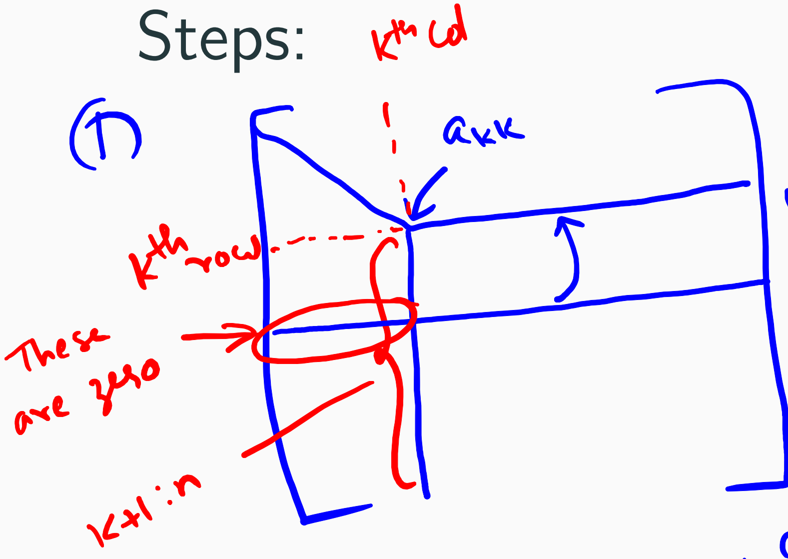
Practical Implementation

1. It is enough to update $A(k+1:n, k+1:n)$
2. We can overwrite $A(k+1:n, k)$ with $L(k+1:n, k)$

Steps:

(1)



When subtracting j^{th} row from k^{th} row we know that $A(j, 1:k) = 0$ so we only need to update $A(j, k+1:n)$

(2) Since we know that entries below the k^{th} col $A(k+1:n, k)$ is going to be zero, we use this to store the multipliers: $L(k+1:n, k)$

A overwritten by U and L.

LU Algorithm

compute multipliers

for $k = 1 : n-1$ **do**

(*) $A(k+1:n, k) = A(k+1:n, k)/A(k,k)$

  for $i = k+1 : n$ **do**

    for $j = k+1 : n$ **do**

      $A(i,j) = A(i,j) - A(i,k)A(k,j)$

    end for

  end for

end for

update the rows

These are multipliers stored in (*) above

Vectorize j th loop.

LU Algorithm: After Vectorization of jth loop

for $k = 1 : n-1$ **do**

  $A(k+1:n, k) = A(k+1:n, k)/A(k,k)$

  for $i = k+1 : n$ **do**

    $\underbrace{A(i, k+1:n)}_{R_i} = \underbrace{A(i, k+1:n)}_{R_i} - \underbrace{A(i,k)}_{m_i}\underbrace{A(k, k+1:n)}_{R_k}$

