Programming using the Message-Passing Paradigm

Mallegowda M

Distributed memory systems

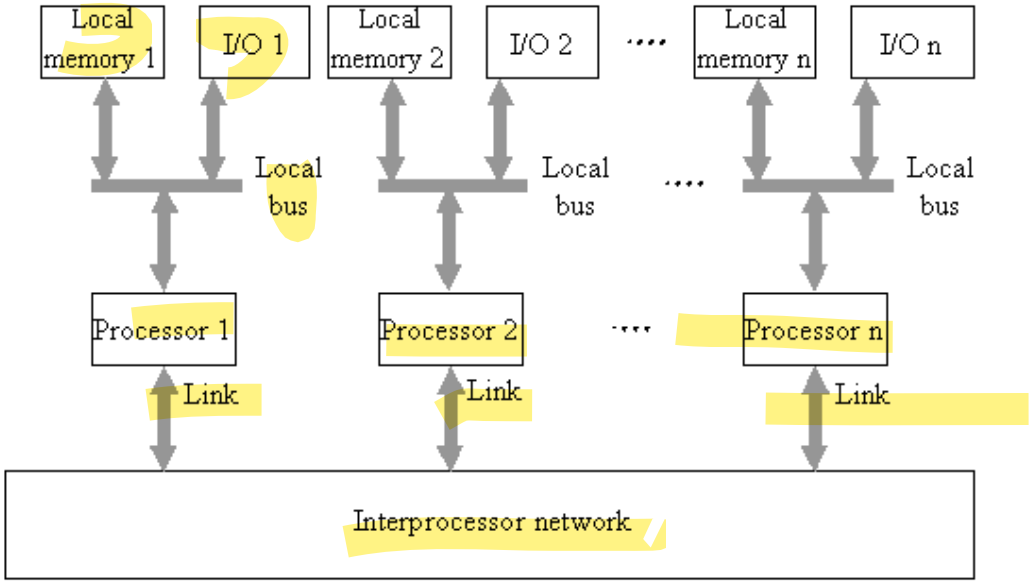


Fig: A multiprocessor system with a distributed memory (loosely coupled system)[1]

Distributed memory

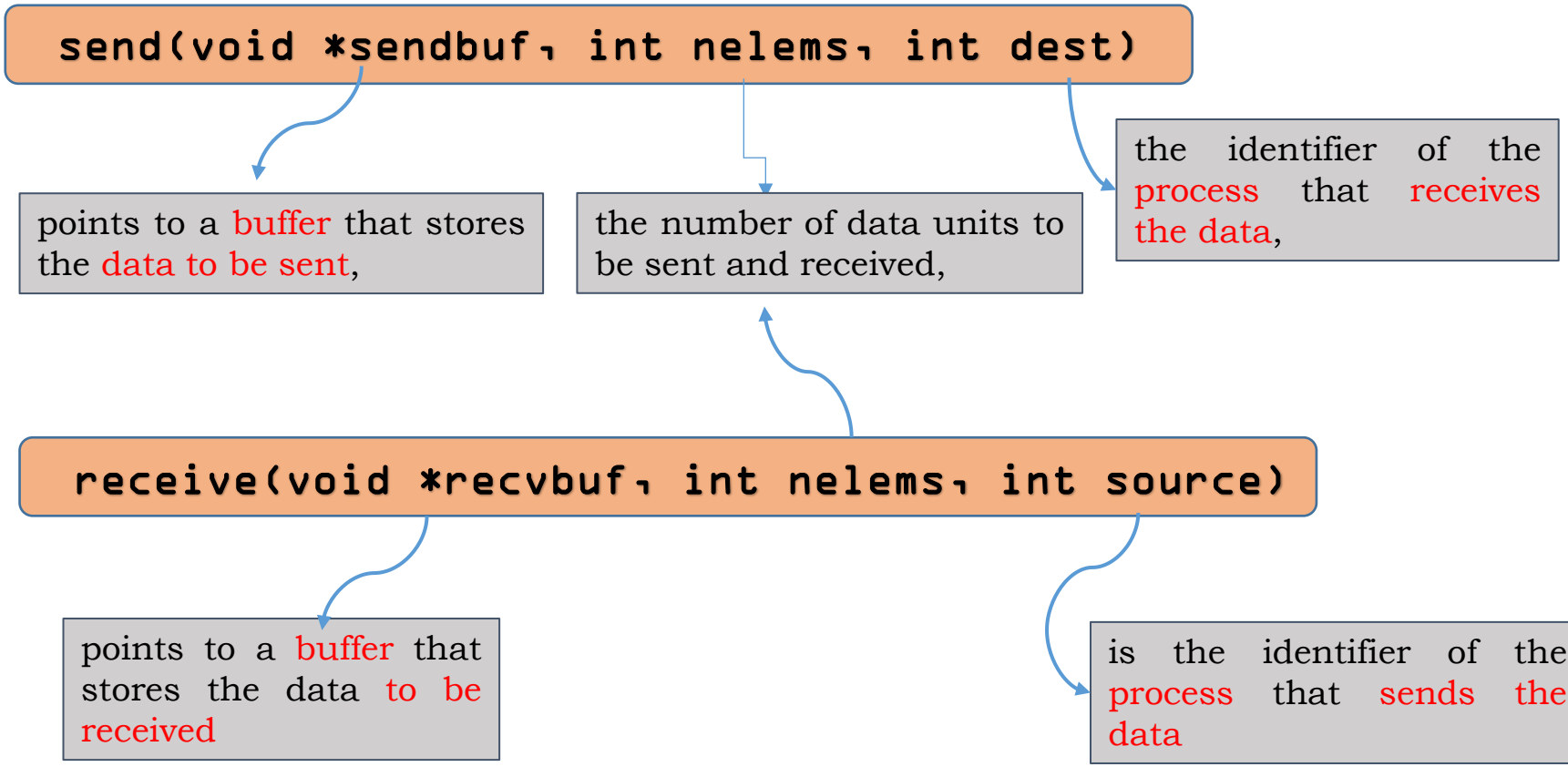
- Each processor has its own private memory.
- Computational tasks can only operate on local data,
- if remote data is required, the computational task must communicate with one or more remote processors. Communication through the **message passing**.

