Practical Parallel Programming Paradigms

at

Inter-University Accelerator Centre (IUAC), New Delhi

4-day school on Scientific Computing, Artificial Intelligence and Machine Learning

by

Rajeev Wankar

wankarcs@uohyd.ac.in

School of Computer and Information Sciences &

Centre for Modelling, Simulation and Design (CMSD)

University of Hyderabad (IoE institute)

Agenda

- Brief Introduction to Parallel Computing
- Parallel Computing Paradigms
- Multi-Computers programming
- Multi-Processor Programming
- Discussions

Why do we need powerful computers?

Some Challenging Computations

- **Science**
  - Global climate modeling
  - Astrophysical modeling
  - Biology: genomics; protein folding; drug design
  - Computational Chemistry
  - Computational Material Sciences
- **Engineering**
  - Crash simulation
  - Semiconductor design
  - Earthquake and structural modeling
  - Computation fluid dynamics (airplane design)
  - Combustion (engine design)
- **Business**
  - Financial and economic modeling
  - Transaction processing, web services and search engines
- **Defense**
  - Nuclear weapons -- test by simulation
  - Cryptography

Galaxy Formation

Planetary Movments

Climate Change

Rush Hour Traffic

Plate Tectonics

Weather

Picture Source: Internet

Simulation

- Traditional scientific and engineering paradigm:
  1) Do theory or paper design.
     (First pillar of Science)
  2) Perform experiments or build system.
     (Second pillar of Science)
- Limitations:
  - Too difficult -- build large wind tunnels.
  - Too expensive -- build an experimental passenger jet.
  - Too slow -- wait for climate evolution.
  - Too dangerous -- weapons, drug design experiments.

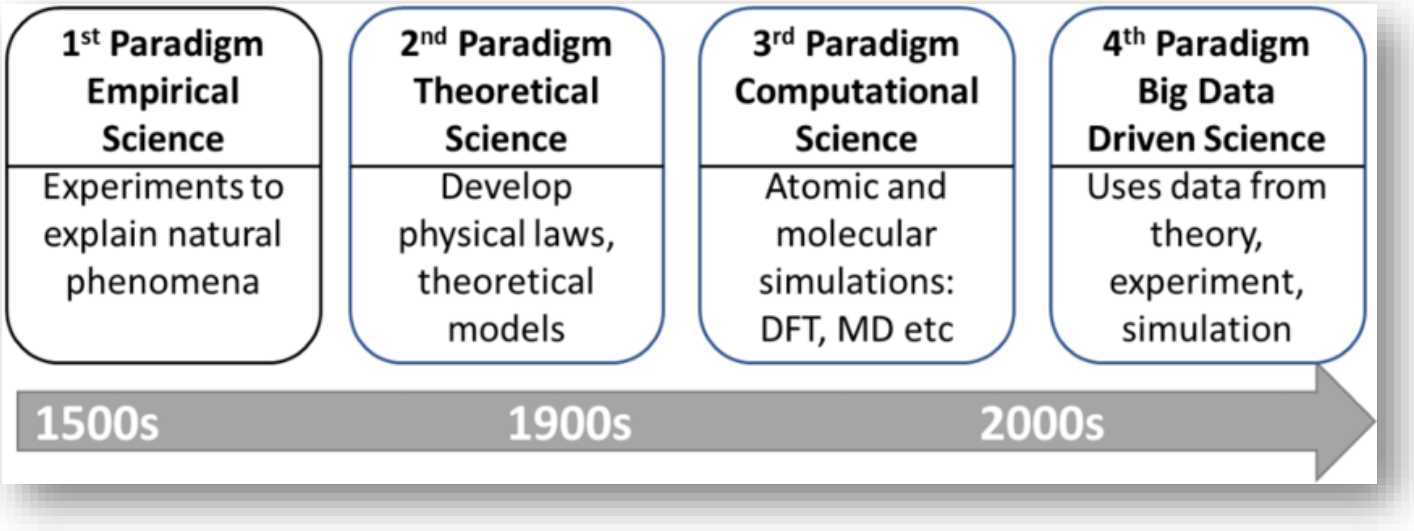
Simulation

- Computational science paradigm: (Third pillar of Science)
  3) Use high performance computer systems to simulate the phenomenon.
     - Based on known physical laws and efficient numerical methods.

Modeling *is a way to create a virtual representation of a real-world system that includes software and hardware.*
Simulation *is used to evaluate a new design, diagnose problems with an existing design, and test a system under conditions that are hard to reproduce in an actual system.*

beyond

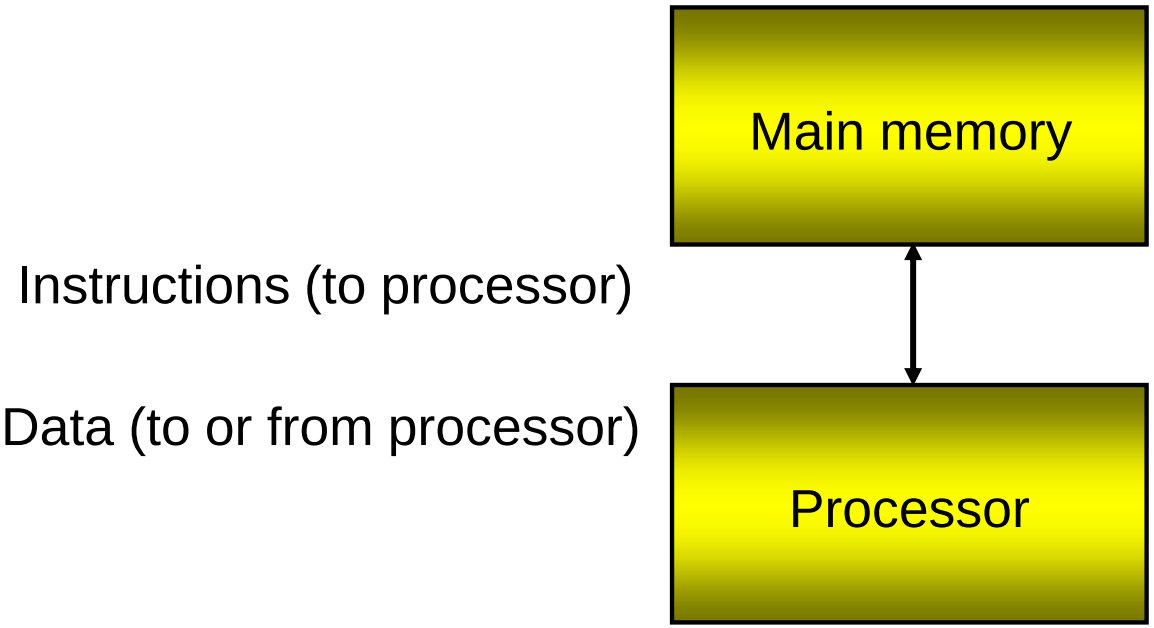
- What is forth pillar of Science?
- Wisdom is the ability to know what is true or right, common sense or the collection of one's knowledge.



Hey, A. J. G., Tansley, S., & Tolle, K. M. (2009). The fourth paradigm: data-intensive scientific discovery. Redmond, WA: Microsoft Research.

Conventional Computer

Consists of a processor executing a program stored in a (main) memory:

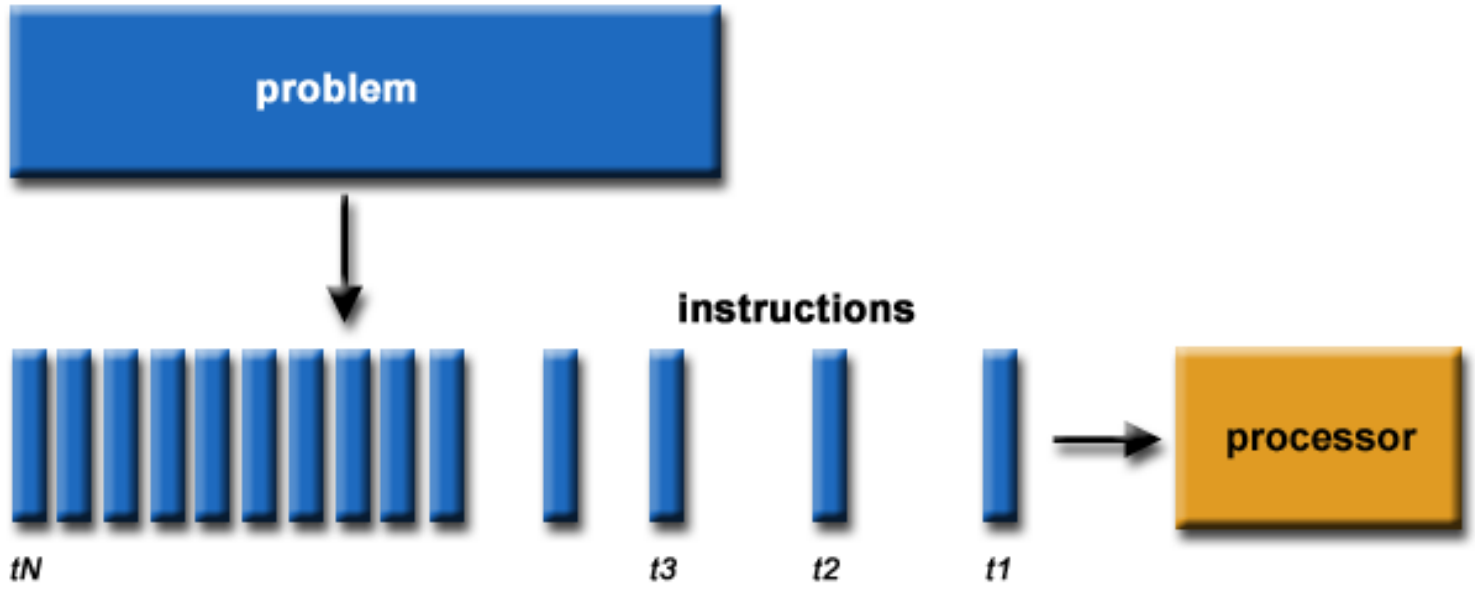


Each main memory location located by its address within a single memory space.

