Technologies

Warwick RSE

12/12/2022

Technologies

- Many people here will not be writing their own software

- Many of the people that are don't want to make major changes to it to use new resources

- The idea of going through the technologies here is to make you aware of what is available if you *do* want to write software to take advantage of parallel computers specifically

- We'll discuss what training is available more specifically

Terminology

- Before we can go through the technologies used in big computing we need to make some terms clear

- Some of these are terms that you will be familiar with but have technical meanings that you should be clear on

- Some of them will (likely) be less familiar

- You might want to refer to this in future

Basics

- Node - Individual computer in a cluster or taskfarm
- Processor - Generically a part of a computer that performs general purpose computational tasks
- Core - The smallest units of a processor that are capable of general purpose computational tasks.
    - 4 core processor is capable of performing 4 independent tasks simultaneously at full speed
- Socket - The number of processors that can be plugged into a node
    - 2 socket nodes have 2 processors plugged in them, each processor will in general have many cores

Computer Internals

- Process - Technical term for an individual program running on a computer. Available cores are split between processes by the **scheduler**

- Scheduler - An operating system program that ensures that all processes have fair access to the cores by determining what is running at any given time

- Not the same as a queuing system - this is much lower level and deals with core system processes as well

Computer Internals

- Thread - A part of an individual **process** that can run separately, usually on another **core**
  - You will often hear the term **multithreaded** for programs that use this to break up their tasks

- Fabric - Fancy term for the network that connects **nodes** in a **cluster**

Parallelism in Computers

- We talked about some real world things that demonstrate parallelism and some of the problems

- With computers things are harder because resources on a computer are not like physical objects

  - Only one person can “hold” a letter

  - It is perfectly possible for two programs to try and use the same resource at the same time

Parallelism in Computers

- At the simplest level - output files
- If two programs try to write to the same output file then one of two things can happen
  - "Race Condition" - What appears in the file will depend on exactly which program happens to write in which order
  - "Mutual Exclusion" - One program will get exclusive use of the resource and the other will have to wait (called file locking for files)
- Much easier to use separate files!

Parallelism in Computers

- The same is true with memory, but now between threads rather than processes (processes don't share memory)
  - You can **read** from memory (or a file) just fine in parallel
- Multiple threads trying to write to the same memory without control cause a race condition again
- Once again having different processors only writing to their own bits of memory is the easiest way of avoiding this

Parallelism in Computers

- Languages and libraries for parallel programming introduce objects (generally called Mutexes) to control mutual exclusion to resources

- This works but
  - Access to the excluded resources has to queue
  - You have to write the locking and unlocking of the mutexes carefully or performance will suffer unacceptably

Deadlocking

- Deadlocking is a kind of pathological mutual exclusion
- For example imagine that you have two files each of which need to be available to a program before it can proceed
    - Program A locks file 1, Program B locks file 2 neither of them have both files so neither can proceed
- Deadlocking is a real problem in parallel programming (of all kinds) if you share resources

