Introduction to Parallel Computing

Iain Miller

Iain.miller@ecmwf.int

© ECMWF March 14, 2022

Overview

- What is Parallel Computing
- Supercomputer Building Blocks
- Shared Memory Parallelism
- Distributed Memory Parallelism
- Hybrid Parallelism
- Scaling Limitations
- Future Challenges
- Further Reading

What is Parallel Computing

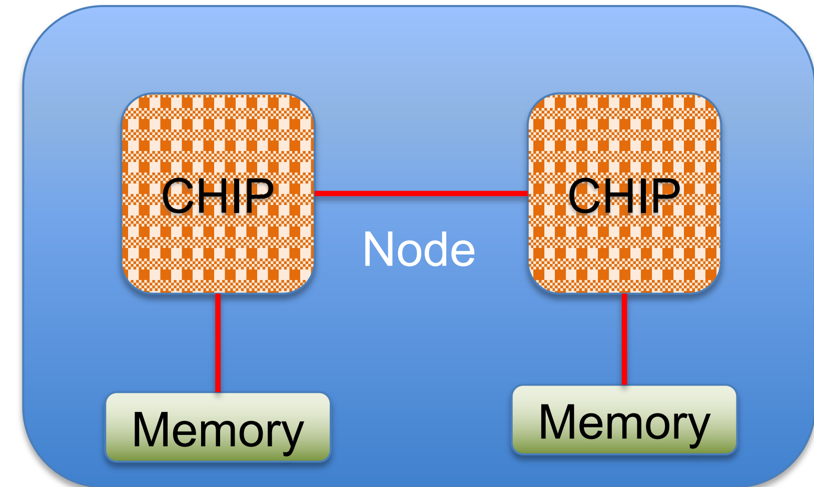
The simultaneous use of more than one processor or computer to solve a problem

Why do we need Parallel Computing

- Generally, it is either:
  - Serial Computing is too slow
  - Need more memory than is accessible by a single processor

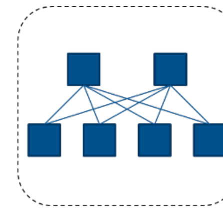
Supercomputer Building Blocks

- Smallest building block is a node
  - Each node will have a number of sockets
  - Each socket will have a processor chip
  - Each processor chip will have a number of cores
  - Each core may or may not have a number of execution hardware threads
  - Each thread will have a vector width
- It is common for the lowest execution unit to be called a "Processing Element"

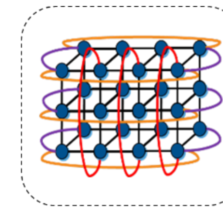


Supercomputing Building Blocks

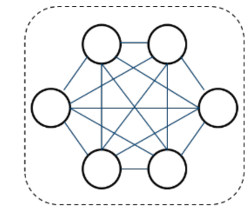
- Nodes will be linked together with a interconnect
- Various Network Topologies can be used
  - Fat Tree is commonly used
    - Can be blocking or non-blocking, which determines the total available bandwidth available
  - Dragonfly is becoming more popular
    - Uses less cables, particularly on long links
    - But less connection between groups



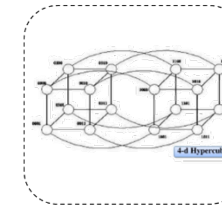
Fat Tree



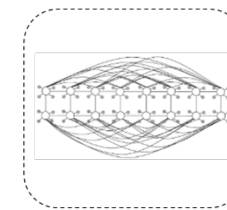
Torus



Dragonfly



Hypercube



HyperX

Image from https://www.hpcwire.com/2019/07/15/super-connecting-the-supercomputers-innovations-through-network-topologies/

Supercomputer Building Blocks

- Traditionally a supercomputers “compute power” is expressed in it’s Flop rate or Flops
  - 1 Flops = 1 double precision floating-point operation per second
  - Double precision uses 64-bits to store a value
  - **THEORETICAL** peak Flops of a supercomputer is Number of Floating-point operations per core per cycle multiplied by the number of cycles per second multiplied by the number of cores
- The world’s top supercomputers are ranked in the Top 500 (www.top500.org), which measures the **SUSTAINED** peak Flops managed by the LINPACK benchmark
  - Solves a dense system of linear equations using LU factorization with partial pivoting
  - Scales with the size of supercomputer and memory available
  - Not representative of most scientific codes
  - No 1 machine is Fugaku in Japan – sustained rate of 442PFlops

