Parallel Computing Platforms

Chieh-Sen (Jason) Huang

Department of Applied Mathematics

National Sun Yat-sen University

Thank Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar for providing slides.

Topic Overview

- Dichotomy of Parallel Computing Platforms
- Communication Costs in Parallel Machines
- The first example of MPI: the Message Passing Interface

Explicitly Parallel Platforms

Dichotomy of Parallel Computing Platforms

- An explicitly parallel program must specify concurrency and interaction between concurrent subtasks.
- The former is sometimes also referred to as the control structure and the latter as the communication model.

Control Structure of Parallel Programs

- Parallelism can be expressed at various levels of granularity – from instruction level to processes.
- Between these extremes exist a range of models, along with corresponding architectural support.

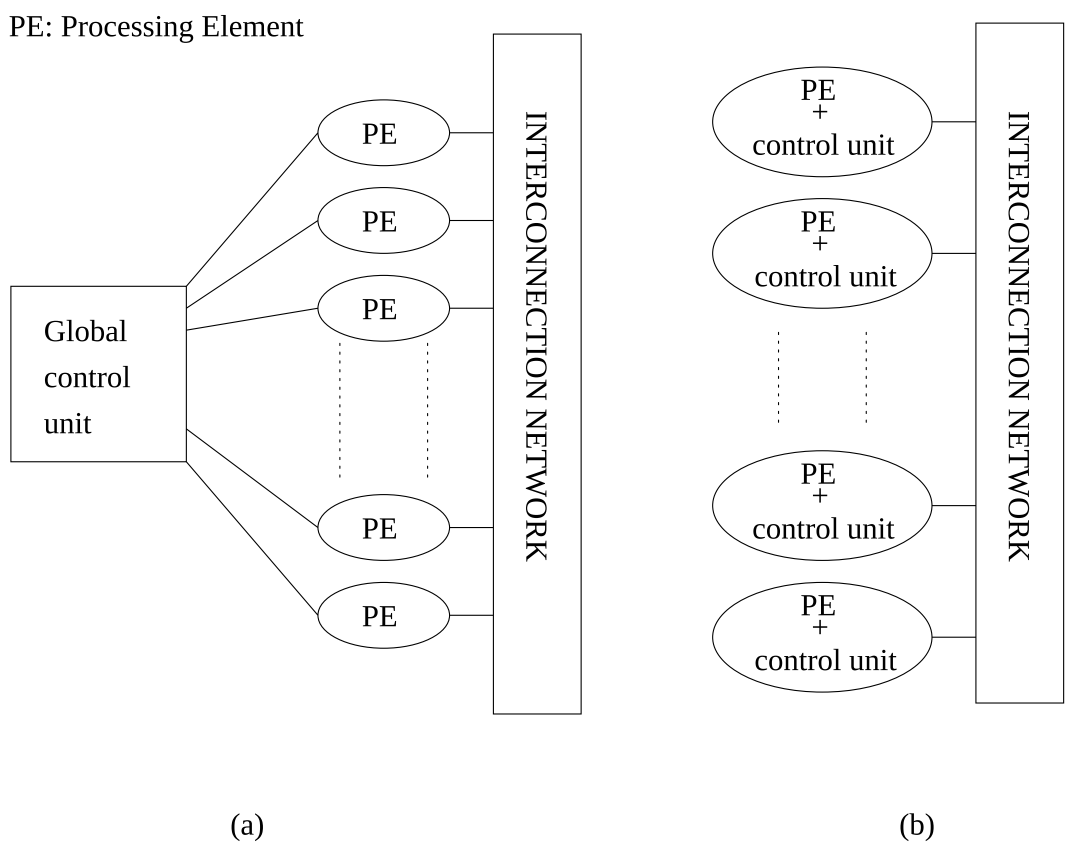
Control Structure of Parallel Programs

- Processing units in parallel computers either operate under the centralized control of a single control unit or work independently.

- If there is a single control unit that dispatches the same instruction to various processors (that work on different data), the model is referred to as single instruction stream, multiple data stream (SIMD).

- If each processor has its own control control unit, each processor can execute different instructions on different data items. This model is called multiple instruction stream, multiple data stream (MIMD).

SIMD and MIMD Processors



A typical SIMD architecture (a) and a typical MIMD architecture (b).

SIMD Processors

- Some of the earliest parallel computers such as the Illiac IV, MPP, DAP, CM-2, and MasPar MP-1 belonged to this class of machines.
- SIMD relies on the regular structure of computations (such as those in image processing).
- CUDA programs compile to the PTX instruction set. That instruction set does not contain SIMD instructions. So, CUDA programs cannot make explicit use of SIMD. Individual threads are part of groups called warps, within which every thread executes exactly the same sequence of instructions. Nvidia call it Single Instruction, Multiple Thread (SIMT), but it's essentially SIMD.

