Uppsala University

Parallel and Distributed Programming

1TD070

Project - report

Authors:
Csongor Horváth

April 11, 2023

The problem setting

[1] In applications such as computer simulations and digital image processing, an operation that is commonly needed is a stencil application. This operation is often performed a large number of times, and the performance of the whole program is highly dependent on it. A stencil can be seen as an operator that can be applied on the elements of a vector, matrix or tensor (a "multi-dimensional matrix"). When applying the stencil on an element, you compute a new value using the current value of the element in question, and its neighbors.
In this assignment, we are going to apply a one-dimensional stencil on an array of elements representing function values $f(x)$ for a finite set of N values $x_0, x_1, x_2, \ldots x_{N-1}$, residing on. Applying the stencil on an the interval $0 \leq x < 2 \cdot \pi$. Here, each value $x_i = i \cdot h,\ h = \frac{2 \cdot \pi}{N}$ element v_0 representing the function value $f(x_i)$. Applying the stencil simply means computing the sum:[2]

$$\frac{1}{12h}v_{-2} - \frac{8}{12h}v_{-1} + 0 \cdot v_0 + \frac{8}{12h}v_{+1} - \frac{1}{12h}v_{+2}$$

where v_{-j} is the element j steps to the left of v_0 that is, $f(x_{i-j})$. Likewise, v_{+j} is the element j steps to the right of v_0 $(f(x_{i+j})$. [3]

A code for serial implementation was given. My task was to change it to multi thread usage with the help of the OpenMPI module. All the provided codes and my parallelized program code can be found in this documents appendix.

[1] The problem setting section was written based on the assignment description
[2] Note that this weights gives a numerical approximation of the derivative
[3] If the indexes are not in $[0, N-1]$, then we use the points periodically

The description of the parallelization

Now I will discussed what modification I made during the parallel implementation. And what techniques and functions I used.

First of all I followed the guidelines described in the assignment. So the process with rank 0 do only the I/O tasks and after reading data it distributes the values between the processes, which includes the number of all elements what is needed to calculate the number of elements in one process, which, due to the number of all elements is dividable by the number of processes, is the same for all process. This communication is done by sending an the integer value from process 0 to all other processes with `MPI_Bcast`, which is created for broadcasting a value from one thread to all other one, so this is the good choice here. I also used `MPI_Barrier` to sync the processes here. This is not necessary, but seemed fine to do it here.

Now all process knows the value of length. So I created a list of this length to store the body of the data processed by each thread. Now that a sufficient size array is given I can distribute the data from process 0 to all thread using the `MPI_Scatter` function, which is created to do exactly this kind of task. So split a list equally between all process. Note that I can do this as it was given in the description that the data is equally dividable between the processes. Otherwise I could use the `MPI_Scatterv` initiative to distribute all the elements unequally between the threads.

So at this point the data is distributed between the processes, and we can start the first iteration of the stencil calculation. Note that as each process is need information from it's left and right neighbours at the beginning of each iteration we need to add a communication section which from each process will send the the necessary information (`EXTENT` number of elements from the boundary) for it's left and right neighbours. The variable with the correct length are created to store these data and then I use `MPI_Isend` to send the data for the left and right neighbours with `MPI_Wait` before making change in the data for the send requests. And use `MPI_Irecv` with `MPI_Wait` before the usage of the given variable to get the messages from the neighbours and store the data in the created arrays for left and right side before trying to use them. I choose the Isend and Irecv with Wait, because as a non-blocking communication it is avoiding the deadlock problem and with the use of the

Wait I guarantee that the data is received before usage and send before there is change in the data. This way I can reach a slight speed up with calculating the main part of the calculations in the threads without the need of wait for the data sending process.

Note that do to the implementation of the communication this process if working even if I only use one processes as at that case it is itself right and left neighbours so it still exchange communication just in this case with itself to get the correct values stored in the arrays for the neighbouring values.

In the stencil process I should modify each loop. First of all as the end index is changed. In the middle loop that's the only modification needed, but since the receive function wait for the data to arrive I moved this for loop above the other two and I wait for the receive functions just right before the loop where they are needed. In the other two loop after computing the index (without shift) I have to decide if it is inside the interval of the process stored number and use the original update of the sum that case or if it is outside the processes range then use the corresponding value from the arrays storing the neighbouring values.

Note that as giving an if statement/ branch to a loop it is insufficient for large loops, but as in this case the `EXTENT` value is 2, so small compared to the whole array, so this shouldn't cause a problem in this case for the run time. Note that in cases where the `EXTENT` value is large compared to the length of the body, then most likely it is not necessary to do parallelization. But if we do, then a more efficient way would be to do that by storing the body left neighbour values the body values and the right neighbour values in one array. That way the use of the branches wouldn't be necessary, which can make the code faster in most case.

So the calculation of the stencil works this way, and after each calculation we still can use the change of input and output pointer just note that in this case the output length was modified to be still the same as the input of one process. And now an other iteration can start of the stencil calculation again with the synchronizations with the neighbours and then the local calculations.

