What is High Performance Computing?

V. Sundararajan
Scientific and Engineering Computing Group
Centre for Development of Advanced Computing
Pune 411 007
vsundar@cdac.in

Plan

- Why HPC?
- Parallel Architectures
- Current Machines
- Issues
- Arguments against Parallel Computing
- Introduction to MPI
- A Simple example

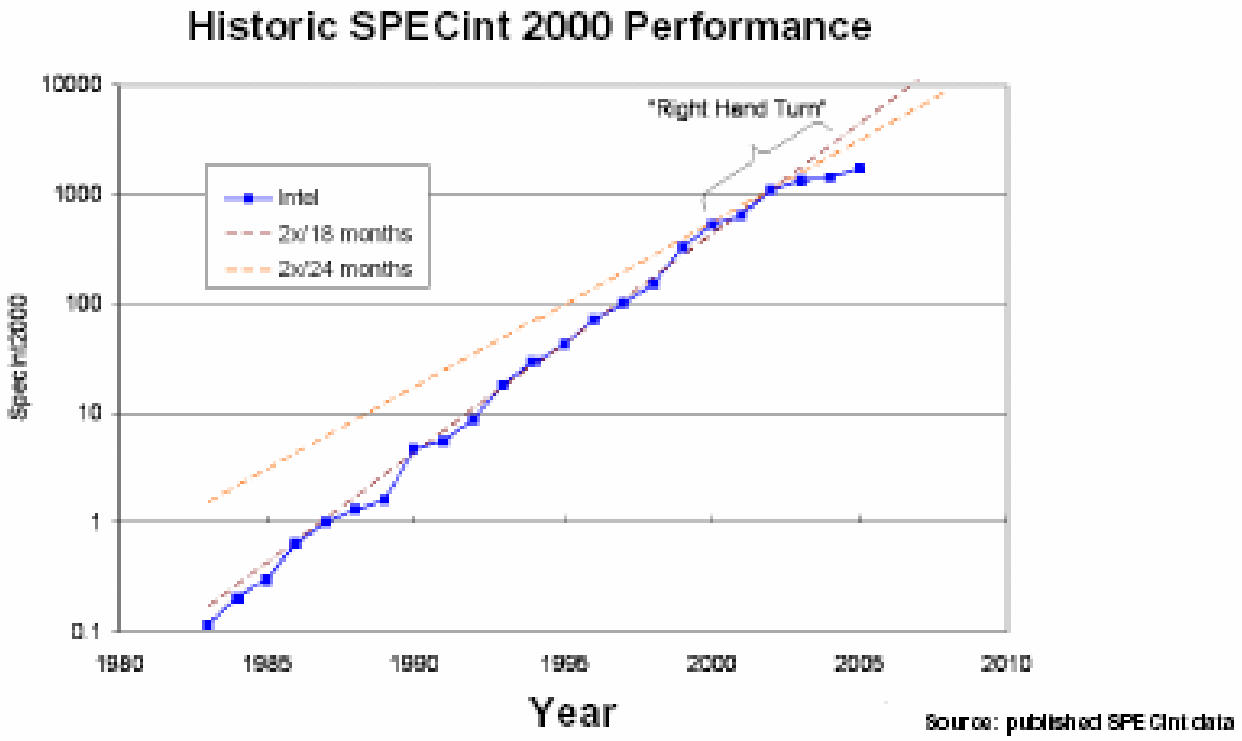
Definition

Supercomputing

Parallel computing: Parallel Supercomputers

High Performance Computing: includes computers, networks, algorithms and environments to make such systems usable. (range from small cluster of PCs to fastest supercomputers)

The bad news:
Single thread performance is falling off



Source: http://www3.intel.com/cd/software/products/asmo-na/eng/337441.htm

Increased Cores add value

Physical Limits of single processor speed– 2007
(Predicted by Paul Messina in 1997)

Pentium	3600 MHz	<0.3 ns
Light travels	30 cm in	1 ns

Signal is already 10% of its speed

Red-Shift:

Reduction in clock speed

Why HPC?

To simulate a bio-molecule of 10000 atoms
Non-bond energy term ~ 10^8 operations
For 1 microsecond simulation ~ 10^9 steps
~ 10^{17} operations
On a 500 MFLOPS machine (5×10^8 operations per second) takes 2×10^9 secs (About 60 years)
(This may be on a machine of 5000 MFLOPS peak)

Need to do large no of simulations for even larger molecules

Why HPC?

Biggest CCSD(T) calculation:
David A. Dixon,
(http://www.nrel.gov/crnare_workshop/pdfs/dixon_plenary.pdf)
Hydrocarbon Fuels: Combustion - Need Heats of Formation to predict reaction equilibria and bond energies;
octane(C8H18);
aug-cc-pVQZ= 1468 basis functions, 50 electrons;
Used NWChem on 1400 Itanium-2 processors (2 and 4 GB/proc) on the EMSL MSCF Linux cluster.