Parallel Thread Execution (PTX, or NVPTX) is a pseudo-assembly language used in Nvidia's CUDA programming environment. The nvcc compiler translates code written in CUDA, a C++-like language, into PTX, and the graphics driver contains a compiler which translates the PTX into a binary code which can be run on the processing cores.

MIMD Processors

- In contrast to SIMD processors, MIMD processors can execute different programs on different processors.
- A variant of this, called single program multiple data streams (SPMD) executes the same program on different processors.
- It is easy to see that SPMD and MIMD are closely related in terms of programming flexibility and underlying architectural support.
- Examples of such platforms include current generation Sun Ultra Servers, SGI Origin Servers, multiprocessor PCs, workstation clusters, and the IBM SP.

SIMD-MIMD Comparison

- SIMD computers require less hardware than MIMD computers (single control unit).
- However, since SIMD processors are specially designed, they tend to be expensive and have long design cycles.
- Not all applications are naturally suited to SIMD processors.
- In contrast, platforms supporting the SPMD paradigm can be built from inexpensive off-the-shelf components with relatively little effort in a short amount of time.
- MPI is primarily for SPMD/MIMD. HPF is an example of a SIMD interface.

Communication Model of Parallel Platforms

- There are two primary forms of data exchange between parallel tasks – accessing a shared data space and exchanging messages.

- Platforms that provide a shared data space are called shared-address-space machines or multiprocessors.

  1. Consistency problems
  2. Global view of memory

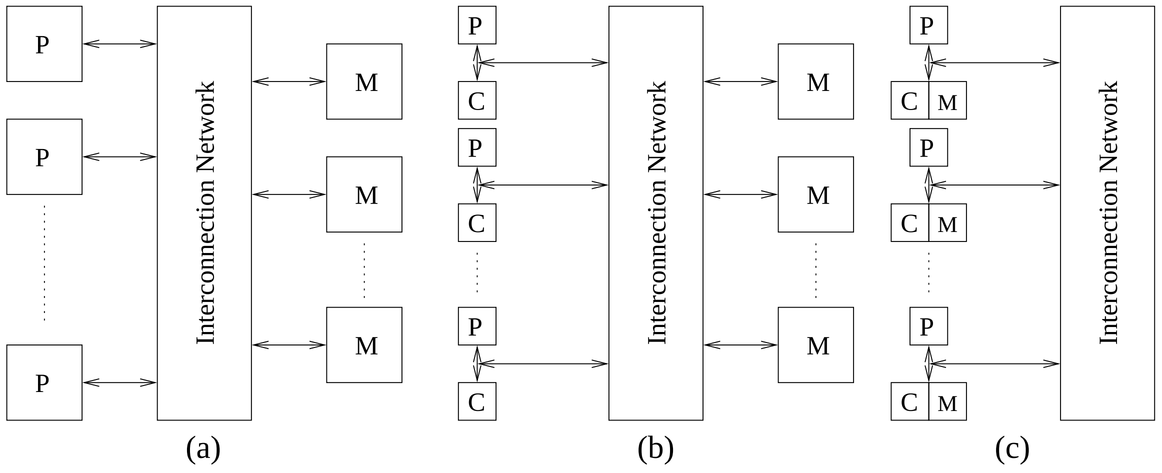
- Platforms that support messaging are also called message passing platforms or multicomputers.

  1. No consistency problems
  2. Lack of global view of memory
  3. Communication is costly

Shared-Address-Space Platforms

- Part (or all) of the memory is accessible to all processors.
- Processors interact by modifying data objects stored in this shared-address-space.
- If the time taken by a processor to access any memory word in the system global or local is identical, the platform is classified as a uniform memory access (UMA), else, a non-uniform memory access (NUMA) machine.

NUMA and UMA Shared-Address-Space Platforms



Typical shared-address-space architectures: (a) Uniform-memory-access shared-address-space computer; (b) Uniform-memory-access shared-address-space computer with caches and memories; (c) Non-uniform-memory-access shared-address-space computer with local memory only.

NUMA and UMA Shared-Address-Space Platforms

- The distinction between NUMA and UMA platforms is important from the point of view of algorithm design. NUMA machines require locality from underlying algorithms for performance.
- Programming these platforms is easier since reads and writes are implicitly visible to other processors.
- However, read-write data to shared data must be coordinated (this will be discussed in greater detail when we talk about threads programming).
- Caches in such machines require coordinated access to multiple copies. This leads to the cache coherence problem.
- A weaker model of these machines provides an address map, but not coordinated access. These models are called non cache coherent shared address space machines.

Shared-Address-Space vs. Shared Memory Machines

- It is important to note the difference between the terms shared address space and shared memory.
- We refer to the former as a programming abstraction and to the latter as a physical machine attribute.
- It is possible to provide a shared address space using a physically distributed memory.