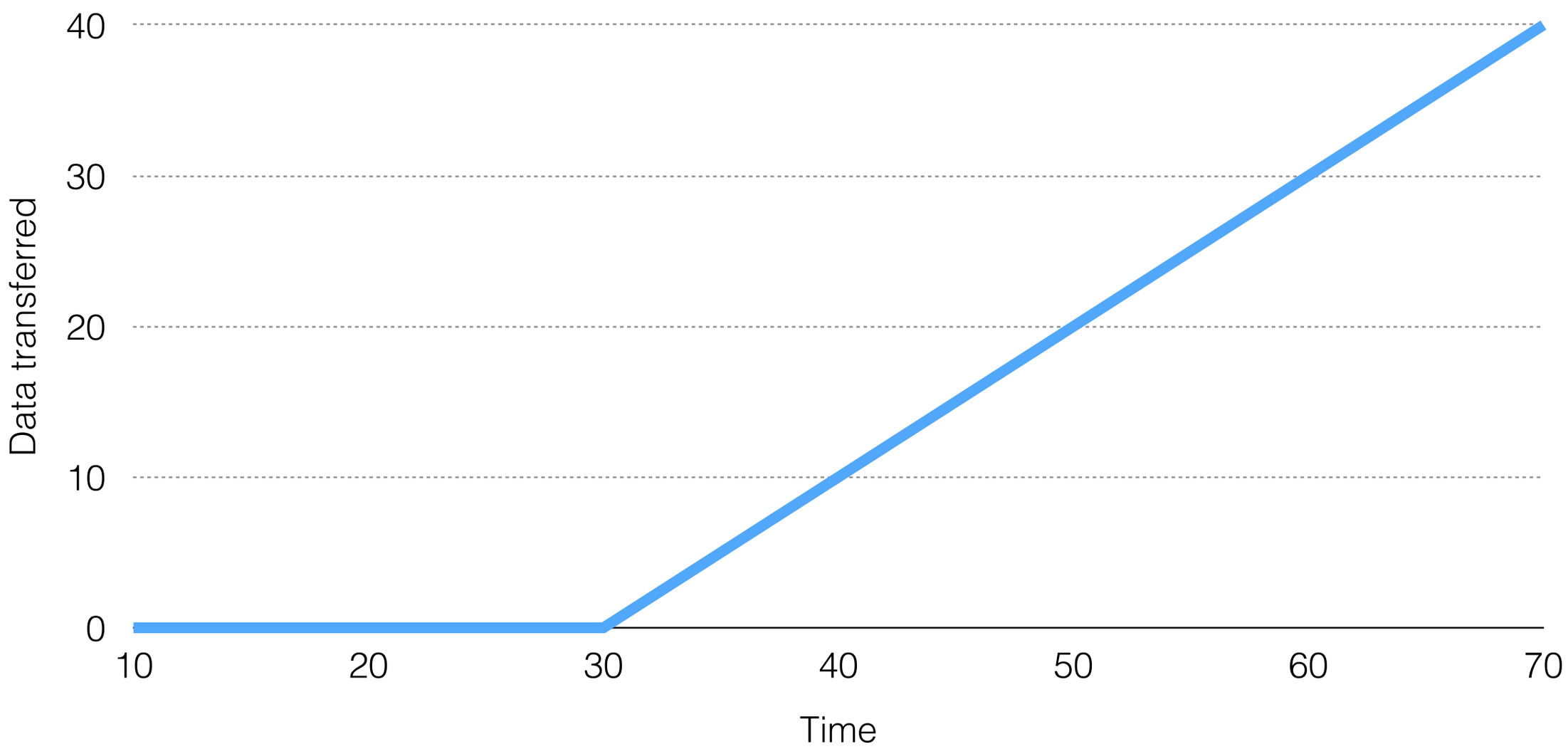
Shared vs Distributed Memory

- We mentioned that a single program shares memory no matter how many threads it has but that different processes don't share memory
- You can make different processes talk to each other in various ways but most of them are some kind of message based system
  - One program sends a message, the other one receives it
- This is often called "distributed" memory as opposed to "shared" memory and it can be generalised to running programs that communicate across different computers using the network between them
- You cannot (really) use threads to parallelise across multiple nodes

