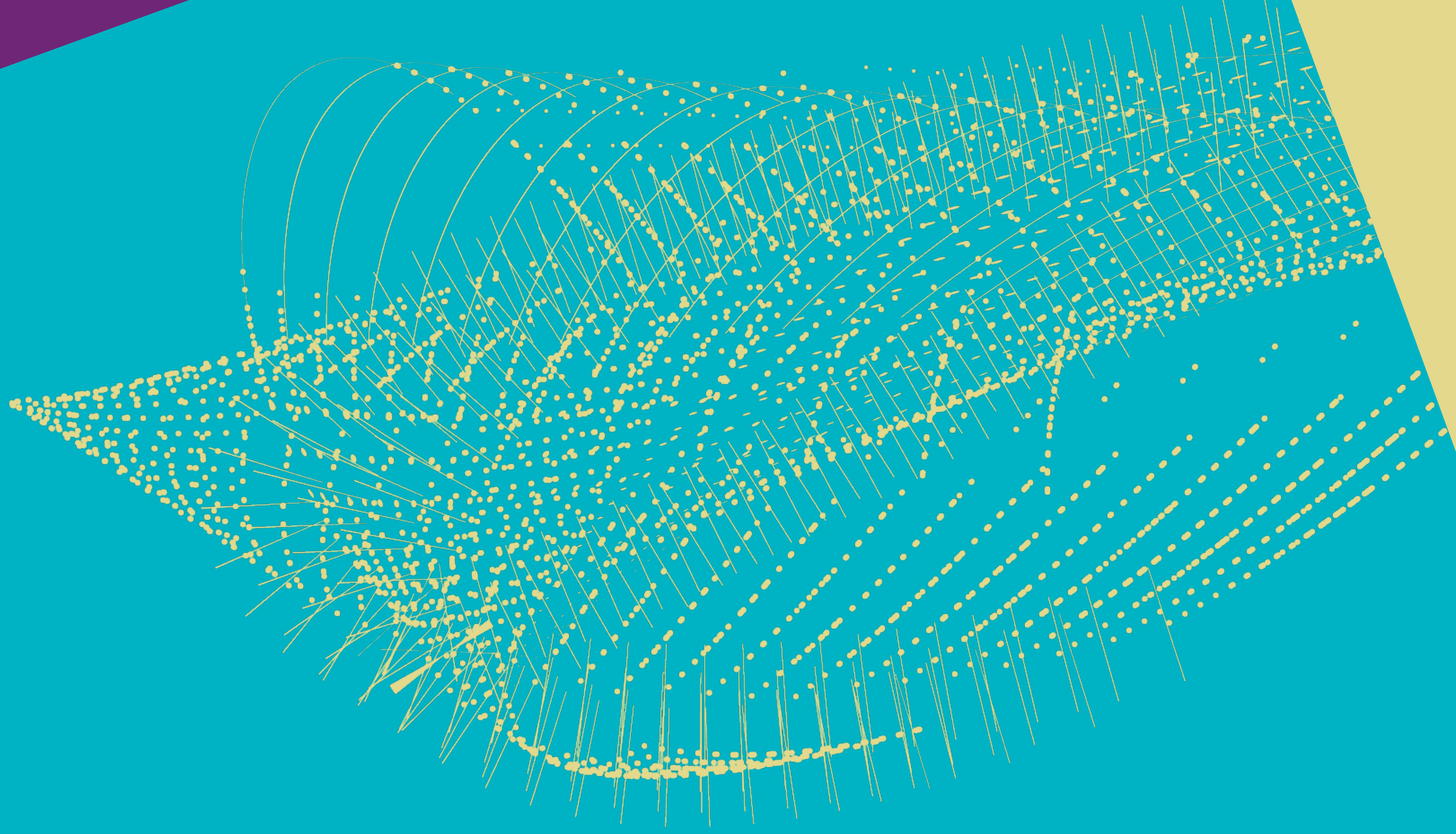


PARALLEL PROGRAMMING: BASIC CONCEPTS

Zef Wolffs

WHAT IS PARALLEL COMPUTING

- A type of computation in which multiple calculations or processes are executed **simultaneously**

- Can be realised in multiple ways
  - Task parallelism: tasks get distributed to **processes** or **threads**, which they execute on the same data
  - Data parallelism: each **process** or **thread** does the same work on its own data
  - Pipeline parallelism: extension of task parallelism where workload is split up in a **sequence of tasks**, which may be dependent on one another