Message-Passing Platforms

- These platforms comprise of a set of processors and their own (exclusive) memory.
- Instances of such a view come naturally from clustered workstations and non-shared-address-space multicomputers.
- These platforms are programmed using (variants of) send and receive primitives.
- Libraries such as MPI and PVM provide such primitives.

Message Passing vs. Shared Address Space Platforms

- Message passing requires little hardware support, other than a network.
- Shared address space platforms can easily emulate message passing. The reverse is more difficult to do (in an efficient manner).

Communication Costs in Parallel Machines

- Along with idling and contention, communication is a major overhead in parallel programs.

- The cost of communication is dependent on a variety of features including the programming model semantics, the network topology, data handling and routing, and associated software protocols.

Message Passing Costs in Parallel Computers

The total time to transfer a message over a network comprises of the following:

- *Startup time* (t_s): Time spent at sending and receiving nodes (executing the routing algorithm, programming routers, etc.).
- *Per-hop time* (t_h): This time is a function of number of hops and includes factors such as switch latencies, network delays, etc.
- *Per-word transfer time* (t_w): This time includes all overheads that are determined by the length of the message. This includes bandwidth of links, error checking and correction, etc.

Store-and-Forward Routing

- A message traversing multiple hops is completely received at an intermediate hop before being forwarded to the next hop.

- The total communication cost for a message of size m words to traverse l communication links is

$$t_{comm} = t_s + (mt_w + t_h)l.$$

- In most platforms, t_h is small and assume $l = 1$ then the above expression can be approximated by

$$t_{comm} = t_s + mt_w.$$

- See homework.

MPI: the Message Passing Interface

- MPI defines a standard library for message-passing that can be used to develop portable message-passing programs using either C or Fortran.

- The MPI standard defines both the syntax as well as the semantics of a core set of library routines.

- Vendor implementations of MPI are available on almost all commercial parallel computers.

- It is possible to write fully-functional message-passing programs by using only the six routines.

- Syntax is the grammar. It describes the way to construct a correct sentence. For example, this water is triangular is syntactically correct.
- Semantics relates to the meaning. this water is triangular does not mean anything, though the grammar is ok.

MPI: the Message Passing Interface

The minimal set of MPI routines.

`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_Recv`	Receives a message.

Starting and Terminating the MPI Library

- `MPI_Init` is called prior to any calls to other MPI routines. Its purpose is to initialize the MPI environment.
- `MPI_Finalize` is called at the end of the computation, and it performs various clean-up tasks to terminate the MPI environment.
- The prototypes of these two functions are:

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

- `MPI_Init` also strips off any MPI related command-line arguments.
- All MPI routines, data-types, and constants are prefixed by "`MPI_`". The return code for successful completion is `MPI_SUCCESS`.

Communicators

- A communicator defines a *communication domain* – a set of processes that are allowed to communicate with each other.
- Information about communication domains is stored in variables of type `MPI_Comm`.
- Communicators are used as arguments to all message transfer MPI routines.
- A process can belong to many different (possibly overlapping) communication domains.
- MPI defines a default communicator called `MPI_COMM_WORLD` which includes all the processes.

Querying Information

- The `MPI_Comm_size` and `MPI_Comm_rank` functions are used to determine the number of processes and the label of the calling process, respectively.

- The calling sequences of these routines are as follows:

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

- The rank of a process is an integer that ranges from zero up to the size of the communicator minus one.

Our First MPI Program

```
#include <mpi.h>

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();
}
```

Sending and Receiving Messages

- The basic functions for sending and receiving messages in MPI are the `MPI_Send` and `MPI_Recv`, respectively.

- The calling sequences of these routines are as follows:

```
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)
```

- MPI provides equivalent datatypes for all C datatypes. This is done for portability reasons.

- The datatype `MPI_BYTE` corresponds to a byte (8 bits) and `MPI_PACKED` corresponds to a collection of data items that has been created by packing non-contiguous data.

- The message-`tag` can take values ranging from zero up to the MPI defined constant `MPI_TAG_UB`.

A sending-receiving MPI Program

```
#include <mpi.h>

main(int argc, char *argv[])
{
  int npes, myrank, a[10], b[10];

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &npes);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

  if (myrank==0){
    a[0]=0; b[0]=1;
    MPI_Send(a,1,MPI_INT,1,1,MPI_COMM_WORLD);
  }
  else{
    a[0]=2; b[0]=3;
    MPI_Recv(a,1,MPI_INT,0,1,MPI_COMM_WORLD,&status);
  }
  printf("From processor: %d out of %d a=%d b=%d\n"
          ,myrank,npes,a[0],b[0]);
  MPI_Finalize();
}
```

- Homework: Compute the start up time (ts) and transmission time per word (tm), note that this is so called uncongested time.