Introduction to Parallel Computing

Iain Miller, Lucian Anton

Iain.miller@ecmwf.int

ECMWF

Overview

- What is Parallel Computing
- Building a Supercomputer
- Parallel Programming Paradigms
- Scaling Limitations
- Future Challenges
- Further Reading

What is Parallel Computing

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

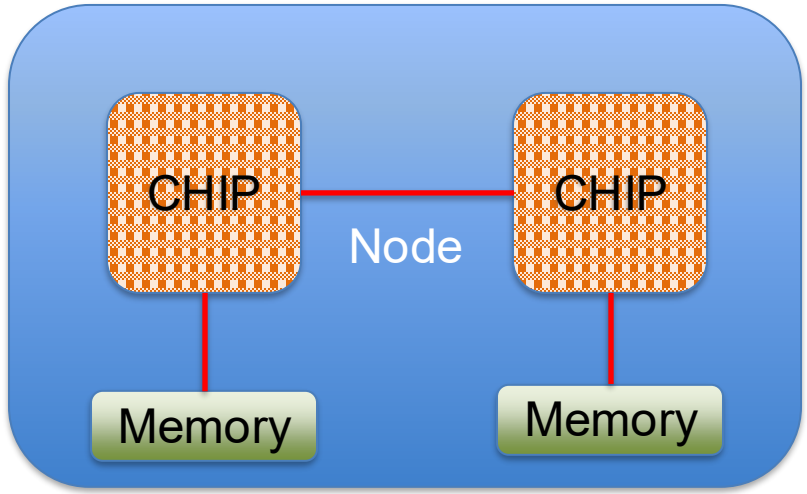
Why do we need Parallel Computing

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

Building a Supercomputer

Supercomputer Building Blocks

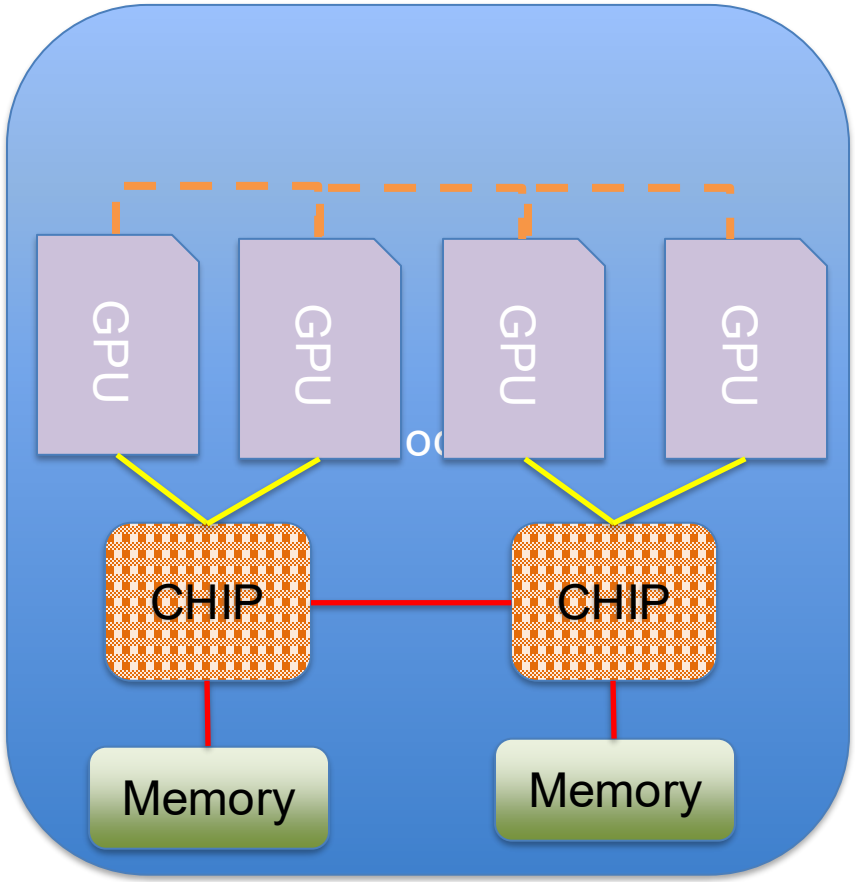
- Smallest building block is a node
  - Each node will have a number of sockets
  - Each socket will have a processor chip
  - Each processor chip will have a number of cores
  - Each core may or may not have a number of execution hardware threads
  - Each thread will have a vector width
- It is common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



- Memory is attached in channels to each socket.
  - Slower access times than on chip memory (cache)
  - Usually accessible by all sockets
  - Will have variable access times depending on core location

Supercomputer Building Blocks : Accelerators

- GPUs are the mostly used accelerators
  - Symmetric multiprocessor
    - Simple cores that are grouped together to execute one instruction
    - Large number of registers
    - Cache and main memories
    - HBM
    - Tensor cores and reduced precision floating point representations
  - The main processor dispatches kernels to the accelerator
    - Kernels are defined in the source code using language extensions (CUDA, OpenACC, OpenMP 4+,…)
    - Data transfers between the CPU and accelerator memories might need application management