After the given number of iterations we want to send all end values and run

time values to process 0. The gathering of the data is done by the `MPI_Gather` function. And the collection and maximum calculation of the run times is done with the `MPI_Reduce` function as these are the most sufficient functions for the given tasks. After sending all necessary data, the final process of outputs are done by process 0 as asked in the task including the creation of output file and the printing out of the longest measured run time in the processes.

So this is how my parallel implementation works. I always tried to choose the function which was created for the given task as most likely they are optimized for this tasks. Note that most of them could be self implemented by using the different send and receive functions. So the choice for `MPI_Bcast`, `MPI_Scatter` and `MPI_Gather` was mostly straight forward. The only real choice was how to implement the communication with neighbours. I wanted to use a simple communicator índ wanted to avoid the possibility of deadlock. So the most simple non-blocking communication choice was `MPI_Isend`, which was seemingly good for my goal. I combined it with `MPI_Irecv` and `MPI_Wait` this way I can control by which time the data should be arrive and be sent so for the calculations I have the necessary data and I don't change the data before sending it to the neighbours. So overall I tried to use a specific function, where I could for more complicated communications.

This concludes the brief description of the parallelization, the used communication and the reasoning for it.

Performance experiment

Strong scaling

First I build an experiment, which shows the strong scaling for a given number of iteration for the stencil application. For reference I also measured a base time with the original serial code.
For the experiment I wrote a bash script. It uses an input argument for the file name, which I want to do the experiment. For the measurements below I used the *input8000000.txt* file. Then the script iterates through some pre defined number and use them as number of stencil application. For measurements I used the 2 powers from 1 to 256 as iteration numbers. Then I iterate in a predefined set of thread numbers. For measurements I used the following set $\{1, 2, 4, 8, 12, 16, 20\}$. Note that the measurement were run in the university arrhenius server (Intel(R) Xeon(R) CPU E5520 2.27GHz processor with 8 core), which has a limited number of resources, so as we can see we only achieve stable speed up until 8 process.
And for each combination of iteration and process number I run 5 times the program and I plot the minimum of thee 5 run time. This way the slight changes in performance between difference runs didn't matter that much.
I also did this measurements for the original serial code as a base line.
All the measurements can be see at Table 7. [4]

#iter	base	$-n$ 1	$-n$ 2	$-n$ 4	$-n$ 8	$-n$ 12	$-n$ 16	$-n$ 20
1	0.0512	0.0511	0.0263	0.0140	0.0080	0.0099	0.0078	0.0196
2	0.0728	0.0728	0.0405	0.0210	0.0138	0.0160	0.0153	0.0206
4	0.1156	0.1247	0.0687	0.0353	0.0255	0.0280	0.0270	0.0302
8	0.2043	0.2394	0.1392	0.0639	0.0490	0.0537	0.0986	0.1022
16	0.3775	0.4123	0.2526	0.1298	0.0985	0.1346	0.1945	0.2058
32	0.7246	0.7653	0.4766	0.2443	0.1917	0.2979	0.3374	0.4154
64	1.4181	1.4485	0.9250	0.4739	0.3798	0.5935	0.7228	0.8202
128	2.8064	2.8982	1.8218	0.9334	0.7812	1.2068	1.4826	1.6043
256	5.7725	5.7091	3.7051	1.9292	1.5694	2.4300	2.9232	3.2165

Table 1: Time measurement for *input8000000.txt* file

[4] I use time measurements with only precision up to 4 digits as the other digits are very noisy and unstable

As in the table we can see the original serial code is only a slightly faster than my parallel implementation with one process. So the parallel implementation don't made the serial performance much worse. And it can be expected that a parallel implementation using one thread is slower then a good serial code. From the table we can also see that after 8 core the speed up stops, this is caused by the limitation of the resources as the university server has only 8 core. So for scaling measurement I will only use the data up to 8 core. Now in Figure 1 we can see the plot of the speed ups with the parallel code using 256, 128 and 64 iteration for 1,2,4 and 8 core. With black the ideal speed up is plotted using the run time of the original serial code for 256 iteration.