[1]https://edux.pjwstk.edu.pl/mat/264/lec/index119.html

➢ Principles of Message-Passing Programming

- Message-passing paradigm consists of p processes, each with its own exclusive address space
- All interactions (read-only or read/write) require cooperation of two processes.
- The programmer is fully aware of all the costs of non-local interactions by Two-way interactions.
- Message-passing programs are often written using the asynchronous or loosely synchronous paradigms.
  - In the asynchronous paradigm, all concurrent tasks execute asynchronously.
  - loosely synchronous :
    - Tasks or subsets of tasks synchronize to perform interactions.
    - Between these interactions, tasks execute completely asynchronously.

Send and Receive Operations



Send and Receive Operations

```
P0                          P1

a = 100;                    receive(&a, 1, 0)
send(&a, 1, 1);             printf("%d\n", a);
a = 0;
```

Blocking Message Passing Operations

- **Blocking Non-Buffered Send/Receive**
- **Blocking Buffered Send/Receive**

Non-Blocking Message Passing Operations

Blocking Non-Buffered Send/Receive

- Send operation does not return until(Block) the matching receive has been encountered at the receiving process.
- Process involves a handshake between the sending and receiving processes
  - The sending process sends a request to communicate to the receiving process.
  - When the receiving process encounters the target receive, it responds to the request.
  - The sending process upon receiving this response initiates a transfer operation

Blocking Send and Receive Protocols

Possible protocols for send and receive operations.

	Blocking Operations	Non−Blocking Operations
Buffered	Sending process returns after data has been copied into communication buffer	Sending process returns after initiating DMA transfer to buffer. This operation may not be completed on return
Non Buffered	Sending process blocks until matching receive operation has been encountered	

Outline

- Principles of Message-Passing Programming
- The Building Blocks:
  - Send and Receive Operations,
- MPI: the Message Passing Interface
- Topologies and Embedding,
- Overlapping Communication with Computation
- Collective Communication and Computation Operations

MPI: the Message Passing Interface

#include <mpi.h>

- MPI defines a standard library for message-passing.
  - Using either C or Fortran
- The MPI standard defines
  - Syntax
  - Semantics of a core set of library routines
- The minimal set of MPI routines can be used write to fully-functional message-passing programs
- MPI Features
  - Communicator information (com. domain).
  - Point to point communication.
  - Collective communication, Topology support, Error handling.

Programming using the Message-Passing Paradigm > MPI

The minimal set of MPI routines

Discuss primitive MPI routines used to provide communication among processes.

MPI_Init	Initializes MPI.
MPI_Finalize	Terminates MPI.
MPI_Comm_size	Determines the number of processes.
MPI_Comm_rank	Determines the label of the calling process.
MPI_Send	Sends a message.
MPI_Send	Receives a message.

MPI Functions: Initialization

- MPI_Init initializes the MPI environment
  - Must be called once by all processes. MPI_SUCCESS (if successful).
- MPI_Finalize performs clean-up tasks, no MPI calls after that (not even MPI_Init)
- MPI_Init, MPI_Finalize must be called by all processes.

.

```
int MPI_Init(int *argc, char ***argv) int
```

```
MPI_Finalize()
```

MPI Functions: Communicator

- Concept of communication domain-
  - Set of processes allowed to communicate with each other.
  - Processes may belong to **different communicators**
- **MPI_COMM_WORLD** default for all processes involved.
  - Rank is an int[0..**comm_size-1**]
- Processes calling **MPI_Comm_size, MPI_Comm_rank** functions must belong to the appropriate communicator otherwise error

```
int MPI_Comm_size(MPI_Comm comm, int *size)
```

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

Programming using the Message-Passing Paradigm > MPI

Hello World!

```
#include  <mpi.h>
int main(int argc, char *argv[])
{
int npes, myrank;
 MPI_Init(&argc, &argv); MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank); printf("From process %d out of %d, Hello
world!\n", myrank, npes); MPI_Finalize();
return 0;
}
```