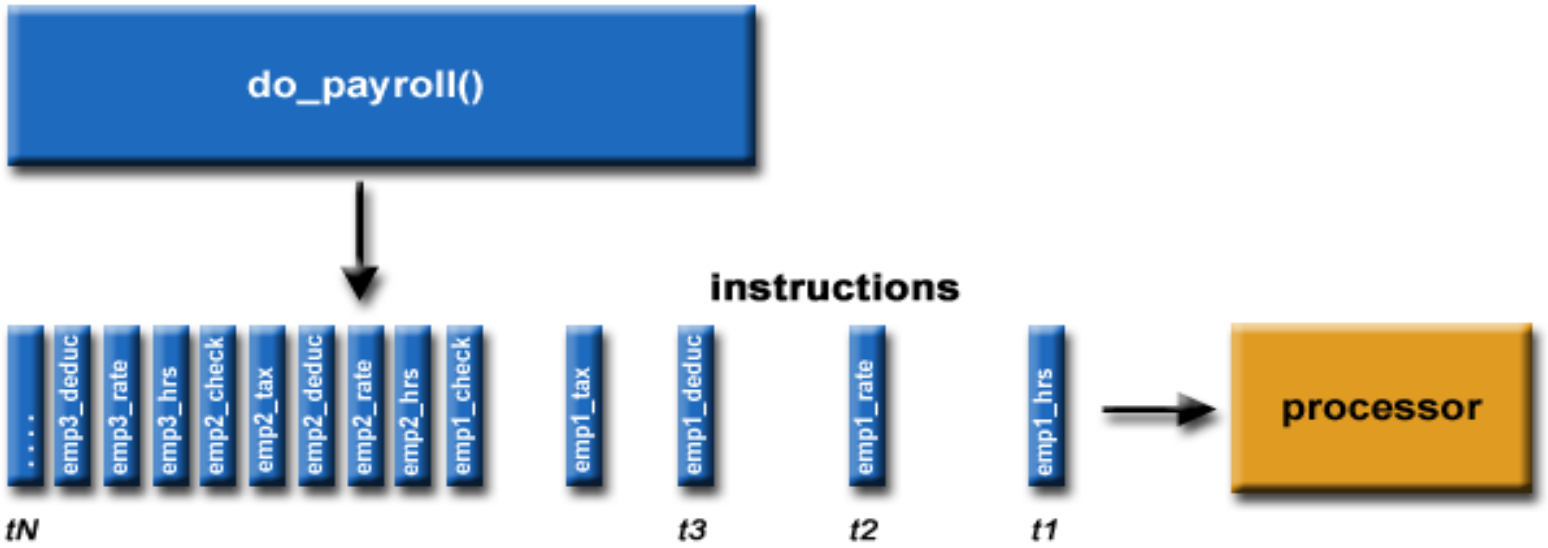
Serial Computing

- Traditionally, software has been written for ***serial*** computation:
  - A problem is broken into a discrete series of instructions
  - Instructions are executed sequentially one after another
  - Executed on a single processor
  - Only one instruction may execute at any moment in time

Serial Computing



Serial Computing



High Performance Computing (HPC)

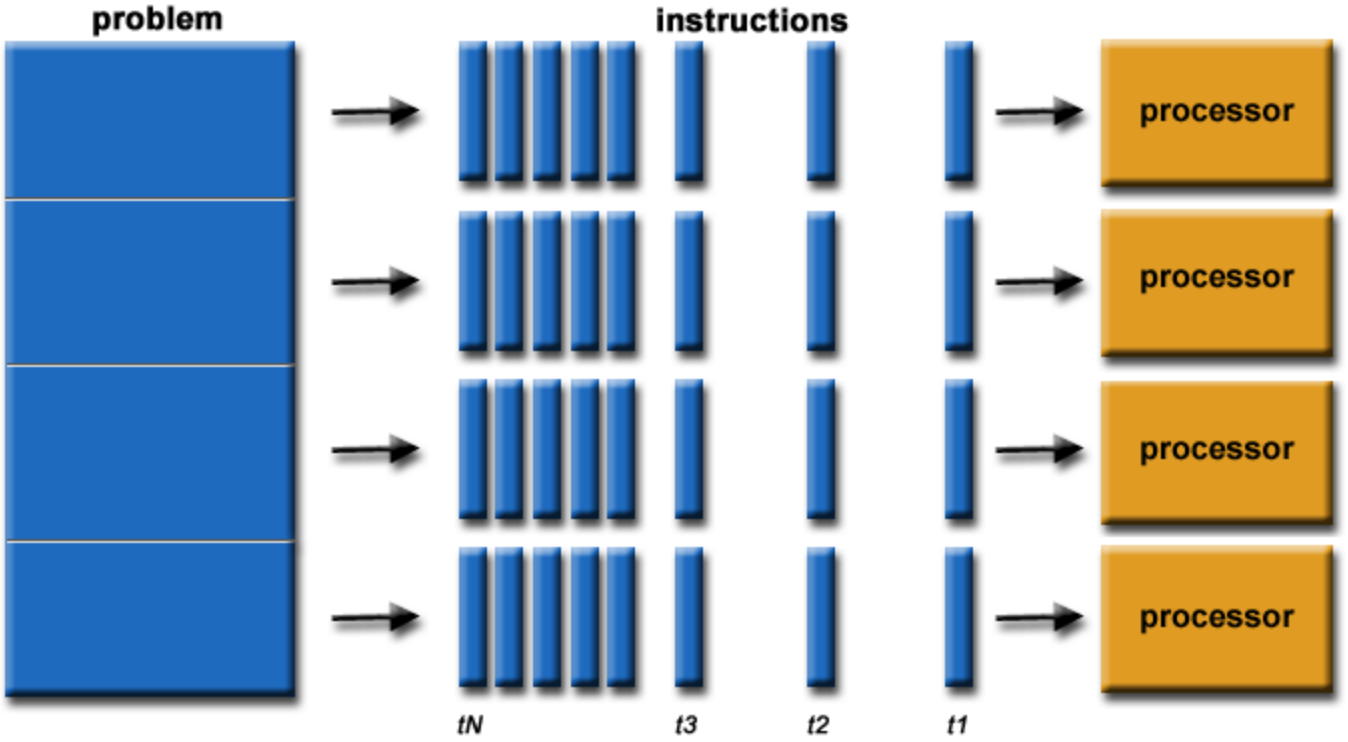
- *Traditionally,* achieved by using the multiple computers together - parallel computing.

- Simple idea! -- Using multiple computers (or processors) simultaneously should be able to solve the problem faster than a single computer.

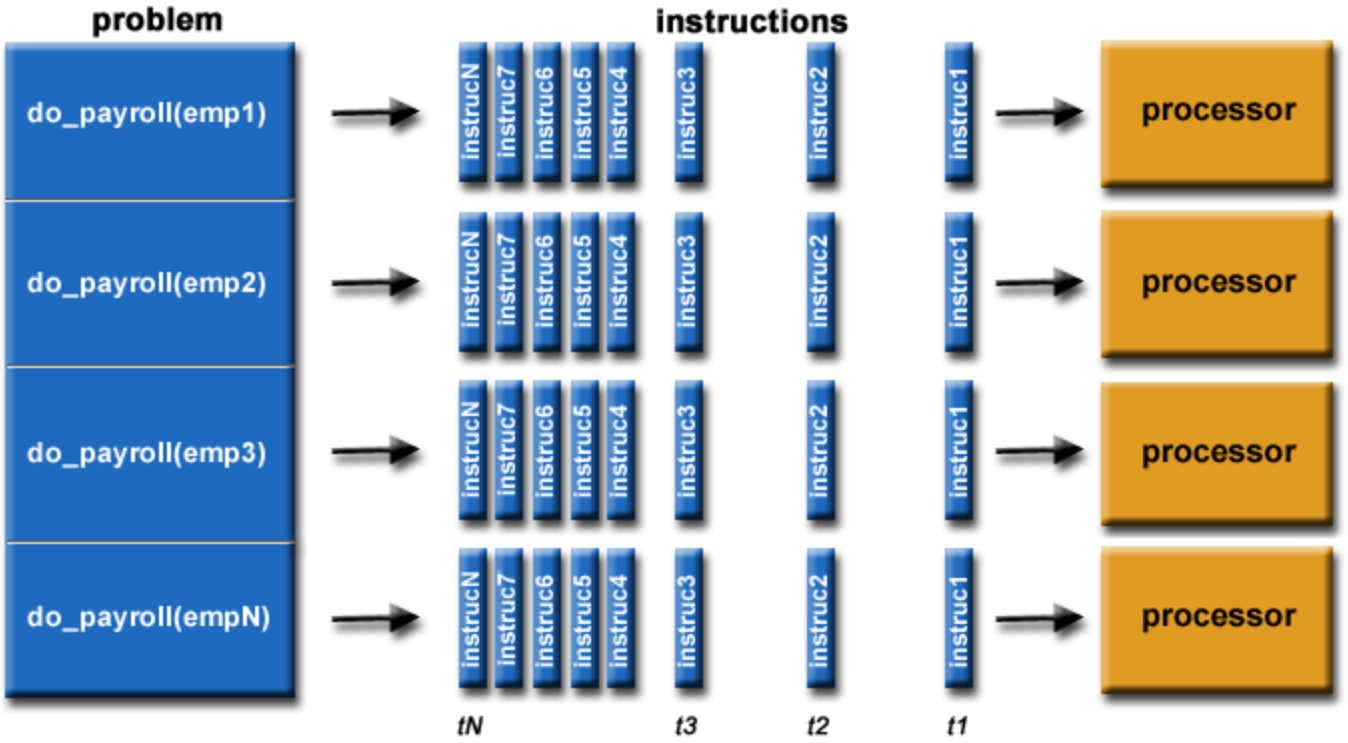
Using multiple computers or processors

- Key concept - dividing problem into parts that can be computed simultaneously.
- Parallel programming – solution of a problem using multiple processes or multi computers.
- Concept very old (60 years).

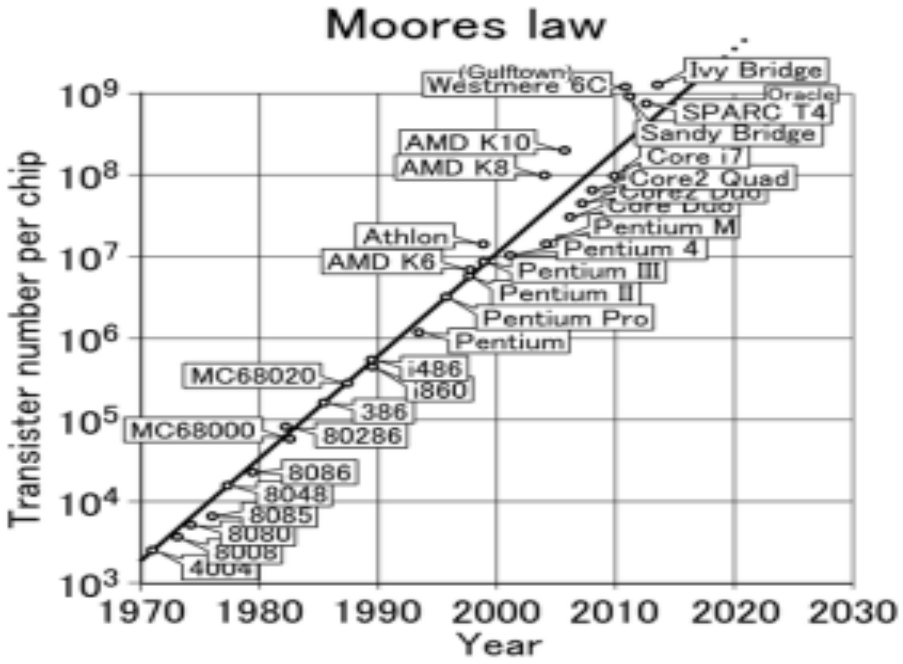
Parallel Processing

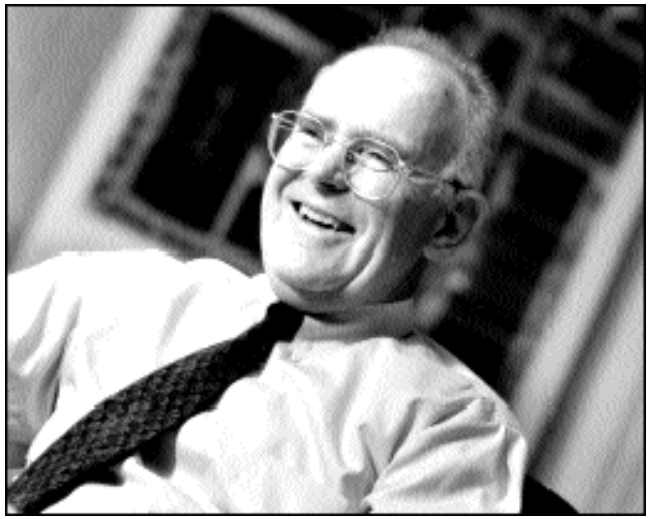


Parallel Processing



Technology Trends: Microprocessor Capacity





Moore's Law: #transistors/chip doubles every 18 months

Microprocessors have become smaller, denser, and more powerful.