Supercomputing Building Blocks

- Nodes will be linked together with an 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

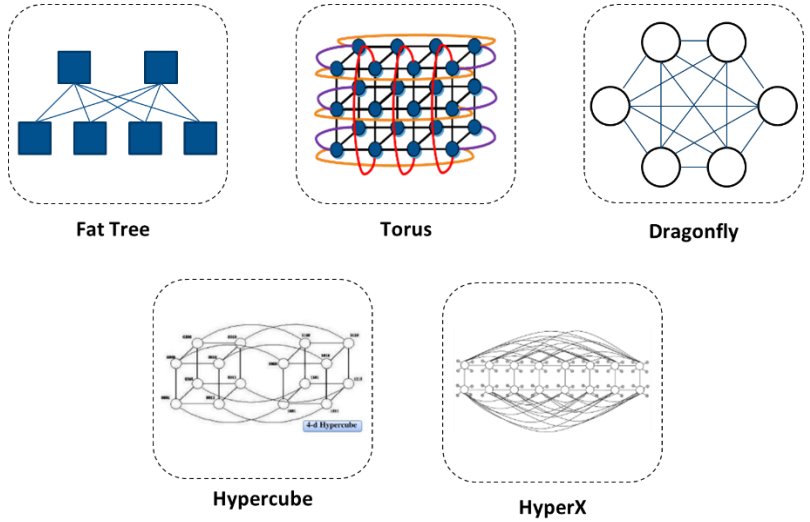
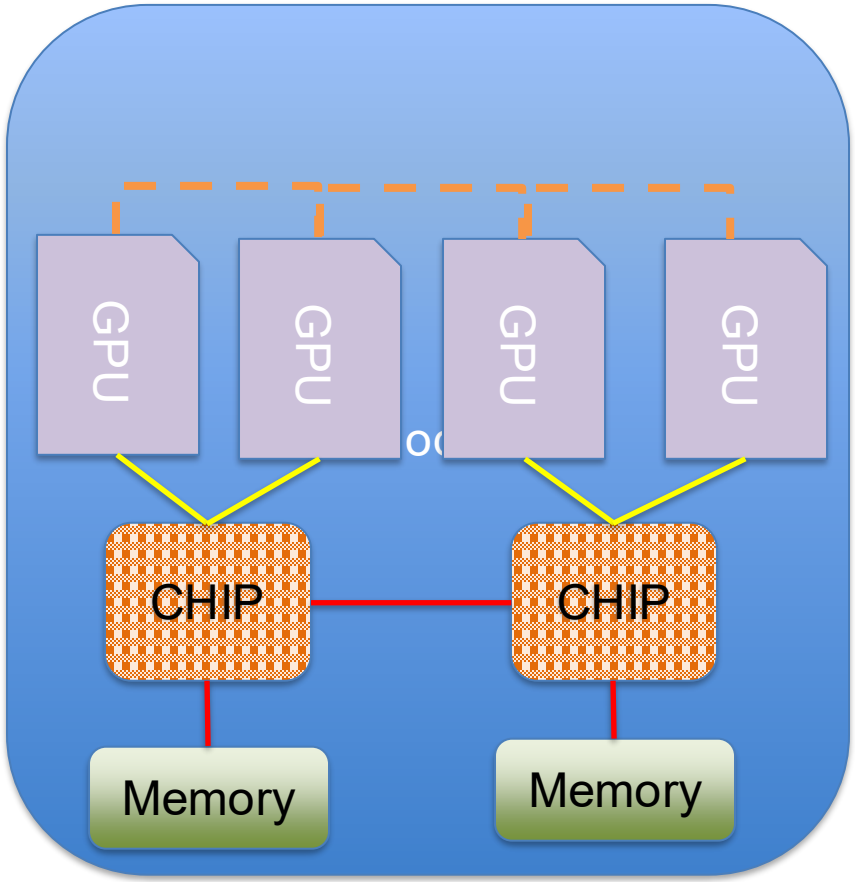


Image from https://www.hpcwire.com/2019/07/15/super-connecting-the-supercomputers-innovations-through-network-topologies/

Supercomputer Building Blocks : Accelerators

- GPUs are the mostly used accelerators
  - Symmetric multiprocessor
    - Simple cores that are grouped in wraps to execute one instruction
    - Large number of registers
    - Cache and main memories
    - HBM
    - Tensor cores and reduced precision floating point representations
  - The main processor dispatches kernels to the accelerator
    - Kernels are defined in the source code using language extensions (CUDA, OpenACC, OpenMP 4+,…)
    - Data transfers between the CPU and accelerator memories might need application management



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 El Capitan at LLNL in USA – sustained rate of 1.742EFlops
    - HPCG 17.41PFlops (No. 1 in world, 0.6% Peak – No 2, Fugaku in Japan has 3.0% Peak)
    - No 1 machine in Europe is JUPITER Booster at Jülich in Germany – sustained rate of 793.40PFlops
      - HPCG Unknown, No 1 in Europe is LUMI in Finland with 4.587PFlops (0.9% of Peak)
    - Represents 63% and 85% efficiency

