Assignment 1

group 14

Name: Houwan Liu, Linjing Shen

## 1. Formulas for block partitioning

* when p/n:

divide n into p blocks, block number = k, block size = n / p.

first indice(k) = k \* (n / p); last indice(k) = (k + 1) \* (n / p) - 1 = first indice(k + 1) - 1.

* when not p/n:

block size = [n / p], remainder = n - p \* [n / p].

first indice = k \* [n / p] + min(k, remainder); last indice = first indice(k + 1) - 1.

## 2. Tree-structured global sum

* Conclusion:

1, Based on the time result and plot, we can know that this code performs well both on strong scalability and weak scalability. Regarding strong scalability, we can see that when the number of processes grows, we can keep the efficiency constant without increasing the problem size. Regarding weak scalability, we can see we can keep the efficiency constant by increasing the problem size at the same rate as we increase the number of cores.

2, The sequential run time is shorter than the time of using MPI when process is 1, because even only one process involved, programs designed to run in parallel can also introduce unnecessary overhead due to steps such as initialization and termination of the parallel framework, distribution and collection of individual data points, and so on, which makes the time on parallel( process = 1) bigger than serial time.

3, when the processes number growing, the time will reduced at the same rate, which follow my initial expectation.

* “num\_procs” is the number of processes, and each core will be arranged a rank. In step one, the right core will send its value, and its left core will receive this value. While in step two, the amount of cores reduced to half, repeat the last calculation ….

When the process is equal to 1, and can't satisfy “i < num\_procs”, this part will be passed directly.

for (int i = 1; i < num\_procs; i \*= 2) {

if (rank % (2 \* i) == 0) {

if (rank + i < num\_procs) {

long long received\_sum;

MPI\_Recv(&received\_sum, 1, MPI\_LONG\_LONG, rank + i, 0, MPI\_COMM\_WORLD, &stat);

local\_sum += received\_sum;

}

} else {

int dest = rank - i;

MPI\_Send(&local\_sum, 1, MPI\_LONG\_LONG, dest, 0, MPI\_COMM\_WORLD);

break;

}

}

* When steps exceed the size, the value of core 0 (rank = 0) will be the final sum. Print the value when rank is equal to 0 as the final result.

if (rank == 0) {

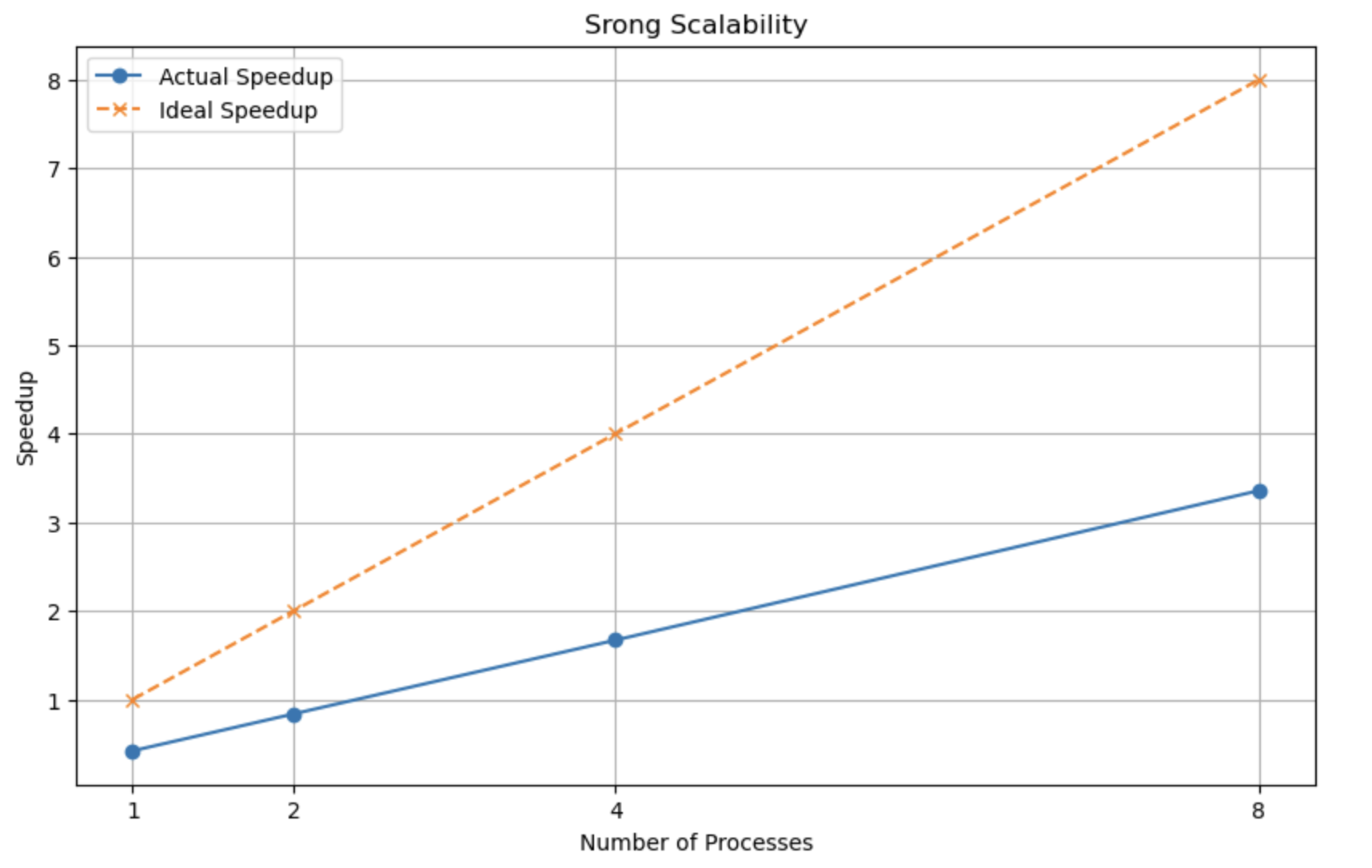
printf("Final sum = %lld\n", local\_sum);

