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

```
[tpatel44@mcs1 assignment2]$ mpicc -O2 sortmpi.c -o sortmpi.x
[tpatel44@mcs1 assignment2]$ mpirun -np 10 ./sortmpi.x
Process 3: 0.109382 0.860086 0.652264 0.331643 0.667135 ... 0.960377 0.490783 0.456525 0.547204 0.012792

Process 5: 0.321713 0.683908 0.253759 0.466483 0.032274 ... 0.799856 0.009878 0.103106 0.023525 0.833187

Process 7: 0.038394 0.508550 0.359623 0.106783 0.398517 ... 0.207982 0.547828 0.720873 0.305221 0.803604

Process 8: 0.895827 0.920021 0.159223 0.925630 0.830794 ... 0.048351 0.531278 0.586136 0.227466 0.217497

Process 9: 0.253278 0.333522 0.463607 0.244769 0.265087 ... 0.181576 0.491100 0.466494 0.048341 0.643944

Process 2: 0.249606 0.946135 0.848285 0.009839 0.732033 ... 0.403296 0.309942 0.065316 0.744503 0.282305

Process 4: 0.964088 0.770910 0.451694 0.147309 0.098381 ... 0.767084 0.681507 0.679271 0.852235 0.536946

Process 6: 0.676575 0.591592 0.550942 0.782399 0.460020 ... 0.464853 0.081999 0.721852 0.657891 0.663255

Process 1: 0.885441 0.028577 0.038914 0.182672 0.292637 ... 0.170786 0.072244 0.340493 0.022609 0.787708

Process 0: 0.031431 0.618055 0.240052 0.867846 0.361785 ... 0.530320 0.495350 0.257075 0.416821 0.404568

Sorted
Process 9: 0.900094 0.900753 0.903000 0.903439 0.903885 ... 0.998011 0.999050 0.999156 0.999857 0.999859
Sorted
Process 2: 0.200525 0.200725 0.201369 0.203190 0.203986 ... 0.296973 0.298137 0.298693 0.298931 0.299711
Sorted
Process 3: 0.300451 0.300669 0.300825 0.302232 0.305221 ... 0.393253 0.393974 0.398517 0.398809 0.399229
Sorted
Process 4: 0.400210 0.401475 0.403118 0.403296 0.404568 ... 0.493042 0.493600 0.495350 0.496399 0.498592
Sorted
Process 5: 0.500697 0.501285 0.501382 0.503601 0.503812 ... 0.595144 0.595265 0.595340 0.596864 0.598698
Sorted
Process 7: 0.700622 0.701013 0.705432 0.706182 0.706600 ... 0.795072 0.796507 0.799050 0.799712 0.799856
Sorted
Process 8: 0.800435 0.800942 0.803220 0.803534 0.803604 ... 0.895827 0.896854 0.897738 0.899210 0.899879
Sorted
Process 0: 0.000192 0.001013 0.004894 0.004979 0.005353 ... 0.095273 0.095482 0.097239 0.098381 0.099927

Execution time: 0.005925 seconds
Sorted
Process 1: 0.101706 0.102036 0.102942 0.103106 0.105974 ... 0.193806 0.193880 0.194765 0.196015 0.198249
Sorted
Process 6: 0.600972 0.603111 0.605551 0.606072 0.606210 ... 0.695624 0.695908 0.697377 0.699304 0.699905
[tpatel44@mcs1 assignment2]$
```

N = 100000

```
[tpatel44@mcs1 assignment2]$ mpicc -O2 sortmpi.c -o sortmpi.x
[tpatel44@mcs1 assignment2]$ mpirun -np 10 ./sortmpi.x
Process 4: 0.927680 0.958658 0.461468 0.599949 0.817456 ... 0.445987 0.252594 0.893162 0.960404 0.907816

Process 3: 0.071593 0.047260 0.159618 0.782473 0.385583 ... 0.634547 0.049621 0.083776 0.602151 0.558035

Process 8: 0.858435 0.607451 0.668103 0.377050 0.049464 ... 0.673715 0.343943 0.217376 0.284043 0.089284

Process 5: 0.286467 0.372402 0.264570 0.920521 0.752282 ... 0.204430 0.113116 0.707653 0.506612 0.168661