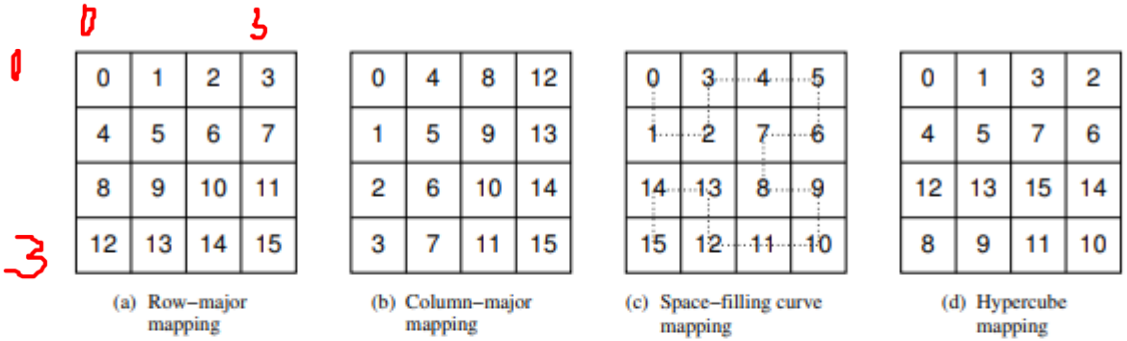
Topologies and Embedding

- MPI views the processes as being arranged in a one-dimensional topology and uses a linear ordering to number the processes.
- Both the computation and the set of interacting processes are naturally identified by their coordinates in that topology.
- MPI does not provide the programmer any control over these mappings.

- An MPI process with rank *rank* corresponds to process (*row*, *col*) in the grid such that
  - ***row = rank/4* and *col = rank%4***
- As an illustration, the process with **rank 7** is mapped to process (1, 3) in the grid.

RAMAIAH Institute of Technology

Programming using the Message-Passing Paradigm > MPI



0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

(a) Row–major mapping

0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

(b) Column–major mapping

0	3	4	5
1	2	7	6
14	13	8	9
15	12	11	10

(c) Space–filling curve mapping

0	1	3	2
4	5	7	6
12	13	15	14
8	9	11	10

(d) Hypercube mapping

Different ways to map a set of processes to a two-dimensional grid. (a) and (b) show a row- and column-wise mapping of these processes, (c) shows a mapping that follows a space-filling curve (dotted line), and (d) shows a mapping in which neighboring processes are directly connected in a hypercube.

- MPI topologies are virtual – no relation to the physical structure of the computer
- Data mapping “more natural” only to the programmer.
- Usually no performance benefits – But code becomes more readable

Topologies and Embedding

- MPI provides a set of routines that allows the programmer to arrange the processes in different topologies.
  - Both the computation and the set of interacting processes are naturally identified by their coordinates in that topology.
  - It is up to the MPI library to find the most appropriate mapping that reduces the cost of sending and receiving messages.
- .Each node in the graph corresponds to a process and two nodes are connected if they communicate with each other. Graphs of processes can be used to specify any desired topology.
- Most commonly used topologies in message-passing programs are one-, two-, or higher-dimensional grid and its referred as "**Cartesian topologies**".

Topologies and Embedding

- MPI provides a set of routines that allows the programmer to arrange the processes in different topologies.
  - Both the computation and the set of interacting processes are naturally identified by their coordinates in that topology.
  - It is up to the MPI library to find the most appropriate mapping that reduces the cost of sending and receiving messages.
- .Each node in the graph corresponds to a process and two nodes are connected if they communicate with each other. Graphs of processes can be used to specify any desired topology.
- Most commonly used topologies in message-passing programs are one-, two-, or higher-dimensional grid and its referred as "**Cartesian topologies**".

Topologies and Embedding

- Creating a topology produces a new communicator.
- Topology Types
  - Cartesian Topologies –
    - Connected to its neighbor in a virtual grid.
  - Graph Topologies -general graphs,
- Creating a Cartesian Virtual Topology
  - New communicator with processes ordered in a Cartesian grid

Cartesian topologies

- MPI's function for describing Cartesian topologies is called MPI_Cart_create.
- A group of processes that belong to the communicator comm_old and creates a virtual process topology.

The shape and properties of the topology are specified by the arguments ndims, dims, and periods.

.

```
int MPI_Cart_create (MPI_Comm comm_old, int ndims, int
*dims, int *periods, int reorder, MPI_Comm *comm_cart)
```

Topology information is attached to a new communicator

Cartesian topologies

- MPI's function for describing Cartesian topologies is called MPI_Cart_create.
- A group of processes that belong to the communicator comm_old and creates a virtual process topology.

It Takes the coordinates of the process as argument in the coords array and returns its rank in rank.

```
int MPI_Cart_rank(MPI_Comm comm_cart, int *coords, int *rank)

int MPI_Cart_coord(MPI_Comm comm_cart, int rank, int maxdims, int *coords)
```

The MPI_Cart_coords takes the rank of the process rank and returns its Cartesian coordinates in the array coords

1. Process coordinates in a Cartesian structure begin their numbering at 0. Row-major numbering is always used for the processes in a cartesian structure.

This means that, for example, the relation between group rank and coordinates for four processes in a (2 × 2) grid is as follows.

coord (0,0):	rank 0
coord (0,1):	rank 1
coord (1,0):	rank 2
coord (1,1):	rank 3

Overlapping Communication with Computation

- In order to overlap communication with computation, MPI provides a pair of functions for performing non-blocking send and receive operations.

```
int MPI_Isend(void *buf, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm,
              MPI_Request *request)
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
              int source, int tag, MPI_Comm comm,
              MPI_Request *request)
```

- These operations return before the operations have been completed. Function `MPI_Test` tests **whether or not the non-blocking send or receive operation identified by its request has finished**.

```
int MPI_Test(MPI_Request *request, int *flag,
    MPI_Status *status)
```

- `MPI_Wait` waits for the operation to complete.