Took 23 hours to complete (3.5 yr on a desktop machine).

Parallel Architectures

1. Single Instruction Single Data (SISD) Sequential machines.
2. Multiple Instructions Single Data (MISD)
3. Single Instruction Multiple Data (SIMD)

   Array of processors with single control unit.

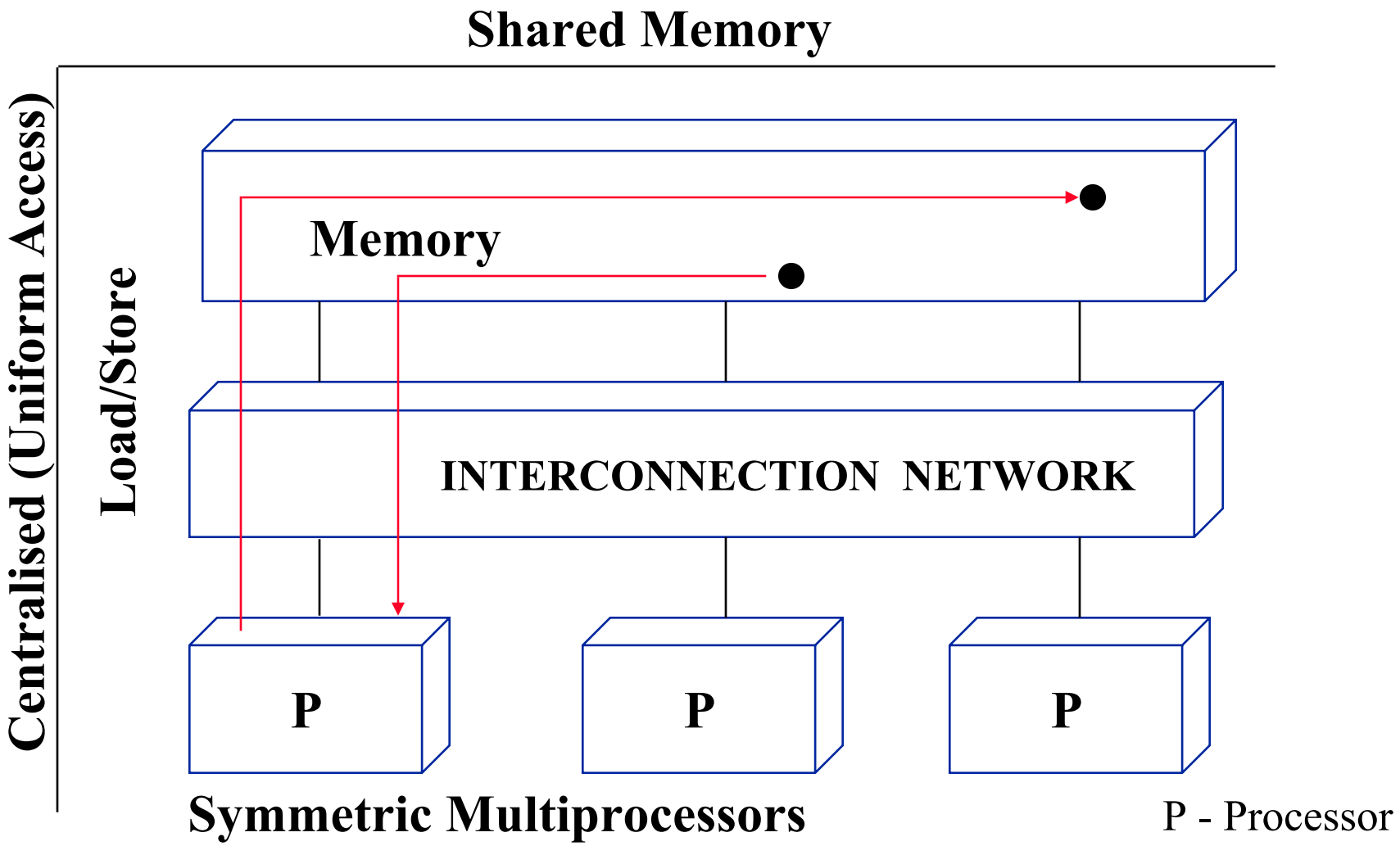
   Connection Machine (CM - 5)

