UPPSALA
UNIVERSITET

Assignment 1 - One-dimensional stencil application
Parallel and Distributed Programming

Author:

Junjie Chu

Hannes Kuchelmeister

Alexander Palfelt

Uppsala

2021-04-20

1 Parallellization

Suppose we will use *size* processes in the program and there are *numvalues* elements in the stencil in total. The root process(rank 0) is used to generate and deliver the elements. It also joins in computing. This parallel code will be in a manager/workers mode. To parallelize the code, we need to deliver the elements to different processes. Each process will get *sendcount* elements. $sendcount = \frac{numvalues}{size}$.

To implement that, MPIScatter is used. This is a blocking function. The root process will deliver the same number of elements to all processes. These elements will be stored in a local input array and the result of computing will be stored in a local output array.

To compute the stencil, we give the local input array and local output array extra 4 elements. 2 are at the beginning of the array and another 2 are at the end of the array. The extra array elements are needed for the stencil application on neighboring nodes. Each process will be given a left neighbor and a right neighbor via the following codes:

```
// Define left and right process
int left, right;
left = myid -1;
if (left < 0){left = size -1;}
right = myid + 1;
if (right > size -1){right = 0;}
```

The first 2 extra elements come from the left neighbor. And the last 2 extra elements come from the right neighbor. Since the following computing is based on the data, we need to make sure all the data have been sent and received. Thus, we will use blocking 'send' and blocking 'receive' here. To avoid 'deadlock', we choose to use MPISendrecv. The code are as follows.

```
MPI_Sendrecv(localinput+2, 2, MPI_DOUBLE, left, 1,
localinput+send_count+2, 2, MPI_DOUBLE, right, 1,
MPI_COMM_WORLD, &status);

MPI_Sendrecv(localinput+send_count, 2, MPI_DOUBLE, right, 0,
localinput, 2, MPI_DOUBLE, left, 0,
MPI_COMM_WORLD, &status);
```

The first MPISendrecv means that the current process will send 2 elements to its left process and receive 2 elements from its right process. The elements sent are 'localinput+2' and 'localinput+3'. They will be stored in the last 2 positions in the local input array of the left process. The elements received will be stored in the last 2 positions in the local input array of the current process. The second MPISendrecv means that the current process will send 2 elements to its right process and receive 2 elements from its left process. The elements sent are 'localinput+sendcount' and 'localinput+sendcount+1'. They will be stored in the first 2 positions in the local input array of the right process. The elements received will be stored in the first 2 positions in the local input array of the current process.

After the communication is completed, the computation starts. Each process has sendcount+4 elements. And the elements we need to update are from the third to the bottom third. To make sure all the processes are in the same time step, an MPIBarrier is set at the end of each time step.

The computing of these elements which need to be updated is only based on the elements in the current process.

Thus, the code is as follows:

```
// Apply stencil on points
// No need to do periodic computing
for (int i=2; i<send_count+2; i++) {
            double result = 0;
            for (int j=0; j<STENCIL_WIDTH; j++) {
                    int index = i - EXTENT + j;
                    result += STENCIL[j] * localinput[index];
            }
            localoutput[i] = result;
}
```

Like the Scatter, Gather is used to collect the data from the other processes and store them in the root process.In addition, a MPIReduce with MPIMax is used to find the longest execution time.

```
//Gather data
MPI_Gather(localoutput+2,send_count,MPI_DOUBLE,
output,send_count,MPI_DOUBLE,0,
MPI_COMM_WORLD);
// Find the longest time
MPI_Reduce( &my_execution_time, &max_execution_time, 1,
MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
```

We also measure the communication time and stencil time in one timestep using a laptop with 4 cores. The result is as follows:

```
junjie-chu@junjiechu-VirtualBox:~/Desktop/PDP/A1/solution_A1$ mpirun -np 4 stenc
il 1000000 3

number of processes: 4
number of data in each process: 250000

Communication time in last timestep: 0.000025

Stencil time in last timestep: 0.000892
```

Figure 1: Time measurement

The communication time could not be ignored. When using more than 1 node or more processes, the communication time will increase. The communication time will affect the performance of the parallel program and we will investigate it in more detail in the discussion part.

2 Verification of correctness

To verify the correctness of the parallelization, the output file of the provided serial program and the output file of the parallel program can be compared. This is done using the $diff$-command, and it is apparent that there is no difference in the output files.

