Università della Svizzera italiana | **Institute of Computing CI**

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

Student: Fabian Gobet Discussed with: Ekkehard Steinmacher, Henrique Gil

Solution for Project 7

1. Parallel Space Solution of a nonlinear PDE using MPI [in total 35 points]

1.1. Initialize and finalize MPI [5 Points]

To initialize MPI, like in prior exercises, we start by using MPI_init() along with MPI_Comm_rank() and MPI_Comm_size().

```
int main(int argc, char* argv[])
    ...
    int mpi_rank, mpi_size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &mpi_size);
    ...
```

And in the end we free the cartesian communicator memory allocation and finalize MPI.

```
    ...
    MPI_Comm_free(&domain.comm_cart);
    MPI_Finalize();
    return 0;
}
```

1.2. Create a Cartesian topology [5 Points]

Next, we want to create a cartesian topology communicator with no periodicity, and retrieve the the rank coordinates and the east,west,south and north neighbours as well.

As such, we begin by setting up the dimensions with the use of MPI_Dims_create() and then create the said topology using MPI_Cart_create()

```
    int dims[2] = { 0, 0 };
    MPI_Dims_create(mpi_size, 2, dims);
    ndomy = dims[0];
    ndomx = dims[1];
    int periods[2] = { 0, 0 };
    MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 0, &domain.comm_cart);
```

Having done this, we want to retrieve the coordinates for the rank and the neighbours using MPI_Cart_coords() and MPI_Cart_shift().

```
    MPI_Cart_coords(domain.comm_cart, mpi_rank, 2, coords);
    domy = coords[0]+1;
    domx = coords[1]+1;

    MPI_Cart_shift(comm_cart, 0, 1, &neighbour_south,&neighbour_north);
    MPI_Cart_shift(comm_cart, 1, 1, &neighbour_west, &neighbour_east);
```

1.3. Extend the linear algebra functions [5 Points]

In order to compute the dot product of vectors all we have to do is to pair-wise iterate through these, multiply their components and add the result to a running sum variable.

Since each process will be computing its part of the dot product but all need the final aggregated value, we use MPI_Allreduce() to achieve this goal.

```
double hpc_dot(Field const& x, Field const& y)
{
    double result = 0;
    double result_global = 0;
    int N = y.length();

    for (int i = 0; i < N; i++)
        result += x[i] * y[i];

    MPI_Allreduce(&result, &result_global, 1, MPI_DOUBLE, MPI_SUM, data::domain.comm_cart);
    return result_global;
}
```

Notice that the L2 norm is a dot product of the vector bye itself, with the result root-squared. Hence, we can call the previously defined function hpc_norm() to achieve this.

```
double hpc_norm2(Field const& x)
{
    return sqrt(hpc_dot(x,x));
}
```

1.4. Exchange ghost cells [10 Points]

To exchange ghost cells we need to define first a running account for the number of communications a given rank does, as well as an array of statuses and requests made (statuses are not mandatory to be kept for this exercise, but might be useful for debugging purposes).

```
    MPI_Status statuses[8];
    MPI_Request requests[8];
    int num_requests = 0;
```

We are now in conditions of beginning communications with neighbouring ranks. A similar block of code will be executed for each of the neighbours, only changing the respective variables.

In order to not show redundant information, I will provide the exchange example for the north neighbour, being that all other cases can be read in the code files.

The ideia here is to check first wether the north neighbour exists or not, and if so then start an asynchronous message receiving process, allocating the message received to the bndN variable. The length of this data will be equal to the size of width assigned to that partition (nx).

Once this is done, we then copy the data to send into a buffer and proceed to to an asynchronous message send in similar fashion.

```
if(domain.neighbour_north>=0) {
    MPI_Irecv(&bndN[0], nx, MPI_DOUBLE, domain.neighbour_north, domain.neighbour_north,
        domain.comm_cart, requests+num_requests);
    num_requests++;

    for(int i=0; i<nx; i++)
        buffN[i] = s(i,ny-1);

    MPI_Isend(&buffN[0], nx, MPI_DOUBLE, domain.neighbour_north, domain.rank,
        domain.comm_cart, requests+num_requests);
    num_requests++;
}
```

Once all the code for the communications as been executed, we can immediatly compute the inner points, whereas for the boundary we have to wait until comunications for the given rank have seized. This can be achieve by using MPI_Waitall() on the requests and statues that we have previously created. Notice that the number of requests is used in such manner that we know exactly how many requests were created for the messages receive.

```
MPI_Waitall(num_requests, requests, statuses);
```

1.5. Scaling experiments [10 Points]

By performing a strong analysis in powers of 2 of number of processes vs. the iterations per second for each grid size we can obtain the following graph.

