XJCO3221 Parallel Computation

Peter Jimack

University of Leeds

Lecture 8: Introduction to distributed memory parallelism

Previous lectures

In the last six lectures we looked at **shared memory parallelism** (SMP) relevant to *e.g.* multi-core CPUs:

- Each **processing unit** (*e.g.* thread, core) sees **all** memory.
- Want to achieve good **scaling**, *i.e.* speed-up for increasing numbers of cores.
- Without proper **synchronisation**, results can be **non-deterministic**.
- Dependencies can lead to **data races**.
- Can reach **deadlock** if threads wait for synchronisation events that never occur.

This lecture

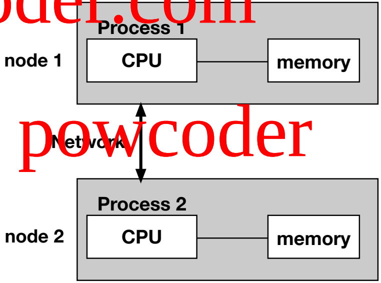
This lecture is the first of six on **distributed memory parallelism**, and we will see that some (but not all) of these issues remain relevant:

- Each **processing unit** sees only a **fraction** of total memory.
- **Data dependencies** treated using **explicit communication**.
  - No **data races**.
- Performance considerations remain the same, except now the primary parallel overhead is **communication**.
- Improper synchronisation can still lead to **non-determinism** and **deadlock**.

Distributed memory systems

Multiple **processes** (rather than threads) that communicate *via* an **interconnection network** or **'interconnect'**.

- For instance, one process per **node**, *e.g.* desktop machine.
- Each process has its own **heap memory**.
- If a process needs data currently held on another node's memory, must **communicate** over the network.



Current fastest supercomputer[1]

Fujitsu Fugaku, RIKEN, Kobe, Japan

- ARM-based A64FX CPU.
- 48 compute cores, and 2 or 4 assistant cores.
- Total 7,630,848 cores.
- No GPUs.
- Draws nearly 30MW of power.
- Benchmarked $\approx$ 442 PFLOPS.
- 1 PFLOPS $= 10^{15}$ FLOPS.
- 1 FLOPS = 1 floating point operation per second.

[1]As of Nov. 2021; `top500.org`.

Clusters as distributed systems

Supercomputers share features with other **distributed systems** such as data centres:

- Nodes perform calculations in parallel.
- Coordination requires explicit communication; there is no 'global clock.'
- May have high energy demand and cooling requirements.

Here focus on **H**igh **P**erformance **C**omputing (HPC) clusters:

- Individual cluster nodes use the same **operating system**.
- Cannot usually be **addressed individually**.
- Requires a special **job scheduler**.

The interconnection network or ‘interconnect’

For the local area networks within HPC clusters, communication between nodes is carried over high performance **interconnects**.

- **Gigabit Ethernet** and **InfiniBand** are the most common[1].
- Latencies (*i.e.* delays) of around $1\mu s$.
- Bandwidths (*i.e.* throughput) of around 1-100 Gb/s.

These numbers are improving with time but **more slowly than CPU performance**.

The need to reduce communication overheads will only become more important in the foreseeable future.

[1] As of Nov. 2021; see `top500.org`.

Network topology

If data sent *via* intermediate nodes, latency is increased.

- Each node must parse data packet and decide where to send.

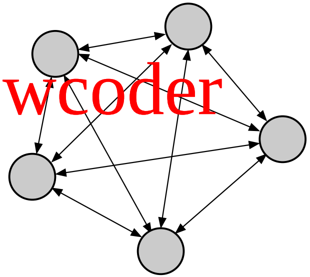
Therefore want smallest **paths** between nodes.

Network as a **graph** $G(V, E)$:

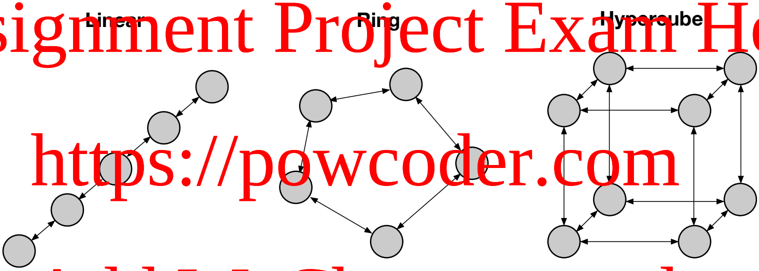
- V = nodes *(vertices)*.
- E = connections *(edges)*.