4. Multiple Instructions Multiple Data (MIMD)

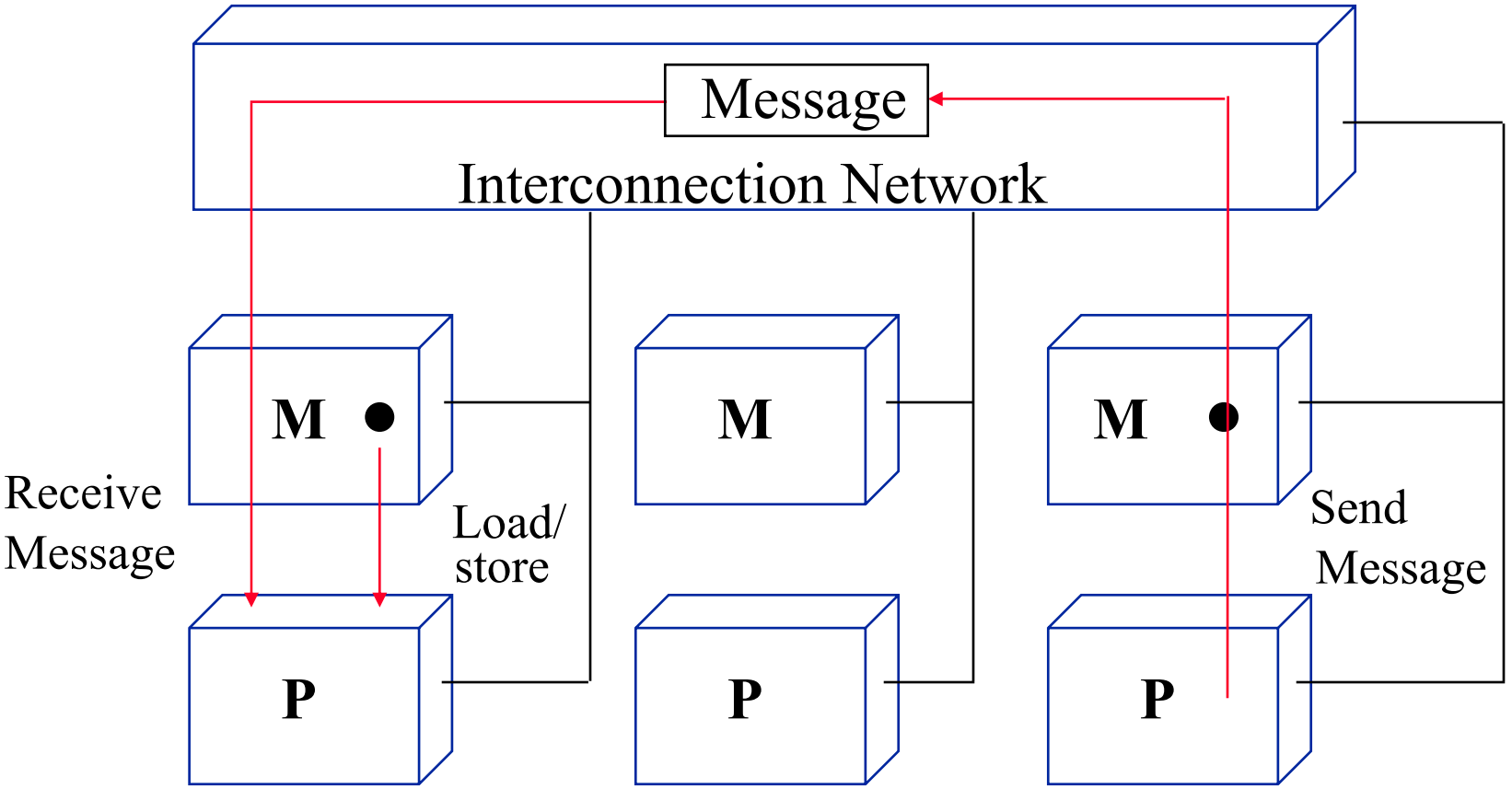
   Several Processors with several Instructions and Data Stream.