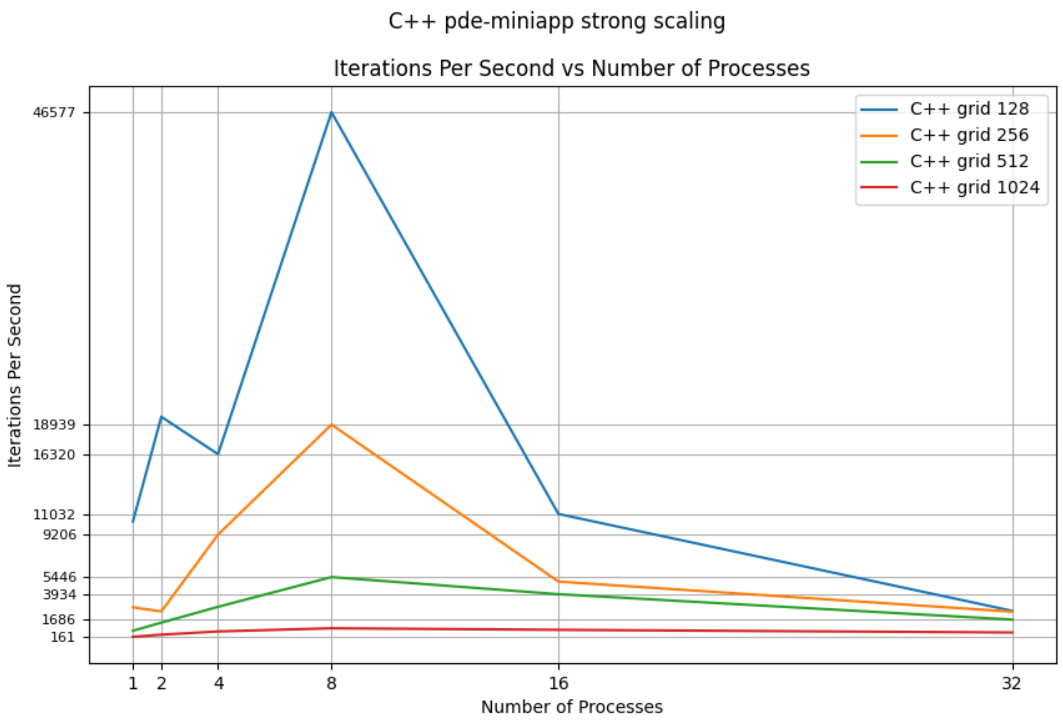

Figure 1: pde-miniapp strong scaling

One can observe that the overall behaviour for all grid sizes is the same, being that there is maximal performance when running 8 processes. Furthermore, comparing each of the grid size cases, running a 128x128 grid size with 8 processes in a single compute node (Macbook Pro 16) seems to be the most efficient.

To perform a weak analysis a single process was considered in a 128x128 grid as the initial case, raising the number of processes to 64 which equates to a 1024x1024 grid.

