Walchand College of Engineering
(Government Aided Autonomous Institute)
Vishrambag, Sangli. 416415

Computer Algorithms

M2-Part 1:Introduction to Parallel programming

23-24
D B Kulkarni
Information Technology Department

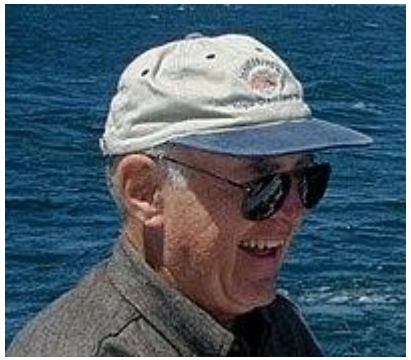
Circuit complexity

How fast is the circuit

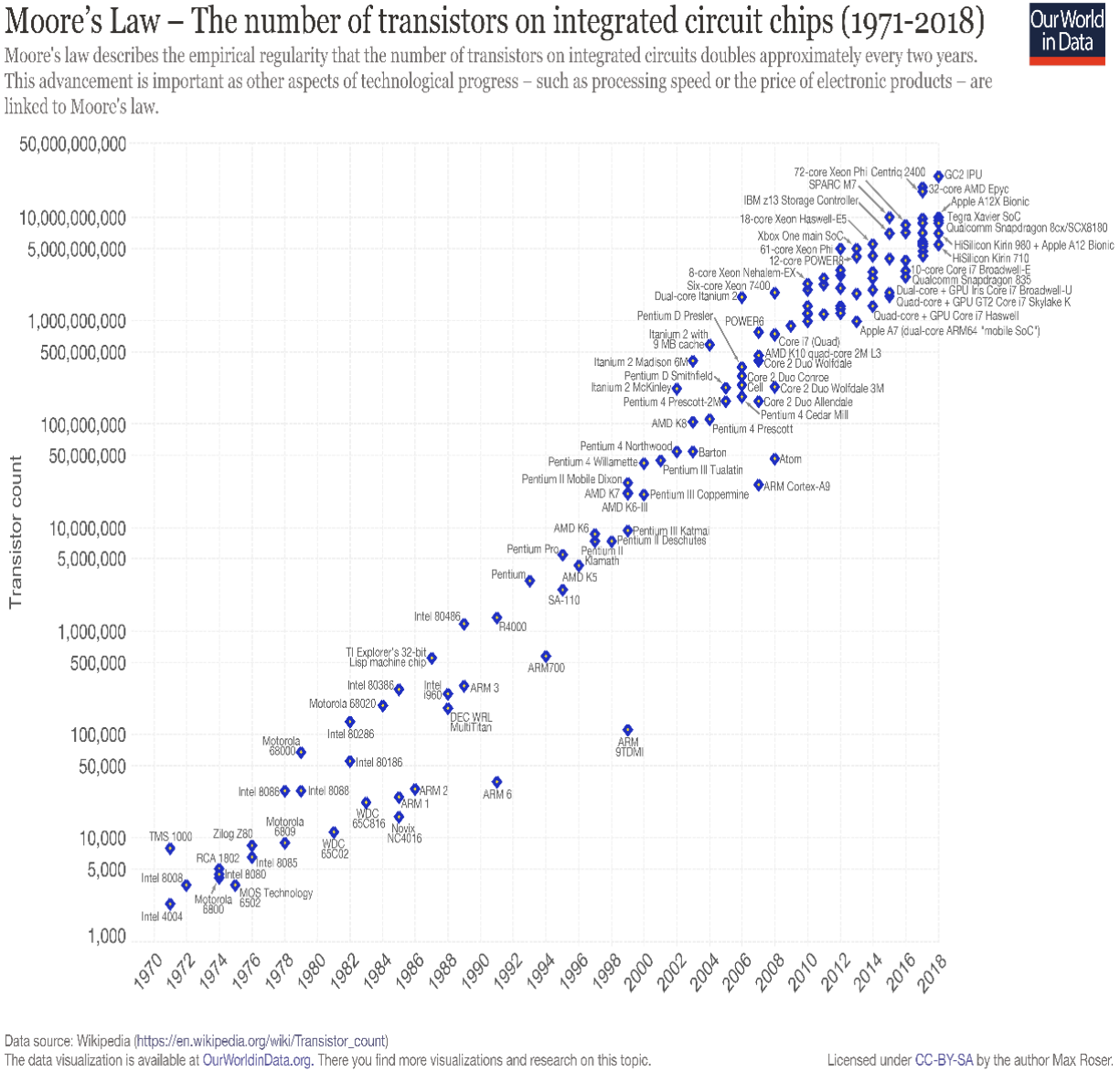
Depends on clock speed

First electromechanical general purpose computer, the Z3, operated at a frequency of about 5–10 Hz

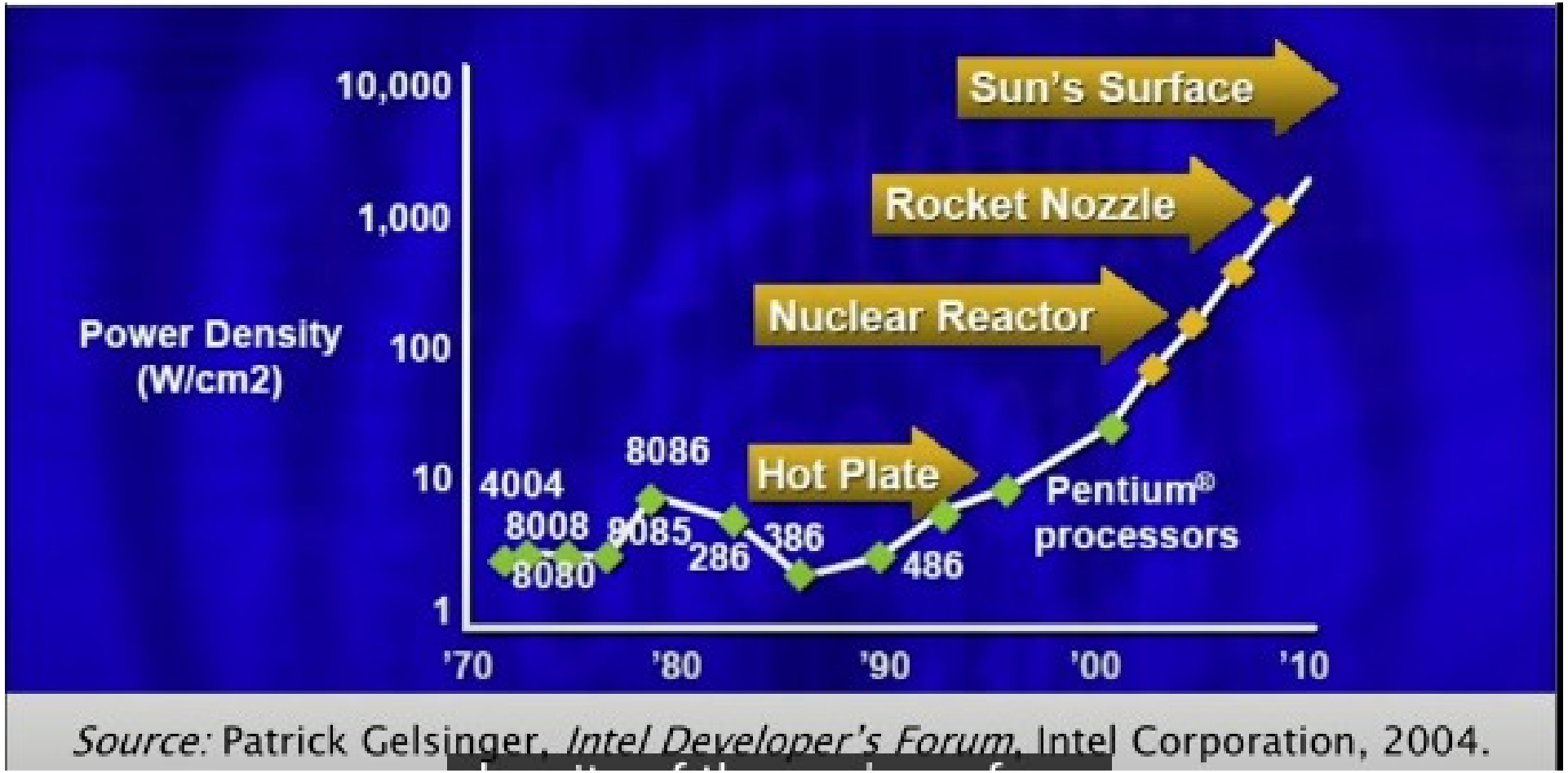
The first commercial PC, the Altair 8800 (by MITS), used an Intel 8080 CPU with a clock rate of 2 MHz



- Transistor count: **Moore's law** is the observation that the number of transistors in a dense integrated circuit (IC) doubles about every two years
- 1965-75-80----25?



