Uppsala University

Parallel and Distributed Programming

1TD070

Title

Author:
Aron Lif
Johan Andersson Östling

April 4, 2025

1 block partitioning

compute the sum of an array of length n using p processing units. Partition the array into p blocks of near uniform size.

- n = array length
- p = processing units
- k = block number where $(0 \leq k < p)$
- r = $n \mod p$, (total number of extra elements to distribute)

$$\text{start(k)} = k \cdot \left\lfloor \frac{n}{p} \right\rfloor + \min(k, r)$$

$$\text{end(k)} = \text{start}(k) + \left\lfloor \frac{n}{p} \right\rfloor + \begin{cases} 1 & \text{if } k < r \\ 0 & \text{otherwise} \end{cases} - 1$$

2 Tree-structured global sum

2.1 creating a algorithm

We needed to create a tree-structured global sum algorithm that also should work for uneven processes. we came up with the code below.

```
61    for (int i = start; i < end; i++) {
62        local_sum += rand() % 100;
63    }
64
65    int active = 1;
66    while (active < size) {
67        if (rank % (2 * active) == 0) {
68            if (rank + active < size) {
69                int received;
70                MPI_Recv(&received, 1, MPI_INT, rank + active, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
71                local_sum += received;
72            }
73        } else {
74            int dest = rank - active;
75            MPI_Send(&local_sum, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
76            break;
77        }
78        active *= 2;
79    }
80
81
82    if (rank == 0) {
83        printf("Parallel Summation done\n");
84        double end_time = MPI_Wtime();
85        double elapsed = end_time - start_time;
86        printf("%f\n", elapsed);
87    }
```

Figure 1: Tree-structured global sum algorithm

1. each process calculates the local sum of random integers for a start and end interval
2. Initialize **active** variable. **active** represents the distance between processes that will exchange data in each step.
3. Check the process role in current step
    - **Receiver**: A process whose rank is evenly divisible by `2*active` is going to receive data from the process with a rank of `rank + active`

- **Sender** if a process is not evenly divisible by `2*active` it will send its local_sum to the process with a rank of `rank-active`. After this the process exits the loop. item **Uneven sender** When doing computations with uneven processes the check `if(rank+active<size)` ensures these processes simply sends their local sum to the closest reciver and exits

4. The process adds the incoming partial sum to its own **local_sum**
5. double the size of **active** to accommodate for the new length between active processes.
6. the **rank 0** process is always the last process standing having received all of the `local_sums` and added them together