Supercomputing in Perspective

- If you compare the **SUSTAINED** computing power of El Capitan to the "Human Computer"
  - If every single one of the 8 billion people on Earth did one calculation per second it would take
    - Over 6 years and 11 months to exceed El Capitan in 1 second
    - Nearly 3 year and 1 month and 22 days to exceed JUPITER Booster in 1 second
    - 38 days 14 hours to exceed ECMWF 4 clusters in 1 second

Code performance main features

- Most codes (or subcomponents) will either be compute or memory bound:
  - Compute bound codes are limited by the clock speed of the processor and hardware features: number of FP units, vector width.
  - Memory bound codes are limited by the memory access bandwidth (β)
  - Operational Intensity is the amount of processing work completed per byte of memory accesses (**I**)

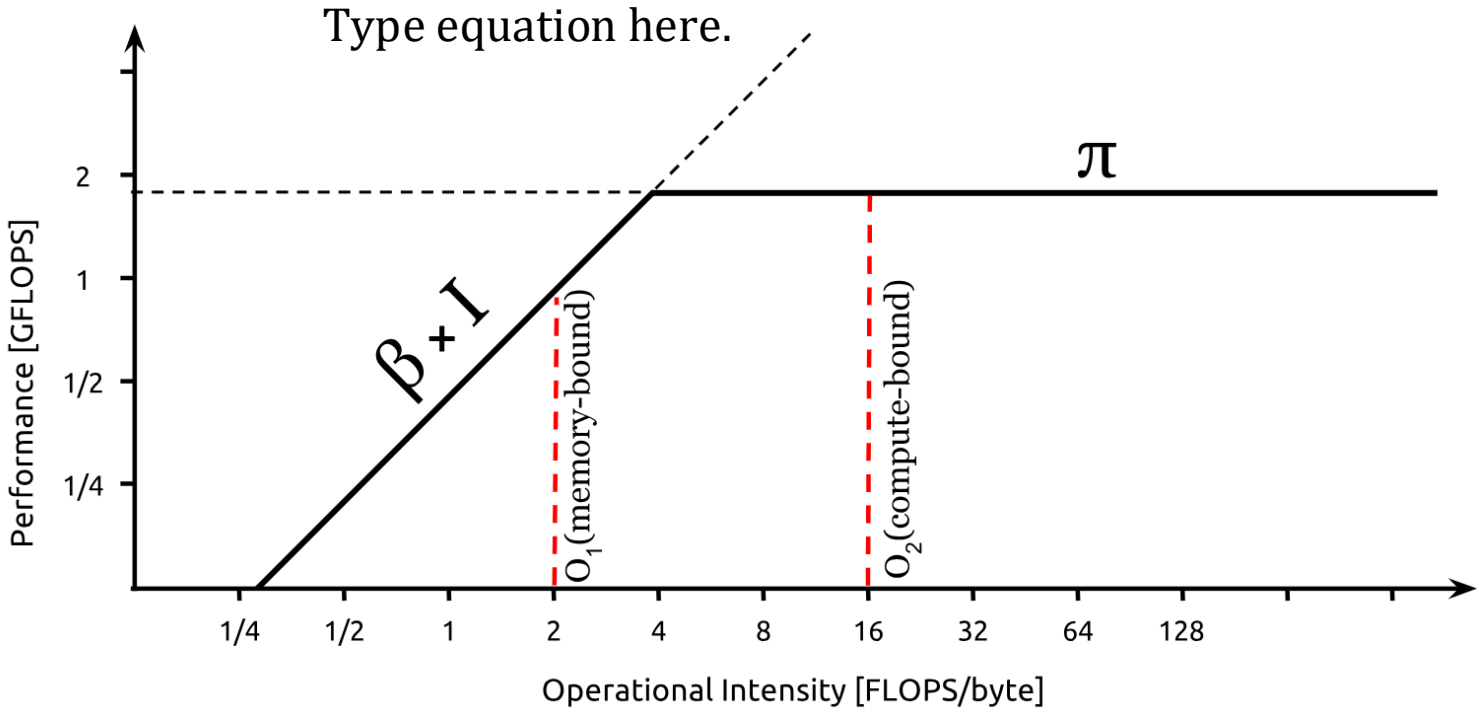
C(1:N) = A(1:N) * B(1:N)

1 FP / 1 store, 2 load

R = MATMUL (T,S)

$$r_{ij} = \sum_k t_{ik} s_{kj}$$

N^3 FP / [3 N^2 load/store]



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

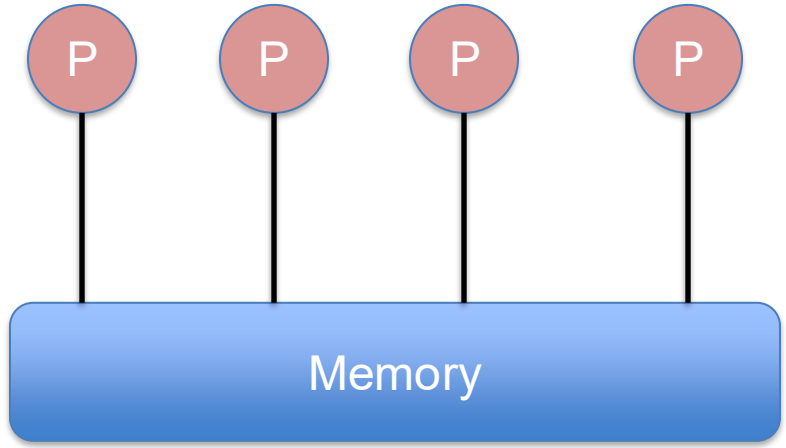
Poll – ECMWF Cluster Theoretical Peak

- 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**
      - **4 instructions per register * 4 registers per core * 128 cores per node * 1920 nodes * 2.25G**

