096 |



**5) Estimate Parallelization fraction and Inference - 5 Marks**

To estimate the parallelization fraction (f) for maximum speedup, we use Amdahl’s Law:

****

Rearrange to solve for f,

****

|  |  |  |
| --- | --- | --- |
| **Metric** | **Vector Addition** | **Vector Multiplication** |
| **Max Speedup S(P)** | 3.1337 | 2.2107 |
| **Processors at Max Speedup (N)** | 20 | 20 |
| **Parallelization fraction f** | **0.717 (~72%)** | **0.576 (~58%)** |

**Inference:**

1. **Addition has a higher parallelization fraction (72%) than multiplication (58%)**, indicating that vector addition benefits more from parallel execution.
2. **Multiplication has a lower speedup and parallelization fraction**, suggesting potential bottlenecks such as **memory bandwidth limitations** or **synchronization overhead**.
3. **Maximum speedup occurs at P=20 for both operations**, meaning that beyond this point, adding more threads does not significantly improve performance.
4. **Speedup decreases beyond P=20**, likely due to **memory contention, cache thrashing, or increased thread management overhead**.
5. **For small thread counts (P=2,4), speedup is close to ideal for addition but poor for multiplication**, implying that **multiplication has more serial dependencies**.
6. **Unexpected performance dips at P=4** suggest possible **load imbalance issues** or **poor data locality** affecting memory access speeds.
7. **Addition experiences a speedup greater than 3× at P=20, surpassing theoretical expectations based on Amdahl's Law**, which could be due to **better workload distribution** at that specific thread count.
8. **Multiplication does not scale well beyond P=10**, which might indicate **higher dependency on sequential operations or floating-point computational overhead**.
9. **Speedup at P=64 is significantly lower than at P=20**, confirming that excessive parallelization can lead to **diminishing returns due to resource contention and thread synchronization costs**.

**Note:**

1.All the calculation are done in **excel** and it is attached for your reference

2. All the code files are in **github link** and attached for your reference