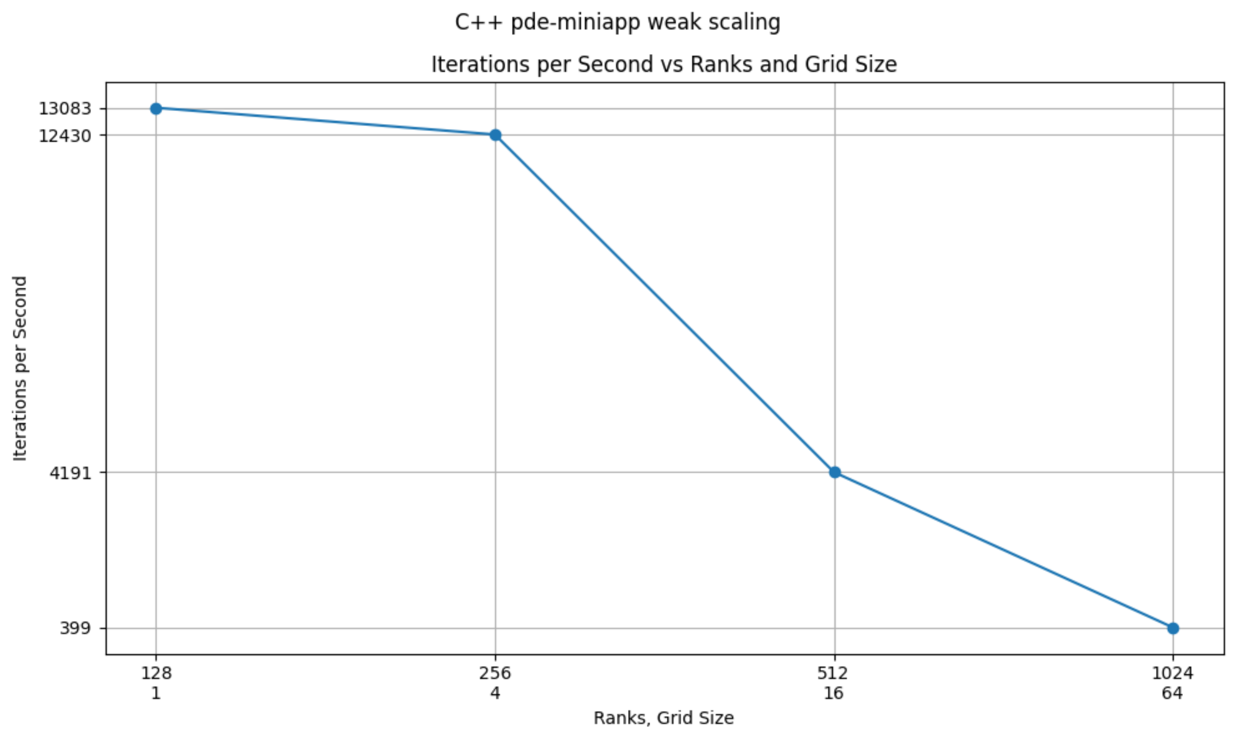

Figure 2: pde-miniapp weak scaling

We can observe from the plot that efficiency slightly drops when maintaining workload per process constant and raising the grid size to 256x256. On the other hand, for values grid sizes bigger than the former, efficiency drops dramatically.
Note that there is a direct correlation between efficiency and the number of iterations per second.

2. Python for High-Performance Computing (HPC) [in total 50 points]

Before addressing the exercises, because there were conflicting libraries of MPI (conda and homebrew), I had to resort to the use of a virtual environment which cannot be included in the submission due to its size. Nevertheless, i will include all steps taken in order to achieve this.

In order to create the virtual environment and activate it, the following commands should be executed in the terminal.

```
python -m venv myenv
source ./myenv/bin/activate
```

Once created and activated, we must then source the path for MPI and install mpi4py in correlation to the installed version of MPI. We can also install in the mean time matplotlib and pandas which will be useful later on.

```
which mpirun
MPICC=/opt/homebrew/bin/mpicc pip install mpi4py
pip install matplotlib
pip install pandas
```

Next, we have to account for the fact the the MPI version called by mpi4py might not be the same as the one installed. Therefore, we must create a symlink for a dummy prior version that points into the current version. This step must be taken with care as it might not always work given the version of mpi4py and homebrew MPI.

```
ln -s /opt/homebrew/lib/libmpi.40.dylib /opt/homebrew/lib/libmpi.12.dylib
ln -s /opt/homebrew/lib/libmpi_usempi_ignore_tkr.40.dylib /opt/homebrew/lib/
    libmpi_usempi_ignore_tkr.12.dylib
ln -s /opt/homebrew/lib/libmpi_usempif08.40.dylib /opt/homebrew/lib/libmpi_usempif08.12.
    dylib
```

Finally, every time the we start the virtual environment we have to export some variables.

```
export DYLD_LIBRARY_PATH=/opt/homebrew/lib:$DYLD_LIBRARY_PATH
export TMPDIR=/tmp
```