Gordon Moore (co-founder of Intel) predicted in 1965 that the transistor density of semiconductor chips would double roughly every 18 months.

Slide source: Jack Dongarra

Performance improvement of the single processor

- ***Architectural*: *(Work Hard)*** amt. of work performed per instruction cycle, bit parallel memory, bit parallel arithmetic, cache, instruction pipelining, multiple functional units etc.

- ***Technological*: *(Work Smart)*** reduce time needed per instruction cycle. Speed of the electronic device is limited by the speed of the light (It travels approximately a foot in a nanosecond)

"Automatic" Parallelism in Modern Machines

- Bit level parallelism
  - within floating point operations, etc.
- Instruction level parallelism
  - multiple instructions execute per clock cycle
- Memory system parallelism
  - overlap of memory operations with computation
- OS parallelism
  - Multiple threads run in parallel on SMPs

There are limits to all of these -- for very high performance, user must identify, schedule and coordinate parallel tasks

High Performance Computing (HPC)

- *Traditionally,* achieved by using the multiple computers together - parallel computing
- Simple idea! Using multiple computers (or processors) simultaneously should be able to solve the problem faster than a single computer
- Dividing problem into parts that can be computed simultaneously

Parallel Processing Terminology

- ***Parallel Processing*** is information processing that emphasizes on the concurrent manipulation of the data elements belonging to one or more processes solving a single problem.

- A multiple processor computer capable of ***parallel processing*** is parallel computer.

Parallel Processing Terminology

- Parallelism is a condition that arises when at least two processes are executing simultaneously
- Concurrency is a condition that exists when at least two processes are making progress.
- Concurrency is a more generalized form of parallelism that can include time-slicing as a form of virtual parallelism.
- In a multithreaded process on a single processor, the processor can switch execution resources between threads, resulting in concurrent execution.

Parallel Processing Terminology

Parallelism can be achieved by

Control parallelism(functional): Applying different operations to different data elements simultaneously. **Pipelining** is a special case of it in which the computation is divided in to segments or stages.

Ex. Assembly line of any car manufacturing process.

Data parallelism(domain): Multiple functional units apply same operation simultaneously to elements of a data set.

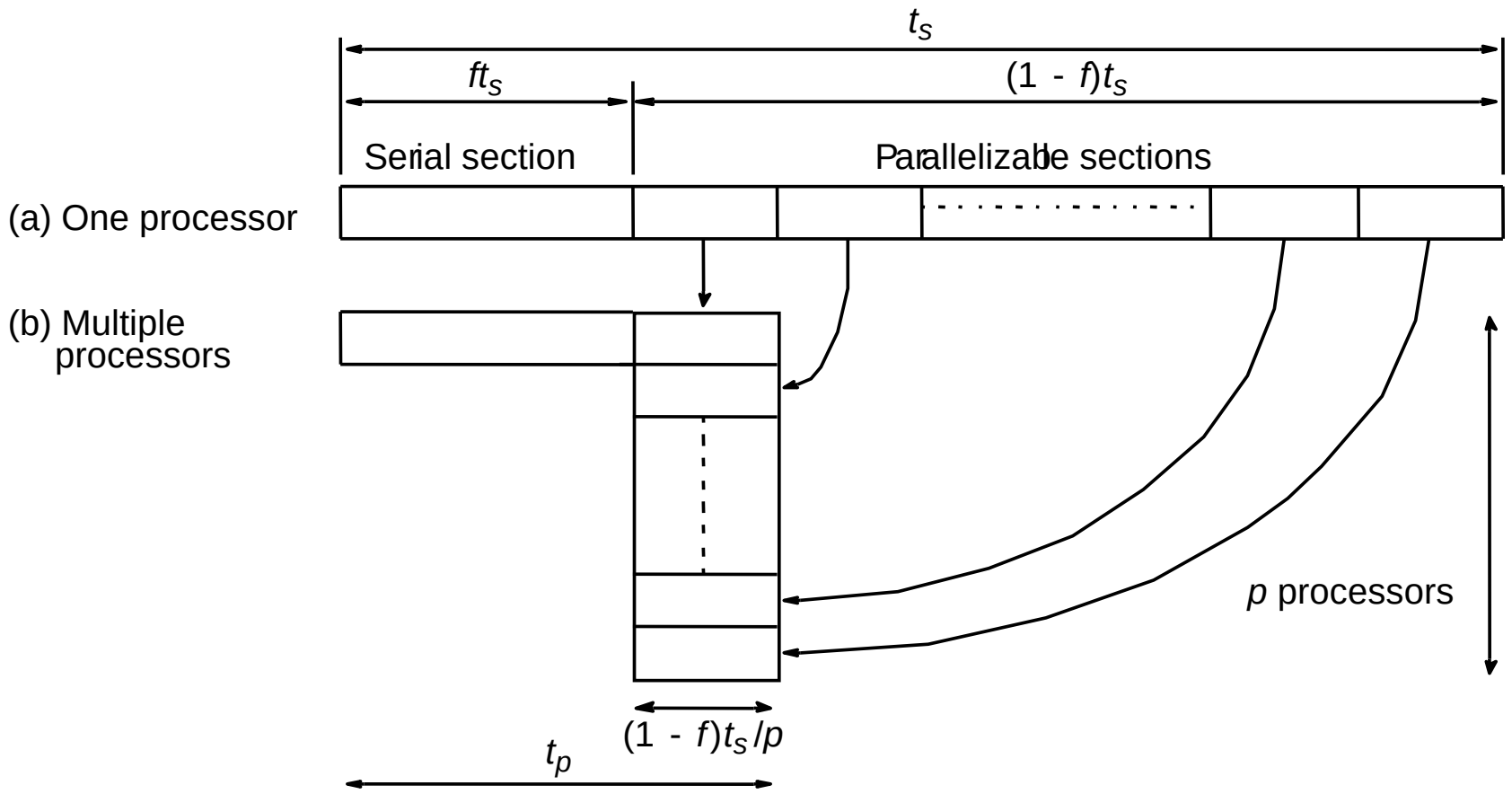
Ex. Matrix addition.

Amdhal's law: Let f be a fraction of operations in a computation that must be performed sequentially, where $0 \leq f \leq 1$. The maximum speedup ***S*** achievable by a parallel computer with ***P*** processors performing the computation is

$$S \leq \frac{1}{f + (1 - f)/P}$$

Maximum Speedup

Amdahl's law

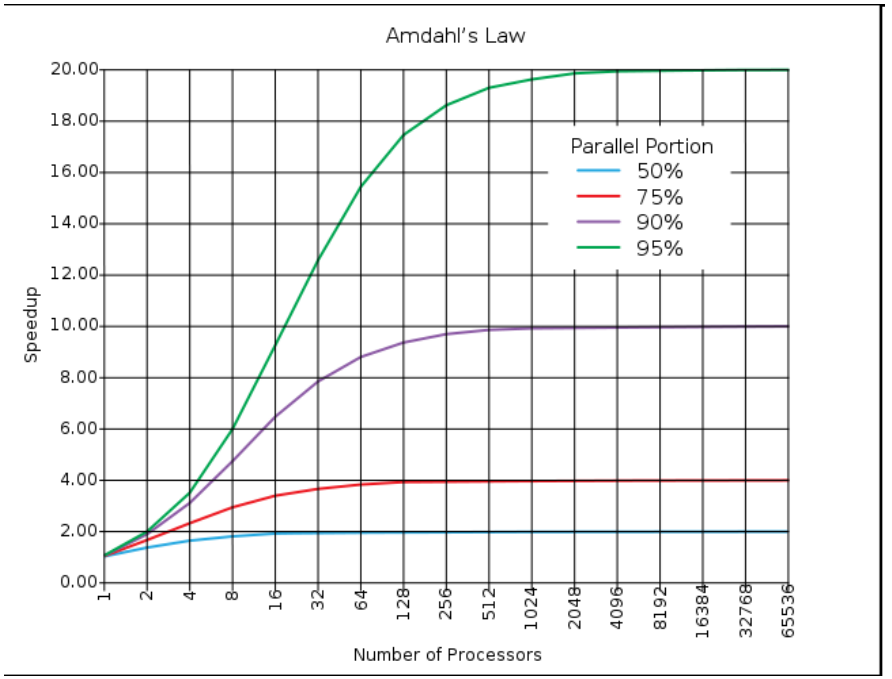


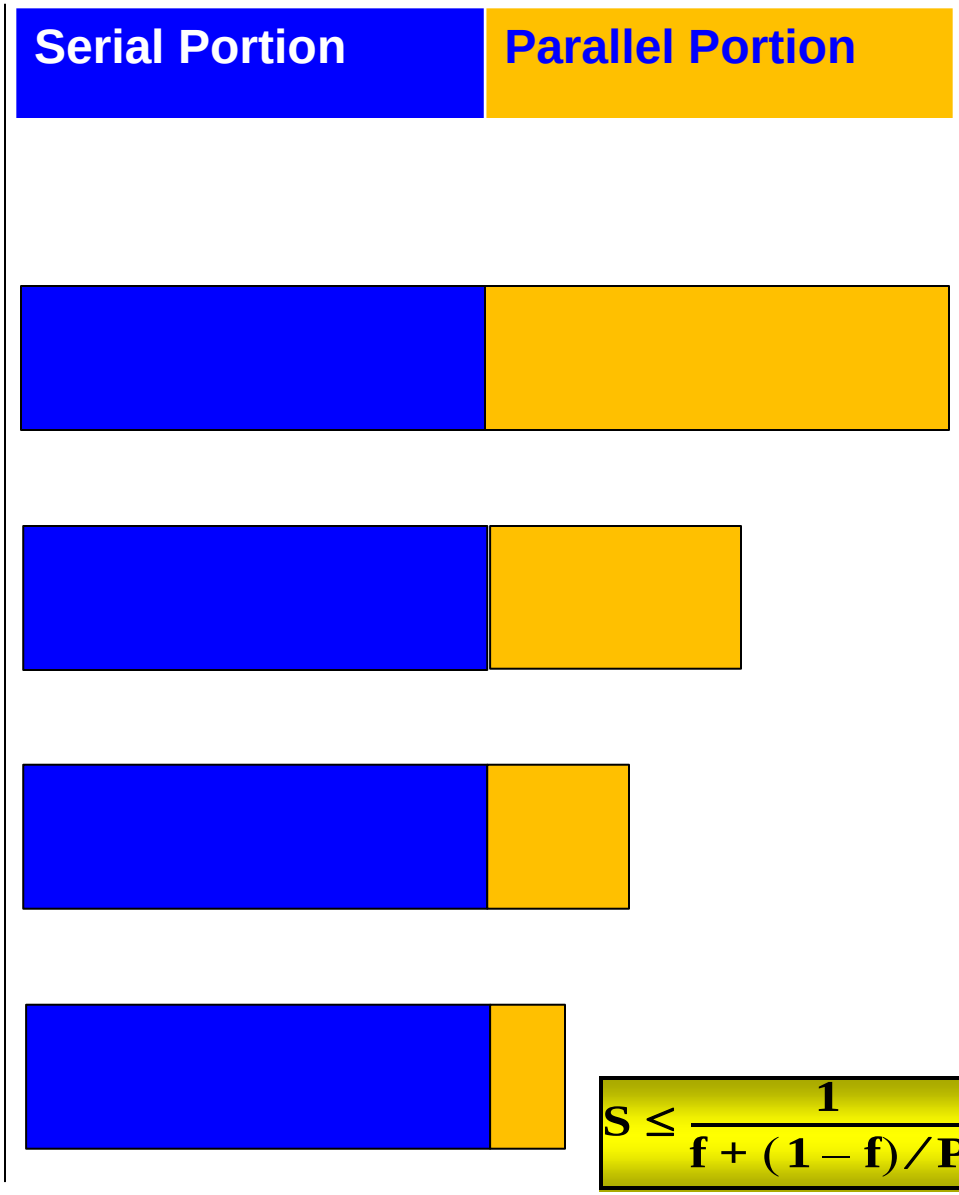
Example 1

- If 95% of a program's execution time occurs inside a loop that can be executed in parallel. What is the maximum speedup we should expect from a parallel version of the program executing on 8 CPUs?

$$S \leq \frac{1}{f + (1 - f)/P}$$

$$S \leq \frac{1}{0.05 + (1 - 0.05)/8} \cong 5.9$$





$$S \leq \frac{1}{f + (1 - f)/P}$$

Processors	Speedup
1	1
2	1.33
4	1.6
8	1.8

If Amdahl's Law is applied

"The speedup is always limited by the sequential portion of the code"

Amdahl's Law Continue…

- Gustafson's law addresses the shortcomings of Amdahl's law, which cannot scale to match availability of computing power as the machine size increases.
- It removes the fixed problem size or fixed computation load constrains on the parallel processors: instead, he proposed a fixed time concept which leads to scaled speed up for larger problem sizes.