Supercomputing building blocks

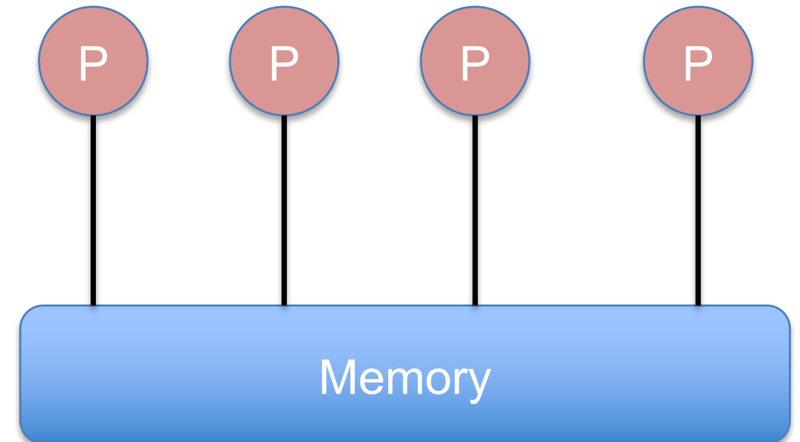
- Most codes will either be compute or memory bound:
  - Compute bound codes are limited by the clock speed of the processor
  - Memory bound codes are limited by the memory access bandwidth
  - Not consistent within the code with some routines being one or the other

Poll – ECMWF Cluster Theoretical Peak

- There are four new clusters being installed into the datacentre in Bologna
  - Each cluster has 1920 nodes
    - Each node has 2 AMD Rome processors
      - Each processor has:
        - » 64 cores
          - Each core can do 4 Floating-point operations per cycle
        - » 2.25GHz clock speed
        - » 256-bit wide vector registers
  - What is the Theorectical Maximum Flop rate for a cluster in PetaFlops?
    - 1.1
    - 2.2
    - 4.4
    - 8.8

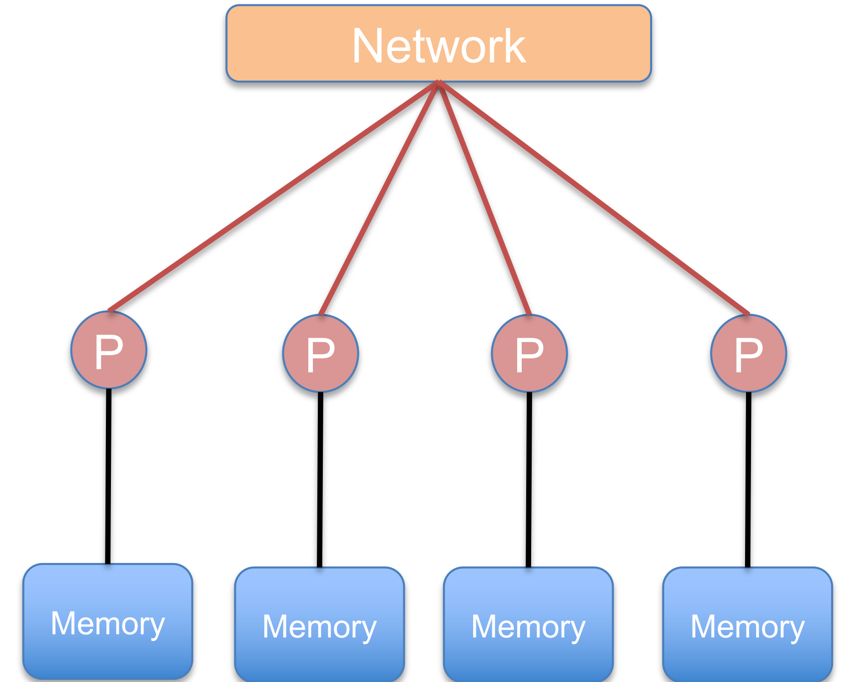
Shared Memory Parallelism

- All processors can see all the memory
  - Bandwidth may not be equal
- Entire domain within the memory
- Execution unit is commonly called a thread
- Need to explicitly protect some variables from being overwritten by other threads
- Most common programming paradigm is via OpenMP
  - Pragma based programming
  - Support is via the compiler
  - Control via environment variables



Distributed Memory Parallelism

- Each processor can only see its own memory
- Domain decomposed across the different memories
- Execution unit is commonly called a Rank
- Data exchange has to be explicitly coded and managed through external library
  - Often needed to store and transfer Halo information
- The most common programming paradigm is using MPI
  - Standardised API
  - Several major implementation libraries
  - Subtle differences between them
  - Control through job launchers