- Processes may have a way of communicating with each other

THREADS

- A **thread** is simply a set of instructions to be executed by the computer
  - Example of a thread for boiling an egg:
    - Take pan -> Fill pan with water -> Put pan on stove -> …
- Processing units such as the **CPU** or the **GPU** can (only) execute threads
  - GPU’s are sometimes capable of running thousands of threads in parallel
  - A CPU can only execute a single thread at a time, however, technologies such as **hyperthreading** still can speed up multiple thread execution on a single CPU
- Threads share **heap (dynamic)** memory, but have their own **stack (static)**

PROCESSES

- A process is an instance of a computer program to be executed by the computer which may contain **multiple threads**, its own **stack** and **heap**, and any other required resources
  - All threads of a process share the heap memory allocated to the process
  - Example of a process could be “making breakfast”, of which boiling an egg could be a thread

- **Multiprocessing** is typically preferred over **multithreading** when more complex workloads (entire programs) are to be executed in parallel but does induce more overhead due to the need for **resource (heap memory) allocation**
  - Terms are often used interchangeably, and are very similar especially on unix

WHY PARALLEL COMPUTING

- This is a valid question, a good parallel program can be hard to set up properly
  - Especially difficult to debug

- Save walltime
  - e.g. for high frequency trading firms which benefit from making trading decisions faster than competitors
- Solve more complex problems
  - e.g. for complex physics models that need to be fit to large datasets
- Provide concurrency
  - e.g. for a webserver that needs to handle multiple website visitors simultaneously

IS IT WORTH IT TO GO PARALLEL FOR YOUR CODE?

- Since parallelising code can be hard, a valid question arises: Does the time won from parallelising the code outweigh the extra time spent in development?

- Problems where parallel computations (tasks) are not dependent on eachother are easy to parallelise, and often called **“embarrassingly parallel”**
  - Example: Generating 100 random numbers. Each random number can be generated independently, and the same program to generate 1 random number can in principle be run independently 100 times

- Most problems have a section that can be parallelised and a section that cannot be parallelised, the latter strongly limits potential speedup

