UPPSALA
UNIVERSITET

Assignment 1
Parallel and Distributed Programming

Written by:
Jakob Gölén
Simon Persson

April 11, 2022

1 Problem Description

In this assignment, a 1D stencil application is investigated. A stencil application is an operator which changes elements in a matrix, or in the one dimensional case a vector, using the element in question and its neighbors. Here, the stencil application is described in equation 1. The interval is from 0 to 2π so $h = \frac{2\pi}{N}$ where N is the number of elements in the vector.

$$f(x_i) = \frac{1}{12h} * f(x_{i-2}) - \frac{8}{12h} * f(x_{i-1}) + 0 * f(x_i) + \frac{8}{12h} * f(x_{i+1}) - \frac{1}{12h} * f(x_{i+2}) \quad (1)$$

The stencil is applied to every element in the vector, and periodic boundary conditions are used so the first element has the last element as a neighbor and vice versa.

The main goal of the assignment was to parallelize a given serial code used to apply the stencil in equation 1 a number of times to every element in a vector. The given code takes three inputs: an input file containing the number of elements in the vector and their values, the name of an output file, and the number of times the stencil should be applied before writing to the output file. The program reads the input file, times how long it takes to apply the stencil the defined number of times, prints the time and then writes the result to the output file.

2 Parallelization

The code was parallelized using OpenMPI. The program would require each processing element to communicate with its neighbors, so a one dimensional cartesian topology was implemented by using MPI_Cart_Create and the neighboring processes was found by using MPI_Cart_Shift.

Next, the processing element which had rank 0 read the input file, copied the values into an array, and broadcasted the number of elements in the vector to every processing element by using MPI_Bcast. The number of elements was divided by the number of processing elements to find how many elements each processing element should take.

Each processing element allocated an array and the input values were distributed evenly from the processor with rank 0 by using MPI_Scatter. Each array had four more elements than the number of values distributed, this would be used to place two values from both neighbors. A contiguous datatype was defined with the purpose of sending two values with one send request and recieve two values with one recieve request.

With the values distributed into smaller arrays local for every processing element, a timer started and the main loop of the program began. First, the processing element sent the two first elements to the west neighbor and the two last elements to the east neighbor. Then the processing element recieved the first two elements from the east neighbor and placed them in the right padding, and recieved the last two elements from the west neighbor and placed them in the left padding. Non-blocking send and recieve requests, i.e. Isend and Irecv, was used so that the program didn't have to wait for the data to be recieved to continue executing code.

Next the stencil was applied. First, the stencil was applied to the elements which didn't need to use the padding and therefore didn't have to wait until the data from the neighbor had been placed in the padding. Next, MPI_Wait was used to ensure the data has been placed in the left padding, and then stencil was applied to the two values in the

array which used the two elements in the left padding. The same thing was then done to the two elements which used the two elements in the right padding, after waiting for the data to be placed in the right padding.

This process was then repeated for as many times as defined by the user. When the main loop was done, the timer stopped. The arrays in each process was combined into one array in processing element 0 by using gather, and all the timings from the processing elements were collected in the same way. Processing element 0 wrote the resulting array into the output file. The largest timing was found and printed in the console, again by the processing element with rank 0.

3 Performance Experiments

With the stencil program parallelized, the next step was to measure the parallel performance. All measurements were done on the Vitsippa Linux server at Uppsala University, using the timings printed by the program. Timing measurements were done on 1, 2, 4, 8 and 16 cores using input files with 1 million, 2 million, 4 million, 8 million and 100 million values. Speedup was calculated by dividing the time on the current number of cores by the time gotten on one core.

For the ideal speedup, both Amdahl's law described in equation 2 and Gustafson's law described in equation 3 was considered. p is the number of processing elements and f_p is the part of the code that is parallelized.

$$S \leq \frac{1}{(1 - f_p) + \frac{f_p}{p}} \tag{2}$$

$$S \leq p + (p - 1) * (1 - f_p) \tag{3}$$

Since only the parallel part of the program is timed, f_p is one. This means that both equation 2 and 3 can be simplified to $S \leq p$. The ideal speedup is therefore just the number of processing elements used.

