Introduction to Parallel Programming

Haitao Wei

University of Delaware

http://www.udel.edu

Computer Architecture and Parallel Systems Laboratory

http://www.capsl.udel.edu

Outline

- Part1:Introduction to parallel programming

- Part2:Parallel Programming Tutorials
  - MPI
  - Pthreads
  - OpenMP

Outline

- Models for Parallel Systems
- Parallelization of Programs
- Levels of Parallelism
  - Instruction level
  - Data Parallelism
  - Loop Parallelism
  - Functional/task Parallelism
- Parallel Programming Patterns
- Performance Metrics

Models for Parallel System

- Machine Model
  - Describe machines at lowest level of abstraction of hardware, e.g., registers,
- Architecture Model
  - Describe architecture at the level of how processing units, memory organization, interconnect organized and execution model of instructions
- Computation Model
  - Formal model of AM for designing and analyzing algorithm, e.g. RAM, PRAM
- Programming Model
  - Describe the machine from the programmer point of view: how the programmer to code

Parallel Programming Model

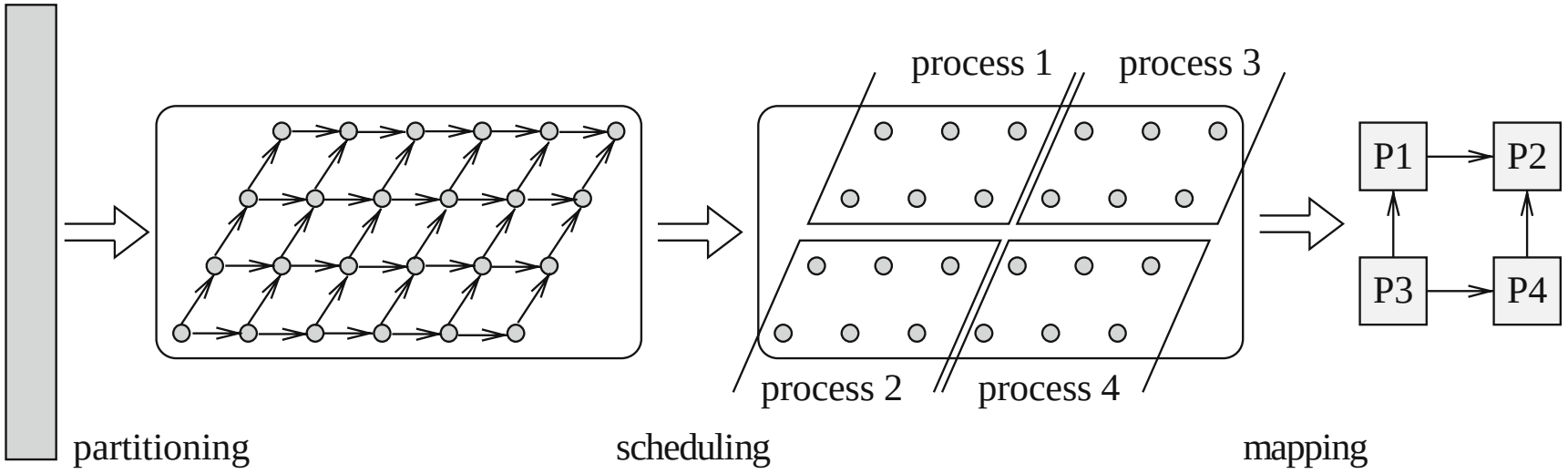
- Parallel Programming model Influenced by
  - Architecture Design
  - Programming Language
  - Compiler
  - Runtime
- Several criteria make them different
  - level of parallelism (instruction level, data level, loops level, procedural level)
  - implicit or use-defined explicit specified parallelism
  - how parallel program parts are specified
  - the execution model of parallel units (SIMD, SPMD, Sync, Async)
  - how to communicate (explicit comm or shared variables)

Outline

- Models for Parallel Systems
- Parallelization of Programs
- Levels of Parallelism
  - Instruction level
  - Data Parallelism
  - Loop Parallelism
  - Functional/task Parallelism
- Parallel Programming Patterns
- Performance Metrics

