Università della Svizzera italiana

Institute of Computing CI

High-Performance Computing Lab **Institute of Computing**

Student: Dennys Mike Huber Discussed with: Alberto Finardi & Leon Ackermann

Solution for Project 5

1. Parallel Space Solution of a nonlinear PDE using MPI

1.1. Domain decomposition

The domain was decomposed using the non-periodic 2D Cartesian domain decomposition strategy, where the grid is divided into smaller rectangular subgrids, which are then assigned to a MPI processes. This method is suited for problems where you have grid and update the values on said grid with a stencil. Furthermore in our case the work is evenly distributed over the grid, which makes this method ideal, because it is simple and results in a good work balancing. The topoloy created by the 2D Cartesian domain, also leads to efficient exchanging of neighboring ghost cells which reduces the communication overhead significantly. Another plus point is that it is very scalable and can be easily used with varying number of processes.

1.2. Linear algebra kernels

The primary rational used in order to decide which functions needed to be updated with MPI code was if the function needed to compute a result from distributed data across the processes or if the computation are done independently on each process. The two functions that modified were:

- `hpc_dot`: Added a `MPI_Allreduce` in order to compute the global sum of the inner local product values.
- `hpc_norm2`: Similar to the dot product the same MPI function was used to computed the global squared norm.

All the other functions can be used locally on the subdomain without needing to share the result, therefore no modification was needed.

1.3. The diffusion stencil

The non-blocking point-to-point MPI communication was used to exchange the ghost cells between processes. The non-blocking approach was used to overlap communication and therefore improve performance. As the initial step the ghost buffers are filled with the boundary values from the local domain, corresponding to the direction they are sent. Furthermore the buffers for the receiving ghost cells are initialized. Next the ghost are exchange using a non-blocking communication using `MPI_Isend` and `MPI_Irecv`, the order in which the send and received are executed does not matter due to the communication being non-blocking, but it is important that there is for each send there is a corresponding receive command. I decided to group them by direction as seen in Listing 1.

```
  // Send/recv east
  MPI_Isend(buffE.data(), buffE.xdim(), MPI_DOUBLE, domain.neighbour_east, 0,
            domain.comm_cart, &send_req[0]);
  MPI_Irecv(bndE.data(), bndE.xdim(), MPI_DOUBLE, domain.neighbour_east, 1,
            domain.comm_cart, &recv_req[1]);
  // Send/recv west
  MPI_Isend(buffW.data(), buffW.xdim(), MPI_DOUBLE, domain.neighbour_west, 1,
            domain.comm_cart, &send_req[1]);
  MPI_Irecv(bndW.data(), bndW.xdim(), MPI_DOUBLE, domain.neighbour_west, 0,
            domain.comm_cart, &recv_req[0]);

  // Send/recv north
  MPI_Isend(buffN.data(), buffN.xdim(), MPI_DOUBLE, domain.neighbour_north, 2,
            domain.comm_cart, &send_req[2]);
  MPI_Irecv(bndN.data(), bndN.xdim(), MPI_DOUBLE, domain.neighbour_north, 3,
            domain.comm_cart, &recv_req[3]);

  // Send/recv south
  MPI_Isend(buffS.data(), buffS.xdim(), MPI_DOUBLE, domain.neighbour_south, 3,
            domain.comm_cart, &send_req[3]);
  MPI_Irecv(bndS.data(), bndS.xdim(), MPI_DOUBLE, domain.neighbour_south, 2,
            domain.comm_cart, &recv_req[2]);

// the interior grid points
  for (int j = 1; j < jend; j++) {
    for (int i = 1; i < iend; i++) {
      f(i, j) = -(4. + alpha) * s_new(i, j)          // central point
                + s_new(i - 1, j) + s_new(i + 1, j) // east and west
                + s_new(i, j - 1) + s_new(i, j + 1) // north and south
                + alpha * s_old(i, j) +
                beta * s_new(i, j) * (1.0 - s_new(i, j));
    }
  }
  MPI_Waitall(4, recv_req, MPI_STATUSES_IGNORE);
```