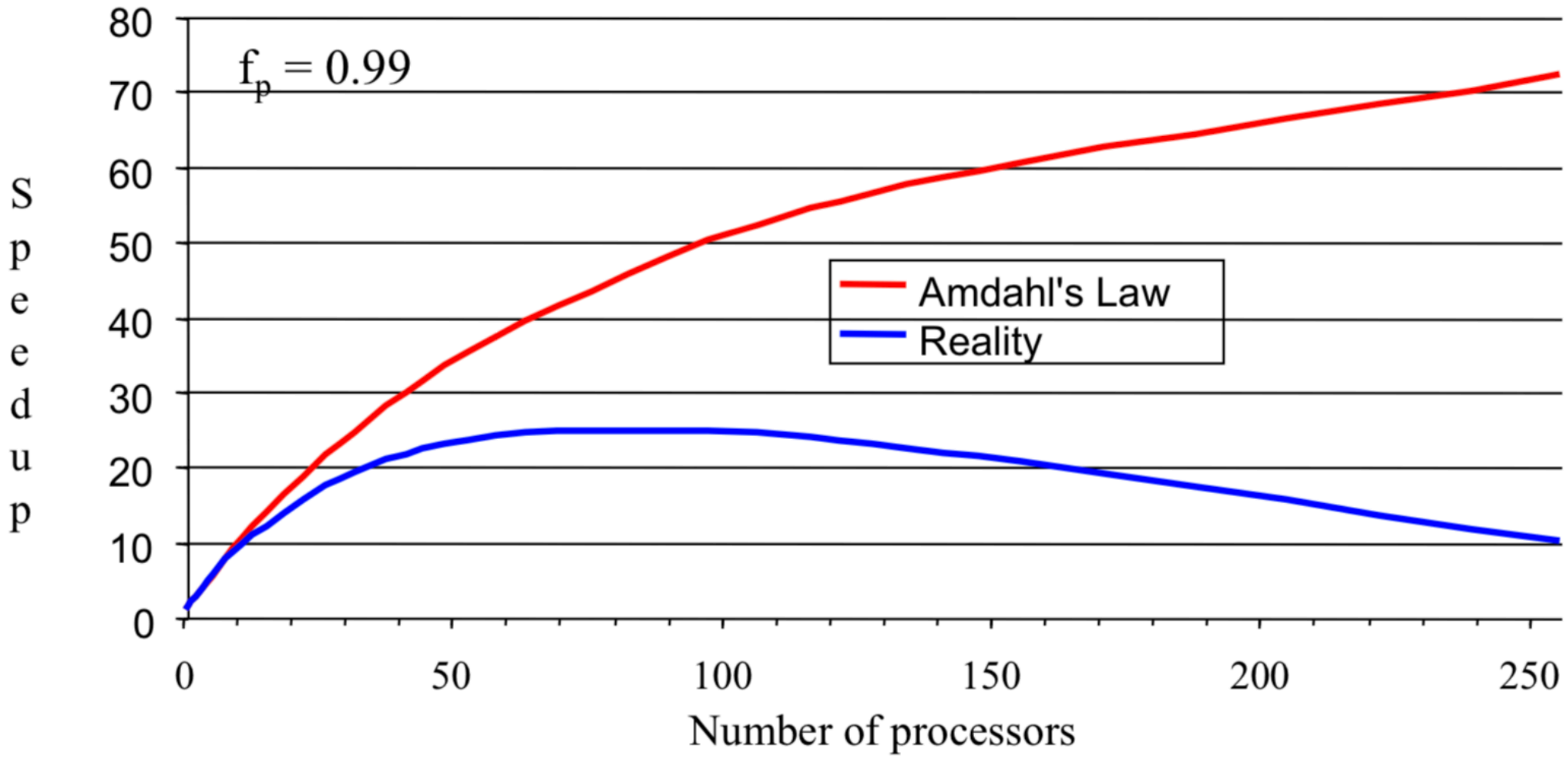
AMDAHL'S LAW

- Amdahl's law quantifies upper limit on parallel speedup: $S = 1/(f_s + f_p/N)$
  - S: speedup, i.e. time with n processors divided by time with one processor
  - f_p: parallel fraction of code
  - f_s: serial fraction of code
  - N: number of processors

- Limit cases
  - $f_s = 0,\ f_p = 1$ → S = N (generating N random numbers)
  - N = ∞ → S = 1/f_s; e.g. if 10% of code is serial, speedup strictly limited to 10

