ETH zürich

High-Performance Computing Lab for CSE **2023**

Student: Simon Aaron Menzi Discussed with: Lukas Bühler

Solution for Project 6

Due date: 15 May 2023 (11:59pm)

HPC Lab for CSE 2023 — Submission Instructions
(Please, notice that following instructions are mandatory: submissions that don't comply with, won't be considered)

- Assignments must be submitted to Moodle (i.e. in electronic format).
- Summarize your results and the observations for all exercises by writing an extended Latex report. Use the Latex template from the webpage and upload the Latex summary as a PDF.
- Provide both executable package and sources (e.g. Python, C/C++ files, Matlab, Julia). If you are using libraries, please add them in the file. Sources must be organized in directories called:
  Project_number_lastname_firstname
  and the file must be called:
  project_number_lastname_firstname.zip
  project_number_lastname_firstname.pdf
- The TAs will grade your project by reviewing your project write-up, and looking at the implementation you attempted, and benchmarking your code's performance.
- You are allowed to discuss all questions with anyone you like; however: (i) your submission must list anyone you discussed problems with and (ii) you must write up your submission independently.

1. Parallel Space Solution of a nonlinear PDE using MPI [in total 40 points]

1.1. Initialize and finalize MPI [5 Points]

The current rank and the total amount of ranks, called size, can be grabbed by the functions `MPI_Comm_rank` and `MPI_Comm_size` respectively.

One needs to free the communicators used in the program with the `MPI_Comm_free` function.

And never forget to initialize and finalize MPI in the `main.cpp`-file.

1.2. Create a Cartesian topology [5 Points]

Cartesian Coordinate based communication topologies can be created by using predefined MPI functions. This can come in handy to not be overwhelmed by complicated communication patterns, especially in higher dimensions. The communicator can be created using `MPI_Comm_create`. There are several helper functions to split up the size of the communicator, get neighboring ranks and so on, with functions from the MPI-Library. The following code shows how to get the neighboring ranks.

```
int rank_north = mpi_rank;
int rank_south = mpi_rank;
int rank_east  = mpi_rank;
```

```
int rank_west  = mpi_rank;

MPI_Cart_shift(comm_cart, 0, 1, &rank_north, &neighbour_north);
MPI_Cart_shift(comm_cart, 0, -1, &rank_south, &neighbour_south);
MPI_Cart_shift(comm_cart, 1, 1, &rank_east, &neighbour_east);
MPI_Cart_shift(comm_cart, 1, -1, &rank_west, &neighbour_west);
```

1.3. Extend the linear algebra functions [5 Points]

Here, one needs to implement the dot product and the norm.

For both, the rank-specific `result` doubles needed to be reduced to the `result_global` doubles by summing them all up.

The code for the dot-product, in the end, was:

```
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;
}
```

And for the norm computation:

```
double hpc_norm2(Field const& x) {
    double result        = 0;
    double result_global = 0;
    int N                = x.length();

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

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

    return sqrt(result_global);
}
```

One could also use the `hpc_dot(...)` function to compute the norm by passing the same Field twice and take the square root of it.

1.4. Exchange ghost cells [10 Points]

Using non-blocking communication for ghost cells exchange was the task here. This can be achived using by using the non-blocking MPI functions with an `I` in front of it, like `Isend` or `Irecv` and wait until the message was recieved.

For each direction, north, east, south, and west, one needs to have buffers at hand to store and send the messages.

As the application has non-periodic boundary conditioins, one must check that one only communicates with its real neighbor.

```
PI_Request reqs[8];
int num_reqs = 0;

if (domain.neighbour_north >= 0) {
    // fill the buffer with the values from the north boundary
    for (int i = 0; i < nx; i++) buffN[i] = s(i, jend);

    MPI_Isend(buffN.data(), nx, MPI_DOUBLE, domain.neighbour_north, 0,
                domain.comm_cart, &reqs[num_reqs++]);
    MPI_Irecv(bndN.data(), nx, MPI_DOUBLE, domain.neighbour_north, 0,
                domain.comm_cart, &reqs[num_reqs++]);
}


if (domain.neighbour_south >= 0) {
    // fill the buffer with the values from the south boundary
    for (int i = 0; i < nx; i++) buffS[i] = s(i, 0);

    MPI_Isend(buffS.data(), nx, MPI_DOUBLE, domain.neighbour_south, 0,
                domain.comm_cart, &reqs[num_reqs++]);
    MPI_Irecv(bndS.data(), nx, MPI_DOUBLE, domain.neighbour_south, 0,
                domain.comm_cart, &reqs[num_reqs++]);
}

if (domain.neighbour_east >= 0) {
    // fill the buffer with the values from the east boundary
```