2.2 Efficiency plots

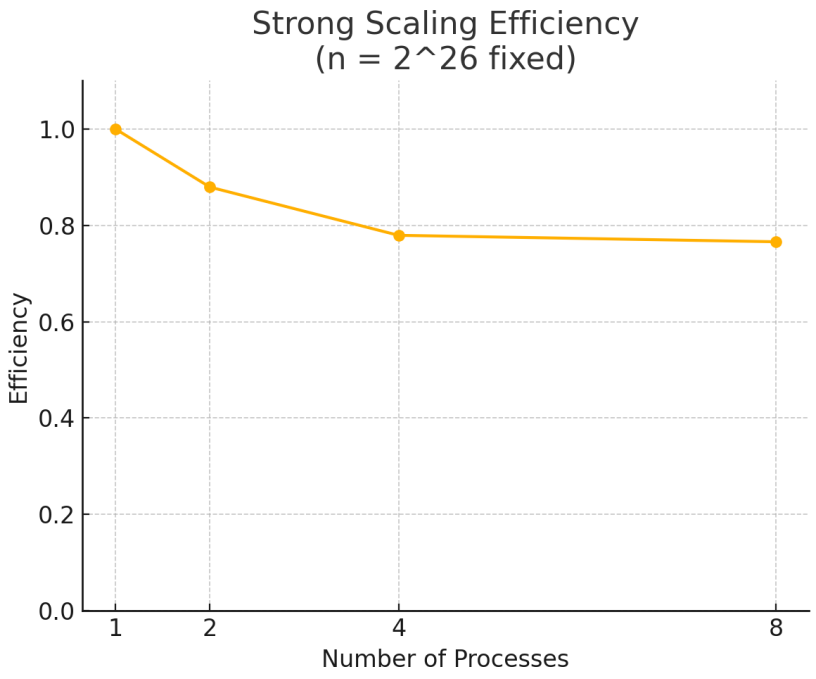

Figure 2: Strong Scaling Efficiency

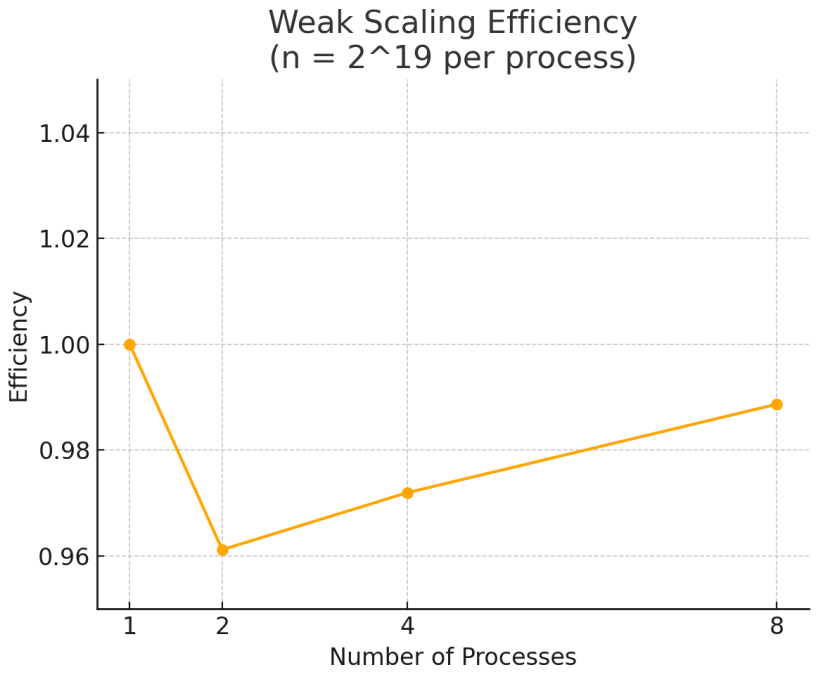

Figure 3: Weak Scaling Efficiency

3 Cost analysis of tree-structured global sum algorithm

In the tree-structured global sum algorithm with p processing units (where p is a power of two), the summation process is performed in $\log_2(p)$ stages. At each stage, half of the processes send their partial sums to the other half, which then perform additions to combine the received values.

Processing unit 0 is involved in all stages as the root of the reduction tree. In each stage, it performs:

- One **receive** operation to get data from another processing unit.
- One **addition** operation to add the received data to its current sum.

Therefore, the total number of operations performed by processing unit 0 is:

- $\log_2(p)$ receive operations.

- $\log_2(p)$ addition operations.

Let r be the time it takes to perform one receive, and a be the time for one addition. Then the total time $T(p)$ for processing unit 0 can be expressed as:

$$T(p) = \log_2(p) \cdot r + \log_2(p) \cdot a$$

4 Speed and efficiency

Threads	**10**	**20**	**40**	**160**	**320**
1	1.0000	1.0000	1.0000	1.0000	1.0000
2	1.9608	1.9900	1.9975	1.9998	1.9999
4	3.7037	3.9216	3.9801	3.9988	3.9997
8	6.4516	7.5472	7.8818	7.9925	7.9981
16	9.7561	13.7931	15.3846	15.9601	15.9900
32	12.3077	22.8571	29.0909	31.8012	31.9501
64	13.2231	32.6531	51.6129	63.0542	63.7609
128	12.8514	39.5062	82.0513	123.6715	126.8897

Table 1: Speedup across different thread counts and workloads

Threads	**10**	**20**	**40**	**160**	**320**
1	1.0000	1.0000	1.0000	1.0000	1.0000
2	0.9804	0.9950	0.9988	0.9999	1.0000
4	0.9259	0.9804	0.9950	0.9997	0.9999
8	0.8065	0.9434	0.9852	0.9991	0.9998
16	0.6098	0.8621	0.9615	0.9975	0.9994
32	0.3846	0.7143	0.9091	0.9938	0.9984
64	0.2066	0.5102	0.8065	0.9852	0.9963
128	0.1004	0.3086	0.6410	0.9662	0.9913

Table 2: Efficiency across different thread counts and workloads

4.1 Fixed problem size

Increasing the number of processing units for a fixed problem size initially improves the speed, but after a certain point, adding more processor yields diminishing returns. Eventually, the speed plateaus and then decreases for small problem sizes. The fixed amount has a limited parallel partition, which diminishes as the processor count increases, while the overhead increases.

4.2 Fixed processor count

With a fixed number of processing units, increasing the size of the problem means there is more work to be done, allowing the processors to be used more fully. The speedup and efficiency improves as the workload becomes large enough to effectively mask the overhead, approaching near-ideal performance.

5 Scalability

The efficiency is defined as:

$$E(n,p) = \frac{\text{Speedup}}{p} = \frac{T_s(n)}{p\,T_p(n,p)} = \frac{n}{p\left(\frac{n}{p} - \log_2(p)\right)} = \frac{n}{n - p\log_2(p)}.$$

