CS-E4690 SHEET 1

Simo Veijola, 1030419

October 10, 2024

1 Communication time measurement

We go through three different send and receive methods used and then represent the average communication times for each and measure the bandwidth of data transfer.

In the code it was chosen that 47 different sized data buffers are to be tested ranging from 1 all the way to 100000 element of datatype 'char', i.e. 100000 $1 - 100000$ bytes. For each buffer size, 1000 iterations were run. In each iteration timing was started at process with rank 0. First data was sent starting from rank 0 to rank 1, and then data was sent the other way. After rank 0 receives data from rank 1, the clock is stopped. This way we can go around the problem of needing to sychronize the MPI clocks. Usin the described way, send and receive is used twice, and thus the measured time has to be divided by two to get the average time for one send and receive. In the end the average of these 1000 measurements was taken and outputted.

We use 3 different send and receive routines. First routine named is the standard usage of functions MPI_Send and MPI_Recv. The second one replaces the MPI_Send with MPI_Bsend. The third one replaces both the send and receive functions with one MPI_Sendrecv. Of course now the situation has changes as in rank 1 we do not first wait for the data to come from rank 0, but rather send the data simultaneously. However, I believe this is okay given that the difference is explicitly recognized. The routines are named Comm=1, Comm=2, and Comm=3 in the figure 1

Next we show the averaged measurements

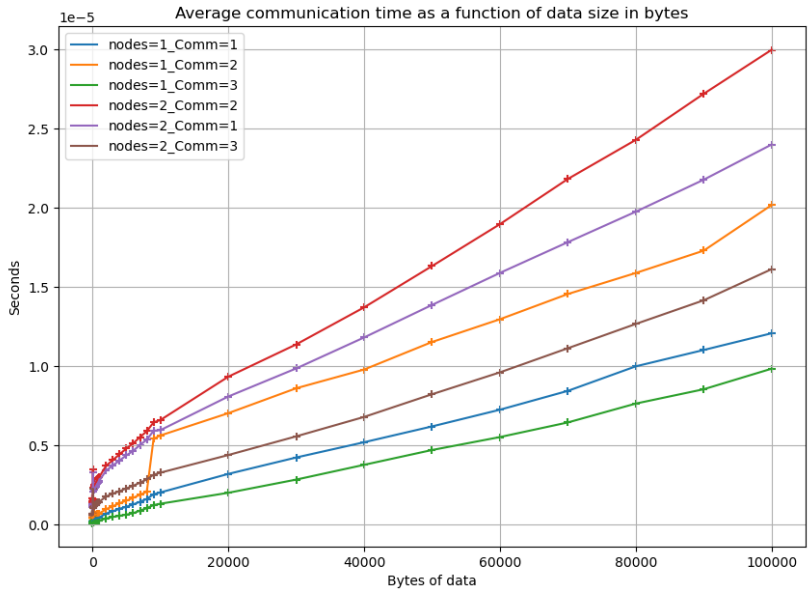


Figure 1: Average communication time per data size in bytes for three different send and receive routines between 2 processes on either 1 or 2 nodes. See the used routine for each Comm=x above.

Figure 1 shows the development of average communication times as a function of buffer size in bytes. It is quite noticable that in our defined range, the communication time behaves rather linearly or in

other words the communication time is linearly dependent on the communication size, and the slope of the dependency is dependent on the used method.

It is noticable that for small buffer sizes, the communication between two processes inside the same node are the fastest ones, and the communications between processes on different nodes is slower. However, at around size of 10000 bytes, the function Bsend becomes slower than Sendrecv between two separate node processes. There is a quite noticable jump here. It is not clear why this happens, but it may have something to do with attaching the associated buffer.

Overall use of Bsend is the slowest. This was somewhat unintuitive me, as we do not measure the attaching and allocating of the associated buffer in our measurements. I was expecting for it to be faster than MPI_Send as I hypothesized the normal Send would use Ssend as a subroutine for large arrays, and thus for it to be slower. Sendrecv was clearly the fastest one, which is not surprising as the code allows for more concurrency in the message sending. If I am correct, some networks allow data to be sent both ways simultaneously, which would make sendrecv highly advantageous.

Let us next calculate the bandwidth. For each method we take the buffer of size 100000 and calculate the bandwidth in gigabytes/s with respect to that. The approximate bandwidths are given in table below. It seems that the bandwidth is 1.5 to 2 times larger for data transfers inside a node vs between two different nodes.