Power Density



Source: Patrick Gelsinger, *Intel Developer's Forum*, Intel Corporation, 2004.

Bemchmarks

A benchmark- Act of running a computer program or a set of programs or other operations, in order to assess the relative performance of an object, normally by running a number of standard tests and trials against it.

- NAS Parallel Benchmarks (NPB) are a set of benchmarks targeting performance evaluation of highly parallel supercomputers.
  - developed and maintained by the NASA Advanced Supercomputing (NAS) Division.
  - Versions NPB1, NPB2, NPB3
- **Linpack Benchmark:** A measure of a computer's floating-point rate of execution determined by running a computer program that solves a dense system of linear equations.
  - The software used in this experiment is based on two routines from the LINPACK collection
    - DGEFA (Double precision GEneral matrix FActor): Complexities O(n^3)
    - DGESL (Double precision GEneral matrix SoLve): Complexities O((n^2)

  .
- The LINPACK software package has been replaced by LAPACK, which features block algorithms to take advantage of cache memories.
- Matrix computation on distributed memory parallel computers with message passing interprocessing communication is now handled by ScaLAPACK
- HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers. It is software package that solves a (random) dense linear system in double precision (64 bits) arithmetic on distributed-memory computers

Supercomputer listing

- List of Supercomputers (rating based)

1. TOP500: Based on FLOPS
2. GREEN500: Based on FLOPS/watts
3. GRAPH500 :Based on TEPS

- Traversed edges per second (TEPS) that can be performed by a supercomputer cluster is a measure of both the communications capabilities and computational power of the machine.

This is in contrast to the more standard metric of floating point operations per second (FLOPS), which does not give any weight to the communication capabilities of the machine.

Processing unit (CPU) speeds

Floating **P**oint **O**perations **P**er **S**econd (FLOPS)

- FP16
- FP32 Single precision
- FP64 Double precision

Unit	Notaion	Size		Unit	Notaion	Size
kiloFLOPS	kFLOPS	10^3		petaFLOPS	PFLOPS	10^{15}
megaFLOPS	MFLOPS	10^6		exaFLOPS	EFLOPS	10^{18}
gigaFLOPS	GFLOPS	10^9		zettaFLOPS	ZFLOPS	10^{21}
teraFLOPS	TFLOPS	10^{12}		yottaFLOPS	YFLOPS	10^{24}

Tensor Processing Unit (**TPU**) is an AI accelerator application-specific integrated circuit (ASIC) developed by Google specifically for neural network machine learning, particularly using Google's own TensorFlow software. (TPU1, TPU2,TPU3,TPU4, Edge TPU)

TOP500-2023

Rank	System	Cores	Rpeak (PFlop/s)	Power (kW)
1	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,699,904	1,679.82	22,703
2	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	537.21	29,899
3	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	2,220,288	428.70	6,016

Rank	System	Cores	Rpeak (PFlop/s)	Power (kW)
75	**AIRAWAT - PSAI** - NVIDIA DGX A100, AMD EPYC 7742 64C 2.25GHz, NVIDIA A100, Infiniband HDR, Netweb Technologies Center for Development of Advanced Computing (C-DAC) India	81,344	13.17	

Green500

Rank	TOP500 Rank	System	Cores	Rmax (PFlop/s)	Power (kW)	Energy Efficiency (GFlops/watts)
1	255	**Henri** - ThinkSystem SR670 V2, Intel Xeon Platinum 8362 32C 2.8GHz, NVIDIA H100 80GB PCIe, Infiniband HDR, Lenovo Flatiron Institute United States	8,288	2.88	44	65.396
2	34	**Frontier TDS** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	120,832	19.20	309	62.684
3	12	**Adastra** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE Grand Equipement National de Calcul Intensif - Centre Informatique National de l'Enseignement Suprieur (GENCI-CINES) France	319,072	46.10	921	58.021
154	169	**Pratyush** - Cray XC40, Xeon E5-2695v4 18C 2.1GHz, Aries interconnect , HPE Indian Institute of Tropical Meteorology India	119,232	3.76	1,353	2.781

Parallelism

Data parallel

distributing the data across different nodes, which operate on the data in parallel

running the same task on different components of data

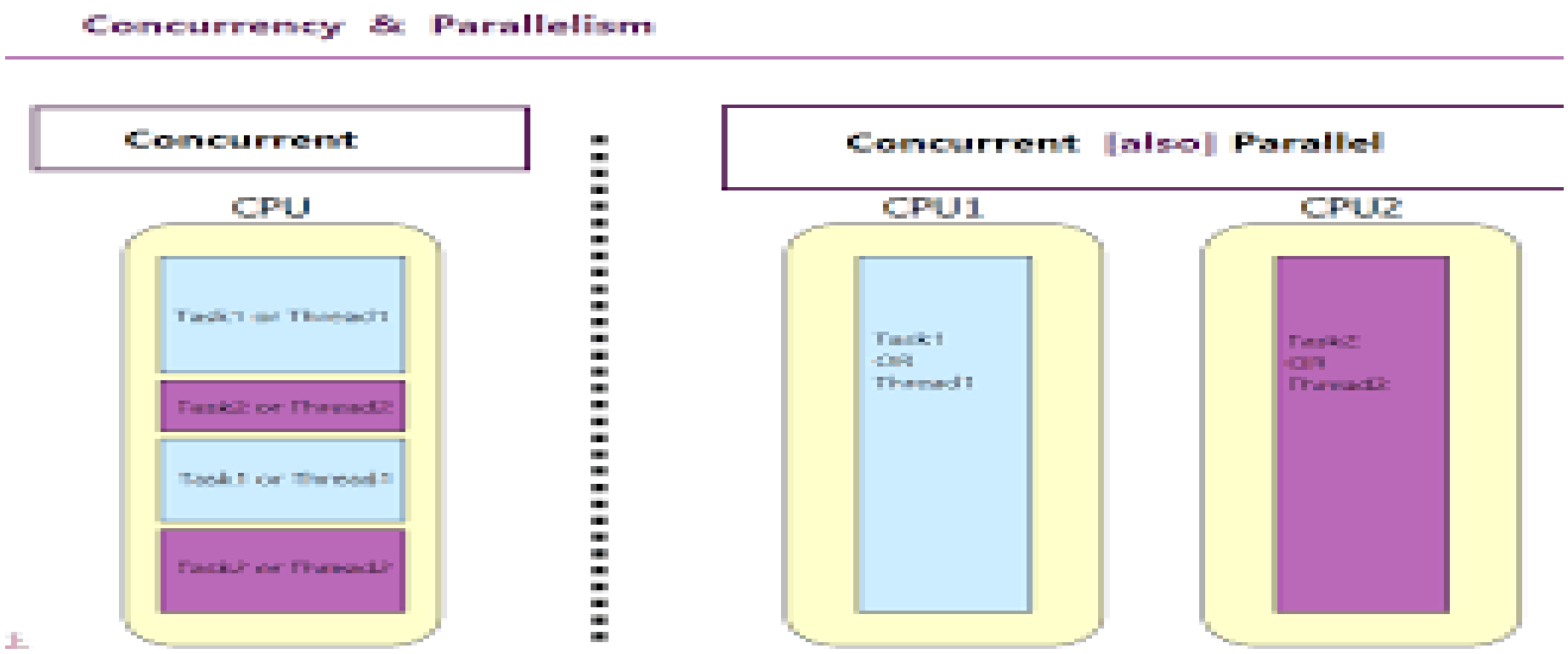
Task Parallel

parallelization of computer code across multiple processors in parallel computing environments
distributing tasks—concurrently performed by processes or threads—across different processors
running many different tasks at the same time on the same data.

