An Overview of Parallel computing

Jayanti Prasad

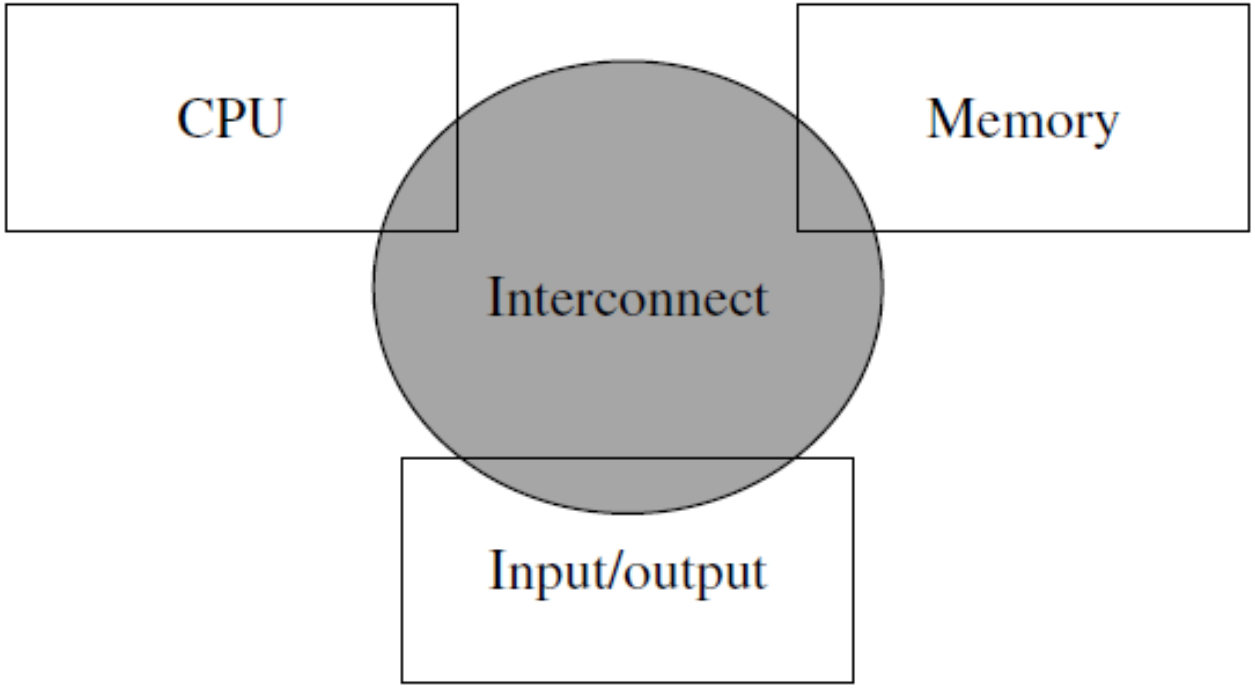
Inter-University Centre for Astronomy & Astrophysics
Pune, India (411007)

May 20, 2011

Plan of the Talk

- Introduction
- Parallel platforms
- Parallel Problems & Paradigms
- Parallel Programming
- Summary and conclusions

High-level view of a computer system



Why parallel computation ?

- The speed with which a single processor can process data (number of instructions per second, FLOPS) cannot be increased indefinitely.
  - It is difficult to cool faster CPUs.
  - Faster CPUs demand smaller chip size which again creates more heat.
- Using a fast/parallel computer :
  - Can solve existing problems/more problems in less time.
  - Can solve completely new problems leading to new findings.
- In Astronomy High Performance Computing is used for two main purposes
  - Processing large volume of data
  - Dynamical simulations

References :
Ananth Grama et. al. 2003, *Introduction to Parallel Computing*, Addison Wesley

Kevin Dowd & Charles R. Severance, *High Performance Computing*, OReily

Performance

Computation time for different problems may depend on the followings:

