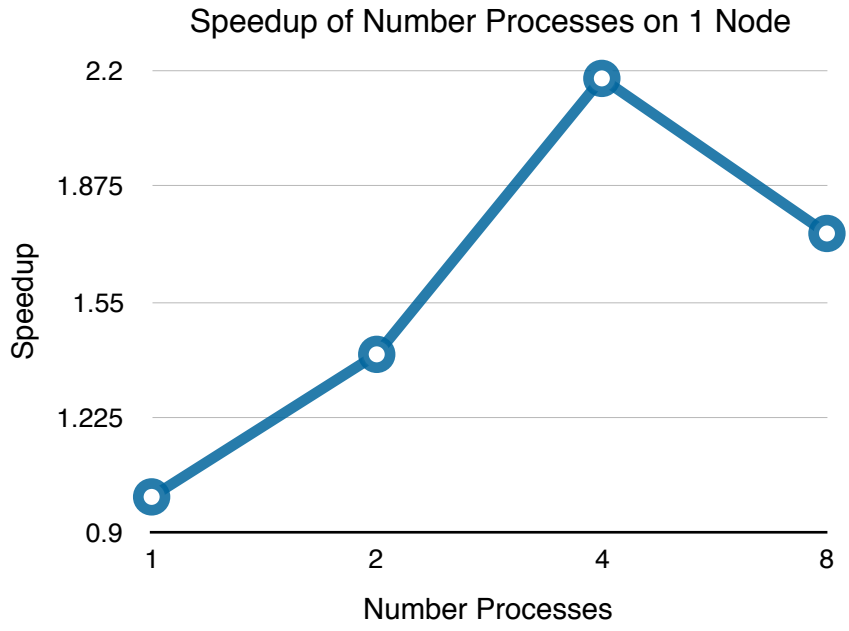
Assignment 2.1 and 2,3: Wave equation (MPI) & Collective Communication
Steven Raaijmakers & Marcus van Bergen
10824242 & 10871993

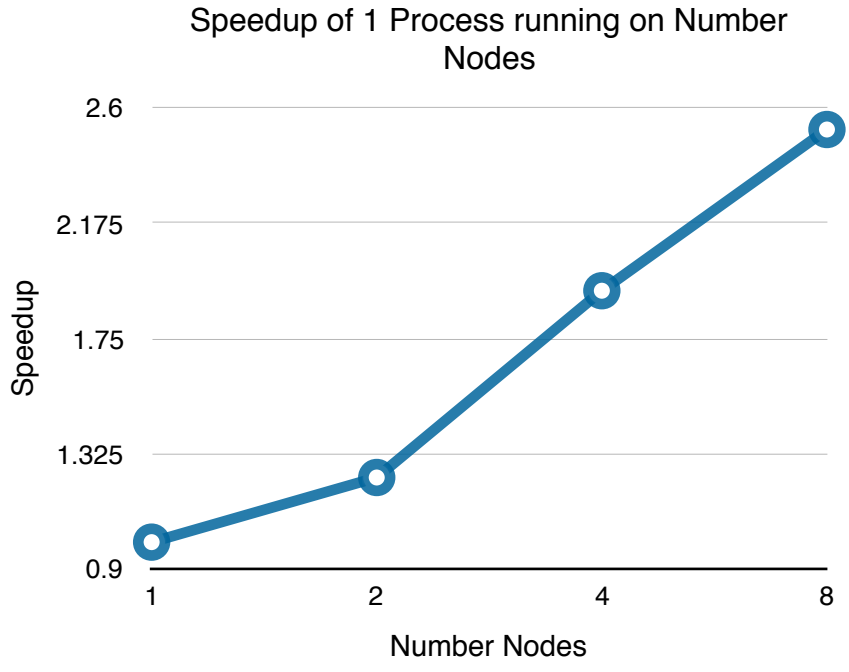
Assignment 2.1: Wave equation with MPI

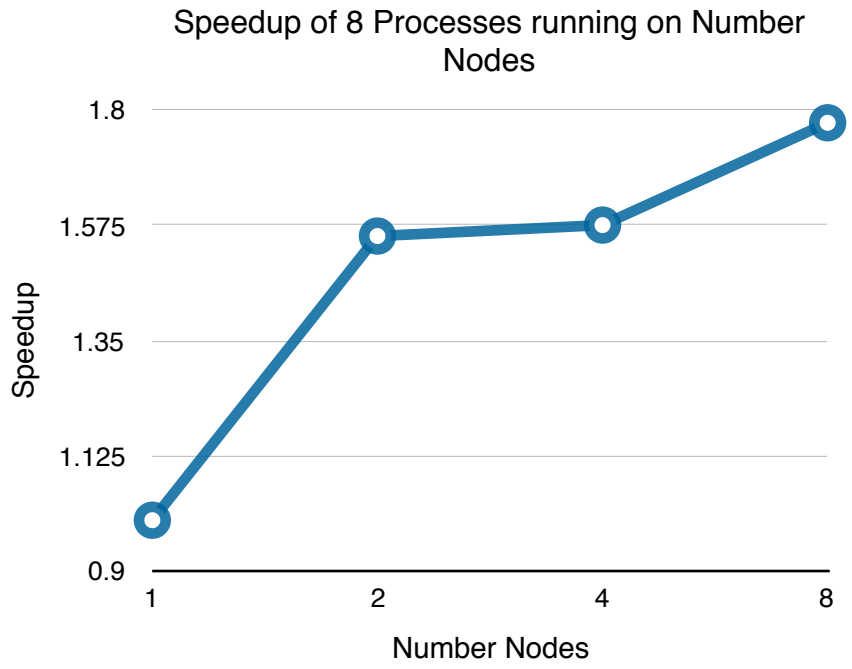
We simulate the wave with the parameters: 10,000 as i_max and 1,000,000 as t_max respectively. Running the simulation with a i_max greater than 10,000 such as 100,000 or higher would cause the program to perform terribly, unless it was being run on 8 nodes with 8 processes in which case it large i-max args could finish in sub 3 seconds. However anything less resulted in very long queue times.