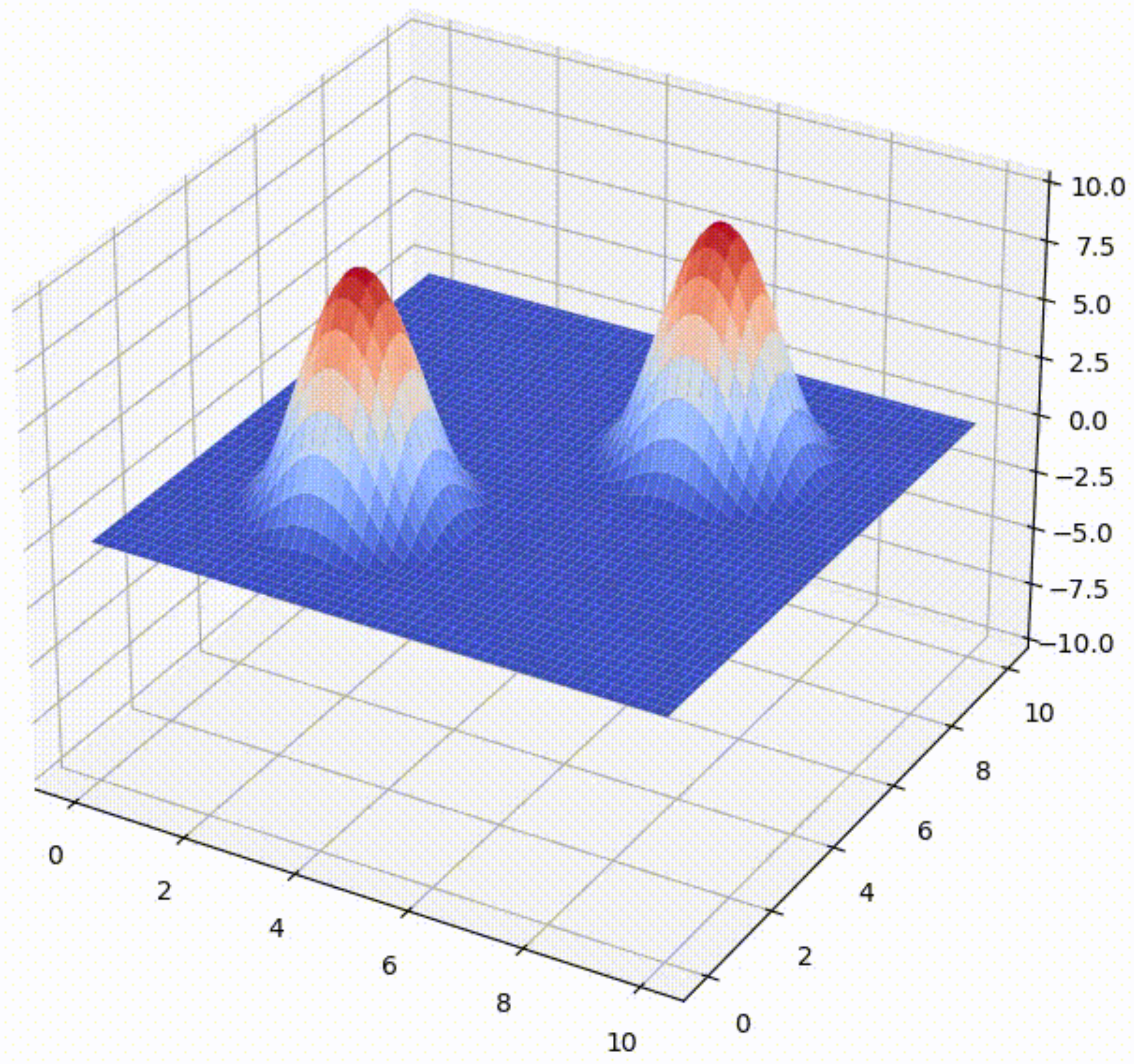
AMDAHL'S LAW

- Amdahl's law is still an upper limit, reality is often worse due to
  - Load balancing (processes waiting for each other)
  - Communications
  - I/O
  - Scheduling



PARALLEL COMMUNICATION

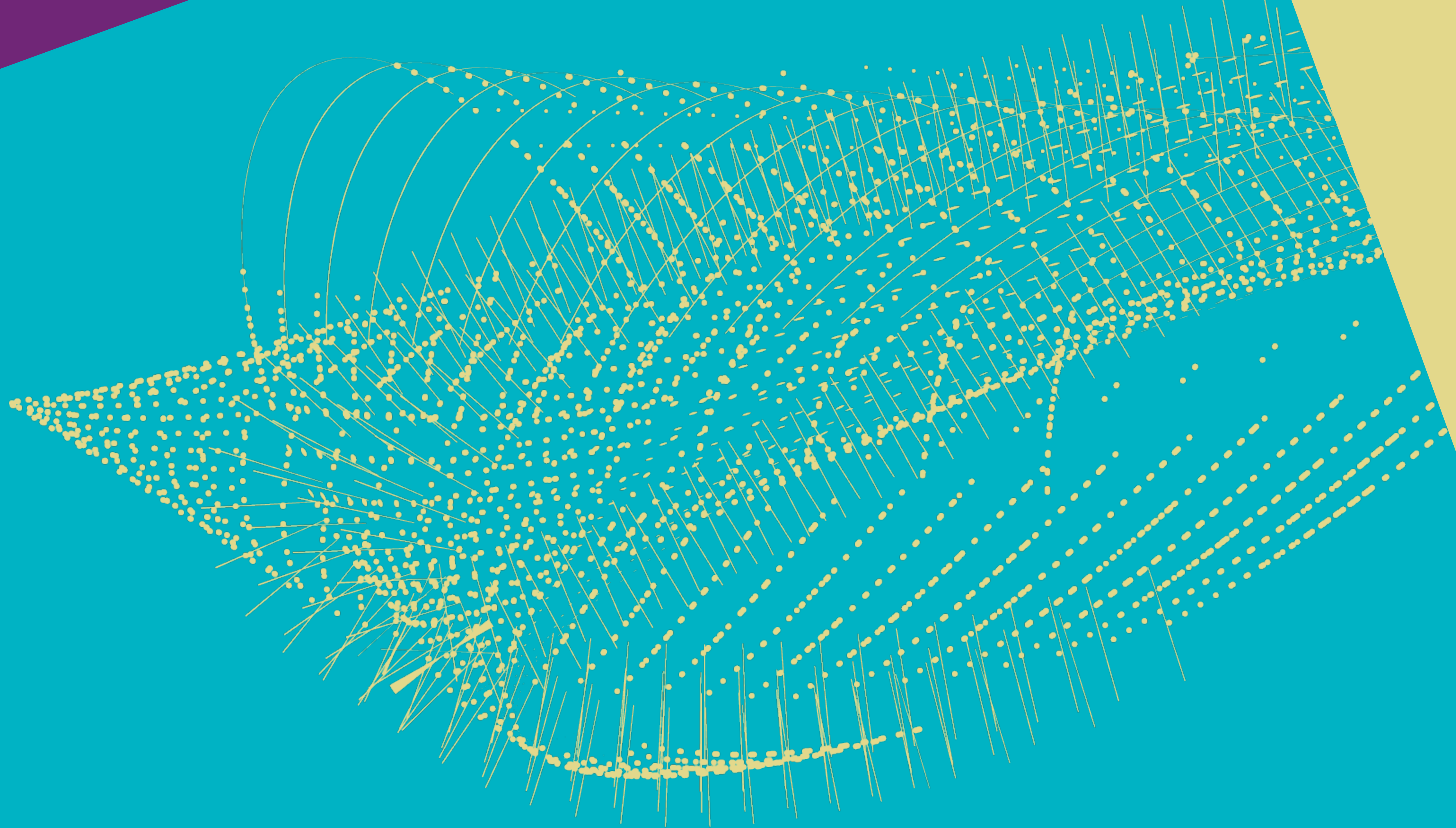
- When parallel tasks are dependent on one another, the processes executing those tasks may need to communicate
  - Example is a numerical simulation on a grid, in which each processor is responsible for simulating part of the grid. In this case the boundary between the processor's domains requires communication