Now, suppose we increase the number of processors from p to $k\,p$ (with $k > 1$) and scale the problem size by a factor i, so that $\tilde{n} = i\,n$. To maintain the same efficiency, we require:

$$E(n,p) = E(i\,n, k\,p).$$

Substituting the efficiency expressions:

$$\frac{n}{n - p\log_2(p)} \;=\; \frac{i\,n}{i\,n - k\,p\log_2(kp)}.$$

Cross multiply:

$$n\Big(i\,n - k\,p\log_2(kp)\Big) = i\,n\Big(n - p\log_2(p)\Big).$$

Divide both sides by n:

$$i\,n - k\,p\log_2(kp) = i\big(n - p\log_2(p)\big).$$

Expanding the right-hand side:

$$i\,n - k\,p\log_2(kp) = i\,n - i\,p\log_2(p).$$

Subtracting $i\,n$ from both sides:

$$-k\,p\log_2(kp) = -i\,p\log_2(p).$$

Dividing both sides by $-p$ (with $p > 0$):

$$k\,\log_2(kp) = i\,\log_2(p).$$

Finally, solving for i:

$$i = k\,\frac{\log_2(kp)}{\log_2(p)}.$$

6 Parallelization

The program uses MPI for parallelizing the summation in a Tree structured fashion. We have implemented both a Strong and a weak scaling for the summation. both the **Weak** and **Strong** scaling implementations uses the algorithm form `Section 2.1`. The **weak scaling** gives each process the same amount of work to be executed. The **Strong scaling** uses the block partitioning algorithm from `Task 1` for dividing the work between the processes.

7 Performance experiments

When measuring the performance we decided to have a timer that starts after initialization of the threads. We only measure the time for Thread 0. The timer ends when we all summations are done. For performance testing we wrote a script that ran the Strong scaling version with process counts of 1, 2, 4, 8 and for step sizes from 2^{22} to 2^{26}

8 Performance results

Raw execution time Speedup table

Processes	2^{22}	2^{23}	2^{24}	2^{25}	2^{26}
1	0.131120	0.262434	0.510198	0.973638	1.728149
2	0.074284	0.137316	0.262036	0.512010	0.919026
3	0.052022	0.093856	0.178701	0.344398	0.683810
4	0.040191	0.073563	0.136810	0.261569	0.510892
5	0.034552	0.063886	0.111137	0.210582	0.410221
6	0.028892	0.052389	0.095374	0.177777	0.346495
7	0.023647	0.045345	0.081812	0.153249	0.296257
8	0.020927	0.039281	0.075208	0.134593	0.263262

Table 3: Execution times (in seconds) for strong-scaling experiments.

Processes	2^{22}	2^{23}	2^{24}	2^{25}	2^{26}
1	1.00	1.00	1.00	1.00	1.00
2	1.77	1.91	1.95	1.90	1.88
3	2.52	2.80	2.86	2.83	2.53
4	3.26	3.57	3.73	3.72	3.38
5	3.80	4.11	4.59	4.62	4.21
6	4.54	5.01	5.35	5.48	4.99
7	5.54	5.79	6.24	6.36	5.83
8	6.27	6.68	6.78	7.23	6.56

Table 4: Speedup values for strong-scaling experiments (Time(1 process) / Time(p processes)).

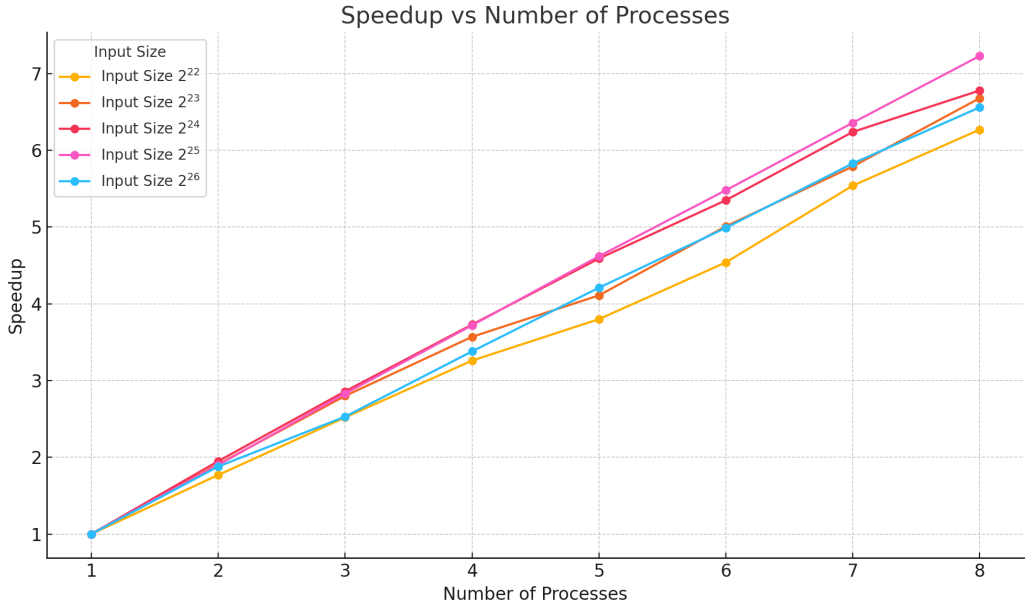

Figure 4: Speedup plot Strong scaling

9 Discussion of results

The result of speedup is as we expected. We can see that the speedup increases as we add processes to the execution of the program. We can also see that increasing the problem size generally also increases speedup. This is also expected since more work means that the overhead of communication and process creation will have less impact on the speedup. We expected the case where we ran the program with problems of 2^{26} to have the greatest speedup and where surprised that it didn't.