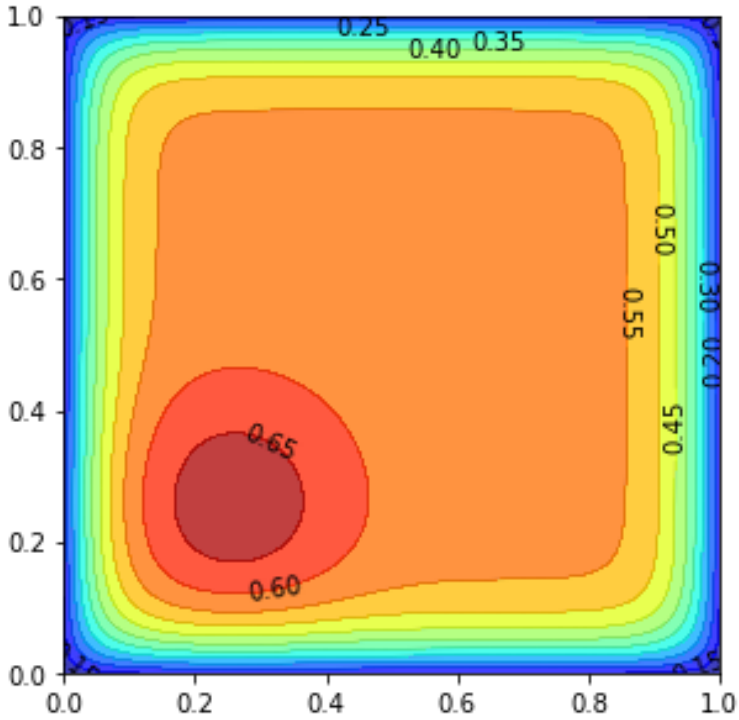
```
    for (int j = 0; j < ny; j++) buffE[j] = s(iend, j);

    MPI_Isend(buffE.data(), ny, MPI_DOUBLE, domain.neighbour_east, 0,
                domain.comm_cart, &reqs[num_reqs++]);
    MPI_Irecv(bndE.data(), ny, MPI_DOUBLE, domain.neighbour_east, 0,
                domain.comm_cart, &reqs[num_reqs++]);
}

if (domain.neighbour_west >= 0) {
    // fill the buffer with the values from the west boundary
    for (int j = 0; j < ny; j++) buffW[j] = s(0, j);

    MPI_Isend(buffW.data(), ny, MPI_DOUBLE, domain.neighbour_west, 0,
                domain.comm_cart, &reqs[num_reqs++]);
    MPI_Irecv(bndW.data(), ny, MPI_DOUBLE, domain.neighbour_west, 0,
                domain.comm_cart, &reqs[num_reqs++]);
}
```

In the meantime, the diffusion for the inner grid-points is computed while the processes communicate with each other. When the application needs the neighboring ghost cells, the application will wait until it received the information needed.

This is achieved with the following MPI function:

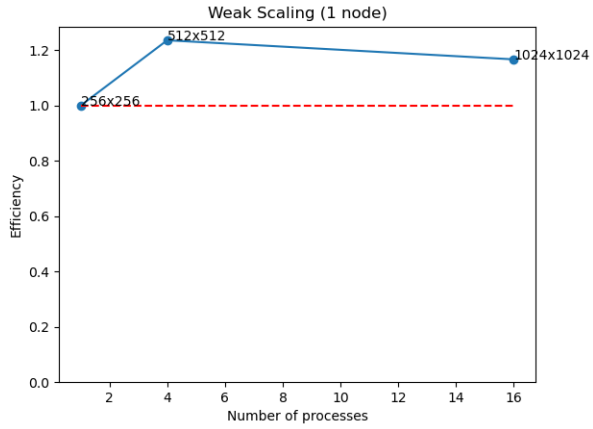
```
MPI_Waitall(num_reqs, reqs, MPI_STATUSES_IGNORE);
```