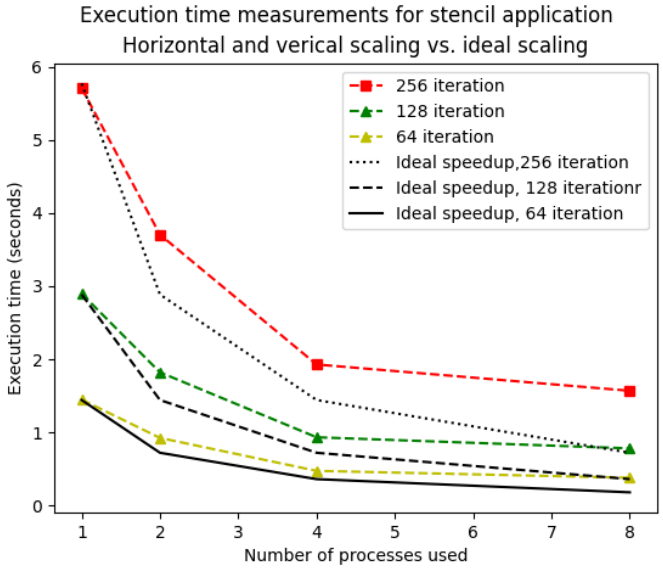


Figure 1: Strong scaling for large number of iterations

Now the strong scaling can be seen next to the x axes. As we can see until 8 core we get significant speed ups with the usage of more resources. It is a bit behind the ideal speed up. But it is normal as we couldn't achieve perfect speed up due to the need of communications between the threads after each iteration. Considering this we still get a nice speed up. As we can see from the figure we get the best speed up relatively to the ideal speed up with 4 process. In my opinion the reason is the following: for 8 process we can't efficiently use all thread as the operation system should use some resources, so as we reach the maximum of our resources we can't use all of it so ef-

ficiently. And I think for 2 thread the relative loss for the synchronization (processes should wait to each other) is larger compared to larger number of processes as for more process the average waiting will be smaller. Note that for 1 process it takes no time as the one process always calls send before the corresponding receiver.

Now looking at the figure and the table we can see the following for "vertical" scaling, which I used here for referring to the change in iteration. As we can see it for large iteration we almost get an ideal speed up if we reduce the number of iterations for any number of processes. But as we reach small number of iterations the speed up becomes much worse. Most likely due to the overhead which is includes the parts which won't scale with the change in iteration (e.g: there are allocation and deallocation in the measurements). So I would conclude that the speed up with the change in iteration is almost perfect for large number of iteration and still good for small number of iteration. But as it was said for lots of application we need a large number of stencil application when doing numerical modelling, so in those cases the fact that for large number of iteration the speed up is almost ideal is useful to know.

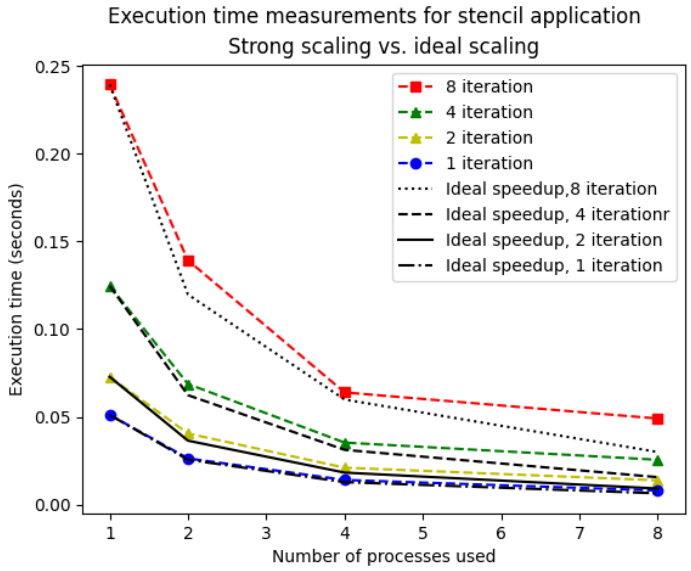


Figure 2: Strong scaling for small number of iterations

In Figure 2 we can see the strong scaling for small number of iterations with the use of the *input8000000.txt* input file. Here the run time for 1,2,4 and 8 iteration is plotted with 1,2,4 and 8 processes. Here the plotted ideal speed up is always starts from the run time on one core with the given number of iteration and the scaling between iteration is not considered here. This way we can see that the horizontal scaling, so the scaling as we use more core is almost perfect with small number of iterations. But note that here the scaling between the iterations is far from perfect.

Weak scaling

For weak scaling we want to compare the run time with same amount of task by each thread as we increase the processes. One way would be to change the iteration number as the scale of the work. This data can be read from the above table and also can be seen in the above figures if we compare run times for different number of processes as the iterations change the same way. The scaling which can be seen in the above figures are the following for 1,2,4 and 8 core on 1,2,4 and 8 iteration the run times are 0.0511, 0.0405, 0.0353, 0.0490 respectively. Which indicates a very good weak scaling. And for 64, 128 and 256 iteration with 1,2 and 4 core the run times are 1.4485, 1.8218, 1.9292 and for 2,4 and 8 core the run times are 0.9250, 0.9334 , 1.5694 respectively. So the weak scaling seems to be very good everywhere for 2 and 4 core. Good for 1 to 2 core and not so good or bad for 8 core.

Another option for weak scaling is to investigate run times as the work which is changing is the size of the input.
For this weak scaling measurements I used the large input files with 1,2,4 and 8 millions elements. And for each time when I doubled the size of input data I also doubled the used number of cores. For measurements I used 1, 2, 10, 100, 200, 300, 400 and 500 iteration.