Gustafson's Law

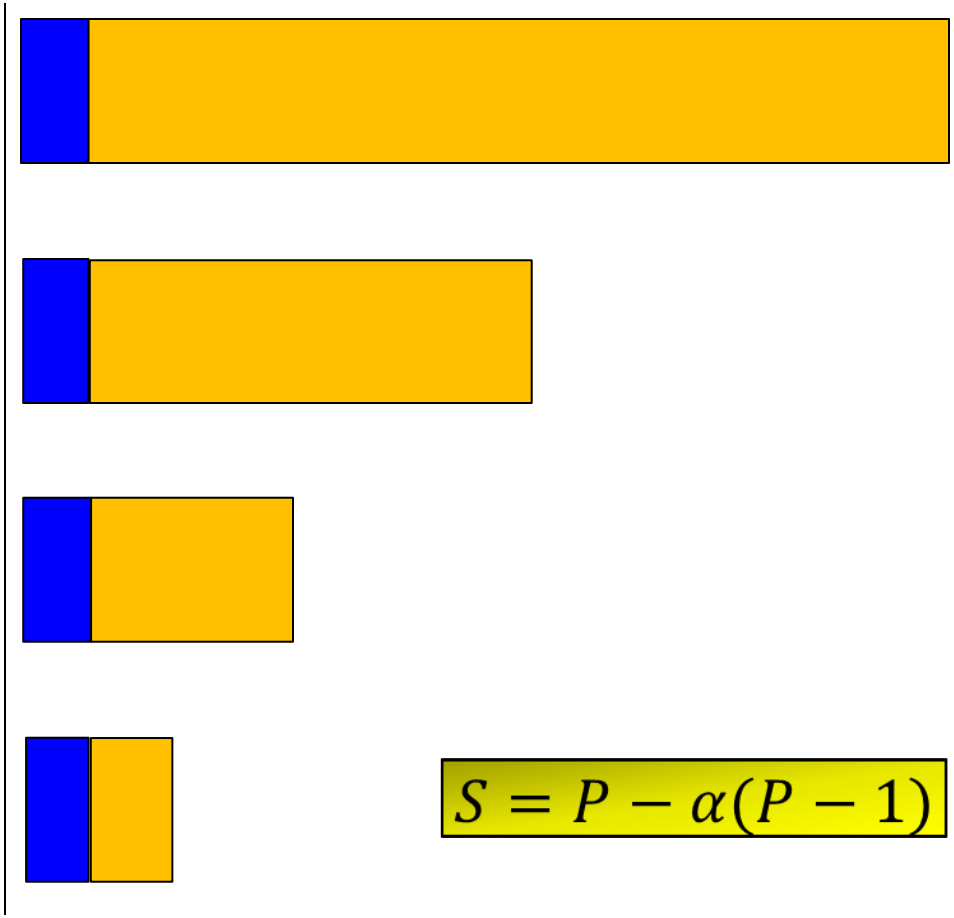
- **Gustafson's Law** (also known as **Gustafson-Barsis' law, 1988**) states that *any sufficiently large problem can be efficiently parallelized*. Gustafson's Law is closely related to Amdahl's law, which gives a limit to the degree to which a program can be sped up due to parallelization.

$$S = P - \alpha(P - 1)$$

where *P* is the number of processors, *S* is the speedup, and α the sequential part of the process.

We require larger problem for large processors

Even if it still limited by serial portion,
It is less important

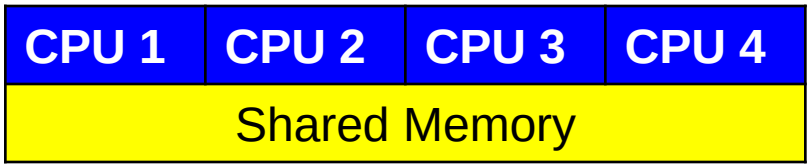


$$S = P - \alpha(P - 1)$$

Processors	Speedup
1	1
2	1.8
4	3.8
8	4.5

Commonly used paradigms

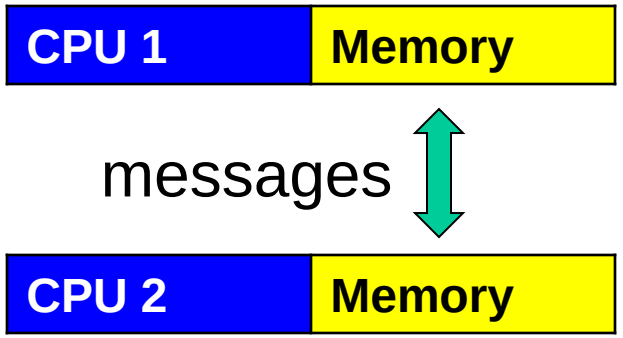
- Shared Memory (Multi-Threading): In this paradigm, multiple threads of execution share a common address space, allowing them to directly access and modify shared data.
- Java and C/C++ (with POSIX threads or OpenMP directives).



It simplifies programming, but developers need to manage synchronization and data consistency explicitly.

Commonly used paradigms

- Message Passing: In the message passing paradigm, parallel tasks communicate by explicitly sending and receiving messages.
- This communication can happen between processes
- Popular message passing libraries and frameworks include MPI, Apache Kafka and RabbitMQ.



Commonly used paradigms

- Data Parallelism: Data parallelism involves breaking down large data sets into smaller chunks and processing them simultaneously using the same computation on different processors or cores.
- This approach is well-suited for tasks that can be divided into independent units (DS) of work, such as array operations.
- Data parallelism is often supported by libraries and frameworks like OpenMP, CUDA (for GPU programming), OpenACC and frameworks like Apache Hadoop and Apache Spark.

Commonly used paradigms

- Task Parallelism: Task parallelism focuses on dividing the program into smaller tasks that can be executed independently or asynchronously.
- Each task is assigned to a separate processing unit for execution. This paradigm is commonly used in frameworks like Intel Threading Building Blocks (TBB), Microsoft's Task Parallel Library (TPL), and frameworks like Akka.

Commonly used paradigms

- Pipelining: Pipelining is a technique where multiple stages of a computation are overlapped and executed concurrently.
- Each stage performs a specific operation on the data and passes it to the next stage.
- This approach is frequently used in signal processing, graphics rendering, and network packet processing.

Commonly used paradigms

- Hybrid Models: Many practical parallel programming scenarios combine multiple paradigms to take advantage of their strengths.
- For example, a program might use shared memory for communication between threads running on the same processor and message passing to communicate between different processors or machines.

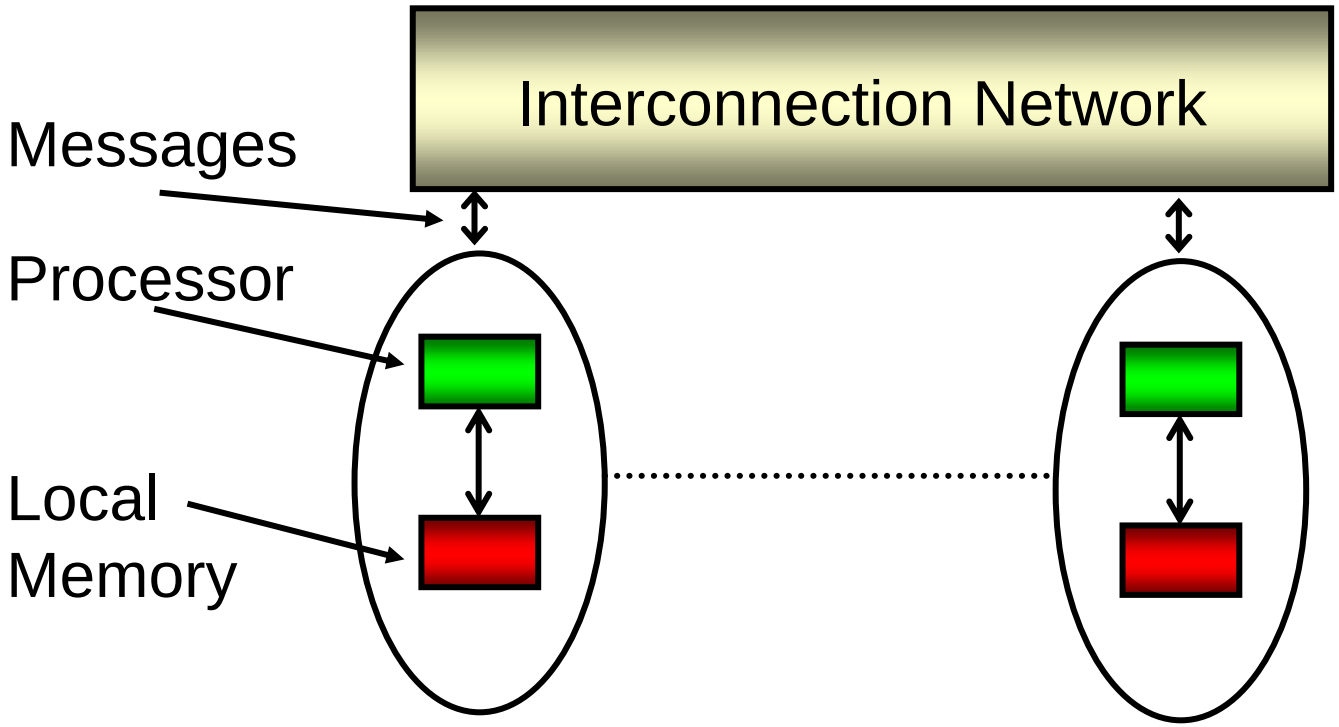
Message-Passing Computing for Distributed Multi-Computers

PART-I

A review of basic concepts

Message-Passing Multicomputer

Complete computers connected through an interconnection network:



Programming

Programming a message-passing multicomputer can be achieved by

- Designing a special parallel programming language
- Extending the syntax/reserved words of an existing sequential high-level language to handle message passing
- Using an existing sequential high-level language and providing a ***library of external procedures for message passing***

Programming

Involves dividing problem into parts (domain or functional) that are intended to be executed simultaneously to solve the problem

Each part executed by separate computers

Parts (processes) communicate by sending messages - the only way to distribute data and collect result

Message Passing Parallel Programming Software Tools

Parallel Virtual Machine (PVM) - developed in late 1980's. Became very popular.