Parallelization of Programs

- Typical Parallelization Steps
  - Partition/Decomposition: Algorithm is split into tasks with dependencies
  - Scheduling: Tasks are assigned to processes
  - Mapping: Processes are mapped to physical processors



Parallelization of Programs

- Partition/Task Decomposition
  - Task: a sequence of computation unit of parallelism, can be at different levels: instruction level, loop level, functional level
  - Task Granularity: Coarse grained and fine grained
  - Compromise between number of tasks and granularity: enough tasks to keep all processors busy and enough granularity to amortize the scheduling/mapping overhead

Parallelization of Programs

- Assign tasks to processes/threads
  - A process normally executes multiple different tasks
  - The goal is load balance: each process should have about the same number of computations to perform
  - Static (at the initialization phrase at program start) or Dynamic (during program execution)

Parallelization of Programs

- Mapping processes to physical processor/cores
  - Each process or thread is mapped to a separate processor or core
  - Goal: get an equal utilization of the physical processors or cores while keeping communication between the processors as small as possible.

Outline

- Models for Parallel Systems
- Parallelization of Programs
- Levels of Parallelism
  - Instruction level
  - Data Parallelism
  - Loop Parallelism
  - Functional/task Parallelism
- Parallel Programming Patterns
- Performance Metrics

Parallelism at Instruction Level

- Task unit: one instruction
- Dependency: True, Anti, Out between instructions
- Scheduling: schedule instructions to execute in different function unit

```
For i (1…n)
   C[i]=A[i]+B[i]-A[i]*B[i]
```

```
Loop:
  LD R1, @A
 LD R2, @B
 ADD R3, R1, R2
 MUL R4, R1, R2
 SUB  R5, R3, R4
 ST R5, @C
 JNZ Loop
```

```
Loop:
   LD R1, @A
  LD R2, @B
  ADD R3, R1, R2   MUL R4, R1, R2
  SUB  R5, R3, R4
  ST R5, @C
  JNZ Loop
```

Parallelism at Instruction Level

- How to program Instruction Level Parallelism?
  - Write assembly code by hand—find the dependencies and schedule by hand
  - Hardware helps you automatically—Superscalar processor
  - Compiler helps you automatically—scheduling techniques for VLIW processor

Data Parallelism

- Task unit: one operation on single element
- Dependency: None
- Schedule: Pack different data element into SIMD instruction

```
for(i=0;i<n;i++)
   C[i]=A[i]+B[i]-A[i]*B[i]
```

```
for i=0;i<n;i+=4){
   C[i…i+3]=A[i…i+3]+B[i…i+3]-
A[i…i+3]*B[i…i+3]
```

Data Parallelism

- How to program Data Parallelism?
  - Write assembly code by hand—using SIMD instructions, e.g. MMX, SSE
  - Let compiler help you automatically—Using auto-vectorization, e.g. gcc “-ftree-vectorize -msse2”, but not as high efficient as hand coded
  - Using data-parallel programming language

Loop Parallelism

- Task unit: one iteration of the loop
- Dependency: dependencies between loop iterations
- Schedule: schedule different loop iterations to execute on different processors/cores

```
for(i=0;i<n;i++)
   C[i]=A[i]+B[i]-A[i]*B[i]
```

```
for(i=0;i<n/2;i++)
   C[i]=A[i]+B[i]-A[i]*B[i]
```

Core0

```
for(i=n/2+1;i<n;i++)
   C[i]=A[i]+B[i]-A[i]*B[i]
```

Core1

without any dependencies between iterations

Loop Parallelism

- How to program Loop Parallelism?
  - Write multithread code by hand—Decompose the loop into different threads
  - Using high level programming language—e.g., OpenMP

```
#pragma omp parallel
for(i=0;i<n;i++)
   C[i]=A[i]+B[i]-A[i]*B[i]
```

