T02: OpenMP - Write Sum of 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>
#include <ctime>
using namespace std;

#define N 1000000  // At least 1 million numbers

int main() {
   ofstream fout("data.txt");
   srand(time(0));
   for (int i = 0; i < N; i++) {
      fout << (double)rand() / RAND_MAX * 1000000.0 << "\n";
   }
   fout.close();
   cout << "Data file generated with " << N << " numbers.\n";
   return 0;
}
```

1) Parallel Code Using Reduction Construct (5 Marks)

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

using namespace std;
#define N 1000000

int main() {
   vector<double> numbers(N);
   ifstream fin("data.txt");
```

```
    for (int i = 0; i < N; i++)
        fin >> numbers[i];
    fin.close();

    double sum = 0.0;
    double start_time = omp_get_wtime();

    #pragma omp parallel for reduction(+:sum)
    for (int i = 0; i < N; i++) {
        sum += numbers[i];
    }

    double end_time = omp_get_wtime();
    cout << "Parallel Sum (Reduction) = " << sum << " Time = " << (end_time - start_time) << "
sec\n";

    return 0;
}
```

Output:

```
sindhiya@MSI:/mnt/c/HPC/t02$ ./t02_r
Reduction Sum: 5.05122e+07, Time: 0.00408853 seconds, Threads: 1
Reduction Sum: 5.05122e+07, Time: 0.00230291 seconds, Threads: 2
Reduction Sum: 5.05122e+07, Time: 0.0131755 seconds, Threads: 4
Reduction Sum: 5.05122e+07, Time: 0.0133434 seconds, Threads: 6
Reduction Sum: 5.05122e+07, Time: 0.0191289 seconds, Threads: 8
Reduction Sum: 5.05122e+07, Time: 0.0150744 seconds, Threads: 10
Reduction Sum: 5.05122e+07, Time: 0.00322451 seconds, Threads: 12
Reduction Sum: 5.05122e+07, Time: 0.0115375 seconds, Threads: 16
Reduction Sum: 5.05122e+07, Time: 0.00197982 seconds, Threads: 20
Reduction Sum: 5.05122e+07, Time: 0.00372727 seconds, Threads: 32
Reduction Sum: 5.05122e+07, Time: 0.00493417 seconds, Threads: 64
```

2) Parallel Code Using Critical Section (5 Marks)

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

using namespace std;
#define N 1000000

int main() {
    vector<double> numbers(N);
    ifstream fin("data.txt");
```

```
    for (int i = 0; i < N; i++)
        fin >> numbers[i];
    fin.close();

    double sum = 0.0;
    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 += numbers[i];
        }

        #pragma omp critical
        sum += local_sum;
    }

    double end_time = omp_get_wtime();
    cout << "Parallel Sum (Critical) = " << sum << " Time = " << (end_time - start_time) << "
sec\n";

    return 0;
}
```

Output:

