A05 - Assignment 5

Student: **Deidier Simone - 133020**

Answers to the theory questions

1. When using **Pthreads** to parallelise calculation, each thread can be assigned to a specific portion of the data grid. However, unlike other parallelisation techniques such as **MPI (Message Passing Interface)**, where processes can run on different nodes and require the exchange of data across the grid, threads in **Pthreads** share the *same memory address space*. This means that all threads can directly access *shared data* without the need to explicitly exchange edges with each other.

2. **OpenMP** and **MPI** are two technologies used for programme parallelisation, but they differ significantly in their approach and use cases. **OpenMP** uses a *shared-memory programming model*, meaning all threads created by OpenMP share the *same memory space*. It is generally used to parallelise programmes on systems with *shared memory*, such as multi-core computers or servers with multiple processors. OpenMP is known for its *ease of use*, as developers can parallelise existing code by adding compilation directives (pragma) to the source code. Since threads share the same memory, there is *no need to exchange data* between them, reducing communication overhead. On the other hand, **MPI** uses a *distributed memory programming model*, where each MPI process has its *own private, non-shared memory space*. MPI is ideal for parallelising programmes on systems with *distributed memory*, such as computer clusters or networks. It is also useful when processes must communicate via a network. However, **MPI** requires *explicit handling of communication* between processes, as developers need to write code to send and receive messages. This makes MPI more complex than OpenMP, and exchanging data through messages introduces *significant communication overhead*, especially on slower networks.

3. **Pthreads** and **OpenMP** implementations differ mainly in how they handle *parallelisation* and *thread synchronisation*. **Pthreads** uses an *explicit thread programming model*, requiring manual creation, management, and synchronisation of threads using primitives such as *barriers*. While this approach offers *fine control* over threads, it increases *programming overhead* and can make the code more difficult to maintain. **OpenMP**, in contrast, uses a *directive-based programming model*. Developers can parallelise code by adding compilation (pragma) directives, and OpenMP *automatically handles thread synchronisation*. For example, the `#pragma omp parallel for` directive divides work between threads and handles synchronisation automatically. **OpenMP** is *easier to use* than **Pthreads**, as developers do not have to manually manage thread creation and synchronisation. This makes OpenMP code simpler, more readable, and reduces *programming overhead*.

4. Parallelising a **recursive problem** with **OpenMP** can be *complex*, as recursion involves nested function calls, which are difficult to handle in parallel. However, OpenMP provides tools to parallelise recursive problems. One common technique is to use the `#pragma omp task` directive to create *parallel tasks*. But creating too many tasks can introduce *significant overhead*, so it is essential to balance the number of tasks with the actual work. Additionally, the `#pragma omp taskwait` directive is necessary to *synchronise tasks* and ensure correct results. While recursive parallelisation can scale well on systems with many cores, it requires *careful handling* to avoid overhead.