  - Compiler do it—Under research, some experimental compilers, e.g. PLUTO

Functional Parallelism

- Task unit: code segments (statements, basic block, loops, functions)
- Dependency: dependencies between tasks
- Schedule: schedule different tasks to execute on different processors/cores

Fib(n)=Fib(n-1)+Fib(n-2)

```
f1=Fib(n-1)
f2=get_result_from_core1   Core0
return f1+f2;

f=Fib(n-2)                 Core1
return f
```

Functional Parallelism

- How to program Functional Parallelism?
  - Write multithread code by hand—Decompose the computation into different threads
  - Using high level programming language—e.g., OpenMP, Cilk, Codelet

```
int fib(int n)
{
   if (n < 2)
      return n;
   int x = cilk_spawn fib(n-1);
   int y = fib(n-2);
   cilk_sync;
   return x + y;
}
```

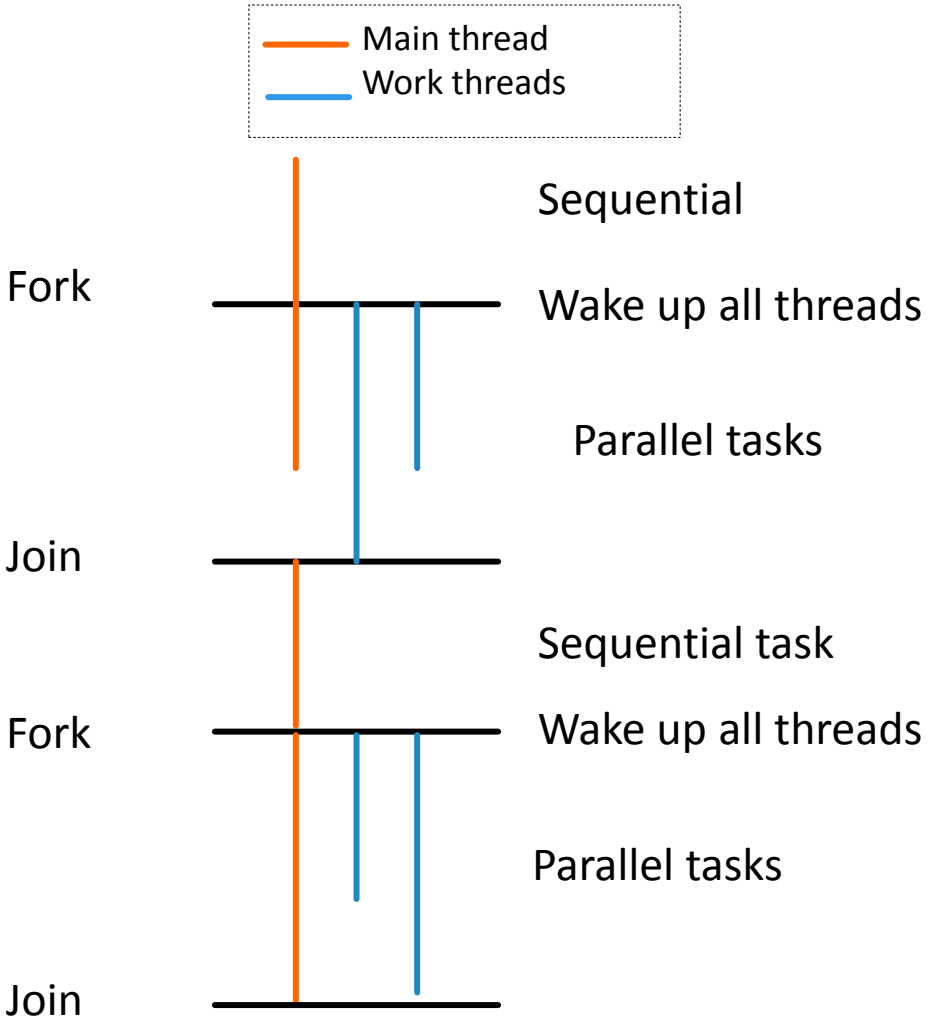
Outline

- Models for Parallel Systems
- Parallelization of Programs
- Levels of Parallelism
  - Instruction level
  - Data Parallelism
  - Loop Parallelism
  - Functional/task Parallelism
- Parallel Programming Patterns
- Performance Metrics

