

Parallel Programming models

Different ways to exploit parallelism

Partners



MANCHESTER
1824
The University of Manchester



Ian Harrow Consulting



Funding



Reusing this material



This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

http://creativecommons.org/licenses/by-nc-sa/4.0/deed.en_US

This means you are free to copy and redistribute the material and adapt and build on the material under the following terms: You must give appropriate credit, provide a link to the license and indicate if changes were made. If you adapt or build on the material you must distribute your work under the same license as the original.

Note that this presentation contains images owned by others. Please seek their permission before reusing these images.

Outline

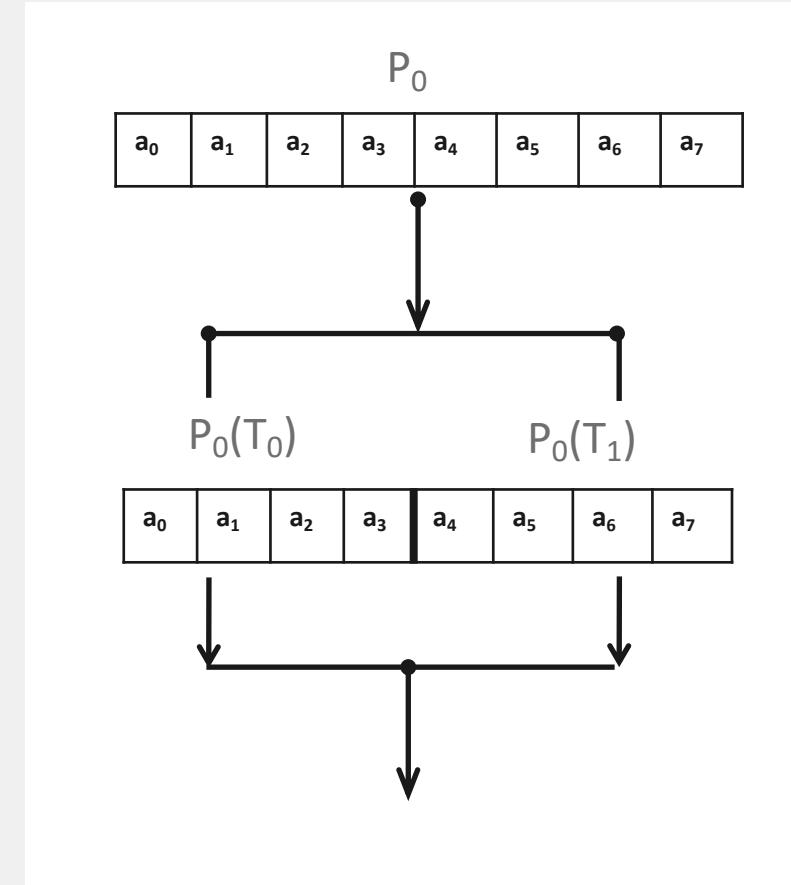
- Shared-memory parallelism
 - Threads
 - OpenMP
 - Shared-memory in HPC machines
- Message-passing (distributed) parallelism
 - Processes
 - MPI libraries
 - Distributed-memory in HPC machines
- Exploiting GPU parallelism
 - CUDA
 - HIP

Shared Memory

Thread-based parallelism

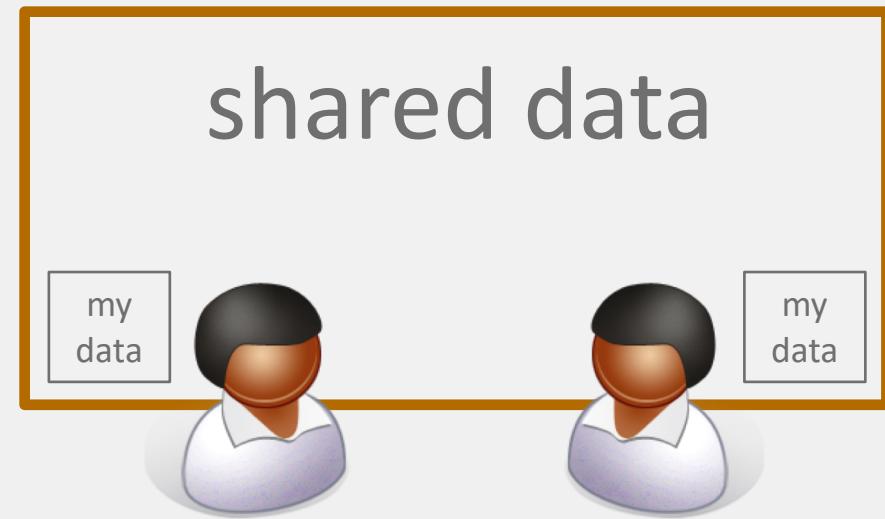
Shared-memory concepts

- Threads can see the data of the parent process
- These threads are assigned to different physical cores on the processor
- The threads can operate on different parts of the data – this allows for parallel speed up