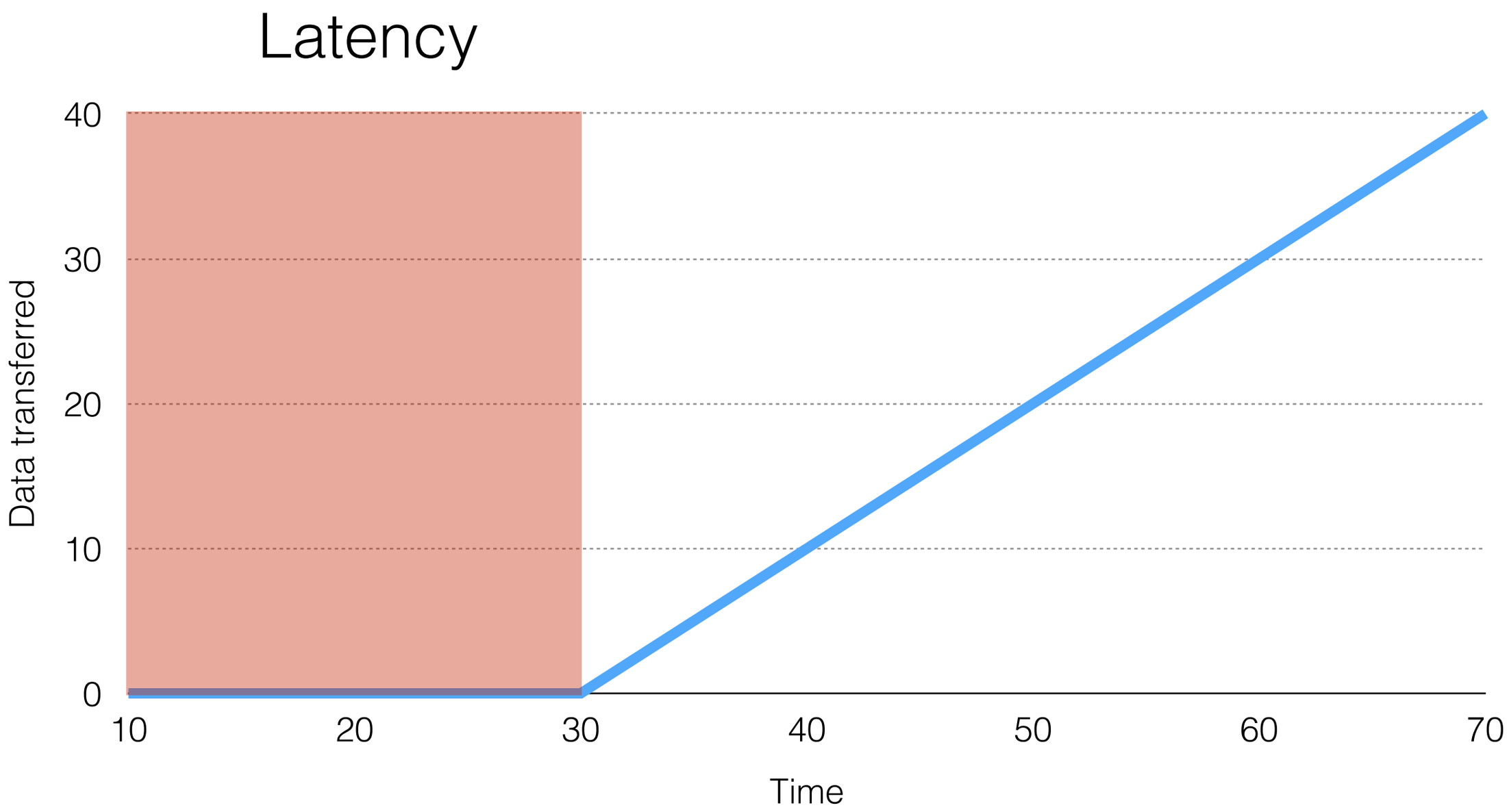
Data flow

- Latency - the time between a program asking for data and it being made available, usually measured in seconds (or milliseconds etc.)
    - So asking for data from a hard drive has a higher latency than asking for data from memory
        - High latency is bad
- Bandwidth - the rate at which data is transferred once it is flowing, usually measured in GB/s (or MB/s etc.)
    - So you also have a higher bandwidth asking for data from memory than from disk
    - High bandwidth is good