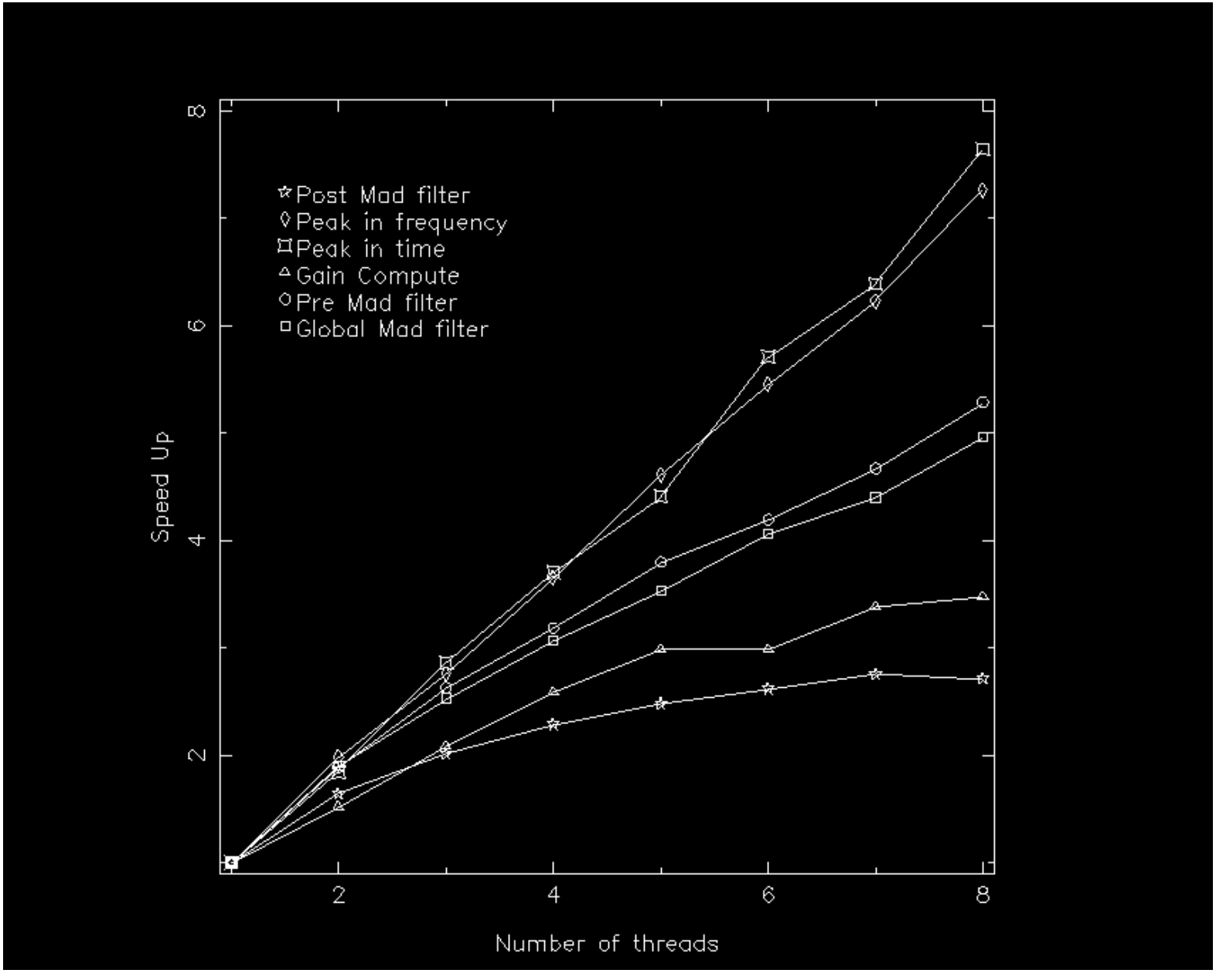
- **Input-Output:-** problems which require a lot of disk read/write.
- **Communication:-** Problems, like dynamical simulations need a lot of data to be communicated among processors. Fast inter-connects (Gig-bit Ethernet, Infiniband, Myrinet etc.) are highly recommended.
- **Memory:-** Problems which need a large amount of data to be available in the main memory (DRAM), demand more memory. Latency and bandwidth are very important.
- **Processor:-** Problems in which a large number or computations have to be done.
- **Cache :-** Better memory hierarchy (RAM,L1,L2,L3..) and efficient use of that benefits all problems.

Performance measurement and benchmarking

- **FLOPS:-** Computational power is measured in units of Floating Point Operations Per Second or FLOPS (Mega, Giga, Tera, Peta,.. FLOPS).
- **Speed UP:-**

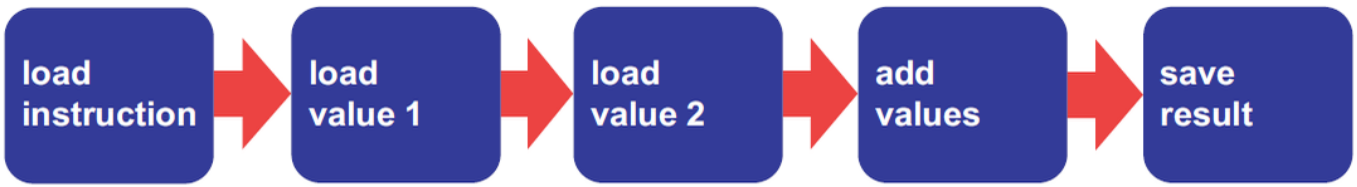
$$\text{Speed Up} = \frac{T_1}{T_N} \tag{1}$$

where T_1 is the time needed to solve a problem on one processor and T_n is the time needed to solve that on N processors.


Post Mad filter
Peak in frequency
Peak in time
Gain Compute
Pre Mad filter
Global Mad filter
Speed Up
Number of threads
2
4
6
8
2
4
6
8

Pipeline

If we want to add two numbers we need five computational steps:



If we have one functional unit for every step then the addition still requires five clock cycles, however, all the units being busy at the same time, one result is produced every cycle. Reference : Tobias Wittwer 2005, *An Introduction to Parallel Programming*