Thus for our testing we kept the run-args the same while changing the amount of nodes and processes; dependent on which test we were working on.

The following results were collected:







From the charts we can derive some interesting results. The most obvious observation we come across is that adding either any n-processes or n-nodes for the simulation leads to a positive speedup. We also notice that most speedups are relatively. With the average speedup being within the range of 1-2.

We suspect this behaviour is directly related with such a low i-max. In such a case the processes calculate very fast but run into extra overhead when having to wait on the swapping and syncronization to happen. The result of this is that the speedup per process stays relatively low.

Results of simulation ran on DAS4

Number of Processes	1	2	4	8
1 Node	21s 623ms	15s 431ms	9s 927ms	12s 415ms
2 Nodes	17s 466ms	13s 957ms	8s 55ms	7s 990ms
4 Nodes	11s 229ms	9s 138ms	7s 317ms	7s 881ms
8 Nodes	8s 583ms	7s 336ms	6s 430ms	6s 996ms

Support for this conclusion is only greatened by the above chart, our result times.

Running 1 processes on n-nodes, gives the most speedup. This coincides with the above conclusions as the lower the n-processes the lower the sync-overhead, while the n-nodes only contribute with positive speedup.

This provides the most optimal scenario; where process overhead is at its least, the speedup of extra nodes is being applied, thus providing us with the highest speedup in general.

This same behaviour is also clear in the last row of the table. Spawning additional processes on 8-nodes does not lead to much extra speedup as the processes's own speedup is diminished by its own overhead.

Which unsurprisingly equates to the highest total speedup being ~3.1 (1 Processes 1 Node *v.s.* 8 Processes 8 Nodes).

To summarise, we conclude that the most affecting factor in this test's* speedup is the number of nodes, and not the number of processes. This goes without saying that processes do indeed give *some** speedup. However the bottleneck processes face is that with our running args is that there seems to be some overhead with an increased amount of processes. This is probably due to the low calculation times and long waits (for t-max) that the processes encounter.

**in our case with a low i-max args*

Assignment 2.3: Collective communication

A) In this example we choose to let the master use a non blocking send, which sends all its processes the correct slice of the array they need to work on. After this it uses a blocking recv because it want to wait for all its processes to have returned the address of their modified version of the array.

In the process we already send the correct addresses back to master, but we need to wait for old and cur to be received, before we can calculate anything so we use a blocking recv. After receiving we calculate the values for next. After this we assume we received to halo cells already (which where non-blocking) and we calculate the correct value.

```
# master process:
isend(old_slice)
isend(current_slice)

for each process that terminated:
    array[proc * size] = recv(proc)
    count++
count == num_proc:
    exit()

# working process:
allocate(next)

# communication
isend_master(next_address)

recv(old)
recv(cur)

# communication
irecv(left_halo)
irecv(right_halo)

# computation
for i = 1 to i = size - 1:
    next = computation(old, curr) # fill next array

next[0] = computation(left_halo)
next[size] = computation(right_halo)
exit()
```

B) Asynchronous operations are a bit more different. In this case a process does not have to wait on the master to be able to further calculate. For this wave simulation it would be possible to make an asynchronous implementation using halo cells and communication with the neighbouring processes.

In the case of our wave simulation a processes who has to calculate from a i-min to an i-max only has to have it's neighbours information regarding their current-array. This is because the function requires the process to have i-1 and i-max+1 of current array. This means the process has to get these value from its neighbours. Meaning the process has to wait on the neighbour to finish calculating and send their information, and can then continue. After which is sends it's information to it's neighbours so they can also continue. This way the master process does not have to block/ wait on any other process and are processes now only dependent on their neighbours information.

```
void calculate_slice_of_array(old, curr, nextarray){
       .......

       //wait for vars from neighbour nodes
       i-1     = MPI_Iwait(//previous neighbour neighbour nodes)
       i_max+1 = MPI_Iwait(//future neighbour neighbour nodes)

       current = calculatewave()

       //send your array to other nodes
       MPI_Isend(current[i-max +1] to future neighbour)
       MPI_Isend(current[i-1] to previous neighbour)

       .......
}
```

Assignment 2.3: Collective communication

Collective communication is a way that processes can share data with each other. In this case our code have one root node, which is permanent, and send its information to the other processes. This way a process waits for information while the root sends it. This is one form of collective communication.

It's also possible to have the root be constantly changing, in which case when a process receives information it takes it's own rank and send the message to it's neighbour. The best way to approach this is to start by requesting my_rank. We then take the next neighbour, so say my_rank +1 and send the message to him. This keeps happening till the highest rank is reached; which is an edge case.

If the my_rank is the size of num_tasks we know that the process is at the highest rank, so it's neighbour will be the starting rank (1). And rank 1 will get the message and continue the process, until the original rank gets the message, in which case the operation is finished.