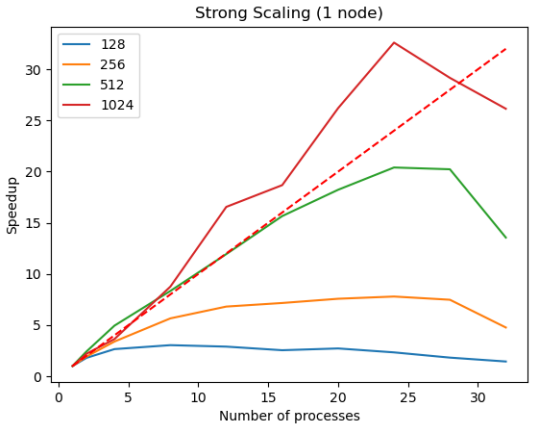
The result is very similar to the result in project 3.

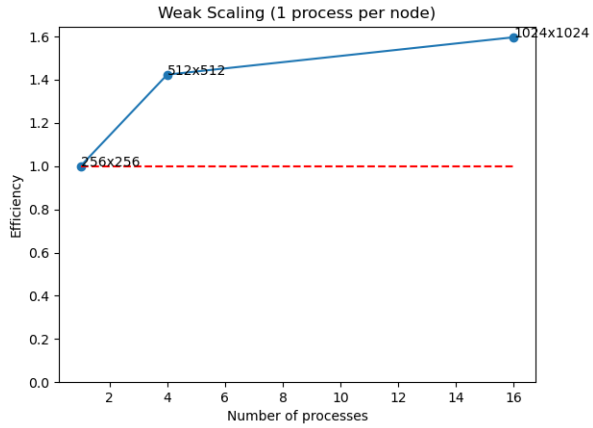


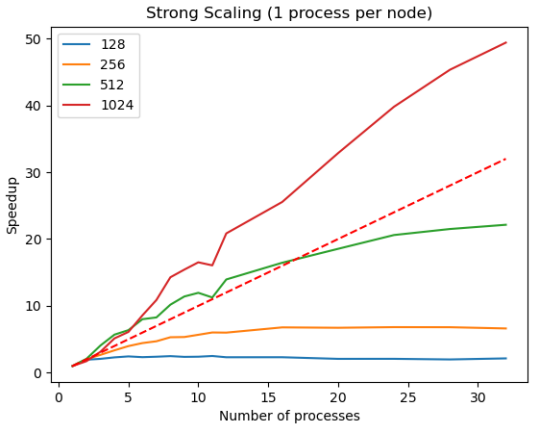
1.5. Scaling experiments [15 Points]

One can see that the problem for smaller problem sizes, the problem is not scaling well, but as soon as the problem size is increased, the speedup is increasing to superlinear scaling. This is due to better cache usage as the problem partitions can fit into the cache and less cache misses occur. After 24 processes, a drop in performance occurs as a result of an increasing communication overhead. This cache usage behaviour can be seen in the strong and in the weak scaling as well.









2. Python for High-Performance Computing (HPC) [in total 60 points]

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

There is a difference between all-lower-case methods and First-capital-case methods in mpi4py. The all-lower-case methods are more pythonic, meaning they are more user-friendly in there arguments and therefore less complex. This is achieved by using pythons data-structures. The First-capital-case methods are in contrast more C-like, but are in exchange faster and more powerful. They use allocated buffers for data exchanges.

2.1.1. Pickle-based communication

This part is done with the all-lower-case MPI methods of mpi4py.

```
from mpi4py import MPI
import pickle

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

```python
# Create a list of all the ranks
ranks = list(range(size))

# Pickle the list and broadcast it to all processes
pickled_ranks = pickle.dumps(ranks)
all_ranks = comm.bcast(pickled_ranks, root=0)

# Compute the sum of all the ranks
sum = comm.allreduce(rank, op=MPI.SUM)

# Print the sum for each process
print(f"Rank {rank} has sum {sum}")
```

2.1.2. Near C-speed approach

In this implementation, we use the near C-speed approach of First-capital-case methods.

```python
comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

# Create an array of all the ranks
ranks = np.arange(size)

# Scatter the array to all processes
my_rank = np.zeros(1, dtype=int)
comm.Scatter(ranks, my_rank, root=0)

# Compute the sum of all the ranks
buf = np.zeros(1, dtype=int)
buf[0] = my_rank
comm.Allreduce(MPI.IN_PLACE, buf, op=MPI.SUM)

# Print the sum for each process without the brackets
print("Rank {} has sum {}".format(rank, buf[0]))
```

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

As in the C++ implementation, the python implementation follows the same principles with the same (or similar) naming scheme of the MPI functions.