4 Results

Table 1 and figures 1 and 2 shows the times noted when running the stencil program. The stencil was applied 10 times and the times were collected for five runs and then averaged. The values of the table were plotted in figure 1 and 2 since the values of applying the stencil on 100 million values took an an order of magnitude longer than the rest of the timings, as can be seen in table 1.

Table 1: Times for different amount of parallel cores and input sizes

cores	1	2	4	8	16
1M	54.2	23.27	14.53	10.71	11.29
2M	104.74	52.9	28.43	22.75	18.77
4M	206.66	104.22	62.14	51.43	53.42
8M	448.43	207.67	119.4	101.42	95.21
100M	5213.67	2603.33	1471.33	1176	1194.33

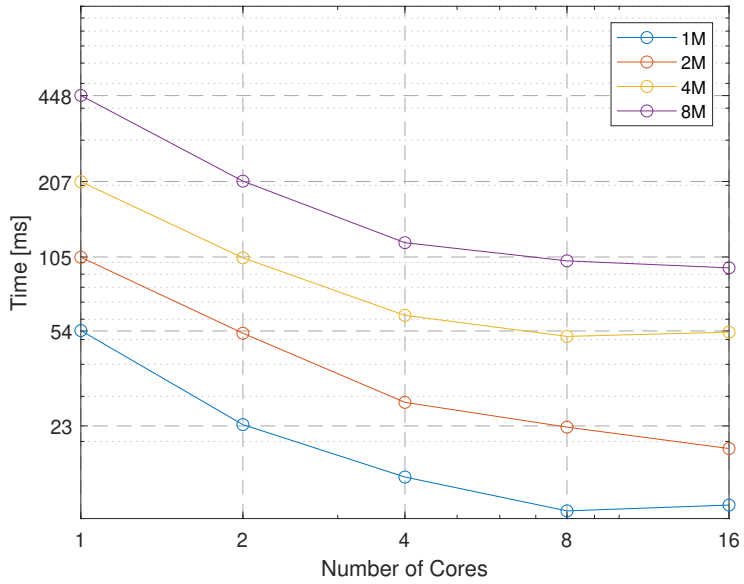

Figure 1: Measured time as a function of number of cores, for a few input sizes between 1 million values and 8 million values

In table 2 and figure 3, the speedup when paralleling the different inputs is visualized. Two means double the speed, i.e. the time is halved compared to one core, four quadruple the speed etc.

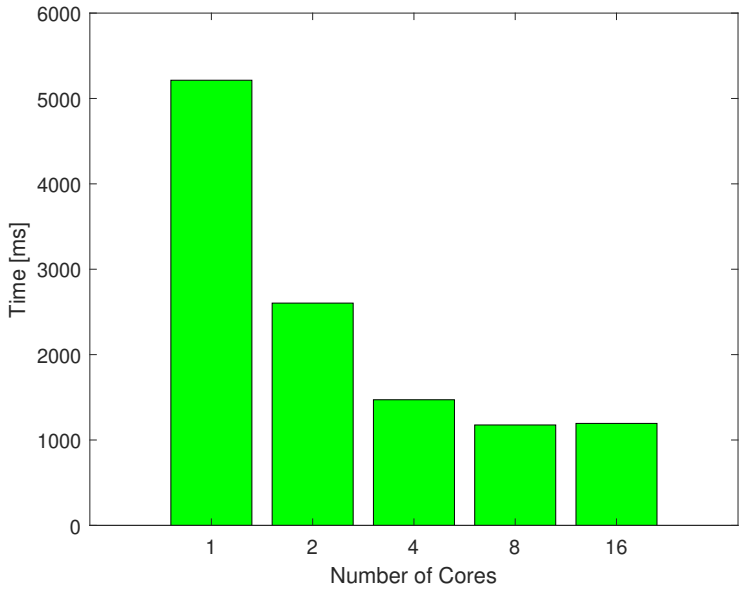

Figure 2: Measured time as a function of number of cores when using an input of 100 million values

Table 2: Speedup compared to one core

cores	1	2	4	8	16
optimal	1	2	4	8	16
1M	1	2.33	3.73	5.06	4.8
2M	1	1.98	3.68	4.6	5.58
4M	1	1.98	3.33	4.02	3.87
8M	1	2.16	3.76	4.42	4.71
100M	1	2	3.54	4.43	4.37