PARALLEL COMMUNICATION

- Communication by **memory sharing**
  - Thread A can write something to the shared memory space, after which thread B can read it and react or vice versa
  - Common in **multithreading**
- Communication by **message passing**
  - Processes can directly send messages to each other with some protocol, or through a master process which communicates with all processes (**master/ slave configuration**)
  - When processors do not share any physical memory, this may be the only way to communicate. For example the case in **computing clusters**
  - Common in **multiprocessing**

Nikhef



PARALLEL PROGRAMMING: OPENMP

Zef Wolffs

OPENMP INTRODUCTION

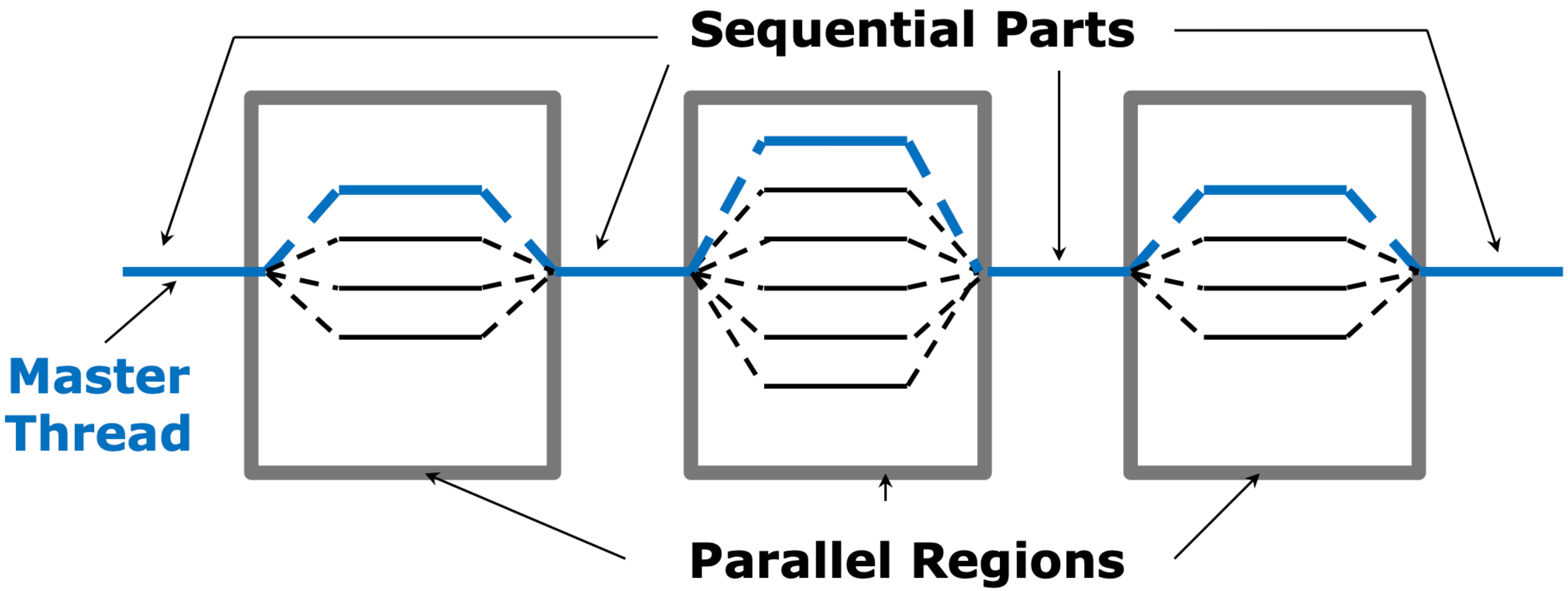
- OpenMP is a **library** for writing **shared memory multithreading** applications in c, c++, and Fortran that comes shipped with most compilers by default