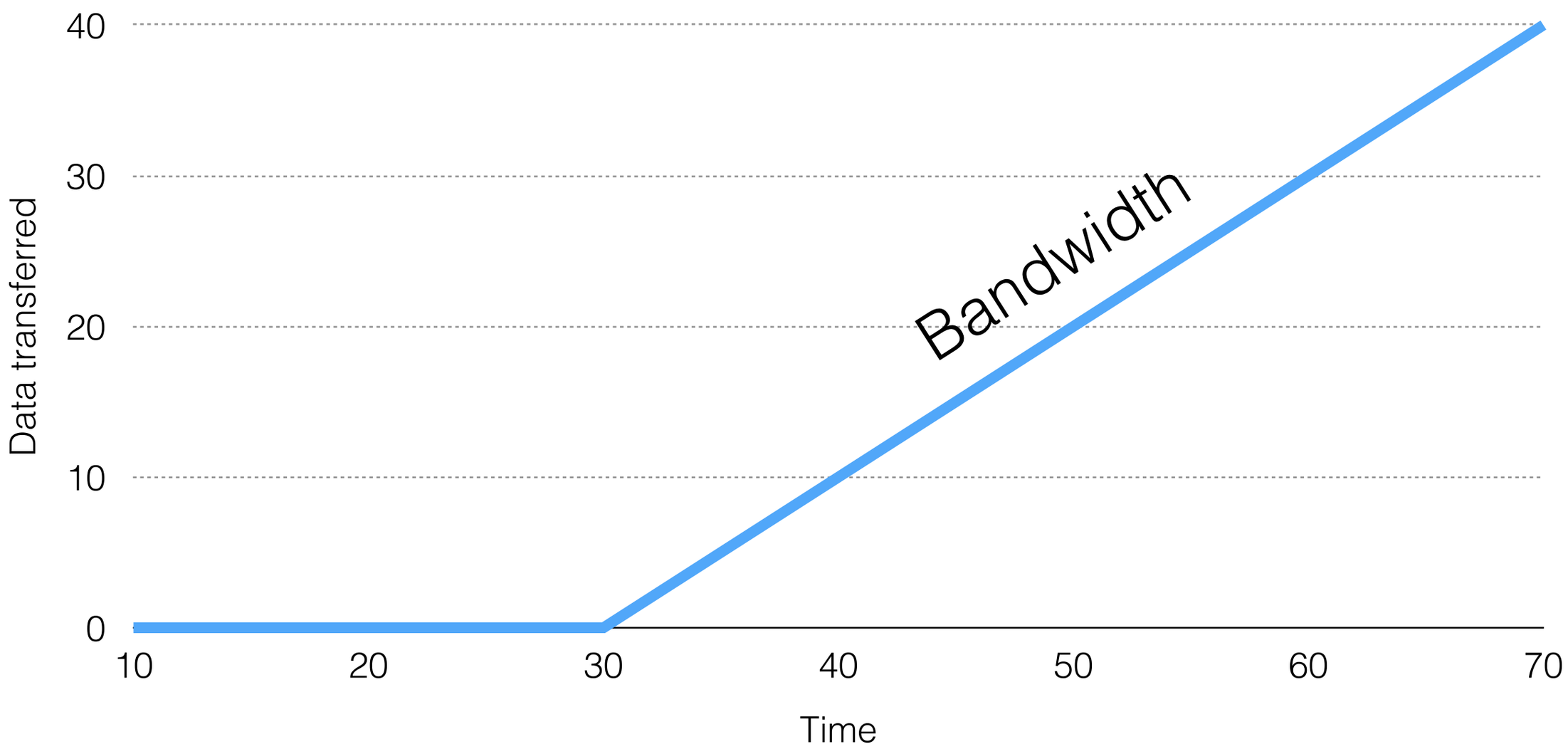
Data flow



Data flow



Data flow


40
30
20
10
0
Data transferred
10
20
30
40
50
60
70
Time
Bandwidth

Types of parallelism

Types of Parallelism

- Ultimately to make use of big computing you are making use of parallelism
    - Parallelism of tasks - doing multiple tasks at the same time
    - Parallelism of data - doing the same tasks to multiple pieces of data
- Tools are generally specialised to one end or the other but generally most tools cover a range

Types of Parallelism

- In some sense it is more important to distinguish between how coupled the parallelism is
  - Uncoupled (or embarrassing) parallelism - every task is unrelated to every other task. Even if running the same task on different data each set of data is self contained
  - Contingent parallelism - you know which tasks to run based on the results of earlier tasks
  - Read coupled - every task creates separate data but needs to refer to (but not modify) common data
  - Tightly coupled parallelism - working with one piece of data requires some access to other pieces of data (classical HPC)