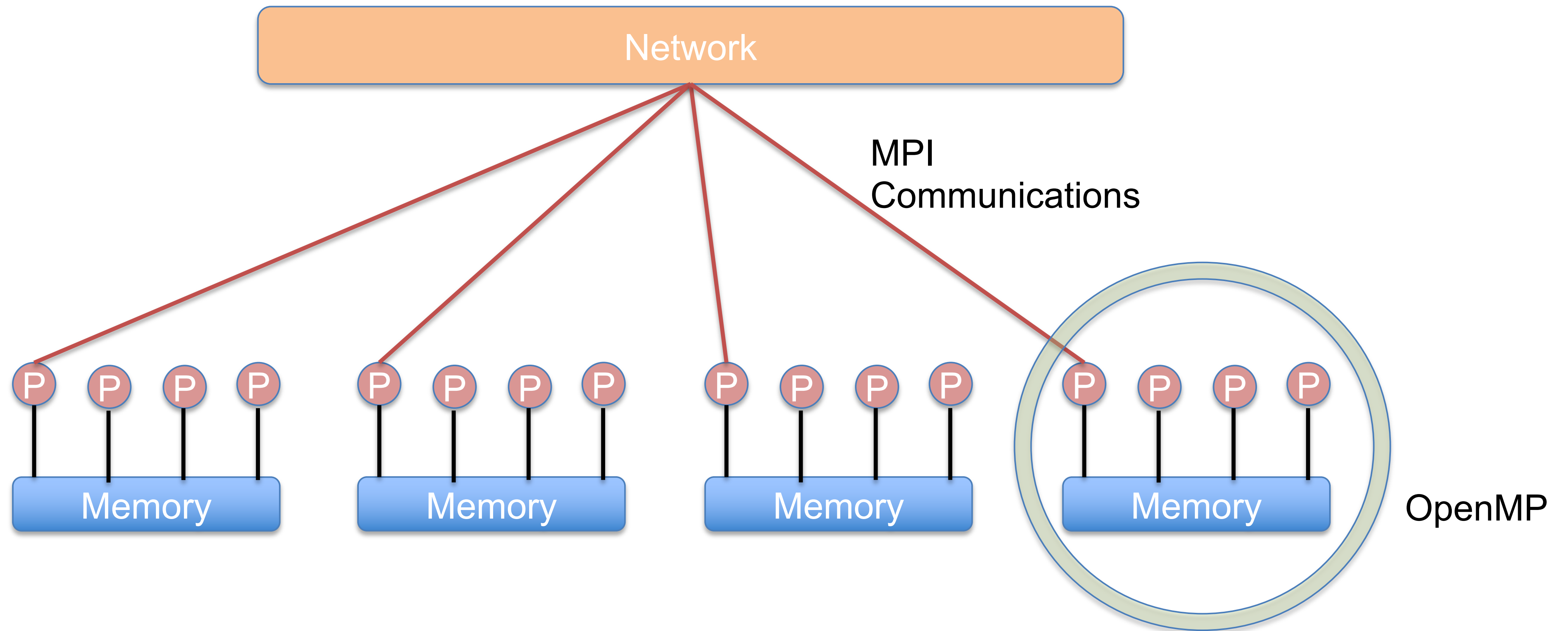
Poll – Whether to use MPI or OpenMP

- A simulation running in a serial code takes too long to complete and you want to parallelise it. The problem comfortably fits into the memory of a single node. What should you use for parallelisation?
  - Shared Memory/OpenMP
  - Distributed Memory/MPI
  - Hybrid methods
  - It depends

Hybrid Parallelism

- Most supercomputers now consist of a series of nodes linked together by a network
  - Each node then consists of a number of processors with access to one or more banks of memory
- It is possible to run MPI across all the available processors
  - But processors compete for access to memory and network
  - Halo exchange becomes expensive
- Therefore hybrid methods have been developed that
  - decompose the domain across memory regions on the nodes
  - Intra-domain calculations use shared memory paradigms
  - Inter-domain exchanges use distributed memory paradigms

Hybrid Parallelism



Scaling Limitations

- There are two types of scaling
  - Weak Scaling
    - The amount of work per processor remains the same, i.e. Problem size is a factor of the number of processors
    - Expectation is that the amount of runtime required stays constant as the number of processors increases
  - Strong Scaling
    - The overall size of the problem remains the same but the work per processor reduces as the number of processors increases
    - Expectation is that the runtime decreases in proportion to the number of processors
- However, neither expectation is realized
- Speedup is limited by Amdahl's Law
  - The theoretical maximum is inversely proportional to portion of the code that cannot be parallelised