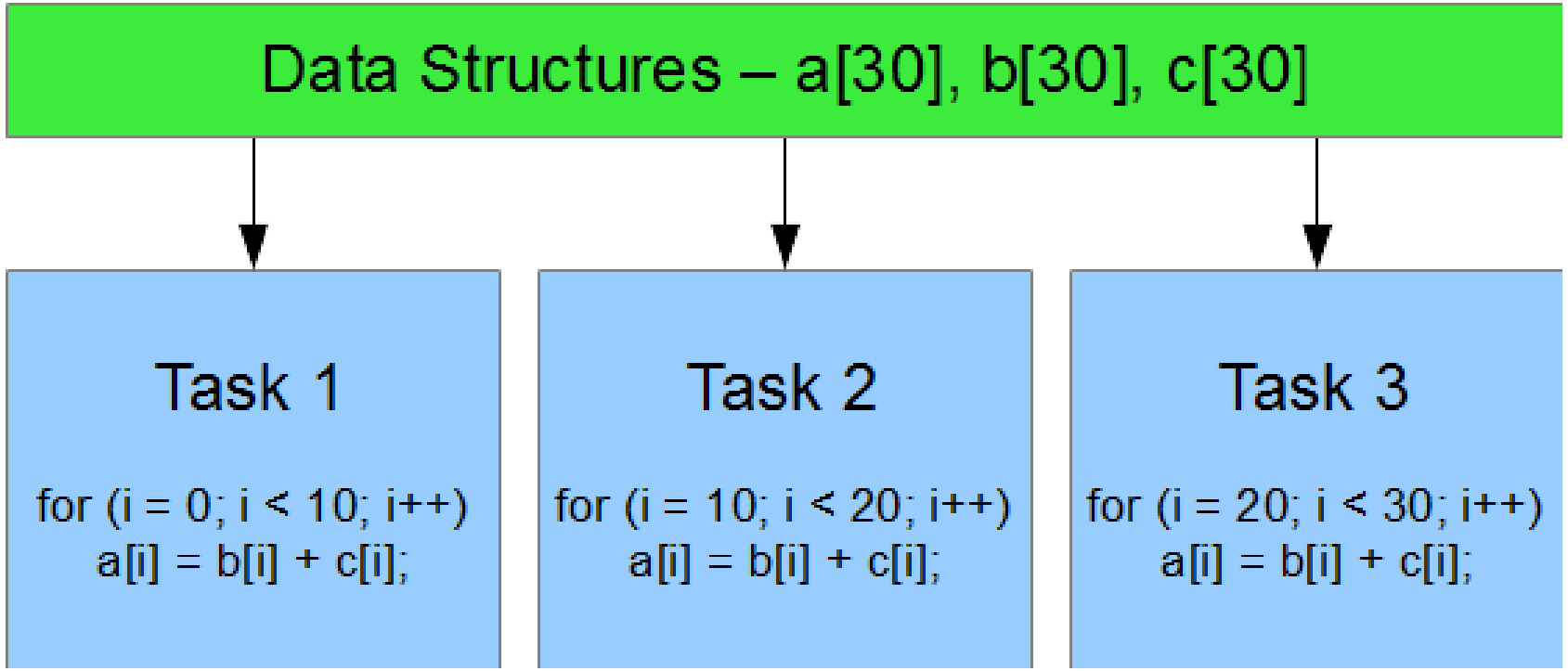
A common type of task parallelism is pipelining which consists of moving a single set of data through a series of separate tasks where each task can execute independently of the others.

out-of-order execution

Concurrency and Parallelism



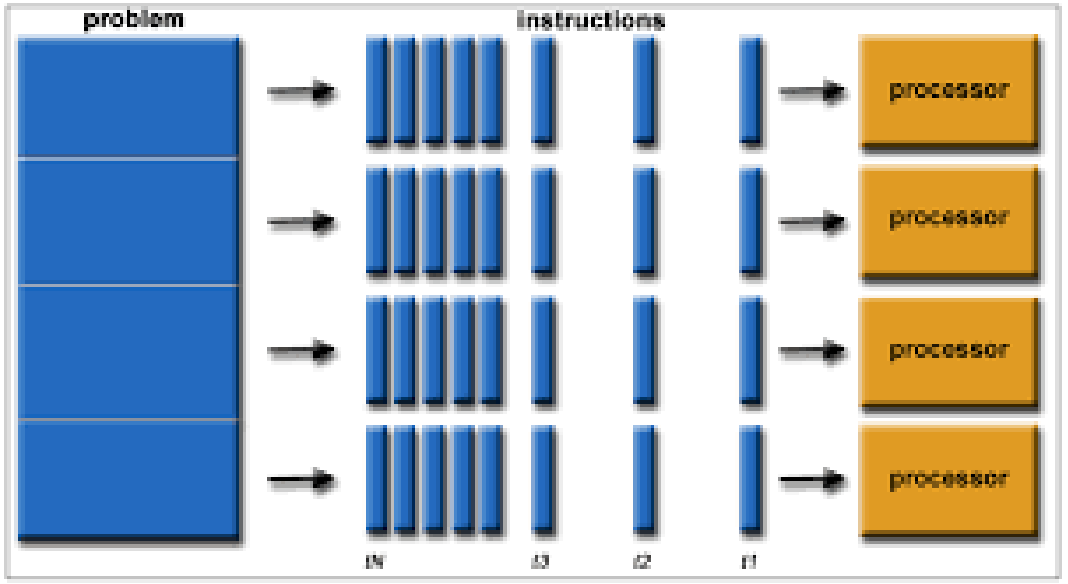
Data Parallel



Data parallel

distributing the data across different nodes, which operate on the data in parallel

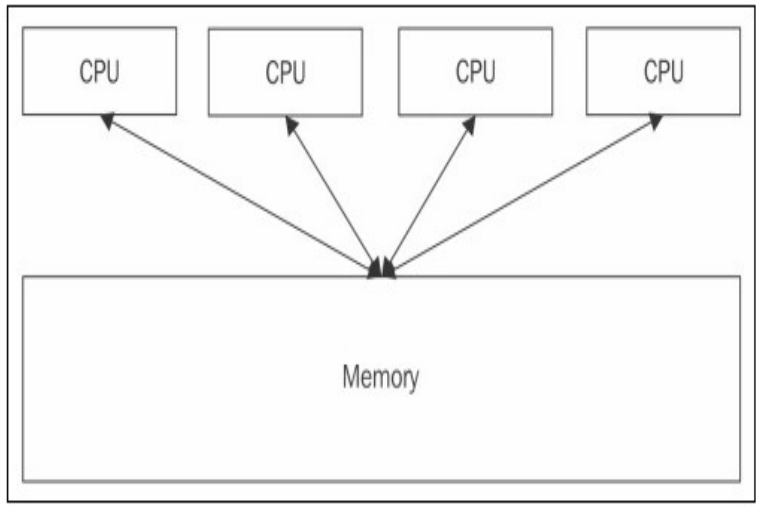
Task Parallel



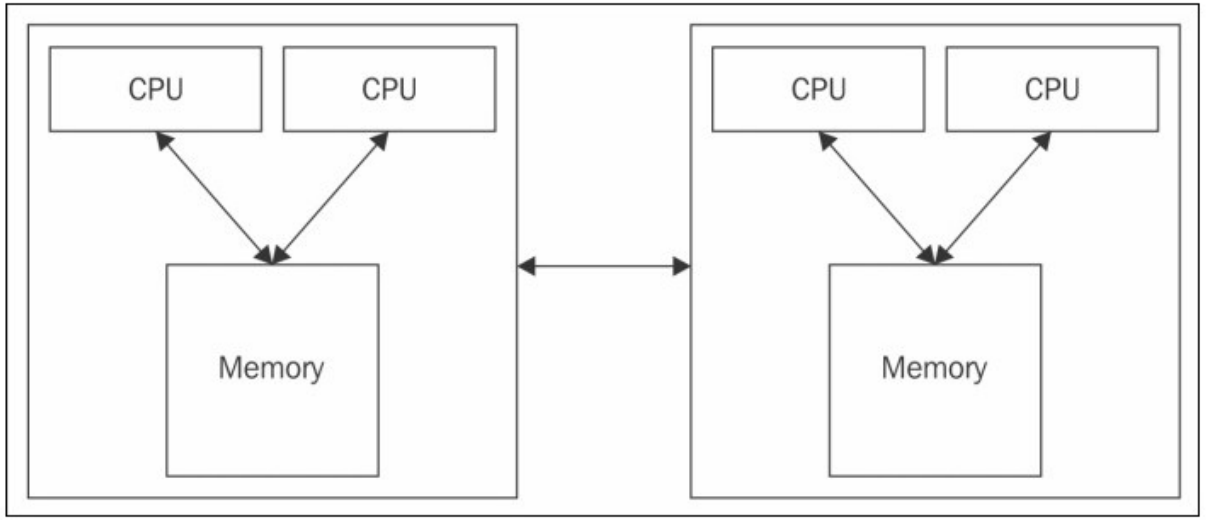
Data parallel

distributing the data across different nodes, which operate on the data in parallel