Process 7: 0.501592 0.697510 0.868965 0.058816 0.119006 ... 0.691052 0.366684 0.551363 0.139163 0.140341

Process 2: 0.710349 0.131244 0.855061 0.459058 0.947944 ... 0.598786 0.343961 0.286254 0.913335 0.407187

Process 6: 0.141728 0.281390 0.562677 0.736881 0.181528 ... 0.901266 0.438424 0.455401 0.035159 0.042586

Process 9: 0.217682 0.520631 0.471174 0.698215 0.983674 ... 0.298503 0.161254 0.610240 0.919380 0.819589

Process 1: 0.851318 0.218609 0.550735 0.137993 0.514486 ... 0.212958 0.559533 0.935176 0.457395 0.150288

Process 0: 0.494814 0.308066 0.250472 0.820106 0.583446 ... 0.037788 0.491004 0.115868 0.456531 0.566981

Sorted
Process 8: 0.800010 0.800010 0.800014 0.800038 0.800070 ... 0.899956 0.899964 0.899980 0.899987 0.899995
Sorted
Process 2: 0.200002 0.200003 0.200008 0.200038 0.200044 ... 0.299946 0.299949 0.299951 0.299992 0.299996
Sorted
Process 5: 0.500013 0.500034 0.500037 0.500039 0.500042 ... 0.599949 0.599949 0.599973 0.599979 0.599987
Sorted
Process 3: 0.300019 0.300020 0.300037 0.300048 0.300062 ... 0.399902 0.399909 0.399927 0.399980 0.399980
Sorted
Process 7: 0.700005 0.700007 0.700022 0.700041 0.700048 ... 0.799936 0.799943 0.799954 0.799955 0.799974
Sorted
Process 4: 0.400009 0.400017 0.400027 0.400039 0.400052 ... 0.499958 0.499962 0.499968 0.499997 0.499997
Sorted
Process 0: 0.000005 0.000018 0.000021 0.000036 0.000043 ... 0.099949 0.099949 0.099963 0.099996 0.099999
Sorted
Process 6: 0.600001 0.600014 0.600019 0.600046 0.600049 ... 0.699971 0.699972 0.699974 0.699981 0.699995
Sorted
Process 9: 0.900051 0.900064 0.900069 0.900085 0.900101 ... 0.999941 0.999966 0.999978 0.999991 0.999995
Sorted
Process 1: 0.100008 0.100012 0.100014 0.100021 0.100043 ... 0.199943 0.199953 0.199964 0.199966 0.199979

Execution time: 0.003405 seconds
[tpatel44@mcs1 assignment2]$
```

N = 1000000

```
[tpatel44@mcs1 assignment2]$ mpicc -O2 sortmpi.c -o sortmpi.x
[tpatel44@mcs1 assignment2]$ mpirun -np 10 ./sortmpi.x
Process 1: 0.548366 0.937540 0.618248 0.469881 0.849463 ... 0.119851 0.172841 0.562335 0.218114 0.131135

Process 3: 0.264488 0.764611 0.225704 0.609471 0.718364 ... 0.736881 0.259213 0.487100 0.091432 0.000504

Process 5: 0.482868 0.592626 0.331009 0.251444 0.588777 ... 0.202702 0.830782 0.678752 0.684483 0.103782

Process 2: 0.402902 0.349597 0.419522 0.285428 0.281983 ... 0.638575 0.473751 0.216144 0.802013 0.388186

Process 0: 0.690196 0.525396 0.816006 0.149985 0.416515 ... 0.389814 0.083357 0.641637 0.542848 0.680493

Process 9: 0.920825 0.250406 0.548441 0.537279 0.831269 ... 0.896883 0.598288 0.779812 0.835497 0.134973

Process 7: 0.199534 0.917668 0.935005 0.891520 0.455622 ... 0.162355 0.580272 0.426927 0.429197 0.021201

Process 6: 0.842018 0.508211 0.134796 0.072359 0.525740 ... 0.487556 0.780861 0.396286 0.558223 0.418244

Process 8: 0.559129 0.333154 0.238981 0.712987 0.892505 ... 0.613424 0.864550 0.879862 0.470820 0.489082

Process 4: 0.126168 0.181012 0.029772 0.433324 0.656531 ... 0.160002 0.351854 0.548082 0.357238 0.600123

Sorted