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

Overlapping Communication with Computation

- In order to overlap communication with computation, MPI provides a pair of functions for performing non-blocking send and receive operations.

```
int MPI_Isend(void *buf, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm,
              MPI_Request *request)
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
              int source, int tag, MPI_Comm comm,
              MPI_Request *request)
```

- These operations return before the operations have been completed. Function `MPI_Test` tests whether or not the non-blocking send or receive operation identified by its request has finished.

```
int MPI_Test(MPI_Request *request, int *flag,
    MPI_Status *status)
```

- `MPI_Wait` waits for the operation to complete.

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

Collective Communication and Computation Operations

- MPI provides an extensive set of functions for performing common collective communication operations.
- Each of these operations is defined over a group corresponding to the communicator.
- All processors in a communicator must call these operations.

Collective Communication Operations

- The barrier synchronization operation is performed in MPI using:

```
int MPI_Barrier(MPI_Comm comm)
```

  The one-to-all broadcast operation is:

```
int MPI_Bcast(void *buf, int count, MPI_Datatype
datatype,                int source, MPI_Comm comm)
```

- The all-to-one reduction operation is:

```
int MPI_Reduce(void *sendbuf, void *recvbuf, int count,
          MPI_Datatype datatype, MPI_Op op, int target,
          MPI_Comm comm)
```

Predefined Reduction Operations

Operation	Meaning	Datatypes
MPI_MAX	Maximum	C integers and floating point
MPI_MIN	Minimum	C integers and floating point
MPI_SUM	Sum	C integers and floating point
MPI_PROD	Product	C integers and floating point
MPI_LAND	Logical AND	C integers
MPI_BAND	Bit-wise AND	C integers and byte
MPI_LOR	Logical OR	C integers
MPI_BOR	Bit-wise OR	C integers and byte
MPI_LXOR	Logical XOR	C integers
MPI_BXOR	Bit-wise XOR	C integers and byte
MPI_MAXLOC	max-min value-location	Data-pairs
MPI_MINLOC	min-min value-location	Data-pairs

Collective Communication Operations

- The gather operation is performed in MPI using:

```
int MPI_Gather(void *sendbuf, int sendcount,
          MPI_Datatype senddatatype, void *recvbuf,
          int recvcount, MPI_Datatype recvdatatype,
          int target, MPI_Comm comm)
```

- MPI also provides the MPI_Allgather function in which the data are gathered at all the processes.

```
int MPI_Allgather(void *sendbuf, int sendcount,
          MPI_Datatype senddatatype, void *recvbuf,
          int recvcount, MPI_Datatype recvdatatype,
          MPI_Comm comm)
```

- The corresponding scatter operation is:

```
int MPI_Scatter(void *sendbuf, int sendcount,
          MPI_Datatype senddatatype, void *recvbuf,
          int recvcount, MPI_Datatype recvdatatype,
          int source, MPI_Comm comm)
```

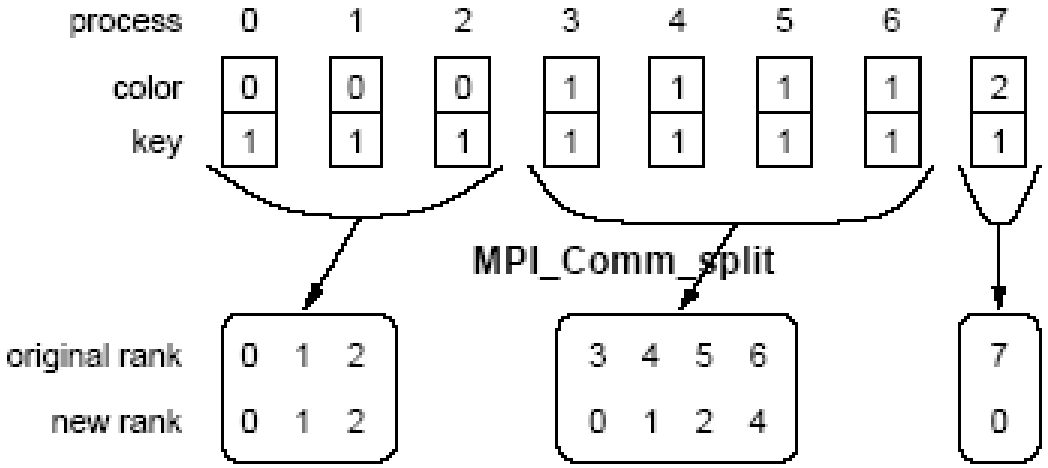
Groups and Communicators

- In many parallel algorithms, communication operations need to be restricted to certain subsets of processes.
- MPI provides mechanisms for partitioning the group of processes that belong to a communicator into subgroups each corresponding to a different communicator.
- The simplest such mechanism is:

```
int MPI_Comm_split(MPI_Comm comm, int color, int key,
                   MPI_Comm *newcomm)
```

- This operation groups processors by color and sorts resulting groups on the key.