   All the recent parallel machines

Tightly Coupled MIMD



Loosely Coupled MIMD



Distributed Message-Passing Machines

Also Called Message Passing Architecture

M - Memory

P - Processor

Architectures

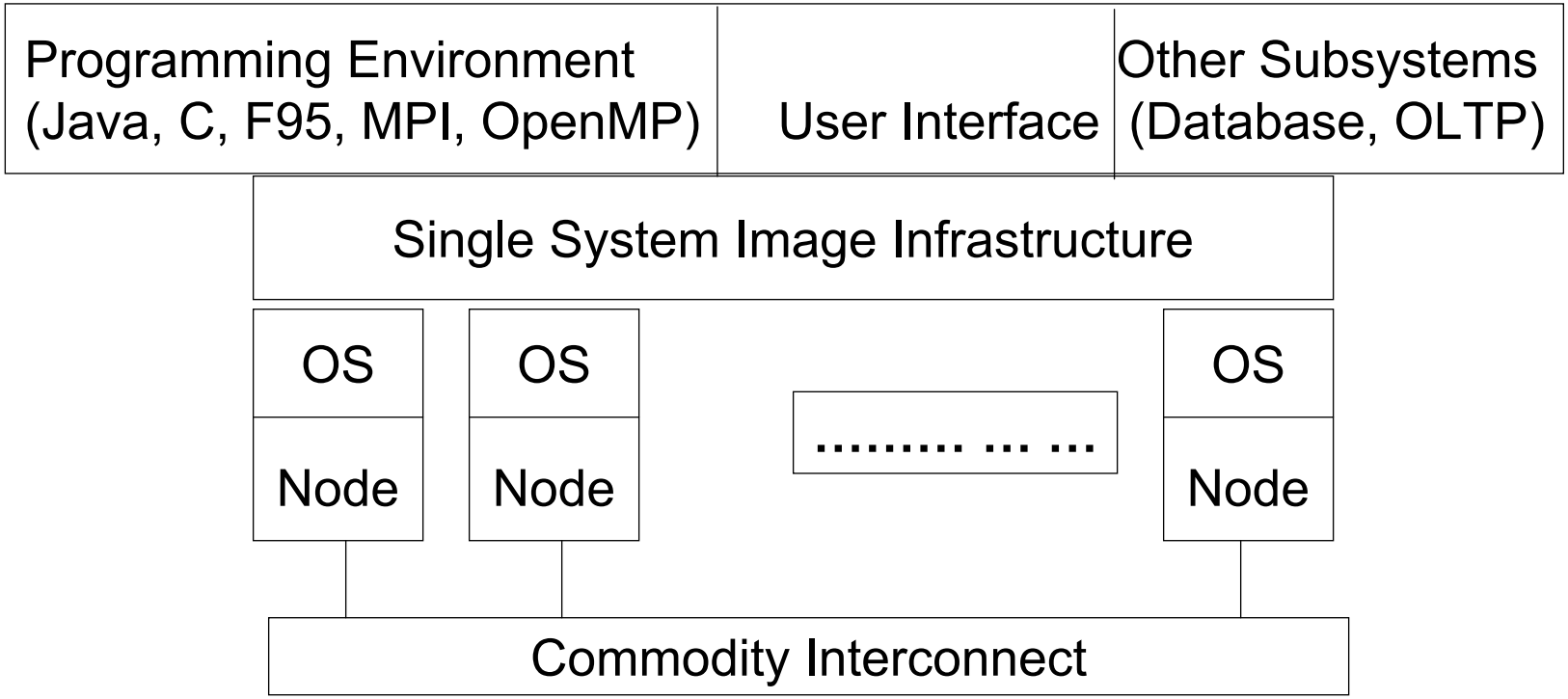
Shared Memory :

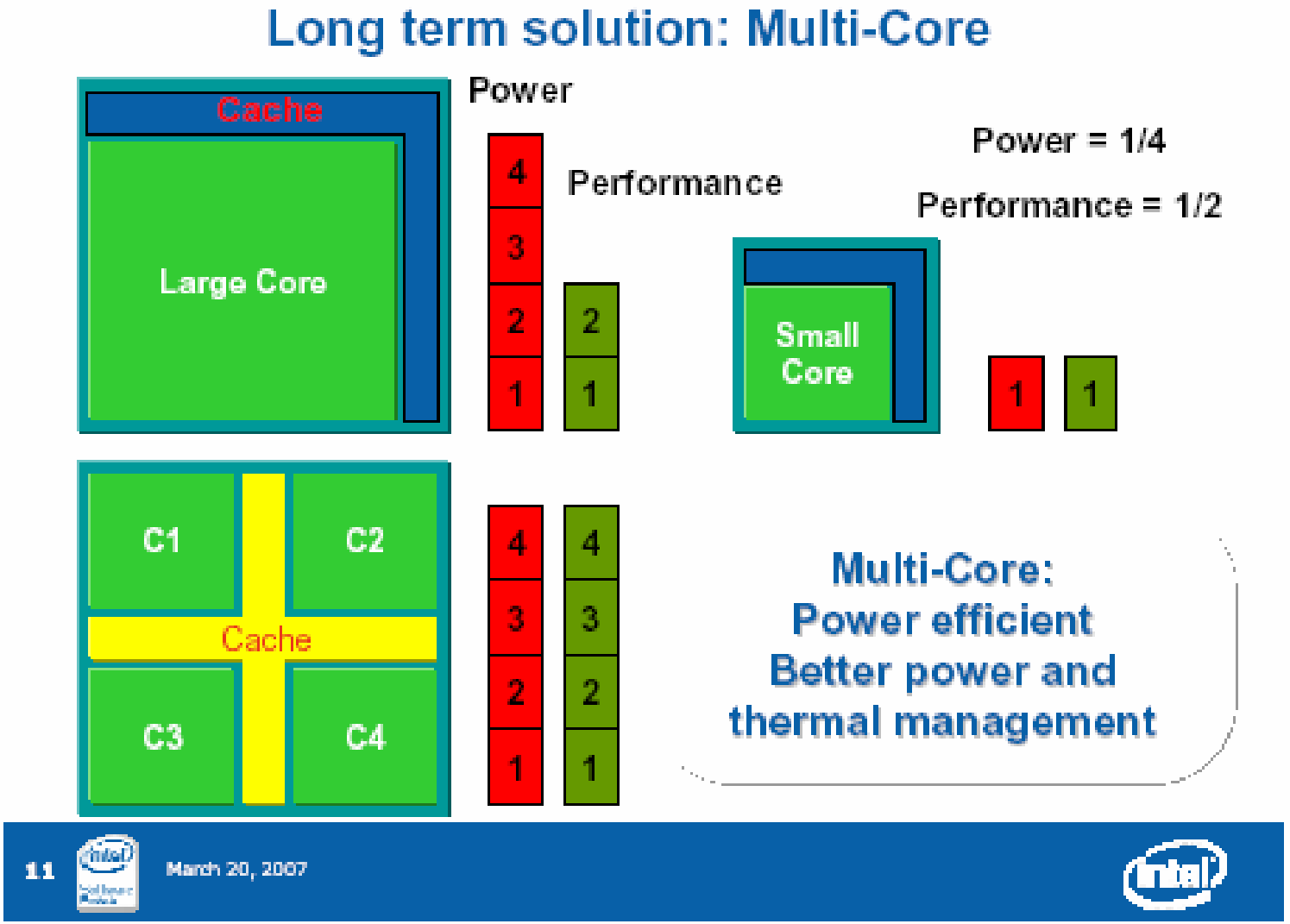
- Scalable up to about 64 or 128 processors but costly
- Memory contention problem
- Synchronisation problem
- Easy to code