Process 5: 0.500000 0.500000 0.500000 0.500000 0.500000 ... 0.600000 0.600000 0.600000 0.600000 0.600000
Sorted
Process 7: 0.700000 0.700000 0.700000 0.700000 0.700000 ... 0.800000 0.800000 0.800000 0.800000 0.800000
Sorted
Process 1: 0.100000 0.100000 0.100000 0.100000 0.100000 ... 0.200000 0.200000 0.200000 0.200000 0.200000
Sorted
Process 6: 0.600000 0.600000 0.600000 0.600000 0.600000 ... 0.700000 0.700000 0.700000 0.700000 0.700000
Sorted
Process 3: 0.300000 0.300000 0.300000 0.300000 0.300000 ... 0.400000 0.400000 0.400000 0.400000 0.400000
Sorted
Process 8: 0.800000 0.800000 0.800000 0.800000 0.800000 ... 0.900000 0.900000 0.900000 0.900000 0.900000
Sorted
Process 9: 0.900000 0.900000 0.900000 0.900000 0.900000 ... 1.000000 1.000000 1.000000 1.000000 1.000000
Sorted
Process 4: 0.400000 0.400000 0.400000 0.400000 0.400000 ... 0.500000 0.500000 0.500000 0.500000 0.500000
Sorted
Process 2: 0.200000 0.200000 0.200000 0.200000 0.200000 ... 0.300000 0.300000 0.300000 0.300000 0.300000
Sorted
Process 0: 0.000000 0.000000 0.000000 0.000000 0.000000 ... 0.100000 0.100000 0.100000 0.100000 0.100000

Execution time: 5.166166 seconds
[tpatel44@mcs1 assignment2]$
```

N = 100,000,000

```
[tpatel44@mcs1 assignment2]$ mpicc -O2 sortmpi.c -o sortmpi.x
[tpatel44@mcs1 assignment2]$ mpirun -np 10 ./sortmpi.x
Process 3: 0.304218 0.729111 0.471596 0.736879 0.039880 ... 0.628884 0.467036 0.901065 0.480086 0.330297

Process 9: 0.952711 0.707437 0.786031 0.155303 0.143821 ... 0.728371 0.974342 0.220137 0.971917 0.053571

Process 2: 0.943345 0.313819 0.166160 0.913763 0.103200 ... 0.180776 0.673914 0.043099 0.625606 0.180272

Process 8: 0.596293 0.295441 0.987499 0.837790 0.710960 ... 0.822621 0.840831 0.562424 0.074202 0.866931

Process 0: 0.232434 0.993966 0.068964 0.280657 0.242538 ... 0.627160 0.075212 0.284093 0.161505 0.773177

Process 7: 0.237095 0.383098 0.683160 0.016501 0.777829 ... 0.325037 0.191509 0.748738 0.662892 0.789425

Process 6: 0.380470 0.472558 0.882235 0.698411 0.846830 ... 0.834492 0.482178 0.169493 0.359258 0.111229

Process 4: 0.661386 0.142195 0.273283 0.555472 0.973864 ... 0.529830 0.383833 0.049536 0.541555 0.370377

Process 5: 0.017494 0.552392 0.571512 0.372738 0.904603 ... 0.209665 0.943282 0.851999 0.375326 0.570533

Process 1: 0.084984 0.402651 0.865610 0.093713 0.171244 ... 0.638456 0.847538 0.026447 0.481667 0.349886

Sorted
Process 6: 0.600000 0.600000 0.600000 0.600000 0.600000 ... 0.700000 0.700000 0.700000 0.700000 0.700000
Sorted
Process 7: 0.700000 0.700000 0.700000 0.700000 0.700000 ... 0.800000 0.800000 0.800000 0.800000 0.800000
Sorted
Process 3: 0.300000 0.300000 0.300000 0.300000 0.300000 ... 0.400000 0.400000 0.400000 0.400000 0.400000
Sorted
Process 8: 0.800000 0.800000 0.800000 0.800000 0.800000 ... 0.900000 0.900000 0.900000 0.900000 0.900000
Sorted
Process 0: 0.000000 0.000000 0.000000 0.000000 0.000000 ... 0.100000 0.100000 0.100000 0.100000 0.100000
Sorted
Process 9: 0.900000 0.900000 0.900000 0.900000 0.900000 ... 1.000000 1.000000 1.000000 1.000000 1.000000
Sorted
