A04 - Assignment 4

Student: **Deidier Simone - 133020**

Answers to the theory questions

1. **MPI_Gather** collects data from all processes and sends it to a single root process; it is used when only one process (*typically the root process*) needs to have all the data collected. Communication is unidirectional towards the root process and can become a bottleneck if the number of processes is large, as the root process must receive data from all other processes. However, only the root process must have sufficient memory to hold all the collected data. **MPI_Allgather** collects data from all processes and distributes a complete copy of the collected data to all processes; it is used when all processes need to have a complete copy of the collected data. Communication is bidirectional, as each process sends and receives data, and may be more efficient than MPI_Gather in some cases, as communication is distributed among all processes. But each process must have sufficient memory to hold a complete copy of the collected data. If the volume of data is large, MPI_Allgather may result in memory-intensive usage, as each process must store a complete copy of the data, whereas MPI_Gather may be more memory-efficient, as only the root process stores all the data. With a large number of processes, MPI_Gather can become a bottleneck due to the concentration of communication on the root process, whereas MPI_Allgather distributes communication, potentially improving scalability. In practice, if all processes need the data collected, MPI_Allgather is more appropriate; if only the root process needs the data, MPI_Gather is more appropriate.

2. To use a **9-point stencil** for the approximation in the current time step, it is necessary to communicate values from neighbouring processes, including diagonal processes. With MPI, we can obtain the rank and size of the communicator, identify neighbouring processes, including diagonal processes (*very favourable if using a Cartesian grid for processes*), use **MPI_Sendrecv** (*or* ***MPI_Isend*** *and* ***MPI_Irecv***) to exchange data with neighbouring processes (*repeated for all necessary neighbours: north, south, east, west, and diagonal*) and then update the buffers with the data received from the neighbouring processes. This approach ensures that each process has the necessary data from its neighbours to use a 9-point stencil.

3. Running the simulation with a domain size of **2048x512** quadruples the size of the domain compared to the basic version (**512x512**), which certainly results in a slowdown and a higher code execution time. As can be seen from the screenshot, running the code with **4 processes** takes much less time than with **16 processes**. This is because, by using a two-dimensional Cartesian grid to divide the work between the various processes, running with 16 processes only increases the overhead given by the cost

of communication and interaction between the various MPI processes, whereas with 4, each process has more work to do, but communication has a much smaller impact.

```
[simonedeidier@dhcp-10-24-147-88 assignment 4 % mpiexec -n 4 --oversubscribe ./parallel -m 2048 -n 512
Mean simulation time: 3.237086 seconds
[simonedeidier@dhcp-10-24-147-88 assignment 4 % mpiexec -n 16 --oversubscribe ./parallel -m 2048 -n 512
Mean simulation time: 9.826690 seconds
simonedeidier@dhcp-10-24-147-88 assignment 4 %
```

Figure 1: *Time taken to run the simulation on a domain of 2048x512 points with 4 and 16 processes.*

4. **Weak scaling** measures how the execution time of a programme varies as the number of processors increases, while keeping the workload per processor constant; in fact, the goal is to keep the execution time constant while increasing the number of processors and the total workload. Ideally, if you double the number of processors, you also double the total workload, so that each processor has the same workload. **Strong scaling** measures how the execution time of a program varies as the number of processors increases, while keeping the total workload constant. In fact, the goal is to reduce the total execution time while increasing the number of processors, while keeping the total workload constant, so if you double the number of processors, the total workload remains the same, but is divided among more processors.