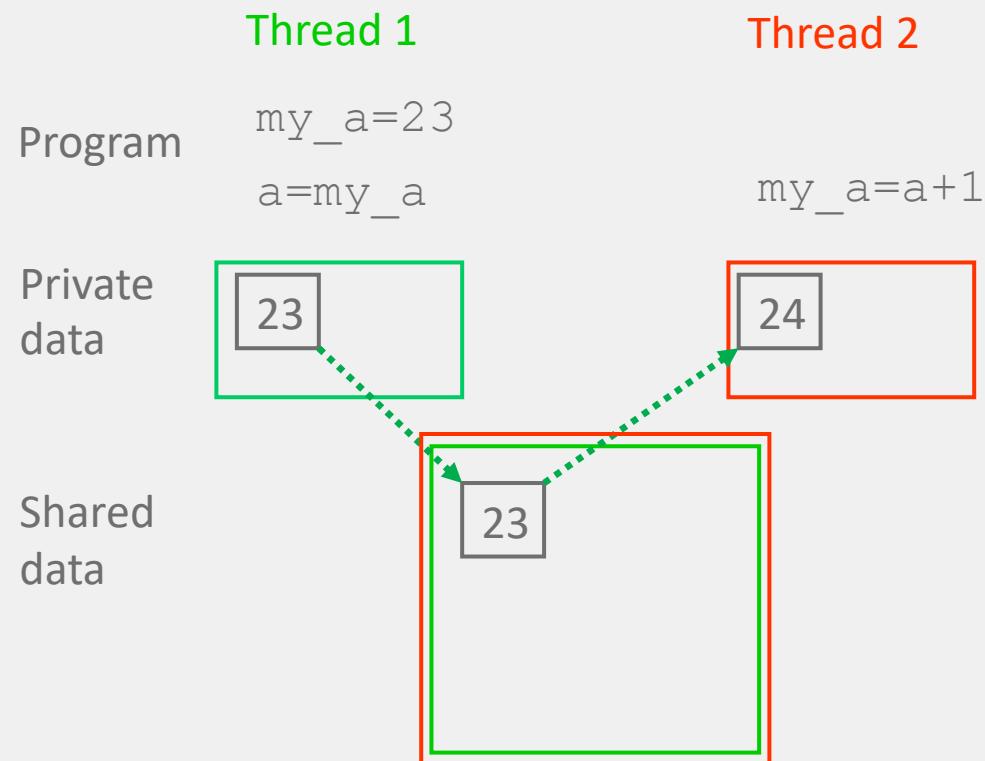
Analogy

- One very large whiteboard in a two-person office
 - the shared memory
- Two people working on the same problem
 - the threads running on different cores attached to the memory
- How do they collaborate?
 - working together
 - but not interfering
- Also need *private* data



Thread communication

- Each thread can read and write to the shared data
- Therefore they can communicate by reading and writing to this shared space.
- Synchronisation crucial for shared variables approach.
 - thread 2's code must execute *after* thread 1



OpenMP

- OpenMP is an Application Program Interface (API) for shared memory programming using threads
 - You can expect OpenMP to be supported by all compilers on all HPC platforms
 - Example usage: `gcc -fopenmp mycode.c -o mycode`
- Parallelism is implicit ie. a lot is abstracted from the programmer
 - You specify which parts of the program you want to parallelise and the compiler produces a parallel executable