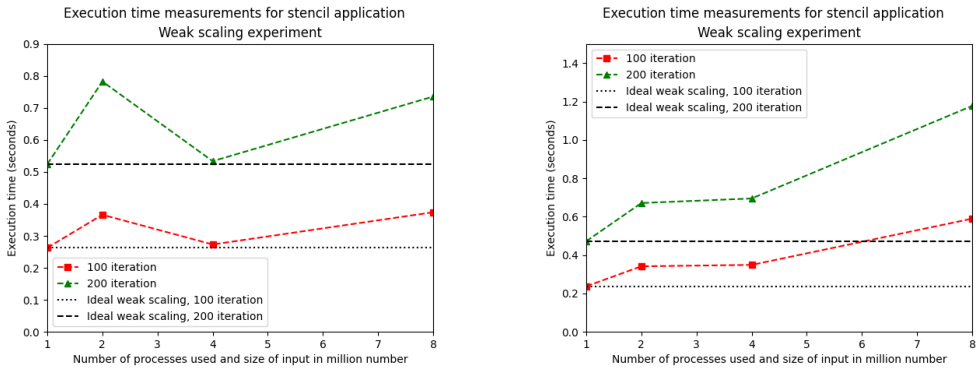
I did the time measurements in two university Linux server *vitsippa* and *fredholm* and I got significantly different results. Note that the fredholm server has the same Intel(R) Xeon(R) CPU as the arhenius server where most of my previous time measurements were made and the vitsippa server has a AMD Opteron(tm) Processor 6282 SE also with 8 core.
So the measurements and the tendency in the weak scaling also highly dependent on the system which we are using as in Table 2 and Table 3 we can

input size $\cdot 10^6$ / # process	1 / 1	2 / 2	4 / 4	8 / 8
Run time: 1 iteration	0.0084	0.0088	0.0085	0.0090
Run time: 2 iteration	0.0109	0.0126	0.0113	0.0126
Run time: 10 iteration	0.0317	0.0421	0.0328	0.0421
Run time: 100 iteration	0.2626	0.3660	0.2731	0.3738
Run time: 200 iteration	0.5240	0.7824	0.5335	0.7355
Run time: 300 iteration	0.7887	1.0992	0.7997	1.1026
Run time: 400 iteration	1.0242	1.4574	1.0718	1.4664
Run time: 500 iteration	1.2875	1.9022	1.3293	1.8308

Table 2: Weak scaling with different input files (vitsippa server)

input size $\cdot 10^6$ / # process	1 / 1	2 / 2	4 / 4	8 / 8
Run time: 1 iteration	0.0071	0.0070	0.0072	0.0082
Run time: 2 iteration	0.0095	0.0104	0.0107	0.0141
Run time: 10 iteration	0.0280	0.0370	0.0383	0.0610
Run time: 100 iteration	0.2373	0.3415	0.3488	0.5896
Run time: 200 iteration	0.4721	0.6711	0.6947	1.1772
Run time: 300 iteration	0.7089	1.0008	1.0341	1.7646
Run time: 400 iteration	0.9406	1.3411	1.3871	2.3486
Run time: 500 iteration	1.1734	1.6748	1.7372	2.9399

Table 3: Weak scaling with different input files (fredholm server)



(a) vitsippa server

(b) fredholm server

Figure 3: Weak scaling

see. Also to se the different tendencies we can look at Figure 3a and Figure 3b.

In general for small number of processes the weak scalability seems almost perfect. This can be caused by the fact that the runtimes are very small, so the differences can't be measured well here. Also in this cases the synchronizations should work relatively fast, as larger waiting can only become relevant in large number of iterations.
If we look at Figure 3a and Figure 3b, then we can see that depending on the CPU architecture the weak scaling seems almost perfect either between 1 and 4 core or between 2 and 4 core. Unsurprisingly the 1 threaded measurement is the fastest as the work by each core is the same, but with more core there is an overhead for synchronization what can cause time loss. Alos as more core is used there is possibility that one core becomes slower for some reason (e.g: for 8 core we use the whole capacity of the CPU so there should be a thread which is working on the background processes, so can't be used fully) and we are interested in the longest threads run time.
I can't explain sufficiently why the difference between the two arthitechture occurse, but it seemed like an interesting thing to note. As I mentioned for 8 core I can reason why it becomes significantly slower, but in the case of the *vitsippa* server I don't know why the run time for 2 process becomes significantly slower as with 1 and 4 processes, while in the *fredholm* server the run time for 2 and 4 core is basically the same, but there the 8 core is the one I can't explain why is it so slow in that system.
A reason could be for slow run time the input file, but that would cause slow runtime in both server, so that can't be the reason here.

And what I said before so that the scaling regarding the number of iteration is becomes chaotic with small number of iterations we can see that here as well. Also the scaling between large number of iterations with some hundred iterations seems to be well scaled respect to the number of iterations, but the nice linear scaling get worse as getting closer to 1 iteration. E.g: between 1 and 2 iteration the run times doesn't becomes twice as much, but almost stays the same.

