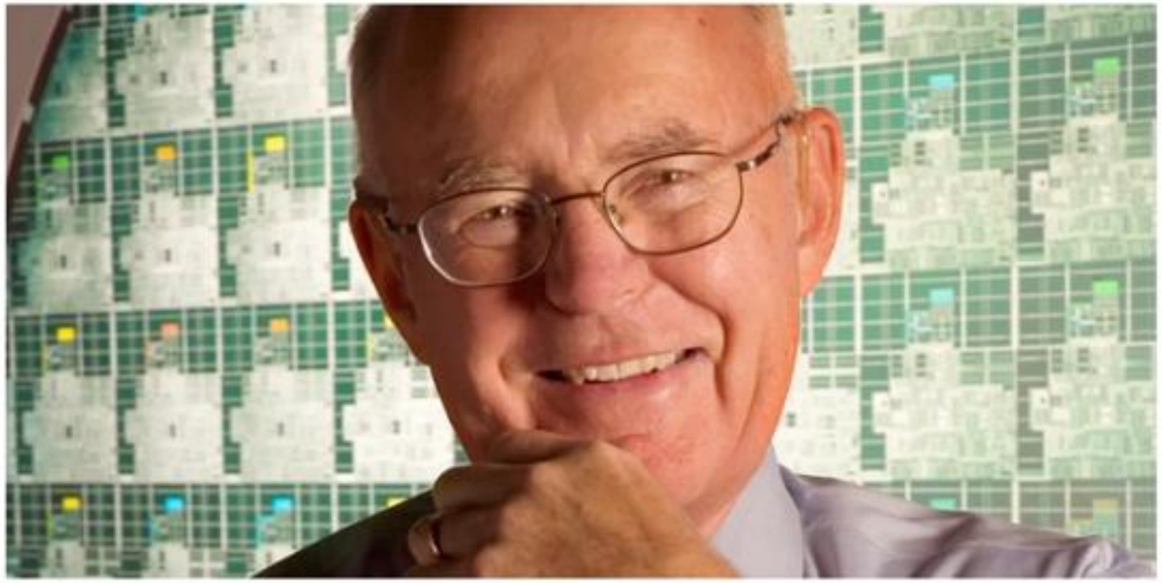
OpenMP and MPI

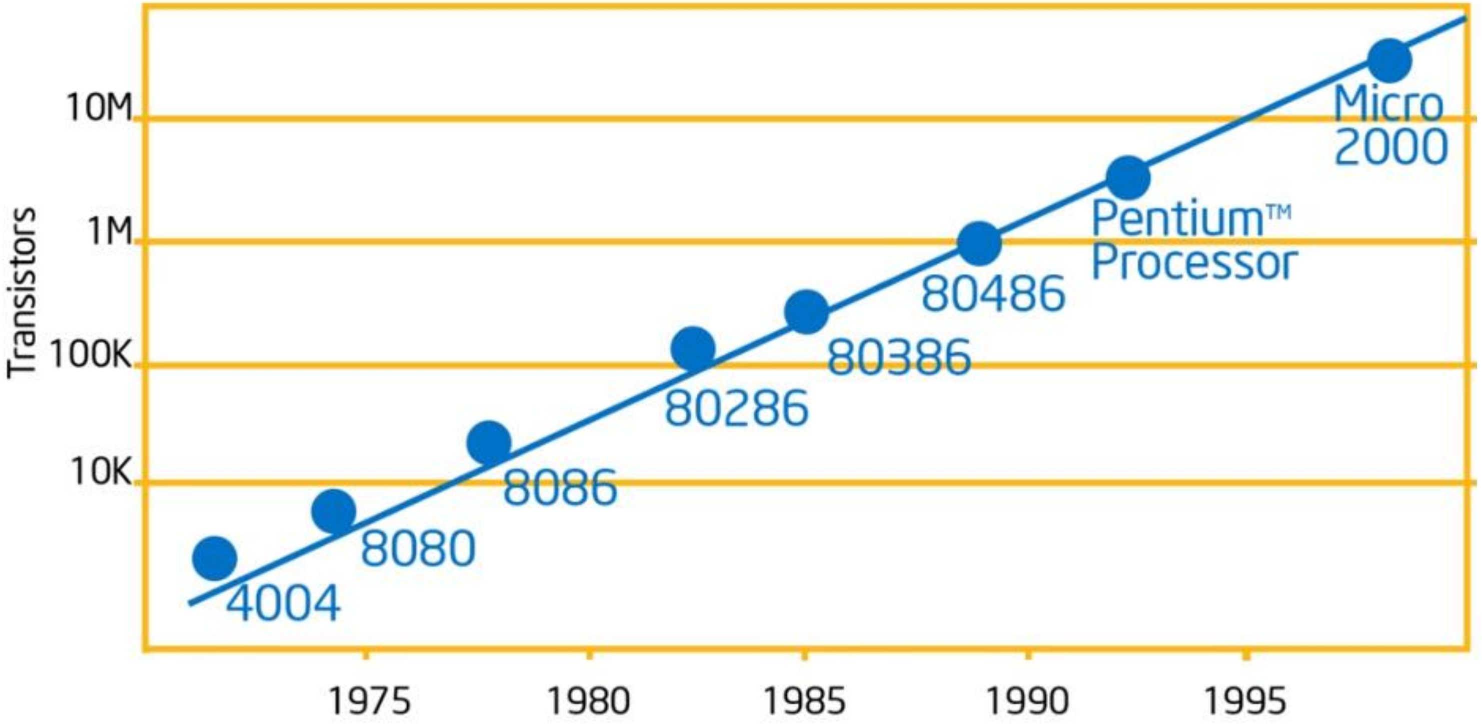
Muhammad Mohsin Raza 19P-0072
Tahawar Ihsan 19P-0097

Moore's Law



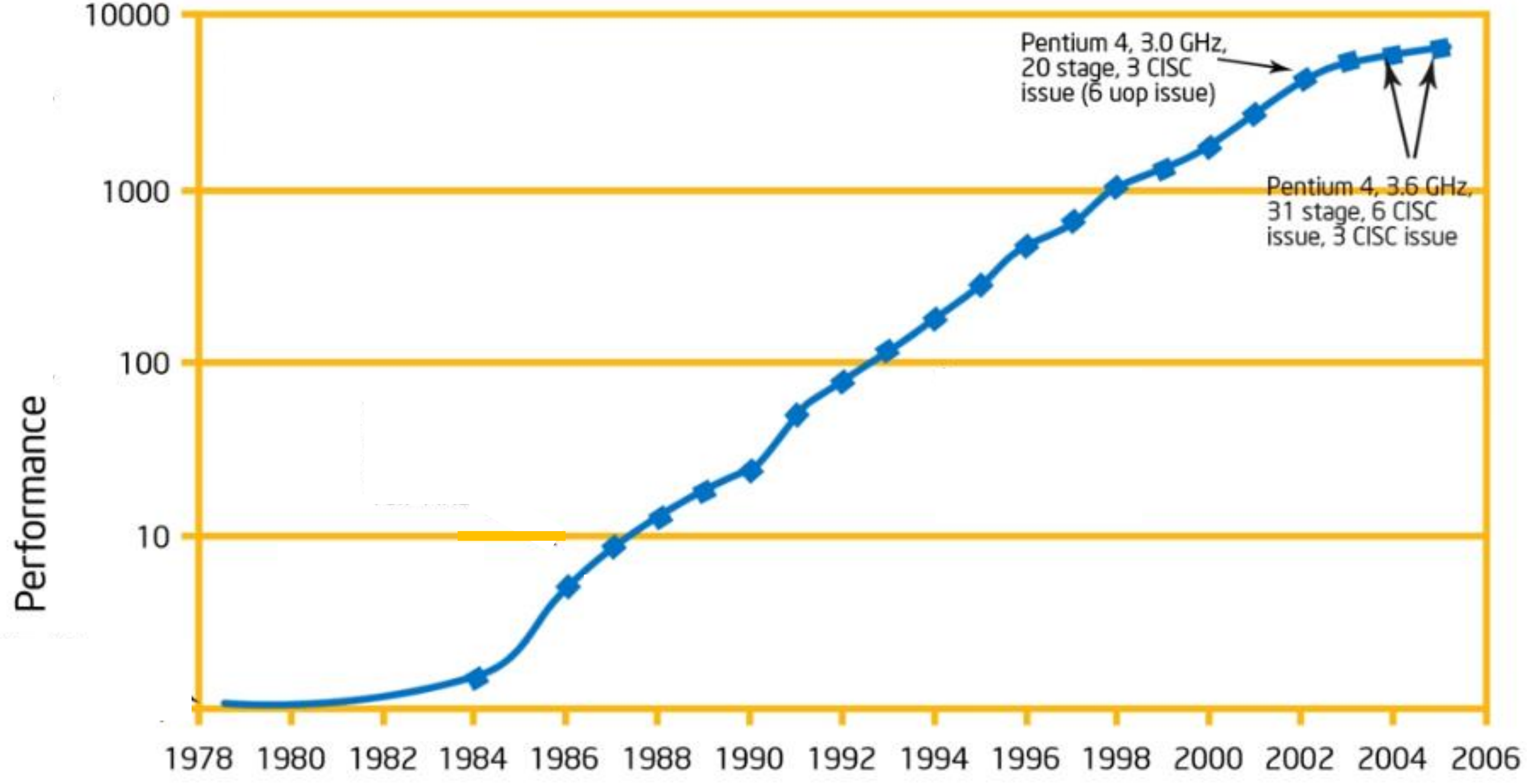
Forecast that the number of transistors incorporated in a chip would approximately double every 24 months.

Moore's Law



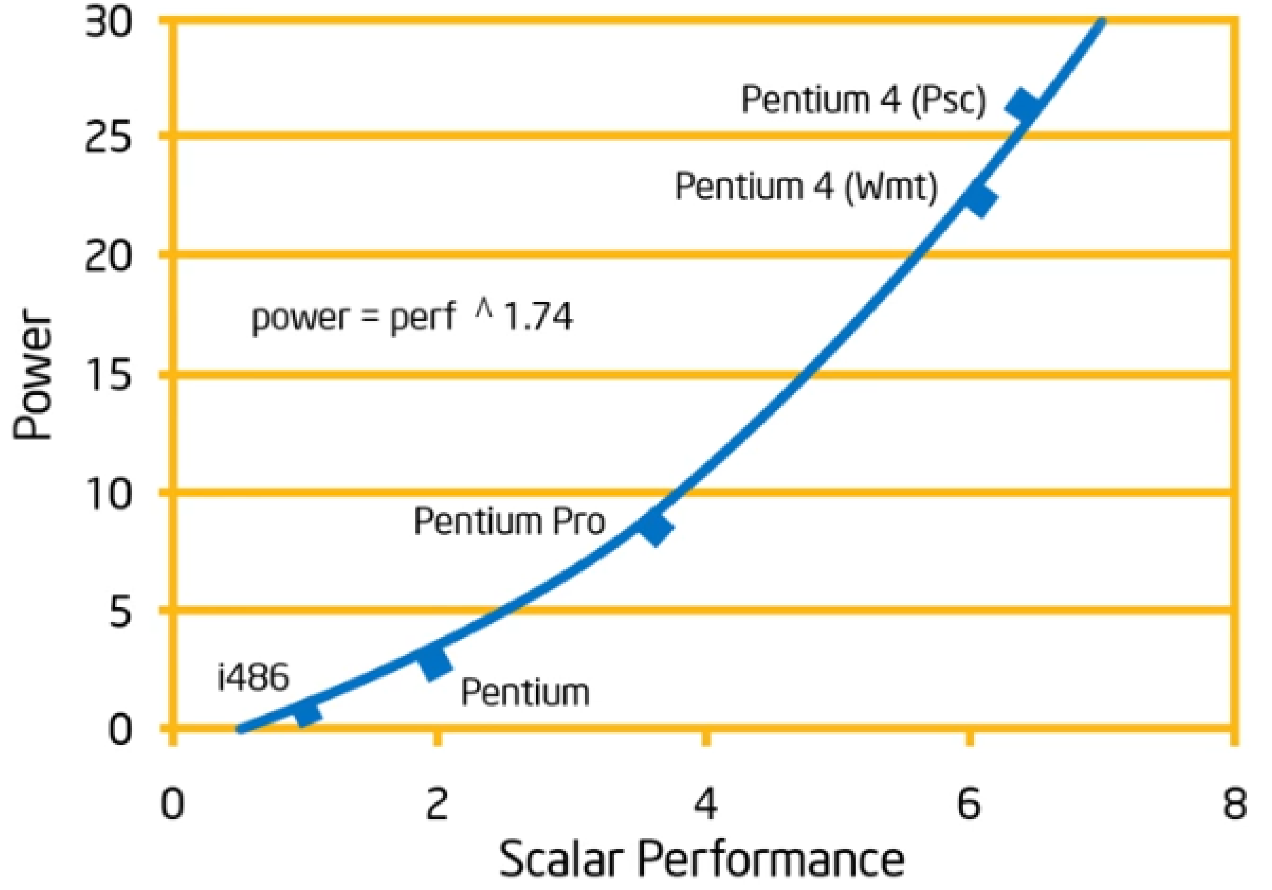
10M
1M
100K
10K
Transistors
1975
1980
1985
1990
1995
4004
8080
8086
80286
80386
80486
Pentium™ Processor
Micro 2000