Parallel system



Shared Memory Model



Distributed Memory Model

Shared memory models

The shared-memory programming models are used to develop solutions for machine architectures that share one common memory space across a set of processors. The models include:

- the *pure* shared memory model
- the multithreading models
  - the programmer-controlled model
  - the API controlled model

Pure Shared memory models

The pure shared memory programming model identifies data independently of all processors. This model does not associate data with any particular processor. It manages access to shared memory through a system of locks and semaphores.

Advantages:

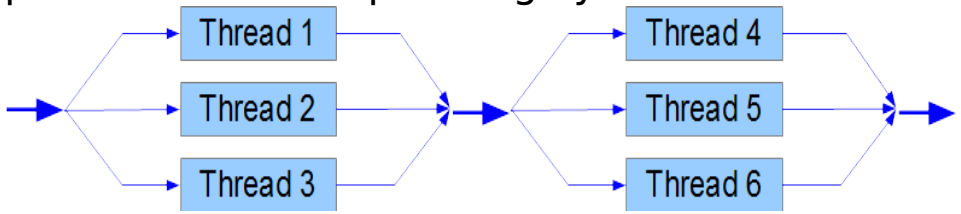
- no concept of data ownership
- program development is simple

Drawbacks:

- difficult to manage data locality
- programming instructions are low-level

Multithreaded Shared memory models

The multi-threading models divide a part of a process into threads. A thread is an independent stream of instructions that the operating system can schedule independently on the processors. A thread exists within a process and uses that process' resources. Thread-creation requires much less operating-system overhead than process-creation.



The multi-threading models associate each thread with some local data. Each thread can execute concurrently with the other threads. Each thread communicates with other threads through shared memory.

The multi-threading models require synchronization to ensure that concurrently executing threads are not updating the same memory address.

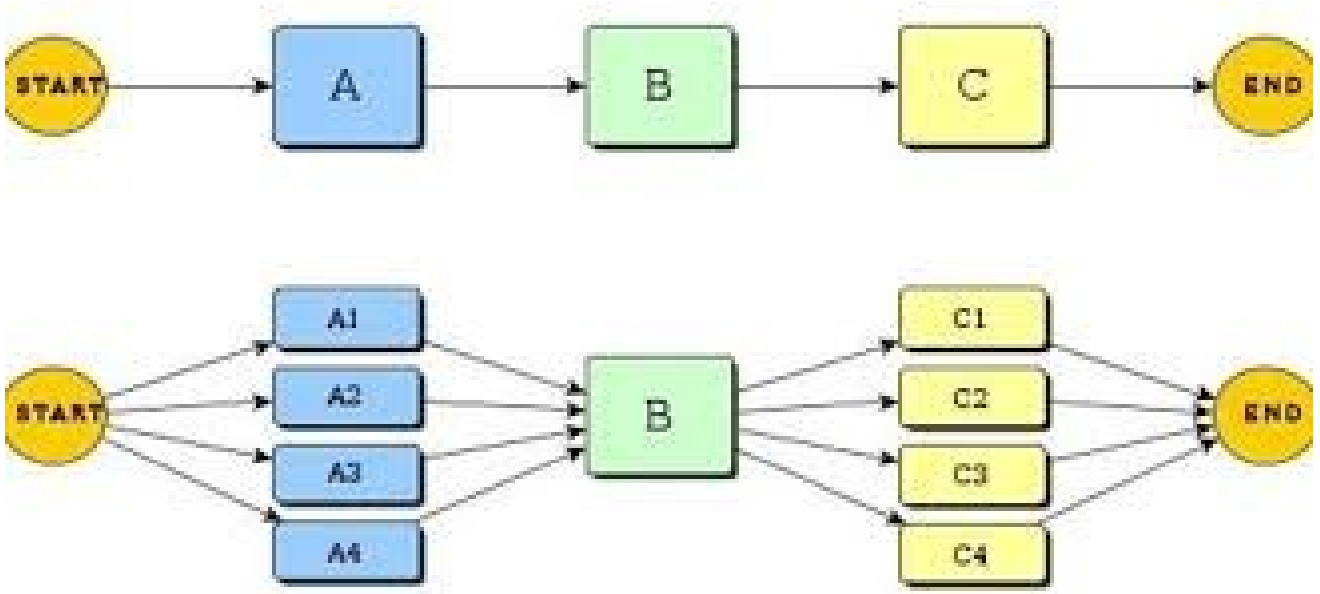
Advantages:

- programmer manages data locality
- run-time system schedules threads

Drawback:

- programmer is responsible for determining the parallelism

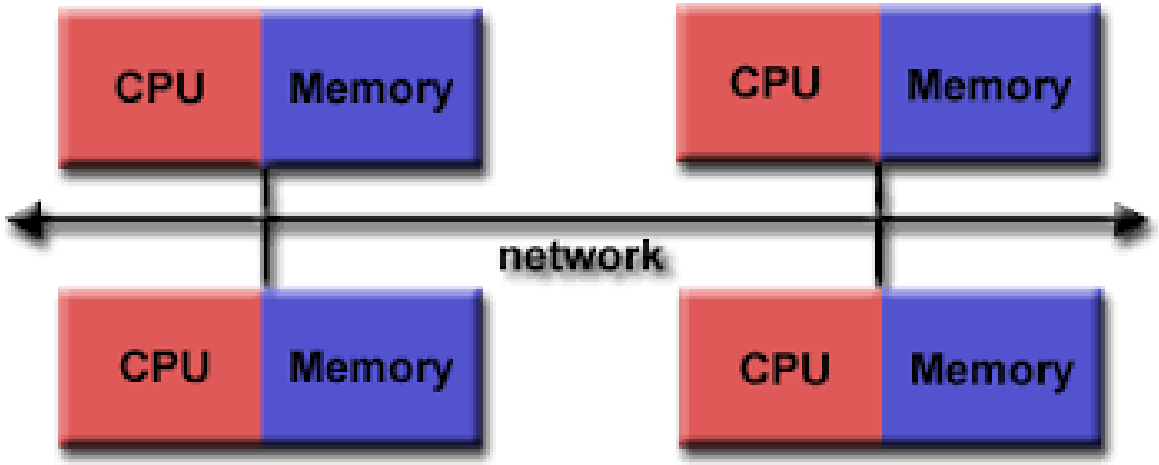
Shared Memory Model: OpenMP



Open Multi Processing (OpenMP)
Data parallel

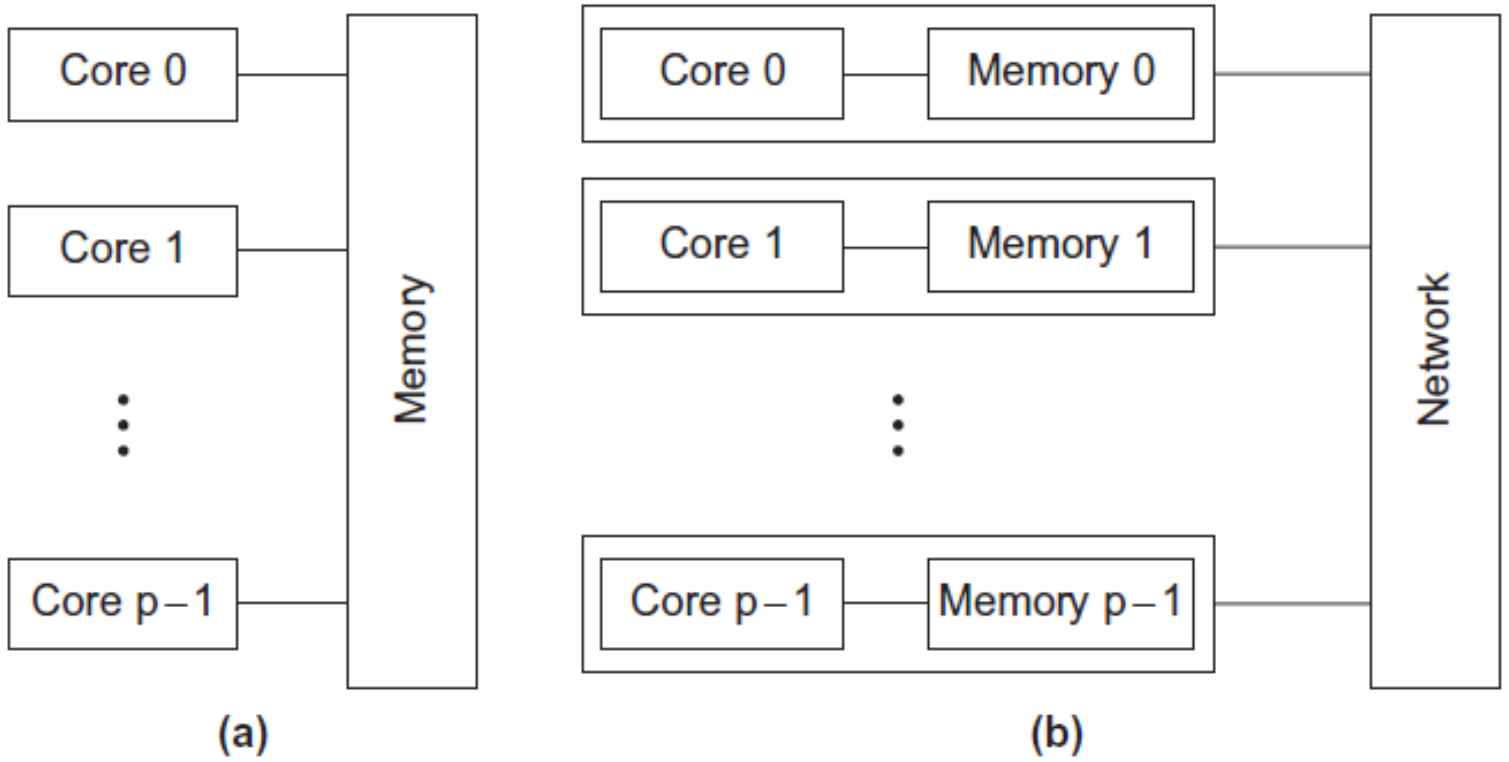
distributing the data across different nodes, which operate on the data in parallel

Distributed Memory Model: MPI

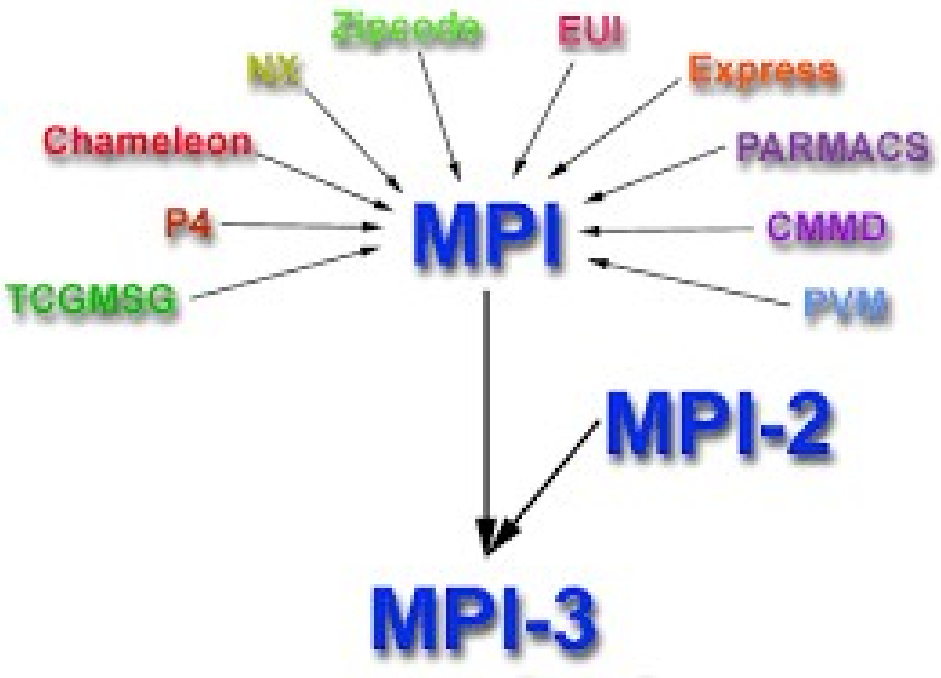


Message Passing Interface

Shared vs Distributed Memory Model



MPI



MPI1: May, 1994

MPI2 : July, 1997:focused on process creation and management, one-sided communications, extended collective communications, external interfaces and parallel I/O

MPI3 : significant extensions to MPI functionality, including non-blocking collectives, new one-sided communication operations, and Fortran 2008 bindings

Parallel system Performance

Computation- circuit complexity

Communication- commu complexity

Flynn's Taxonomy

SISD Single instruction stream Single data stream	(SIMD) Single instruction stream Multiple data stream
MISD Multiple instruction stream Single data stream	(MIMD) Multiple instruction stream Multiple data stream

Parallel system Performance

Performance Metrics • How do we quantify performance when running in parallel?

Consider execution time T(N,P) measured while running on P “processors” (cores) with problem size/complexity N

1. Speedup:
   $S(N,P)= T(N,1)/T(N,P)=T_s/T_p$

   typically S(N,P) < P

2. Parallel efficiency:
   E(N,P)=S(N,P)/P=

   typically E(N,P) < 1

   Serial efficiency: typically E(N,1) OR E(N) <= 1

3. Scaling: Scaling describes how the runtime of a parallel application changes as the number of processors is increased

Parallel system Performance: Scaling

Two types of scaling: •

1. Strong Scaling (increasing P, constant N): problem size/complexity stays the same as the number of processors increases, decreasing the work per processor

   Ideal strong scaling: runtime keeps decreasing in direct proportion to the growing number of processor used

2. Weak Scaling (increasing P , increasing N): problem size/complexity increases at the same rate as the number of processors, keeping the work per processor the same

   Ideal weak scaling: runtime stays constant as the problem size gets bigger and bigger

Good strong scaling is generally more relevant for most scientific problems, but more difficult to achieve than good weak scaling

Extent of parallelism

Consider a typical program

- Sections of code that are inherently serial so can't be run in parallel
- Sections of code that could potentially run in parallel •

Suppose serial code accounts for a fraction a of the program's runtime

- Assume the potentially parallel part could be made to run with 100% parallel efficiency, then:
- Hypothetical runtime in parallel = $T(N,P) = \alpha T(N,1) + (1-\alpha)T(N,1)/P$
- Hypothetical speedup = $S(N,P) = T(N,1) / T(N,P) = P / (\alpha P + (1-\alpha))$