  end for

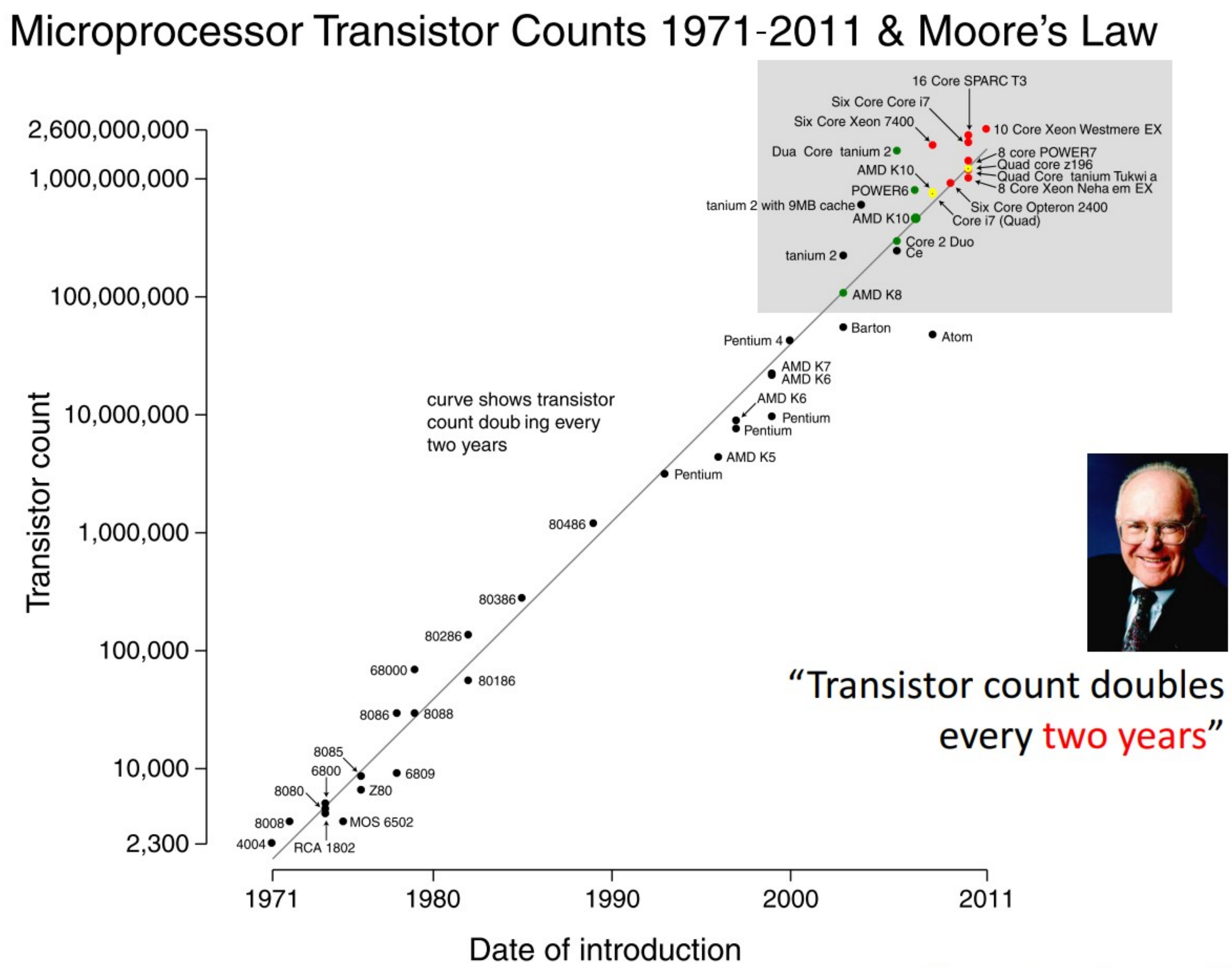
end for

$R_i \leftarrow R_i - m_i R_k$

After vectorizing j-loop, we see row operations!

Parallel Computing: MPI

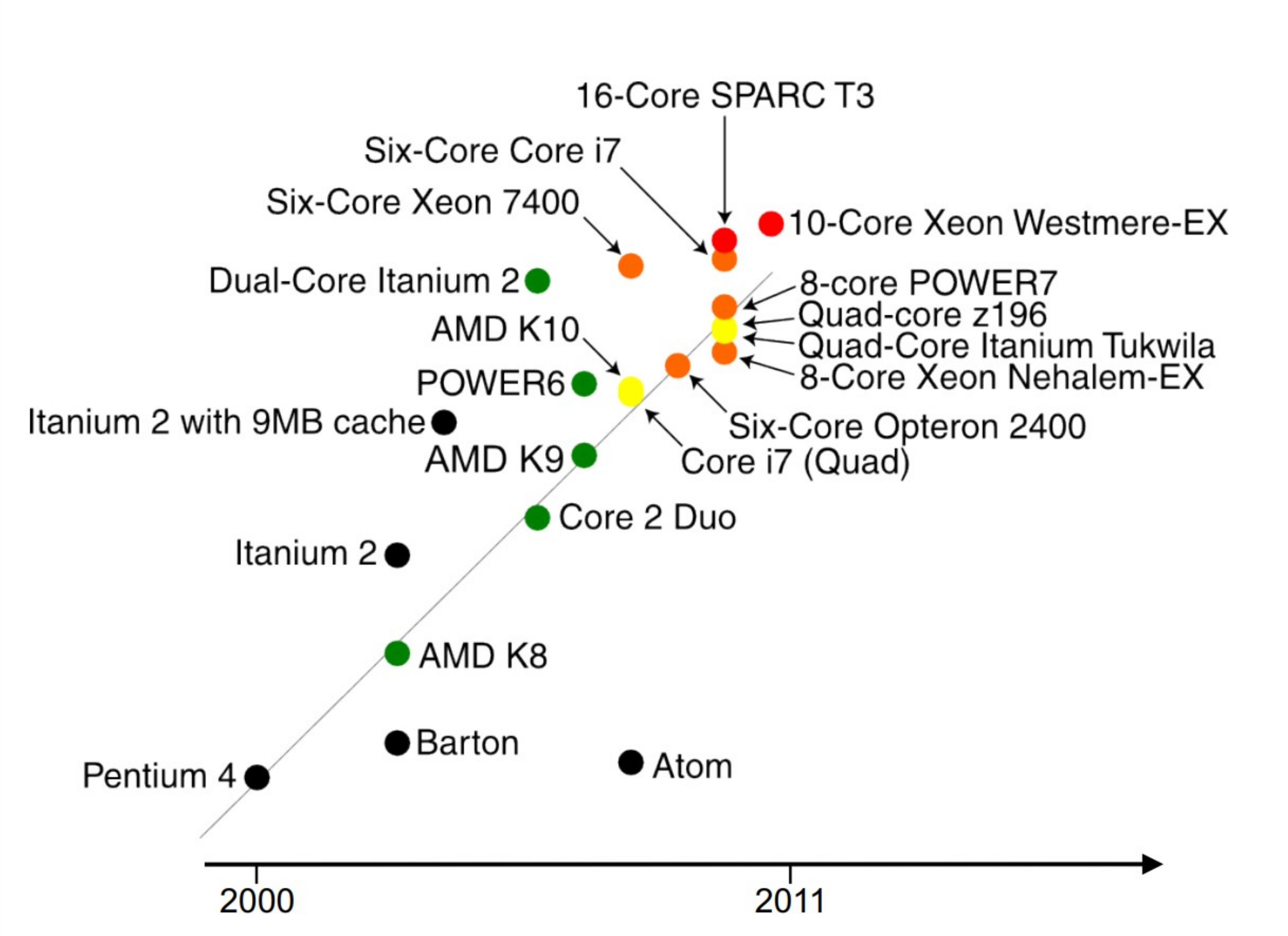
Moore's Law



"Transistor count doubles every two years"

