One-dimensional stencil application

David Hovstadius
Oskar Lernholt
Gustav Malmsten

UPPSALA
UNIVERSITET

Parallel and distributed programming (1TD070)
Teknisk fysik, Uppsala Universitet
April 9, 2023

1 Introduction

A stencil application is an operator that can be applied to a vector, matrix or a tensor. When applying a stencil to an element a value is computed using the elements current value and the values of its neighbours. In this report a one dimensional stencil will be applied on an array of elements. These elements represent a function value $f(x)$. On the interval $0 \leq x < 2\pi$.

Applying the stencil on an element x_0. The following sum is being computed:

$$\frac{1}{12h}f(x_{-2}) - \frac{8}{12h}f(x_{-1}) + 0 \cdot f(x_0) + \frac{1}{12h}f(x_1) - \frac{1}{12h}f(x_2)$$

Where x_i is the element i steps to the right of element x_0 if i is positive and to the otherwise. $h = \frac{2\pi}{N}$ where N is the total number of elements in the array. This sum is an approximation of the first derivative $f(x_i)$.

2 Description of your parallelization

The parallelization was achieved using MPI. Circular communication was implemented using a virtual topology created using MPI's `MPI_Cart_create` function to facilitate the communication between the processing elements to the left and right of each processing element. This communication method was selected because of the periodic nature of the problem to be solved.

To handle the edge cases, the sub arrays were extended by the extent, 2 in this case, of the stencil on both sides of the array.

At the beginning of each stencil application, all processes exchange data, as illustrated in Figure 1. The data is sent using non-blocking data transfer and received with blocking data transfer. After that the stencil is used on each process array. If more than one application of the stencil is to be done, the process is repeated on the previous result of the stencil computations. If this is the case, the program needs to check that the non blocking send is completed, to avoid modifying the data scheduled that is to be sent. This is done using the MPI_Waitall call.

After completing all computations, the master process gathers all the outputs from the other processes into a single list, and fetches the execution times from all other processes and reduces these into one global execution time, the largest one.

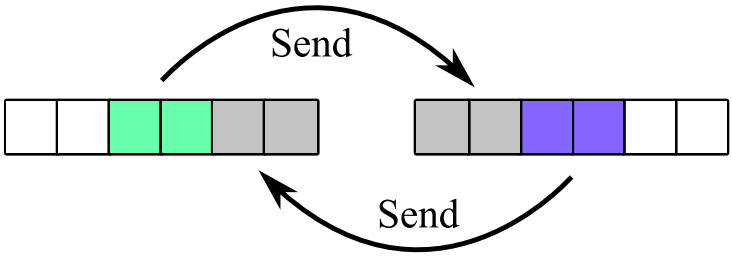

Figure 1: Diagram showing communication between two processes

3 Experiments

The performance experiments were run on the Uppsala University linux host, Vitsippa. Vitsippa has 16 cores and thus 32 hardware threads. The performance experiments will therefore be conducted using powers of two processing elements, up to 32 processing elements. Both the weak and strong scalability will be evaluated by running the stencil on 1e8 elements. For the strong scalability, the stencil will be applied once, while for the weak scalability, the stencil will be applied np times, where np is the number of processing elements. The strong scaling will be evaluated regarding relative speedup.

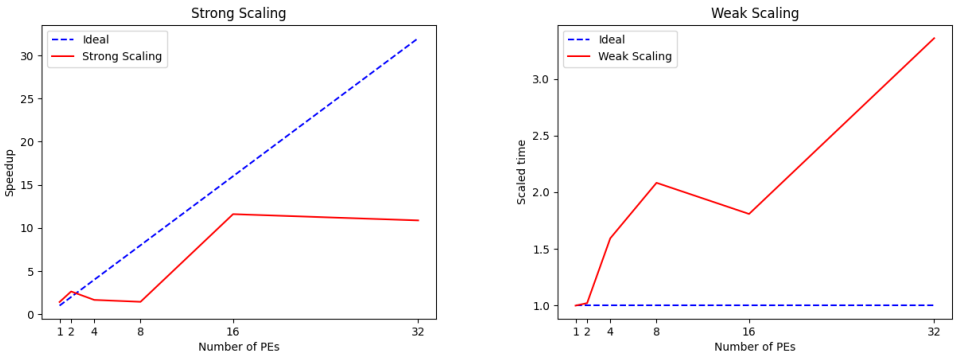

Figure 2: Strong and weak scaling

Table 1: Execution times for stencil application on variable problem size $N \cdot (nApplications) = 10^8 \cdot (nApplications)$ for varying number of processing elements

n PEs	$nApplications$	Execution time [s]
1	1	0.708979
2	2	0.724634
4	4	1.129856
8	8	1.477219
16	16	1.282406
32	32	2.381665

Table 2: Execution times for stencil application on fixed problem size $N = 10^8$ for varying number of processing elements

n PEs	Execution time [s]
1	0.706233
2	0.379075
4	0.602186
8	0.694319
16	0.086181
32	0.091930

4 Discussion

Based on the experiments illustrated in Figure 2, it is clear that none of the experiments demonstrate ideal speedup. While the strong scaling approach shows some similarity to the ideal speedup, the weak scaling exhibits significant differences.

The reason for the suboptimal increase in speed can be attributed to the communication that takes place between different processes. This additional time taken for communication becomes more noticeable in weak scaling tests, where each iteration involves additional communication time.

In figure 2 it can be seen that no speedup is present for 2 to 8 processes. The reason for this is unknown. It could be an overhead reason where the overhead of creating threads and exchanging data is outweighing the benefits of parallelising the computations. This is consistent with the behaviour demonstrated in the weak scaling test where the scaled time increases significantly in the same range, to then

decrease until the limit of cores is surpassed. Apart from this range of processes, the speedup proves to be superlinear, until the limit of cores is passed.

Load balancing is not an issue in this assignment since the calculations are equally heavy in all processes. This makes it unnecessary to split the stencil application into three parts, one using only the data local to the thread, and two using the edge data to the left and right, respectively, of the process, since these computations should take the same time. If that would not have been the case, one would need to do the above described split and let each process first do the calculations that only depend on the local data, and then do the other calculations in the order it received the necessary data, using e.g. a non blocking receive in combination with e.g. two MPI_Wait calls.