Groups and Communicators



Using `MPI_Comm_split` to split a group of processes in a communicator into subgroups.

Groups and Communicators

- In many parallel algorithms, processes are arranged in a virtual grid, and in different steps of the algorithm, communication needs to be restricted to a different subset of the grid.
- MPI provides a convenient way to partition a Cartesian topology to form lower-dimensional grids:

```
int MPI_Cart_sub(MPI_Comm comm_cart, int *keep_dims,
          MPI_Comm *comm_subcart)
```

- If `keep_dims[i]` is true (non-zero value in C) then the `i`th dimension is retained in the new sub-topology.
- The coordinate of a process in a sub-topology created by `MPI_Cart_sub` can be obtained from its coordinate in the original topology by disregarding the coordinates that correspond to the dimensions that were not retained.

Introduction: GPUs as Parallel Computers

Mallegowda M

Introduction to GPU

GPUs as Parallel Computers,

- Microprocessors based on a single central processing unit (CPU)
  - Giga (billion)floating-point operations per second (GFLOPS)
- Most software developers have relied on the advances in hardware to increase the speed of their applications.
- Due to energy consumption and heat-dissipation issues that have limited the increase of the clock frequency and the level of productive activities.
- **Processor cores.**
  - Vast majority of software applications are written as sequential programs, as described by von Neumann.
  - A sequential program will only run on one of the processor cores, which will not become significantly faster than those in use today.
  - concurrency revolution
    - multiple threads of execution cooperate to complete the work faster

Introduction to GPU

GPUs as Parallel Computers,

- Microprocessors based on a single central processing unit (CPU)
  - Giga (billion)floating-point operations per second (GFLOPS)
- Most software developers have relied on the advances in hardware to increase the speed of their applications.
- Due to energy consumption and heat-dissipation issues that have limited the increase of the clock frequency and the level of productive activities.
- **Processor cores.**
  - Vast majority of software applications are written as sequential programs, as described by von Neumann.
  - A sequential program will only run on one of the processor cores, which will not become significantly faster than those in use today.
  - concurrency revolution
    - multiple threads of execution cooperate to complete the work faster

Introduction to GPU

GPUs as Parallel Computers,

- few elite applications can justify the use of these expensive computers, thus limiting the practice of parallel programming to a small number of application developers.

- Many-core trajectory focuses more on the execution throughput of parallel applications.
- The many-cores began as a large number of much smaller cores, and, once again, the number of cores doubles with each generation.
  - Example : NVIDIA GeForce GTX 280 graphics processing unit (GPU) with 240 cores.
  - Each of which is a heavily multithreaded, in-order, single-instruction issue processor that shares its control and instruction cache with seven other cores

Introduction to GPU

GPUs as Parallel Computers,

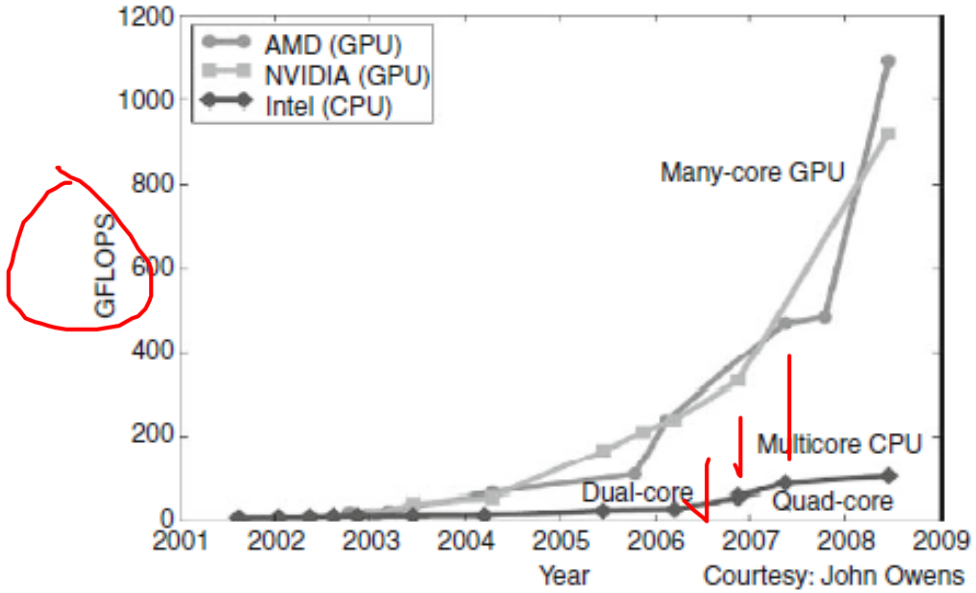


FIGURE 1.1

Enlarging performance gap between GPUs and CPUs.

GPUs as Parallel Computers,

Why there is such a large performance gap between many-core GPUs and general-purpose multicore CPUs?

- Differences in the fundamental design philosophies between the two types of processors
- The design of a CPU is optimized for sequential code performance.
- Control logic to allow instructions from a single thread of execution to execute in parallel while maintaining the appearance of sequential execution
- Control logic nor cache memories contribute to the peak calculation speed.
- Deliver strong sequential code performance on multicore.
- Memory bandwidth is another important issue.