Process 5: 0.500000 0.500000 0.500000 0.500000 0.500000 ... 0.600000 0.600000 0.600000 0.600000 0.600000
Sorted
Process 1: 0.100000 0.100000 0.100000 0.100000 0.100000 ... 0.200000 0.200000 0.200000 0.200000 0.200000
Sorted
Process 4: 0.400000 0.400000 0.400000 0.400000 0.400000 ... 0.500000 0.500000 0.500000 0.500000 0.500000
Sorted
Process 2: 0.200000 0.200000 0.200000 0.200000 0.200000 ... 0.300000 0.300000 0.300000 0.300000 0.300000

Execution time: 34.328837 seconds
[tpatel44@mcs1 assignment2]$
```

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

```
[tpatel44@mcs1 assignment2]$ scl enable devtoolset-7 bash
[tpatel44@mcs1 assignment2]$ gcc -fopenmp -O2 sumopenmp.c -o sumopenmp.x -std=c99
[tpatel44@mcs1 assignment2]$ ./sumopenmp.x
Block size: 64, Threads: 4, N: 1000
Serial computation time: 0.013439 seconds
OpenMP tasks computation time: 3.159977 seconds
OpenMP tasks with blocks computation time: 0.025182 seconds
All matrices match!

Block size: 64, Threads: 8, N: 1000
Serial computation time: 0.002298 seconds
OpenMP tasks computation time: 6.776396 seconds
OpenMP tasks with blocks computation time: 0.025987 seconds
All matrices match!

Block size: 64, Threads: 16, N: 1000
Serial computation time: 0.002207 seconds
OpenMP tasks computation time: 8.413126 seconds
OpenMP tasks with blocks computation time: 0.052810 seconds
All matrices match!

Block size: 64, Threads: 32, N: 1000
Serial computation time: 0.002203 seconds
OpenMP tasks computation time: 11.824622 seconds
OpenMP tasks with blocks computation time: 0.020328 seconds
All matrices match!

Block size: 64, Threads: 64, N: 1000
Serial computation time: 0.002558 seconds
OpenMP tasks computation time: 13.320402 seconds
OpenMP tasks with blocks computation time: 0.038985 seconds
All matrices match!
```

N = 2000

```
[tpatel44@mcs1 assignment2]$ gcc -fopenmp -O2 sumopenmp.c -o sumopenmp.x -std=c99
[tpatel44@mcs1 assignment2]$ ./sumopenmp.x
Block size: 64, Threads: 4, N: 2000
Serial computation time: 0.007296 seconds
OpenMP tasks computation time: 11.956034 seconds
OpenMP tasks with blocks computation time: 0.004848 seconds
All matrices match!

Block size: 64, Threads: 8, N: 2000
Serial computation time: 0.007464 seconds
OpenMP tasks computation time: 13.843703 seconds
OpenMP tasks with blocks computation time: 0.002981 seconds
All matrices match!

Block size: 64, Threads: 16, N: 2000
Serial computation time: 0.007745 seconds
OpenMP tasks computation time: 13.287195 seconds
OpenMP tasks with blocks computation time: 0.005959 seconds
All matrices match!

Block size: 64, Threads: 32, N: 2000
Serial computation time: 0.008470 seconds
OpenMP tasks computation time: 27.779879 seconds
OpenMP tasks with blocks computation time: 0.006517 seconds
All matrices match!

Block size: 64, Threads: 64, N: 2000
Serial computation time: 0.009419 seconds
OpenMP tasks computation time: 29.347066 seconds
OpenMP tasks with blocks computation time: 0.009496 seconds
All matrices match!
```

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.