Comm	GB/s for 1 node	GB/s for 2 nodes
MPI_Bsend	4.9	3.3
MPI_Send	8.3	4.2
MPI_Sendrecv	10	6.3

The calculation are done using the courses partition, which then utilized the nodes pe[xx] of the triton cluster. If we see the page Triton Overview, we can see that these nodes utilize Infiniband FDR. Wikipedia shows that it has a undirectional throughput of 109.08Gbit/s for 8 lanes, so if this means that this is the total amount of data that can be sent simultaneously, without considering direction, we may assume that this is what Triton utilizes.

2 Physical application

I will shortly describe the main solutions inside my code.

The communication problem was solved by defining a Cartesian coordinate grid for the processes, where neighbouring integer coordinate points define the processes neighbours. This was done using the MPI_Cart_create function, after which ranks of the neighbour were solved using MPI_Cart_shift function. The topology of torus was imposed by setting the Cartesian coordinates to be periodic in both x and y directions. The corner cases, where process does not have neighbours either up or down were solved by specifying the process itself to be its own neighbour.

The communicators that MPI_Cart_create creates were used to define the windows for one-sided communication for each process. These windows were then used along with MPI_Put to place the required data on the neighbouring processes. The direction of the neighbour, where the data is placed, was solved with respect to the velocity components in both x and y directions. Between the put operations synchronization with MPI_Win_fence was applied. In the beginning of an iteration, processes belonging to a window were synchronized. After that, while for all the processes inside a window MPI_Put functions were placing the data in neighbouring processes, the data points independent of the neighbouring data points were calculated. The points which can be calculated at this point were solved with respect to the velocity components. After calculating the locally dependent data, all the processes in a window were once again synchronized. This step assures, that moving of the data has been completed between the processes inside the current window. This signals a consent for the process to calculate rest of the data points, using the data placed in to its memory by neighbouring processes. This process was continued for specified amount of iterations.

As one has to place non-contiguous data in to the neighbours memory, MPI_type_vector was used with suitable block counts, blocks sizes, and strides. This allowed sending of subcolumns or subrows from the local data conveniently. Here at every iteration we have to communicate 2 rows of data and 2 columns of data, and the side depending on the velocity components. Thus, a vector type suffices.

Finally, results were compares against analytical ones, and time measurements for calculations with one-sided communication and without communication were done. Next we will present some acquired time measurements and compare the results of the discrete approximation against analytical solution.

Note that there are two code files: advec_wave_2D.c and advec_correctness.c, where the first one was used for time measurements and the second one for checking the correctness of the output. Essentially there is no other difference between the two, than that the first one has removed all the printing of solutions to files to exclude that time for the measurements.

2.1 Correctness

Correctness of the code was verified by debugging intermediate steps in the code. For example it was verified that the transferred data between processes were sent to correct memory placements, and that all the data had been sent before calculations dependent on that data were done.

Then the time step variable was adjusted, and the domain was partitioned finely enough to allow accuracy in the solutions. The data below represents the approximate, and analytical solution with 4 processes divided in direction x, y as 2 and 2, initial condition $c(x, y) = sin(x)$, velocity components of 0.5 in both directions, domain size of 200×200, time step $dt = 0.001257$, iteration count of 100, and total change in time $\Delta t = 0.1257$. Note that due to our initial condition the data is constant in y direction. Thus we have show here the data after final iteration for the process 0, that is subdomain $[0, \pi/2] \times [0, \pi/2]$. The data above are from the approximated solution and the data below are the analytical solution. As can be seen, approximately on average an error of magnitude $3 * 10^{-2}$ is present in the solution for these variables.

Also other initial condition functions, and velocity components (both negative and positive) were tested, and the solution was found to be close to the analytical one. It was also verified that solutions without one-sided communication were wrong.

```
-0.001276 0.061503 0.124039 0.186084 0.247394 0.307729 0.366849 0.424521 0.480518 0.534618
0.586608 0.636284 0.683448 0.727915 0.769510 0.808067 0.843435 0.875475 0.904060 0.929076
0.950426 0.968025 0.981804 0.991708 0.997699 0.999751 0.997859 0.992028 0.982282 0.968659
0.951214 0.930014 0.905145 0.876703 0.844801 0.809565 0.771134 0.729661 0.685307 0.638248
0.588671 0.536770 0.482752 0.426828 0.369219 0.310153 0.249864 0.188588 0.126568 0.064048

-0.031411 0.031411 0.094108 0.156434 0.218143 0.278991 0.338738 0.397148 0.453990 0.509041
0.562083 0.612907 0.661311 0.707106 0.750111 0.790155 0.827080 0.860742 0.891006 0.917754
0.940880 0.960293 0.975917 0.987688 0.995562 0.999507 0.999507 0.995562 0.987689 0.975917
0.960294 0.940881 0.917755 0.891007 0.860743 0.827082 0.790156 0.750112 0.707108 0.661313
0.612909 0.562085 0.509043 0.453993 0.397150 0.338740 0.278993 0.218146 0.156437 0.094111
```