Types of Parallelism

- In this course we're going to give you a very brief overview of a lot of technologies to let you know which one is of interest to you

- We're mainly going to talk about one approach to loosely coupled parallelism that has wide applicability
  - A specific talk this afternoon

- The aim of this section is that you are able to know where you should be putting your effort to do your research

- We're loosely going to go through the technologies from most coupled to least

MPI

MPI

- **MPI is a library!**
  - You have to write code to use it
  - You have to have an installation of an MPI library to compile or run a program containing MPI code.
- You can write programs that compile with and without MPI support but you have to do something to remove or replace the MPI commands
- MPI is the most common way of programming for distributed cluster systems

MPI

- MPI programs are separate programs that communicate with each other
  - Generally the same program running multiple times although that isn't necessary
- To exchange data between programs you write **sending** code that sends data and **receiving** code that receives the data
- Up to you to tie up sending and receiving code so that the program works

MPI

- MPI can be used for almost anything from just farming out tasks and getting results back right up to very tightly coupled problems where remote data is required regularly to continue computation

- MPI isn’t hard to learn from a computational perspective but mostly you don’t need to unless you are writing classical HPC code

Direct Threaded Programming

Threads

- Remember that threads are the way in which a program can be split up so that bits of it run on different cores
    - Literally just “Take this function and run it on another core”
- You can manually create threads in various ways
- All modern operating systems have the ability to create threads (but it is different on any given OS)
- Most modern languages (C++/Julia/Rust etc.) have threads built into them
    - More portable than using the OS level code

Threads

- Threads are very low level (i.e. you have to actually think about how the computer works)
- Problem is that creating and destroying threads is **expensive**
- One way or another you want to keep threads alive and give them new work to do (generally called a **thread pool**)
    - Most languages with thread support don’t help you with this!
- Generally want to avoid writing threads manually!
    - Lots of work and easy to make a mistake affecting either performance or correctness

Python Numba

- Python has a library called Numba that does many things
- The most important thing that it does is **compile** Python code
  - There are quite a few things that restrict how well it works but in theory you can just add **@jit** as a decorator to a function and that function can become 10-50x faster
- It can also try to compile code so that it runs with threads by adding a parameter to **@jit**
  - You don’t have much control and it fails as often as it succeeds but if it does work it can give a useful speedup