2.1. Sum of ranks: MPI collectives [5 Points]

On this exercise we could approach it by making so that all processed have the resulting value or only one (all-reduce vs. reduce). Therefore, I've implemented both and left one of the ways commented. For both lower case methods we don't need numpy, but because upper case methods uses a buffer, we will be importing both numpy and mpi4py. Furthermore, once mpi4py has been imported, we can immediately access the communication world and the size of it

```
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
```

For the lower case methods, given a communicator comm, we can use the reduce() and allreduce() methods in similar fashion to the c++ version, specifying only the sending object, the type of agregate operation (and the root in case of reduce).

```
# LOWER CASE
total_sum = comm.allreduce(rank, op=MPI.SUM) # all ranks have the sum value
#total_sum = comm.reduce(rank, op=MPI.SUM, root=0) # only rank 0 gets the sum value
if rank == 0:
    print(f"Total Sum (Pickle-Based): {total_sum}")
```

For the upper case methods we need to define a buffer that will hold the sum of all values, and a array tfor the value to send. Because in both cases it will be a single integer value, we may initialize a numpy array containing one zero of dtype integer for the sum, and an array containing the rank number for the sending value.

```
rank_array = np.array(rank, dtype='i')
sum_array = np.empty(1, dtype='i') # all ranks have the sum value
```

having done this, we can now use comm.Allreduce(), passing in as arguments the value to send, the buffer to receive and the agregator operation.

```
'''
if rank == 0: # only rank 0 gets the sum value
    sum_array = np.empty(1, dtype='i')
else:
    sum_array = None
comm.Reduce(rank_array, sum_array, op=MPI.SUM, root=0)
'''
comm.Allreduce(rank_array, sum_array, op=MPI.SUM)
if rank == 0:
    print(f"Total Sum (Array-Based): {total_sum}")
```

The python script for this exercise can be found in the hpc-python folder under the name rankSum.py.

2.2. Domain decomposition: Create a Cartesian topology [5 Points]

For this exercise, we first start out by importing MPI from mpi4py and numpy and initializing the rank and comm size, as done in the previous exercise.

```
from mpi4py import MPI
import numpy as np
comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()
```

We then use the MPI.compute_dims() function with the size and a generic [0,0] list that serves the same purpose as in the c++ version, for the exception that now the value is written to a new variable dims.

Having done this, we define the periods in a list and create the cartesian topology comunicator with the MPI.Create_cart() function, setting the reorder flag to false to keep rank numbering, analogous to C++ version. Then, we can fetch the coords for the given rank using the communicator's method .Get_coords().

```
dims = MPI.Compute_dims(size, [0, 0])
periods = [False, False]  # Non-periodic in both dimensions
cart_comm = comm.Create_cart(dims, periods=periods, reorder=True)
coords = cart_comm.Get_coords(rank)
```

Similarly to C++, we can also fetch the adjacent neighbour with the use of comm.Shift() method, which takes in as arguments the dimension and displacement.

```
east, west = cart_comm.Shift(0, 1)  # Shift along the first dimension (x-axis)
north, south = cart_comm.Shift(1, 1) # Shift along the second dimension (y-axis)
```

2.3. Exchange rank with neighbours [5 Points]

To start communications withing neighbouring ranks, i first define a map for each neighbour that will save the hold the respective rank value, and define a function that both sends and receives the message withing neighbours, as long as they exists. We can check for non existing neighbours using the shift() method and the MPI.PROC_NULL flag

```
# Initialize variables for neighbor ranks
neighbor_ranks = {'east': None, 'west': None, 'north': None, 'south': None}
def exchange_rank(neighbor, direction):
    if neighbor != MPI.PROC_NULL:
        send_rank = np.array(rank, dtype='i')
        recv_rank = np.empty(1, dtype='i')
        cart_comm.Sendrecv(sendbuf=send_rank, dest=neighbor, recvbuf=recv_rank, source=
            neighbor)
        neighbor_ranks[direction] = recv_rank[0]
```

Finally, I proceed to exchange messages between all neighbouring ranks using the previously defined function, and print out the result for each one of them.

```
# Exchange ranks with neighbors
exchange_rank(east, 'east')
exchange_rank(west, 'west')
exchange_rank(north, 'north')
exchange_rank(south, 'south')
# Output the results
print(f"Rank: {rank}, Coordinates: {coords}, Neighbors - East: {east}, West: {west}, North:
    {north}, South: {south}")
print(f"East: {neighbor_ranks['east']}, West: {neighbor_ranks['west']},",
      f"North: {neighbor_ranks['north']}, South: {neighbor_ranks['south']}\n")
```

2.4. Change linear algebra functions [5 Points]

In order to compute the inner product like we do in the C++ version, we have to notice that the inputs will be ojects of type Field, and the matrix holding the needed values is found at Field.inner.