Additional time measurements

I also run my script for some other input file which results can be found in the tables below. I didn't present these time measurements for *input96.txt* and *input120.txt* as those run times are to small to measure (mostly $< 10^{-4}$s). From these data the conclusions would be very similar as it was from the data presented in the above sections.

#iter	base	$-n$ 1	$-n$ 2	$-n$ 4	$-n$ 8	$-n$ 12	$-n$ 16	$-n$ 20
1	0.0064	0.0063	0.0033	0.0016	0.0009	0.0011	0.0009	0.0019
2	0.0093	0.0092	0.0051	0.0023	0.0014	0.0015	0.0017	0.0033
4	0.0147	0.0146	0.0086	0.0034	0.0023	0.0026	0.0025	0.0032
8	0.0253	0.0254	0.0155	0.0056	0.0039	0.0042	0.0042	0.0052
16	0.0466	0.0469	0.0293	0.0102	0.0074	0.0079	0.0077	0.0099
32	0.0892	0.0895	0.0572	0.0193	0.0142	0.0158	0.0142	0.0210
64	0.1739	0.1754	0.1125	0.0372	0.0271	0.0334	0.0288	0.0374
128	0.3434	0.3441	0.2236	0.0728	0.0555	0.0954	0.1063	0.1145
256	0.6816	0.6930	0.4450	0.1431	0.1098	0.1909	0.2149	0.2383

Table 4: Time measurement for *input1000000.txt* file

#iter	base	$-n$ 1	$-n$ 2	$-n$ 4	$-n$ 8	$-n$ 12	$-n$ 16	$-n$ 20
1	0.0130	0.0126	0.0065	0.0034	0.0020	0.0025	0.0023	0.0033
2	0.0185	0.0183	0.0102	0.0053	0.0034	0.0039	0.0038	0.0048
4	0.0294	0.0291	0.0173	0.0088	0.0063	0.0066	0.0067	0.0082
8	0.0510	0.0506	0.0313	0.0159	0.0121	0.0131	0.0126	0.0136
16	0.0944	0.0938	0.0594	0.0302	0.0236	0.0255	0.0242	0.0227
32	0.1814	0.1799	0.1157	0.0585	0.0467	0.0499	0.0963	0.0451
64	0.3554	0.3618	0.2280	0.1150	0.0926	0.1295	0.1452	0.1900
128	0.7000	0.7066	0.4565	0.2287	0.1854	0.2925	0.3346	0.3386
256	1.3945	1.3960	0.9065	0.4574	0.3699	0.5840	0.7227	0.7065

Table 5: Time measurement for *input2000000.txt* file

#iter	base	$-n$ 1	$-n$ 2	$-n$ 4	$-n$ 8	$-n$ 12	$-n$ 16	$-n$ 20
1	0.0257	0.0253	0.0132	0.0069	0.0040	0.0046	0.0044	0.0086
2	0.0367	0.0363	0.0203	0.0105	0.0069	0.0079	0.0069	0.0122
4	0.0586	0.0577	0.0343	0.0176	0.0128	0.0142	0.0231	0.0180
8	0.1032	0.1032	0.0646	0.0328	0.0256	0.0272	0.0272	0.0284
16	0.1890	0.1920	0.1206	0.0630	0.0503	0.0819	0.0980	0.1028
32	0.3626	0.3590	0.2301	0.1181	0.0951	0.1309	0.1953	0.2050
64	0.7099	0.7039	0.4554	0.2333	0.1888	0.2966	0.3382	0.4088
128	1.4047	1.3920	0.9032	0.4632	0.3762	0.5929	0.7257	0.8307
256	2.7926	2.7893	1.8056	0.9225	0.7531	1.1895	1.5037	1.6264

Table 6: Time measurement for *input4000000.txt* file

#iter	base	$-n$ 1	$-n$ 2	$-n$ 4	$-n$ 8	$-n$ 12	$-n$ 16	$-n$ 20
1	0.8448	0.6321	0.3271	0.1738	0.1067	0.1751	0.1993	0.2441
2	0.9054	0.9019	0.5040	0.2575	0.1940	0.2925	0.3308	0.4368
4	1.4497	1.4468	0.8560	0.4800	0.3418	0.5221	0.6185	0.7074
8	2.5519	2.5270	1.6086	0.8412	0.6147	0.9505	1.2212	1.3150
16	4.7177	4.6920	2.9957	1.5559	1.2063	1.8814	2.3328	2.5314
32	11.8177	9.0055	5.8012	2.9797	2.3918	3.7300	4.7253	4.9766
64	17.7376	17.7308	11.4278	5.8443	4.7442	7.3372	11.3793	10.0110
128	35.2268	35.1087	22.6978	11.5701	9.4316	14.5679	18.8943	19.3220
256	70.1820	69.8585	45.2338	23.0405	18.8622	28.8257	37.6152	39.1180

Table 7: Time measurement for *input100000000.txt* file

Summary

So in summary the parallel implementation with MPI seemingly scales well for multiple number of processes until there is empty resource in the system. For 1 core the parallel implementation is faster almost as fast as the serial, and it is relatively faster compared to multiple processes as in that cases the processes may wait for each other at the communication steps. Also in experience as I reached the limits of the system resources the scaling becomes significantly worse then the ideal speed up.

