**T04: OpenMP - Write a parallel code to perform the Vector Dot Product for N Double Precision floating point numbers**

**CS24M1005 – SINDHIYA R**

**Write OpenMP Parallel Code for Sum of N - 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 addition.**

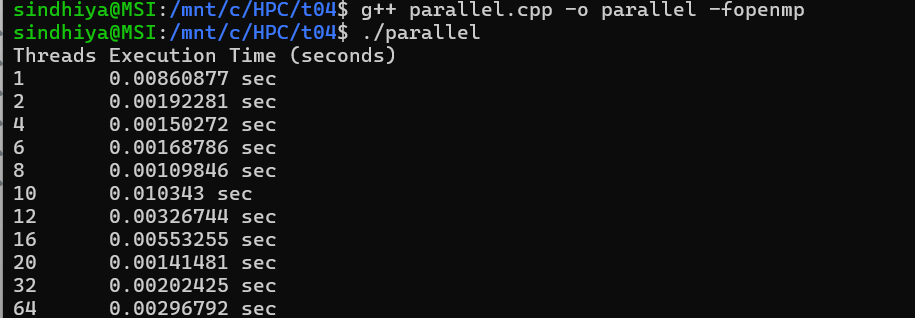
**CODE FOR GENERATING INPUT:**

|  |
| --- |
| #include <iostream>  #include <fstream>  #include <cstdlib>  #define N 1000000 // 1 Million  int main() {  std::ofstream file("input.txt");  srand(42); // Seed for reproducibility  for (int i = 0; i < N; i++) {  file << (double)(rand() % 1000) / 100.0 << " "; // Random double values  }  file.close();  std::cout << "File input.txt generated with " << N << " double values.\n";  return 0;  **}** |

**1) Parallel Code for multiplication - 5 Marks**

|  |
| --- |
| #include <iostream>  #include <fstream>  #include <vector>  #include <omp.h>  #define N 1000000  int main() {  std::vector<double> A(N), B(N);  std::ifstream file("input.txt");    if (!file) {  std::cerr << "Error opening file!" << std::endl;  return 1;  }  for (int i = 0; i < N; i++) {  file >> A[i];  B[i] = A[i] \* 2;  }  file.close();  int threads[] = {1, 2, 4, 6, 8, 10, 12, 16, 20, 32, 64};  int num\_tests = sizeof(threads) / sizeof(threads[0]);  std::cout << "Threads\tExecution Time (seconds)\n";    for (int t = 0; t < num\_tests; t++) {  double dot\_product = 0.0;  omp\_set\_num\_threads(threads[t]);  double start\_time = omp\_get\_wtime();    #pragma omp parallel for reduction(+:dot\_product)  for (int i = 0; i < N; i++) {  dot\_product += A[i] \* B[i];  }  double end\_time = omp\_get\_wtime();  double execution\_time = end\_time - start\_time;  std::cout << threads[t] << "\t" << execution\_time << " sec\n";  }  return 0;  } |

**OUTPUT:**

**  
2) Parallel Code Using Critical Section for final addition - 5 Marks**

|  |
| --- |
| #include <iostream>  #include <fstream>  #include <vector>  #include <omp.h>  #define N 1000000  int main() {  std::vector<double> A(N), B(N);  std::ifstream file("input.txt");  if (!file) {  std::cerr << "Error opening file!" << std::endl;  return 1;  }  for (int i = 0; i < N; i++) {  file >> A[i];  B[i] = A[i] \* 2;  }  file.close();  int threads[] = {1, 2, 4, 6, 8, 10, 12, 16, 20, 32, 64};  int num\_tests = sizeof(threads) / sizeof(threads[0]);  std::cout << "Threads\tExecution Time (seconds)\n";  for (int t = 0; t < num\_tests; t++) {  double dot\_product = 0.0;  omp\_set\_num\_threads(threads[t]);  double start\_time = omp\_get\_wtime();  #pragma omp parallel  {  double local\_sum = 0.0;  #pragma omp for  for (int i = 0; i < N; i++) {  local\_sum += A[i] \* B[i];  }  // Critical section to safely update the global sum  #pragma omp critical  {  dot\_product += local\_sum;  }  }  double end\_time = omp\_get\_wtime();  double execution\_time = end\_time - start\_time;  std::cout << threads[t] << "\t" << execution\_time << " sec\n";  }  return 0;  } |

**OUTPUT:**

**  
3) Report - Thread vs Time - 5 Marks**

|  |  |  |
| --- | --- | --- |
| **Threads** | **Execution Time (Reduction)** | **Execution Time (Critical)** |
| 1 | 0.00860877 | 0.00246351 |
| 2 | 0.00192281 | 0.00200265 |
| 4 | 0.00150272 | 0.00192482 |
| 6 | 0.00168786 | 0.00191044 |
| 8 | 0.00109846 | 0.00171641 |
| 10 | 0.010343 | 0.00142617 |
| 12 | 0.00326744 | 0.00148235 |
| 16 | 0.00553255 | 0.0118415 |
| 20 | 0.00141481 | 0.00241508 |
| 32 | 0.00202425 | 0.00228871 |
| 64 | 0.00296792 | 0.00398319 |