One way of computing the inner product could be done by reshaping the matrices into a vector, and iterate through these and compute a running sum of the element wise multiplication. On the other hand, we could use numpys element wise multiplication operation and then employ the use of a matrix full sum function.

In terms of performance, the second option, using numpys functionalities, offers a performance that is up to x6 faster than the iterative method. As such, I will be employing the use of that method whilst leaving the other commented.

Analogous to the C++ version, once the inner product is computed, we need to perform an Allreduce operation to agregate the values of each process.

```
def hpc_dot(x, y):
    """Computes the inner product of x and y"""
    '''
    a = x.inner.reshape(-1)
    b = y.inner.reshape(-1)
    sum = 0.0
    for i,j in zip(a,b):
        sum = sum + i*j
    sum = np.array(sum, dtype='d')
    sum_array = np.empty(1, dtype='d')
    x.domain.comm.Allreduce(sum, sum_array, op=MPI.SUM)
    '''
    rank_array = np.sum(x.inner * y.inner)
    sum_array = np.empty(1, dtype='d')
    x.domain.comm.Allreduce(rank_array, sum_array, op=MPI.SUM)
    return sum_array.item()
```

Similarly to the C++ implementation, we can use the dot product function to calculate the L2 norm.

```
def hpc_norm2(x):
    return np.sqrt(hpc_dot(x,x))
```

2.5. Exchange ghost cells [5 Points]

The exchange procedure will be similar to the C++ version in terms of communication. To achieve this, I first create 2 lists into the object itself (self), sendRequests and recvRequests.

Having done this, I then see what neighbours are defined in the domain (¿=0). For each defined neoighbour i use Irecv() in the domain communicator, sending the respective boundary to the respective neighbour. Then, I copy the inner border values to the respective buffer, and use Isend() method on the domain communicator, sending the buffer to the respective neighbour.

Each process of communication returns a request which are appended to the send and receive requests lists. This will be useful later on.

```
self.recvRequests = []
self.sendRequests = []
domain = self._domain # copy for convenience
if domain.neighbour_north>=0:
    self.recvRequests.append(domain.comm.Irecv(self.bdryN, source=domain.neighbour_north))
    self._buffN = self.inner[:, -1].copy()
    self.sendRequests.append(domain.comm.Isend(self._buffN, dest=domain.neighbour_north))
```

To avoid redundancy, only neighbour north is shown, whereas the other 3 follow a very similar implementation.

Having implemented the exchange_startall function, we now have list of requests that we can use to employ the Waitall function, in a similar fashion to C++ implementation.

```
def exchange_waitall(self):
    """Wait until exchanging boundary field data is complete"""
    MPI.Request.Waitall(self.recvRequests)
```

To make sure everything is working as intended, I ran the code with similar initial inputs as the C++ version and used draw.py to get the respective image, thus confirming that the program is working as intended.

```
mpiexec -n 4 python ./main.py 128 100 0.005
./draw.py
```