Overall I would conclude that for real stencil application with large input files and large number of iteration, both of which appears regularly in numerical modelling tasks I would recommend using scaling up to the available resources. After the max number of executor is reached further improvement can't be reached with larger number of executor using `-oversubscribe`.

I also note that in most case the overhead of the program so the input output processes are more time consuming then the stencil application itself. At least this is the case for the large files. E.g: with the largest input using 100 iteration the measured time is 25s, while using `time` the real time is around 72s. For really large number of iteration this ain't the case. E.g: with the *input96.txt* input file for 10^8 iteration the overhead is smaller then the measured run time 10%. And I guess that would be the case for large number of iteration with large files. E.g: for 1000 iteration with the *input1000000.txt* input the measured time is 1.48s and the real time is 2.1s, which already a much better ratio then it was with 100 iteration.
So what I am trying to say is that for real application with large data sets the read time is also significant, so it would be possible to consider distribute the starting data (e.g: every thread reads its input data). But this is outside the aim of the current project.

So this implementation can be useful for large scale stencil application, but for depending on the application, there may be changes which could make the performance of the stencil application process faster.

Appendix

Original serial code

```
#include "stencil.h"


int main(int argc, char **argv) {
        if (4 != argc) {
                printf("Usage: stencil input_file output_file number_of_applications\n")
                    ;
                return 1;
        }
        char *input_name = argv[1];
        char *output_name = argv[2];
        int num_steps = atoi(argv[3]);

        // Read input file
        double *input;
        int num_values;
        if (0 > (num_values = read_input(input_name, &input))) {
                return 2;
        }

        // Stencil values
        double h = 2.0*PI/num_values;
        const int STENCIL_WIDTH = 5;
        const int EXTENT = STENCIL_WIDTH/2;
        const double STENCIL[] = {1.0/(12*h), -8.0/(12*h), 0.0, 8.0/(12*h), -1.0/(12*h)
             };

        // Start timer
        double start = MPI_Wtime();

        // Allocate data for result
        double *output;
        if (NULL == (output = malloc(num_values * sizeof(double)))) {
                perror("Couldn't allocate memory for output");
                return 2;
        }
        // Repeatedly apply stencil
        for (int s=0; s<num_steps; s++) {
                // Apply stencil
                for (int i=0; i<EXTENT; i++) {
                        double result = 0;
                        for (int j=0; j<STENCIL_WIDTH; j++) {
                                int index = (i - EXTENT + j + num_values) % num_values;
                                result += STENCIL[j] * input[index];
                        }
                        output[i] = result;
                }
                for (int i=EXTENT; i<num_values-EXTENT; i++) {
                        double result = 0;
                        for (int j=0; j<STENCIL_WIDTH; j++) {
                                int index = i - EXTENT + j;
                                result += STENCIL[j] * input[index];
                        }
                        output[i] = result;
                }
                for (int i=num_values-EXTENT; i<num_values; i++) {
                        double result = 0;
                        for (int j=0; j<STENCIL_WIDTH; j++) {
                                int index = (i - EXTENT + j) % num_values;
                                result += STENCIL[j] * input[index];
                        }
                        output[i] = result;
                }
                // Swap input and output
                if (s < num_steps-1) {
                        double *tmp = input;
                        input = output;
                        output = tmp;
                }
        }
```

```
        free(input);
        // Stop timer
        double my_execution_time = MPI_Wtime() - start;

        // Write result
        printf("%f\n", my_execution_time);
#ifdef PRODUCE_OUTPUT_FILE
        if (0 != write_output(output_name, output, num_values)) {
                return 2;
        }
#endif

        // Clean up
        free(output);

        return 0;
}


int read_input(const char *file_name, double **values) {
        FILE *file;
        if (NULL == (file = fopen(file_name, "r"))) {
                perror("Couldn't open input file");
                return -1;
        }
        int num_values;
        if (EOF == fscanf(file, "%d", &num_values)) {
                perror("Couldn't read element count from input file");
                return -1;
        }
        if (NULL == (*values = malloc(num_values * sizeof(double)))) {
                perror("Couldn't allocate memory for input");
                return -1;
        }
        for (int i=0; i<num_values; i++) {
                if (EOF == fscanf(file, "%lf", &((*values)[i]))) {
                        perror("Couldn't read elements from input file");
                        return -1;
                }
        }
        if (0 != fclose(file)){
                perror("Warning: couldn't close input file");
        }
        return num_values;
}


int write_output(char *file_name, const double *output, int num_values) {
        FILE *file;
        if (NULL == (file = fopen(file_name, "w"))) {
                perror("Couldn't open output file");
                return -1;
        }
        for (int i = 0; i < num_values; i++) {
                if (0 > fprintf(file, "%.4f ", output[i])) {
                        perror("Couldn't write to output file");
                }
        }
        if (0 > fprintf(file, "\n")) {
                perror("Couldn't write to output file");
        }
        if (0 != fclose(file)) {
                perror("Warning: couldn't close output file");
        }
        return 0;
}
```