Want G with smallest **diameter** δ *(largest path length between nodes)*.

A **complete graph** *(right)* has $\delta = 1$, but is impractical *(too many connections for each machine)*.



Example topologies for *p* nodes



$\delta = p - 1$ $\quad\delta = \left\lfloor \frac{p}{2} \right\rfloor$ $\quad\delta = \log_2 p$

Hypercube topology preferred due to its short path lengths[1].

[1]Rauber and Rünger, *Parallel programming for multicore and cluster systems* (Springer, 2013).

Processes *versus* threads

Recall from Lecture 2 that **processes** communicate with other processes using *e.g.* sockets.

- Must have **at least** one process per node to communicate across the network.

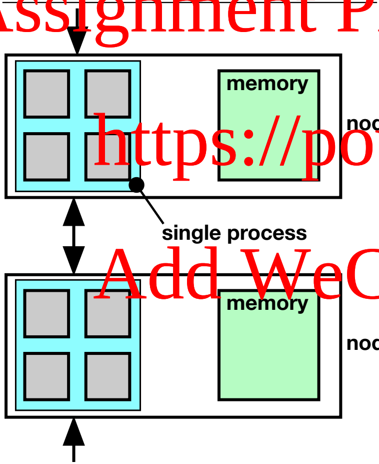
For multi-core nodes, could have one **multi-threaded process** per node, with one thread per core.

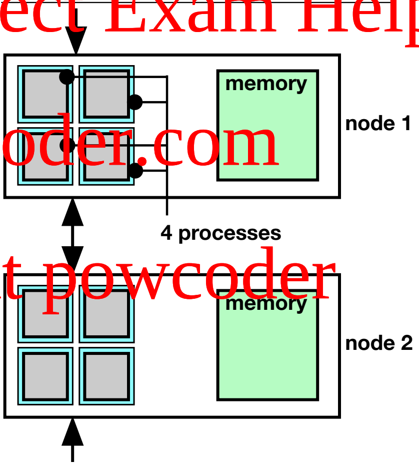
- Avoids communication **within** a node.
- Combination of OpenMP and MPI is quite common (*'hybrid'*).

For simplicity, we consider one **single-threaded process per core**, and therefore **multiple processes per node**.

Example for quad core nodes

- One 4-thread process per node
- 4 one-thread processes per node





Books

Wilkinson and Allen *[Lecture 1]* covers distributed memory parallelism (MPI), and a little OpenMP, but no GPU.

- General parallel algorithms but few code examples.
- Slightly old (2005) and covers architectures we will not consider *(e.g. distributed shared memory systems)*.

A more practical book for MPI coding is:

- **Parallel Programming with MPI**, *Pacheco* (Morgan-Kauffman).
  - Old (1997), only covers distributed memory systems and MPI.
  - Many code examples and snippets.

Distributed HPC programming

For distributed HPC, there is essentially only one option[1]: **MPI**.

- Stands for **Message Passing Interface**.
- Specifies a **standard** for communication *('message passing')*.
- MPI v1.0 finalised in 1994.
- MPI v3.0 finalised in 2012, now widely implemented.
- Fully supports C, C++ and FORTRAN.
  - Most online examples are in one of these languages.
- Unofficial bindings for Java, MATLAB, Python, ...

[1]Has superseded PVM = Parallel Virtual Machine (1989). Others such as Spark, Chapel *etc.* not (yet?) widely used in HPC.

Implementations

The MPI standard only defines the **interface**; it is still down to a vendor to provide an **implementation**.

- Code should be **portable** between implementations.

There are various **freely available implementations**:

- **MPICH**: `www.mpich.org`
- **OpenMPI**: `www.open-mpi.org`
- Don't confuse OpenMPI with OpenMP ...!

There are also commercial implementations:

- *e.g.* Intel MPI, Spectrum MPI (IBM).