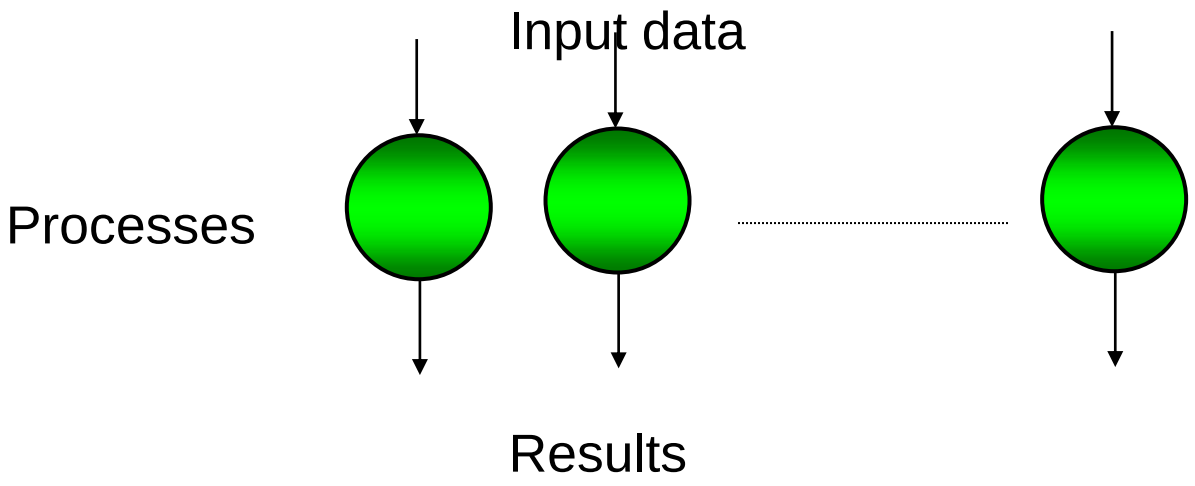
Message-Passing Interface (MPI) - standard defined in 1990s.

Both provide a set of user-level libraries for message passing. Use with regular programming languages (FORTRAN, C, C++, ...).

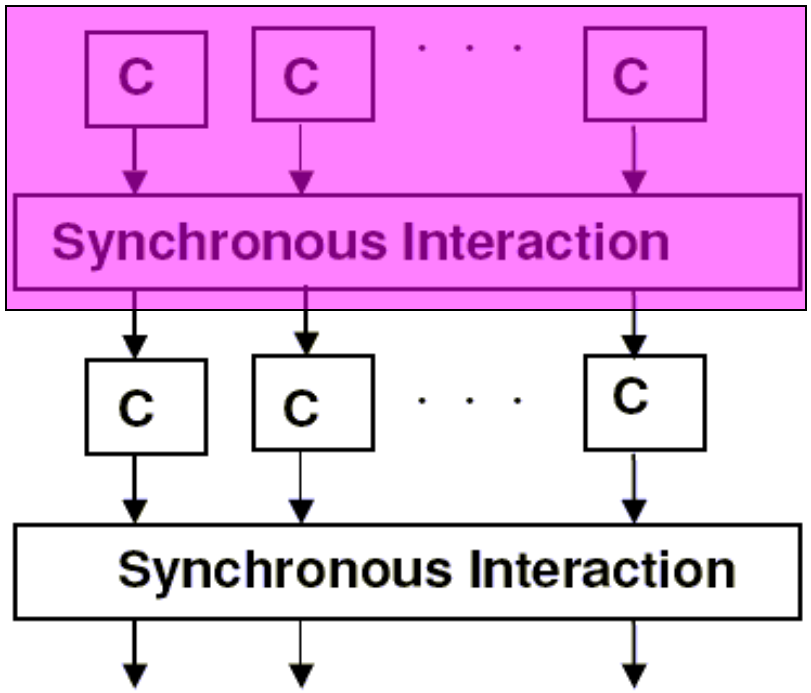
Embarrassingly Parallel Computations

A computation that can be divided into a number of completely independent parts, each of which can be executed by a separate process(or).

No communication or very little communication between processes



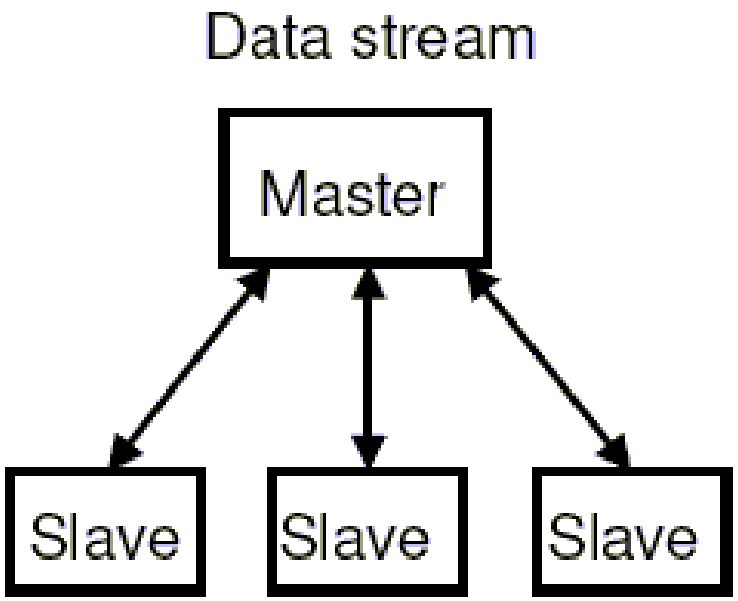
Phase Parallel Model



Nearly embarrassing

- The phase-parallel model offers a paradigm that is widely used in parallel programming
- The parallel program consists of a number of super steps, and each has two phases.
- In a computation phase, multiple processes each perform an independent computation C.
- In the subsequent interaction phase, the processes perform one or more synchronous interaction
- operations, such as a barrier or a blocking communication.
- Then next super step is executed.

Process farm



- This paradigm is also known as the master-slave paradigm.
- A master process executes the essentially sequential part of the parallel program and spawns a number of slave processes to execute the parallel workload.
- When a slave finishes its workload, it informs the master which assigns a new workload to the slave.
- This is a very simple paradigm, where the coordination is done by the master.

MPI (Message Passing Interface)

Standard developed by group of academics and industrial partners to foster more widespread use and portability

Defines routines, not implementation

Several free implementations of MPI standard exist.

List of Few active products/projects

- CRI/EPCC
- Hitachi MPI
- HP MPI
- IBM Parallel Environment for AIX-MPI Library
- LAM/MPI (Supplier: Indiana University)
- MPI for UNICOS Systems
- MPICH (Supplier: Argonne National Laboratory)
- OS/390 Unix System Services Parallel
- RACE-MPI
- SGI Message Passing Toolkit
- Sun MPI

List of Few active products/projects

- CRI/EPCC
- Hitachi MPI
- HP MPI
- IBM Parallel Environment for AIX-MPI Library
- **Open MPI**
- MPI for UNICOS Systems
- **MPICH (Supplier: Argonne National Laboratory)**
- OS/390 Unix System Services Parallel
- RACE-MPI
- SGI Message Passing Toolkit
- Sun MPI

What is MPI

- A **message-passing library** specification
  - Not a compiler specification
  - Not a specific product
- Used for parallel computers, clusters, and heterogeneous networks as a message passing library
- Designed to be used for the development of parallel software libraries
- Designed to provide access to advanced parallel hardware for
  - End users
  - Library writers
  - Tool developers

Where to use MPI?

- We need a portable parallel program
- We are writing a parallel Library

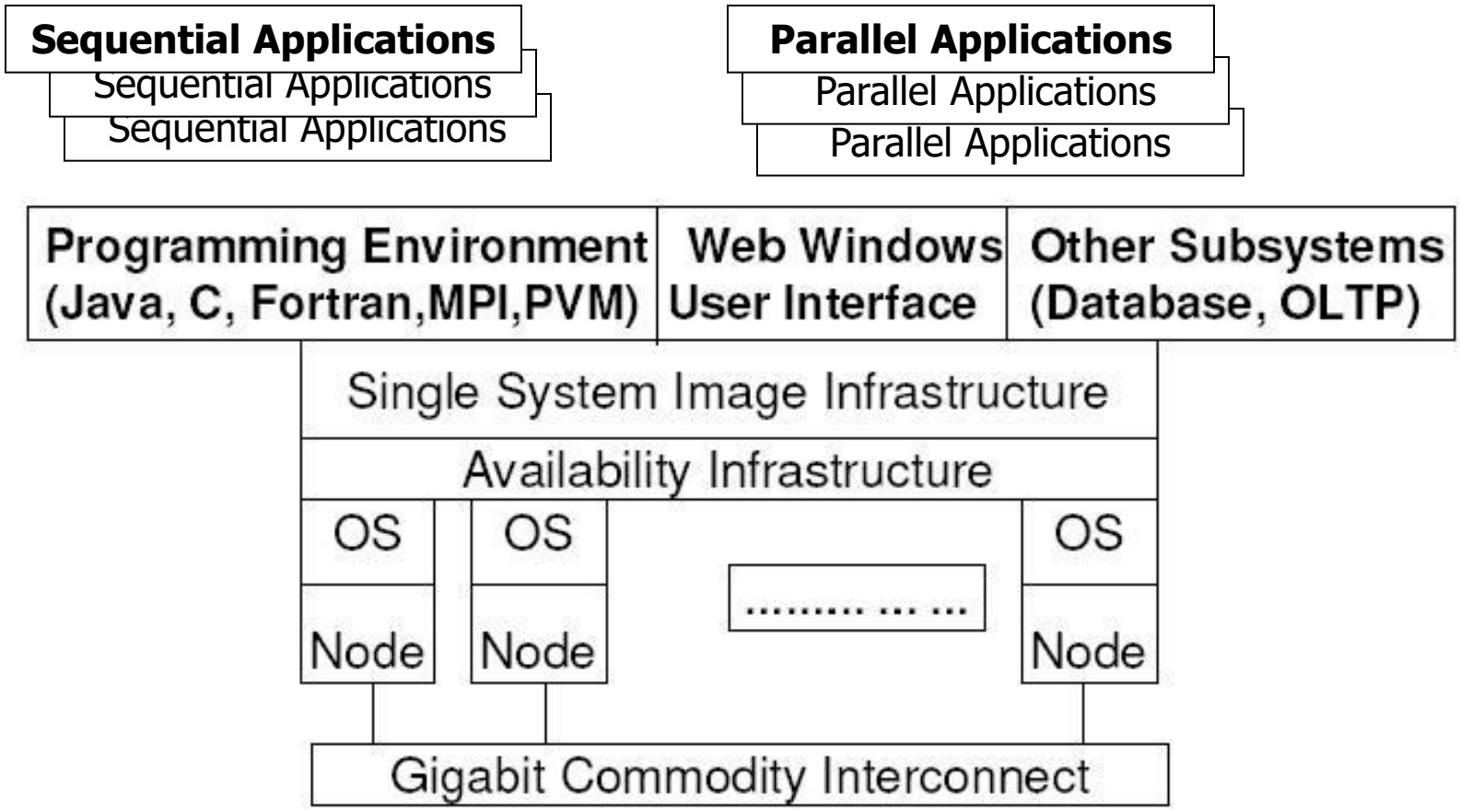
Why learn MPI?