The Good Old Days



From Hennessy and Patterson, *Computer Architecture: A Quantitative Approach*, 4th edition, Sept. 15, 2006

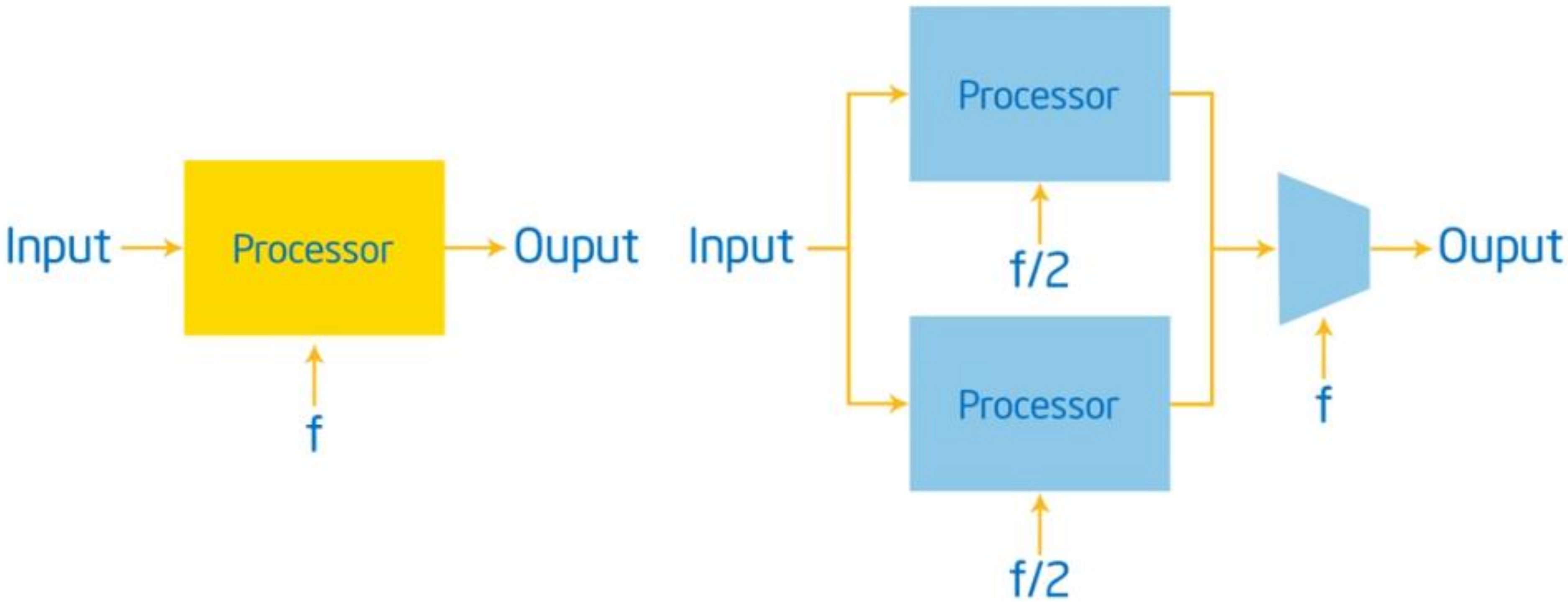
Computer Architecture and the Power Wall



Growth in Power is Unsustainable

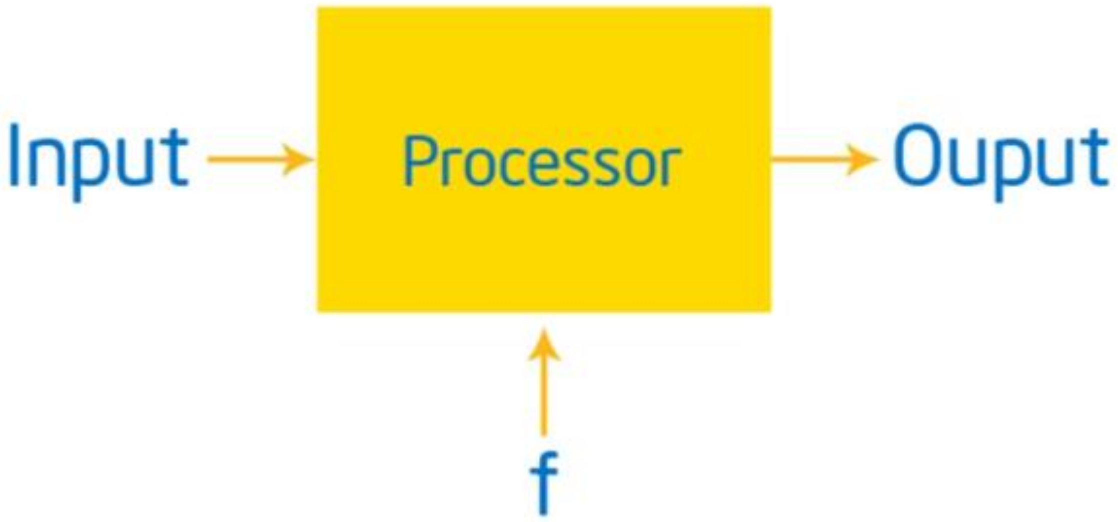
Source: E. Grochowski of Intel

$$\text{Power} = CV^2F$$



Chandrakasan, A.P.; Potkonjak, M.; Mehra, R.; Rabaey, J.; Brodersen, R.W., "Optimizing power using transformations," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,, vol.14, no.1, pp.12-31, Jan 1995

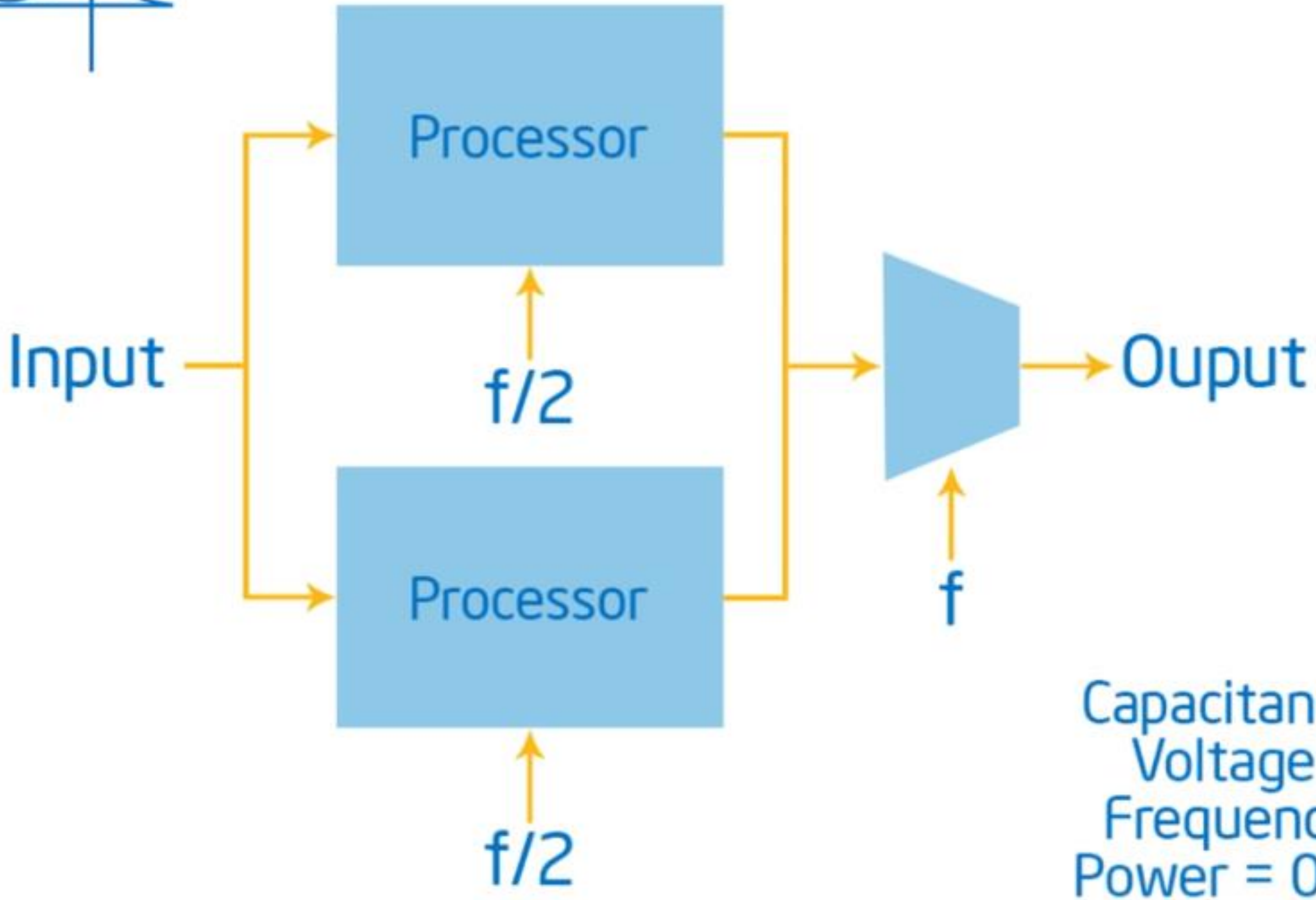
Source:
Vishwani Agrawal



Capacitance = C
Voltage = V
Frequency = f
Power = CV^2F

Chandrakasan, A.P.; Potkonjak, M.; Mehra, R.; Rabaey, J.; Brodersen, R.W., "Optimizing power using transformations," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,, vol.14, no.1, pp.12-31, Jan 1995

Source:
Vishwani Agrawal



Capacitance = 2.2C
Voltage = 0.6V
Frequency = 0.5f
Power = $0.396CV^2F$

Chandrakasan, A.P.; Potkonjak, M.; Mehra, R.; Rabaey, J.; Brodersen, R.W., "Optimizing power using transformations," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,, vol.14, no.1, pp.12-31, Jan 1995

Source:
Vishwani Agrawal

Concurrency vs. Parallelism

Concurrency

A condition of a system in which multiple tasks are logically active at one time.

Parallelism

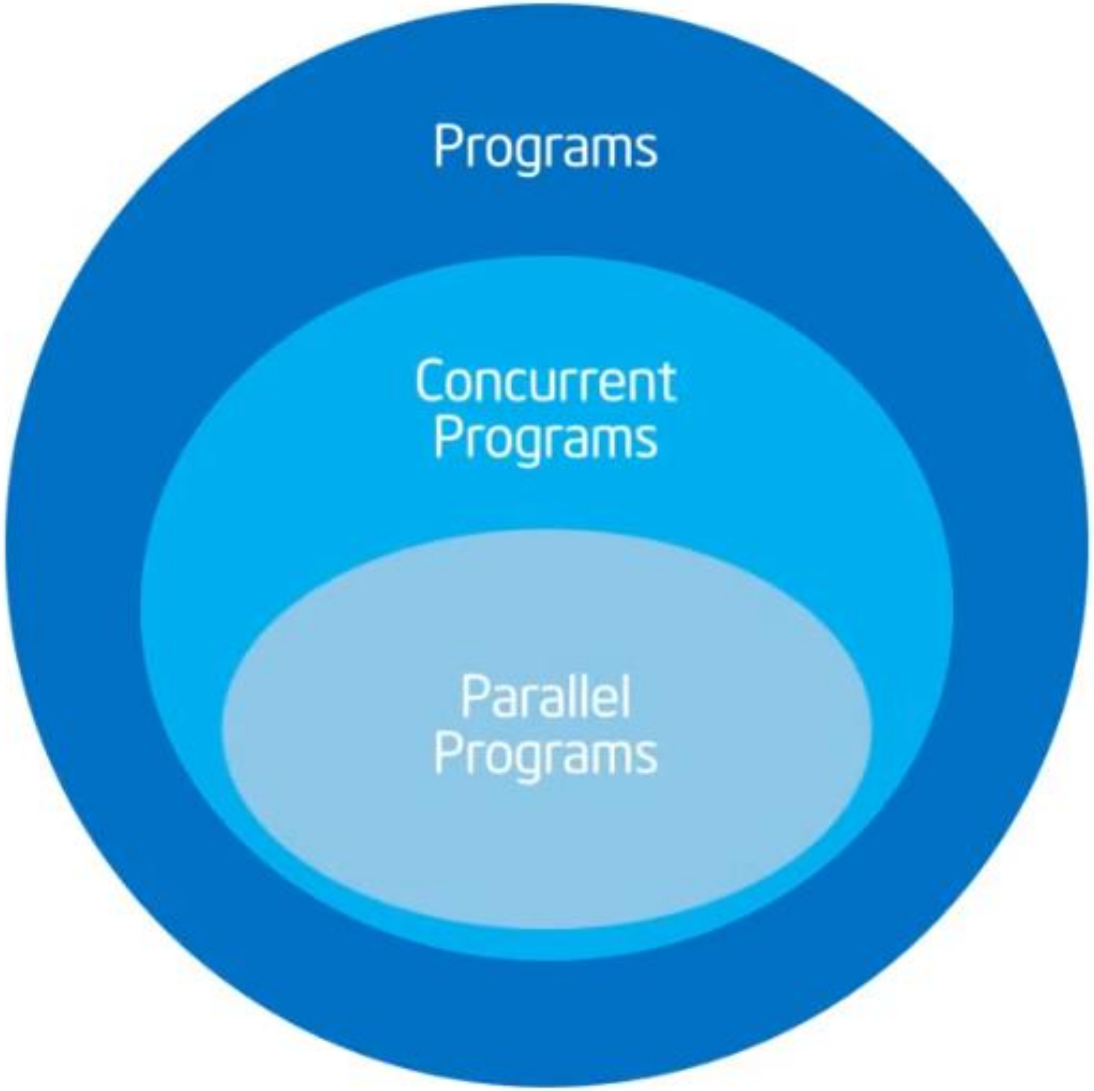
A condition of a system in which multiple tasks are actually active at one time.