Given header file

```c
/**
 * This program applies a five point stencil on a vector of function values
 * f(x) for 0<=x<2*PI in order to approximate the derivative of the function.
 * The program takes 3 arguments:
 * - the path to the input file.
 * - the path to the output file. Note that this file will be overwritten!
 * - an integer specifying how many times the stencil will be applied
 * The formats of the files are described in the documentation of the functions
 * read_input and read_output. The program also prints the number of seconds
 * needed for the stencil application to standard out.
 */

#ifndef _ASSIGNMENT1_STENCIL_H_
#define _ASSIGNMENT1_STENCIL_H_

#define PI 3.14159265358979323846
#define PRODUCE_OUTPUT_FILE

#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

/**
 * Read function data from an input file and store in an array. The input file
 * is supposed to contain an integer representing the number of function values,
 * followed by the function values. All values should be separated by white
 * spaces.
 * @param file_name Name of input file
 * @param values Pointer to array where the values are to be stored
 * @return 0 on success, -1 on error
 */
int read_input(const char *file_name, double **values);

/**
 * Write function data to a file, with 4 decimal places. The values are
 * separated by spaces.
 * @param file_name Name of output file
 * @param output Function values to write
 * @param num_values Number of values to print
 * @return 0 on success, -1 on error
 */
int write_output(char *file_name, const double *output, int num_values);

#endif /* _ASSIGNMENT1_STENCIL_H_ */
```

Given Make file

```
################################################################################
# Makefile for assignment 1, Parallel and Distributed Computing 2022.
################################################################################

CC = mpicc
CFLAGS = -std=c99 -g -O3
LIBS = -lm

BIN = stencil

all: $(BIN)

stencil: stencil.c stencil.h
	$(CC) $(CFLAGS) -o $@ $< $(LIBS)

clean:
	$(RM) $(BIN)
```