printf("total time for %d process is %lf\n", num\_procs, end\_time-start\_time);

}

* Sequential run time: The best sequential algorithm time. I removed all parts related to MPI and changed the method to record run time.
* Speedup: S = Ts/Tp, where Ts is time to execute the best serial algorithm.
* Efficiency: E = Ts/(p\*Tp)
* strong scalability: problem size n = 2 ^22

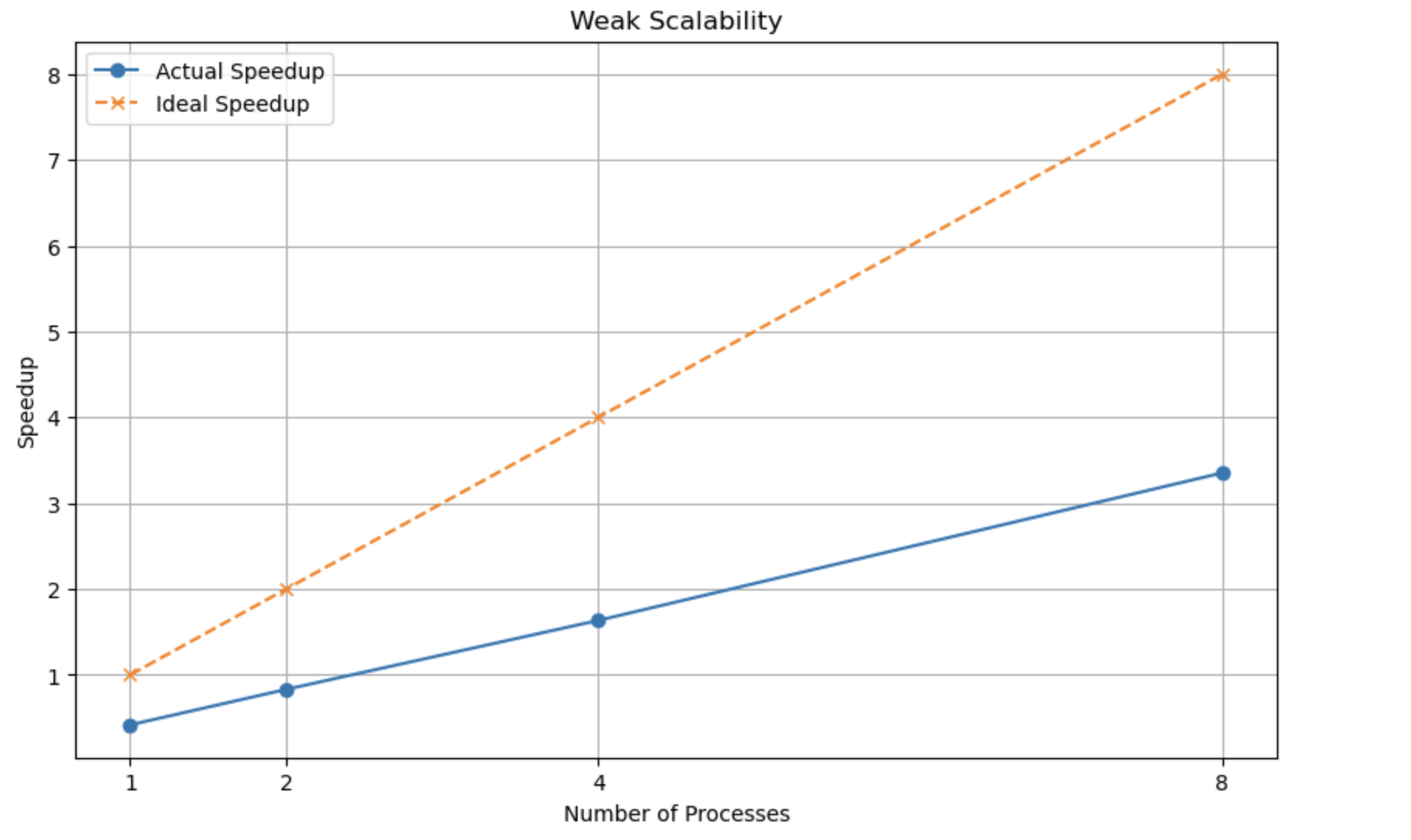
| num of proc | 1 | 2 | 4 | 8 |
| --- | --- | --- | --- | --- |
| time | 0.1422 | 0.0714 | 0.0358 | 0.0178 |
| sequential time | 0.0598 | 0.0598 | 0.0598 | 0.0598 |
| speedup | 0.4205 | 0.8375 | 1.6704 | 3.3596 |
| efficiency | 0.4205 | 0.4188 | 0.4176 | 0.4199 |



