PDP Assignment 1

Arvid Ehrengren, Viktor Kangasniemi

April 4, 2025

1 Answers to all tasks in A1

2.1

When p is divisible by n:

$$\text{First index: } k * \frac{n}{p}$$

$$\text{last index: } ((k+1) * \frac{n}{p}) - 1$$

When p is not divisible by n:

$$\text{first index: } k * \frac{n}{p} \tag{1}$$

$$\text{last index: } (k * \frac{n}{p}) + (n\, mod\, p) - 1$$

2.2

Strong Scalability

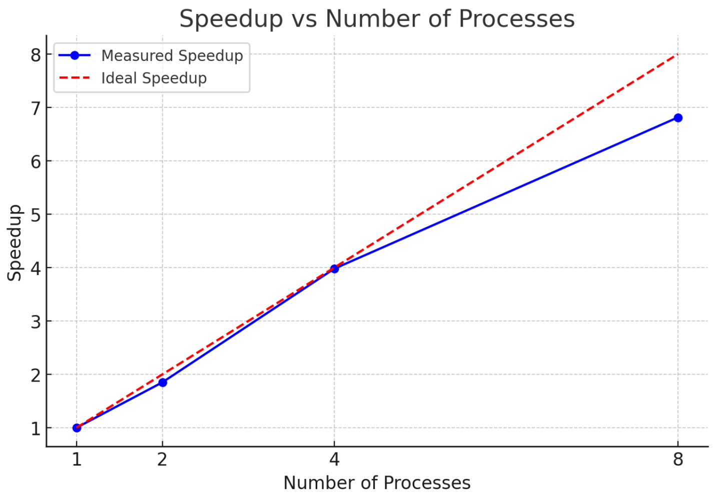

Figure 1: Speedup comparison between 1,2,4 and 8 processes for $n = 2^{22}$

Processes	Time (s)	Speedup
1	0.002869	1.00
2	0.001550	1.85
4	0.000721	3.98
8	0.000421	6.81

Table 1: Execution times and speedups for $n = 2^{22}$

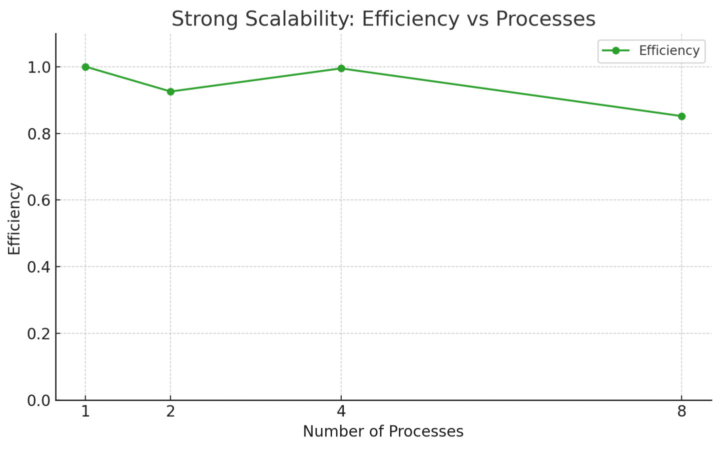

Figure 2: Efficiency comparison between 1,2,4 and 8 processes for $n = 2^{22}$

Weak Scalability

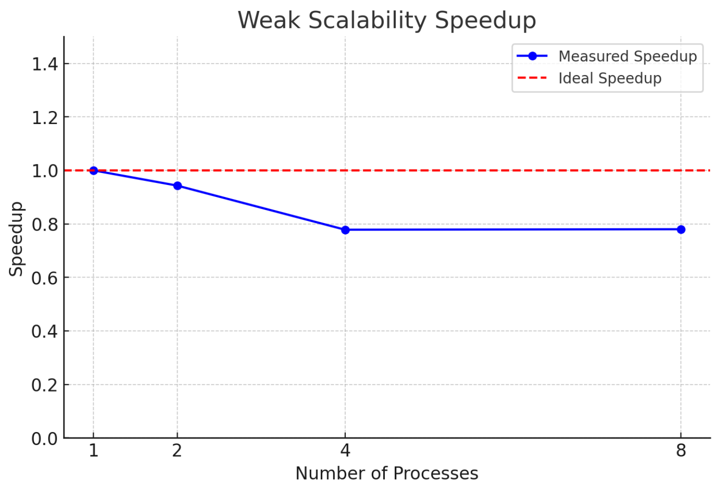

Figure 3: Speedup comparison between 1,2,4 and 8 processes for n increasing with 2^{19} per process.