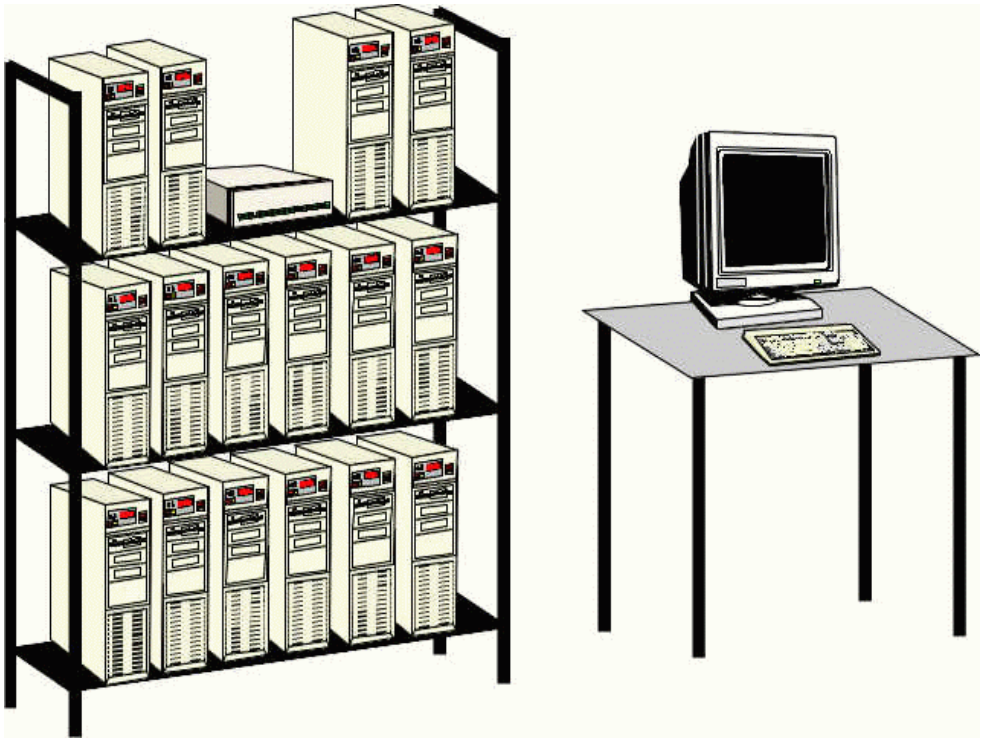
- Portable
- Expressive
- Good way to learn about subtle issues in parallel computing
- Universal acceptance

Cluster of Computers – Features

- A **Compute Cluster** is a type of parallel or distributed processing system, which consists of a collection of interconnected ***stand-alone computers*** cooperatively working together as a single, integrated computing resources. "***stand-alone***" (whole) computer that can be used on its own (full hardware and OS)
- Collection of nodes physically connected over commodity/ proprietary network
- Cluster computer is a collection of complete independent workstations or Symmetric Multi Processors
- Network is a decisive factors for scalability issues (especially for fine grain applications)
- High volumes driving high performance
- Network using commodity components and proprietary architecture is becoming the trend

Cluster system architecture





Source: Internet

Common Cluster Modes

- **High Performance (dedicated)**
- **High Throughput (idle cycle collection)**
- **High Availability**

Users View of Cluster

The users view the entire cluster as **Single system**, which has multiple processors. The user could say: “Execute my application using five processors.” This is different from a distributed system.

- **Single Entry**
- **Single File Hierarchy**
- **Single Networking**
- **Single Input/Output**
- **Single Point of Control**
- **Single Memory Space**
- **Single Job Management System**
- **Single User Interface**
- **Single Process Space**
- **Single System**
- **Symmetry**
- **Location Transparent**

Basics of Message-Passing Programming

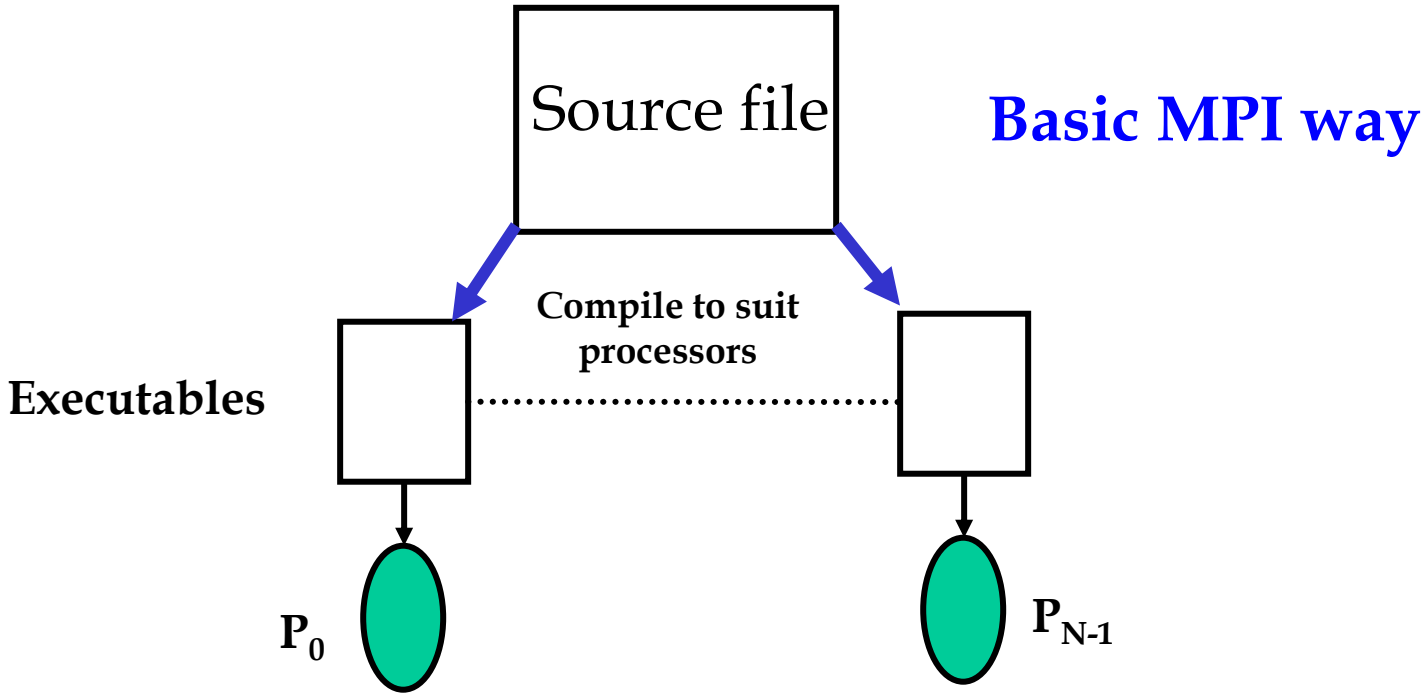
Two primary mechanisms needed:

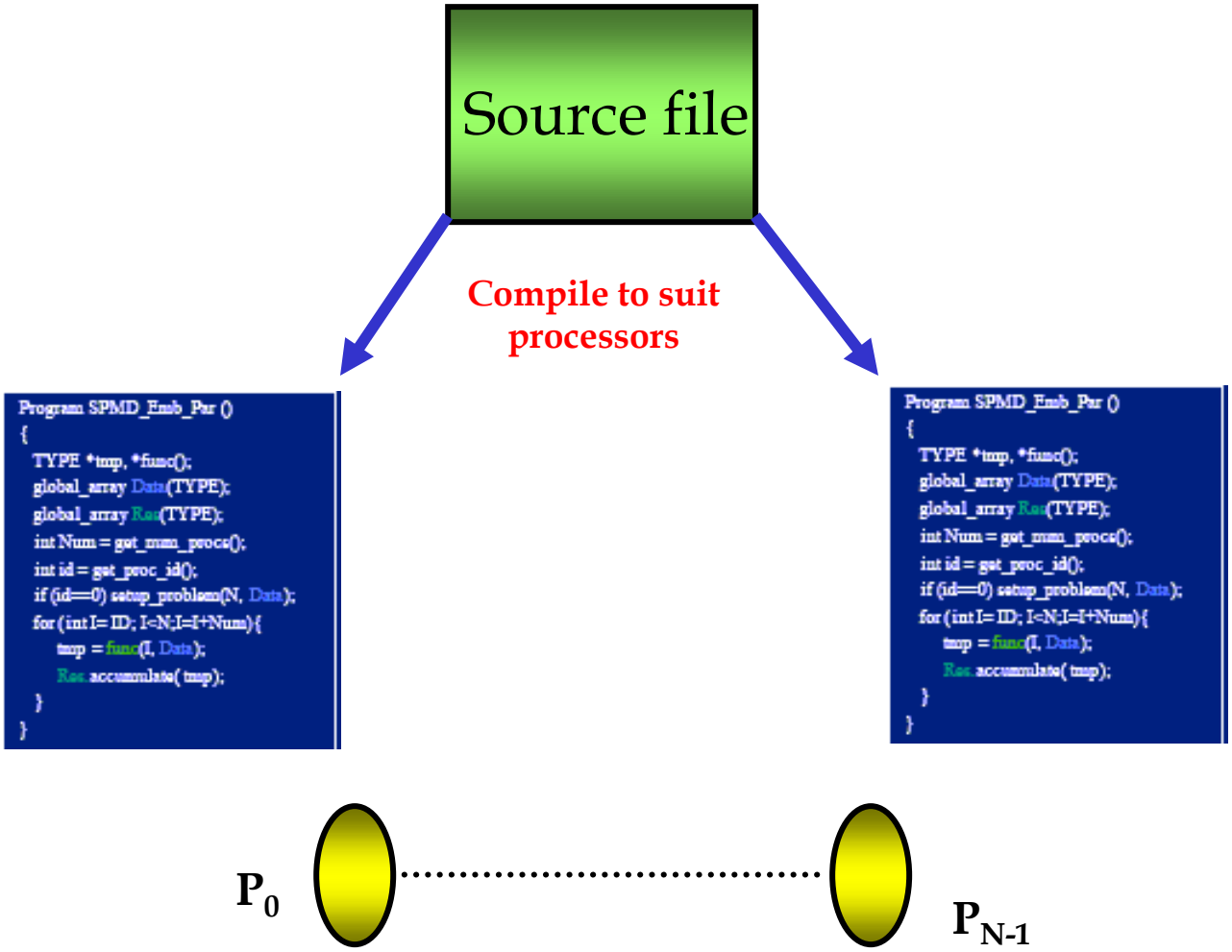
1. A method of creating separate processes for execution on different computers

2. A method of sending and receiving messages

Single Program Multiple Data (SPMD) model

Different processes merged into one program. Within program,control statements select different parts for each processor to execute. All executables start together - static process creation.




Source file
Compile to suit processors
Program SPMD_Emb_Par ()
{
TYPE *tmp, *func();
global_array Data(TYPE);
global_array Res(TYPE);
int Num = get_num_procs();
int id = get_proc_id();
if (id==0) setup_problem(N, Data);
for (int I= ID; I<N;I=I+Num){
tmp = func(I, Data);
Res.accumulate(tmp);
}
}
Program SPMD_Emb_Par ()
{
TYPE *tmp, *func();
global_array Data(TYPE);
global_array Res(TYPE);
int Num = get_num_procs();
int id = get_proc_id();
if (id==0) setup_problem(N, Data);
for (int I= ID; I<N;I=I+Num){
tmp = func(I, Data);
Res.accumulate(tmp);
}
}
P0
PN-1

Evaluating General Message

Message Passing SPMD : C program

```
main (int argc, char **argv)
{
    if (process is to become a controller process)
                {
                        Controller (/* Arguments /*);
                }
        else
                {
                        Worker (/* Arguments /*);
                }
}
```

MPICH

MPICH is an open-source, portable implementation of the Message-Passing Interface Standard.

Designed at Mathematics and Computer Science Division of **Argonne National Laboratory**.

The “CH" in MPICH stands for Chameleon symbol of adaptability to one's environment and thus of portability.

Is MPICH Large or Small?