Concurrency vs. Parallelism

Concurrent, non-parallel execution

Concurrency vs. Parallelism

Concurrent, parallel execution


Programs
Concurrent
Programs
Parallel
Programs

OpenMP

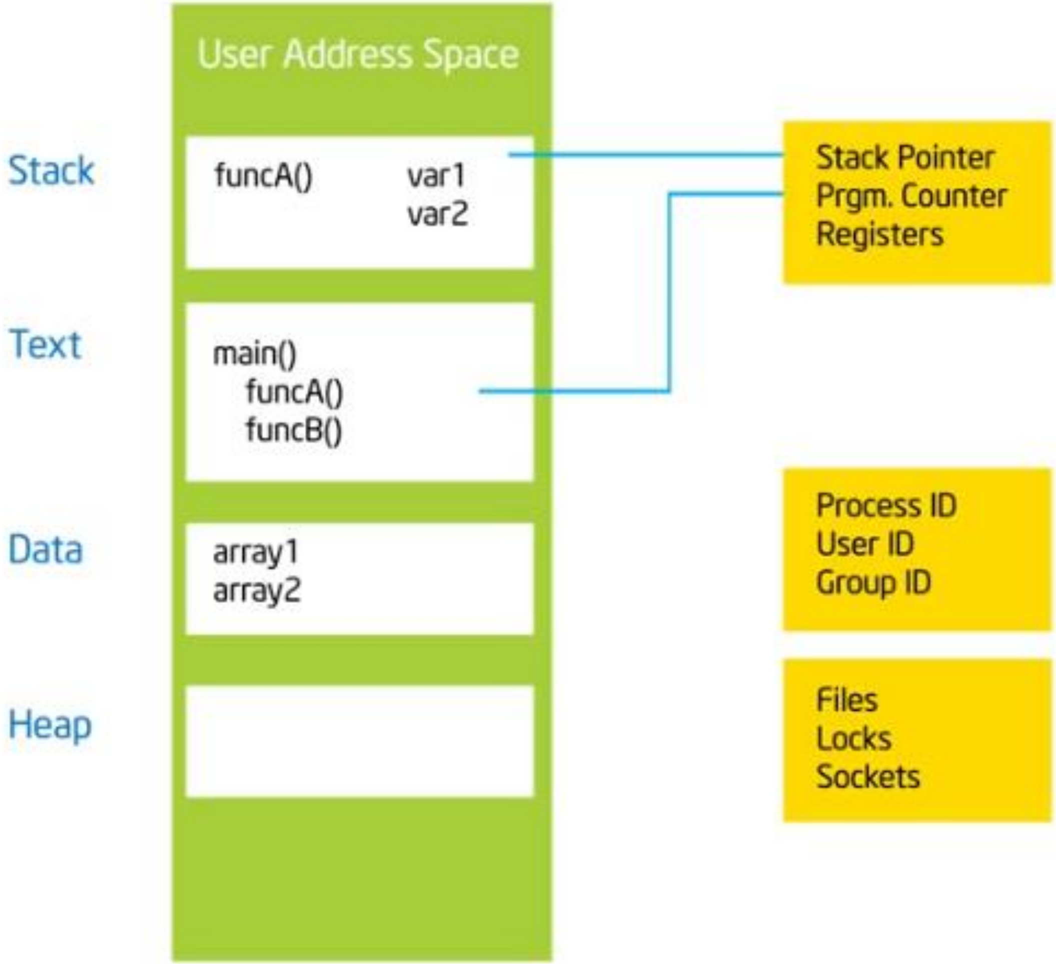
- openMP="open multi-processing"
- Compiler directives, library routines for parallelism
- used to achieve parallelism on a single multicore computer
- it follows shared memory
- You can specify how variables are shared
- private: each thread has its own copy of these variable
- Shared: threads read/write to these common variable
- it gave openmp api for c/c++ languages.

Which header file should we include in C/C++ language?

- For C/C++, we have to include "omp.h" header file. "omp.h" is available for gcc/g++ compilers.
- . A simple example to create threads:
- #pragma omp parallel
- {
- printf("Hello World");
- }

Shared Memory Machines

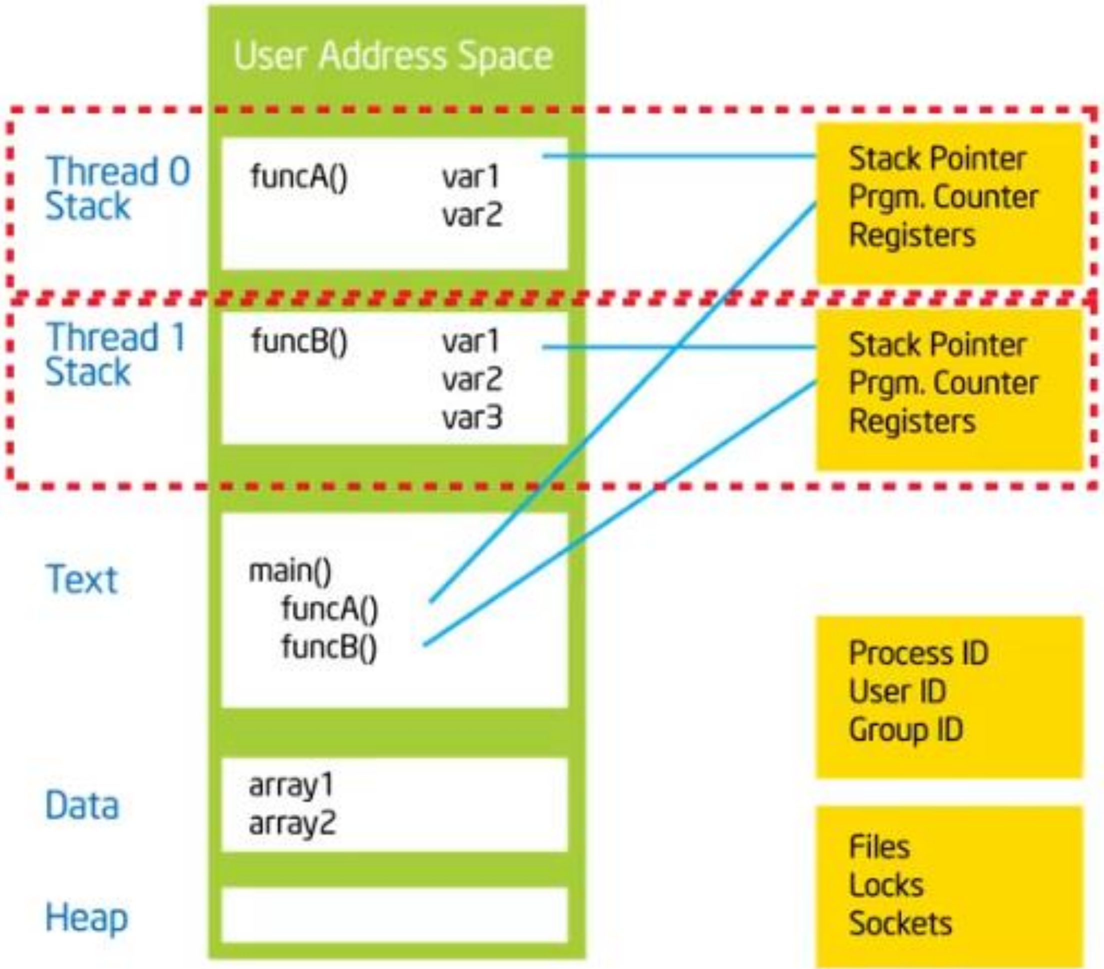
Programming Shared Memory Computers



Process:

* An instance of a program execution.
* The execution context of a running program... i.e. the resources associated with a program's execution.

Programming Shared Memory Computers

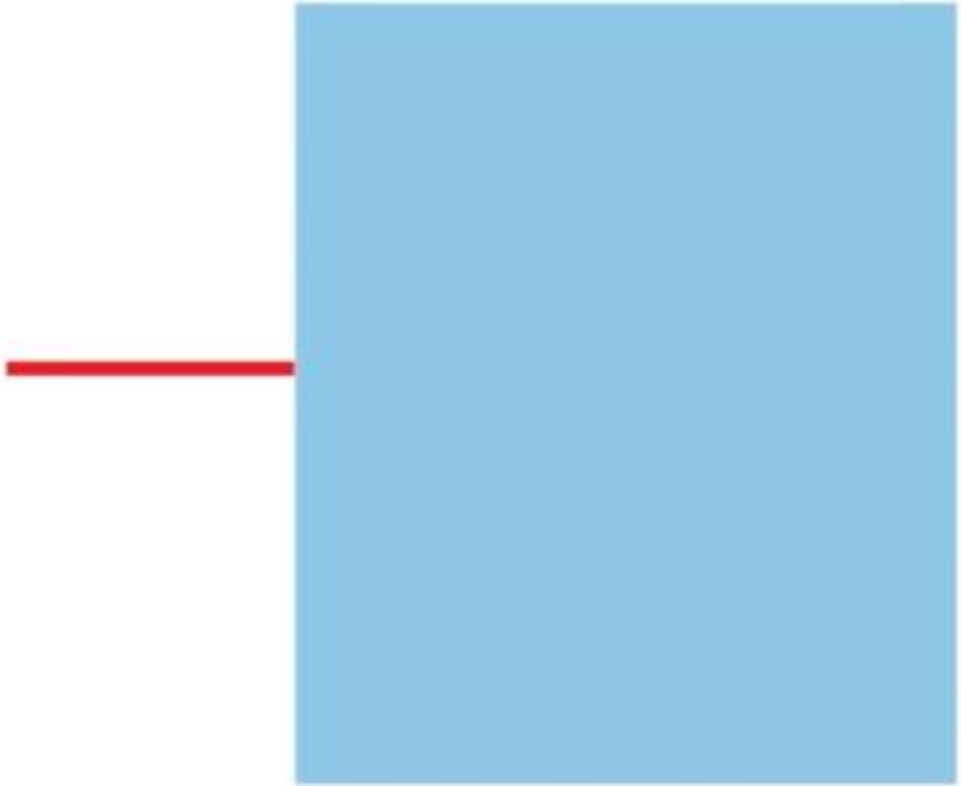


Threads:

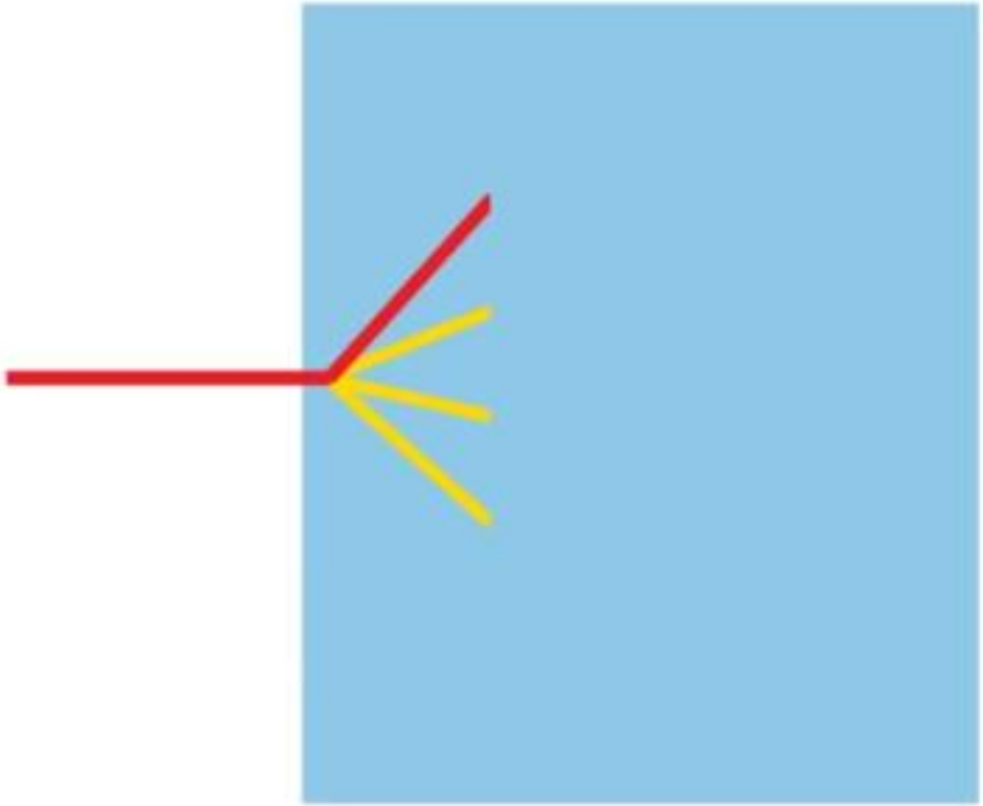
* Threads are "light weight processes"
* Threads share Process state among multiple threads. This greatly reduces the cost of switching context.