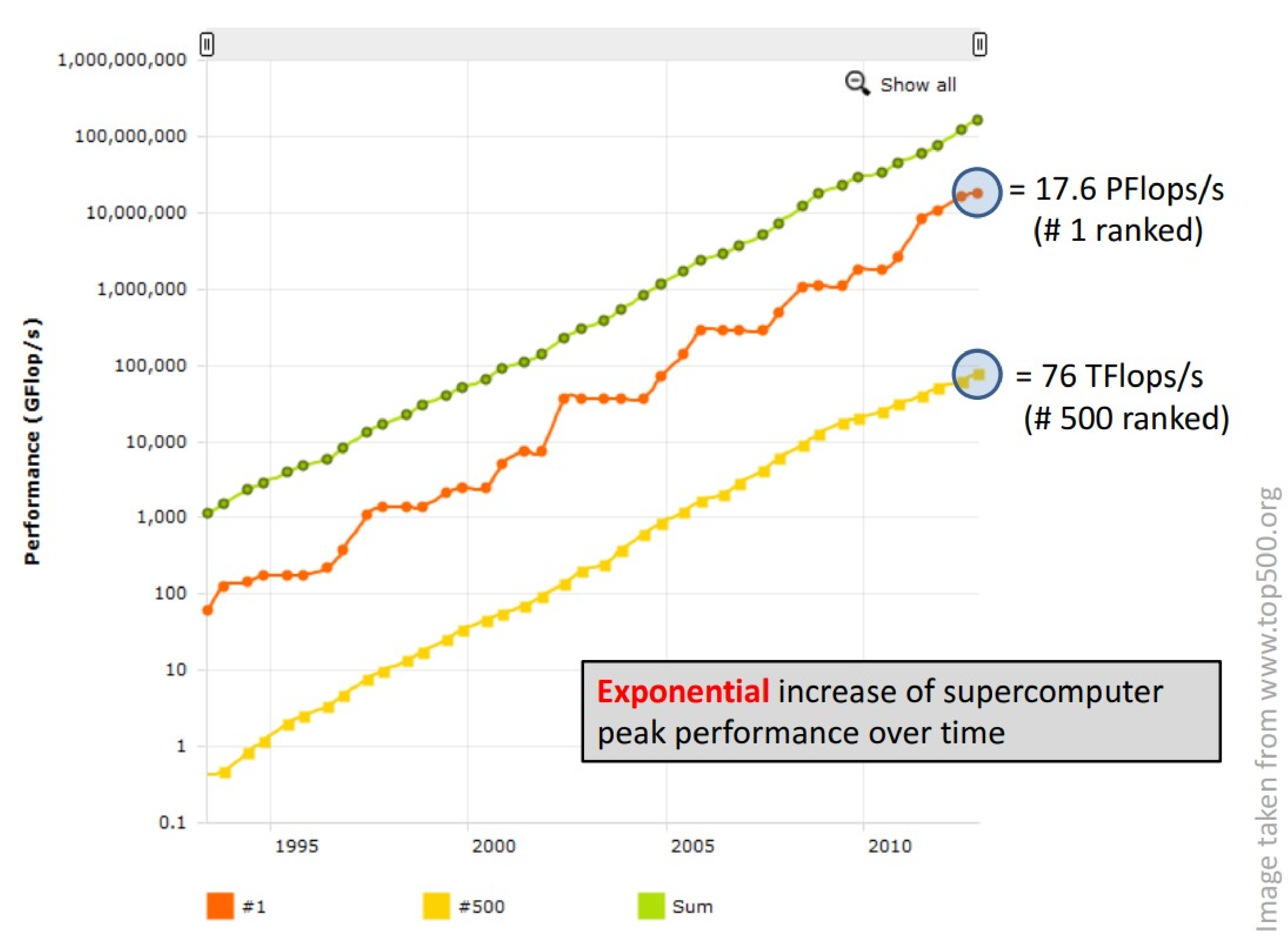
Illustration from Wikipedia

Moore's Law

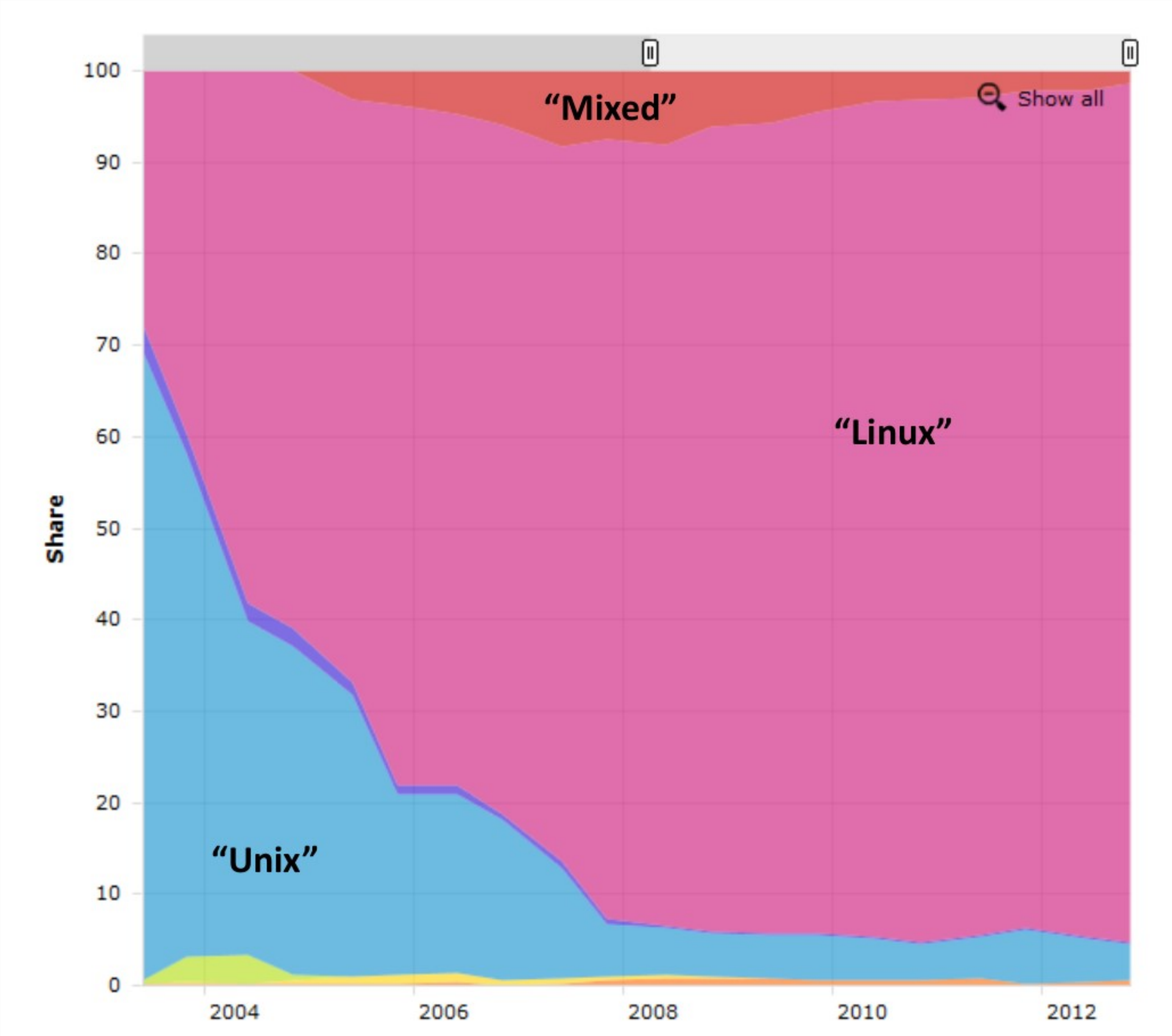


16-Core SPARC T3
Six-Core Core i7
Six-Core Xeon 7400
10-Core Xeon Westmere-EX
Dual-Core Itanium 2
8-core POWER7
Quad-core z196
Quad-Core Itanium Tukwila
8-Core Xeon Nehalem-EX
AMD K10
POWER6
Itanium 2 with 9MB cache
Six-Core Opteron 2400
AMD K9
Core i7 (Quad)
Core 2 Duo
Itanium 2
AMD K8
Barton
Atom
Pentium 4