```python
comm = MPI.COMM_WORLD
size = comm.Get_size()

# Compute the dimensions of the Cartesian topology
dims = MPI.Compute_dims(size, 2)

# Create the Cartesian topology
periods = (True, True)
cart_comm = comm.Create_cart(dims, periods=periods)

# Get the rank, coordinates, and neighboring processes of the current process
rank = cart_comm.Get_rank()
coords = cart_comm.Get_coords(rank)
```

```
north, south = cart_comm.Shift(0, 1)
west, east = cart_comm.Shift(1, 1)
```

2.3. Exchange rank with neighbours [5 Points]

The rank exchange is done with simple, pythonic send and recv functions.

```
cart_comm.send(rank, dest=north, tag=0)
rank_south = cart_comm.recv(source=south, tag=0)

cart_comm.send(rank, dest=south, tag=1)
rank_north = cart_comm.recv(source=north, tag=1)

cart_comm.send(rank, dest=west, tag=2)
rank_east = cart_comm.recv(source=east, tag=2)

cart_comm.send(rank, dest=east, tag=3)
rank_west = cart_comm.recv(source=west, tag=3)

# Output the exchanged data for the current process
. . .
```

2.4. Change linear algebra functions [5 Points]

The "C-style" for loop needs more lines of code and is probably (not tested) also slower as the np approach, as np is highly optimized library in contrast to simple for loops.

```
def hpc_dot(x, y):
    """Computes the inner product of x and y"""

    # For loop implementation
    #prod = 0.0
    #for i in range(0, x.domain.nloc):
    #    prod += x.inner[i]*y.inner[i]
    #result = np.zeros(1)
    #comm = x.domain.comm
    #comm.Allreduce(prod, result, op=MPI.SUM)
    #return result

    prod = np.multiply(x.inner, y.inner).sum()
    result = np.zeros(1)
    x.domain.comm.Allreduce(np.array(prod), result, op=MPI.SUM)
    return result[0]


def hpc_norm2(x):
    """Computes the 2-norm of x"""

    return np.sqrt(hpc_dot(x, x))
```

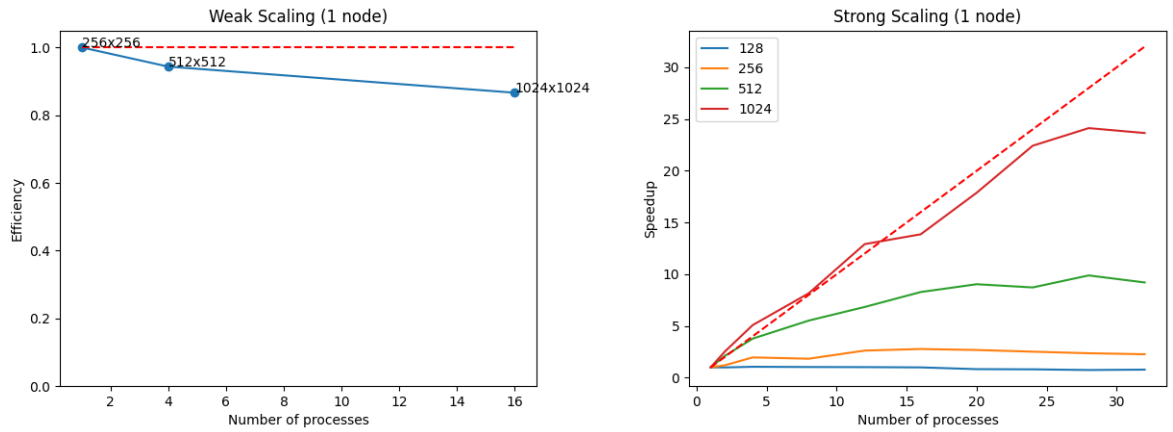
This time, the approach of just using the dot product in combination with the square root was used to compute the norm.

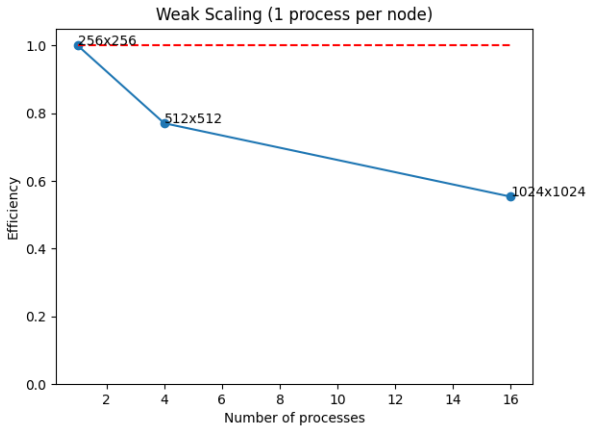
2.5. Exchange ghost cells [5 Points]

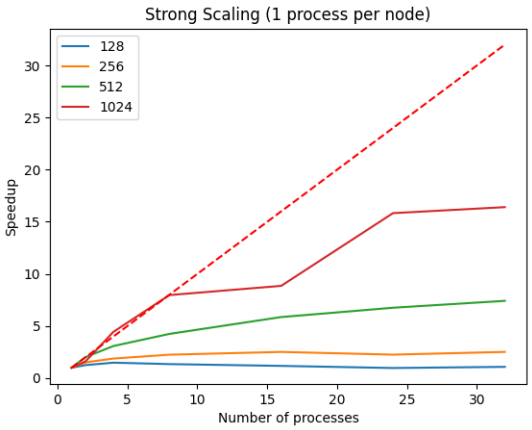
The exchange of ghost cells in this implementaiton is done by using a function to send and receive the information with not blocking functions `Isend` and `Irecv`. When the data is needed, the function `exchange`$_w aitallisusedtowaitforthemessagestobereceived.Inbothofthesefunctions, oneneedstobeawareofthe$