Fork-Join Parallelism

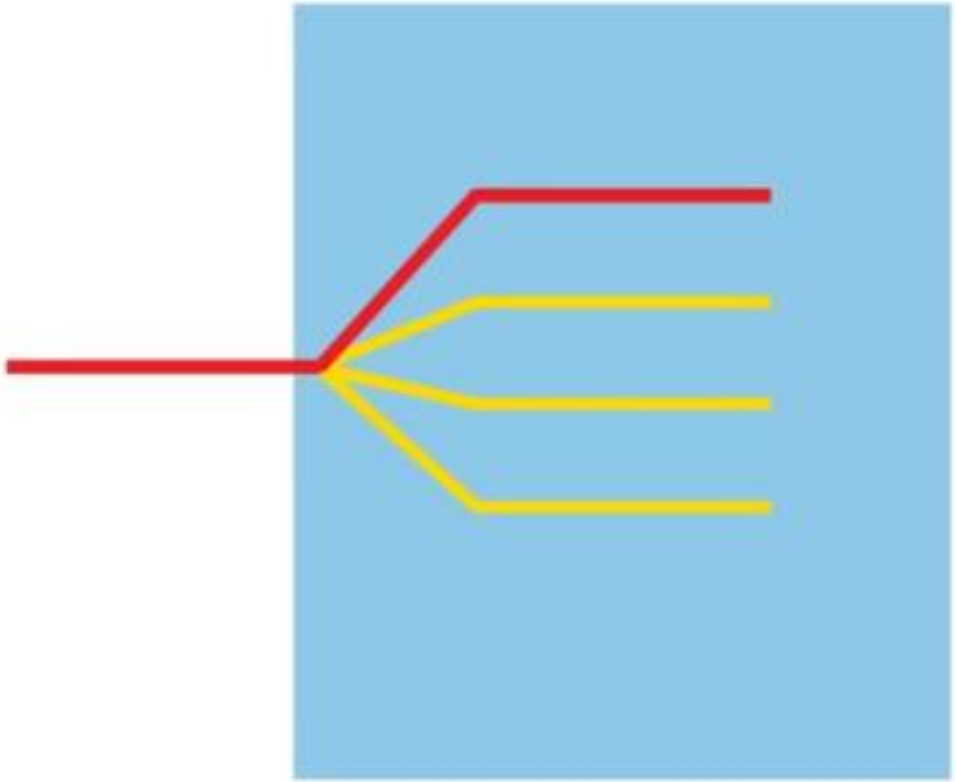
Fork-Join Parallelism



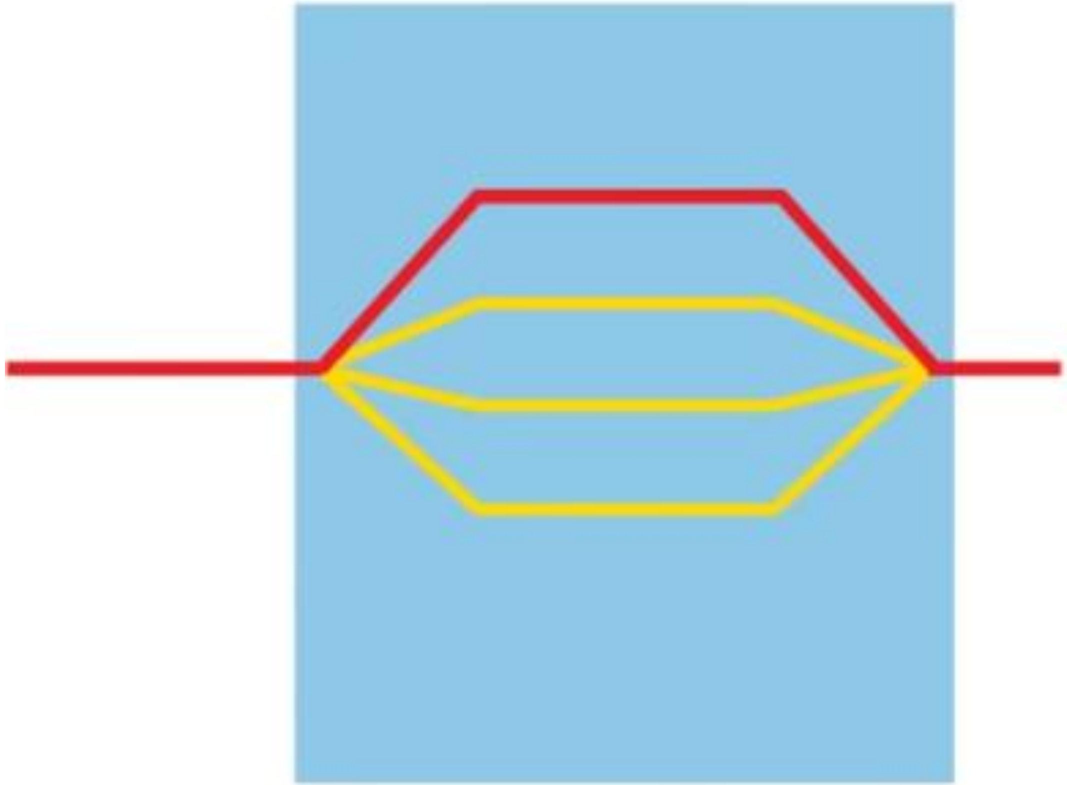
Fork-Join Parallelism



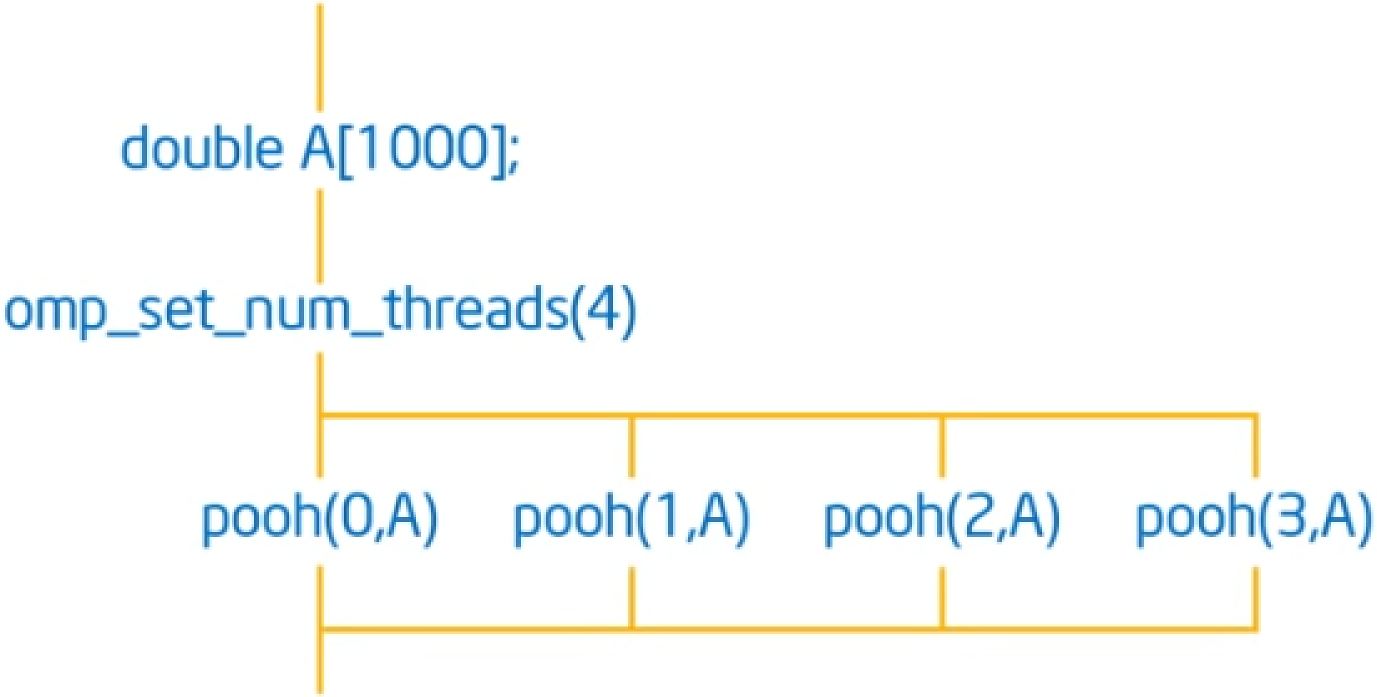
Fork-Join Parallelism



Fork-Join Parallelism

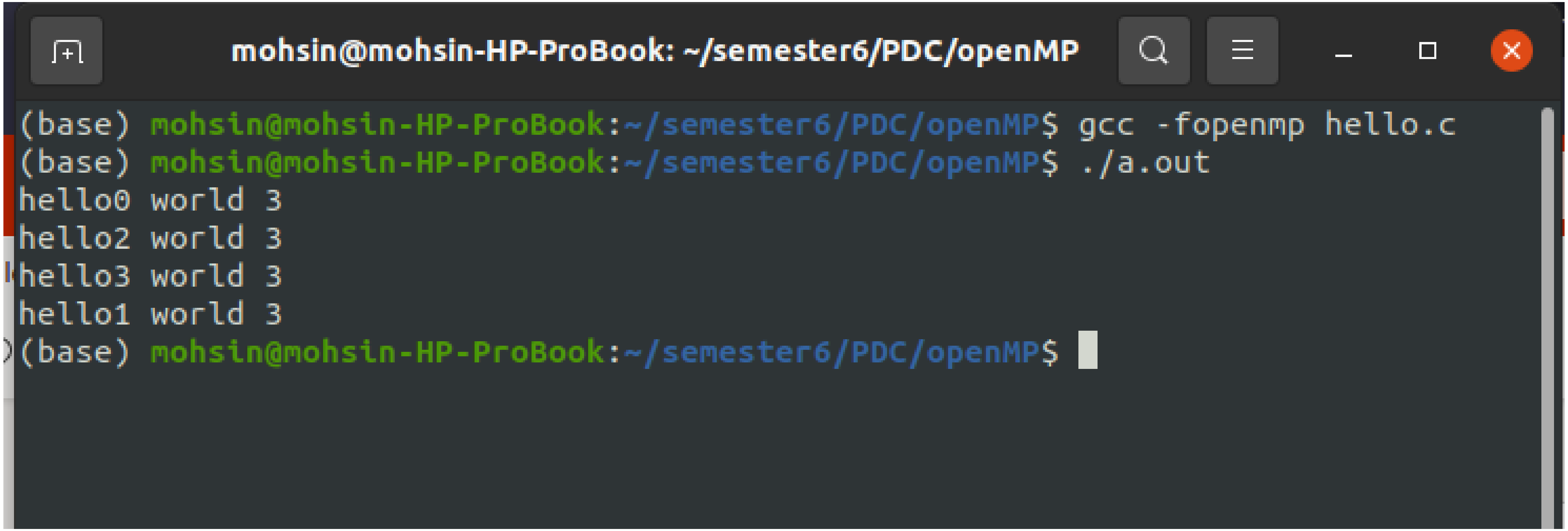


```c
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
        int ID = omp_get_thread_num();
        pooh(ID,A);
}
printf("all done\n");
```




```c
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
        int ID = omp_get_thread_num();
        pooh(ID,A);
}
printf("all done\n");
```

```
#include "omp.h"
int main()
{
#pragma omp parallel
 {
   int ID = omp_get_thread_num();
   printf("hello(%d) ", ID);
   printf(" world(%d) \n", ID);
  }
}
```


mohsin@mohsin-HP-ProBook: ~/semester6/PDC/openMP
(base) mohsin@mohsin-HP-ProBook:~/semester6/PDC/openMP$ gcc -fopenmp hello.c
(base) mohsin@mohsin-HP-ProBook:~/semester6/PDC/openMP$ ./a.out
hello0 world 3
hello2 world 3
hello3 world 3
hello1 world 3
(base) mohsin@mohsin-HP-ProBook:~/semester6/PDC/openMP$