Parallel Programming Paradigms

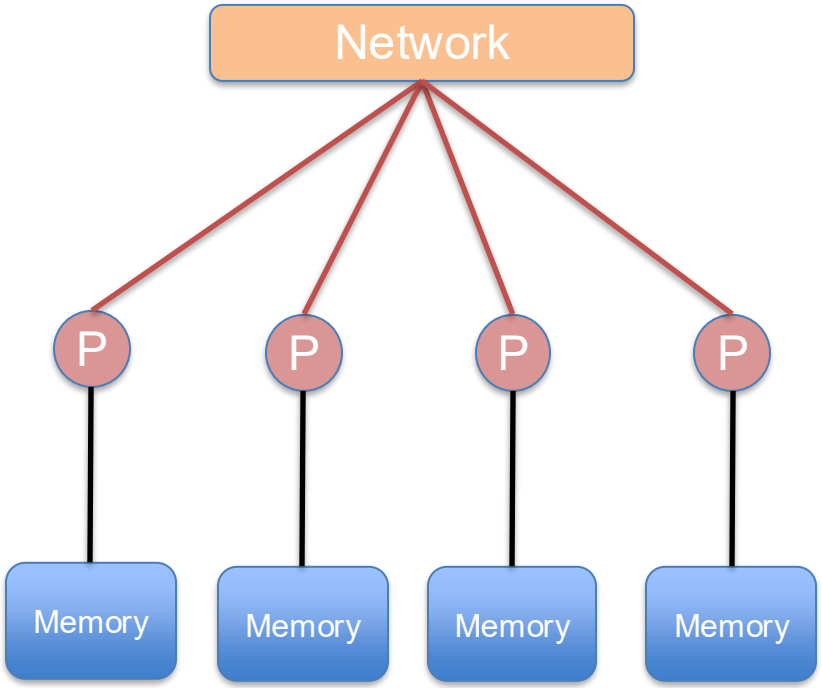
Shared Memory Parallelism

- 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

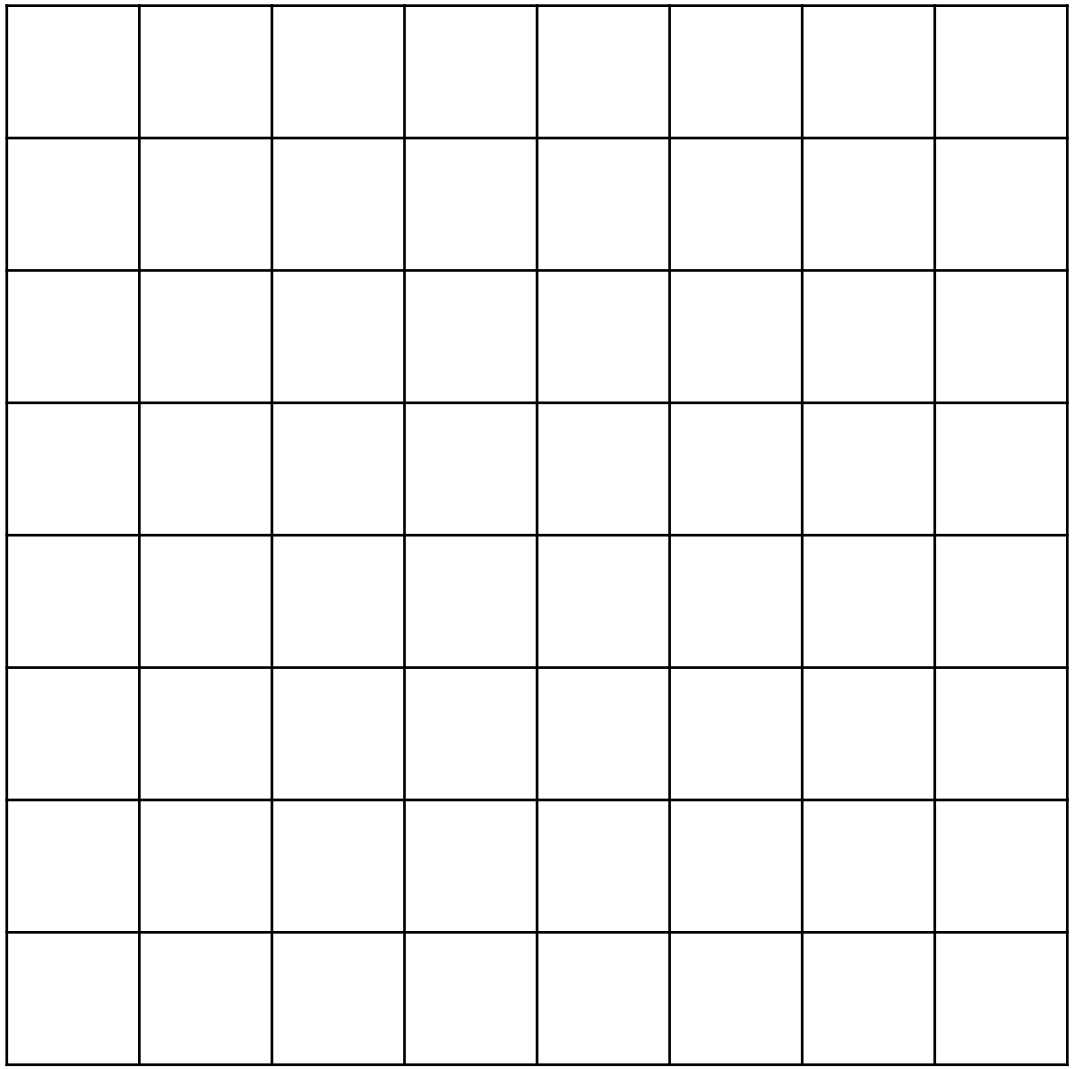


Distributed Memory Parallelism

- 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

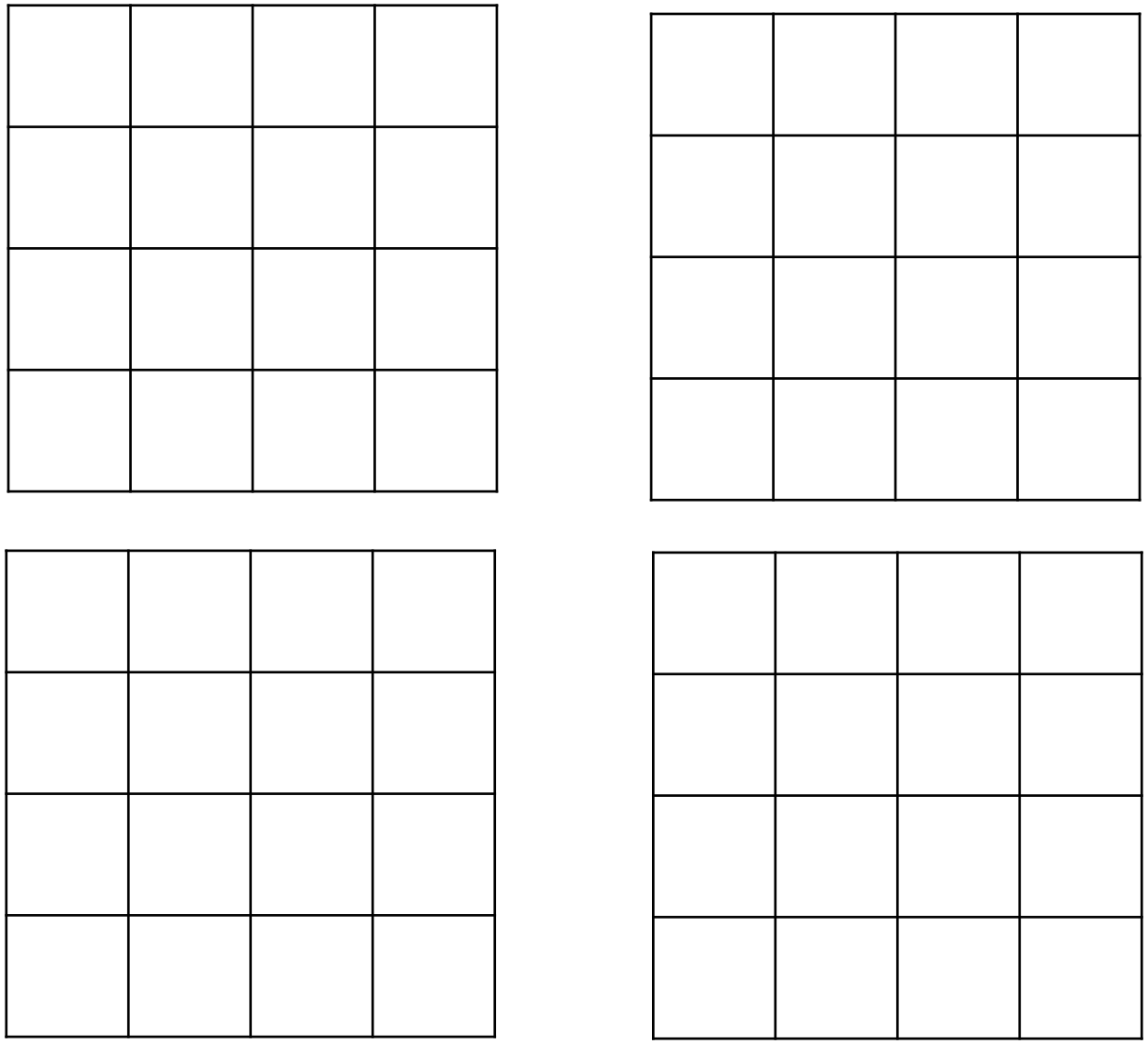


Domain Splitting