Speedup fundamentally limited by the serial fraction

- Speedup will always be less than $1/\alpha$ no matter how large P

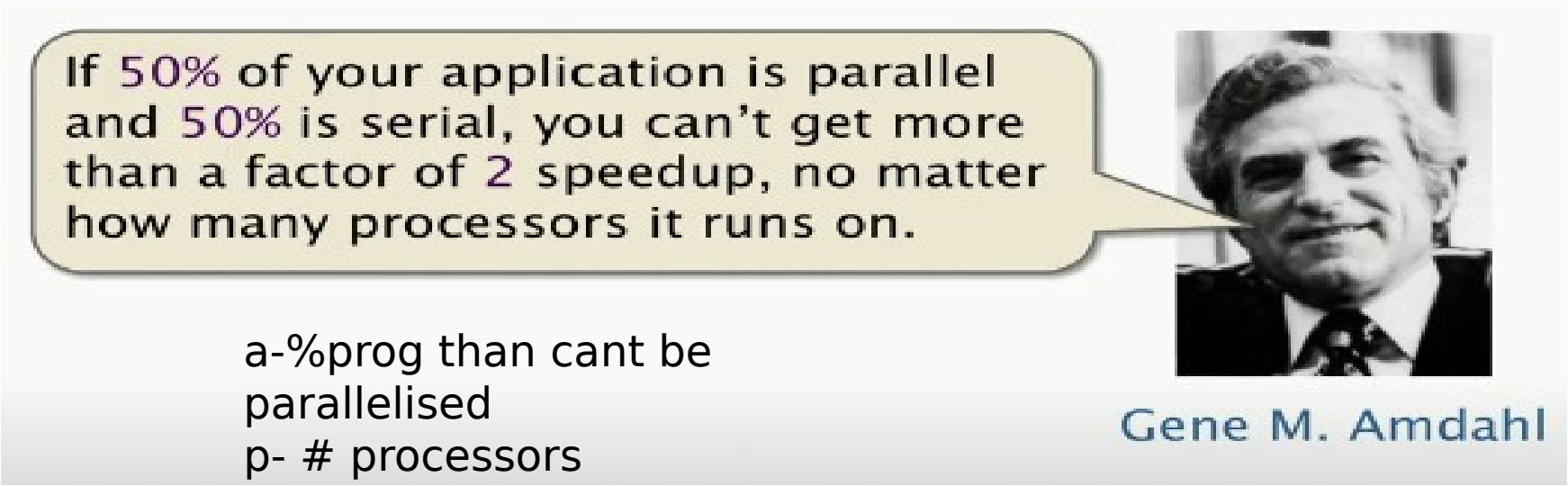
E.g. for $\alpha = 0.1$:

- hypothetical speedup on 16 processors = S(N,16) = 6.4
- hypothetical speedup on 1024 processors = S(N,1024) = 9.9
- ...
- maximum theoretical speed up is 10.0

Gustafson's Law

We need larger problems for larger numbers of processors

Amdahl's law



a-%prog than cant be parallelised
p- # processors
s=1/(a+(1-a)/p)

E.g. for a = 0.1:

- hypothetical speedup on 16 processors = S(N,16) =1/(0.1+0.9/16)= 6.4
- hypothetical speedup on 1024 processors = S(N,1024) = 9.9
- ...
- maximum theoretical speed up is 10.0

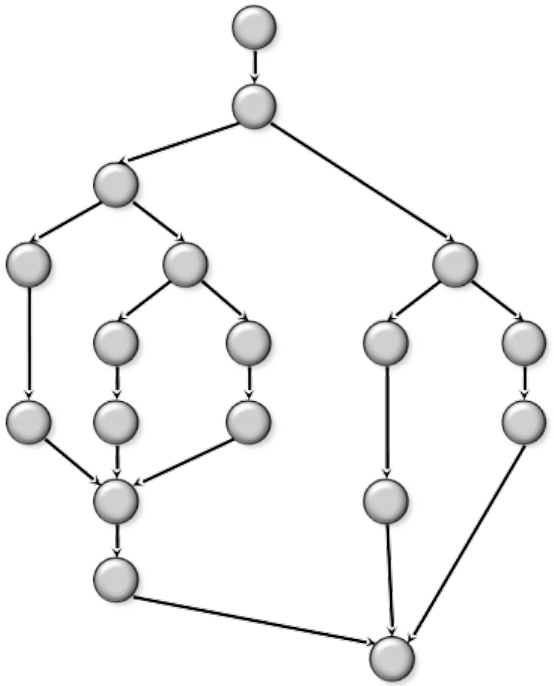
Gustafson's Law

We need larger problems for larger numbers of processors

Work Span Model

Work / Span Model

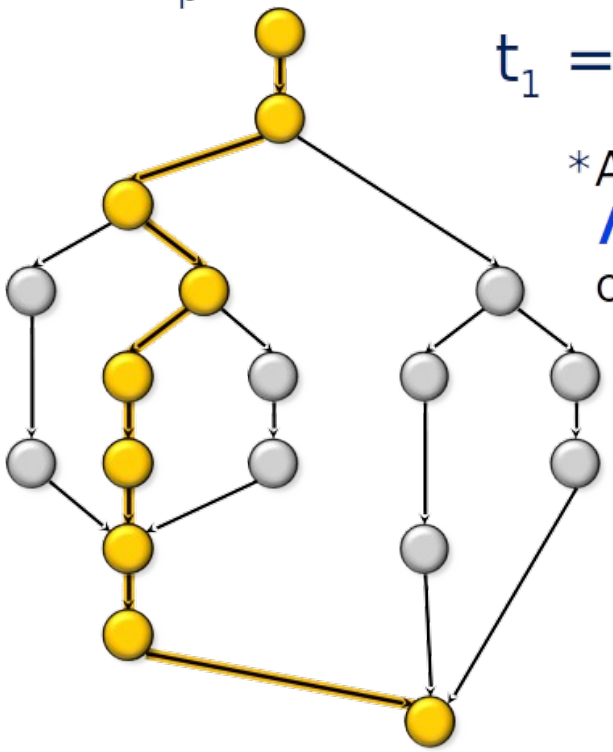
t_p = execution time on p processors



Work Span Model

Work / Span Model

t_p = execution time on p processors



t_1 = ***work*** t_∞ = ***span*** *

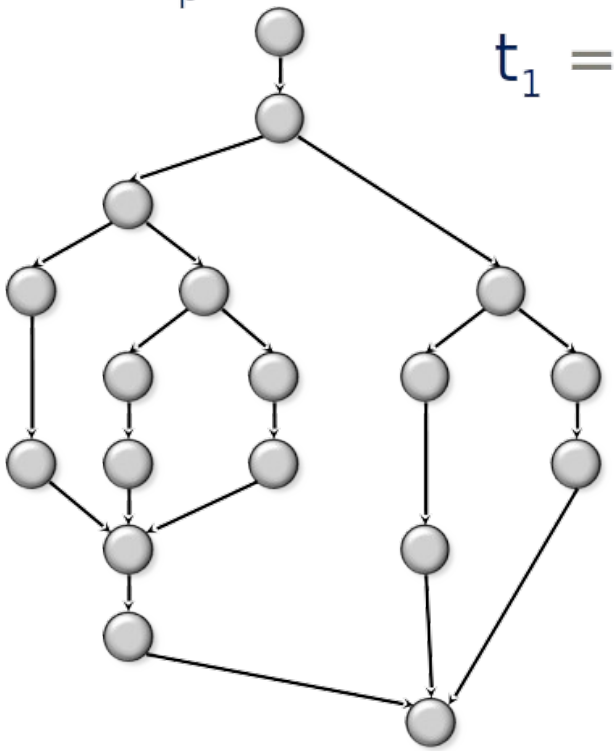
*Also called ***critical-path length*** or ***computational depth***.

Work Span Model

Work / Span Model

t_p = execution time on p processors

t_1 = ***work*** t_∞ = ***span*** *



WORK LAW

- $t_p \geq t_1/p$

SPAN LAW

- $t_p \geq t_\infty$

Speedup

Def. t_1/t_P = speedup on p processors.

If t_1/t_P = O(p), we have linear speedup,

= p, we have perfect linear speedup,

> p, we have superlinear speedup,

(which is not possible in this model,because of the Work Law $t_p \geq t_1/p$)

<p normally

Parallel Program Examples

Addition of n numbers

Fibonacci series

Prime number - Finding, generation

prefix sum

Factorial computation

MPI Structure (Initilaization)

- **int MPI_Init (int *argc, char ***argv)**
- **int MPI_Comm_size (MPI_Comm comm, int *size)**

returns with the size of the group associated with the MPI communicator, comm.