MPICH is large (>210 Functions)

- MPICH's extensive functionality requires many functions
- Number of functions not necessarily a measure of complexity

MPICH is small (6 Functions)

- Many parallel programs can be written with just 6 basic functions

MPICH is just **right** candidate for message passing

- One need not master all parts of MPICH to use it

Some Basic Concepts

- Processes can be collected into groups
- Each message is sent in a context, and must be received in the same context
- A group and context together form a communicator
- A process is identified by its rank in the group associated with a communicator
- There is a default communicator whose group contains all initial processes, called **`MPI_COMM_WORLD`**

Begin programming with 6 MPI function calls

- MPI_INIT — *Initializes MPI*
- MPI_COMM_SIZE — *Determines number of processes*
- MPI_COMM_RANK — *Determines the label of the calling process*
- MPI_SEND — *Sends a message*
- MPI_RECV — *Receives a message*
- MPI_FINALIZE — *Terminates MPI*

MPI Datatypes

- The data in a message to send or receive is described by a triple (address, count, datatype), where
- An MPI *datatype* is recursively defined as:
  - predefined, corresponding to a data type from the language (e.g., MPI_INT, MPI_DOUBLE)
  - a contiguous array of MPI datatypes
  - a strided block of datatypes
  - an indexed array of blocks of datatypes
  - an arbitrary structure of datatypes
- There are MPI functions to construct custom datatypes, in particular ones for subarrays

C data types

- **MPI_CHAR**
  char
- **MPI_BYTE**
  like unsigned char
- **MPI_SHORT**
  short
- **MPI_INT**
  int
- **MPI_LONG**
  long
- **MPI_FLOAT**
  float
- **MPI_DOUBLE**
  double
- **MPI_UNSIGNED_CHAR**
  unsigned char
- **MPI_UNSIGNED_SHORT**
  unsigned short
- **MPI_UNSIGNED**
  unsigned int
- **MPI_UNSIGNED_LONG**
  unsigned long
- **MPI_LONG_DOUBLE**
  long double (some systems may not implement)
- **MPI_LONG_LONG_INT**
  long long (some systems may not implement)

MPI_Op Options (Collective Operation)

- MPI_BAND
- MPI_BOR
- MPI_BXOR
- MPI_LAND
- MPI_LOR
- MPI_LXOR
- MPI_MAX
- MPI_MAXLOC*
- MPI_MIN
- MPI_MINLOC
- MPI_PROD
- MPI_SUM

* Maximum and Location

Communicators

Defines *scope* of a communication operation.

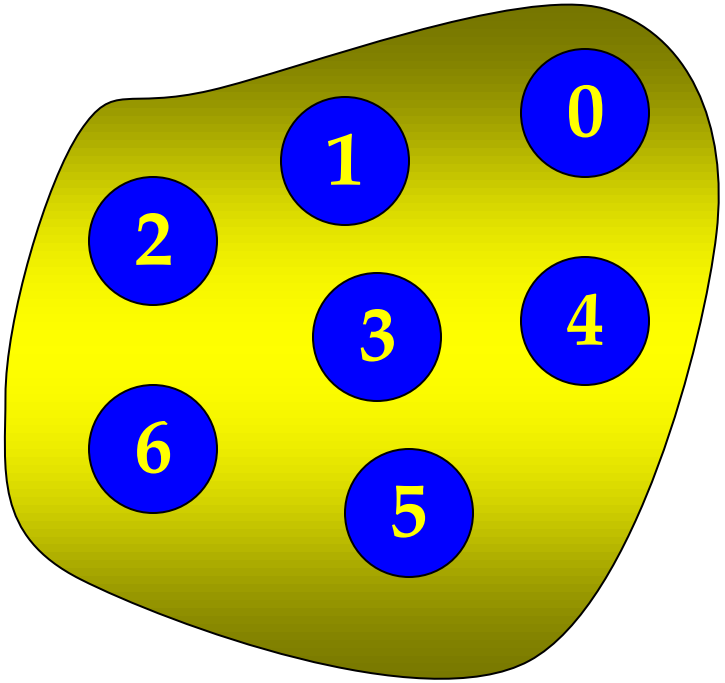
Processes have **ranks** associated with the communicator.

Initially, all processes enrolled in a **"universe"** called **MPI_COMM_WORLD** and each process is given a unique rank, a number from 0 to *n* - 1, where there are *n* processes.

Other communicators can be established for groups of processes.

MPI_COMM_WORLD communicator

This Default Communicator is MPI's mechanism for establishing individual communication universes



MPI_COMM_WORLD

MPI Messages

Message : data (3 parameters) + envelope (3 parameters)

Data: startbuf, count, datatype

- **Startbuf:** address where the data starts
- **Count:** number of elements (items) of data in the message

Envelope: dest, tag, comm

- **Destination or Source:** Sending or Receiving processes
- **Tag:** Integer to distinguish messages

Communicator:

The communicator is communication “universe.”

Messages are sent or received within a given “universe.”

Synchronous Message Passing (Blocking)

Routines that actually return when message transfer completed.

Synchronous send routine Waits until complete message can be accepted by the receiving process before sending the next message.

Synchronous receive routine Waits until the message it is expecting arrives.

Synchronous routines intrinsically perform two actions: They transfer data and they synchronize processes.

Asynchronous Message Passing (Non-Blocking)

Routines that do not wait for actions to complete before returning. Usually require local storage for messages.

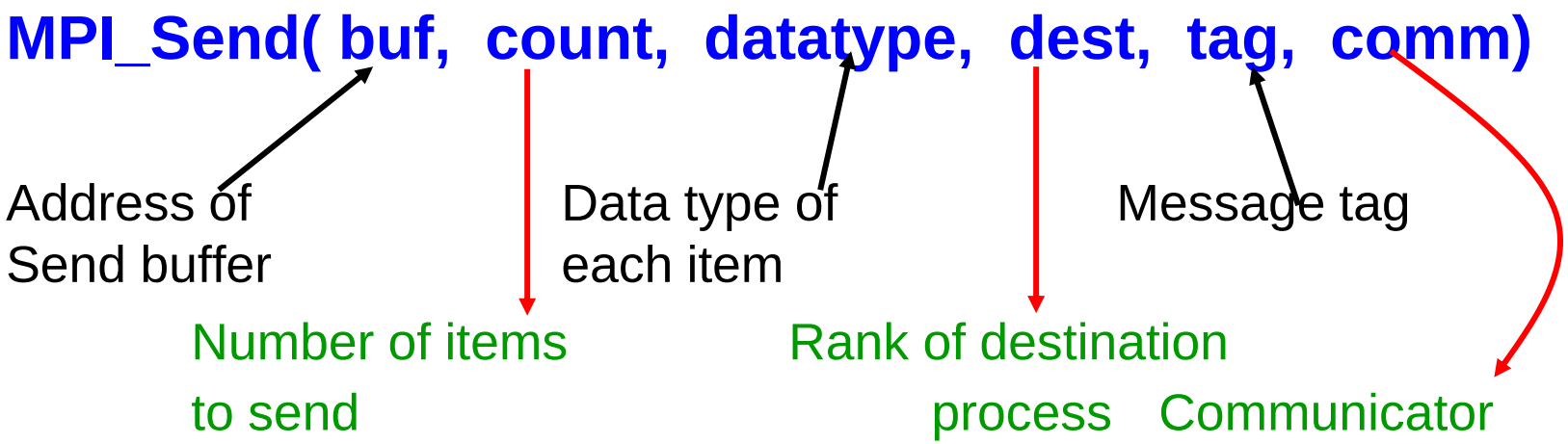
More than one version depending upon the actual semantics for returning.

In general, they do not synchronize processes but allow processes to move forward sooner. Must be used with care.

MPICH Send and Recv

- Communication between two processes
- *Source* process sends message to *destination* process
- Communication takes place within a *communicator*
- Destination process is identified by its *rank* in the communicator

Parameters of the blocking send

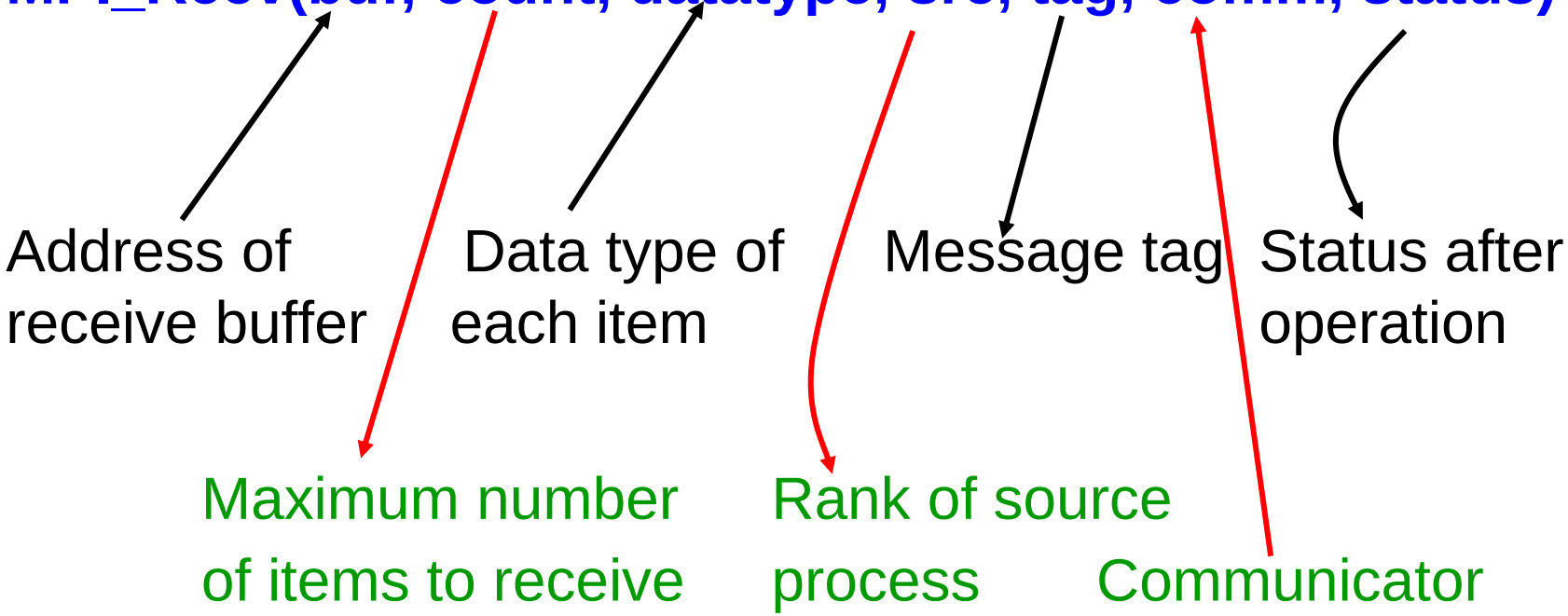


MPI Basic (Blocking) Send

- When this function returns, the data has been delivered to the system and the buffer can be reused.