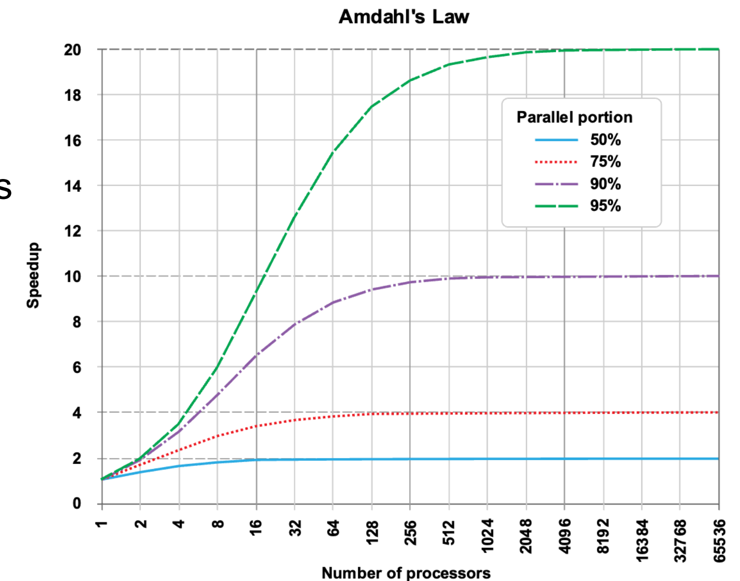


Image from Wikipedia under creative commons
https://en.wikipedia.org/wiki/Amdahl%27s_law

Scaling Limitations

- Some factors that affect scaling:
  - Serial portions of code
  - Load imbalance
    - Not all processors are doing the same amount of work during the same period of time
  - Synchronisation
  - Limits in network
  - Algorithmic limitations
  - Running out of parallelism

Poll – Theorectical Scaling limits

- A serial code takes 1000s to run
  - When parallelised there is parts of the code that still have to be run serially on each rank that takes 100s
  - If perfect parallelisation can be achieved in the non-serial parts, what is the maximum speedup that can be reached?
    - 2x
    - 10x
    - 100x
    - 500x

Future Challenges

- Data locality
  - Increasing levels of memory hierarchy, including in NUMA and cache regions
- Accelerated computing
  - Increases in computing coming more and more from attached “accelerator” such as General Purpose GPUs
    - Need to change algorithms to expose more parallelism, may need to change programming language and paradigms too
- Increasing levels of parallelism
- Bottom of chain for hardware design
- Hardware resilience
  - Fault-tolerant algorithms
- Power requirements
- Bit-reproducibility

Further Reading

- OpenMP Standards Community: https://www.openmp.org/
- Basic OpenMP Tutorial: https://hpc.llnl.gov/openmp-tutorial
- MPI Standards Website: https://www.mpi-forum.org/docs/
- Basic MPI Tutorial: https://mpitutorial.com/tutorials/
- Online taught course from Jülich: http://www.cpex-lab.org/SharedDocs/Termine/IAS/JSC/EN/courses/2022/mpi-intro-2022-2.html

OpenMP Example

```
!$OMP PARALLEL DO SCHEDULE(STATIC,1)&
!$OMP& PRIVATE(JMLOCF,IM,ISTA,IEND)
      DO JMLOCF=NPTRMF(MYSETN),NPTRMF(MYSETN+1)-1
        IM=MYMS(JMLOCF)
        ISTA=NSPSTAF(IM)
        IEND=ISTA+2*(NSMAX+1-IM)-1
        CALL SPCSI(CDCONF,IM,ISTA,IEND,LLONEM,ISPEC2V,&
         &ZSPVORG,ZSPDIVG,ZSPTG,ZSPSPG)
      ENDDO
!$OMP END PARALLEL DO
```

MPI Examples

```
int ping_pong_count = 0;
int partner_rank = (world_rank + 1) % 2;

while (ping_pong_count < PING_PONG_LIMIT) {
    if (world_rank == ping_pong_count % 2) {
        ping_pong_count++;

        MPI_Send(&ping_pong_count, 1, MPI_INT,
        partner_rank, 0, MPI_COMM_WORLD);

        printf("%d sent and incremented
        ping_pong_count " "%d to %d\n",
        world_rank, ping_pong_count,
        partner_rank);
    } else {
        MPI_Recv(&ping_pong_count, 1, MPI_INT,
        partner_rank, 0, MPI_COMM_WORLD,
        MPI_STATUS_IGNORE);

        printf("%d received ping_pong_count %d
        from %d\n", world_rank, ping_pong_count,
        partner_rank);
    }
}
```

```
if (world_rank == 0) {
    rand_nums = create_rand_nums(elements_per_proc
    * world_size);
}

float *sub_rand_nums = malloc(sizeof(float) *
elements_per_proc);

MPI_Scatter(rand_nums, elements_per_proc, MPI_FLOAT,
sub_rand_nums, elements_per_proc, MPI_FLOAT, 0,
MPI_COMM_WORLD);

float sub_avg = compute_avg(sub_rand_nums,
elements_per_proc);

float *sub_avgs = NULL;
if (world_rank == 0) {
    sub_avgs = malloc(sizeof(float) * world_size);
}

MPI_Gather(&sub_avg, 1, MPI_FLOAT, sub_avgs, 1,
MPI_FLOAT, 0, MPI_COMM_WORLD);

if (world_rank == 0) {
    float avg = compute_avg(sub_avgs, world_size);
}
```