Installing MPI

The system `cloud-hpc1.leeds.ac.uk` has OpenMPI[1] installed:

```
module load mpi/openmpi-x86_64
```

For personal Unix machines, should be straightforward to install *(cf. links on previous slide)*.

- Mac users might like to try `homebrew`.

On Windows machines, Microsoft MPI[2] is free.

- Based on MPICH.

[1]Note the linux command "`module avail`" shows what modules are installed.

[2]`https://docs.microsoft.com/en-us/message-passing-interface/microsoft-mpi`

Building an MPI program

Need to use a **special compiler** for MPI programs:

- Standard installation includes `mpicc`, `mpic++`, `mpifort`.
- Essentially a wrapper around a standard compiler.
- **Passes command line arguments to the C compiler**.

For example, to compile a file `helloWorld.c`:

```
mpicc -Wall -o helloWorld helloWorld.c
```

- Will generate the executable `helloWorld`.
- All warnings on ('`-Wall`').
- Add *e.g.* `-lm` for the maths library.

Executing an MPI program

Also need a special **launcher** to execute an MPI program[1].

For multiple processes all on the **same local machine**:

```
mpiexec -n 2 ./helloWorld
```

- Creates 2 processes running the **same** program.
- Trying to launch more processes than cores *may* lead to an error (*'too many slots'*)[2].
- `mpirun` is the same/very similar to `mpiexec`.

Best to develop/debug code on a single machine (e.g. login node of `cloud-hpc1.leeds.ac.uk`), then run on multiple cores in batch mode for *e.g.* timing runs.

[1] Executing as usual ('`./helloWorld`') will launch *one* process, *i.e.* serial.
[2] With OpenMPI, can override with the argument `-oversubscribe`.

Launching via the batch queue

The system cloud-hpc1.leeds.ac.uk has been set up to allow access to two 8-core nodes via slurm.

- Follow a similar approach to running batch jobs for OpenMP:
  - sbatch script.sh
- Below is an example script...

```
#!/bin/bash
#Request a single node, and 8 cores (adjust as necessary)
#SBATCH -N1 -n8

module add mpi/openmpi3-x86_64
mpiexec -n 8 ./helloWorld
```

A 'Hello World' example

```
#include "stdio.h"
#include "stdlib.h"
#include "mpi.h"                    // Need to include mpi.h

int main( int argc, char **argv )
{
  int numprocs, rank;

  MPI_Init( &argc, &argv );
  MPI_Comm_size( MPI_COMM_WORLD, &numprocs );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );

  printf( "Process %d of %d.\n", rank, numprocs );

  MPI_Finalize();
  return EXIT_SUCCESS;
}
```

Initialising and finalising

The first MPI call **must** be `MPI_Init()`:

- Pass command line arguments `argc` and `argv`.
- Will remove arguments relevant to MPI.
- Specific to the implementation and not of interest here.

The final MPI call **must** be `MPI_Finalize()`:

- Note the US spelling: *finalize* not *finalise*.

Any MPI calls before `MPI_Init()` or after `MPI_Finalize()` will result in a runtime error.

Number of processes and rank

`MPI_Comm_size(MPI_COMM_WORLD,&numprocs)`

- Sets `numprocs` to the **total** number of processes.
- Should return the '`-n`' argument in `mpiexec`.
- Similar to `omp_max_thread_num()`.

`MPI_Comm_rank(MPI_COMM_WORLD,&rank)`

- Sets `rank` to the process number, known as the **rank** in MPI.
- Ranges from `0` to `numprocs-1` inclusive.
- Similar to `omp_get_thread_num()`.

Communicators

For our purposes, whenever you see an MPI call with the argument **communicator**, just use `MPI_COMM_WORLD`:

- Means 'all processes available to us.'
- The **only** communicator we consider in this course.

In general, communicators allow processes to be **partitioned**.

- *e.g.* when developing a parallel library, don't want the library processes to accidentally communicate with application processes.
- An advanced feature we won't consider.

Summary and next lecture

Today we have started looking at **distributed memory parallelism**:

- Realised in **clusters** and **supercomputers**.
- Requires **communication** between **nodes**.
- For HPC, use **MPI = Message Passing Interface.**
- Seen how to build and execute a 'Hello World' program.

Next time we will see how MPI supports communication between processes, and use this to solve real problems.