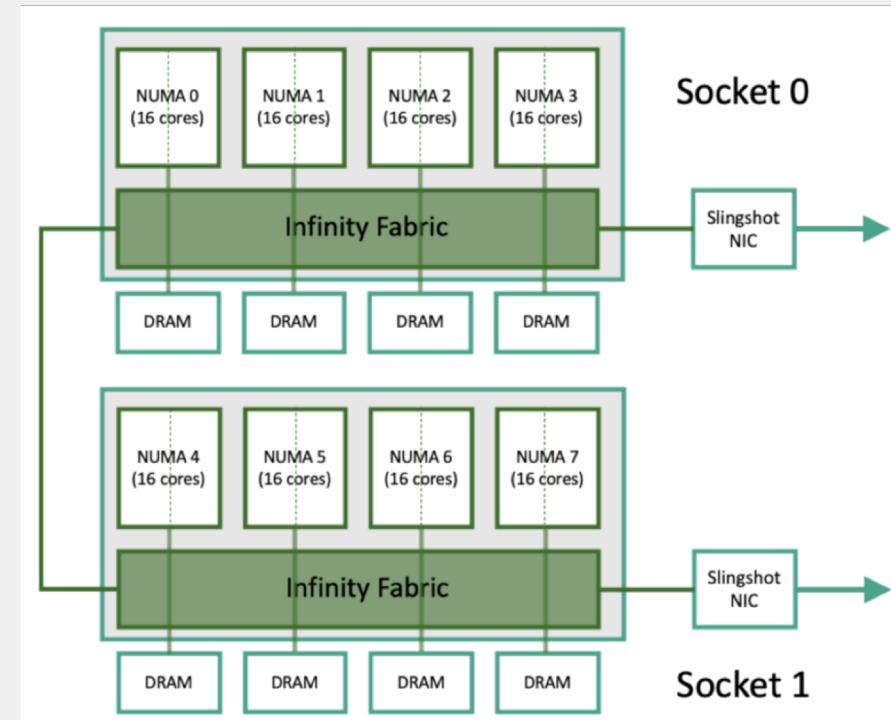
Example: OpenMP Loop parallelism

- The most common form of OpenMP parallelism is to parallelise the work in a loop
 - The OpenMP directives tell the compiler to divide the iterations of the loop between the threads

```
#pragma omp parallel shared(a,b,c) private(i)
{
    #pragma omp for
    for (i=0; i < N; i++) {
        c[i] = a[i] + b[i];
    }
}
```

Hardware

- Shared-memory parallelism can only take place on cores that share memory i.e a single node or memory region.
- Would expect poor performance if used across multiple memory regions.
- We are usually restricted to only a few 10s of threads on most machines



Usage

- Number of threads set by using the environment variable OMP_NUM_THREADS e.g.

```
export OMP_NUM_THREADS=2
```

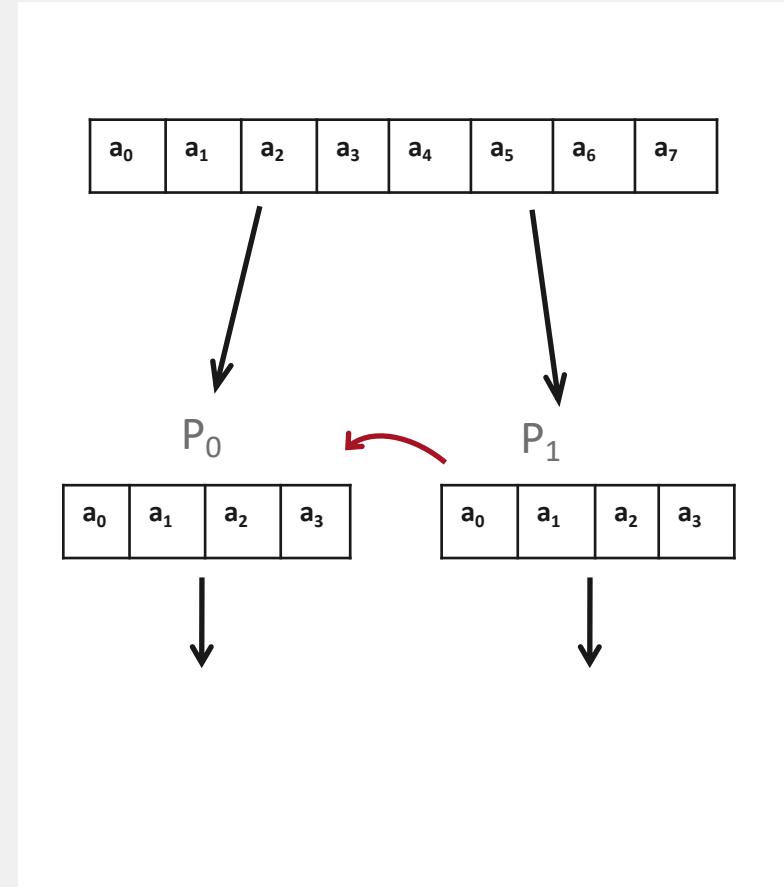
- On ARCHER2 (Slurm), also set the SBATCH option --cpus-per-task. This is the number of cores allocated to a single process (task) and should be the same as the number of threads to ensure one core per thread.