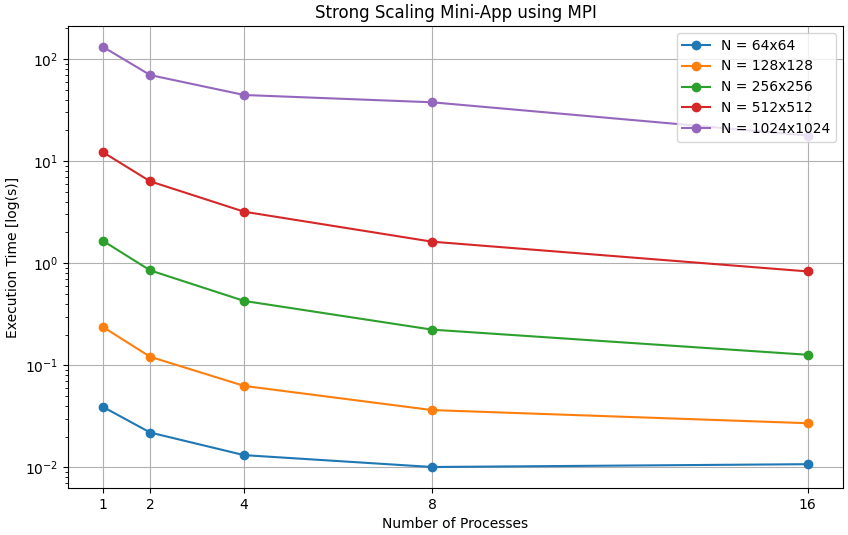
Listing 1: Non-blocking send and receive pattern

The main reason why we use non-blocking communication is to overlap communication with computation, hence it is of upmost importance where the wait clause is placed. Because for computing the inner grid points the ghost cells are not necessary, we only need to ensure that all messages were received after the inner grid points calculation and before the boundary grid points are computed. In Listing 1 the `MPI_Waitall` for all the receiving requests is placed after the interior grid points for computed.

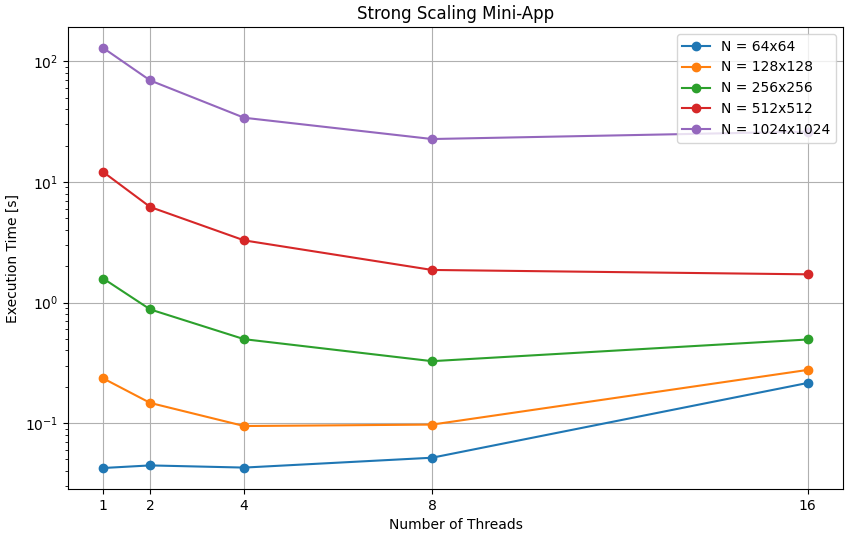
1.4. Strong scaling

The graphs seen in illustration 1 show the strong scaling behavior of the mini application using MPI 1a and OpenMP 1b for its implementation. For both implementation were run on a single node on the Rosa cluster. Analyzing the MPI implementation we can clearly see that the execution time decreases for all grid sizes. Larger grid sizes benefit more from the parallelization, due to the computation outweighing the communication overhead. For smaller grid sizes on the other hand the reduction of execution flattens out when the communication overhead becomes large in relation to computation.
Comparing this result to the restults from Project 3, we can observe that MPI achieves better across all resolutions, especially for larger grids size due to iths distributed memory model. OpenMP on the other hand struggles when the thead size increases, especially for smaller grids. Furthermore, by enabling overlapping communication with computation, the MPI version redues idle time and improves scaling, compared to the OpenMP, where the memory is shared and this lead to potential bottlenecks, especially for larger thread count. In conclusion for small and medium grid sizes the OpenMP is more suited due to it performing very similar to the MPI version and it is a lot easier and quicker to implement. For large grid sizes on the other hand it is advised to take the time and

implement the MPI version.