OpenMP vs POSIX threads

- Any programmers find posix to be hard, cumbersome
- Crypted functions calls such as:
- pthread_create(), pthread_exit(), pthread_join()

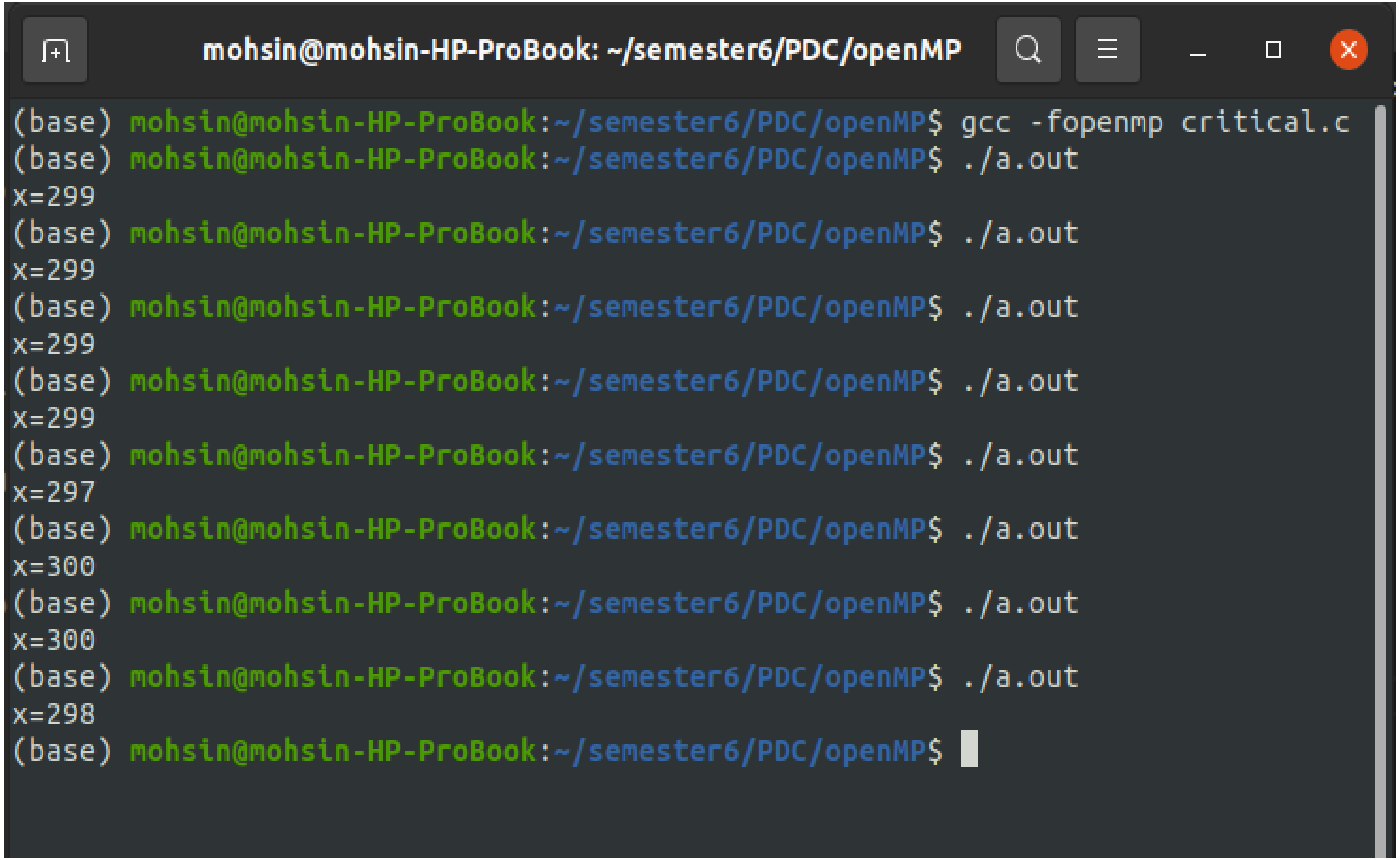
- Code is dependent on Posix compatible platforms

```
void thunk ()
{
        foobar ();
}

pthread_t tid[4];
for (int i = 1; i < 4; ++i)
   pthread_create (
         &tid[i],0,thunk, 0);
think();

for (int i = 1; i < 4; ++i)
   pthread_join (tid[i]);
```

```
#pragma omp parallel num_threads(4)
{
        foobar ();
}
```

How to synchronize threads in openMP?

- We can avoid race condition by using preprocessor directive "#pragma omp critical".
- Check following example:
- #pragma omp parallel num_threads(300)
- { #pragma omp critical
- {
- x=x+1;
- }
- printf("x=%d\n",x);
- }


mohsin@mohsin-HP-ProBook: ~/semester6/PDC/openMP
(base) mohsin@mohsin-HP-ProBook:~/semester6/PDC/openMP$ gcc -fopenmp critical.c
(base) mohsin@mohsin-HP-ProBook:~/semester6/PDC/openMP$ ./a.out
x=299
(base) mohsin@mohsin-HP-ProBook:~/semester6/PDC/openMP$ ./a.out
x=299
(base) mohsin@mohsin-HP-ProBook:~/semester6/PDC/openMP$ ./a.out
x=299
(base) mohsin@mohsin-HP-ProBook:~/semester6/PDC/openMP$ ./a.out
x=299
(base) mohsin@mohsin-HP-ProBook:~/semester6/PDC/openMP$ ./a.out
x=297
(base) mohsin@mohsin-HP-ProBook:~/semester6/PDC/openMP$ ./a.out
x=300
(base) mohsin@mohsin-HP-ProBook:~/semester6/PDC/openMP$ ./a.out
x=300
(base) mohsin@mohsin-HP-ProBook:~/semester6/PDC/openMP$ ./a.out
x=298
(base) mohsin@mohsin-HP-ProBook:~/semester6/PDC/openMP$

Synchronization

- #include<stdio.h>
- #include<omp.h>
- void main()
- {
- int x=0;
- omp_lock_t writelock;.
- omp_init_lock(&writelock);
- #pragma omp parallel num_threads(300)
- {
- omp_set_lock(&writelock);
- x=x+1;
- omp_unset_lock(&writelock);
- }
- printf("x=%d\n",x);
- omp_destroy_lock(&writelock);
- }

```
mohsin@mohsin-HP-ProBook: ~/semester6/PDC/openMP
(base) mohsin@mohsin-HP-ProBook:~/semester6/PDC/openMP$ gcc -fopenmp sync.c
(base) mohsin@mohsin-HP-ProBook:~/semester6/PDC/openMP$ ./a.out
x=300
(base) mohsin@mohsin-HP-ProBook:~/semester6/PDC/openMP$ ./a.out
x=300
(base) mohsin@mohsin-HP-ProBook:~/semester6/PDC/openMP$ ./a.out
x=300
(base) mohsin@mohsin-HP-ProBook:~/semester6/PDC/openMP$ ./a.out
x=300
(base) mohsin@mohsin-HP-ProBook:~/semester6/PDC/openMP$ ./a.out
x=300
(base) mohsin@mohsin-HP-ProBook:~/semester6/PDC/openMP$ ./a.out
x=300
(base) mohsin@mohsin-HP-ProBook:~/semester6/PDC/openMP$
```

MPI (Message Passing Interface)?

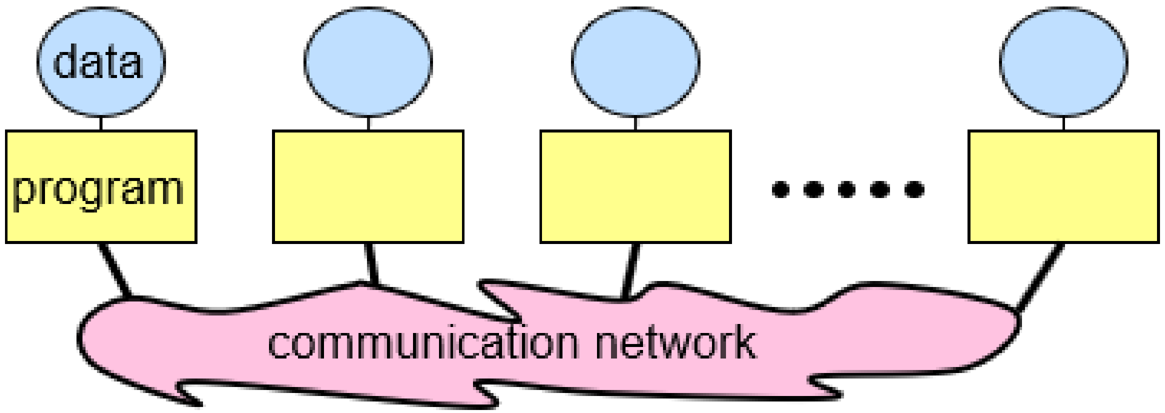
- Good at multi processing
- Usually memory is not shared(separate computers work together in parallel to perform common thing)
- Library of routines allow you to talk to other computers
- Behind the scene, most implementations set up communication between processes on other computers using TCP sockets
- Supported by Fortran, C, C++ (but modules also available for python, & Java)
- Hides hardware details of underlying system (so portable)
- Many high-performance libraries have MPI versions of API calls
- MPI version 3.0 specification has 400+ commands (function calls). Knowledge of only 11-12 of them can help you do the job in more than 90% of cases.

Information about MPI

- **MPI: A Message-Passing Interface Standard** (1.1, June 12, 1995)
- **MPI-2: Extensions to the Message-Passing Interface** (July 18,1997)
- **MPI: The Complete Reference**, Marc Snir and William Gropp et al, The MIT Press, 1998 (2-volume set)
- **Using MPI: Portable Parallel Programming With the Message-Passing Interface** and **Using MPI-2: Advanced Features of the Message-Passing Interface.** William Gropp, Ewing Lusk and Rajeev Thakur, MIT Press, 1999 – also available in a single volume *ISBN 026257134X.*
- **Parallel Programming with MPI**, Peter S. Pacheco, Morgen Kaufmann Publishers, 1997 - *very good introduction.*
- **Parallel Programming with MPI**, Neil MacDonald, Elspeth Minty, Joel Malard, Tim Harding, Simon Brown, Mario Antonioletti. Training handbook from EPCC which can be used together with these slides -

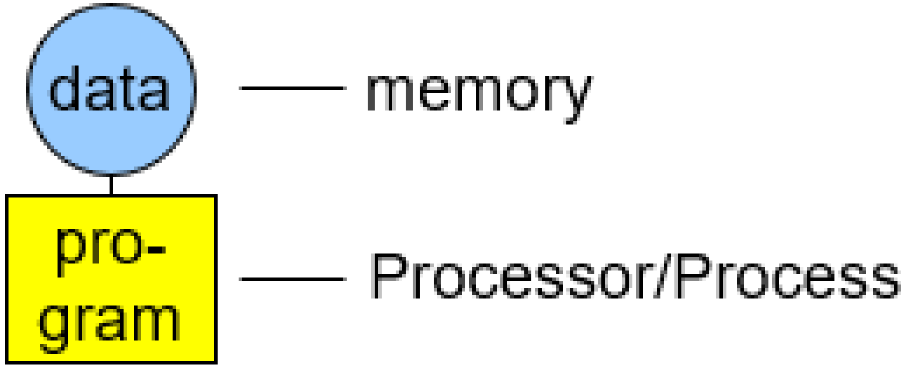
http://www.epcc.ed.ac.uk/computing/training/document_archive/mpi-course/mpi-course.pdf

- A **process** is a program performing a task on a **processor**
- Each processor/process in a message passing program runs a instance/copy of a ***program:***
  - written in a conventional sequential language, e.g., C or Fortran,
  - typically a single program operating of multiple dataset
  - the variables of each sub-program have
    - the same name
    - but different locations (distributed memory) and different data!
    - i.e., all variables are local to a process
  - communicate via special send & receive routines (***message passing***)