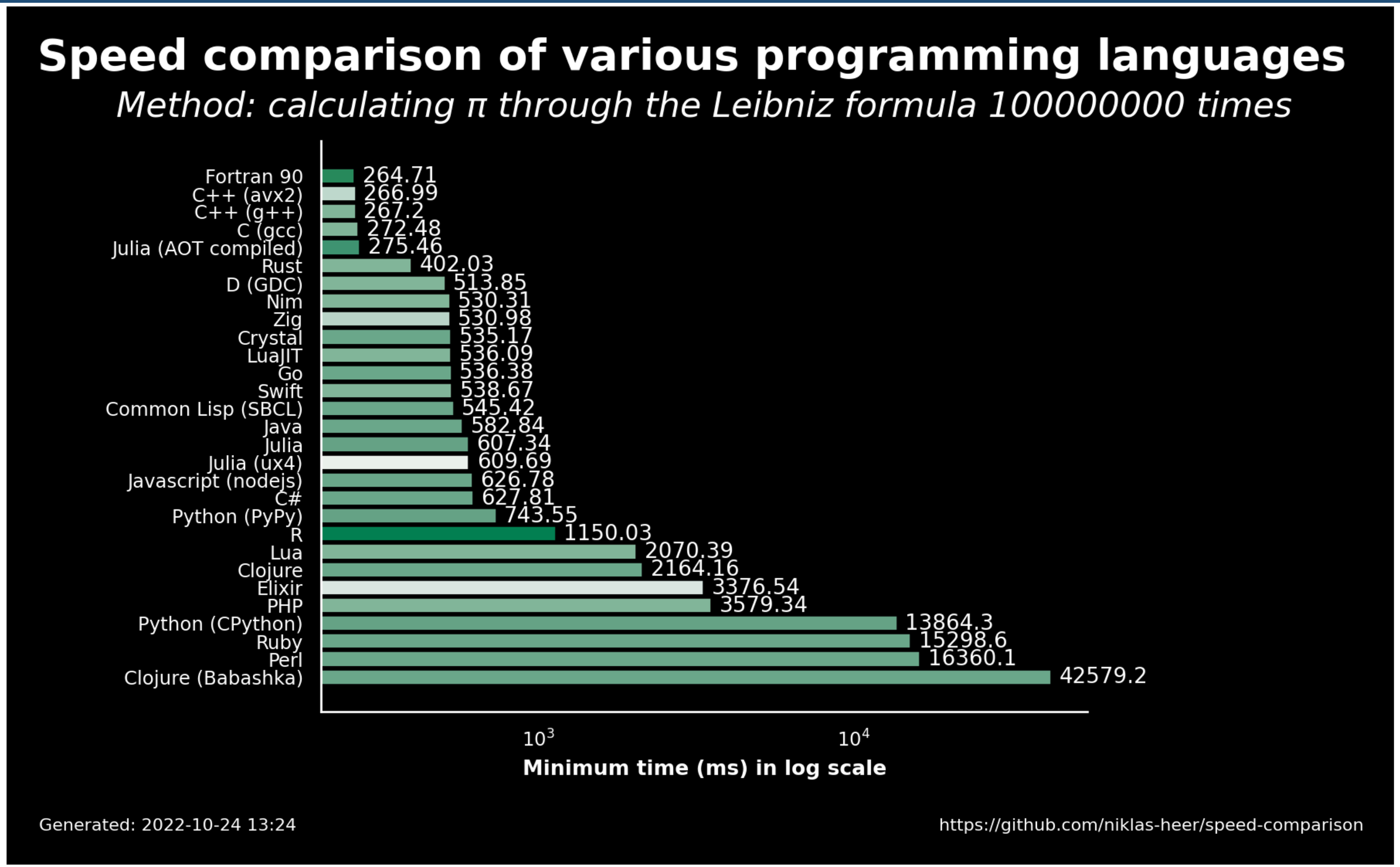
Note on Python

- Python is a very popular language but it is **not** well suited to parallel programming
- Python's native parallelism model is a threading model related to Pthreads
- BUT the normal Python interpreter actually doesn't work in parallel! (Others like PyPy don't have this problem)
- Your threads will all queue up one after the other until they leave Python code and enter an external library of some kind
- Called the Global Interpreter Lock (GIL)
- There is work going on to remove the GIL but it isn't clear when this is going to become mainstream

Note on Python

- If your code uses a lot of external library code
  - Or uses the Numba JIT compiler
- then threading might help you
  - **Multiprocessing** does work but invokes multiple Python interpreters - process based parallelism through the backdoor and not very fast. Only use it if each task takes at least 30 seconds and preferably a few minutes to complete. Think how long it takes until Python starts for you!
  - If your code isn't using libraries heavily in Python you might get the best effort/reward ratio from rewriting it in a compiled language rather than using parallelism at all

Python performance



OpenMP

OpenMP

- OpenMP is a combination of
  - a set of directives that tell the compiler how to parallelise bits of the code
    - Needs an OpenMP aware compiler to be respected but are ignored by a non OpenMP aware compiler
  - a library that gives your code access at runtime to information about the number of processors, how to split work up etc.
    - Will fail to compile on a non OpenMP aware compiler

OpenMP

- OpenMP allows for almost completely general parallel programming

- But by **far** the most common use of OpenMP is to split loops up so that different bits of the loop are handled by different processors

  - There's an obvious limitation to this: only loops that have independent iterations can be parallelised over

  - If you have a loop to advance a quantity in time then you **can't** parallelise that since iteration 2 depends on iteration 1, which depends on iteration 0 etc. etc.