We can further verify the correctness by comparing the input and output files of the parallelized program. The function that is used for the input is set to $sin(x)$, which is seen in the left plot of Figure 2. The output, showing the result of applying the stencil once on the input, is seen in the right plot. The right plot has the shape of $cos(x)$, which is expected since it is the first derivative of the input function.

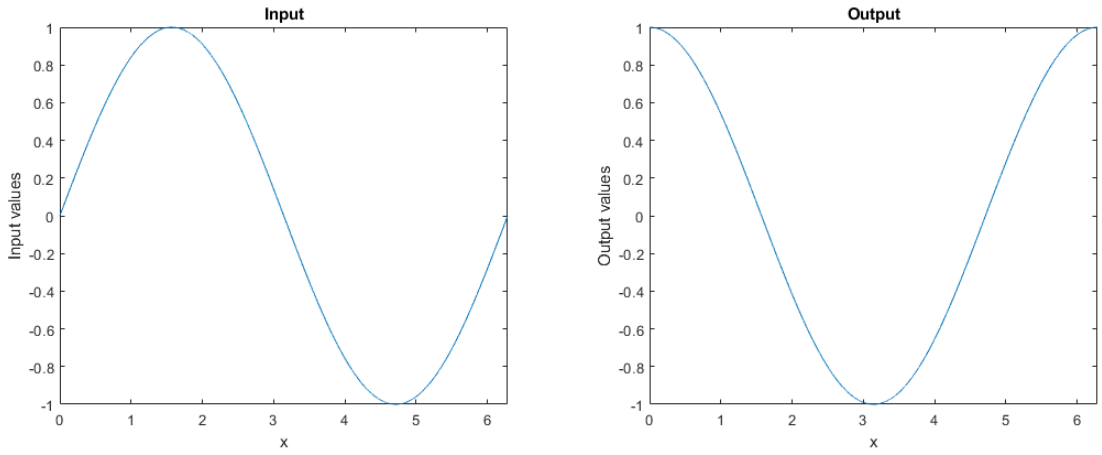

Figure 2: Left: Input values, with f(x) = sin(x). Right: Output values after applying the stencil once.

3 Performance experiments

3.1 Weak scalability

The weak scalability of the system is shown in Table 1 and Figure 3. The benchmark for weak scalability was run on the student computer with `mpirun --hostfile nodes -np 8 stencil 800000000 2`. The hostfile contains the nodes *arrhenius*, *atterbom*, *barany*, *gullstrand*, *berzelius*, *cronstedt*, *enequist*, *fredholm*, *hedenius*, *myrdal*, *siegbahn* and *trygger*.

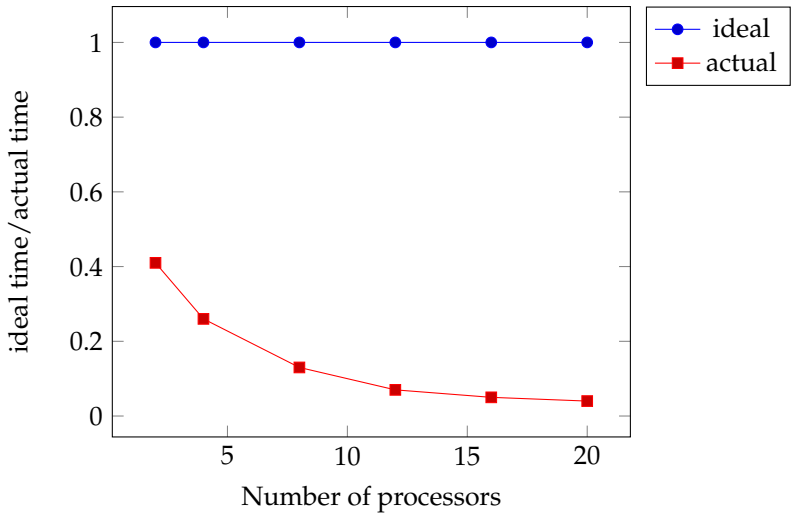

Figure 3: $\frac{\text{ideal parallel time}}{\text{actual parallel time}}$ by number of processors when the size of the problem is increased at the same rate as the number of processors (weak scalability)

Table 1: Weak scalability of the program. The asterisk (*) stands for the sequential version.