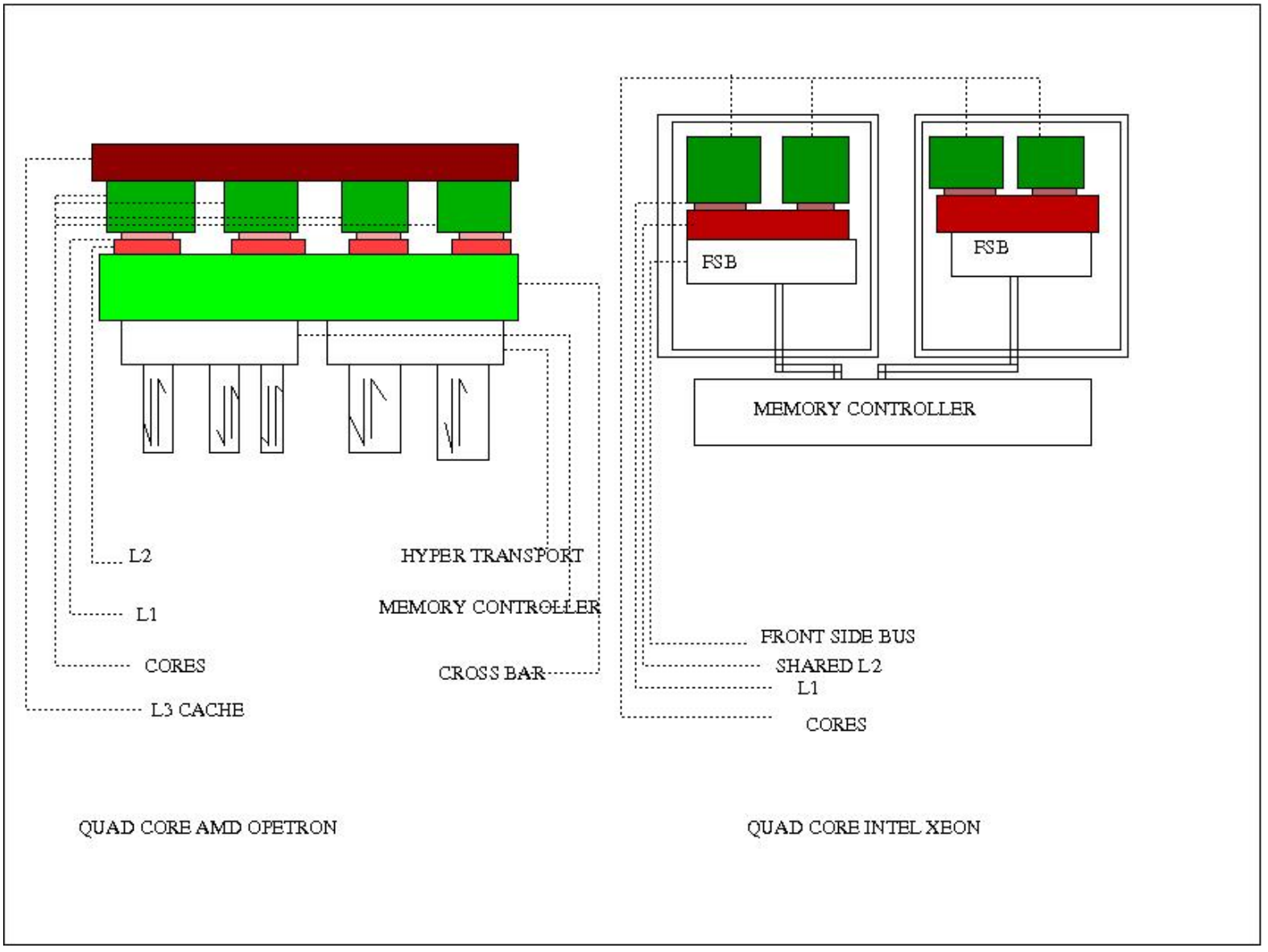
Vector Processors

- A processor that performs one instruction on several data sets is called a vector processor.
- The most common form of parallel computation is in the form of Single Instruction Multiple Data (SIMD) i.e., same computational steps are applied on different data sets.
- Problems which can be broken into small problems for parallelization are called embarrassingly parallel problems e.g., SIMD.

Multi-core Processors

- That part of a processor which execute instructions in an application is called a core.
- Note that in a multiprocessor system we can have each processor having one core or more than one cores.
- In general the number of hardware threads are equal to the number of cores.
- In some processors one core can support more than one software threads.
- A multi-core processor presents multiple virtual CPUs to the user and operating system which are not physically countable but OS can give you clue (check /proc).

Reference : Darryl Gove, 2011, *Multi-core Application Programming*, Addison-Wesley


L2
L1
CORES
L3 CACHE
HYPER TRANSPORT
MEMORY CONTROLLER
CROSS BAR
FSB
FSB
MEMORY CONTROLLER
FRONT SIDE BUS
SHARED L2
L1
CORES
QUAD CORE AMD OPETRON
QUAD CORE INTEL XEON