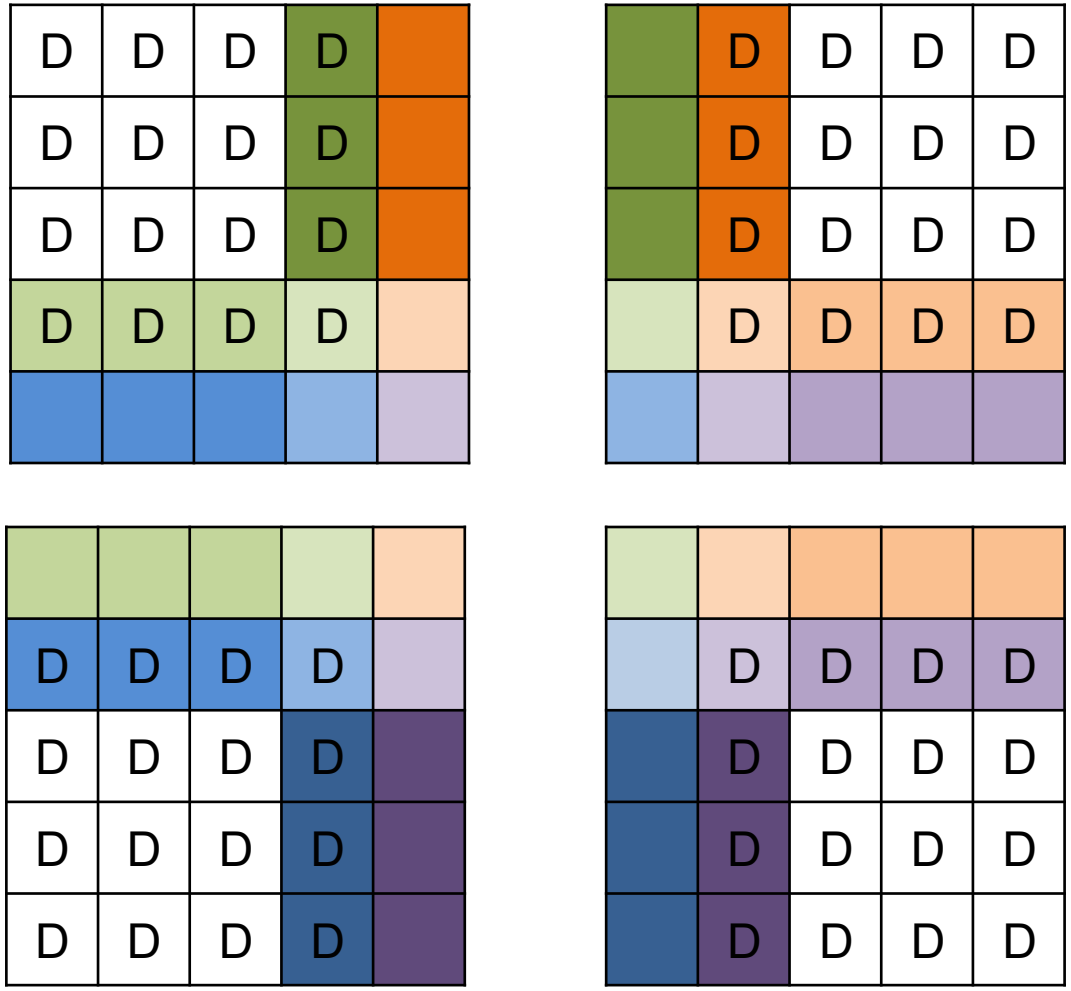
Imagine a domain grid like this

Domain Splitting – 4 Processors



Grid is split evenly over
4 processors

Domain Splitting - Haloes



- Cells labelled “D” are actual Domain Data for that processor
  - what the processor applies its algorithms to
- The different colours indicate what data needs to be shared with neighbouring processors
  - May be needed for algorithms to work
- After each step data in the “haloes” needs to be exchanged to update each processor on changes calculated.

Domain partition and comms for 3D FFT

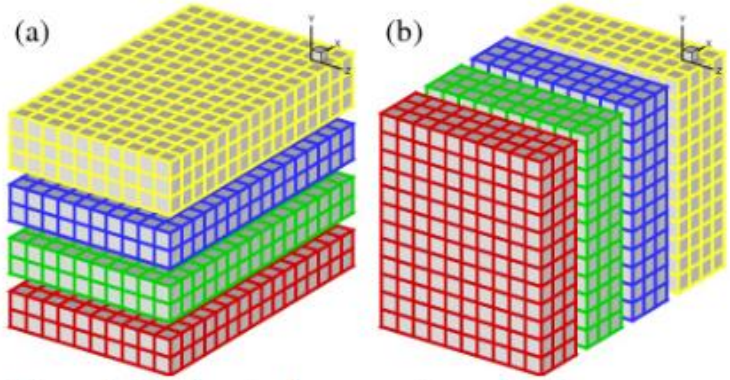


Figure 1: 1D domain decomposition using 4 processors: (a) decomposed in Y; (b) decomposed in X.