```
def exchange_startall(self):
        """Start exchanging boundary field data"""
        domain = self._domain # copy for convenience
        comm = domain._comm   # copy for convenience
        # ... implement ...
        last_i = domain.local_nx -1
        last_j = domain.local_ny -1

        # send and receive data
        if domain.neighbour_north >= 0:
            self._buffN = np.copy(self._inner[:, last_j])
            self.send_north = comm.Isend(self._buffN, dest=domain._neigh_north)
            self.recv_north = comm.Irecv(self.bdryN, source=domain._neigh_north)
        if domain.neighbour_south >= 0:
            .
            .
            .

def exchange_waitall(self):
        """Wait until exchanging boundary field data is complete"""
        domain = self._domain # copy for convenience
        # ... implement ...
        # wait for data to be sent and received
        if domain.neighbour_north >= 0:
            self.send_north.Wait()
            self.recv_north.Wait()
        if domain.neighbour_south >= 0:
            .
            .
            .
```

2.6. Scaling experiments [5 Points]







In this scaling experiment, we see that the python implementation of this problem scales worse than the C++ implementation. This is especially apparent when using a small problemsize. The superlinear scaling is also gone due to not being able to use cache as efficiently.

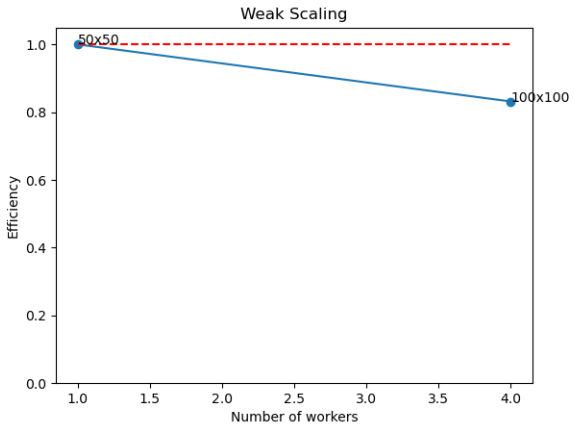
2.7. A self-scheduling example: Parallel Mandelbrot [30 Points]

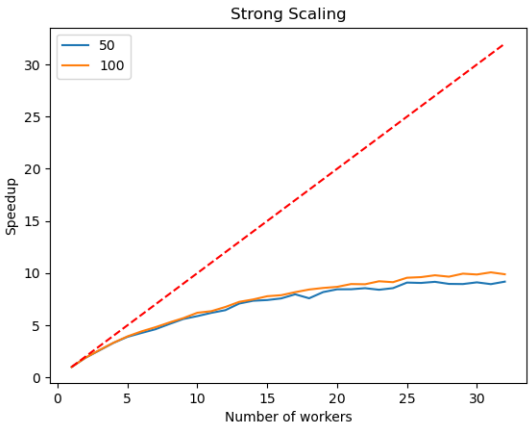
In this task, one had to implement the manager worker algorithm to compute the mandelbrot set. This is done by assigning one rank the manager role, which assigns tasks and the other ranks will complete tasks in the role of a worker.

The following code implements this and is commented such that it should be possible to understand how this behaviour is accomplished.

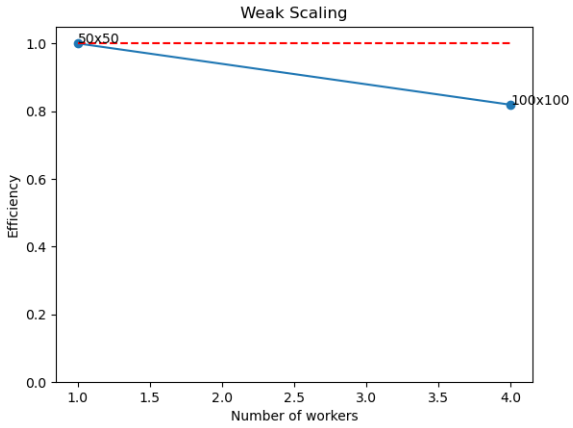
```
def manager(comm, tasks):