(a) MPI



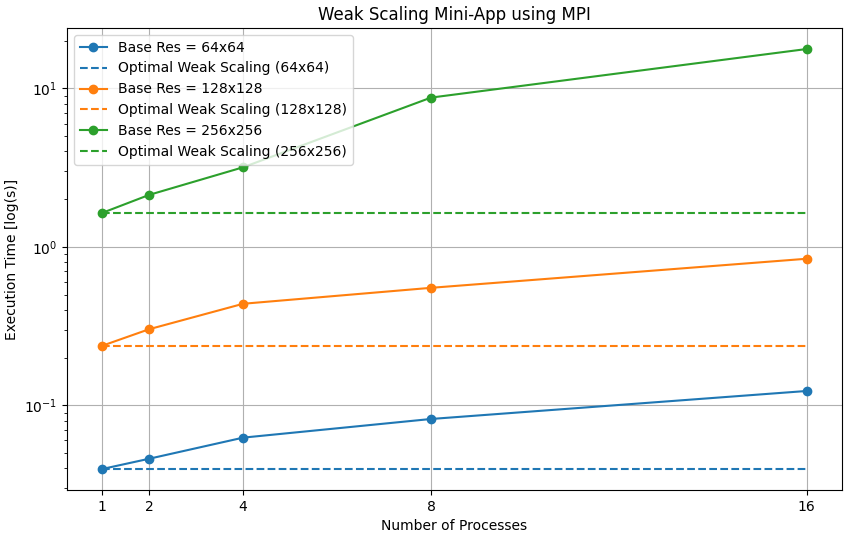
(b) OpenMP

Figure 1: Strong Scaling

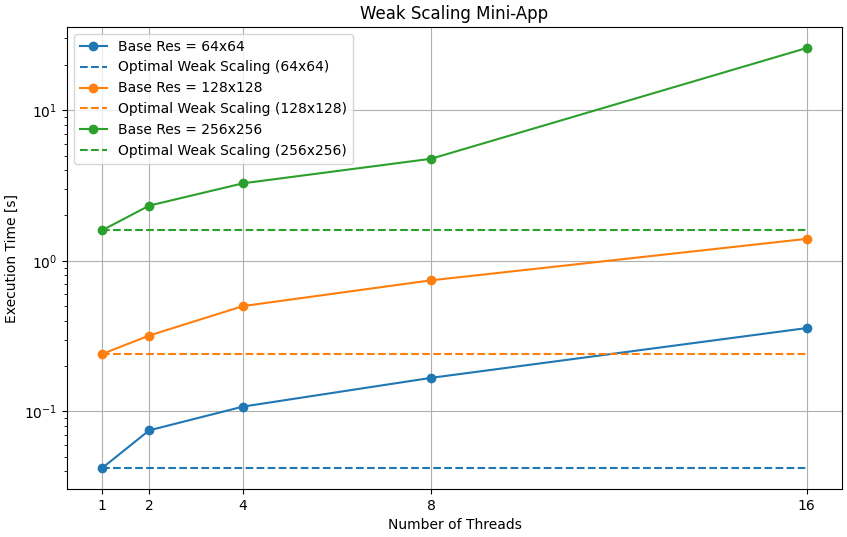
1.5. Weak scaling

The weak scaling of the mini-app for the MPI (2a) implementation, the efficiency decreases by a lot when increasing the base resolution, which is increased by multiplying the base resolution by the square root of the number processes. Larger base resolutions have stronger in increase of execution time compared to lower base resolution, this indicates inefficiencies as the problem size increases per process.
Comparing it to the result obtained in Project 3 for the OpenMP implementation, which can be seen in Subfigure 2b. In terms of scaling efficiency the MPI outperforms the OpenMP version especially for larger problem sizes with higher number of processes. This shows that the MPI implementation handles large grid size more effectively, while the OpenMP's performance degrades. One of the factor attested to that is the higher overhead usually encountered in application that use OpenMP. Eventhough the mini-app is rather inefficient, we can clearly see that the MPI version is closer

to the optimal scaling indicated by the dashed lines. Overall the execution times are consistently lower for the MPI implementation. In conclusion MPI is the preferred choice especially if you have a larger workload, but it is also comes with the trade off of being more time intensive to implement.