2000
2011

Top 500 Supercomputers



Operating Systems on HPC systems



HPC systems are mostly dominated by Unix-like system

Motivation for Parallel Computing

1. Want to run the same program faster

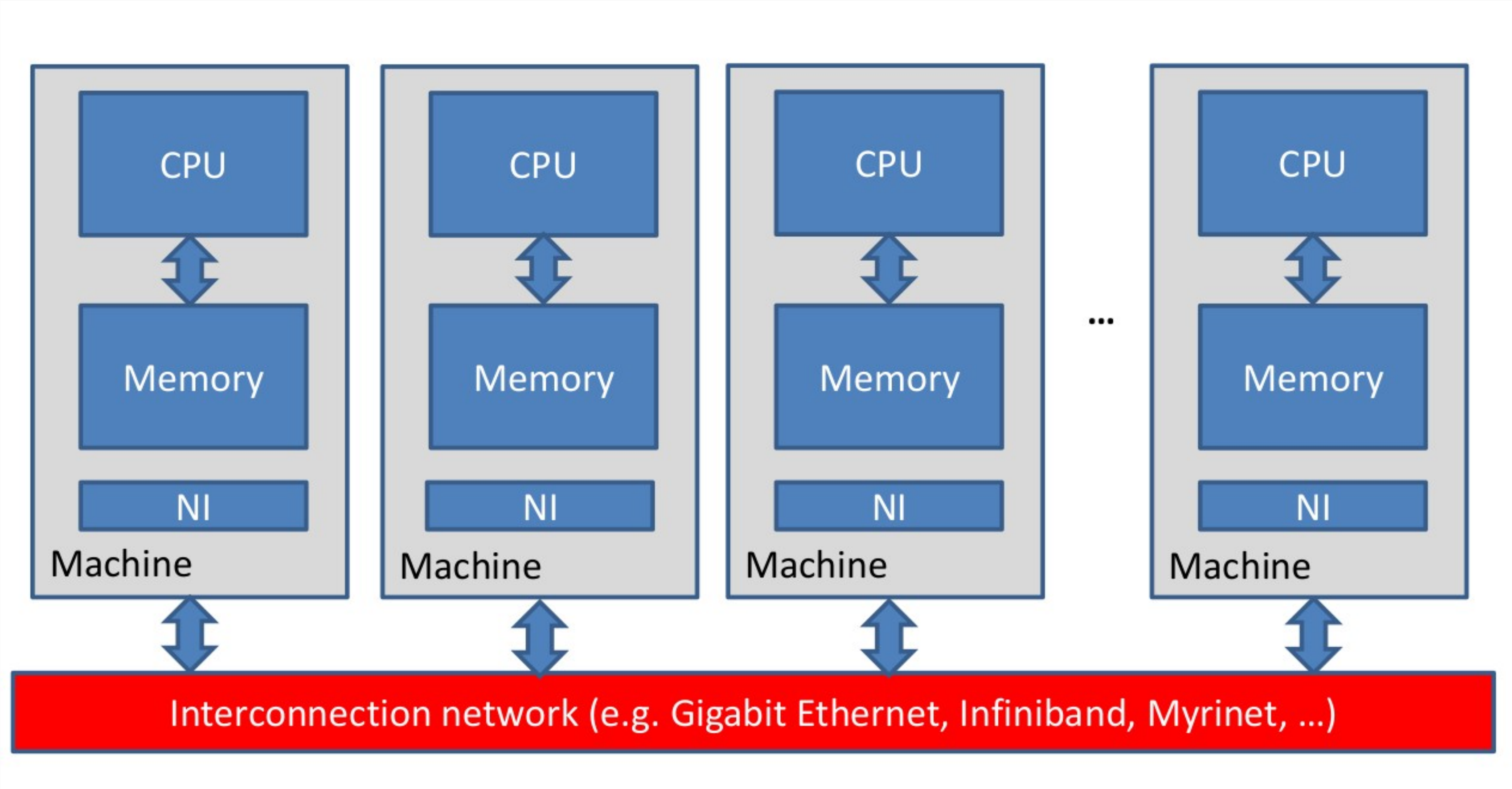
2. Want to run bigger datasets

   Datasets may be so big that it does not fit in RAM

3. Want to reduce financial cost or power consumption

   PhD students waiting for weeks for results!