- OpenMP is **mostly data-parallel**, meaning that the same operations are executed on different parts of some full dataset

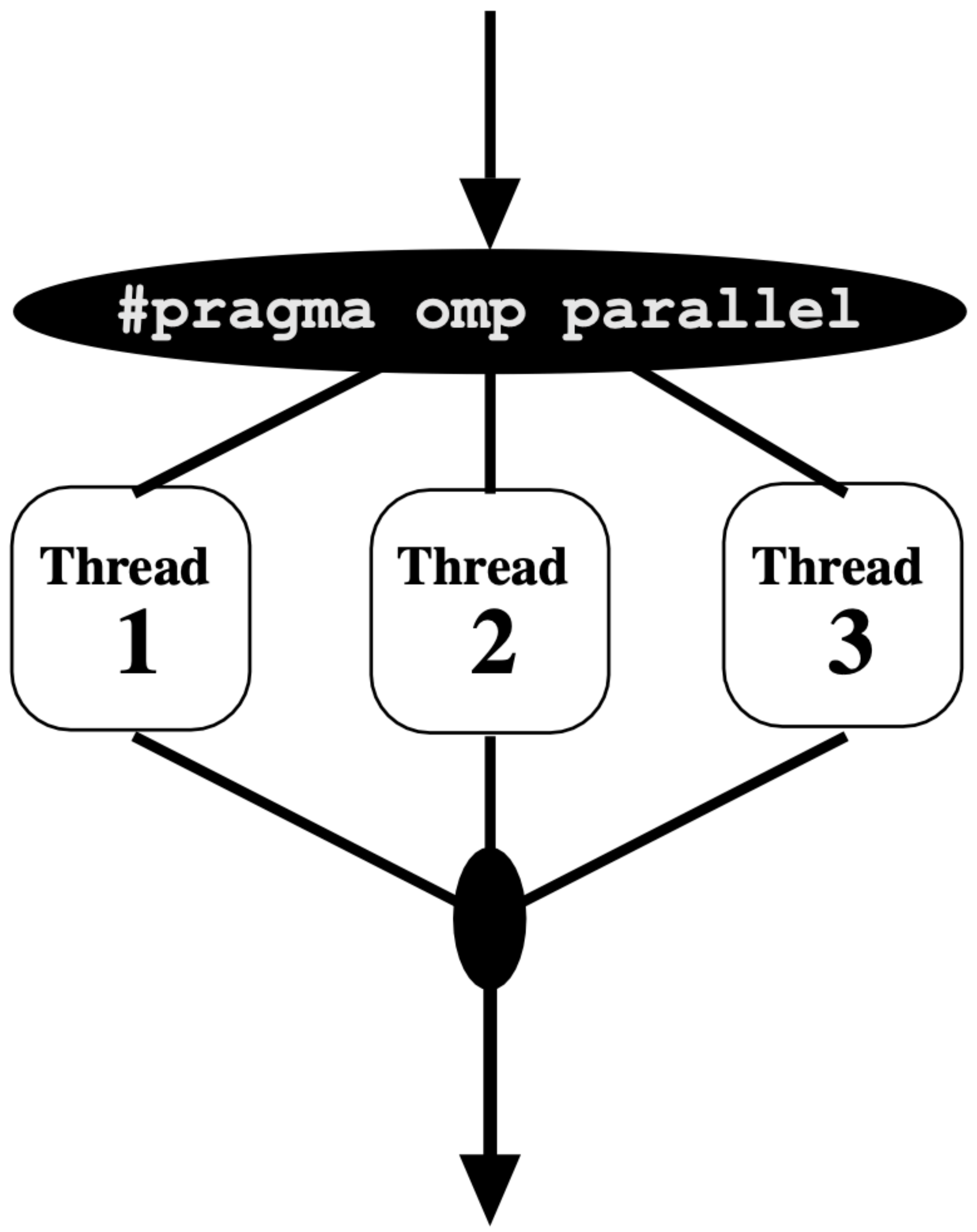
- OpenMP is **simple** and at a **high level of abstraction**, making use of **compiler directives** to achieve parallelism without requiring the user to write complex parallel code
  - `#pragma omp construct [clause [clause]...]`