Distributed Memory :

- Scalable up to larger no. of processors
- Message passing overheads
- Latency hiding
- Difficult to code

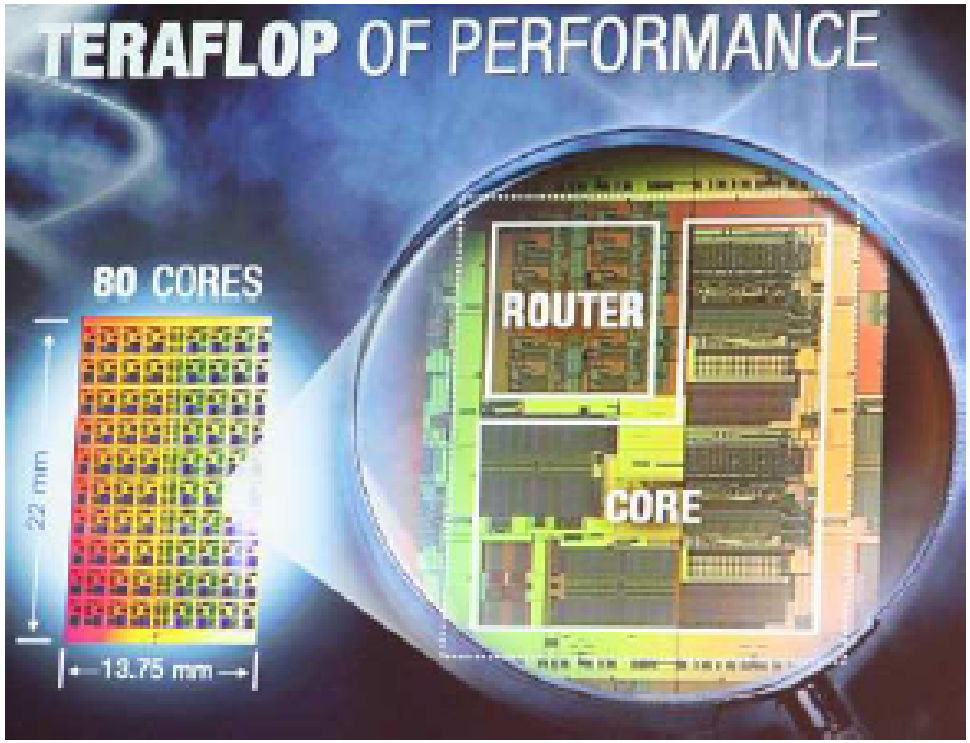
Cluster of Computers





Source: http://www3.intel.com/cd/software/products/asmo-na/eng/337441.htm

A many core example: Intel's 80 core test chip



Source: A 80-tile 1.28 TFLOP Network-on-Chip in 65 nm CMOS, ISSCC'07, Sriram Vangal, Jason Howard, Gregory Ruhl, Saurabh Dighe, Howard Wilson, James tschanz, David Finan, Priya Iyer, Arvind Singh, Raju Jacob, Shailendra Jain, Sriram venkataraman, Yatin Hoskote and Nitin Borkar.