Parallel Programming Patterns

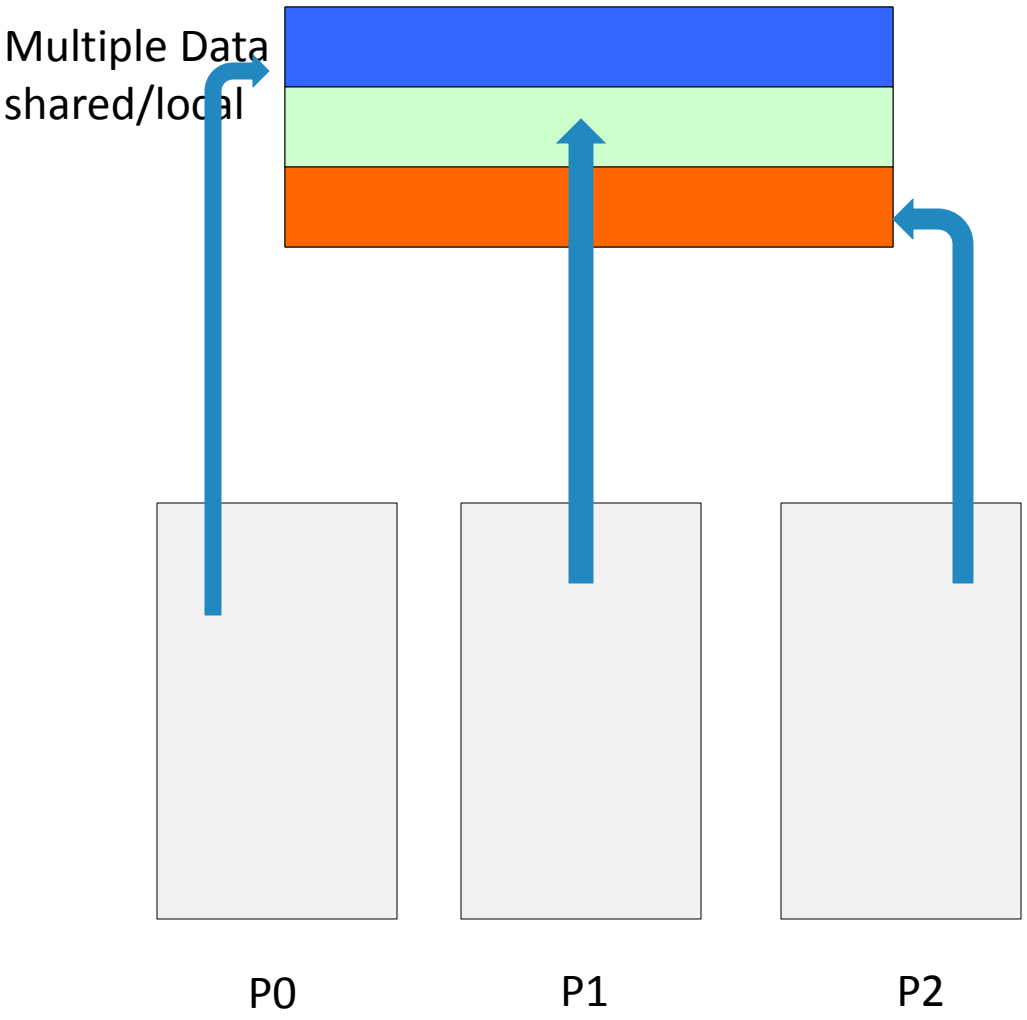
- Parallel programs consists of a collection of tasks that are executed by processes/threads
- Patterns: provide specific coordination structures for processes/threads
  - Fork-Join
  - SPMD and SIMD
  - Master–Slave
  - Client–Server
  - Pipelining
  - Task Pools
  - Producer–Consumer