GPUs as Parallel Computers,

- Graphics chips have been operating at approximately 10 times the bandwidth of contemporaneously available CPU chips

GeForce 8800 GTX
- 85 GB/s : in and out

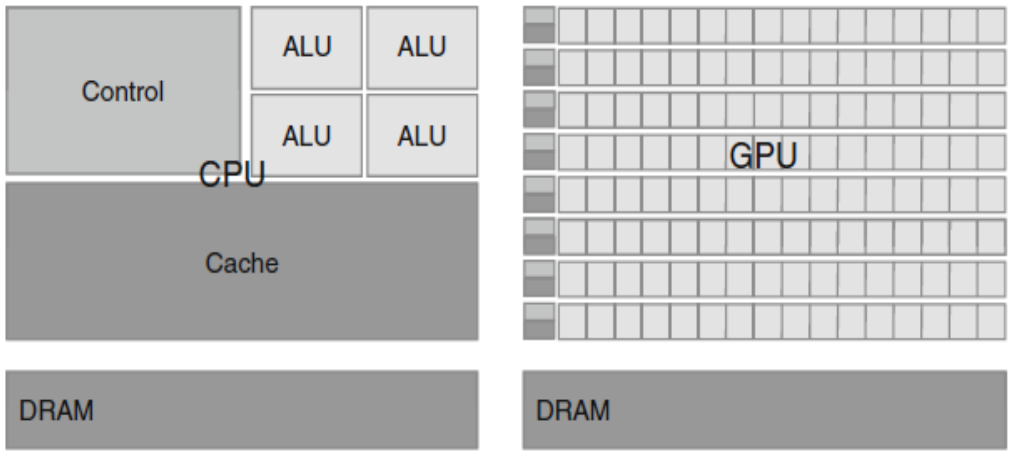


FIGURE 1.2

CPUs and GPUs have fundamentally different design philosophies.

GPUs AS PARALLEL COMPUTERS

- GPUs is shaped by the fast growing video game industry
- ability to perform a massive number of floating-point calculations per video frame
- GPUs are designed as numeric computing engines, and they will not perform well on some tasks on which CPUs are designed to perform well;
- One should expect that most applications will use both CPUs and GPUs, executing the sequential parts on the CPU and numerically intensive parts on the GPUs.
- CUDA (Compute Unified Device Architecture) programming model, introduced by NVIDIA in 2007, is designed to support joint CPU/GPU execution of an application

Explain the architecture of a modern Graphics Processing Unit (GPU) and how it differs from a traditional Central Processing Unit (CPU) in terms of parallel processing capabilities.

ARCHITECTURE OF A MODERN GPU

- CUDA-capable GPU
- It is organized into an array of highly threaded streaming multiprocessors. Two SMs form a building block;
- Each SM in has a number of streaming processors (SPs) that share control logic and instruction cache.
- (GDDR) DRAM- Memory.
- The massively parallel G80 chip has 128 Sps- (16 SMs, each with 8 SPs)
- Each SP has a multiply–add (MAD) unit and an additional multiply unit.
- Intel CPUs support 2 or 4 threads, depending on the machine model, per core. But The G80 chip supports up to 768 threads
- per SM, which sums up to about 12,000 threads for this chip.