Shared Memory System

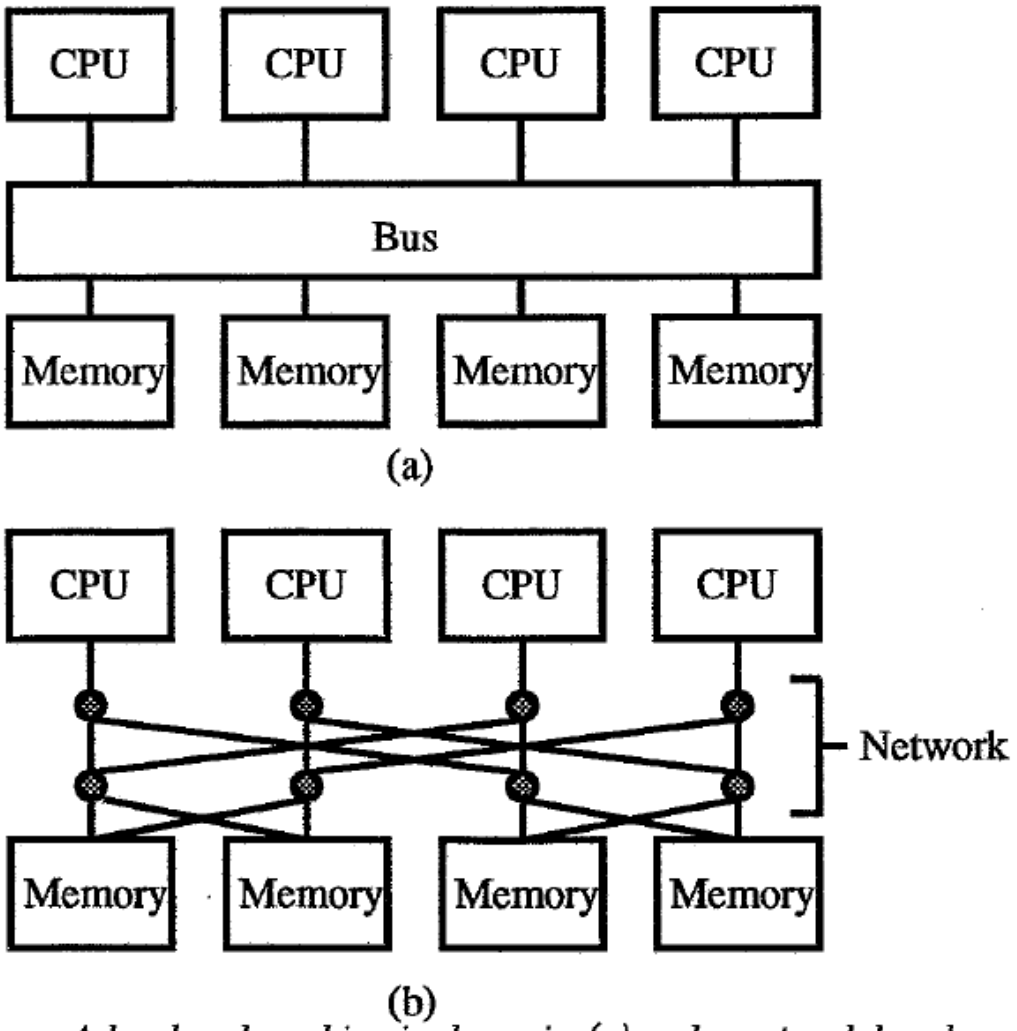


FIG. 2. *Shared memory. A bus-based machine is shown in (a) and a network-based machine is shown in (b).*

Distributed Memory system

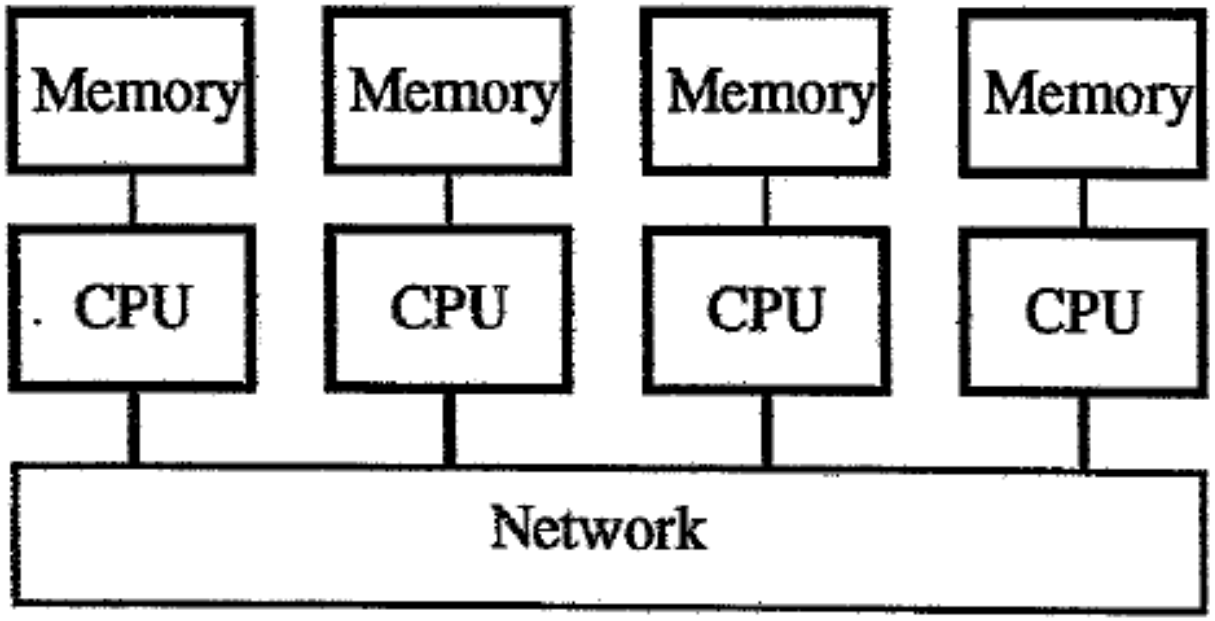


FIG. 3. *Distributed memory*

Topology : Star, N-Cube, Torus ...