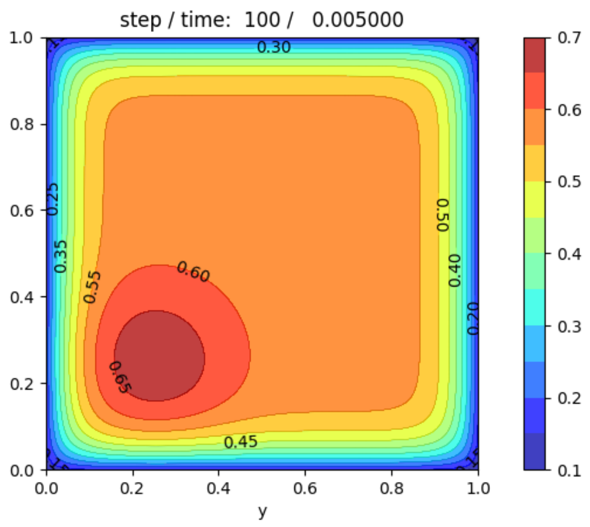

Figure 3: hpc-python, pde-miniapp result

2.6. Scaling experiments [5 Points]

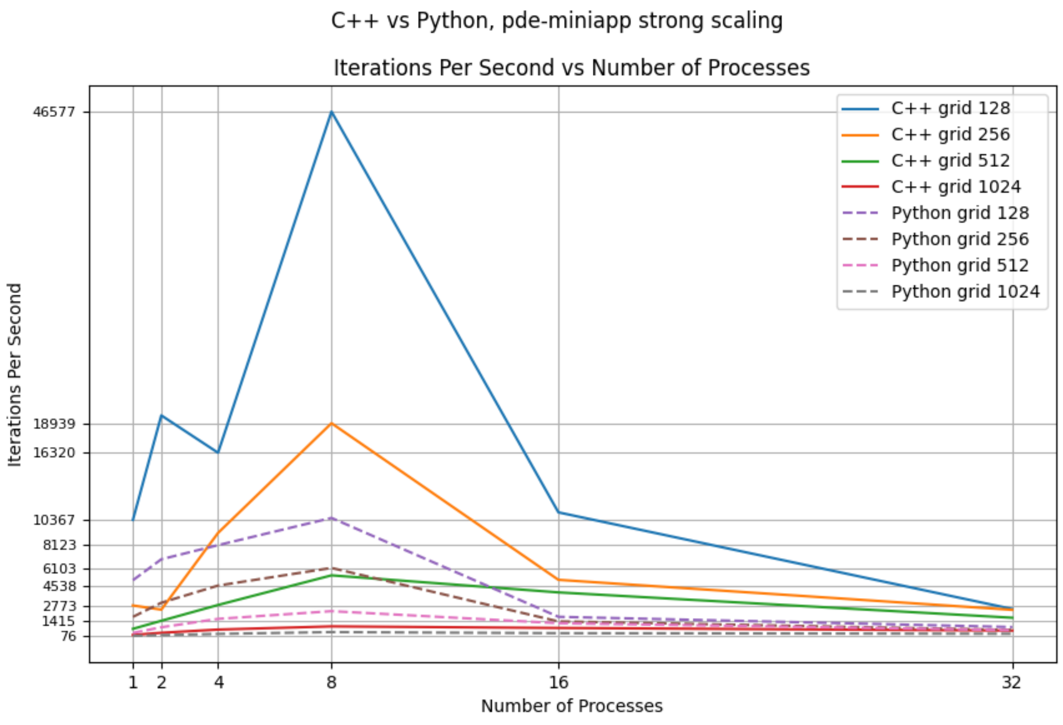

Figure 4: hpc-python, pde-miniapp strong scaling

We can observe that the strong scaling plot for the python version seems to follow the same gradient tendencies of the C++ strong scaling. As far as the efficiency of python compared with C++, we can observe the expect degrading efficiency of the python version