Distributed Memory Architecture

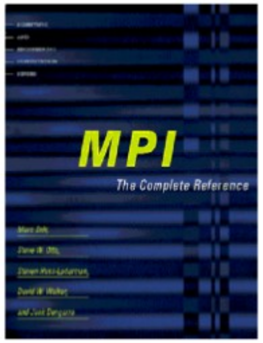


Message Passing Interface

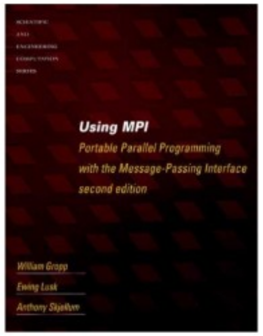
1. MPI: Library specification, not an implementation
2. Most important implementations: OpenMPI, Intel MPI
3. Specific routines for:
   3.1 Point-to-Point Communication
   3.2 Collective communications
   3.3 Topology setup
   3.4 Parallel I/O
4. Binding for C/C++

References for MPI

- **MPI standards:** http://www.mpi-forum.org/docs/



- **MPI: The Complete Reference** (M. Snir, S. Otto, S Huss-Lederman, D. Walker, J. Dongarra)
  Available from http://switzernet.com/people/emin-gabrielyan/060708-thesis-ref/papers/Snir96.pdf



- **Using MPI: Portable Parallel Programming with the Message Passing Interface**, 2^{nd} ed. (W. Gropp, E. Lusk, A. Skjellum).

History of MPI

Started in 1992 (Workshop for standards for Message Passing in a Distributed Memory Environment) with support from vendors, library writers, and academia.

1. MPI Version 1.0 (May 1994)
    1.1 Final pre draft in 1993
    1.2 Final version June 1994
2. MPI version 2.0
    2.1 Support for one sided communication
    2.2 Support for process management
    2.3 Support for parallel I/O
3. MPI version 3.0
    3.1 Support for non-blocking collective routine
    3.2 New one sided communication routines

Hello World in MPI

```
#include <mpi.h>
#include <iostream>
#include <cstdlib>

using namespace std;

int main(int argc, char* argv[]) {
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );

    cout << "Hello World from process" << rank << "/" << size << endl;

    MPI_Finalize();
    return EXIT_SUCCESS;
}
```

Output **order** is random

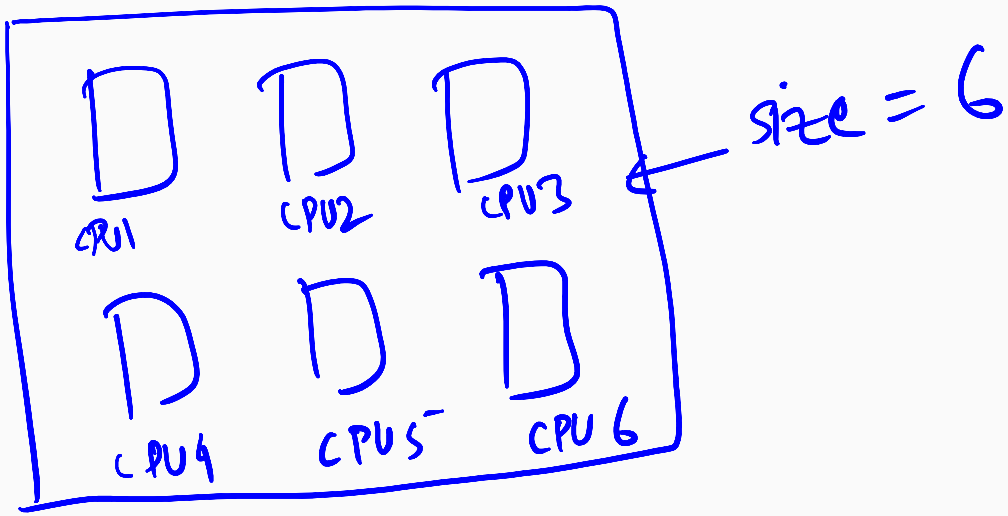
```
john@doe ~]$ mpirun -np 4 ./helloWorld
Hello World from process 2/4
Hello World from process 3/4
Hello World from process 0/4
Hello World from process 1/4
```

Basic MPI routines

- `int MPI_Init(int *argc, char ***argv)`
  - Initialization: All process must call this prior to any MPI routine
  - Strips off possible arguments provided by "mpirun"
- `int MPI_Finalize(void)`
  - Cleanup: all processes must call this routine at the end of the program
  - All pending communication should have finished before calling this