Fork-Join



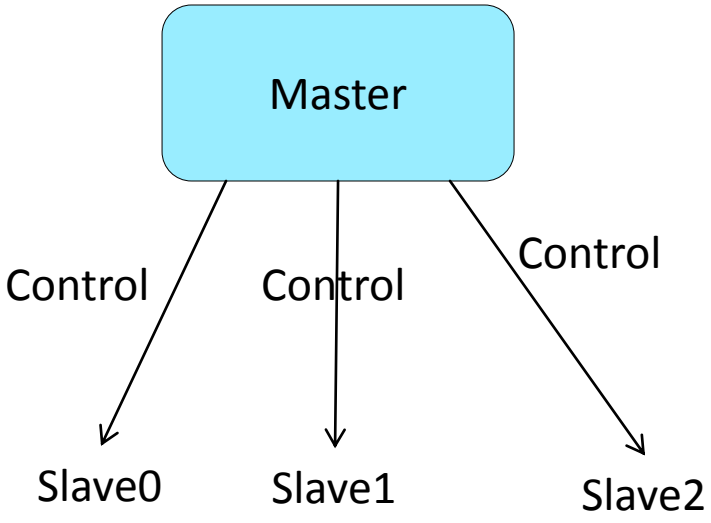
- Initial time, there is only one main thread do sequential work
- Fork all work threads to do the work in parallel
- Join all the threads and continue to do sequential work

Single Program and Multiple Data



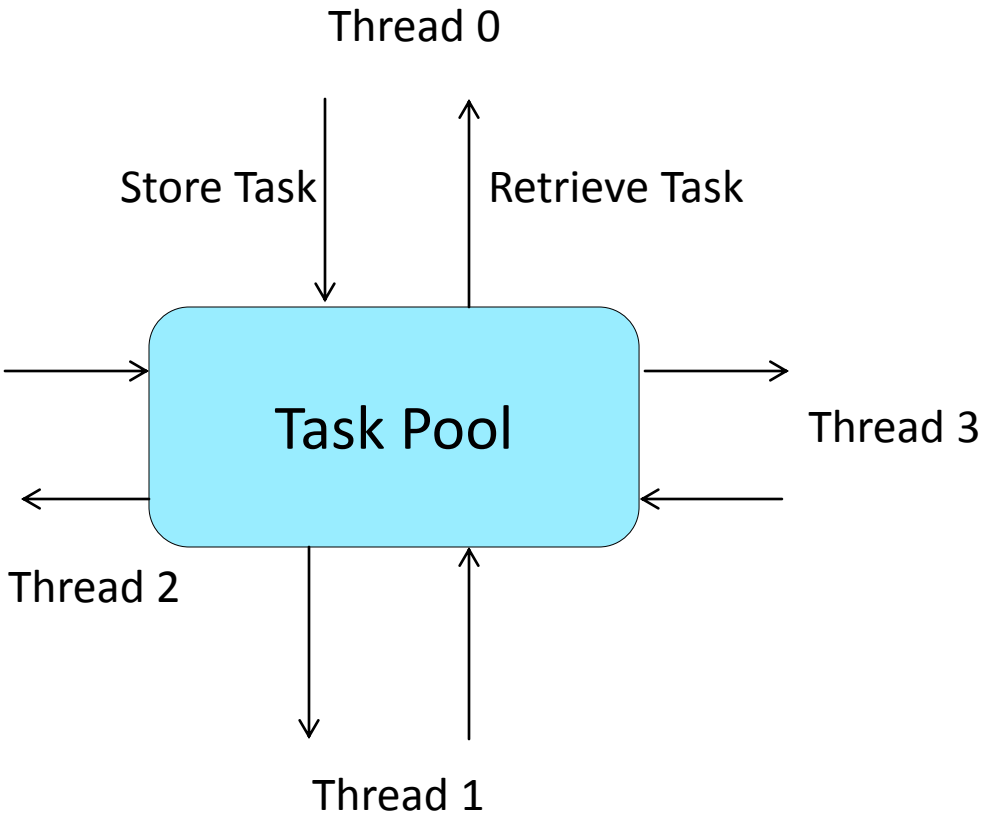
- Each processor executes the same copy of the program
- Each processor has a logical copy of data
- Each processor uses *p_id* to find their own part data