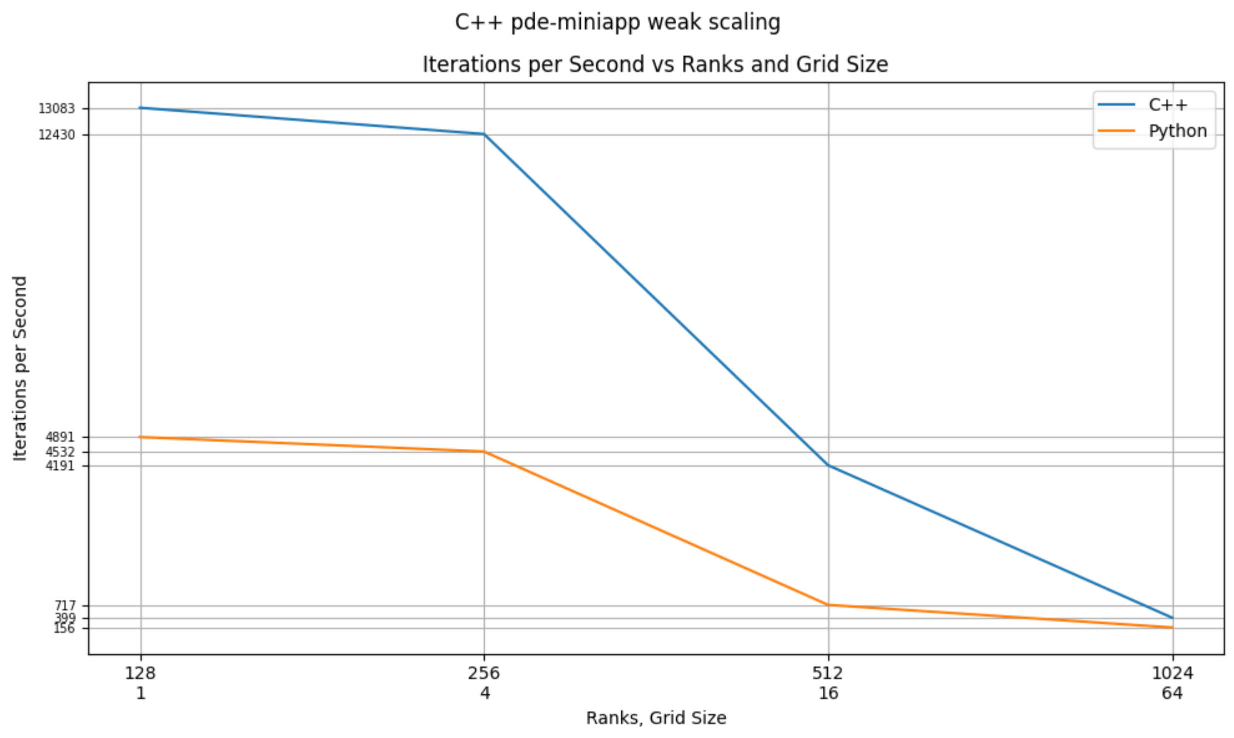

Figure 5: hpc-python, pde-miniapp weak scaling

The same thing can be said in regards to the weak analysis. Gradient tendencies of the plot are equal on both python and C++, but python does show far less efficient values than the C++ version.

2.7. A self-scheduling example: Parallel Mandelbrot [20 Points]

The idea for the implementation of the manage worker script is to have a manager that distributes tasks to the set of workers in the MPI communication world, and then proceeds to wait for responses and continuously give out tasks until no more tasks are available. After handing out all the tasks, the manager then proceeds to wait to receive the result of the remaining ones.

So then, we first initialize the manager routine by fetching the size of the communication channel and the number of tasks. We also initialize a counter to see how many tasks have been sent so far and start by distributing as many tasks as initially possible

```python
def manager(comm, tasks):
    size = comm.Get_size()
    num_tasks = len(tasks)
    send_index = 0
    for i in range(1, min(size,num_tasks)):
        comm.send(tasks[send_index], dest=i, tag=TAG_TASK)
        send_index += 1
```

Then, we need to initialize a counter for the tasks that we already received. As long as there are still tasks available, we need to wait for any incoming message (result) and further distribute tasks.

```python
    receive_index = 0
    TasksDoneByWorker = [[] for _ in range(size)]
    while send_index < num_tasks:
        status = MPI.Status()
        data = comm.recv(source=MPI.ANY_SOURCE, tag=TAG_TASK_DONE, status=status)
        TasksDoneByWorker[status.Get_source()].append(data)
        comm.send(tasks[send_index], dest=status.Get_source(), tag=TAG_TASK)
        send_index += 1
        receive_index +=1
```

Having handed out all tasks, we need to wait to receive the result of the remaining (sending counter has to equal received counter), and finally signal all workers to end their process. Note that all results are saved into a list of lists, where the index of the former corresponds to the rank of the worker and the content of the to the tasks that he solved.

```
    while(receive_index<send_index):
        status = MPI.Status()
        data = comm.recv(source=MPI.ANY_SOURCE, tag=TAG_TASK_DONE, status=status)
        TasksDoneByWorker[status.Get_source()].append(data)
        receive_index +=1
    for i in range(1, size):
        comm.send(None, dest=i, tag=TAG_DONE)
    return TasksDoneByWorker
```