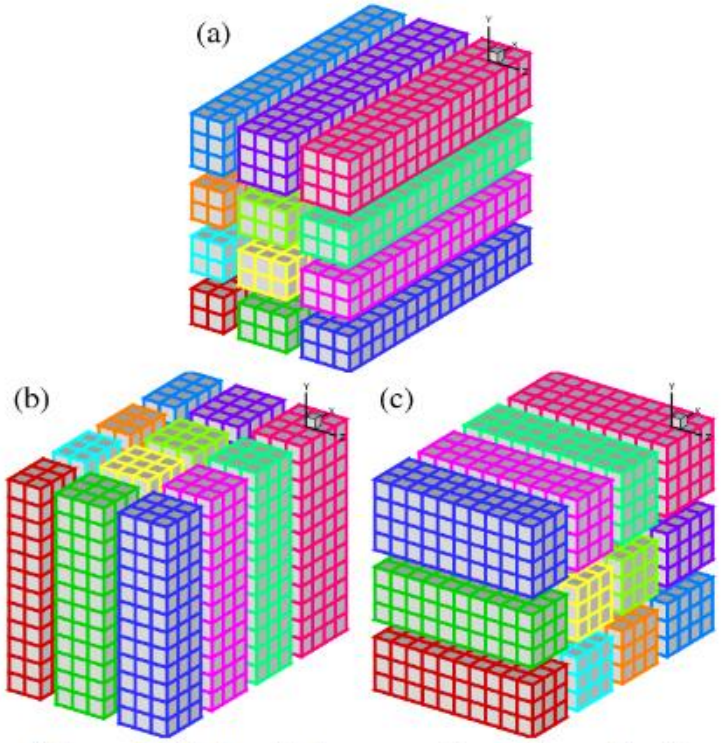


Figure 2: 2D domain decomposition using a 4 by 3 processor grid.

2DECOMP&FFT - A Highly Scalable 2D Decomposition Library and FFT Interface

Ning Li, *the Numerical Algorithms Group (NAG) and*
Sylvain Laizet, *Imperial College London*

Exercise – Parallelism models

- Split the class into two groups
  - 1 will be the Shared Memory group
  - 1 will be the Distributed Memory group

Shared Memory Group rules:

- Everyone can access the ”data” in the envelope
- Cannot pass data to other members
- Take two pieces of data
  - Add them together
  - write the result on one piece paper
  - return to envelope
  - Throw second piece of data away
- Repeat until there is only one piece of data left in the envelope
- This is your answer