Performance numbers* at 4.27 Ghz:

peak performance:
* 1.37 SP TFLOPS

Explicit PDE solver:

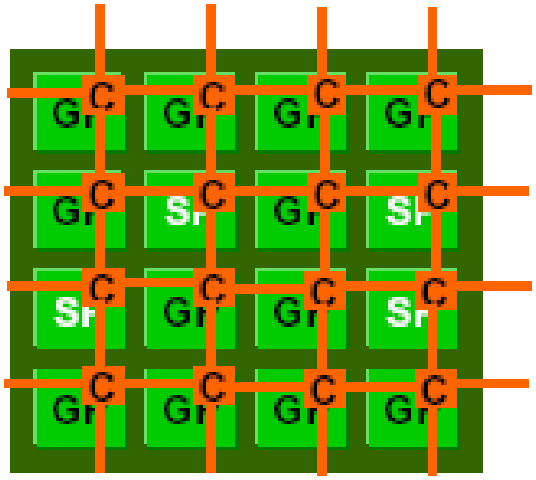
* 1 SP TFLOPS

matrix multiplication:
* 0.51 SP TFLOPS

* 2 independent FMAC units – each can retire 2 Single Precision FLOPS (+ and *) per cycle.

Source: http://www3.intel.com/cd/software/products/asmo-na/eng/337441.htm

Future Multi-core Platform



General Purpose Cores

Special Purpose HW

Interconnect fabric

Heterogeneous Multi-Core Platform

This is an architecture concept that may or may not be reflected in future products from Intel Corp.

Source: http://www3.intel.com/cd/software/products/asmo-na/eng/337441.htm

Simulation Techniques

N-body simulations
Finite difference
Finite Element
Pattern evolution

Classic methods -
Do Large scale problems
Faster
New class of Solutions

Parallel Problem Solving Methodologies

- Data parallelism
- Pipelining
- Functional parallelism

Parallel Programming Models

1. Directives to Compiler (OpenMP, threads)
2. Using Parallel libraries (SCALAPACK, PARPACK)
3. Message passing (MPI)

Programming Style

SPMD : Single program multiple Data.

Each processor executes the same program but with different data.

MPMD : Multiple programs, multiple data.

Each processor executes a different program. Usually this is a master slave model.

Performance Issues

1. **Speed up** $= \dfrac{\text{Time for sequential code}}{\text{Time for parallel code}}$

$$S_P = \frac{Ts}{Tp} \qquad 1 \leq S_P \leq P$$

2. **Efficiency** $E_P = \dfrac{Sp}{p} \qquad 0 < E_P < 1$

$E_P = 1 \Rightarrow S_P = P$ 100% efficient

Communication Overheads

Latency

Startup time for each message transaction 1 μs

Bandwidth

The rate at which the messages are transmitted across the nodes / processors 10 Gbits / Sec.

Strongest argument

Amdahl's Law

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

f = Sequential part of the code.

Ex. f = 0.1 assume P = 10 processes

$$S = \frac{1}{0.1 + (0.9) / 10}$$

$$= \frac{1}{0.1 + (0.09)} \cong 5$$

As $P \longrightarrow \infty$ $S \longrightarrow 10$

Whatever we do, 10 is the maximum speedup possible.

What is MPI?

An industry-wide standard protocol for passing messages between parallel processors.

Small: Require only six library functions to write any parallel code

Large: There are more than 200 functions in MPI-2

What is MPI?

It uses ‘communicator’ a handle to identify a set of processes and ‘topology’ for different pattern of communication

A communicator is a collection of processes that can send messages to each other.

A topology is a structure imposed on the processes in a communicator that allows the processes to be addressed in different ways

Can be called by Fortran, C or C++ codes

Small MPI

MPI_Init
MPI_Comm_size
MPI_Comm_rank
MPI_Send
MPI_Recv
MPI_Finalize

MPI_Init

Initialize the MPI execution environment

Synopsis

include "mpif.h"
Call MPI_Init(Error)

Output Parameter

Error – Error value (integer) – 0 means no error

Default communicator *MPI_COMM_WORLD* is initialised

MPI_Comm_size

Determines the size of the group associated with a communictor

Synopsis

include "mpif.h"
Call MPI_Comm_size (MPI_COMM_WORLD,size,Error)

Input Parameter

MPI_COMM_WORLD – default communicator (handle)

Output Parameter

size **-** number of processes in the group of communicator *MPI_COMM_WORLD* (integer)

Error – Error value (integer)

MPI_Comm_rank

Determines the rank of the calling process in the communicator

Synopsis

include "mpif.h"
Call MPI_Comm_rank (MPI_COMM_WORLD, myid, Error)

Input Parameters

MPI_COMM_WORLD – default communicator (handle)

Output Parameter

myid - rank of the calling process in group of communicator *MPI_COMM_WORLD* (integer)
Error – Error value (integer)

MPI_Send

Performs a basic send

Synopsis

include "mpif.h"
Call MPI_Send(buf, count, datatype, dest, tag, MPI_Comm_World,Error)

Input Parameters

buf	initial address of send buffer (choice)
count	number of elements in send buffer (nonnegative integer)
datatype	data type of each send buffer element (handle)
dest	rank of destination (integer)
tag	message tag (integer)
comm	communicator (handle)

Output Parameter

Error	Error value (integer)

Notes

This routine may block until the message is received.