Master and Slave



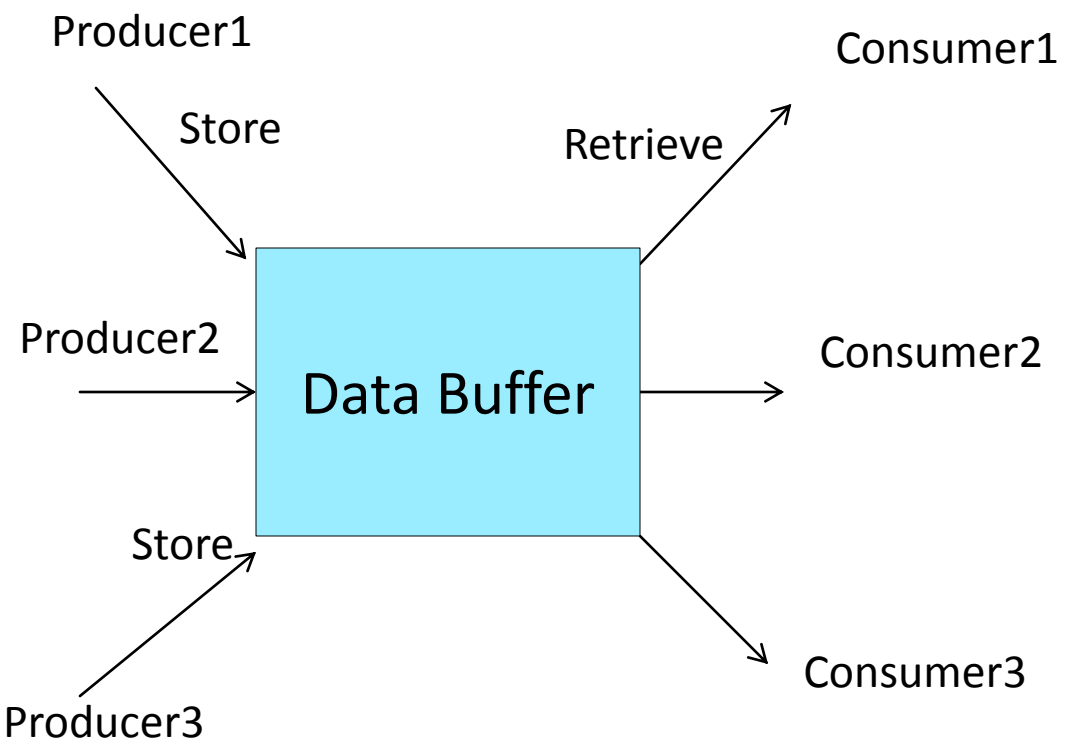
- Master control the main function of program execution
- Slave does the actual computation which is assigned by Master thread

Task Pool



- Tata structure in which tasks to be performed are stored and from which they can be retrieved for execution
- A fixed number of threads is used for the processing of the tasks
- a thread can generate new tasks and insert them into

Producer-Consumer



- Producer threads produce data which are used as input by consumer threads
- Common data buffer is used, which can be accessed by both of threads
- Synchronization has to be used to ensure a correct coordination between producer and consumer threads