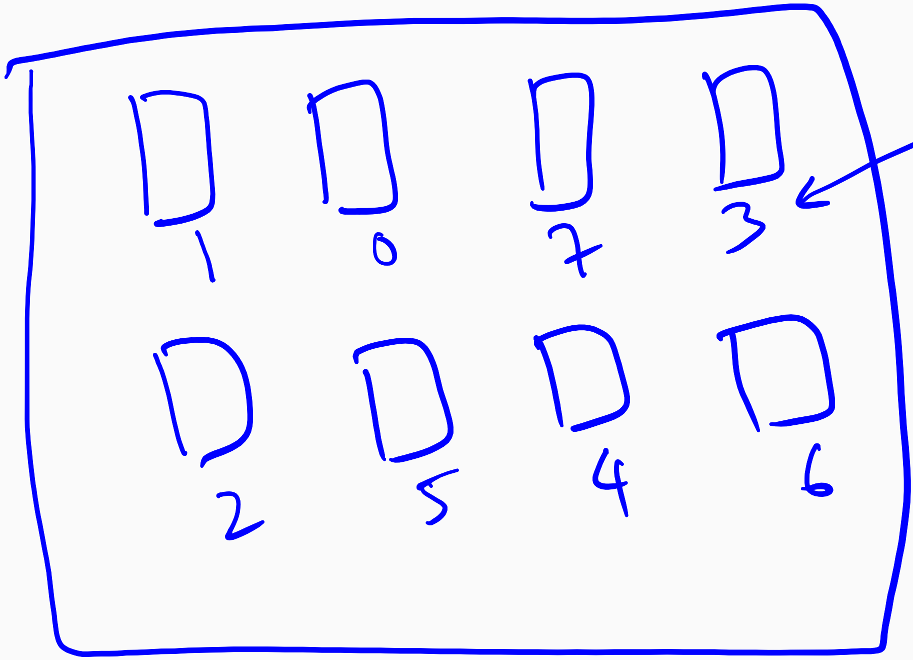
Basic MPI Routines

- `int MPI_Comm_size(MPI_Comm comm, int *size);`
  - Returns the size of the "Communicator" associated with "comm"
  - Communicator is user defined subset of processes
  - `MPI_COMM_WORLD` = communicator that involves all processes



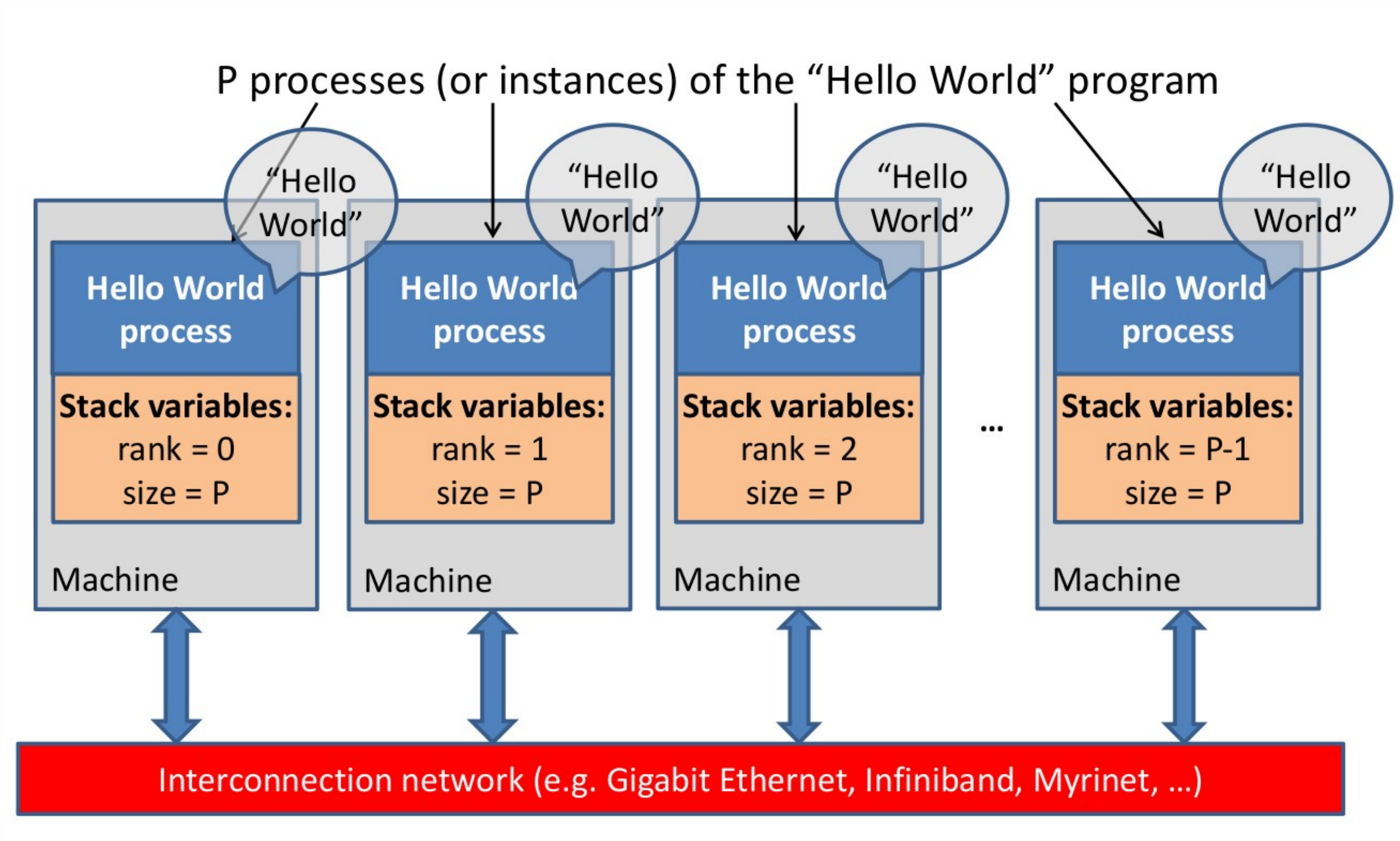
Basic MPI Routines

- `int MPI_Comm_rank(MPI_Comm comm, int *rank);`
  - Return the rank of the process in the Communicator
  - Range: [0 . . . size - 1]



After calling this, unique id called rank is assigned to each CPUs.