* weak scalability: problem size per process n = 2 ^ 19

—when evaluating weak scaling, I choose to modify total problem size to 2^19, 2^20…respectively, to make sure the problem size per process is always the same.

| num of proc | 1 | 2 | 4 | 8 |
| --- | --- | --- | --- | --- |
| time | 0.0177 | 0.0178 | 0.0178 | 0.0178 |
| sequential time | 0.0074 | 0.0148 | 0.0291 | 0.0597 |
| speedup | 0.4181 | 0.8315 | 1.6348 | 3.3539 |
| efficiency | 0.4181 | 0.4158 | 0.4087 | 0.4192 |



## 3. Cost analysis of tree-structured global sum algorithm

The “?” in this function should be log2(p), final function is T(p) = log2(p) \* r + log2(p) \* a, where p is the number of processes. The number of receives and additions that processing unit 0 both are log2(p), so the result is times of receives multiply time it takes to do one receive and times of additions multiply time it takes to do one addition.

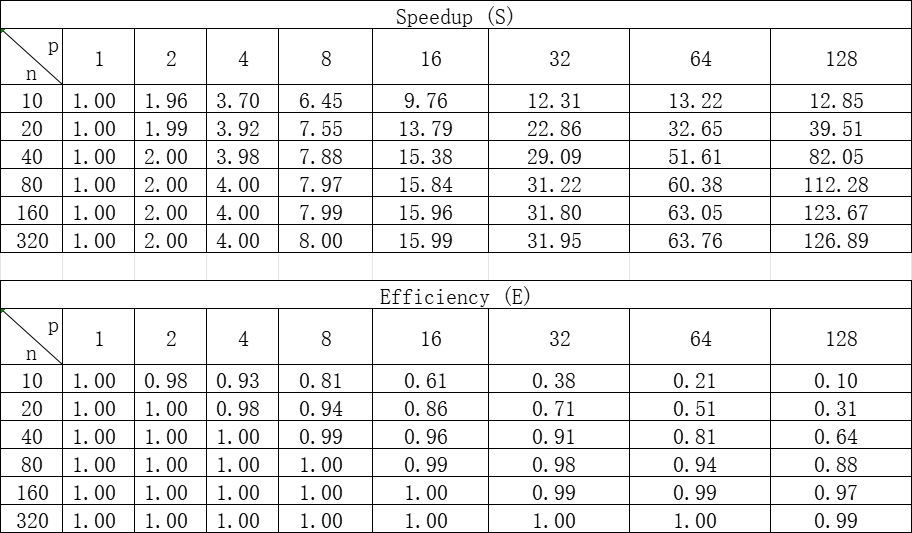
## 6. Speedup and efficiency

Ts(n) = n^2

Tp(n,p) = n^2/p + log2(p)

S(n,p) = Ts(n)/Tp(n,p)

E(n,p) = S(n,p)/p



In the case of a fixed problem size, as the number of processing units increases, the speed correspondingly improves because more processing units allow for more parallel distribution of workload, resulting in shorter execution times. However, with the increase in processing units, the efficiency decreases accordingly. This is reasonable because the denominator in the formula grows faster than the numerator with the increase in the number of processing units, likely due to the added overhead of parallelization, such as increased communication costs between processing units and potential load imbalance issues.

In the case of a fixed number of processing units, as the problem size increases, the speed likewise increases as larger problem sizes benefit more from parallelization, requiring more workload distribution among the processing units. However, with the increase in problem size, the efficiency correspondingly decreases. This is also reasonable because as the problem size grows, the parallelization-related overhead increases accordingly.

## 7. Scalability

Given that E(n,p) = Ts(n)/(Tp(n,p)\*p).

Based on the question, we increase p to k \* p, where k > 1. Suppose the new problem size is n', then the new parallel execution time is Tp(n', k \* p).

According to the same efficiency, we get the equation:

Ts(n) / (Tp(n, p) \* p) = Ts(n') / (Tp(n', k \* p) \* (k \* p))

Substitute the values of Ts(n) and Tp(n, p) into n':

n / ((n / p + log2(p)) \* p) = n' / ((n' / (k \* p) + log2(k \* p)) \* (k \* p))

Solving the equation, we get:

n' = n \* k \* (1 + log2(k) / log2(p))

Therefore, we need to increase n by **k\*(1+log2(k)/log2(p)) times** to maintain the same efficiency.