The communicator, MPI_COMM_WORLD is predefined to include all processes. Communicator has two distinguishing characteristics:

Group Name - a unique name given to a collection of processes

Rank - unique integer id within the group: assigned to the process by the system, contiguous and start from 0 within each group

Group and rank serve to uniquely identify each process

Context - the context defines the communication domain and is allocated by the system at run time

- **int MPI_Comm_rank (MPI_Comm comm, int *rank)**

returns with the rank of the processor within the group comm.

MPI : Code Implementation Template

1	2	3	4
MPI_Init	**MPI_Init**	**MPI_Init**	**MPI_Init**
------	**MPI_Comm_Size**	**MPI_Comm_Size**	**MPI_Comm_Size**
------	**MPI_Comm_rank**	**MPI_Comm_rank**	**MPI_Comm_rank**
------	**-----**	**MPI_Send**	**MPI_Send**
------	**-----**	**------**	**MPI_Barrier**
------	**-----**	**MPI_Recv**	**MPI_Recv**
MPI_Finalize	**MPI_Finalize**	**MPI_Finalize**	**MPI_Finalize**

Parallel Paradigm: MPI

```
//first.c
#include "mpi.h"
#include <stdio.h>

int main( int argc, char ** argv )
{
        MPI_Init( &argc, &Sarge );

        printf( "Hello world\n" );

        MPI_Finalize();

        return 0;
}
```

#include "mpi.h" provides basic MPI definitions and types

MPI_Init starts MPI

Note that all non-MPI routines are local; thus the printf run on each process

MPI_Finalize exits MPI

The MPICH implementation provides the commands mpicc for compilation

Simple program

How many processors are there? MPI_Comm_size.
Who am I ? MPI_Comm_rank.
The rank is a number between zero and size-1.
Compile program with mpicc and run with mpiexec -n 2 ./a.out

```
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 );
printf( "Hello world! I'm %d of %d\n",  rank, size );
MPI_Finalize();
return 0;
}
```

Process 1	Process 2
Hello world! I'm 0 of 2	**Hello world! I'm 1 of 2**

Message passing- Blocking: send and receive

int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)				
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)				
RED colour: Sender's info Green colour: Receiver's info				
Sr No.	Parameter	Information	MPI_Send	MPI_Recv
1	buf	starting address of buffer	I, send buffer	O, recv buffer
2	count	number of elements in buffer	I, send buffer	I
3	datatype	datatype of each buffer element e.g. MPI_INT	I	I
4	int dest, source	rank of process wildcard MPI_ANY_SOURCE	I, dest	I, source
5	tag	message tag wildcard MPI_ANY_TAG	I	I
6	comm	communicator	I	I
7	status	status structure containing a minimum of three entries, specifying the source, tag, and error code of the received message.	NA	O

Example of send/receive

```
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 );
if(rank == 0)  {
send_data("Hello Worker", size);
}
else {
receive_data(rank);
}
```

```
void send_data(char * data int size)
{
//int MPI_Send( void *buf, int count, MPI_Datatype
datatype, int dest,  int tag, MPI_Comm comm )
for( int i = 1 ; i< size ; i++ )
MPI_Send( data, strlen(data)+1, MPI_CHAR, i,  0,
MPI_COMM_WORLD );
}
```

```
void receive_data(int my_rank)
{
char r_data[500];
MPI_Status status;
//int MPI_Recv(void *buf, int count, MPI_Datatype
datatype,int source, int tag, MPI_Comm comm,
 MPI_Status *status)
MPI_Recv(r_data, 500, MPI_CHAR, 0, 0,
MPI_COMM_WORLD, &status) ;
printf("%s %d\n",r_data, my_rank);
}
```

Visualization of send/receive

```
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 );
}
```

```
//sender
// rank==0
void send_data(char * data int size)
{
for( int i = 1 ; i< size ; i++ )
MPI_Send( data, strlen(data)+1, MPI_CHAR, i,  0,
MPI_COMM_WORLD );
}
```

```
//Receiver
// rank <> 0
void receive_data(int my_rank)
{
char r_data[500];
MPI_Status status;
MPI_Recv(r_data, 500, MPI_CHAR, 0, 0,
MPI_COMM_WORLD, &status) ;
printf("%s %d\n",r_data, my_rank);
}
```

- **int MPI_Barrier(MPI_Comm comm)**

  Blocks until all members of the group comm have made this call.

  This synchronizes all processes.

  comm the name of the communicator

- **int MPI_Finalize(void)**

  Cleans up all MPI states.

  Should be called by all processes.

  User must ensure all pending communications complete before calling this routine.

- **MPI_Wtime ()**

  returns wall clock (elapsed) time in seconds from some arbitrary initial point

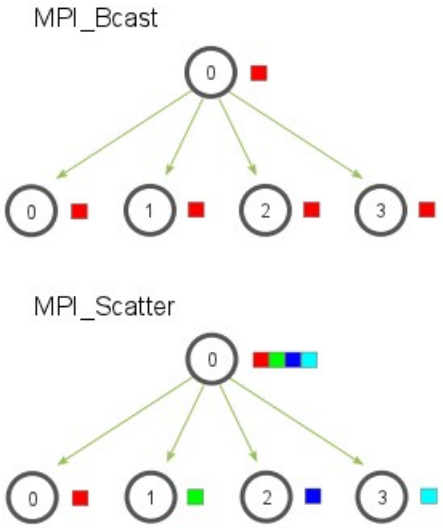
- **MPI_Wtick ()**

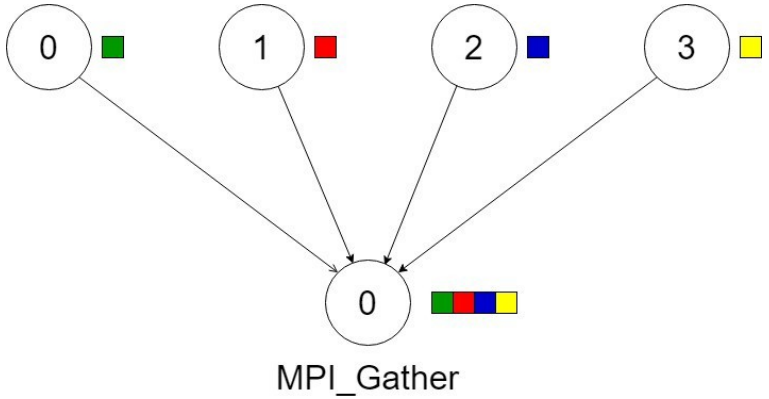
  returns resolution of MPI_Wtime in seconds

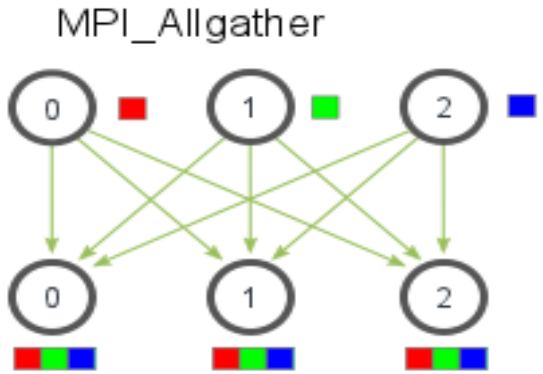
MPI broadcast and all-reduce

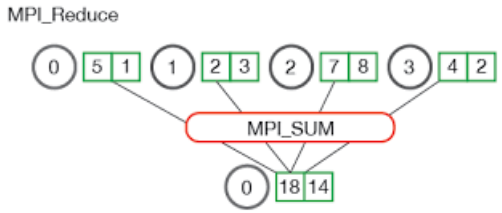
int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
int MPI_Allreduce(void *sendbuf, void *recvbuf, int count,MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

MPI Functions









Other MPI Functions:

- There are many additional functions, at least 133 functions
- These functions add flexibility, robustness, efficiency,modularity, or convenience.