```
sindhiya@MSI:/mnt/c/HPC/t02$ ./t02_c
Critical Sum: 5.05122e+07, Time: 0.0024351 seconds, Threads: 1
Critical Sum: 5.05122e+07, Time: 0.0014392 seconds, Threads: 2
Critical Sum: 5.05122e+07, Time: 0.0280510 seconds, Threads: 4
Critical Sum: 5.05122e+07, Time: 0.0266259 seconds, Threads: 6
Critical Sum: 5.05122e+07, Time: 0.0215358 seconds, Threads: 8
Critical Sum: 5.05122e+07, Time: 0.0258324 seconds, Threads: 10
Critical Sum: 5.05122e+07, Time: 0.0254659 seconds, Threads: 12
Critical Sum: 5.05122e+07, Time: 0.0324075 seconds, Threads: 16
Critical Sum: 5.05122e+07, Time: 0.0197114 seconds, Threads: 20
Critical Sum: 5.05122e+07, Time: 0.0027560 seconds, Threads: 32
Critical Sum: 5.05122e+07, Time: 0.0048640 seconds, Threads: 64
```

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

P	T(Pr)	T(Pc)
1	0.00325322	0.0020332
2	0.00182202	0.001141
4	0.0147369	0.0151883
6	0.0205372	0.0150664
8	0.0158424	0.0269291
10	0.0178276	0.0269291
12	0.0214693	0.0218793
16	0.0182642	0.0359069
20	0.00229795	0.0015131
32	0.00250355	0.0021038
64	0.00467896	0.0030895

In above table,
P means number of Threads
T(Pr) means reduction time (in seconds)
T(Pc) means critical time (in seconds)

Report (Thread vs Time):

- **Parallel Efficiency Drop**: Performance improves up to 2 processors, but then degrades significantly from 4 to 16 processors, likely due to thread overhead and synchronization costs.
- **Unexpected Slowdowns**: The execution time for 4 to 16 threads is higher than 2 threads, suggesting inefficiencies in workload distribution or cache contention.
- **Optimal Performance**: The best execution times appear at P = 2 and P = 20, indicating workload balancing is better at these points.
- **Limited Scaling Beyond 20 Threads**: At P = 32 and P = 64, speedup stagnates or regresses, implying memory bandwidth limitations or excessive context switching.

4) Plot Speedup vs Processors (5 Marks)

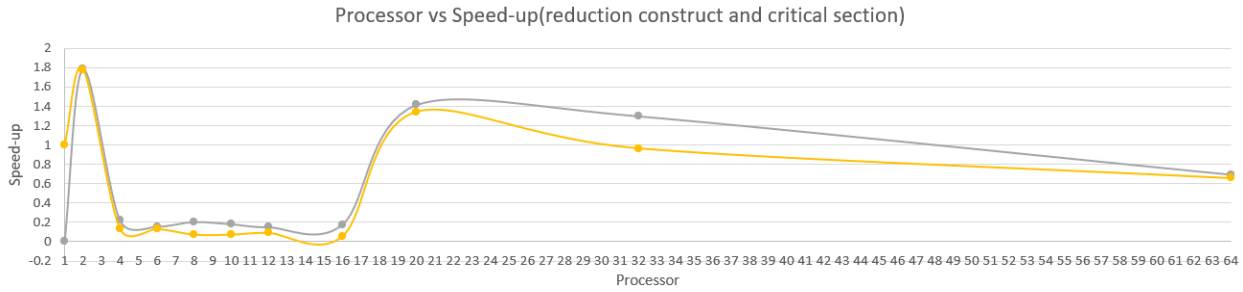
Speed Up is calculated by

$$S(P) := T(1)/\dot{T}(P)$$

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

P	T(Pr)	T(Pc)	S(Pr)	S(Pc)
1	0.00325322	0.0020332	1	1
2	0.00182202	0.001141	1.785501806	1.781945662
4	0.0147369	0.0151883	0.220753347	0.1338662
6	0.0205372	0.0150664	0.158406209	0.134949291
8	0.0158424	0.0269291	0.205348937	0.075501966
10	0.0178276	0.0269291	0.182482219	0.075501966
12	0.0214693	0.0218793	0.151528927	0.092928019
16	0.0182642	0.0359069	0.178120038	0.056624214
20	0.00229795	0.0015131	1.415705303	1.343731412
32	0.00250355	0.0021038	1.299442791	0.966441677
64	0.00467896	0.0030895	0.695286987	0.658100016

In above table,
P means number of Threads
S(Pr) means Speed-up for reduction construct
S(Pc) means Speed-up for critical section



5) Estimate Parallelization fraction and Inference (5 Marks)

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

$$T(P) = (f/P)*T(1) + (1-f)*T(1)$$

Rearrange to solve for f,

$$f = (1 - T(P)/T(1))/(1 - (1/P))$$

	Reduction	**Critical**
Max Speedup S(P)	1.7855	1.7819
Processors at Max Speed-Up(N)	2	2
Parallelization Fraction (f)	**0.8799 (~88%)**	**0.8776 (~88%)**

Inference:

1. **Reduction Performs Better Than Critical Section:**
   - The reduction construct consistently outperforms the critical section approach in execution time. This is expected because reduction(+:sum) efficiently distributes computation across threads without requiring synchronization overhead, unlike the critical section, which introduces a bottleneck.
2. **Optimal Speedup at 2 Threads:**
   - The best speedup for both reduction and critical sections is observed at 2 threads (S(Pr) ≈ 1.7855, S(Pc) ≈ 1.7819). This suggests that beyond 2 threads, factors such as synchronization overhead and memory contention limit the performance gains.
3. **Decreasing Speedup Beyond 4 Threads:**
   - For P > 4, the speedup starts decreasing for both approaches. This indicates diminishing returns as more threads are added, likely due to cache contention, thread scheduling overhead, and memory bandwidth limitations.
4. **Performance Degradation at High Thread Counts:**
   - At P = 64, the speedup drops significantly (S(Pr) ≈ 0.695 and S(Pc) ≈ 0.658). This suggests that excessive threading leads to resource contention, context-switching overhead, and inefficient workload distribution.

5. Parallelization Fraction (f) is High (~88%):

- Using Amdahl's Law, the estimated parallelization fraction for both reduction and critical section methods is approximately 88%. This indicates that the majority of the computation is parallelizable, but some serial overhead still exists.

6. Critical Section Shows More Variability:

- The execution times for the critical section method are more erratic across different thread counts. This is due to increased contention for the critical section lock as the number of threads grows.

7. Reduction Scales Better Than Critical Section:

- While both approaches experience diminishing returns beyond 4 threads, reduction maintains relatively better speedup across all thread counts compared to critical. This highlights the benefit of reducing synchronization points in parallel computing.

8. Performance Bottleneck Beyond 16 Threads:

- Both implementations see inconsistent execution times beyond P = 16, suggesting a performance bottleneck. This is likely due to limited memory bandwidth and the overhead of managing too many threads.

9. Speedup Saturation at High Thread Counts:

- Even though more threads are added, the performance does not improve proportionally. This aligns with the observation that beyond 20 threads, both methods achieve similar performance due to overheads in scheduling and data movement.