Parallel Problems

- Scalar Product

$$S = \sum_{i=1}^{N} A_i B_i \tag{2}$$

- Linear-Algebra: Matrix multiplication

$$C_{ij} = \sum_{k=1}^{M} A_{ik} B_{kj} \tag{3}$$

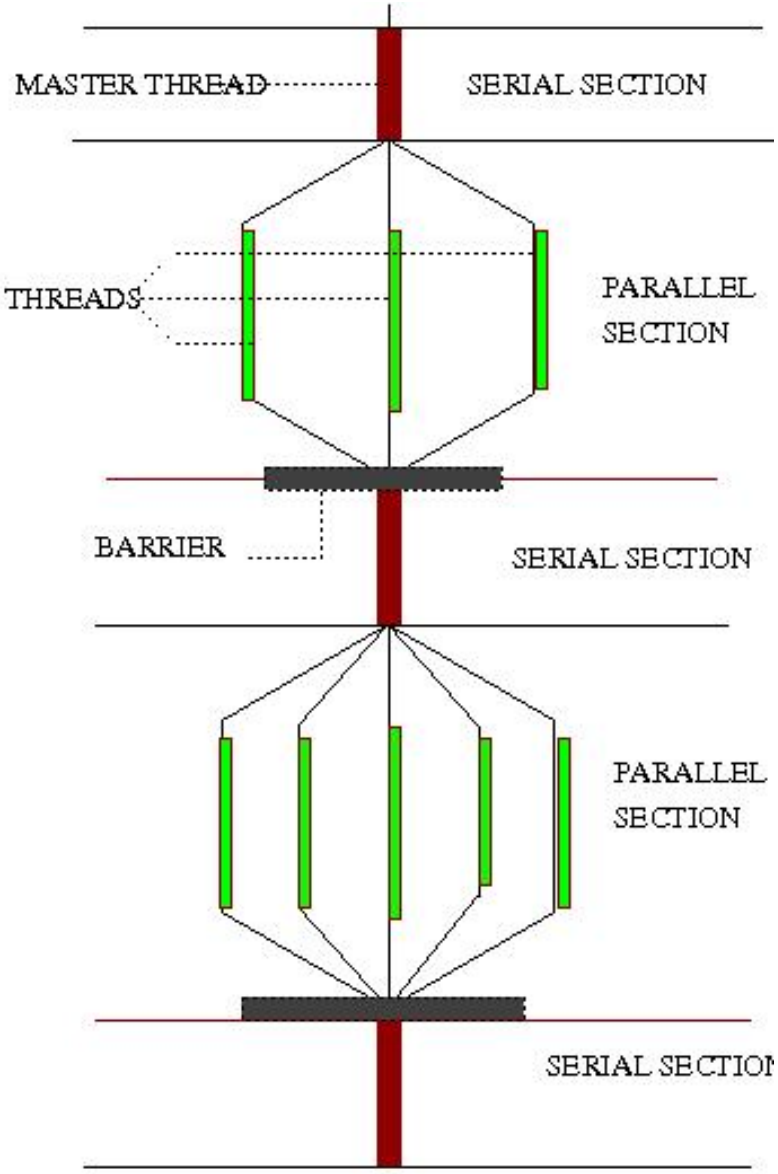
- Integration

$$y = 4 \int_0^1 \frac{dx}{1 + x^2} \tag{4}$$

- Dynamical simulations

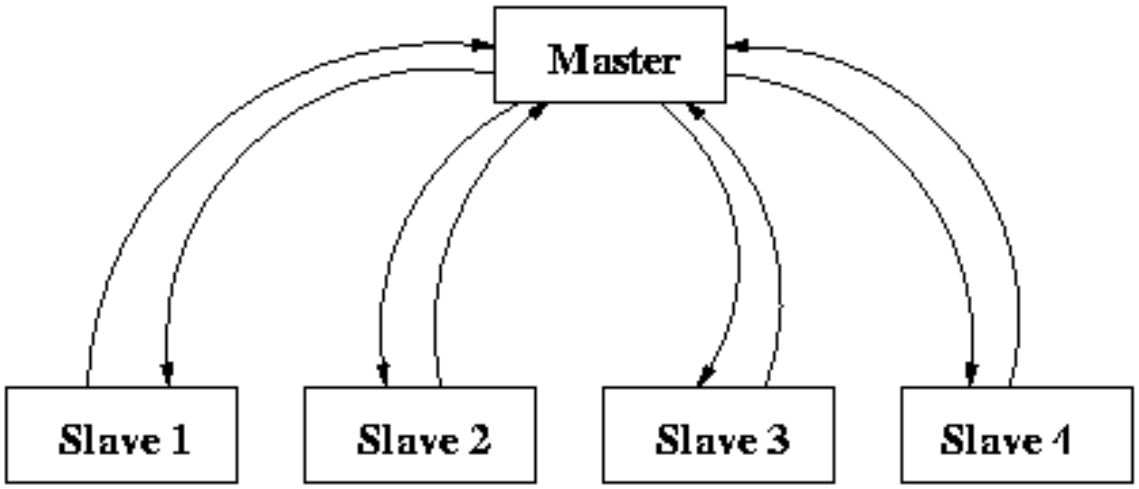
$$f_i = \sum_{j=1}^{N} \frac{m_j(\vec{x}_j - \vec{x}_i)}{(|\vec{x}_j - \vec{x}_i|)^{3/2}} \tag{5}$$