MPI : Message Transfer

In Message communication, when

- **the data buffer is transmitted to the receiver, however, what if the receiver is not ready?**

  3 Scenarios:

1. Wait for receiver to be ready **(blocking)**
2. Copy the message to an internal buffer (on sender, receiver, or elsewhere) and then return from the send call **(nonblocking)**
3. Fail

- **the data buffer is to be received from sender, however, what if the sender has not sent the data**

  3 Scenarios:

1. Wait for sender to send **(blocking)**
2. Instruct the internal buffer for this transfer (on sender, receiver, or elsewhere) and then return from the send call **(nonblocking)**
3. Fail

Parallel Sum with MPI (Complete code)

```
#include <mpi.h>                                                    #include <stdio.h>
#include <stdlib.h>                                                 #include <unistd.h>
#define n 10                                                                    // size of array
int a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int a2[1000];                                                                   // Temporary array for slave
process
int main(int argc, char* argv[]) {
      int pid, np, elements_per_process, n_elements_recieved; // np -> no. of processes      // pid -> process id
            MPI_Status status;
            MPI_Init(&argc, &argv);                                 // Creation of parallel processes
            MPI_Comm_rank(MPI_COMM_WORLD, &pid);          // find out process ID, how
            MPI_Comm_size(MPI_COMM_WORLD, &np);           //many processes were started
            if (pid == 0) {                                                     // master process
                        int index, i;           elements_per_process = n / np;


                        // check if more than 1 processes are run
```

Parallel Sum with MPI: Part 1 : Intialization

```
#include <mpi.h>                                        #include <stdio.h>
#include <stdlib.h>                                     #include <unistd.h>
#define n 10                                            // size of array
int a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int a2[1000];                                           // Temporary array for slave process
int main(int argc, char* argv[]) {
        int pid, np, elements_per_process, n_elements_recieved; // np -> no. of processes
and pid -> process id
// Creation of parallel processes
        MPI_Init(&argc, &argv);                 MPI_Status status;
        // find out process ID, and how many processes were started
        MPI_Comm_rank(MPI_COMM_WORLD, &pid);
        MPI_Comm_size(MPI_COMM_WORLD, &np);
```

Parallel Sum with MPI: Part 2 : Master process :Work distribution

```
// master process to send data and count to slave
     if (pid == 0) {        int index, i;                          elements_per_process = n / np;
     for (i = 1; i < np - 1; i++) { // distributes the portion of array to child processes                  index
= i * elements_per_process;
    MPI_Send(&elements_per_process,1, MPI_INT, i, 0,MPI_COMM_WORLD);
    MPI_Send(&a[index],elements_per_process,MPI_INT, i, 0,MPI_COMM_WORLD);}
                    for (int sum=0,i = 0; i < elements_per_process; i++)
                            sum += a[i];                                  // master process add its own sub
array
                    int tmp;                                      // collects partial sums from other processes
                    for ( i = 1; i < np; i++)          {
MPI_Recv(&tmp,1,MPI_INT,MPI_ANY_SOURCE, 0,MPI_COMM_WORLD, &status);
                    int sender = status.MPI_SOURCE;
                sum += tmp;                          }
// master process to display result
                    printf("Sum of array is : %d\n", sum);                  // prints the final sum of array   }
```

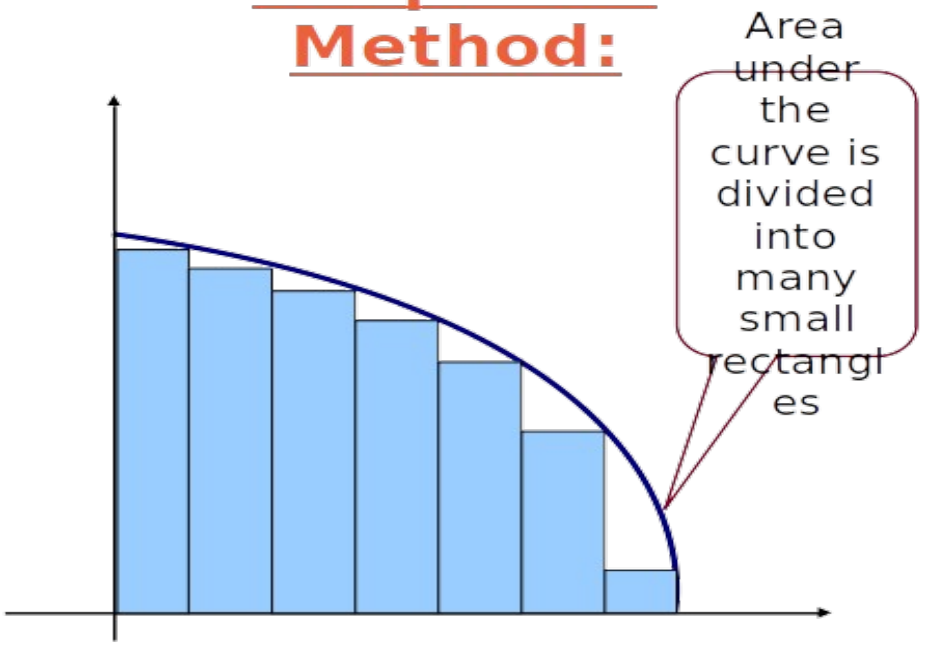
Parallel Sum with MPI: Part 3: Slave

```
        else    {
MPI_Recv(&n_elements_recieved,1, MPI_INT, 0, 0,MPI_COMM_WORLD,&status);
MPI_Recv(&a2, n_elements_recieved,MPI_INT, 0, 0,MPI_COMM_WORLD,&status);
        // calculates its partial sum
        int partial_sum = 0;
        for (int i = 0; i < n_elements_recieved; i++)
        partial_sum += a2[i];
// sends the partial sum to the root process
MPI_Send(&partial_sum, 1, MPI_INT,0, 0, MPI_COMM_WORLD);
                        }
MPI_Finalize();
        return 0;
```

Example: PI Calculation

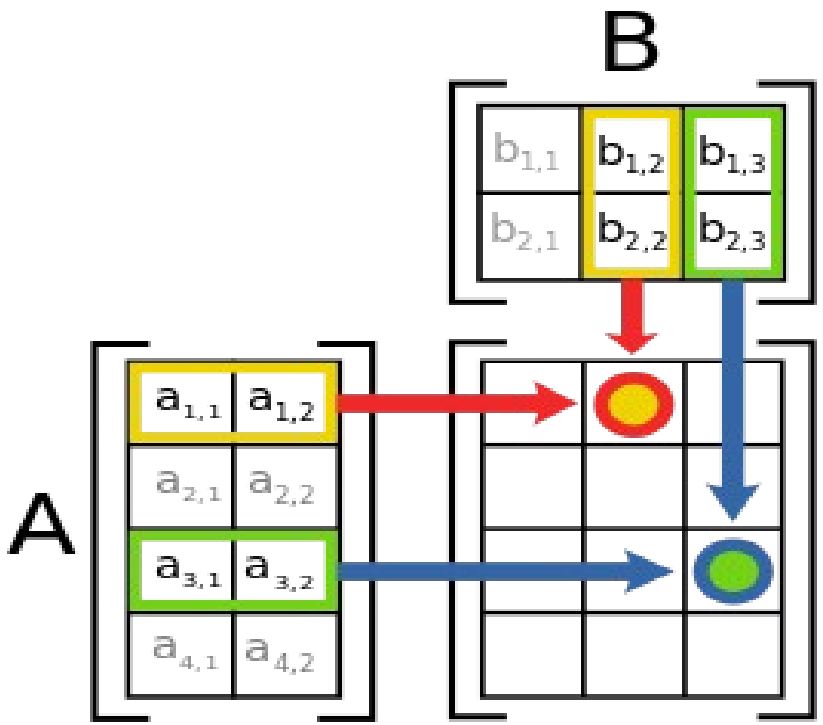
Example - PI

Simpson's Method:



Area under the curve (Approx.) = Sum of the areas of all rectangles

Matrix Multiplication



C=A*B (Compatibility: Rows(A)=Columns(B)

Summary: Parallel Terminology

- Concurrent computing – a program is one in which multiple tasks can be in progress at any instant.
- Parallel computing – a program is one in which multiple tasks cooperate closely to solve a problem
- Distributed computing – a program may need to cooperate with other programs to solve a problem.

Thank You