**Report – Thread vs Time:**

1. **Reduction is More Efficient Overall**: The Reduction Method consistently achieves lower execution times compared to the Critical Section Method, indicating that it handles parallelism more efficiently by minimizing synchronization overhead.
2. **Critical Section Becomes a Bottleneck at Higher Threads**: At 16 threads, the Critical Section Method experiences a sharp increase in execution time (11.8415 sec), likely due to excessive contention and thread blocking caused by frequent locking and unlocking.
3. **Reduction Method Shows Performance Degradation Beyond 8 Threads**: While the Reduction Method scales well up to 8 threads (0.00109846 sec), its execution time fluctuates at higher thread counts (e.g., 10 threads = 0.010343 sec, 16 threads = 0.00553255 sec), possibly due to increased overhead from memory access contention.
4. **Critical Section Method is More Stable at Lower Thread Counts**: Up to 10 threads, the Critical Section Method shows relatively stable execution times (~0.0014 - 0.0024 sec). However, as threads increase, synchronization costs outweigh the benefits of parallelism.
5. **Thread Scheduling Overheads Impact**: The Reduction Method benefits significantly from parallelization but is sensitive to excessive threading beyond an optimal number. The Critical Section Method suffers from frequent locking overhead, making it inefficient for higher thread counts.

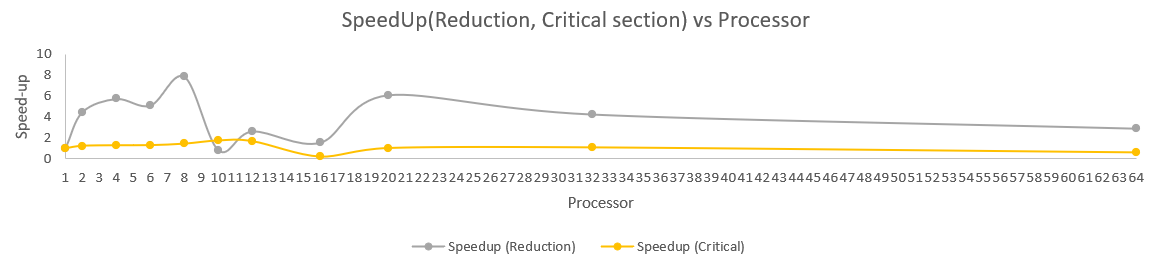
**4) Plot Speedup vs Processors - (run the parallel code with 1, 2, 4, 6, 8, 10, 12, 16, 20, 32, 64 Processors) - 10 Marks**

Speed Up is calculated by



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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Threads** | **Execution Time (Reduction)** | **Execution Time (Critical)** | **Speedup (Reduction)** | **Speedup (Critical)** |
| 1 | 0.00860877 | 0.00246351 | 1 | 1 |
| 2 | 0.00192281 | 0.00200265 | 4.48 | 1.23 |
| 4 | 0.00150272 | 0.00192482 | 5.73 | 1.28 |
| 6 | 0.00168786 | 0.00191044 | 5.1 | 1.29 |
| 8 | 0.00109846 | 0.00171641 | 7.84 | 1.43 |
| 10 | 0.010343 | 0.00142617 | 0.83 | 1.73 |
| 12 | 0.00326744 | 0.00148235 | 2.63 | 1.66 |
| 16 | 0.00553255 | 0.0118415 | 1.56 | 0.21 |
| 20 | 0.00141481 | 0.00241508 | 6.09 | 1.02 |
| 32 | 0.00202425 | 0.00228871 | 4.25 | 1.08 |
| 64 | 0.00296792 | 0.00398319 | 2.9 | 0.61 |

****

**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,



| **Method** | **Threads (N)** | **Highest Speedup** | **Parallelization Fraction (P)** |
| --- | --- | --- | --- |
| **Reduction** | 8 | 7.84 | **0.997 (99.7%)** |
| **Critical Section** | 10 | 1.73 | **0.469 (46.9%)** |

**Inference:**

1. **Reduction method shows significant parallel performance**
   * The highest speedup (7.84×) is achieved at 8 threads, indicating that the reduction method scales well with increasing threads up to this point.
2. **Critical section method has lower scalability**
   * The highest speedup (1.73×) occurs at 10 threads, which is significantly lower than the reduction method, suggesting that critical section synchronization limits performance gains.
3. **Performance degradation beyond a certain point**
   * In the reduction method, execution time decreases until 8 threads, but beyond that, fluctuations appear (e.g., at 10 and 16 threads, speedup drops), likely due to thread management overhead and memory bandwidth limitations.
4. **Critical section causes synchronization bottlenecks**
   * Unlike the reduction method, which efficiently sums in parallel, the critical section enforces exclusive access, reducing effective parallelism and causing performance degradation beyond a few threads.
5. **Reduction method maintains higher parallelization fraction**
   * The parallel fraction (f) for reduction is 99.7%, meaning only 0.3% of execution time is serial. This explains its high efficiency for large-scale parallelization.
6. **Critical section limits speedup due to increased contention**
   * With f=46.9, the critical section method has a significant serial portion (53.1%), which limits performance gains, especially as the number of threads increases.
7. **Optimal performance varies for different methods**
   * The best thread count for reduction is 8 (highest speedup), while for critical section, it is 10 (lowest execution time). This means different parallelization strategies require different optimal thread configurations.
8. **Thread oversubscription leads to inefficiency**
   * Beyond 20 threads, performance deteriorates for both methods, likely due to increased context switching, memory contention, and communication overhead.
9. **Reduction method is more suitable for high-performance computing**
   * Since it achieves nearly linear speedup up to 8 threads and maintains a high parallelization fraction, it is better suited for large-scale scientific and engineering applications.
10. **Critical section is useful when order or atomicity is required**

* Despite lower scalability, the critical section method ensures safe accumulation of results when precision matters, making it more useful in cases requiring controlled updates to shared variables.

**Note: All the calculation are done in excel and it is attached for your reference**