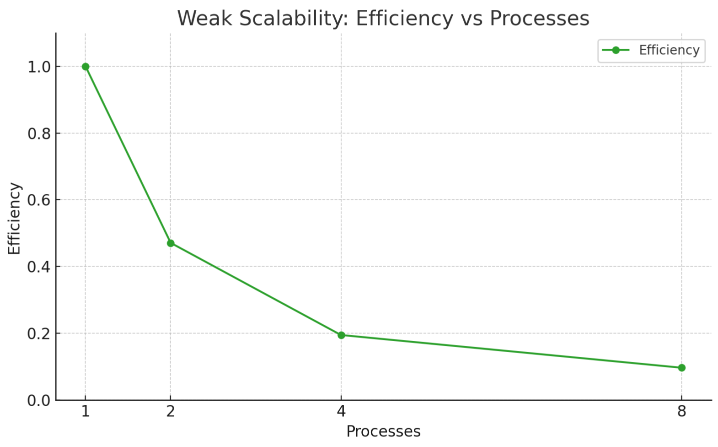

Figure 4: Efficiency comparison between 1,2,4 and 8 processes for n increasing with 2^{19} per process.

Processes	Time (s)	Speedup
1	0.000347	1.00
2	0.000368	0.94
4	0.000437	0.79
8	0.000435	0.80

Table 2: Speedup comparison between 1,2,4 and 8 processes for n increasing with 2^{19} per process.

2.3

As the number of processes increases, the amount of adds and recives that process 0 performs will be equal to log2(p) always rounded upward, where p is the number of processes. This happens because for each iteration of the reduction tree, the number of active processes are halved. Thus giving us the final time complexity formula:

$$T(p) = log_2(p) \cdot r + log_2(p) \cdot a \tag{2}$$

2.4

The functions

$$T_s(n) = n^2$$

$$T_p(n,p) = \frac{n^2}{p} + \log_2(p)$$

$$\text{Speedup: } S(n,p) = \frac{T_s(n)}{T_p(n,p)} = \frac{n^2}{\frac{n^2}{p} + \log_2(p)}$$

$$\text{Efficiency: } E(n,p) = \frac{S(n,p)}{p} = \frac{n^2}{p\left(\frac{n^2}{p} + \log_2(p)\right)} = \frac{n^2}{n^2 + p\log_2(p)}$$

where:

- $T_s(n)$ is the sequential execution time
- $T_p(n,p)$ is the parallel execution time
- n is the problem size
- p is the number of processors

are used to produce Table 3 showcase speed up and Table 4 the efficiency for n = 10, 20, 40, 80, 160, 320 and p = 1, 2, 4, 8, 16, 32, 64, 128.

p \n	10	20	40	80	160	320
1	1.00	1.00	1.00	1.00	1.00	1.00
2	1.96	1.99	2.00	2.00	2.00	2.00
4	3.70	3.92	3.98	4.00	4.00	4.00
8	6.45	7.55	7.88	7.97	7.99	8.00
16	9.76	13.79	15.38	15.84	15.96	15.99
32	12.31	22.86	29.09	31.22	31.80	31.95
64	13.22	32.65	51.61	60.38	63.05	63.76
128	12.85	39.51	82.05	112.28	123.67	126.89

Table 3: Speedup Table

When the problem size is fixed and the number of processors is increasing, the speedup improves and efficiency decreases as the fixed workload is divided on more processors. This occurs because the overhead exceeds the actual computational workload. On the other hand, given an fixed amount of processes with increasing problem size, the speed up and efficiency are both increasing as more work is present for the processors to preform.

p \n	10	20	40	80	160	320
1	1.00	1.00	1.00	1.00	1.00	1.00
2	0.98	1.00	1.00	1.00	1.00	1.00
4	0.93	0.98	1.00	1.00	1.00	1.00
8	0.81	0.94	0.99	1.00	1.00	1.00
16	0.61	0.86	0.96	0.99	1.00	1.00
32	0.38	0.71	0.91	0.98	0.99	1.00
64	0.21	0.51	0.81	0.94	0.99	1.00
128	0.10	0.31	0.64	0.88	0.97	0.99