OPENMP PROGRAMMING MODEL

- OpenMP uses the "**fork-join model**" for parallelisation
  - Threads are forked and later joined upon compiler directives



PARALLEL SECTIONS



- A parallel section is created as follows

C/C++ :

```
#pragma omp parallel
{
        commands
}
```

- Threads are spawned as pragma compiler directive is crossed and get killed as closing “}” is crossed
- Number of threads spawned is the number of processors by default, but can be set with the OMP_NUM_THREADS environment variable
- Data is shared among threads by default

PARALLEL FOR LOOP

Example 1 & 2

- The most simple parallel construct is the parallel for loop, implemented as follows

```
#pragma omp parallel
#pragma omp for
      for (i = 0; i < n; i++)
      {
        call_function(i);
      }
```

- Above can also be one-liner: `pragma omp parallel for`
- Splits loop iterations in threads
- Can be scheduled with the `schedule()` clause
  - Can be set to `dynamic` or `static`
  - Allows for setting a chunk size

VARIABLE SCOPE

- Scope in the context of c++ generally refers to the section of code in which a variable was initialised and can be used

- In OpenMP contexts scope refers to the set of threads that can see a variable

- By default variables defined **before** parallel section are **shared** between threads
  - The same address space is used for that variable in each thread
  - Need to be careful with **race conditions** for shared variables
- By default variables defined **within** parallel section are **private** to each thread
  - Each thread has a unique address space for all of its private variables
  - Private variables are not retained after the parallel section

SHARED AND PRIVATE VARIABLES

Example 3

- Outside of the default behaviour variables can also be set to shared or private as follows
  - Making variable x private

```
#pragma omp parallel private(x)
#pragma omp for
    for (i = 0; i < n; i++)
    {
        call_function(x);
    }
```

  - Explicitly making variable x shared (should already be the case by default)

```
#pragma omp parallel shared(x)
#pragma omp for
    for (i = 0; i < n; i++)
    {
        call_function(x);
    }
```

SYNCHRONISATION

Example 4

- OpenMP lets you synchronise threads to avoid **race conditions**
  - barrier `#pragma omp barrier`
    - All threads wait here for each other before continuing
  - critical `#pragma omp critical`
    - The execution of the block of code encapsulated within the critical directive is restricted to a single thread at a time
  - Atomic `#pragma omp atomic`
    - Same as critical, but only works for simple memory updates, for example adding two ints. It is faster though for those operations!

SECTIONS

Example 5

- OpenMP sections are blocks of code that can be executed in parallel

```
#pragma omp parallel
#pragma omp sections
{
    #pragma omp section
    {
        // Executed by one thread
    }
    #pragma omp section
    {
        // Executed by one other thread
    }
}
```

- By default, there is a barrier after the sections block such that threads wait for all threads to finish before continuing
  - You can disable this behaviour by adding a “nowait” clause after “sections”