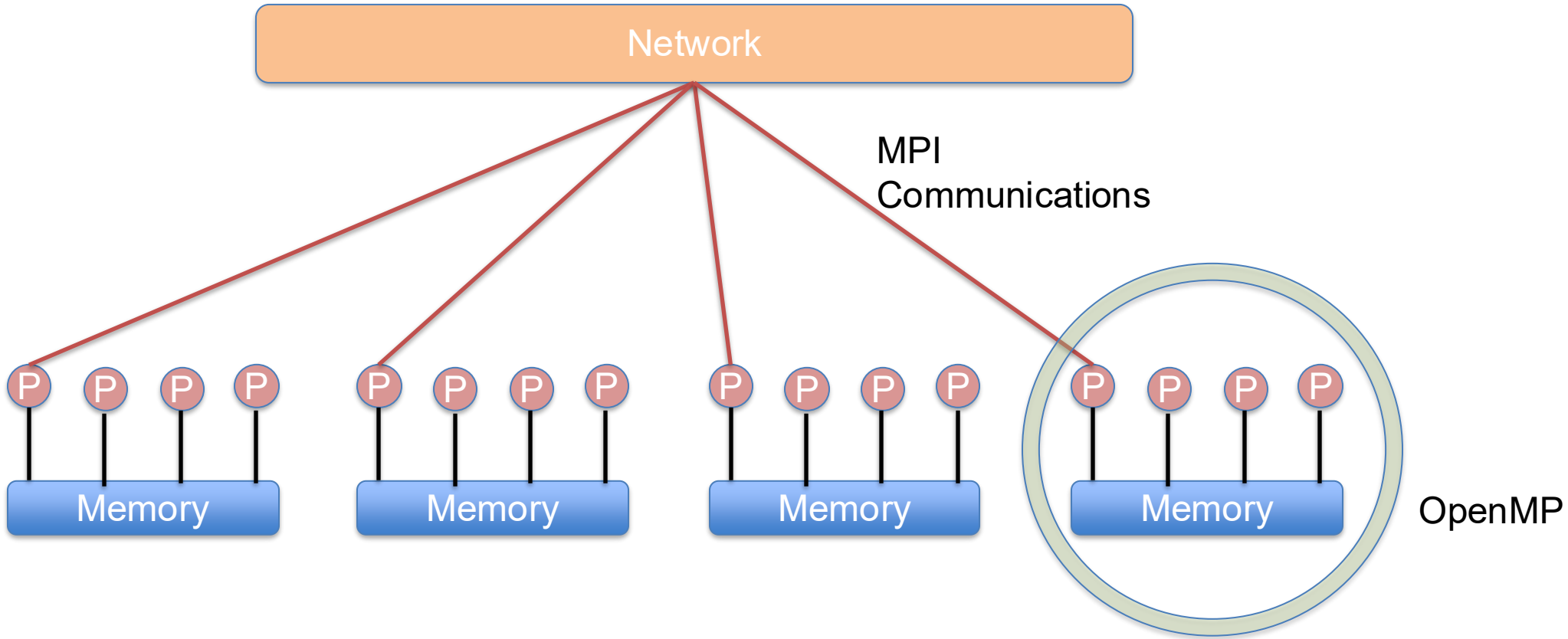
Distributed Memory Group Rules

- Choose someone to be the ”Master”
- Only the Master can access the ”data” in the envelope
- Data can be shared between members
- Can only hold two pieces of “data” together at a time
  - Add them together
  - Write result on one piece of paper
  - Throw second piece away
  - Either pass result to another ready member or receive new data
  - Maintain ”data” limits at all times
- When one result is left = answer

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



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

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**

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 parallelized
  - $T = T_{serial} + \frac{T_{par}}{N_{proc}} + T_{comm}(N_{proc}, \dots)$

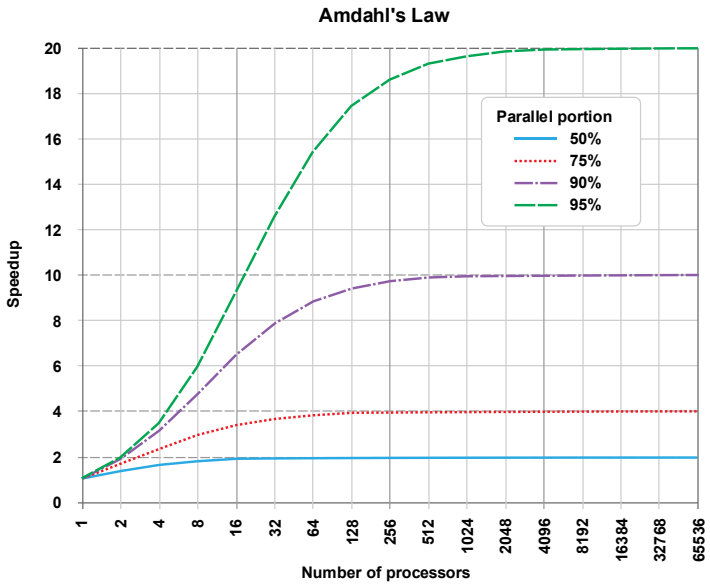


Image from Wikipedia under creative commons
https://en.wikipedia.org/wiki/Amdahl%27s_law

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

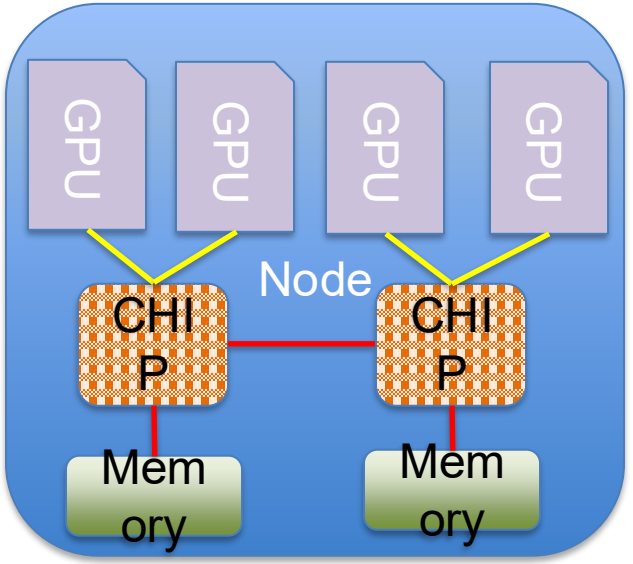
- A serial code takes 1000s to run
  - When parallelised there is parts of the code that still have to be run serially on each rank that takes 100s
  - If perfect parallelisation can be achieved in the non-serial parts, what is the maximum speedup that can be reached?
    - 2x
    - 10x
    - 100x
 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

Poll – Theorectical Scaling limits

- A serial code takes 1000s to run
  - When parallelised there are parts of the code that still run serially on each rank that take 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

- Heterogeneity in many ways
  - Processor – complex compute modes with scalar and vector, prefetch, etc.
  - Many (but not all) include separate accelerators (GPUs and others)
  - Memory – Cache was bad enough; now HBM, other
  - I/O – Burst buffers (often violating POSIX semantics), on node, central, remote (cloud)
- For a high level of performance system programming elements will be needed in the app code to guide code generation.
- Data management
- Results validation (Bit-reproducibility)
- In NWP what is the right system design to balance DL with physics based models.



Further Reading

- OpenMP Standards Community: https://www.openmp.org/
- Basic OpenMP Tutorial: https://hpc-tutorials.llnl.gov/openmp/
- MPI Standards Website: https://www.mpi-forum.org/docs/
- Basic MPI Tutorial: https://mpitutorial.com/tutorials/
- Jülich do online and in-person courses, next years to be posted: https://www.fz-juelich.de/en/jsc/news/events/training-courses

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);
}
```


Examples from www.mpitutorial.com