OpenMP

- OpenMP (used like this) works by creating threads and splitting up the work done by different iterations of a loop between threads

- It handles things like thread pools etc.

- If you are writing code in C/C++/Fortran that does most of it's work inside loops without interdependencies then it can be well worth learning the basics of OpenMP

    - If you get lucky then you can get parallel performance with very little work

- More advanced OpenMP can work with more complex parallel problems and even do things like move calculations to a GPU

GPUs

GPU Programming

- Not really going to cover GPU programming
- The basic idea is “threads but moreso”
- GPUs have many more cores than CPUs do
  - 64 (soon to be 94) for CPUs (x2 for dual socket machines)
  - 16,384 for the GeForce 4090 (slightly higher for the “Hopper” data centre card)
- You have to be able to program your work to split 16,384 ways to realise the performance
- Theoretical performance of GPUs ~ 5-10x CPUs

GPU Programming

- Most GPU programming is done using Nvidia's proprietary CUDA language
- You can also use OpenMP to program GPUs (the directives look different though)
    - There are similar but different things called OpenACC and OpenCL which are more meant for GPU programming
- There are libraries and frameworks such as Kokkos or SYCL that are intended to allow you to write code that will run on CPU or GPU

GPU Programming

- Many libraries have what is generally called **GPU offload** - i.e. it moves some of the calculation to the GPU
    - Very common with libraries involved in AI and machine learning Tensorflow and things built on top of it
- Check whether your library supports GPU offload and how to use it but be aware that you may not get a performance benefit from it
    - GPUs are fast but moving data on and off them isn't!

Threaded Libraries

Libraries with internal threads

- Many libraries include their own internal threading (c.f. GPU offload)
- Classical example are libraries for linear algebra
- Plenty of other libraries do it to
- Generally this is good
  - Might be faster to not use all of the cores like this though! (Even for good programmers it is hard to get good performance from splitting one task)
  - Find a balance between using the available cores for threading in your library and for multiple tasks
- READ THE MANUAL

Libraries with internal threads

- Can also be bad if you don't know about it
- If you don't know that your library is creating threads then you might want to run one program per core yourself
- That would lead to you creating n_processors^2 total threads which will not perform well!
- You want the total number of threads of your problems equal to the number of cores in the machine that you are running on
- READ THE MANUAL

Language Features

Languages with Parallelism

- Many languages have features above the level of threads to perform parallel functions

- C++ - Parallel Algorithms (sort, for_each etc.) from C++17 - just add an execution policy and they will run in parallel

- Fortran - Coarray Fortran - full distributed programming model - adds extra index to arrays that is split across processors

    - Powerful but not easy to use

Languages with Parallelism

- Python, C++ and various other languages have things called futures and promises
    - A way of packaging up work so that it can be run asynchronously and only actually block waiting for the result when it is wanted by another computation
        - Asynchronously doesn't necessarily mean on another processor but it certainly can do
    - In some languages futures are automatically or semi-automatically mapped onto multiple processors (e.g. Python) in others you have to manually run a future in another thread (e.g. C++)
- Futures and promises are a good way of splitting up work but they are still fairly low level and can be hard to work with

Task Execution

Libraries

Libraries to Run Multiple Tasks

- Since threads work by running specific functions on separate processors you can obviously use them for running multiple tasks
    - You have to worry about lots of technical details like thread pools though
- There are libraries that deal with this for you automatically (and often provides more as well!)
- C++ - Intel One Threaded Building Blocks (part of their ONEAPI hence the name)
- Python - Multiprocessing (built in), joblib(library), Dask(library)

Libraries to Thread your Code

- Multiprocessing and joblib are almost entirely intended to work on separate tasks
    - They can’t get around the GIL problem but they are as efficient as they can be using multiprocessing
- Dask is a massive library covering a lot of things that we will mention later
- OneTBB is more complex (it also handles parallelism over data) but if you want to run multiple tasks there are sections intended for that, especially
    - **parallel_invoke, parallel_for** and others are intended to run over a set of tasks