The worker only needs to wait for more tasks and solve them until a signal is receive to break the while loop and terminate. Once every task is solved, a message with the result is sent back to the manager.

```
def worker(comm):
    while True:
        status = MPI.Status()
        task = comm.recv(source=0, tag=MPI.ANY_TAG, status=status)
        if status.Get_tag() == TAG_DONE:
            break
        task.do_work()
        comm.send(task, dest=0, tag=TAG_TASK_DONE)
```

On the main body, we initialize MPI and the manager starts a time counter, fetches all needed tasks and starts to distribute them amongst workers. Once completed, all tasks are combined and the computed mandelbrot picture is drawed.

```
if __name__ == "__main__":
    # get COMMON WORLD communicator, size & rank
    comm    = MPI.COMM_WORLD
    size    = comm.Get_size()
    my_rank = comm.Get_rank()

    # report on MPI environment
    if my_rank == MANAGER:
        print(f"MPI initialized with {size:5d} processes")

    # read command line arguments
    nx, ny, ntasks = readcmdline(my_rank)

    if my_rank == MANAGER:
        x_min = -2.
        x_max  = +1.
        y_min  = -1.5
        y_max  = +1.5
        timespent = - time.perf_counter()

        M = mandelbrot(x_min, x_max, nx, y_min, y_max, ny, ntasks)
        tasks = M.get_tasks()

        TasksDoneByWorker = manager(comm, tasks)
        m = M.combine_tasks([item for sublist in TasksDoneByWorker for item in sublist])

        plt.imshow(m.T, cmap="gray", extent=[x_min, x_max, y_min, y_max])
        plt.savefig("mandelbrot.png")

        timespent += time.perf_counter()
        print(f"Run took {timespent:f} seconds")
        for i in range(size):
            if i == MANAGER:
                continue
            print(f"Process {i:5d} has done {len(TasksDoneByWorker[i]):10d} tasks")
        print("Done.")
    else:
        worker(comm)
```

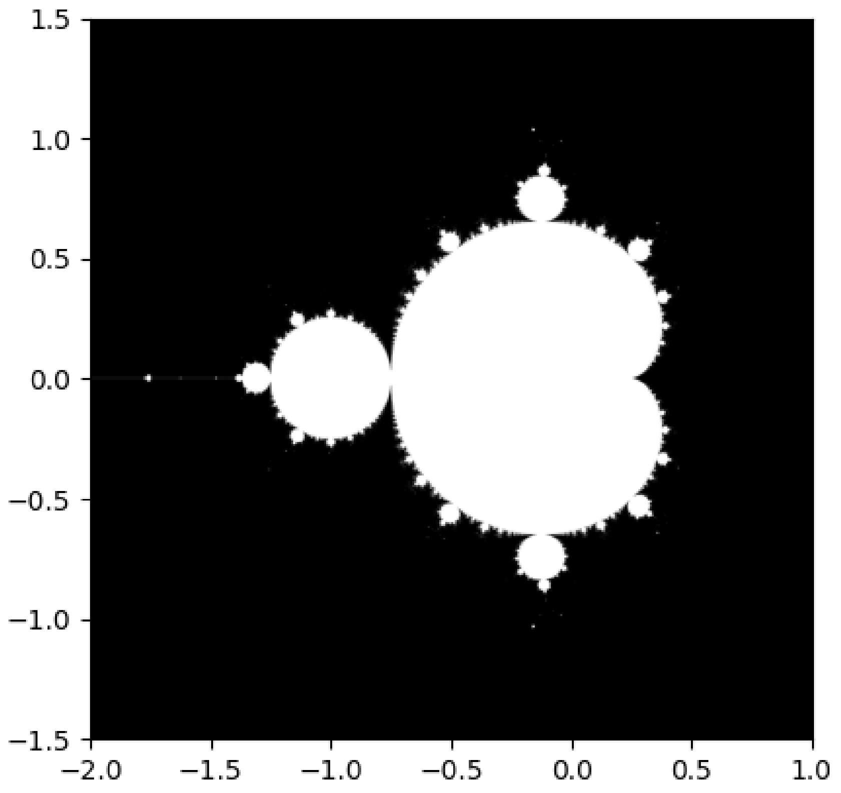

Figure 6: manage-worker mandelbrot result