MPI_Recv

Basic receive

Synopsis

include "mpif.h"
Call MPI_Recv(buf, count, datatype, source, tag, MPI_Comm_World, status, Error)

Output Parameters

buf	initial address of receive buffer (choice)
status	status object (Status)
Error	Error value (integer)

Input Parameters

count	maximum number of elements in receive buffer (integer)
datatype	datatype of each receive buffer element (handle)
source	rank of source (integer)
tag	message tag (integer)

MPI_Comm_World communicator (handle)

Notes

The count argument indicates the maximum length of a message; the actual number can be determined with *MPI_Get_count*.

MPI_Finalize

Terminates MPI execution environment

Synopsis

include "mpif.h"
Call MPI_Finalize()

Notes

All processes must call this routine before exit. The number of processes running *after* this routine is called is undefined; it is best not to perform anything more than a *return* after calling *MPI_Finalize*.

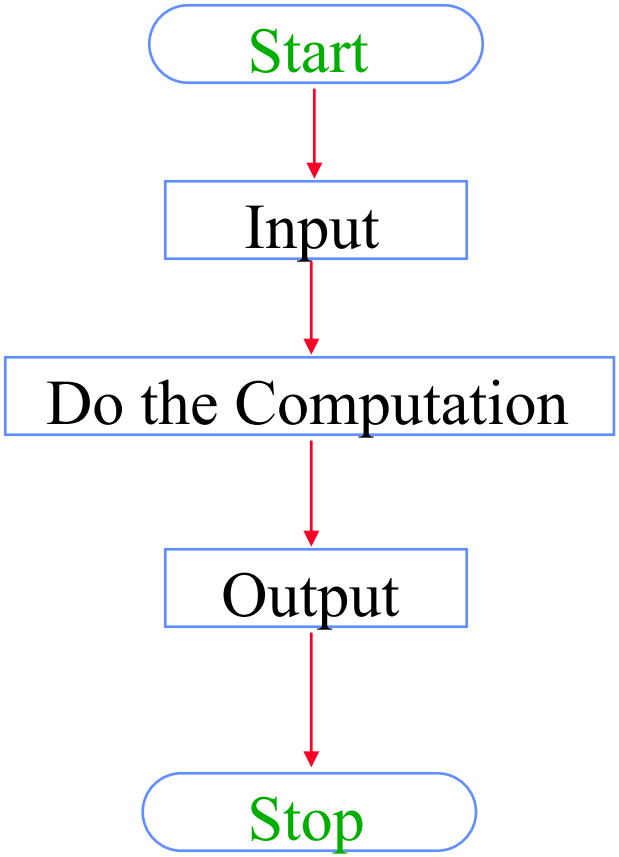
Examples

Summation

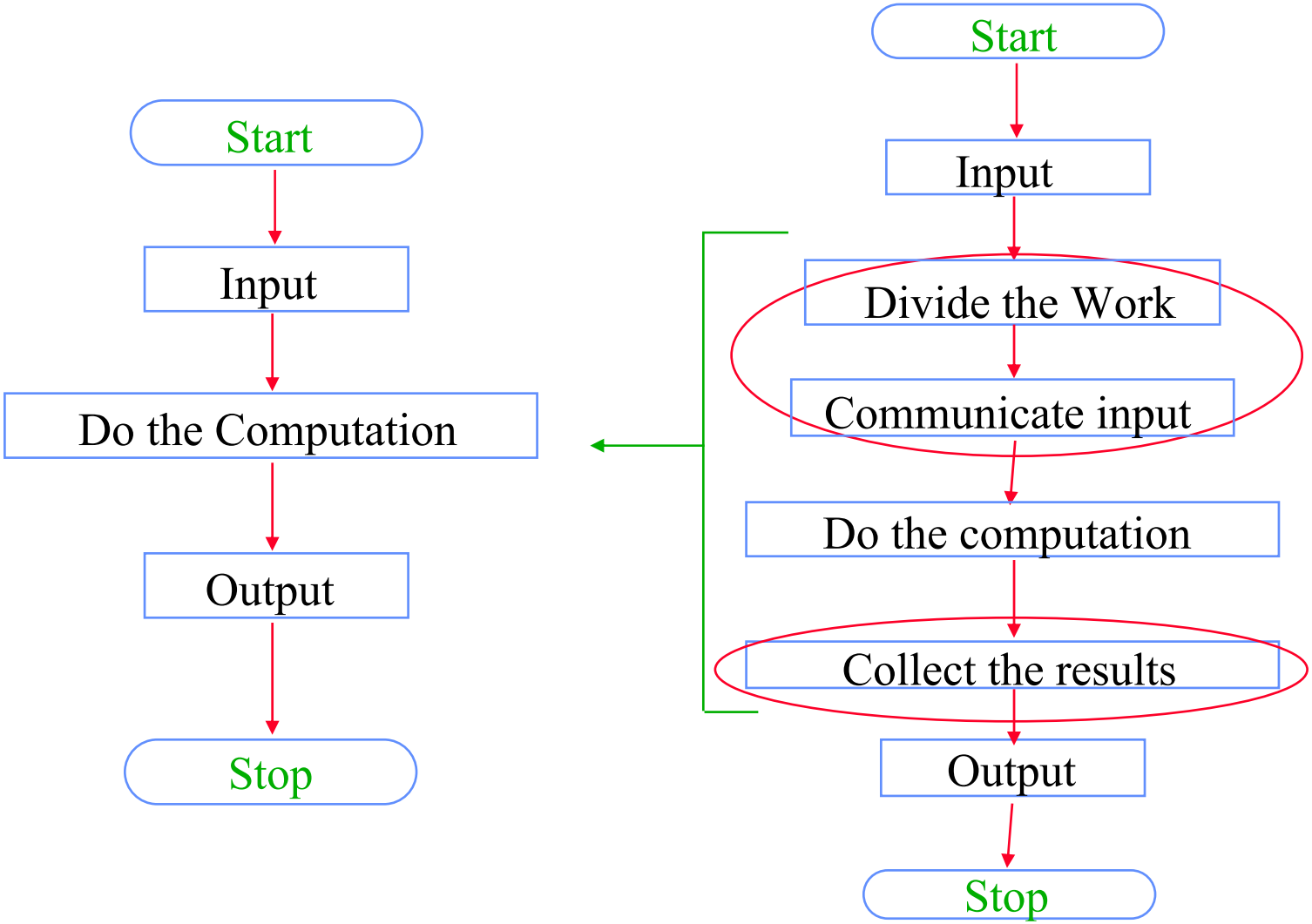
Calculation of pi

Matrix Multiplication

Sequential Flow



Parallel using MPI



Simple Example

Let us consider summation of 10^{12} real numbers between 0 and 1.

– To compute $\sum_i x_i$ for (i=1,2,…… 10^{12})

This may sound trivial but, useful in illustrating the practical aspects of parallel program development

Total computation of 10^{12} operations to be done

On a PC, it would take roughly 500 secs assuming 2 GFLOPS sustained performance

Can we speed this up and finish in $1/8^{th}$ on 8 processors

Sequential Code

```
program summation
Implicit none
double precision x,xsum
integer i,iseed
xsum=0.0

do i=1,1000000000000
     x=float(mod(i,10))/1000000.0
    xsum=xsum+x