Message Passing

Distributed parallelism

Distributed memory parallelism

- The data is distributed amongst the processes.
- Create smaller sub-problems - parallelism
- Processes can only see their own data.
- Communication between them is done via messages.
 - A process can send data to another process



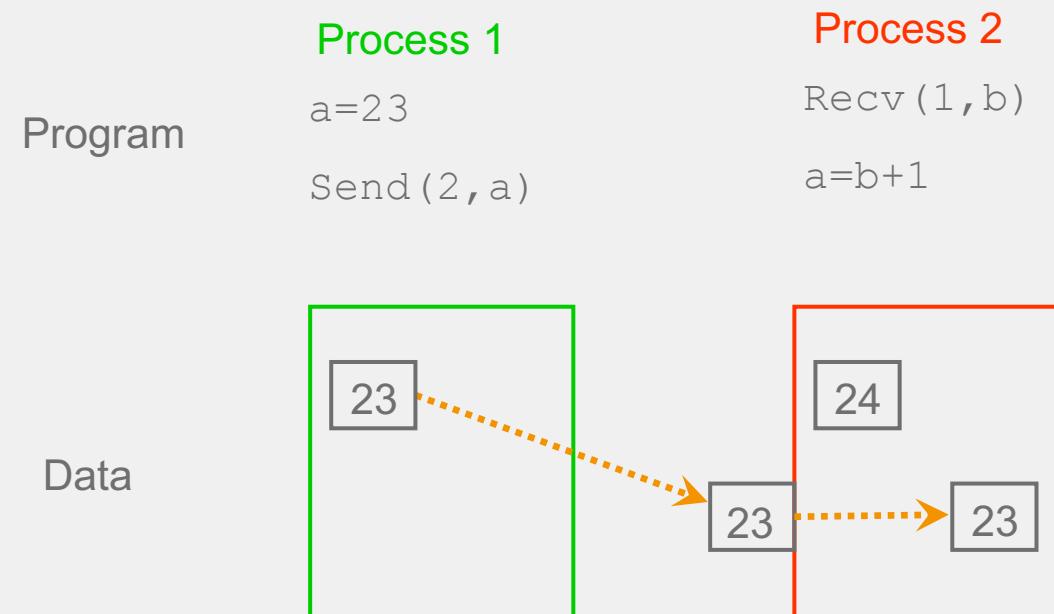
Analogy

- Two whiteboards in different single-person offices
 - the distributed memory
- Two people working on the same problem
 - the processes on different nodes attached to the interconnect
- How do they collaborate?
 - to work on single problem
- Explicit communication
 - e.g. by telephone
 - no shared data