(a) MPI



(b) OpenMP

Figure 2: Weak Scaling

2. Python in High-Performance Computing

2.1. Sum of ranks

The sum of ranks code performs a ring-based communication among the processes using MPI. In my implementation each process sends or receives data from or to its neighbors based on its rank. The process are divided in two disjointed groups, based on if their rank number is even or odd. This two groups are then alternating between sending and receiving. Each process has a local sum variable that sums up all the values and sends the last received rank to its neighbor with the `dest=(rank + 1) %size` After the number of rounds equal to the number of processes, each process sum variable holds the sum of all ranks. Because this version of sum of ranks was implemented with

mpi4py, there are two possible ways to implement this, one using the pickle-based communication of generic Python objects and the other the near C-speed direct array communication. These are the differences in implementation between the two:

- **Pickle-based (lowercase)**: In this version you can send generic python object making the implementation straight forward. One can just put the generic python object and the function call and then save the return value of the receive function in an other variable.
- **Direct-array (uppercase)**: In this version the communication is done by using a buffer-like object, where the arguments need to be specified explicitly by using a tuple for example `[data, MPI.DOUBLE]` and also returns a tuple by using refernces and not a specific return statement.

2.2. Ghost cells

This program is an reimplementation of the ghost cells in Project 4 in mpi4py. We construct a cartesian grid dynamically using the `MPI.Compute_dims` function. Each process sends its ghost cells to its neighbours (north, east, south, west). The communication is done using the (Sendrecv), where the ghost cells are send in one direction and at the same time received from the opposite one.

```
# S:left/R:right
recv_temp = np.empty(SUBDOMAIN, dtype=np.float64)
domain_send = DOMAINSIZE + 1
domain_recv = DOMAINSIZE + (SUBDOMAIN + 1)
comm_cart.Sendrecv(
    [data[domain_send:], 1, T_col],
    dest=rank_left,
    sendtag=0,
    recvbuf=[recv_temp, MPI.DOUBLE],
    source=rank_right,
    recvtag=0,
)

data[domain_recv : domain_recv + SUBDOMAIN * DOMAINSIZE : DOMAINSIZE] = recv_temp
```

Listing 2: Ghost cell Sendrecv example

In Listing 2 an example `Sendrecv` is shown, where we assign a temporary variable for the incoming buffer, set the starting point of the domain of the continously stored data for both the receiving and sending ghost cells. Subsequently the data is send and received using a costume type named `MPI_Datype T_col`. Finally we put the received data into the local matrix.