enddo

print *,'xsum= ',xsum

stop
end
```

Parallel Code – SPMD style

```
program summation
Implicit none
double precision x,xsum
integer i



xsum=0.0
do i=1,1000000000000
     x=float(mod(i,10))/1000000.0
    xsum=xsum+x
enddo


print *,'xsum= ',xsum
stop
end
```

```
program summation
include "mpif.h"
Implicit none
double precision x,xsum,tsum
integer i
Integer myid,nprocs,IERR
Call MPI_INIT(IERR)
Call MPI_COMM_RANK(MPI_COMM_WORLD,myid,IERR)
Call MPI_COMM_SIZE(MPI_COMM_WORLD,nprocs,IERR)
xsum=0.0
do i=myid+1,1000000000000,nprocs
    x=float(mod(i,10))/1000000.0
    xsum=xsum+x
enddo
Call MPI_REDUCE(xsum,tsum,1,MPI_DOUBLE_PRECISION,
                        MPI_SUM,0,MPI_COMM_WORLD,IERR)
If(myid.eq.0) print *,'tsum= ',tsum
stop
end
```

Sequential vs Parallel

Sequential	Parallel
Design algorithm step by step and write the code assuming single process to be executed	Redesign the same algorithm with the distribution of work assuming many processes to be executed simultaneously
f90 –o sum.exe sum.f gcc –o sum.exe sum.c Only one executable created	mpif90 –o sum.exe sum.f mpicc –o sum.exe sum.f Only one executable created
sum.exe Executes a single process	mpirun –np 8 sum.exe Executes 8 processes simultaneously
Single point of failure Good tools available for debugging	Cooperative operations but, multiple points of failure; Tools are still evolving towards debugging parallel executions.

A Quote to conclude

James Bailey
(New Era in Computation Ed. Metropolis & Rota)

We are all still trained to believe that orderly, sequential processes are more likely to be true. As we were reshaping our computers, so simultaneously were they reshaping us. May be when things happen in this world, they actually happen in parallel