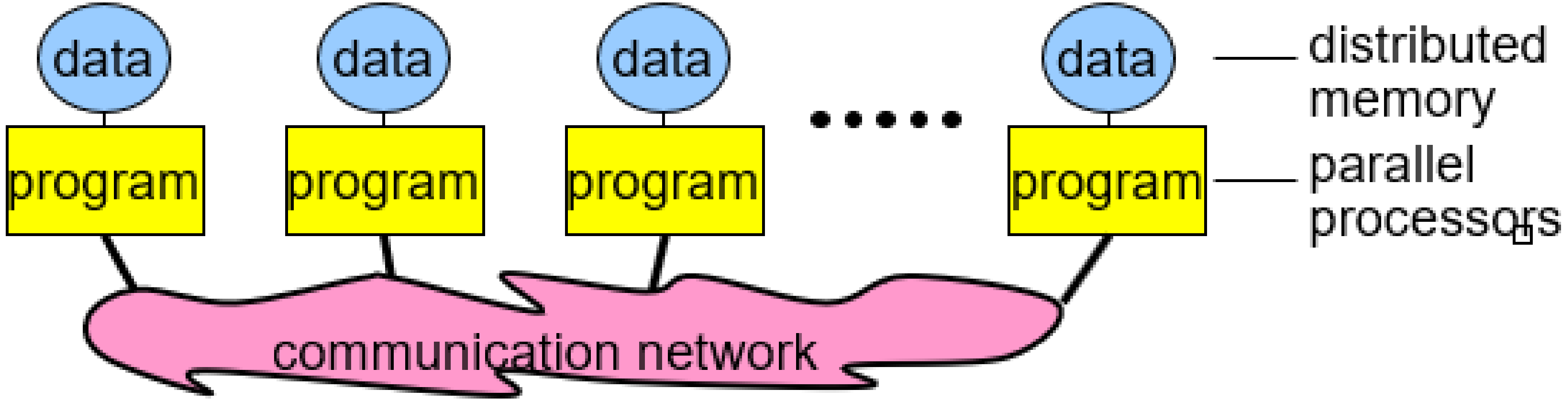
The Message-Passing Programming Paradigm

- **Sequential Programming Paradigm**



A processor may run many processes

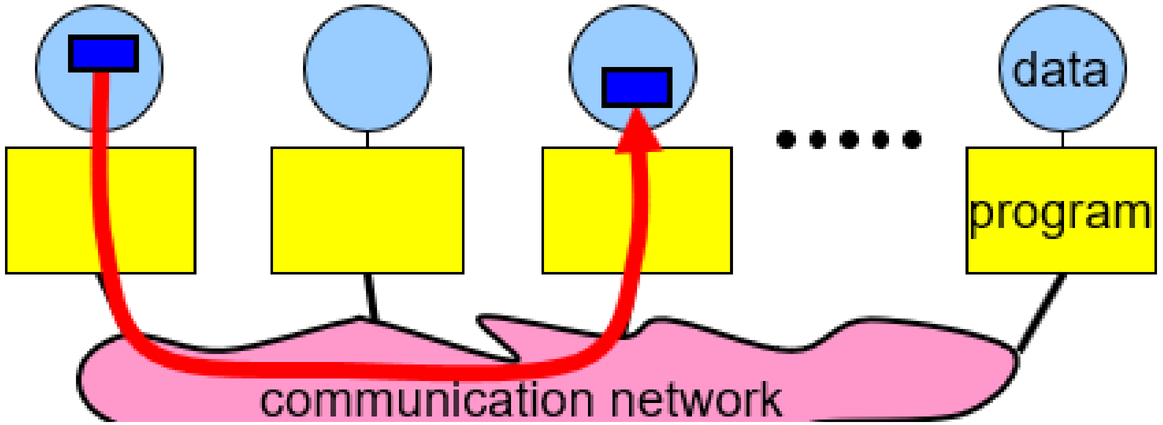
- **Message-Passing Programming Paradigm**



Message passing

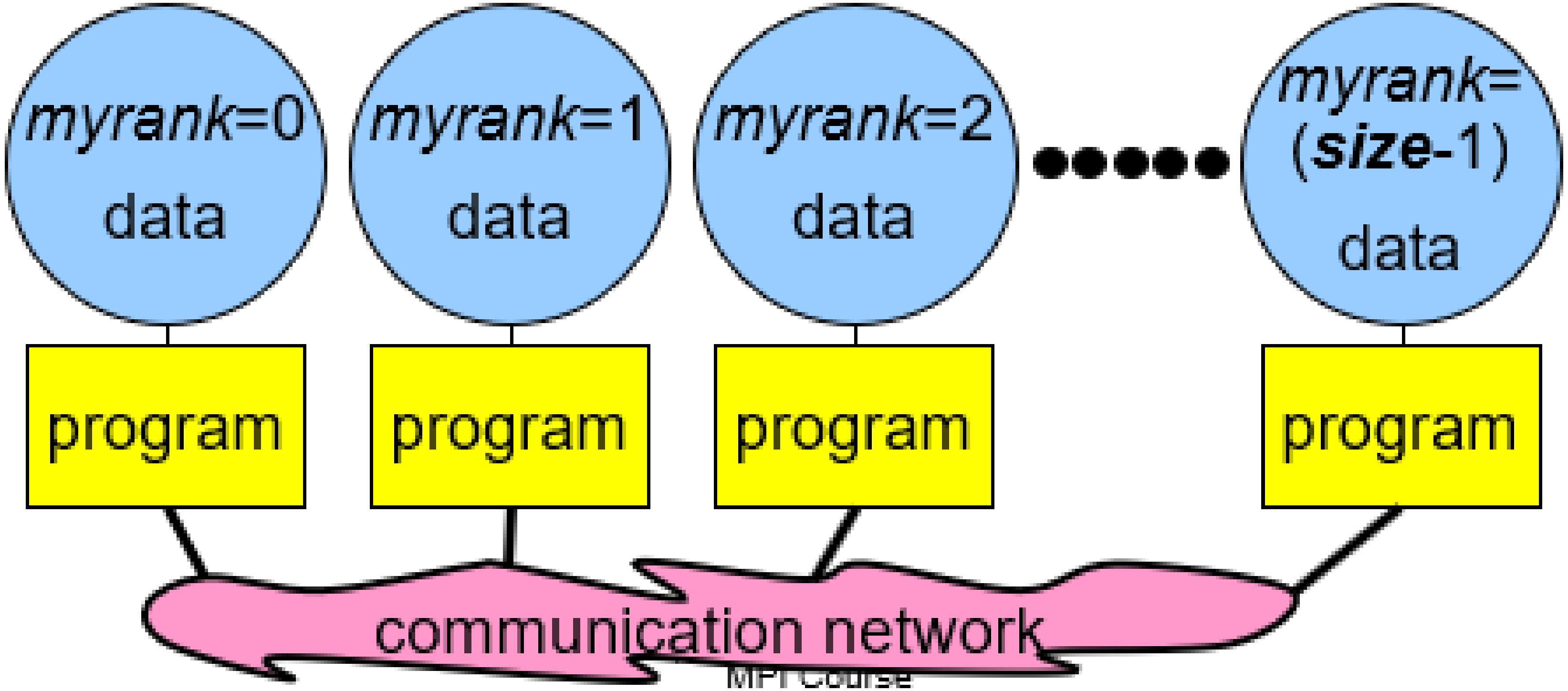
- Messages are packets of data moving between sub-programs
- Necessary information for the message passing system:

– sending process	– receiving process	i.e., the ranks
– source location	– destination location	
– source data type	– destination data type	
– source data size	– destination buffer size	



MPI_COMM_WORLD: Name of default MPI Communicator

- A communication universe (communication domain, communication group) for a group of processes
- Stored in variables of type MPI_COMM
- Communicators are used as arguments to all message transfer MPI routines
- Each process within communicator has a rank; a unique integer identifier ranging between [0, #processors − 1]
- Multiple communicators can be established in a single MPI program


myrank=0
data
program
myrank=1
data
program
myrank=2
data
program
myrank=
(size-1)
data
program
communication network

Point-to-Point Communication

Simplest form of message passing.

- One process sends a message to another.

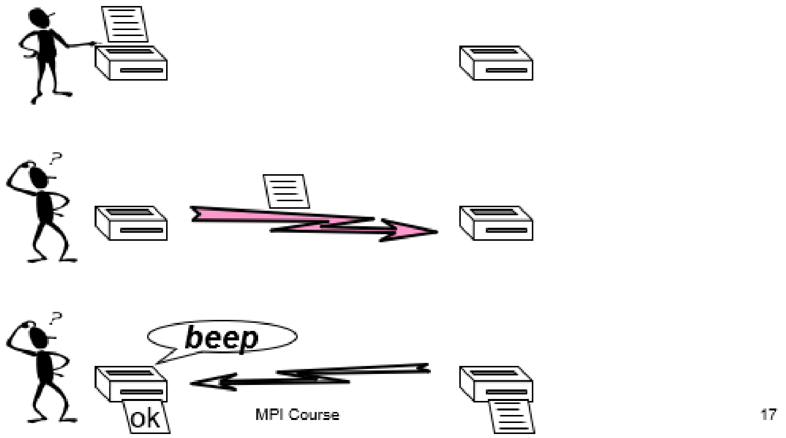
Different types of point-to-point communication:

  - synchronous send
  - asynchronous send

WORK

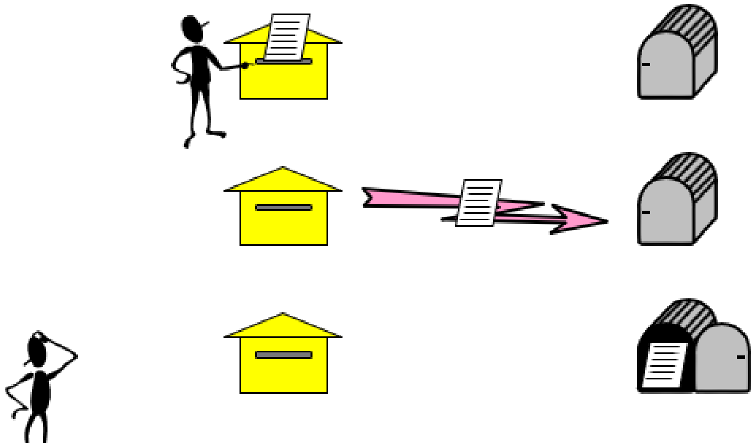
Synchronous Sends

- The sender gets an information that the message is received.
- Analogue to the *beep* or *okay-sheet* of a fax.



Asynchronous Sends

- Only know when the message has left.



Types of Point-to-Point Send/Receive Calls

- **Synchronous Transfer**: Send/Receive routines return only when the message transfer is completed. Not only does this transfer data, but it also synchronizes processes

```
MPI_Send()      // Blocking Send
MPI_Recv()      // Blocking Receive
```

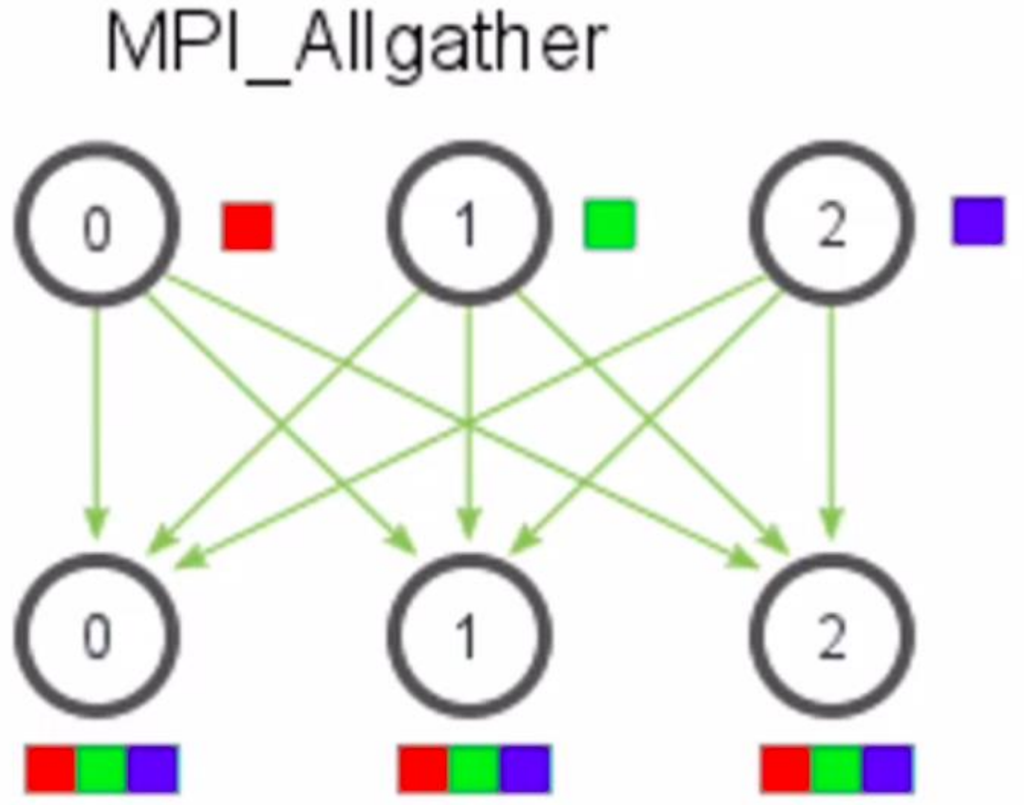
- **Asynchronous Transfers**: Send/Receive do not wait for transfer data and proceeds with execution next line of instruction. (Precaution: Do not modify the send/receive buffers)

```
MPI_Isend()     // Non-Blocking Send
MPI_Irecv()     // Non-Blocking Receive
```