MPI Mechanisms



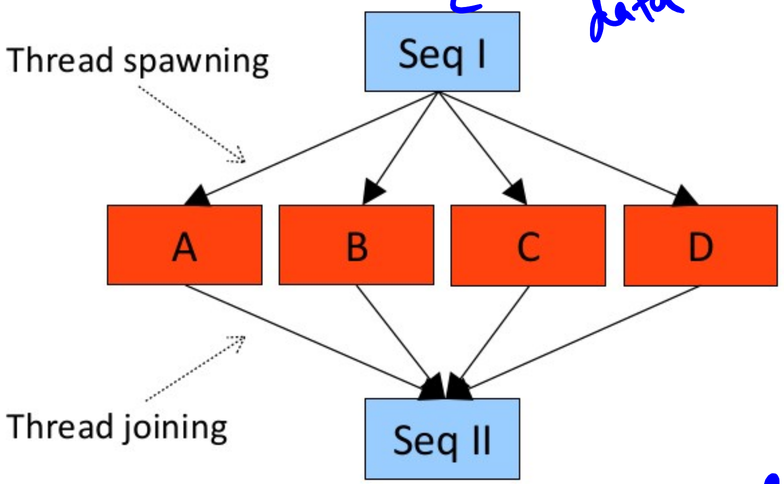
- MPI launches P independent processes: each process is an instance of same program

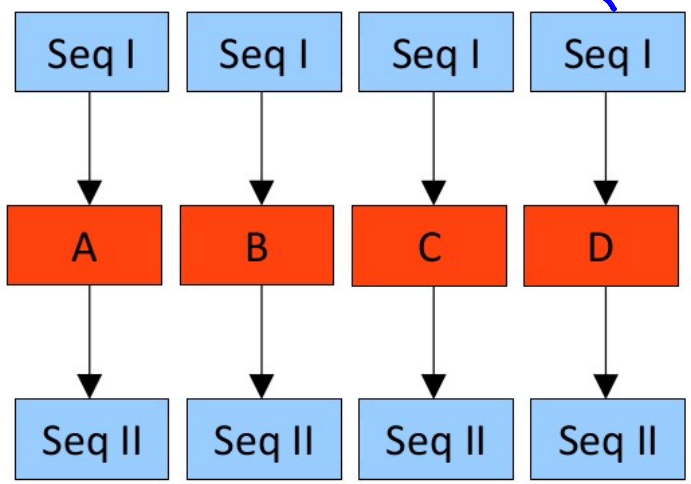
Multithreading Versus Multiprocessing

Multi-threading		Message passing
Single process	⟺	Multiple processes
Shared memory address space	⟺	Separate memory address spaces
Protect data against simultaneous writing	⟺	Explicitly communicate everything
Limited to a single machine	⟺	Multiple machines possible
E.g. Pthreads, CILK, OpenMP, etc.	⟺	E.g. MPI, Unified Parallel C, PVM

shared data

distributed data





Threads work on shared data

Processes work on their local data

Compiling and Running MPI programs

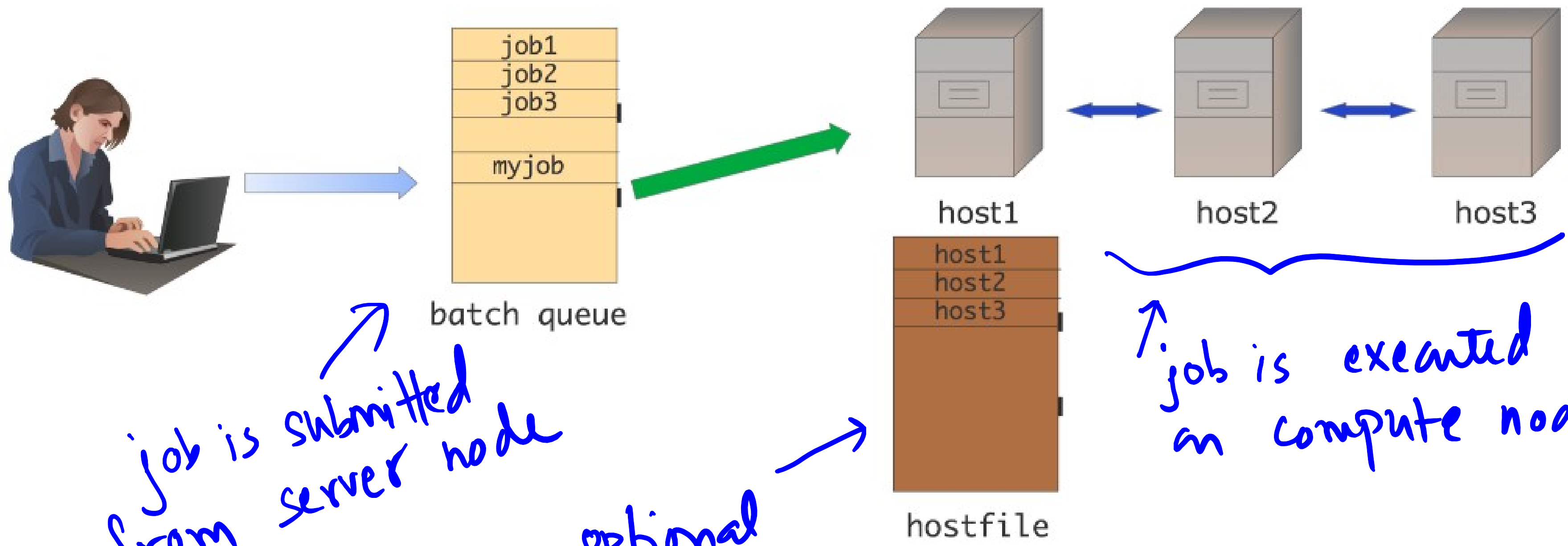
- Compiling: `mpicc -O3 main.cpp -o main`
- Running: `mpirun -np <number of program instances> <your program>`

Using MPI on ABACUS cluster at IIITH

1. check modules: `module avail`
2. load modules: `module load intel/2017a`
3. submit job using job script: `sbatch job_script.sh`