Models of parallel programming: Fork-Join



FORK–JOIN MODEL OF MULTI–THREADING

Models of parallel programming: Master-Slave

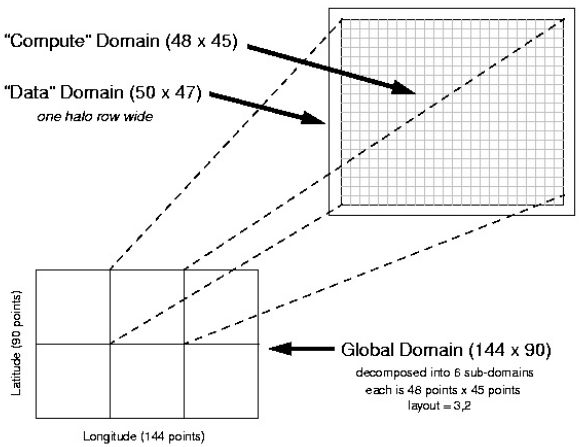


Synchronization

- More than one processors working on a single problem must coordinate (if the problem is not embarrassingly parallel problem).
- The **OpenMp** CRITICAL directive specifies a region of code that must be executed by only one thread at a time.
- The BARRIER (in **OpenMp, MPI** etc.,) directive synchronizes all threads in the team.
- In **pthreads** Mutexes are used to avoid data inconsistencies due to simultaneous operations by multiple threads upon the same memory area at the same time.

Partitioning or decomposing a problem: Divide and

- Partitioning or decomposing a problem is related to the opportunities for parallel execution.
- On the basis of whether we are diving the executions or data we can have two type of decompositions:
  - Functional decomposition
  - Domain decomposition



- Load balance

Domain decomposition

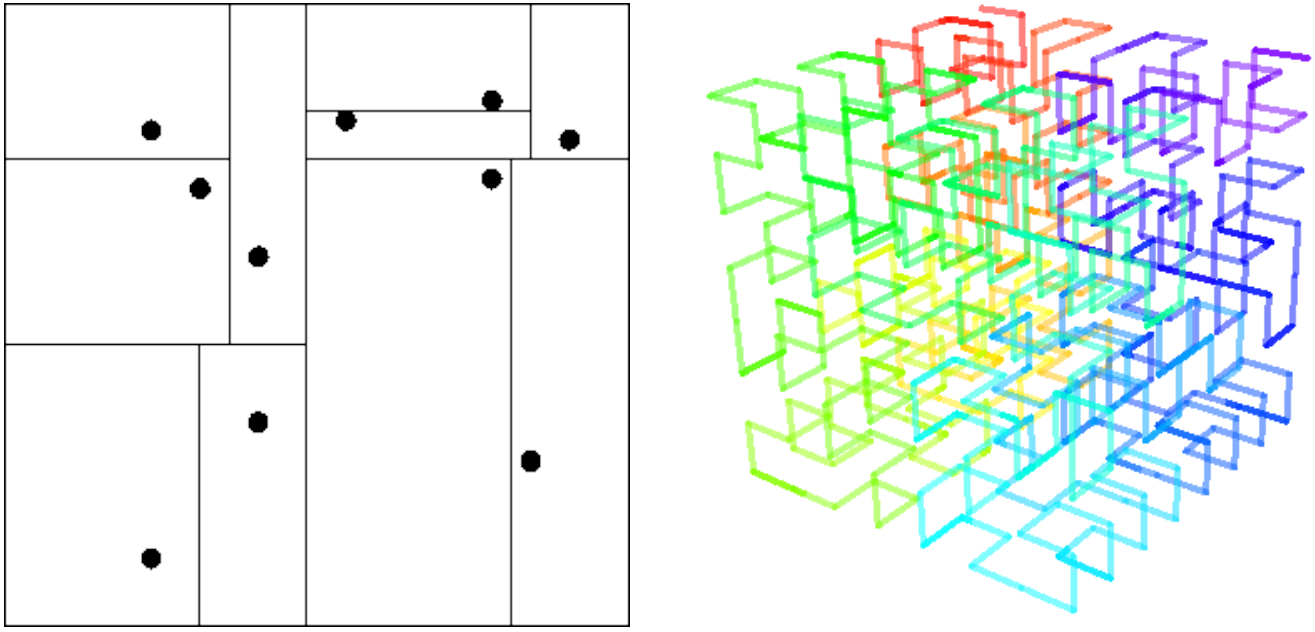


Figure: The left figure shows the use of orthogonal recursive bisection and the right one shows Peano-Hilbert space filling curve for domain decomposition. Note that parts which have Peano-Hilbert keys close to each other are physically also close to each other.