Table 4: Efficiency Table

2.5

To know what factor we have to increase n with, we can solve for x in the following algebraic relation:

$$E(n,p) = E(x \cdot n, k \cdot p) \tag{3}$$

where k >1.

$$\frac{n}{n + p\log_2(p)} = \frac{x \cdot n}{x \cdot n + k \cdot p \cdot log_2(k \cdot p)}$$

$$\frac{n}{n + p\log_2(p)} = \frac{x \cdot n}{x \cdot n + k \cdot p \cdot log_2(k \cdot p)}$$

$$n\left(x \cdot n + k \cdot p\log_2(k \cdot p)\right) = x \cdot n\left(n + p\log_2(p)\right)$$

$$x \cdot n^2 + n \cdot k \cdot p\log_2(k \cdot p) = x \cdot n^2 + x \cdot n \cdot p\log_2(p)$$

$$n \cdot k \cdot p\log_2(k \cdot p) = x \cdot n \cdot p\log_2(p)$$

$$k\log_2(k \cdot p) = x\log_2(p)$$

$$k\left(\log_2(k) + \log_2(p)\right) = x\log_2(p)$$

$$k\log_2(k) + k\log_2(p) = x\log_2(p)$$

$$x\log_2(p) = k\log_2(k) + k\log_2(p)$$

$$x = k \cdot \left(1 + \frac{\log_2(k)}{\log_2(p)}\right)$$

After solving x, we can see that we need to increase n with $x = k \cdot \left(1 + \frac{\log_2(k)}{\log_2(p)}\right)$ to maintain the same efficiency.

2 Description of your parallelization

The program, from **2.2**, sums all the integers from the array by distributing the work across multiple processes and combines the results using a tree-based reduction algorithm. It works by initially dividing the array in to equal sized blocks,in the root process (rank 0) and then distributes the blocks to all of the processes evenly through MPI_Scatterv. If the the number of elements in the array is not divisible by the number of processes, the remainder elements are distributed one by one to processes until all have been assigned. After the distribution of work is done, each process computes the sum of its local array. Then the reduction phase begins, which uses a logarithmic communication pattern to combine the results. In this phase, the processes are paired in each iteration, where processes with higher ranks send their computed sum to processes with lower ranks, until the final sum is computed in rank 0. In each iteration of this process, the number of active processes are halved, giving a complexity of $O(log(n))$.

3 Description performance experiments

The performance experiments were conducted by running the summation program under varying configurations of processes and array sizes. A bash script was used to automate the process of running the program with different parameters using `mpirun`.

For strong scaling, the array size was fixed at $n = 2^{22}$ while the number of processes was varied between 1, 2, 4, and 8. This allowed us to see how increasing the number of processors affects the execution time of a fixed workload.

For weak scaling, the size of the array ($n = 2^{19}$) was increased proportionally with the number of processes to maintain a constant workload per processor. This lets us evaluate the programs efficiency when the problem size scales with the number of processors.

4 results from performance experiemnets

Evident from the results, the program scales well when we increase the number of processes. The speedup becomes better as more processes are added, but the improvements slow down a bit due to the extra overhead.

In strong scaling, the speedup increases when more processes are used. The efficiency decreases, but is still relatively high. Overall, the performance follows what we would expect from theory and shows that the parallelization strategy is working well.

For weak scaling, which is shown in Table 2, the total execution time stays almost the same even as we add more processes. This is expected as we're increasing the workload proportionally with each process used, which ideally should result in the same execution time regardless of how many processes are used. We can also see that efficiency decreases as we increase the number of processes. This is expected, as with more processes, there's more overhead.

5 Discussion of results

The results match what we expected. As seen in Figure 1 for strong scaling, the speedup is almost linear, showing that the work is well divided and overhead is low. The tree-based reduction helps keep communication fast and efficient.However, efficiency as seen in Figure 2 decreases as more processes are added. This is due to the overhead introduced by communication and coordination, the reduced workload per process leading to idle time.

For weak scaling, the execution time stays about the same as more processes are added which can be seen in Figure 3. But efficiency drops a bit because due to the added overhead displayed in Figure 4.

Overall, the program scales well. The use of MPI_Scatterv and tree-based reduction keeps the workload balanced, ultimately speeding up the process.