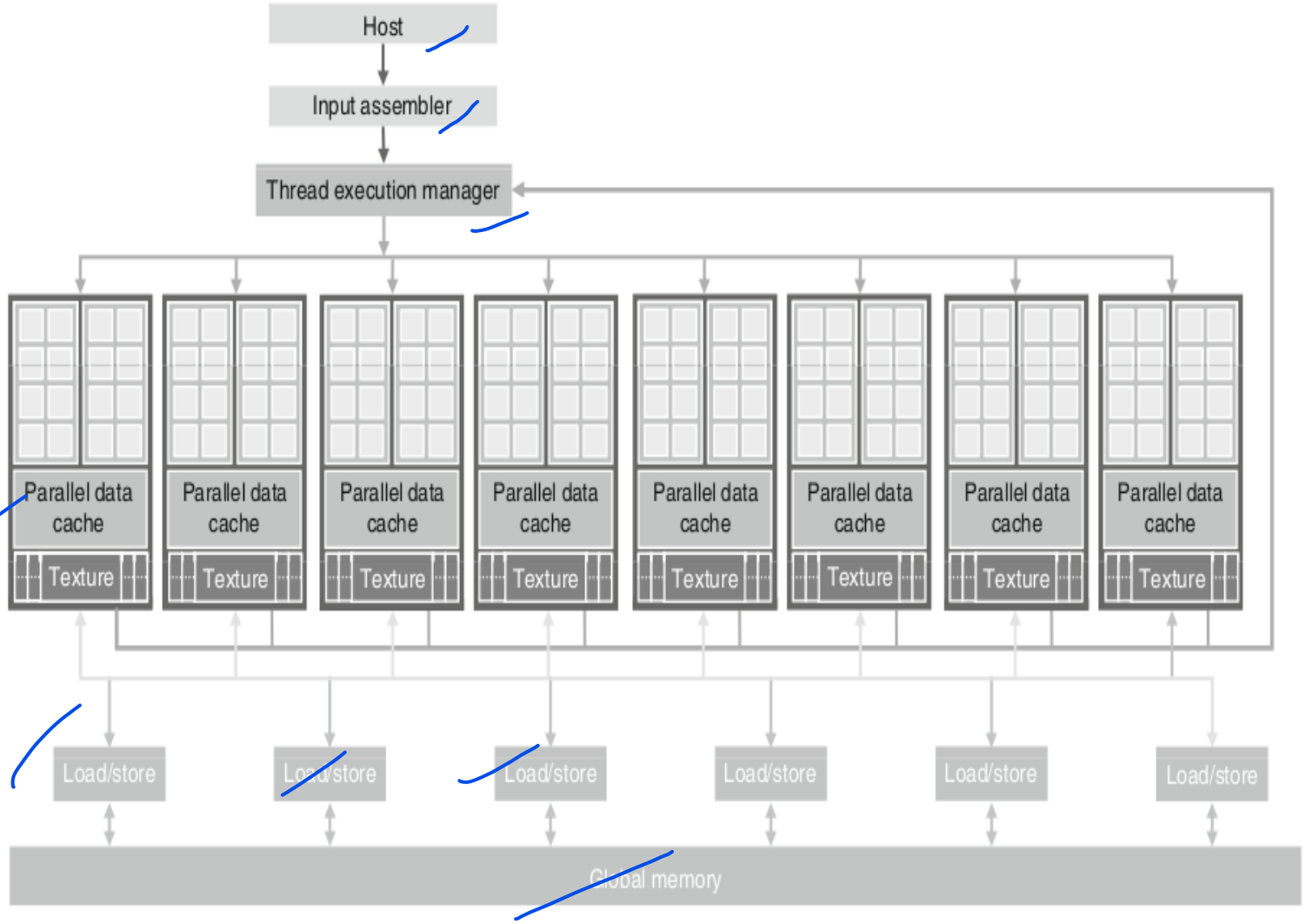


FIGURE 1.3

Architecture of a CUDA-capable GPU.

- WHY MORE SPEED OR PARALLELISM?
  - supercomputing applications, or superapplications.
  - molecular-level observation: computational model to simulate the underlying molecular activities with boundary conditions set.
  - video and audio coding and manipulation.
    - high-definition television (HDTV)
  - 3D imaging and visualization.
  - consumer electronic gaming
    - imagine driving a car in a game today; the game is, in fact, simply a prear-ranged set of scenes.

- **PARALLEL PROGRAMMING LANGUAGES AND MODELS**
  - The ones that are the most widely used are the Message Passing Interface (MPI) for scalable cluster computing and OpenMPI for shared-memory multiprocessor systems.
  - MPI has been successful in the high-performance scientific computing domain. Applications written in MPI have been known to run successfully on cluster computing systems
  - CUDA, on the other hand, provides shared memory for parallel execution in the GPU to address this difficulty.
  - CUDA achieves much higher scalability with simple, low-overhead thread management and no cache coherence hardware requirements.
  - OpenCL- Apple, Intel, AMD/ATI, and NVIDIA, have jointly developed a standardized program-ming mode
  - OpenCL programming model defines language extensions and runtime APIs to allow programmers to manage parallelism and data delivery in massively parallel proce_{ssors.}

- **WHY MORE SPEED OR PARALLELISM?**
  - supercomputing applications, or superapplications.
  - molecular-level observation: computational model to simulate the underlying molecular activities with boundary conditions set.
  - video and audio coding and manipulation.
    - high-definition television (HDTV)
  - 3D imaging and visualization.
  - consumer electronic gaming
    - imagine driving a car in a game today; the game is, in fact, simply a prear-ranged set of scenes.

History of GPU Computing

- The Era of Fixed-Function Graphics Pipelines

- The remarkable advancement of graphics hardware performance has
- been driven by the market demand for high-quality, real-time graphics in
- computer applications.

- Graphics application programming interface (API) libraries became popular

- An API is a standardized layer of software (i.e., a collection of library functions) that allows applications (such as games) to use software or hardware services and functionality.

- DirectX, Microsoft's proprietary API for media functionality

- Major API is OpenGL

Three dimensional (3D) graphics pipeline hardware evolved from large expensive systems of the early 1980s to small workstations and then PC accelerators in the mid to late 1990s.

During this period, the performance increased:

from 50 millions pixels to 1 billion pixels per second,

from 100,000 vertices to 10 million vertices per second.

This advancement was driven by market demand for high quality, real time graphics in computer applications.

The architecture evolved

From a simple pipeline for drawing wire frame diagrams.

To a parallel design of several deep parallel pipelines capable of rendering the complex interactive imagery of 3D scenes.

In the mean time, graphics processors became programmable

Explain the stages of the NVIDIA GeForce graphics pipeline. How do these stages contribute to rendering an image from a 3D model?