More details:
`http://hpc.iiit.ac.in/wiki/index.php/Main_Page`

Batch Job Submission in MPI



job is submitted from server node

optional
Can specify the nodes list

job is executed on compute nodes

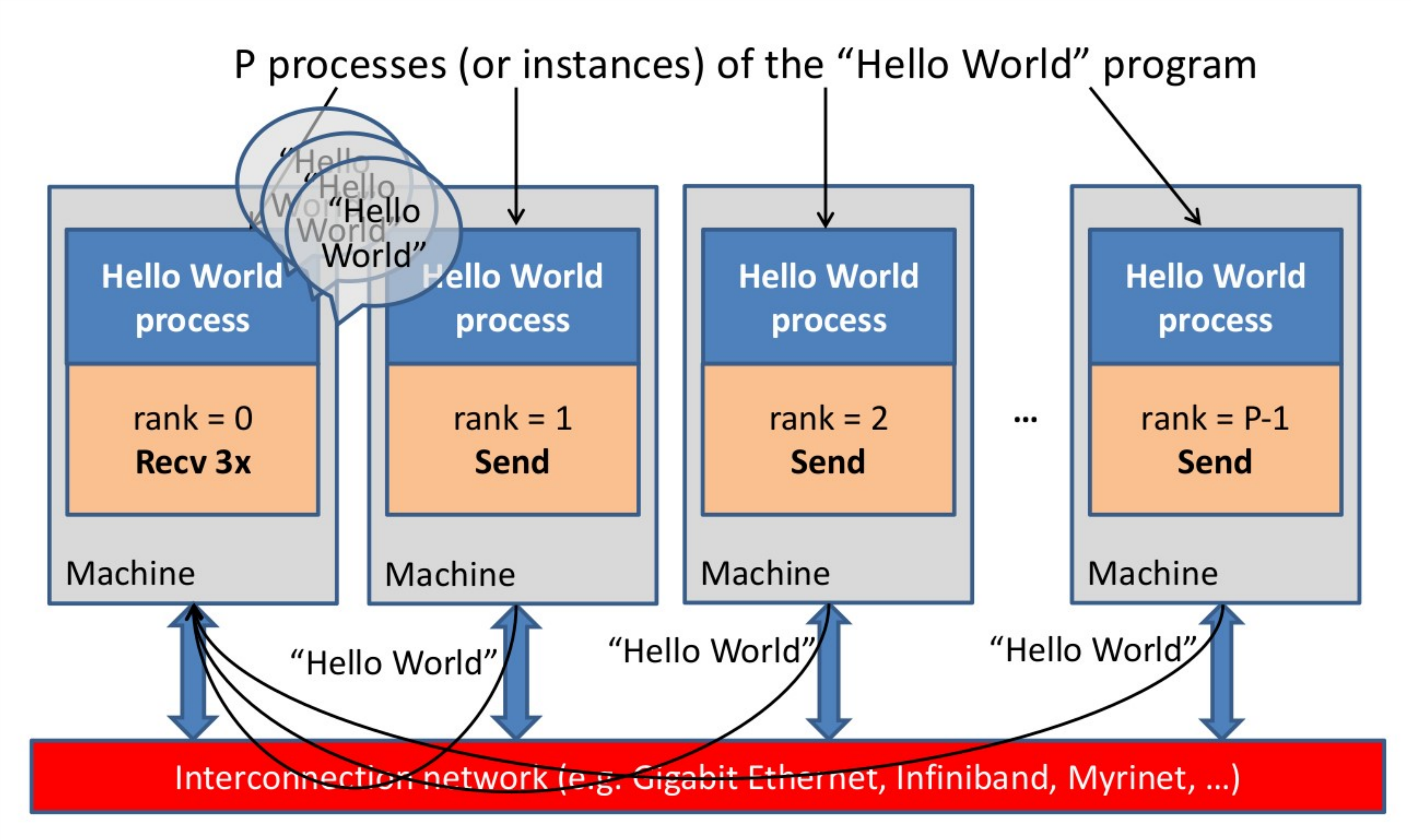
MPI: Point-to-Point Communication

```
...
int rank, size, count;
char b[40];
MPI_Status status;

...   // init MPI and rank and size variables

if (rank != 0) {
  char * str = "Hello World";
  MPI_Send(str, 12, MPI_CHAR, 0, 123, MPI_COMM_WORLD);
} else {
  for (int i = 1; i < size; i++) {
    MPI_Recv(b, 40, MPI_CHAR, i, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    MPI_Get_count(&status, MPI_CHAR, &count);
    printf("I received %s from process %d with size %d and tag %d\n",
           b, status.MPI_SOURCE, count, status.MPI_TAG);
  }
}
...
```

branching on rank

```
john@doe ~]$ mpirun -np 4 ./ptpcomm
I received Hello World from process 1 with size 12 and tag 123
I received Hello World from process 2 with size 12 and tag 123
I received Hello World from process 3 with size 12 and tag 123
```

MPI: Point-to-Point Communication View



MPI: send and receive

int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

- **buf**: pointer to the message to send
- **count:** number of items to send
- **datatype:** datatype of each item
- **dest:** rank of destination process
- **tag:** value to identify the message [0 . . . at least (32 767)]
- **comm:** communicator specification (e.g. `MPI_COMM_WORLD`)

MPI: send and receive

int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)

- **buf:** pointer to the buffer to store received data
- **count:** upper bound (!) of the number of items to receive
- **datatype:** datatype of each item
- **source:** rank of source process (or `MPI_ANY_SOURCE`)
- **tag:** value to identify the message (or `MPI_ANY_TAG`)
- **comm:** communicator specification (e.g. `MPI_COMM_WORLD`)
- **status:** structure that contains `MPI_SOURCE`, `MPI_TAG`, `MPI_ERROR`

Sending and Receiving

Two-sided communication:

- Both the sender and receiver are involved in data transfer
- Posted send must match receive

When do MPI_Send and MPI_recv match ?

- Rank of receiver process
- Rank of sending process
- Tag (custom value to distinguish messages from same sender)
- Communicator

Rationale for communicators:

- Used to create subsets of processes