**Assignment 2**

**Question 1:**

This program is designed to sort an array present on each of the processes such that the final sequence of the array is continuous on all processes and that each process has all the number in the range defined by the total num. of the processes.

For example, let’s say.

process 0 has {0.5, 0.1, 0.3},

process 1 has {0.2, 0.7, 0.6},

process 2 has {0.8, 0.4, 0.9}

After sorting, the final results would look like this:

Process 0: {0.1, 0.2, 0.3}

Process 1: {0.4, 0.5, 0.6}

Process 2: {0.7, 0.8, 0.9}

Process 0 has all the numbers in range 0 to 0.33 (1/p), Process 1 has all the numbers in range 0.33(1/p) to 0.67(2/p), and Process 2 has all the numbers in range 0.67(2/p) to 1.0(3/p), here p = 3.

To perform this, I dynamically generated two arrays named ‘send\_counts’ and ‘recv\_counts’. The length of both arrays is equal to the number of processes ‘p’. ‘send\_count’ contains how many elements a process will send to other process. In our exqmple the arrays will have the following values.

Process 0:

Send\_counts: {0, 1, 0}

Recv\_counts: {0, 1, 0}

Process 1:

Send\_counts: {1, 0, 1}

Recv\_counts: {1, 0, 1}

Process 2:

Send\_counts: {0, 1, 0}

Recv\_counts: {0, 1, 0}

The ‘send\_counts’ of Process 0 is {0, 1, 0} meaning it will send Process 0 (itself) 0 elements, Process 1 one element, and Process 2 zero elements.

Similarly, ‘recv\_counts’ of Process 0 is {0, 1, 0} meaning it will receive zero elements from Process 0 (itslef), one element from Process 1, and zero elements from Process 2. This information is conveyed to all using ‘MPI\_Alltoall()’ function.

After receiving the arrays, the final sizes of the arrays are determined as after sorting it is not necessary that all the processes will have equal number of elements. The final size of the arrays can be determined by summing all the values in ‘recv\_count’ array. The ‘local\_arr’ is then sorted to calculate the offsets which indicate where in the buffer to start sending or receiving for each process.

The send offset for Process 0 will be:

send\_offset: {0, 0, 1}

recv\_offset: {0, 0, 1}

Meaning for Process 0, it starts sending to Process 1 at index 0 of its local array, and to Process 2 at index 1.

Similarly, it starts receiving from Process 1 at index 0 of its new local array, and from Process 2 at index 1.

Then using this offset values with ‘MPI\_Alltoall()’ function the array elements are sent to the respective processes. After this operation is completed, the elements are then sorted to achieve the desired output.

**Output:**

N = 100

A screenshot of a computer program

Description automatically generated

N = 100000

A screenshot of a computer program

Description automatically generated

N = 1000000

A screenshot of a computer screen

Description automatically generated

N = 100,000,000

A screenshot of a computer screen

Description automatically generated

**Performance Analysis:**

Speedup (S): Speedup measures how much faster the parallel algorithm is compared to the best single-process algorithm. S = Time taken by serial algorithm / Time taken by parallel algorithm.

Efficiency (E): Efficiency is the speedup divided by the number of processes. E = Speedup / Number of processes.

I'm using 10 processes with varying array length maximum being 100,000,000.

Speedup - Initially, when N is small, the speedup is low because there's extra work needed to manage the processes. But as N increases, the speedup improves significantly, indicating the parallel algorithm works more efficiently for larger problems. For instance, with N=100,000,000, the speedup is about 4.5, meaning the parallel algorithm is approximately 4.5 times faster than the single-process version.

Efficiency - For small N, efficiency is low due to the extra work required to manage the processes. However, as N grows, efficiency approaches 1 (though it remains less than 1), indicating better utilization of the parallel system. For example, with N=100,000,000, the efficiency is about 0.71, meaning the parallel algorithm is using about 71% of the potential performance of the 10-process system.

Conclusion - From speedup and efficiency calculations, we can see a significant improvement in the parallel algorithm we have used.

**Question 2:**

This program parallelizes the program to find the Matrix made using Sums of a matrix. An element in the Asum is the sum of all previous elements in A from that index. Asumtask is the Matrix of sums generated using OpenMP tasks. Similary Asumtaskblock is the Matrix of sums generated uses blocks to optimize the OpenMP tasks.

The ‘compute\_tasks()’ function performs the same computation as compute\_serial, but in parallel using OpenMP tasks. It uses the #pragma omp parallel directive to create a parallel region and the #pragma omp single directive to ensure only one thread executes the code, while tasks are created for other threads to execute. Dependencies are set using depend on clauses to ensure tasks are executed in the correct order. Each element task depends on its left and top neighbors, ensuring correct prefix sum computation in parallel.

The ‘compute\_tasks\_blocks()’ function further optimizes the computation by dividing the matrix into smaller blocks and using OpenMP tasks. Each block is computed separately in parallel, reducing the number of dependencies and improving performance. The matrix is divided into blocks of size block\_size, and computations are performed block by block. Dependencies are set up similarly to compute\_tasks but on a block level, ensuring each block is computed after its required dependencies.

**Output:**

N = 1000  
Block size = 64

**A computer screen shot of a program

Description automatically generated**

N = 2000

A computer screen shot of a program

Description automatically generated

Time taken for different processes for matrix size 1000.

|  |  |  |  |
| --- | --- | --- | --- |
| **No. of processes** | **Serial** | **OpenMP Task** | **OpenMP Task Block** |
| 4 | 0.01 | 3.16 | 0.025 |
| 8 | 0.0022 | 6.77 | 0.025 |
| 16 | 0.0022 | 8.4 | 0.052 |
| 32 | 0.0022 | 11.8 | 0.020 |
| 64 | 0.0025 | 13.2 | 0.038 |

Time taken for different processes for matrix size 2000.

|  |  |  |  |
| --- | --- | --- | --- |
| **No. of processes** | **Serial** | **OpenMP Task** | **OpenMP Task Block** |
| 4 | 0.0072 | 11.95 | 0.0048 |
| 8 | 0.0074 | 13.84 | 0.0029 |
| 16 | 0.0077 | 13.28 | 0.0059 |
| 32 | 0.0084 | 27.77 | 0.0065 |
| 64 | 0.0094 | 29.34 | 0.0094 |

The time for serial computation increases linearly as the array size (N) grows, and it is the quickest method. The OpenMP tasks approach does worse than the serial method because managing many small tasks adds overhead, especially with more threads. However, for larger matrices (N = 2000), OpenMP tasks perform much better compared to the serial method than they did when N was 1000.