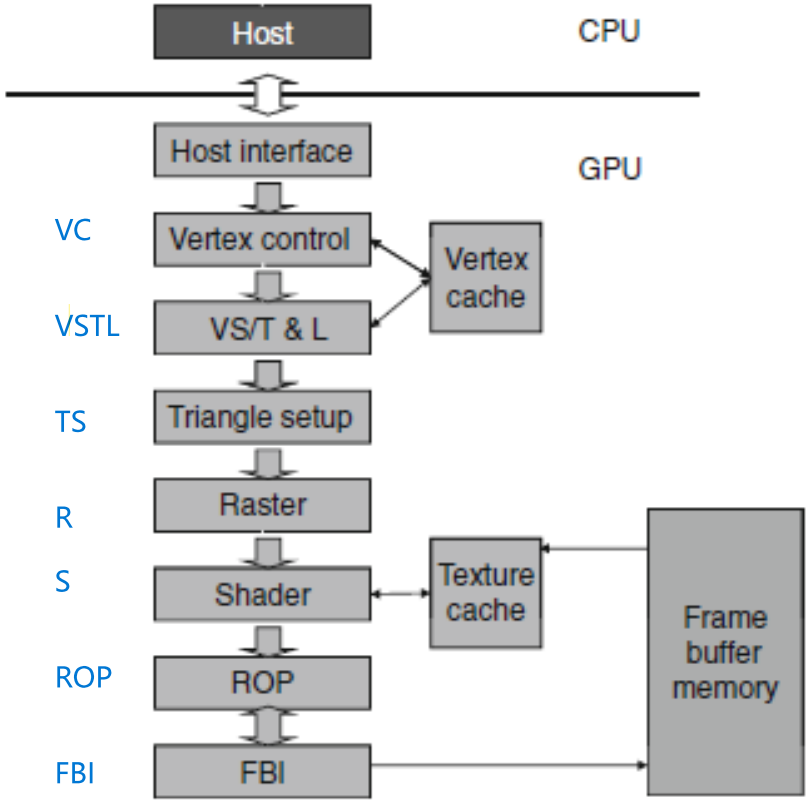


FIGURE 2.1

A fixed-function NVIDIA GeForce graphics pipeline.

Stages in the first part of the pipeline1 vertex

vertex control

This stage receives parametrized triangle data from the CPU.

The data gets converted and placed into the vertex cache.

VS/T & L (vertex shading, transform, and lighting)

The VS/T & L stage transforms vertices and assigns per-vertex values, e.g.: colors, normals, texture coordinates, tangents.

The vertex shader can assign a color to each vertex, but color is not applied to triangle pixels until later.

Triangle setup

Edge equations are used to interpolate colors and otherper-vertex data across the pixels touched by the triangle.

Raster

The raster determines which pixels are contained in each triangle.Per-vertex values necessary for shading are interpolated.

shader

The shader determines the final color of each pixel as a combined effect of interpolation of vertex colors, texture mapping, per-pixel lighting, reflections, etc.

ROP (Raster Operation)

The final raster operations blend the color of overlapping/adjacent objects for transparency and antialiasing effects.

For a given viewpoint, visible objects are determined and occluded pixels (blocked from view by other objects) are discarded.

FBI (Frame Buffer Interface)

The FBI stages manages memory reads from and writes to the display frame buffer memory

This data independence as the dominating characteristic is the key difference between the design assumption for GPUs and CPUs.
A single frame, rendered in 1/60th of a second, might have a million triangles and 6 million pixels.

Unified graphics and computing processors

Introduced in 2006, the GeForce 8800 GPU mapped the separate programmable graphics stages to an array of unified processors

High-clock-speed design made programmable GPU processor array ready for general numeric computing

Original GPGPU programming used APIs (DirectX or OpenGL): to a GPU everything is a pixel

Interpolation of vertex colors, texture mapping, per-pixel lighting mathematics, reflections.
Many effects that make the rendered images more realistic are incorporated in the shader stage.
Raster operation (ROP) stage in Figure 2.2 performs the final rasteroperations on the pixels

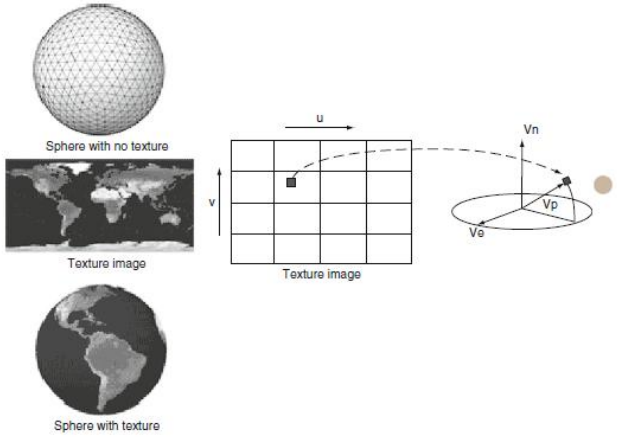


FIGURE 2.2
Texture mapping example: painting a world map texture image onto a globe object.

GPU computing

Drawbacks of the GPGPU model:

The programmer must know APIs and GPU architecture well
Programs expressed in terms of vertex coordinates, textures,shader programs, add to the complexity
Random reads and writes to memory are not supported
No double precision is limiting for scientific applications

Evolution of Programmable Real-Time Graphics

These programmable pixel shader processors were part of a general trend towards unifying the functionality of the different stages as seen by the Application programmer

The GeForce 6800 and 7800 series were built with separate processor designs dedicated to vertex and pixel processing.

The XBox 360 introduced an early unified processor GPU in 2005, allowing vertex and pixel shaders to execute on the same processor Two particular programmable stages stand out: the vertex shader and the pixel shader.

Vertex shader programs map the positions of triangle vertices onto the screen, altering their position, color, or orientation