2.2 Time measurement

Here the process time was measured for different amount of iterations and different sizes of discrete points. In theory as we are communicating only 2 rows and 2 columns at each iteration, as we increase the number of discrete points in each dimension, the fraction of required data from other processes decreases. Then, if we ere to increase the number of discrete points, the computation time should approach the time that it takes for the function without communication, as the calculation of locally dependent data takes longer than the retrieval of outside data.

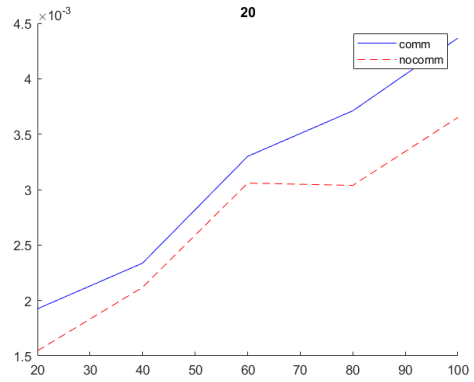
To see this let nx, ny depict the number of discrete points in the subdomain in x and y directions respectively. Then we require $2 * nx + 2 * ny$ data points from other processes. Now the fraction of data we need from other processes approaches zero as $nx * ny$ approaches infinity

$$\lim_{nx \to \infty, ny \to \infty} \frac{2nx + 2ny}{nxny} = 0,$$

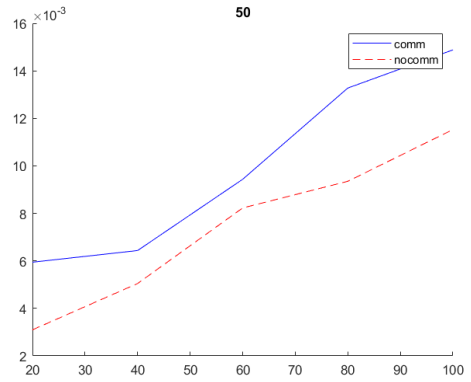
since the numerator is $O(nx + ny)$ and the denumerator is $O(nxny)$.

Figure 2 shows this theory in practice. For smaller domain sizes (smaller subdomain size), the calculation with communication takes a bit longer than without it. When we increase the domain size, both take approximately the same time. Here only one measurement was made for each point, which induces inaccuracy. This is because Triton was sometimes clogged and it took a considerable time to even get these measurements done. However, even from this data, we can be quite sure that this is what is happening, as measurement for many different iteration counts were however made.

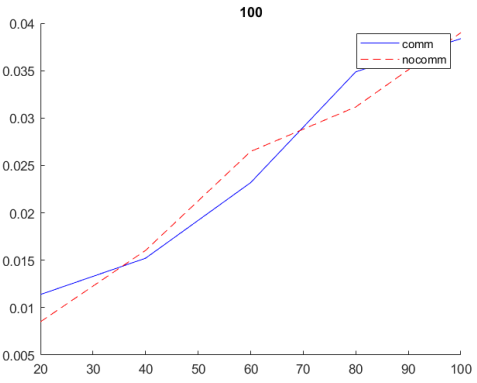
This shows us that the level of concurrency in the program approaches the number of nodes.



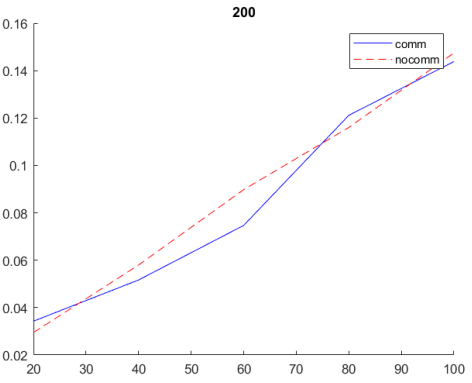
(a) Times in domain 20×20 with respect to iteration count



(b) Times in domain 50×50 with respect to iteration count



(c) Times in domain 100×100 with respect to iteration count



(d) Times in domain 200×200 with respect to iteration count

Figure 2: Computation times with one-sided communication and without it for different discretizations, plotted with respect to iteration count