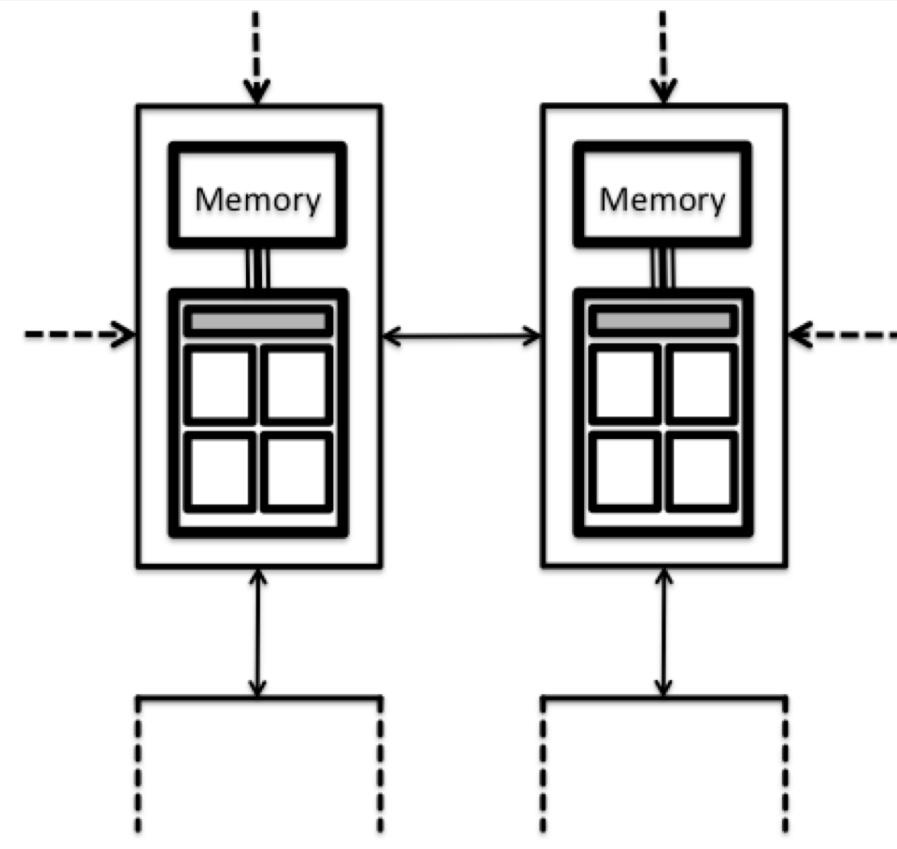
Process communication

- Explicit messages between processes.
- Messages are two-sided, they require a send and a receive.
- To send a message must specify the data to send and the destination process.
- The receive counterpart must match the sender process number.



Hardware

- Processes can be distributed across multiple nodes with one process per core, and messages across in the interconnect
- Intra-node messages are fast.
- Message passing is the standard parallelism for modern machines as it can exploit cores across multiple nodes.

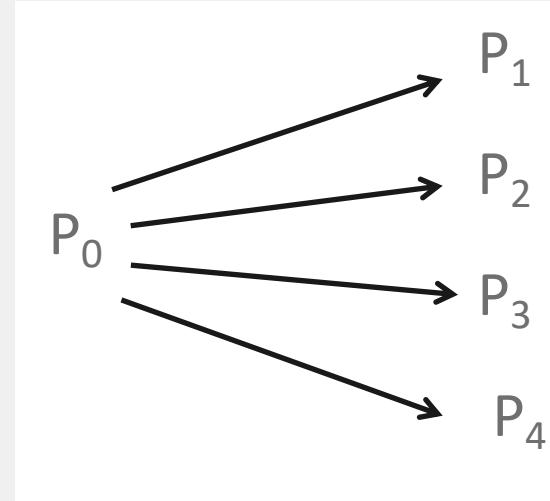


Message passing interface (MPI)

- MPI is a portable **library** used for writing parallel programs using the message passing model
- There are a number of different implementations but all should support the MPI standard
 - Examples: MPICH, Open MPI, Intel MPI
- In message-passing all the parallelism is **explicit**
 - The programmer needs to decide how to decompose the problem over processes
 - Then **what** to send/receive and **when/how often**

Communications

- Point-to-point communications
 - A message sent by one process and received by another
- Collective communications
 - Involve “all” processes
 - E.g. Broadcasting data to all processes
 - More efficient than many point-to-point messages



Usage

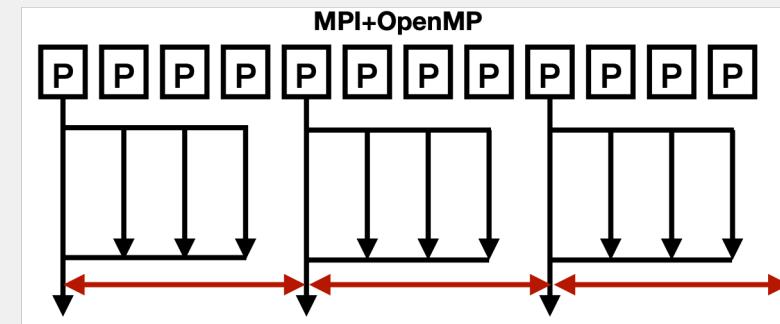
- For pure MPI one process is associated with one physical core
- The job launcher launches the same program across multiple processes.
 - e.g. srun, mpirun, mpiexec, aprun
- Need to specify the number of processes in the job
 - `--ntasks-per-node` or `--ntasks`
 - This is the number of copies of the program created