Parameters of the blocking receive

MPI_Recv(buf, count, datatype, src, tag, comm, status)



MPI Basic (Blocking) Receive

- Waits until a matching (both **source** and **tag**) message is received from the system, and the buffer can be used
- **source** is rank in communicator specified by **comm**, or **MPI_ANY_SOURCE**
- **tag** is a tag to be matched on or **MPI_ANY_TAG**
- receiving fewer than **count** occurrences of **datatype** is OK, but **receiving more is an error**
- **status** contains further information (e.g. size of message)

Message Tag

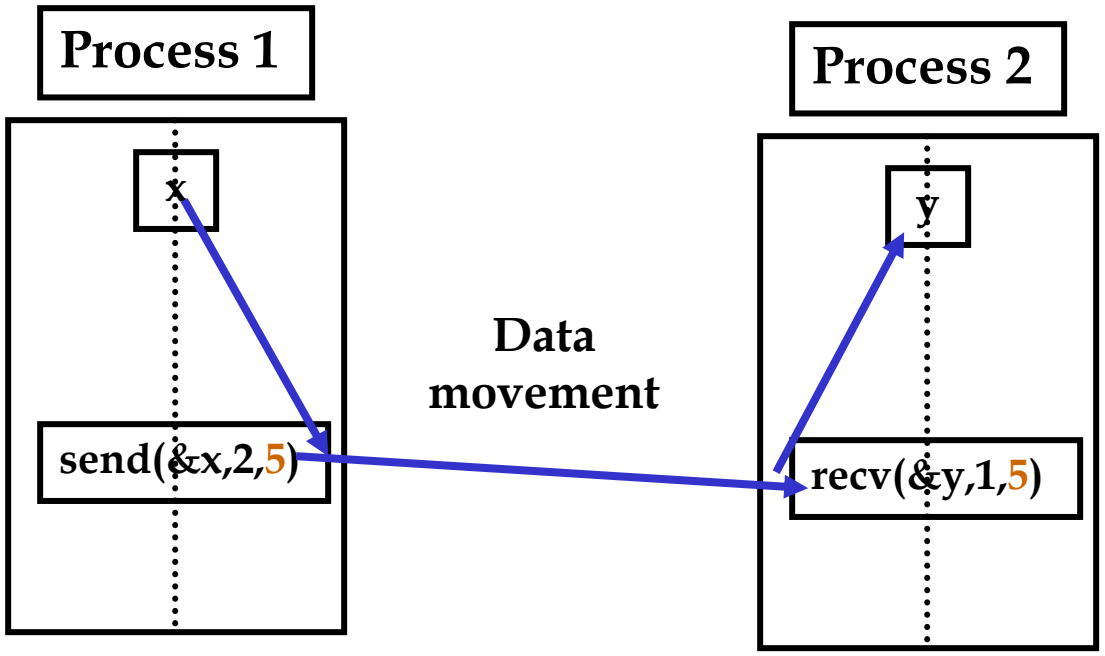
Used to differentiate between different types of messages being sent.

Message tag is carried within message.

If special type matching is not required, a *wild card* message tag is used, so that the recv() will match with any send().

Message Tag Example

To send a message **x** with message tag 5 from a source process 1 to a destination process 2 and assign to **y**:



Waits for a message from process 1 with a tag of 5

Initializing MPICH

- Must be first routine called
- int MPI_Init(int *argc, char **argv);

What makes an MPICH Program?

Include files

- mpi.h (c)
- mpif.h (Fortran)

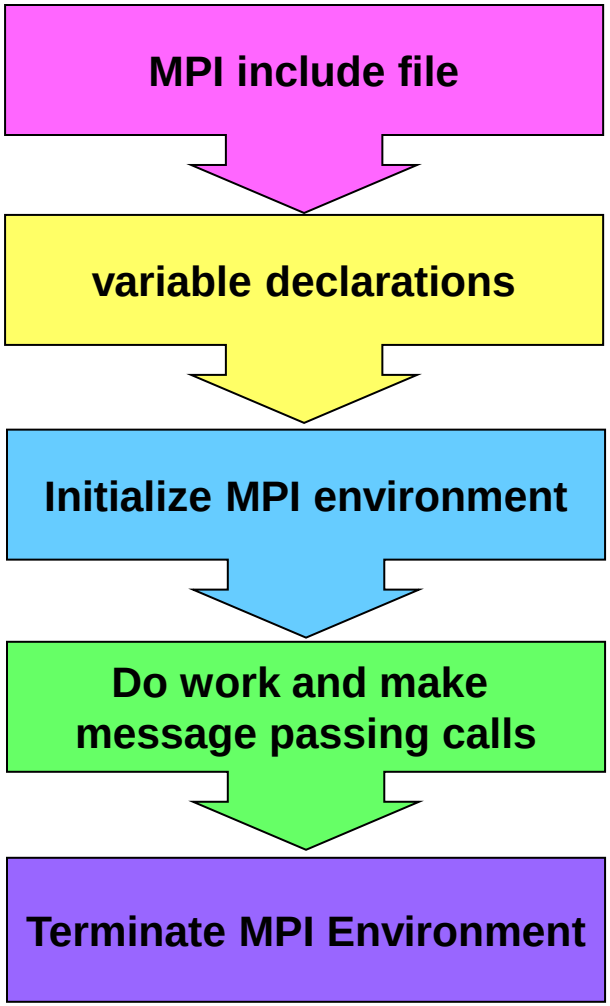
Initiation of MPI

- MPI_INIT

Completion of MPI

- MPI_FINALIZE

General MPI Program Structure




```
#include <mpi.h>
void main (int argc, char **argv)
{
int np, rank;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Comm_size(MPI_COMM_WORLD,&np);
/*      Do Some Works              */
MPI_Finalize();
}
```

MPI_Init

- The command line arguments are provided to MPI_Init to allow an MPI implementation to use them in initializing *the MPI environment*.

- They are passed by reference to allow an MPI implementation to provide them in environments where the command-line arguments are not provided to **main function**.

MPI_Init

- At least one process has access to `stdin`, `stdout`, and `stderr`
- The user can find out which process this is by querying the attribute MPI_IO on MPI_COM WORLD
- In MPICH all processes have access to `stdin`, `stdout`, and `stderr` and on networks these I/O streams are routed back to the process with rank 0 in MPI_COMM_WORLD.

MPI_Init

- On most systems, these streams also can be redirected through `mpirun`, as follows

```
mpirun –np 64 myprog -myarg 13 <data.in>
results.out
```

- Here we assume that –myarg 13 are command-line arguments processed by the application myprog. After MPI_Init, each process will have these arguments in its `argv`

Example

To send an integer *x* from process 0 to process 1,

```
MPI_Comm_rank(MPI_COMM_WORLD,&myrank); /* find rank */
if (myrank == 0) {
        int x;
            MPI_Send(&x, 1, MPI_INT, 1, msgtag,
                                MPI_COMM_WORLD);
            } else if (myrank == 1) {
            int x;
            MPI_Recv(&x, 1, MPI_INT, 0,msgtag,
                                MPI_COMM_WORLD, status);
}
```

Let us write the first Complete C/MPICH program

Write a simple parallel program in which every process with rank greater than 0 sends a message "Hello-Participants" to a process with rank 0. The processes with rank 0 receives the message and prints it.

```
#include "mpi.h"
main (int argc, char **argv) {
int MyRank, Numprocs, tag, ierror, i;
MPI_Status status;
char send_message[20], recv_message[20];
MPI_Init (&argc, &argv);
MPI_Comm_size (MPI_COMM_WORLD, &Numprocs);
MPI_Comm_rank (MPI_COMM_WORLD, &MyRank);
tag = 100;
strcpy (send_message, "Hello-Participants");
if (MyRank==0) {
        for (i=1; i<Numprocs; i++) {
          MPI_Recv (recv_message,20, MPI_CHAR, i, tag, MPI_COMM_WORLD,&status);
          printf ("node %d : %s \n", i, recv_message);
         }
} else
         MPI_Send(send_message, 20, MPI_CHAR,0, tag, MPI_COMM_WORLD);
         MPI_Finalize();
}
```

Cluster at CMSD

- Total number of cores over all the compute nodes (42) is 1680 (40 cores per node) and the number of threads is 3360 (intel hyper threading);
- Peak performance is ~120TFlops
- Maximum overall performance of the HPCF is ~100 TFlops.


14.9
29.5


PARAM

Compiling/executing (SPMD) C/MPICH program

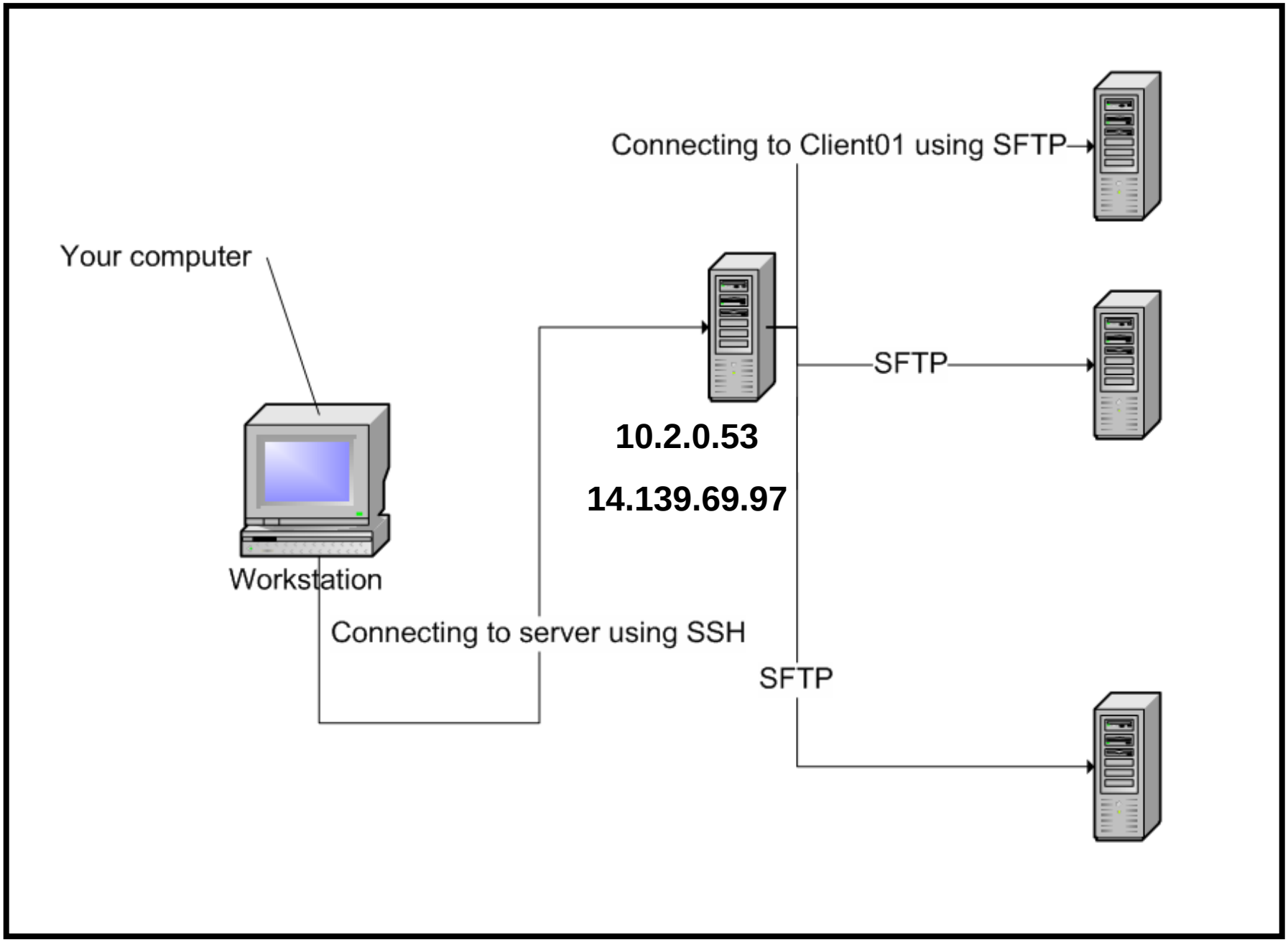
To compile MPI programs:

```
mpicc -o file file.c
```

Or

```
mpiCC -o file file.cpp
```

To execute MPI program: `mpirun -np no_processes file`


Connecting to Client01 using SFTP
Your computer
SFTP
10.2.0.53
14.139.69.97
Workstation
Connecting to server using SSH
SFTP