Examples: N-body, antenna-baselines, sky maps (l,m)

Mapping and indexing

Indexing in MPI

MPI_Comm_rank(MPI_COMM_WORLD, &id);

Indexing in OpenMP

id = omp_get_thread_num();

Indexing in CUDA

```
1   tid=threadIdx.x + blockDim.x*(threadIdx.y+blockDim.y * threadIdx.z);
2   bid=blockIdx.x  + gridDim.x * blockIdx.y;
3   nthreads = blockDim.x * blockDim.y * blockDim.z;
4   nblocks  = gridDim.x * gridDim.y;
5   id   = tid + nthreads * bid;
```

```
1   dim3 dimGrid(grszx,grszy), dimBlock(blsz,blsz,blsz);   \\
2   VecAdd $<<<dimGrid,dimBlock>>>$ ();
```

Mapping

- A problem of size N can be divided among N_P processors in the following ways:
  - A processor with identification number id get the data between the data index i_{start} and i_{end} where

$$i_{start} = id \times \frac{N}{N_p} \tag{6}$$

$$i_{end} = (id + 1) \times \frac{N}{N_p} \tag{7}$$

  - Every processor can pick data skipping N_P elements

$$for(i = 0; i < N; i+ = N_p) \tag{8}$$

- Note that N/N_p may not always an integer so keep load balance in mind.

Shared Memory programming : Threading Building Blocks

- Intel Threading building blocks (ITBB) is provided in the form of C++ runtime library which and can run on any platform.
- The main advantage of TBB is that it works at a higher level than raw threads, yet does not require exotic languages or compilers.
- Most threading packages require you to create, join, and manage threads. Programming directly in terms of threads can be tedious and can lead to inefficient programs because threads are low-level, heavy constructs that are close to the hardware.
- TBB runtime library automatically schedules tasks onto threads in a way that makes efficient use of processor resources. The runtime is very effective in load-balancing also.

Shared Memory programming : Pthreads

```
1   #include<pthread.h>
2     void *print_hello_world(void * param)  {
3           long tid =(long)param;
4           printf("Hello world from  %ld ! \n",tid);
5     }
6
7   int main(int argc, char *argv[]){
8           pthread_t  mythread[NTHREADS];
9           long   i;
10          for(i=0; i < NTHREADS; i++)
11              pthread_create(&mythread[i],NULL,&print_hello_world,(void *)i);
12          pthread_exit(NULL);
13          return(0);
14    }
```

Reference : David R. Butenhof 1997, *Programming with Posix threads*, Addison-Wesley

Shared Memory programming : OpenMp

- No need to make any change in the structure of the program.
- Need only three things to be done :
  - #include< *omp.h* >
  - #pragma omp parallel for shared () private ()
  - -fopenmp when compiling
- In general available on all GNU/Linux system by default

References :

Rohit Chandra et. al. 2001, *Parallel Programming in OpenMP*, Morgan Kaufmann
Barbara Chapman et. al. 2008, *Using OpenMP*, MIT Press

http://www.iucaa.ernet.in/∼jayanti/openmp.html

OpenMP: Example

```
1   #include<stdio.h>
2   #include<omp.h>
3   int  main (int argc, char*argv[]){
4      int nthreads, tid, numthrd;
5      // set the number of threads
6      omp_set_num_threads(atoi(argv[1]));
7
8   #pragma omp parallel private(tid)
9
10     {
11       tid = omp_get_thread_num(); // obtain the thread id
12
13       nthreads = omp_get_num_threads(); // find number of threads
14
15       printf("\tHello World from thread  %d of %d\n", tid,nthreads);
16
17     }
18     return(0);
19  }
```

Distributed Memory Programming : MPI

- MPI is a solution for very large problems
- Communication
  - Point to Point
  - Collective
- Communication overhead can dominate computation and it may be hard to get linear scaling.
- Is distributed in the form of libraries e.g., libmpi,libmpich or in the form of compilers e.g., mpicc,mpif90.
- Programs have to be completely restructured.

References :

Gropp, William et. al. 1999, *Using MPI 2*, MIT Press

http://www.iucaa.ernet.in/∼jayanti/mpi.html

MPI : Example

```
1   #include<stdio.h>
2   #include<mpi.h>
3   int main(int argc, char *argv[]){
4     int rank, size, len;
5     char name[MPI_MAX_PROCESSOR_NAME];
6     MPI_Init(&argc, &argv);
7     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
8     MPI_Comm_size(MPI_COMM_WORLD, &size);
9     MPI_Get_processor_name(name, &len);
10    printf ("Hello world! I'm %d of %d on %s\n", rank, size, name);
11    MPI_Finalize();
12    return(0);
13  }
```

GPU Programming : CUDA

- Programming language similar to C with few extra constructs.
- Parallel section is written in the form of kernel which is executed on GPU.
- Two different memory spaces : one for CPU and another of GPU. Data has to be explicitly copied back and forth.
- A very large number of threads can be used. Note that GPU cannot match the complexity of CPU. It is mostly used for SIMD programming.