MPI+OpenMP

Mixed mode parallelism

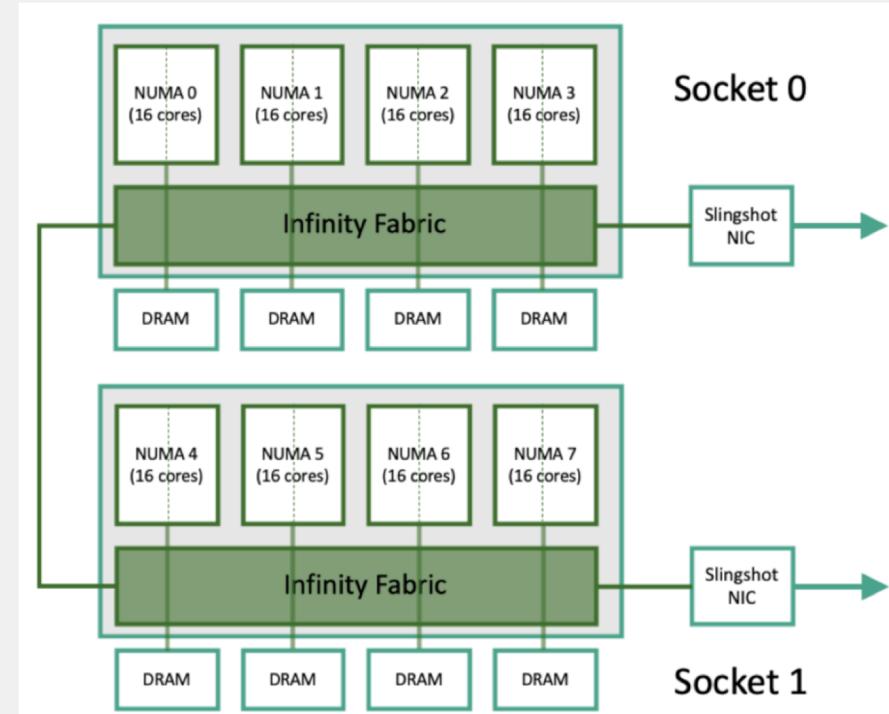
Hybrid MPI+OpenMP

- Modern HPC machines are distributed clusters
 - Cannot use shared memory (threads) across nodes – limited to a single node
 - Can instead use shared memory within a node and message passing between nodes. i.e. each process spawns multiple threads
- This is known as MPI+OpenMP
 - Has the possibility to improve the performance
 - Can also decrease the overall memory requirements
 - Less copies of data on a node



MPI+OpenMP Usage

- Implemented in lots of HPC applications
 - However not always more performant than using a single thread (e.g. pure MPI)
 - Number of threads to use needs to be tuned to the use case and the machine
 - Need to be aware of the underlying node structure when running e.g. memory regions



Usage

- Need to think about the placement of processes and threads on the node architecture
 - Ensure each thread is placed on a separate core
 - Want to populate entire nodes
- Set the no_processes x no_threads = total_no_cores
 - Options in the job script to do this
- The shared memory portion (i.e. groups of threads) should not span more than one NUMA region – see exercise

GPU parallelism

GPU based parallelism

- GPUs ideal for some HPC applications as they are designed for doing many numerical operations at once
 - Many many threads (many many cores)
 - Ideal if cores execute same operations (e.g. on different subsets of data)
- Specific numerically intense calculations offloaded to GPU
 - Rest of code runs on CPU
- GPU (“device”) memory is separate from main (“host”) memory
 - Requires copying data onto and off the GPU
- Application may run on multiple GPUs per node, and on many nodes
 - Communication between GPUs typically MPI between host memory spaces
 - NVLINK enables fast communication directly between GPU memory spaces (i.e. bypassing host memories)

GPU programming

- Nvidia GPUs: CUDA
 - Proprietary Nvidia software (available on all systems with Nvidia GPUs)
 - Application Programming Interface (API) and runtime platform
 - Rewrite numerically intensive code as GPU-specific function: *kernel*
 - Includes functions to shift data between CPU and GPU memory
- Most recent OpenMP standard (4.0) incorporates simple syntax for offloading execution to GPUs
- HIP – kernel language for Nvidia and AMD GPUs
 - Can convert CUDA code to HIP (increased portability)

Scientific libraries

- Scientific libraries contain highly optimised code used by different scientific (HPC) applications
 - Library code contains implementations of common mathematical routines
 - Dense & sparse linear algebra, fast fourier transforms, etc.
- Parallel (MPI & thread-based) versions of many of these libraries are available
 - Critical for performance of many large-scale HPC applications
 - Includes parallel IO (writing and reading large amounts of data quickly in parallel)
 - More recently, versions that offload to GPU using CUDA increasingly available

Summary

- Shared-variables parallelism e.g. with OpenMP
 - uses threads
 - requires shared-memory
 - easy to implement but limited scalability
- Distributed memory e.g with MPI
 - uses processes
 - can run on any machine: messages can go over the interconnect
 - harder to implement but better scalability
- MPI+OpenMP
 - Shared memory within a node, message passing across nodes
 - Can have advantages over pure MPI, but not always
- GPU parallelism
 - Using GPUs for HPC