Processes	Problem size ($\times 10^8$)	Time (s)	$\frac{\text{ideal parallel time}}{\text{actual parallel time}}$
1*	1	7.42	1
1	1	12.7	0.62
2	2	18.13	0.41
4	4	28.96	0.26
8	8	56.81	0.13
12	12	102.12	0.07
16	16	149.71	0.05
20	20	193.02	0.04

3.2 Strong scalability

The strong scalability of the stencil application is shown in Figure 4, and Table 2. The program was run on the Linux servers, using `mpirun --bind-to-none --hostfile nodes -np x stencil 800000000 2`. The hostfile contained the same nodes as in the weak scalability experiment. The speedup is calculated as $\frac{\text{Maximum stencil application time}}{\text{Serial stencil application time}}$, where the maximum stencil application time is the largest time for any of the processors to finish the stencil application in the parallel program.

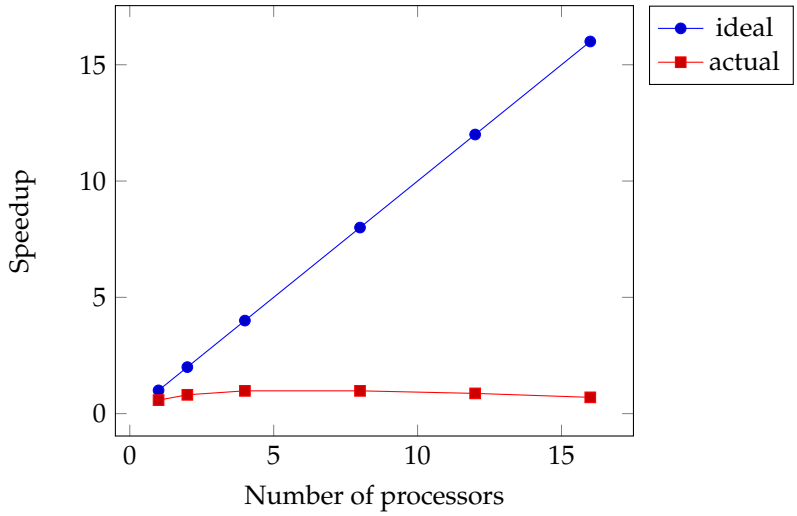

Figure 4: Strong scaling of the stencil application part. numvalues = 8×10^8, numsteps = 2.

Processes	Time (s)	Speedup
1*	7.41	1
1	12.69	0.58
2	9.17	0.81
4	7.54	0.98
8	7.57	0.98
12	8.55	0.87
16	10.55	0.7
20	10.18	0.73
24	10.41	0.71
28	10.47	0.71
32	10.73	0.69

Table 2: Maximum stencil application time, and speedup for different amount of processes used. The problem size is fixed to numvalues = 8×10^8, numsteps = 2.

4 Discussion

The weak scalability of our algorithm is not ideal. According to *Gustafson-Barsi's law* $S = p - (p-1)(1-fp)$, fp is the proportion of parallel part. In a very ideal situation, if the proportion of serial part is nearly 0, the ideal speedup is equal to p. And the running time is nearly the same. But in our case, due to the main overhead being communication, the serial part cannot be ignored and the running time is much longer than that of the ideal situation. Larger problems take much longer to distribute and therefore when increasing problem size and the number of processors at the same time, the communication time increases a lot and we end up with scaling that is not linear.

Our application does not scale very well either when looking at strong scaling. This is in part because of the parallel version (on one processor) being only half as fast as the sequential version but also due to the non-optimal scaling possibly due to communication and sequential parts. We can notice a performance increase up to eight processes, this is most likely because the machine we are using has in total 8 cores (+ hyperthreading). Hyperthreading does not seem to help with our scaling as it seems to already degrade our performance to similar levels as when we are using a second machine too.

Overall we notice that our program scales not ideal. When measuring the time spent by nodes on actual computations it stays rather constant however as the problem size increases so do the amount of data that needs to be send from one processor to another. This can be explained by *Amdahl's law* $S = \frac{1}{(1-p)+\frac{p}{s}}$. The parallel part of our program p benefits from more processors however as we have a communication part $(1-p)$ that does not benefit from parallelism we end up with a limited speedup. Moreover, due to the nature of distributed systems, the communication time even increases with more processors or more data being used.