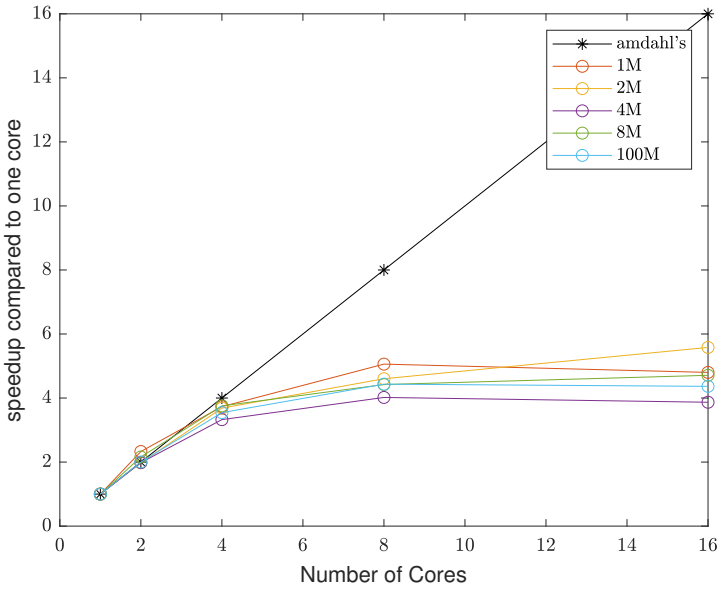

Figure 3: Speedup for the different measurements and the ideal speedup, for example Amdahl's law as explained in the Performance Experiments section

5 Discussion

Two main components of the performance were investigated. The strong scalability is when the input file remains the same but the number of cores increases, ideally dividing the workload between the cores and speeding up the total execution time. The strong scalability was investigated by comparing the speedup of the program to the ideal speed up. The weak scalability was also investigated. The weak scalability is when the input file changes as the number of cores increases in such a way that the number of calculations in each process constant. An example of this is to run the input file with one million elements on two cores, thus giving 500 000 elements to each core, and then running the input file with two million elements on four cores, which also gives 500 000 elements to each processing element.

The speedup from increasing the cores seem to follow the optimal linear speedup, up to four cores if the input size is sufficiently large. After this the speedup slows down and may even go down between eight and sixteen cores in some cases. Because of the almost linear speedup between one and four cores the input with one million elements with one core is equally as fast as the two million elements with two cores and the four million elements with four cores. This is extra visible by observing where the time for the different inputs crosses the horizontal line from 54 milliseconds in figure 1.

From these findings, the weak and strong scalability is as expected up to four cores. The strong scalability is as expected up to four cores since the speedup follows the ideal speedup closely as can be seen in figure 3. The weak scalability is as expected up to four cores since the time seems to remain somewhat constant when the input file changes in a way so that the number of elements in each process is constant as the number of processes are constant. This can be seen by following the horizontal lines in figure 1.

When running the program on 8 or 16 cores, the results are not as expected. The time appears to remain constant as more cores are added, regardless of the input data as can be seen in figure 3. This leads to the speedup no longer following the ideal speedup when checking the strong scalability and time increasing when checking the weak scalability as can be seen in figure 1. Using the top command on the Vitsippa Linux machine concluded that all assigned cores performed work, so there is no issue that fewer cores are used than what is expected. One reason for the slowdown could be that there were other people on the Vitsippa Linux server running other programs. This could lead to fewer system resources being available, especially when running on many cores, thus affecting the time measurements.

Another reason why the speedup slows down is that there are more messages in the system thus creating a large overhead. For two processes, each process sends four doubles for every stencil application, thus a total of eight doubles are being sent for every stencil application. If 16 cores are used, this number increases to 64 doubles for every stencil application. This means that as the number of processes go up, each processing element spends a larger proportion of the time sending, receiving or waiting for messages in relation to performing stencil applications. Since the number of messages sent from each process is constant, as the number of processes increases, the times approaches the times it takes to send and receive messages, i.e. a constant.