Important MPI Calls

- MPI_Init(int*, char**); // Initiate an MPI Computation
- MPI_Finalize(void); // Terminate an MPI Computation
- MPI_Comm_size(MPI_COMM, int); // How many processes
- MPI_Comm_rank(MPI_COMM, int); // Who am I?
- MPI_Get_processor_name(char*, int); // What is the hostname?
- MPI_Wtime(void); // Elapsed time in seconds
- MPI_Abort(MPI_COMM); // Terminate all processes

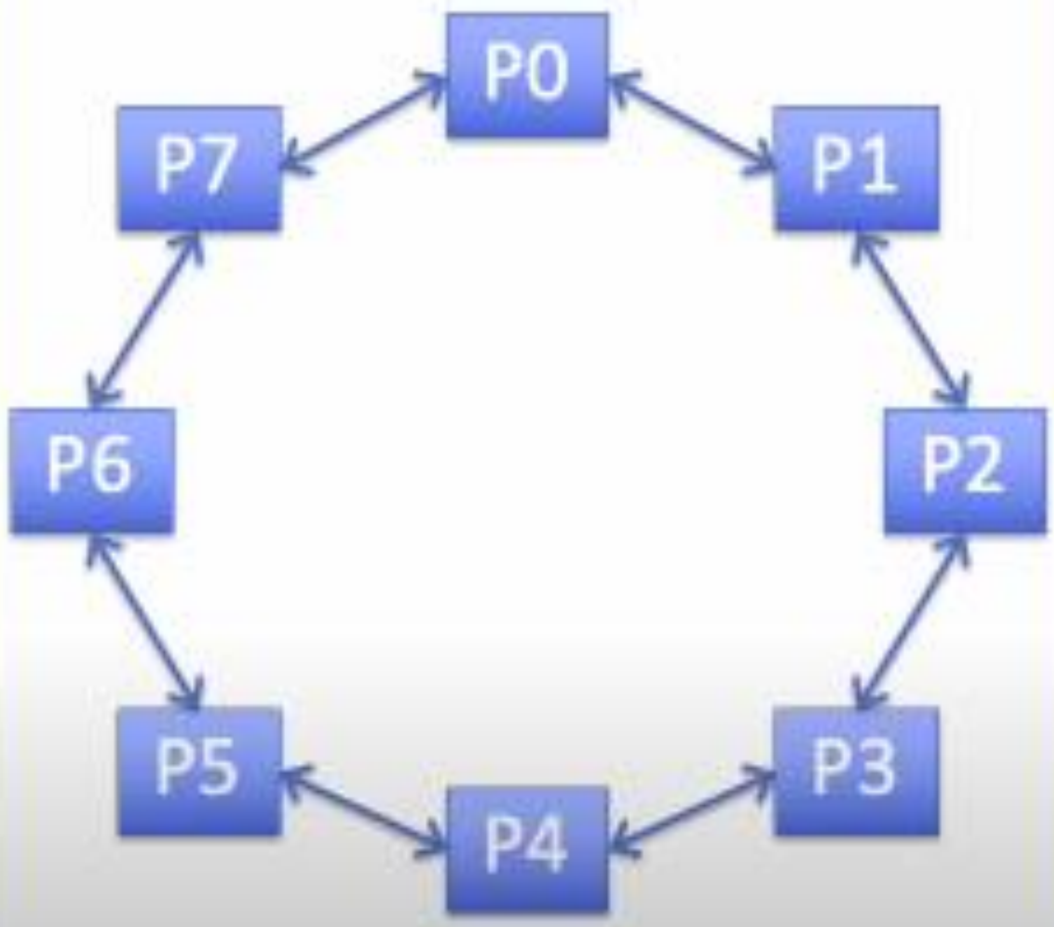
Fancier Functions: MPI_All______



List of *all* MPI routines: https://www.mpich.org/static/docs/v3.2/www3/index.htm

Steps in MPI

- Initialize MPI envirnment and services
- MPI_Init()
- gets the processes talking together
- Creating a communicator (group) optional
- defines topology of network(ring etc or might be no topology)
- Each MPI process have rank(ID) each process uses its rank to determine what to do

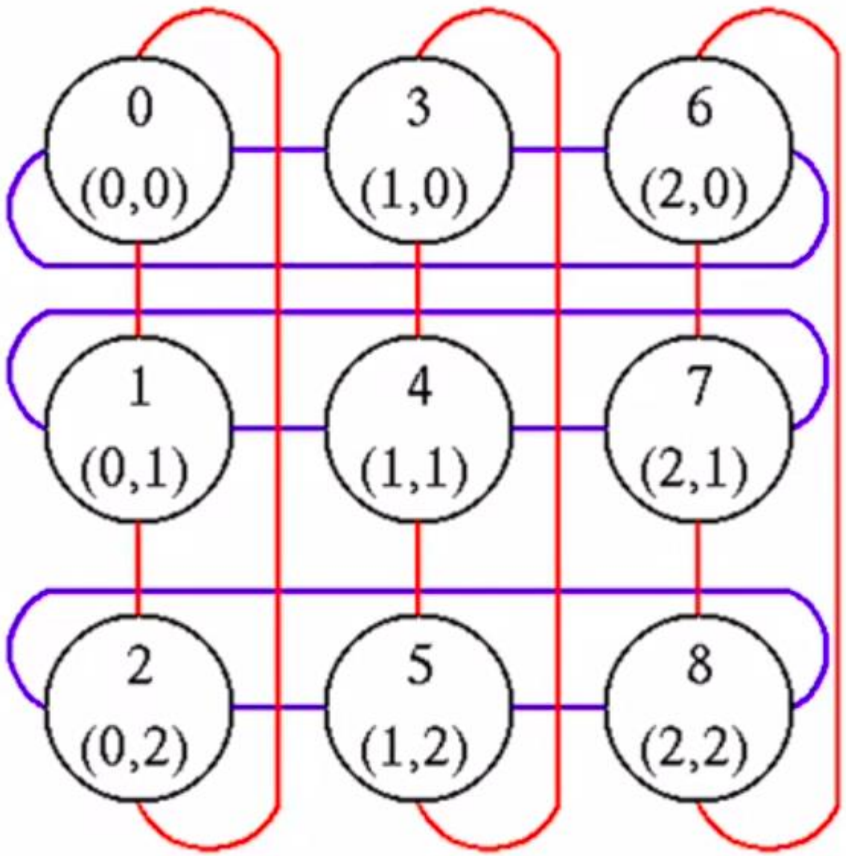


Ring Graph Topology

NP = 8

2D Cartesian Graph Topology

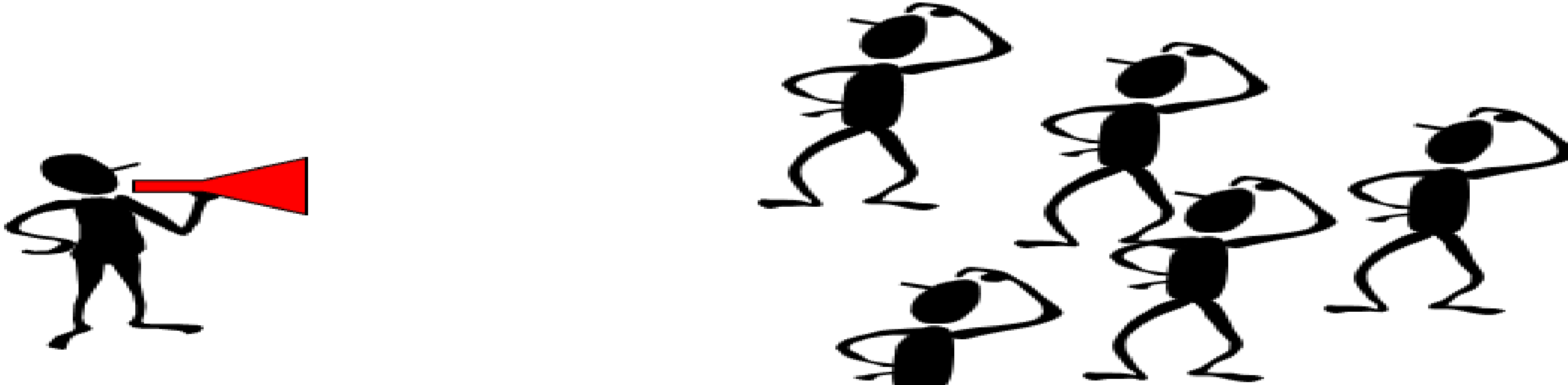
NP = 9



GROUP COMMUNICATION IN MPI

Broadcast

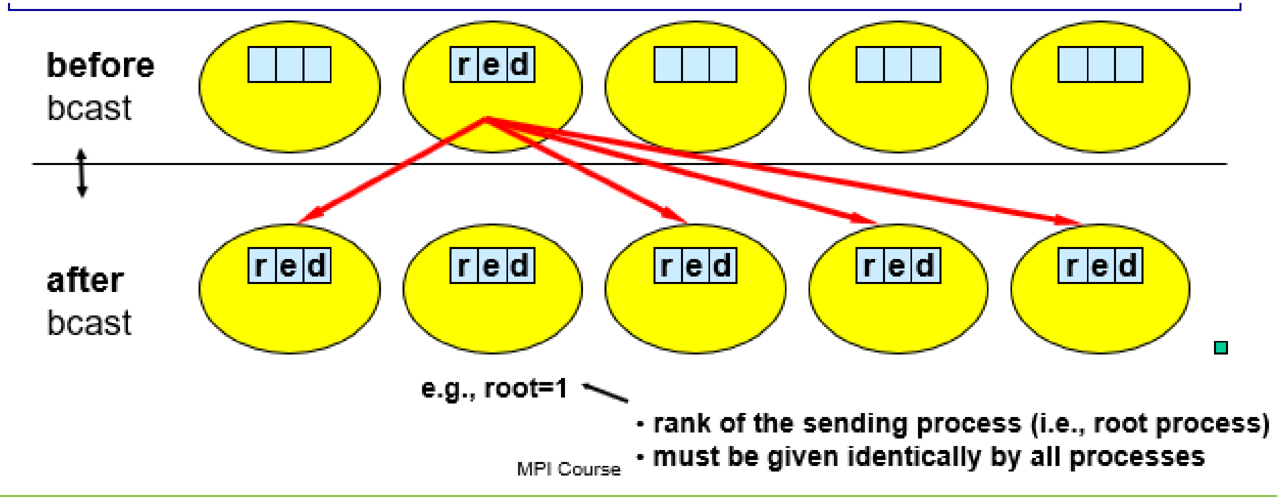
- A one-to-many communication.



Broadcast

sending data from one process to
group of other processes

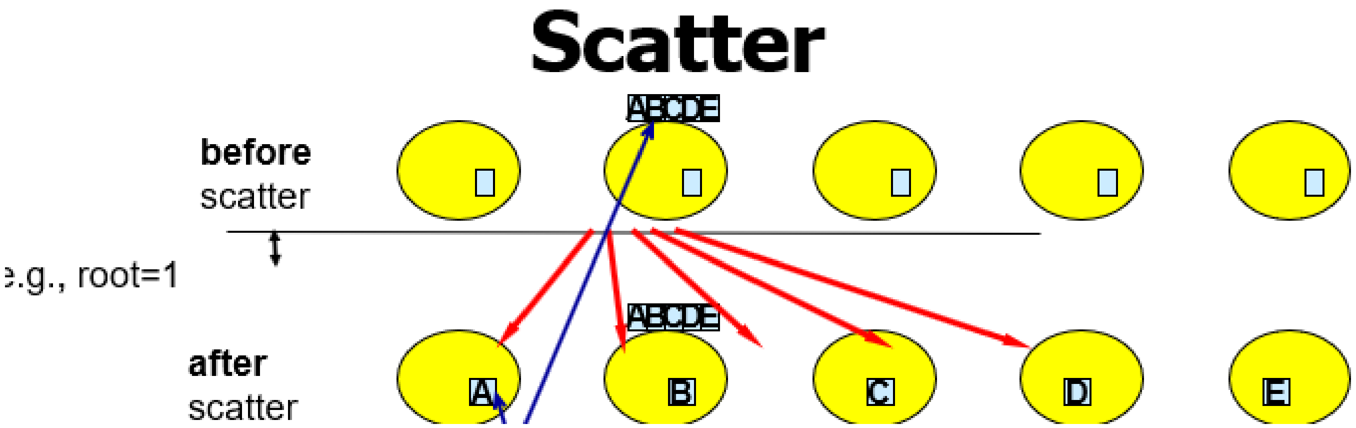
Broadcasting: MPI_Bcast()