My parallelized code with OpenMPI

```
#include "stencil.h"
#define root 0
#define to_right 201
#define to_left 102

int main(int argc, char **argv) {
        if (4 != argc) {
                printf("Usage: stencil input_file output_file number_of_applications\n")
                    ;
                return 1;
        }
        char *input_name = argv[1];
        char *output_name = argv[2];
        int num_steps = atoi(argv[3]);


        double *input0=NULL;
    double *input=NULL;
    double *left_stecil=NULL;
    double *right_stecil=NULL;
    double *run_times=NULL;
    int num_values, length;
    int left, right;
    int size;
    int rank;
        double max_runtime;

        MPI_Request recv_status_right;
        MPI_Request request_right;
        MPI_Request recv_status_left;
        MPI_Request request_left;

    //MPI init
    MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    right = rank + 1;
        if (right == size) right = 0;

        left = rank - 1;
        if (left == -1) left = size-1;


    if(rank==root){
    // Read input file

        if (0 > (num_values = read_input(input_name, &input0))) {
                return 2;
        }
    }
        //Broadcastin the length of the input
    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Bcast(&num_values, 1, MPI_INT, root, MPI_COMM_WORLD);
    MPI_Barrier(MPI_COMM_WORLD);
    length=num_values/size;
    input=(double*)malloc(length*sizeof(double));

        // Distributing the input data
    MPI_Scatter(input0, length, MPI_DOUBLE, input, length, MPI_DOUBLE, root,
        MPI_COMM_WORLD);


        // Stencil values
        double h = 2.0*PI/num_values;
        const int STENCIL_WIDTH = 5;
        const int EXTENT = STENCIL_WIDTH/2;
        const double STENCIL[] = {1.0/(12*h), -8.0/(12*h), 0.0, 8.0/(12*h), -1.0/(12*h)
             };


    left_stecil=(double*)malloc(EXTENT*sizeof(double));
    right_stecil=(double*)malloc(EXTENT *sizeof(double));
```

```
// Start timer
MPI_Barrier(MPI_COMM_WORLD);
double start = MPI_Wtime();

// Allocate data for result
double *output;
if (NULL == (output = malloc(length * sizeof(double)))) {
        perror("Couldn't allocate memory for output");
        return 2;
}

// Repeatedly apply stencil
for (int s=0; s<num_steps; s++) {

        // Send processes for sync with neighbours
        MPI_Isend(input, EXTENT, MPI_DOUBLE, left, to_left, MPI_COMM_WORLD, &
            request_left);
        MPI_Isend(&input[length-EXTENT], EXTENT, MPI_DOUBLE, right, to_right,
            MPI_COMM_WORLD, &request_right);
        MPI_Irecv(&left_stecil[0], EXTENT, MPI_DOUBLE, left, to_right,
            MPI_COMM_WORLD, &recv_status_left);
        MPI_Irecv(&right_stecil[0], EXTENT, MPI_DOUBLE, right, to_left,
            MPI_COMM_WORLD, &recv_status_right);


        for (int i=EXTENT; i<length-EXTENT; i++) {
                double result = 0;
                for (int j=0; j<STENCIL_WIDTH; j++) {
                        int index = i - EXTENT + j;
                        result += STENCIL[j] * input[index];
                }
                output[i] = result;
        }

        // Receive process for sync with neighbours - just before it's
            application
        MPI_Wait(&recv_status_left, MPI_STATUS_IGNORE);


        for (int i=0; i<EXTENT; i++) {
                double result = 0;
                for (int j=0; j<STENCIL_WIDTH; j++) {
                        int index = (i - EXTENT + j);
                        if (index>=0)
                                result += STENCIL[j] * input[index];
                        else
                                result += STENCIL[j] * left_stecil[EXTENT+index
                                    ];
                }
                output[i] = result;
        }


        // Receive process for sync with neighbours - just before it's
            application
        MPI_Wait(&recv_status_right, MPI_STATUS_IGNORE);

        for (int i=length-EXTENT; i<length; i++) {
                double result = 0;
                for (int j=0; j<STENCIL_WIDTH; j++) {
                        int index = (i - EXTENT + j);
                        if (index<length)
                                result += STENCIL[j] * input[index];
                        else
                                result += STENCIL[j] * right_stecil[index-length
                                    ];
                }
                output[i] = result;
        }
        // Swap input and output
        MPI_Wait(&request_left, MPI_STATUS_IGNORE);
        MPI_Wait(&request_right, MPI_STATUS_IGNORE);

        if (s < num_steps-1) {
                double *tmp = input;
                input = output;
```

```
                        output = tmp;
                }
        }
        free(input);
        // Stop timer
        double my_execution_time = MPI_Wtime() - start;


    //Gather data and runtime on root
        MPI_Gather(output, length, MPI_DOUBLE, input0, length, MPI_DOUBLE, root ,
            MPI_COMM_WORLD);

    if(rank==root)
        run_times=(double *)calloc(size,sizeof(double));

        MPI_Reduce(&my_execution_time, &max_runtime, 1, MPI_DOUBLE, MPI_MAX, root,
            MPI_COMM_WORLD);
    // Write results from root
    if(rank==root){

            printf("%f\n", max_runtime);
        #ifdef PRODUCE_OUTPUT_FILE
            if (0 != write_output(output_name, input0, num_values)) {
                return 2;
            }
        #endif
    }

        // Clean up
        free(output);
    free(input0);
    MPI_Finalize();
        return 0;
}


int read_input(const char *file_name, double **values) {
        FILE *file;
        if (NULL == (file = fopen(file_name, "r"))) {
                perror("Couldn't open input file");
                return -1;
        }
        int num_values;
        if (EOF == fscanf(file, "%d", &num_values)) {
                perror("Couldn't read element count from input file");
                return -1;
        }
        if (NULL == (*values = malloc(num_values * sizeof(double)))) {
                perror("Couldn't allocate memory for input");
                return -1;
        }
        for (int i=0; i<num_values; i++) {
                if (EOF == fscanf(file, "%lf", &((*values)[i]))) {
                        perror("Couldn't read elements from input file");
                        return -1;
                }
        }
        if (0 != fclose(file)){
                perror("Warning: couldn't close input file");
        }
        return num_values;
}


int write_output(char *file_name, const double *output, int num_values) {
        FILE *file;
        if (NULL == (file = fopen(file_name, "w"))) {
                perror("Couldn't open output file");
                return -1;
        }
        for (int i = 0; i < num_values; i++) {
                if (0 > fprintf(file, "%.4f ", output[i])) {
                        perror("Couldn't write to output file");
                }
        }
        if (0 > fprintf(file, "\n")) {
                perror("Couldn't write to output file");
```

```
        }
        if (0 != fclose(file)) {
                perror("Warning: couldn't close output file");
        }
        return 0;
}
```

Time measurement script

```
#!/bin/bash

make
mpicc -std=c99 -g -O3 -o original stencil_old.c -lm

# vertical scaling
for i in 1 2 4 8 16 32 64 128 256
do
echo -n "$i & "

    res=$(./original $1 out.txt $i)
    for p in {1..4}
    do
        res2=$(./original $1 out.txt $i)
        if [[ $res > $res2 ]];
        then
            res=$res2
        fi
    done
    echo -n "$res & "


    for n in 1 2 4 8 12 16 20
    do
        #echo "mpirun -n $n -oversubscribe --bind-to none ./stencil $1 out.txt $i"
        res=$(mpirun -n $n -oversubscribe ./stencil $1 out.txt $i)
        for p in {1..4}
        do
            res2=$(mpirun -n $n -oversubscribe ./stencil $1 out.txt $i)
            if [[ $res > $res2 ]];
            then
                res=$res2
            fi
        done
        #echo "File: $1"
        #echo "$res s for $i repetation with $n thread"
        #echo ""
        echo -n "$res & "
    done
    echo ""
done
```