size = comm.Get_size()
requests = np.zeros(size - 1, dtype=MPI.Request)
ids = np.zeros(size - 1, dtype=int)
upcommingTask = 0

for reqs in range(1, size):
    comm.send(tasks[upcommingTask], dest=reqs, tag=TAG_TASK)

    buffer = bytearray(10**7)
    requests[reqs - 1] = comm.irecv(buffer, source=reqs, tag=TAG_TASK_DONE)


    ids[reqs - 1] = upcommingTask
    upcommingTask += 1

while len(tasks) != upcommingTask:
    req = MPI.Request.waitany(requests)
    rank = req[0] + 1
    task = req[1]

    # Save completed work
    tasks[ids[rank - 1]] = task
    TasksDoneByWorker[rank] += 1


    # Send new task
```

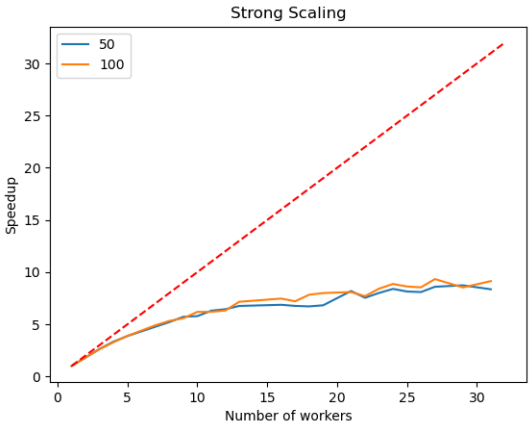
```
        comm.send(tasks[upcommingTask], dest=rank, tag=TAG_TASK)
        buf = bytearray(1 << 20) # allocate enough space
        requests[rank - 1] = comm.irecv(buf, source=rank, tag=TAG_TASK_DONE) # Start receiving
        ids[rank - 1] = upcommingTask # Save id of task
        upcommingTask += 1

    req = MPI.Request.waitall(requests)

    # Save remaining results into tasks
    for i in range(len(req)):
        task = req[i]
        if task != None:
            tasks[ids[i]] = task
            TasksDoneByWorker[rank] += 1

    # Send done message
    message = np.array([TAG_DONE])
    req = comm.Ibcast(message, root=0)
    req.Wait()

    def worker(comm):

    message = np.array([TAG_DONE])
    isDone = comm.Ibcast(message, root=0)

    # Do tasks until done message is received
    while True:
        request = comm.irecv(source=MANAGER, tag=TAG_TASK)

        tmp = MPI.Request.waitany(requests=np.array([request, isDone]))
        # Check if done
        if isDone.Test():
            request.Cancel()
            break

        # Do task
        task = tmp[1]
        task.do_work()
        # Send result
        comm.send(task, dest=MANAGER, tag=TAG_TASK_DONE)
```





As expected and also said in the task description, this is not a well scaling problem for the manager worker algorithm. This is clearly visible. It also does not scale with the problem size.





It also does not make a difference when using multiple nodes and distribute each task to these nodes.