Scatter

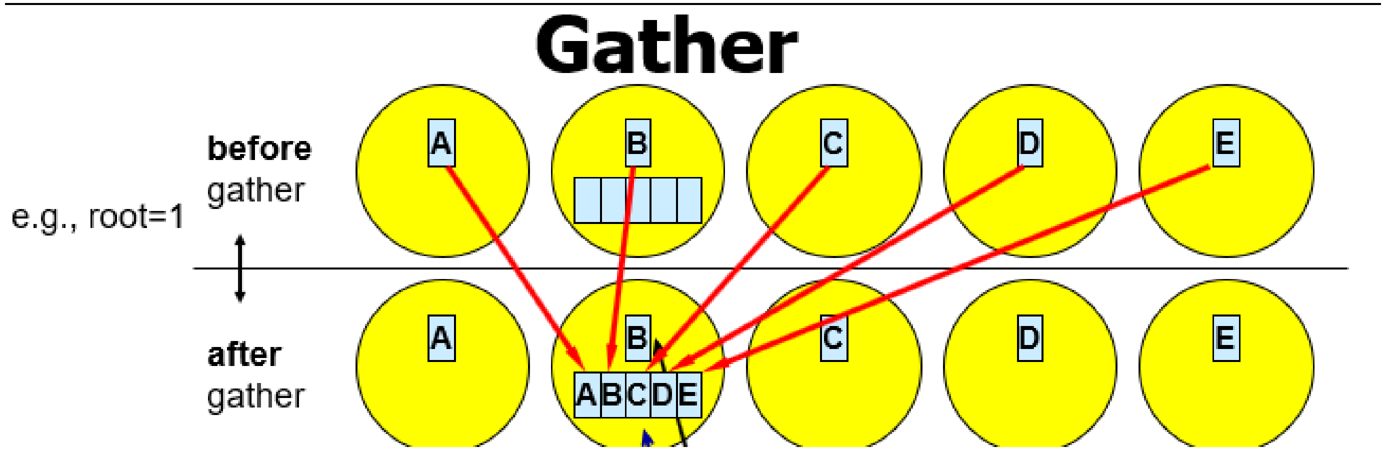
- in scatter data is divided into separate chunks and different pieces are send to different processes for execution(dividing work for speedup execution)

Scattering: MPI_Scatter()



Gather

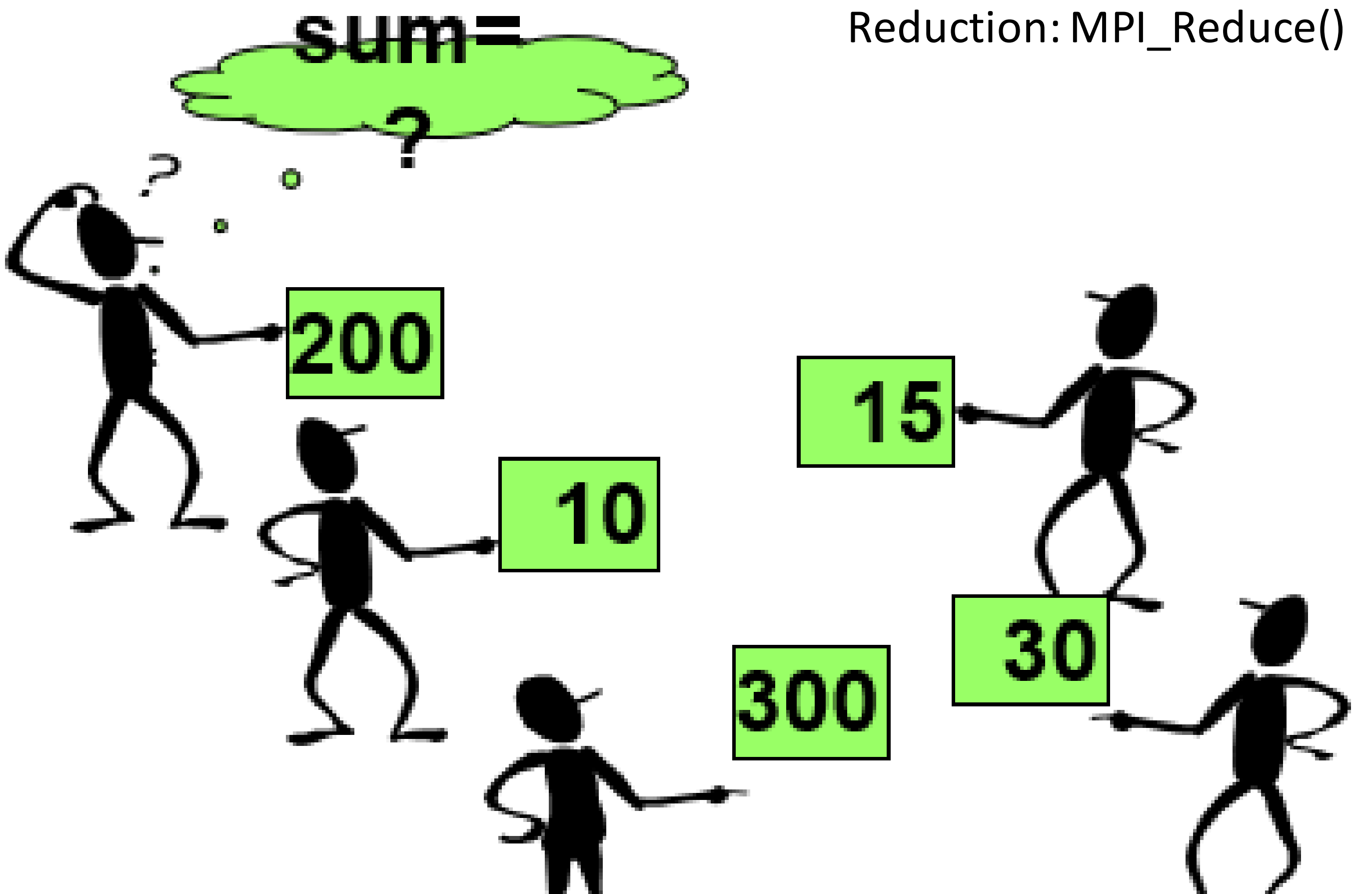
Gathering: MPI_Gather()

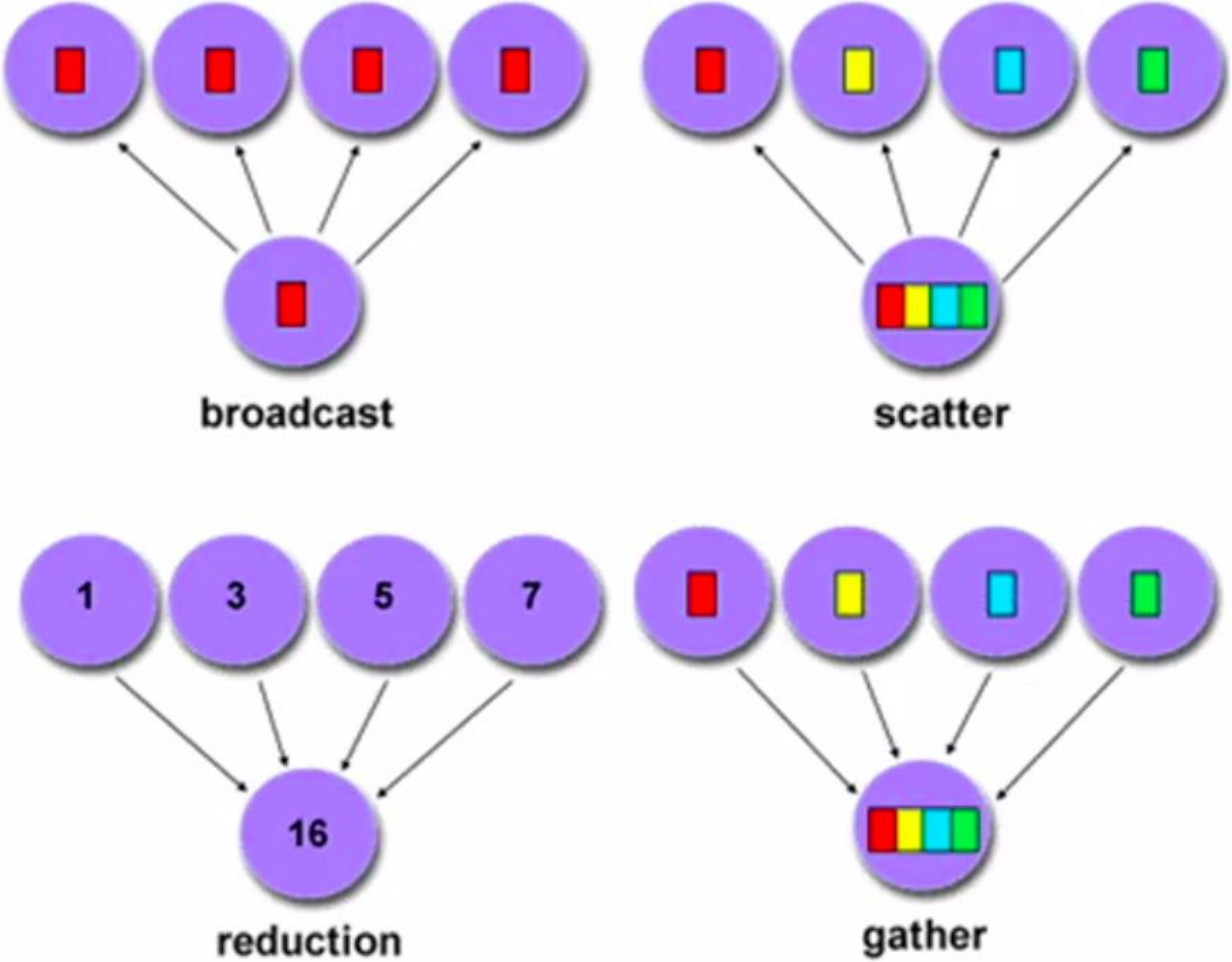


Reduction Operations

- Combine data from several processes to produce a single result.

Reduction: MPI_Reduce()





Source: https://pages.tacc.utexas.edu/~eijkhout/pcse/html/mpi-collective.html

OPENMP VS MPI

Code Feature	MPI	OpenMP Fortran
target of compile	single process	multiple threads
local variables saved variables module variables common variables	private private private private	private shared shared shared (or private)
communication	messages	shared variables
synchronization	implicit	global
I/O namespace I/O operations	independent writes unsafe	single and shared unsafe to same unit
Incompatible with Fortran standard	non-blocking calls copy-in/out, I/O	STOP, copy-in/out

PROBLEMS IN PARALLEL EXECUITION

- open MP and MPI all have some similarities like they all are concurrent up to some extent, there are some of problems with all of them

RACE CONDITION:

- If we do not know the order of execution ,different threads could access shared data and try to change it, leading to unexpected results.

THREAD SAFETY

Goals:

- avoid race condition
- avoid deadlock(two or more threads waiting for something from one of the other threads)
- avoid unexpected behavior

Tools:

- Mutually exclusive locks : only one thread access a resource at a time.
- Atomic operations: operations that can not be interrupted by other threads

PARALLEL I/O

- If I want to read and write files from local disk. How would I do this?
- Normally reading files from disc in multiprocessing or multi threading is dangerous .

- Like what if one thread overwrites what another thread just wrote?
- What if the data is all out of order?

PARALLEL I/O

SIMPLE SOLUTION:

- choose one thread/process to do all the I/O and have the other threads/processes wait as necessary
- not great since a lot of threads aren't working

MPI can perform collective I/O

- processes work together to read/write files in parallel
- Requires a shared filesystem

```c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv)
{
    // Initialize the MPI environment
    MPI_Init( NULL, NULL );

    // Get the number of processes
    int world_size;
    MPI_Comm_size( MPI_COMM_WORLD, &world_size );

    // Get the rank of the process
    int world_rank;
    MPI_Comm_rank( MPI_COMM_WORLD, &world_rank );

    // Get the name of the processor
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name( processor_name, &name_len );

    // Print off a hello world message
    printf( "Hello world from processor %s, rank %d out of %d processors\n",
            processor_name, world_rank, world_size );

    // Finalize the MPI environment.
    MPI_Finalize();
}
```

Library and code execution of mpi

- for mpi: setup :
- sudo apt-get update
- sudo apt-get upgrade
- sudo apt-get install libopenmpi-dev
- sudo apt-get install openmpi-bin
- compile:
- mpicc filename.c -o filename
- run : mpirun -np 4 ./filename

Output

```
usman@usman-VirtualBox:~/Downloads$ mpicc tah.c -o tah
usman@usman-VirtualBox:~/Downloads$ mpirun -np 4 ./tah
hello world from processer usman-VirtualBox, rank 1 out of 4 processors
hello world from processer usman-VirtualBox, rank 3 out of 4 processors
hello world from processer usman-VirtualBox, rank 0 out of 4 processors
hello world from processer usman-VirtualBox, rank 2 out of 4 processors
```

```
usman@usman-VirtualBox:~/Downloads$ mpirun -np 8 --oversubscribe  ./tah
hello world from processer usman-VirtualBox, rank 7 out of 8 processors
hello world from processer usman-VirtualBox, rank 3 out of 8 processors
hello world from processer usman-VirtualBox, rank 0 out of 8 processors
hello world from processer usman-VirtualBox, rank 2 out of 8 processors
hello world from processer usman-VirtualBox, rank 6 out of 8 processors
hello world from processer usman-VirtualBox, rank 1 out of 8 processors
hello world from processer usman-VirtualBox, rank 5 out of 8 processors
hello world from processer usman-VirtualBox, rank 4 out of 8 processors
```