References :

David B. Kirk and Wen-mei W. Hwu 2010, *Programming Massively Parallel Processors*, Morgan Kaufmann
Jason Sanders & Edward Kandrot 2011, *Cuda By examples*, Addison-Wesley

http://www.iucaa.ernet.in/∼jayanti/cuda.html

CUDA : Example

```
1   __global__ void force_pp(float *pos_d, float *acc_d,int n){
2
3     int tidx = threadIdx.x;
4     int tidy = threadIdx.y;
5     int tidz = threadIdx.z;
6
7     int myid =   (blockDim.z * (tidy+blockDim.y *tidx)) + tidz;
8     int nthreads = blockDim.z * blockDim.y * blockDim.x;
9
10    for(int i = myid; i < n; i+= nthreads){
11     for(int l=0; l < ndim; l++)
12       acc_d[l+ndim*i] = ...;
13
14    }//
15    __syncthreads();
16  }
17  // this was the device part
```

```
1   dim3 dimGrid(1);
2   dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE,BLOCK_SIZE);
3   cudaMemcpy(pos_d, pos,npart*ndim*sizeof(float),cudaMemcpyHostToDevice);
4   force_pp  <<<dimGrid,dimBlock>>>(pos_d,acc_d,npart);
5   cudaMemcpy(acc,acc_d,npart*ndim*sizeof(float),cudaMemcpyDeviceToHost );
```

Nvidia Quadro FX 3700

— General Information for device 0 —
Name: Quadro FX 3700
Compute capability: 1.1
Clock rate: 1242000
Device copy overlap: Enabled
Kernel execition timeout : Disabled
— Memory Information for device 0 —
Total global mem: 536150016
Total constant Mem: 65536
Max mem pitch: 262144
Texture Alignment: 256
— MP Information for device 0 —
Multiprocessor count: 14
Shared mem per mp: 16384
Registers per mp: 8192
Threads in warp: 32
Max threads per block: 512
Max thread dimensions: (512, 512, 64)
Max grid dimensions: (65535, 65535, 1)

Dynamic & static libraries

- Most of the open source software are distributed in the form of source codes and from which libraries are created for the use.
- In general libraries are not portable.
- One of the most common problems which a user face is due to not linking libraries.
- The most common way to create libraries is: source code $\longrightarrow$ object code $\longrightarrow$ library.

```
gcc -c first.c
gcc -c second.c
ar rc libtest.a first.o second.o
gcc -shared -Wl,-soname, libtest.so.0 -o libtest.so.0 first.o second.o -lc
```

- The above library can be used in the following way

```
gcc program.c -L/LIBPATH -ltest
```

- Dynamic library gets preference over static one.

Hyper threading

- Hyper-Threading Technology used in Intel® Xeon™ and Intel® Pentium™ 4 processors, makes a single physical processor appear as two logical processors to the operating system.
- Hyper-Threading duplicates the architectural state on each processor, while sharing one set of execution resources.
- sharing system resources, such as cache or memory bus, may degrade system performance and Hyper-Threading can improve the performance of some applications, but not all.

Summary

- It is not easy to make a super-fast single processor so multi-processor computing is the only way to get more computing power.
- When more than one processors (cores) share the same memory shared memory programming is uded e.g., pthread,OpenMp, itbb etc.
- Shared memory programming is fast and it easy to get linear scaling since communication is not an issue.
- When processors having their own memory are used for parallel computation, distributed memory programming is used e.g., MPI, PVM.
- Distributed memory programming is the main way to solve large problems (when thousands of processors are needed).
- General Purpose Graphical Processing Units (GPGPU) can provide very high performance at very low cost, however, programming is somewhat complicated and parallelism is limited to only SIMD.

Top 10

Rank	Site	Computer
1	National Supercomputing Center in Tianjin China	Tianhe-1A - NUDT TH MPP, X5670 2.93Ghz 6C, NVIDIA GPU, FT-1000 8C NUDT
2	DOE/SC/Oak Ridge National Laboratory United States	Jaguar - Cray XT5-HE Opteron 6-core 2.6 GHz Cray Inc.
3	National Supercomputing Centre in Shenzhen (NSCS) China	Nebulae - Dawning TC3600 Blade, Intel X5650, NVidia Tesla C2050 GPU Dawning
4	GSIC Center, Tokyo Institute of Technology Japan	TSUBAME 2.0 - HP ProLiant SL390s G7 Xeon 6C X5670, Nvidia GPU, Linux/Windows NEC/HP
5	DOE/SC/LBNL/NERSC United States	Hopper - Cray XE6 12-core 2.1 GHz Cray Inc.
6	Commissariat a l'Energie Atomique (CEA) France	Tera-100 - Bull bullx super-node S6010/S6030 Bull SA
7	DOE/NNSA/LANL United States	Roadrunner - BladeCenter QS22/LS21 Cluster, PowerXCell 8i 3.2 Ghz / Opteron DC 1.8 GHz, Voltaire Infiniband IBM
8	National Institute for Computational Sciences/University of Tennessee United States	Kraken XT5 - Cray XT5-HE Opteron 6-core 2.6 GHz Cray Inc.
9	Forschungszentrum Juelich (FZJ) Germany	JUGENE - Blue Gene/P Solution IBM
10	DOE/NNSA/LANL/SNL United States	Cielo - Cray XE6 8-core 2.4 GHz Cray Inc.

Thank You !