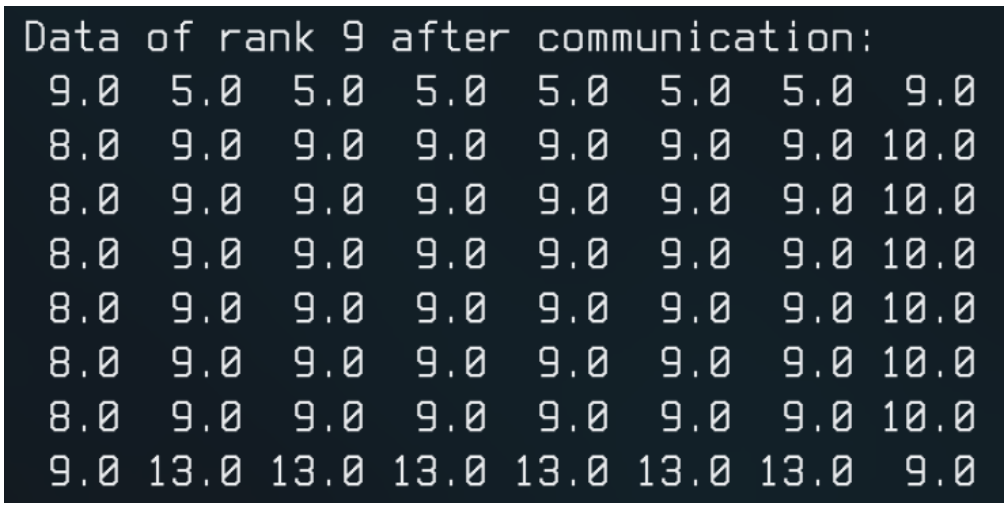


Figure 3: Output for ghost cells exchange for rank 9 with 16 processes

2.3. A self-scheduling Parallel Mandelbrot

The manager-worker paradigm handles the workload by dynamically efficiently distributing it among the process. This is especially useful when we deal with irregular workloads, such as the Mandelbrot set. My implementation uses a queue where all the subbatches are stored and as soon a process is done with the subbatch the next one in the queue is assigned to the worker, which in our case is just a process. This ensures efficient load is dynamically redistributed depending on availability of the worker. This comes at a cost of having higher communication overhead due to having more data exchanges and the manager can potentially act as a bottelneck.
For our scaling analysis the Mandelbrot domain is fixed and of sizes 4001×4001. We split the workload into 50 and 100 tasks respectivelly. In Figure 4 we can observe that execution time decreases significantly when the number of workers is increased. Beginning from 8 workers onwards we can see that the efficiency starts to decrease to do increased communication overhead, compared to workload per worker. Looking at the different number of tasks, the blue line indicating a 100 tasks scales slightly better in this case the granurality results in a better load balancing which is counter acted by the prevously mentioned costs. This result shows that the load balancing and the cost pretty much offset and there is not a significant difference between the number of tasks in the case of the mandelbrot set.

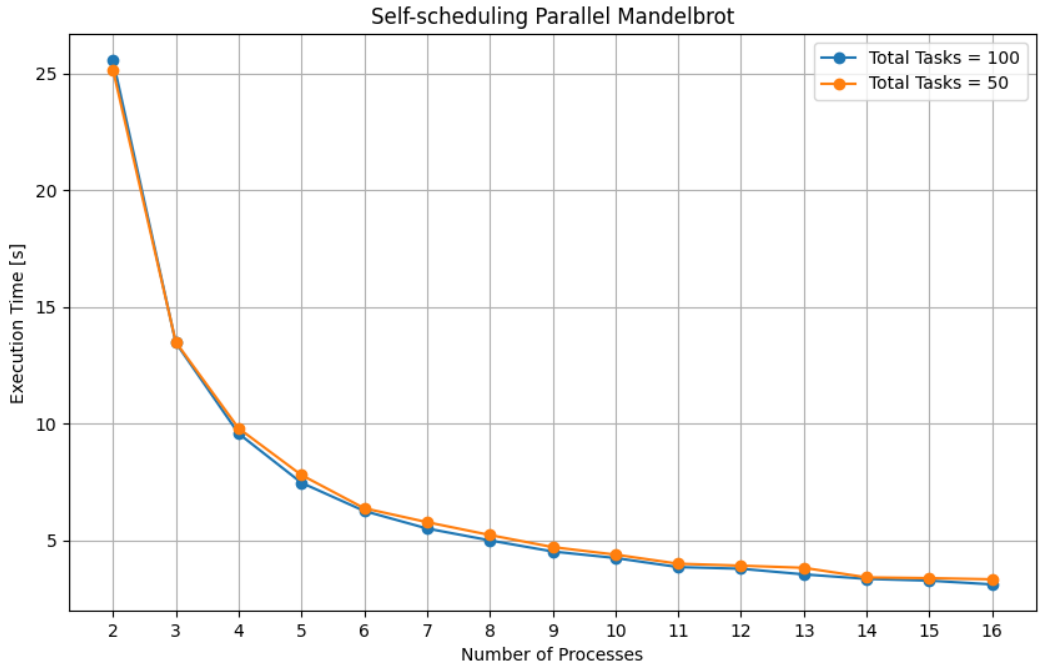


Figure 4: Manager Worker scheme scaling with different number of tasks