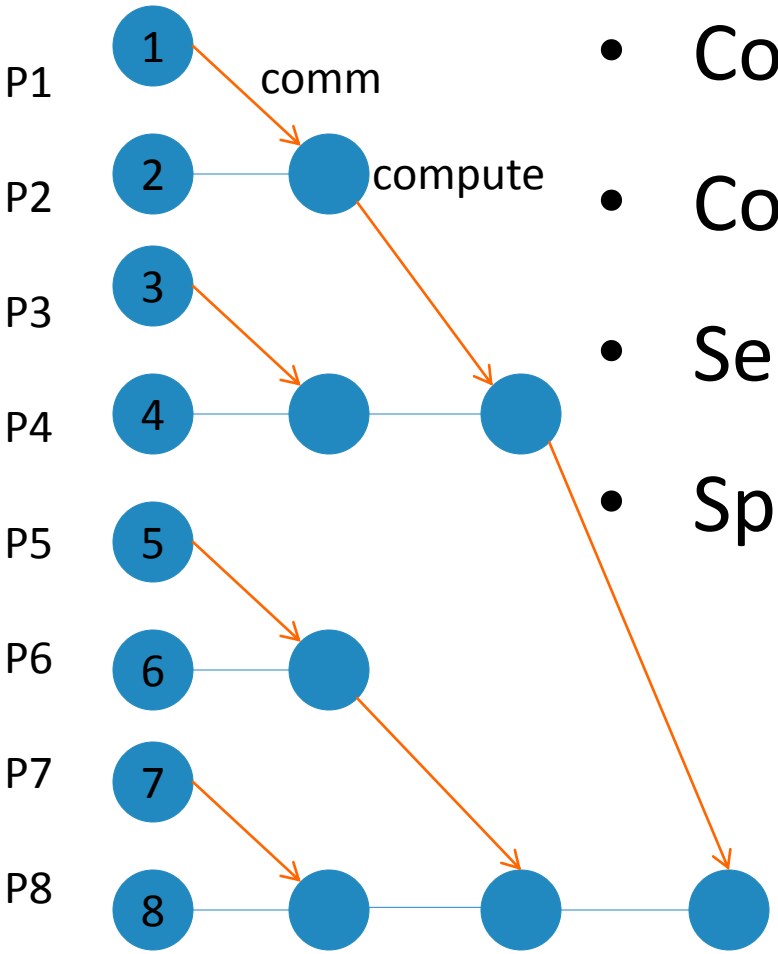
Outline

- Models for Parallel Systems
- Parallelization of Programs
- Levels of Parallelism
  - Instruction level
  - Data Parallelism
  - Loop Parallelism
  - Functional/task Parallelism
- Parallel Programming Patterns
- Performance Metrics

Performance Metrics for Parallel Programs

- Execution Time
  - Sequential execution time: T_s
  - Parallel execution time: T_p
  - Overhead: $To = pT_p - T_s$
- Speedup
  - $Speedup = T_s / T_p$

Adding N numbers using N processing elements



- Communication: LogN
- Compute: LogN
- Sequential: N-1
- Speedup=(N-1)/(2LogN)

  =O(N/LogN)

Amdahl's law

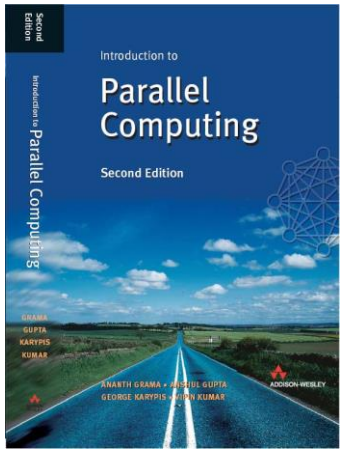
- A program execution time T_s is composed of a fraction of sequential execution time $T_s * f$ and a fraction of parallel execution time $T_s * (1-f)$
- *Speedup=*

$$\frac{T_s}{T_s * f + T_s * \frac{1 - f}{p}} = \frac{1}{f + \frac{1 - f}{p}} £ \frac{1}{f}$$

Amdahl's law

- Do you really need parallel computing for your program?
- Speedup is limited by the sequential part of your program
- What is the bottleneck, how many benefits can you get if you try to parallelize it?

References



Introduction to Parallel Computing
http://www-users.cs.umn.edu/~karypis/parbook/

Parallel Programming: for Multicore and Cluster Systems
http://www.amazon.com/Parallel-Programming-Multicore-Cluster-Systems/dp/364204817X

MPI Programming: A Tutorial

Haitao Wei

Thanks to slides from Robert Pavel

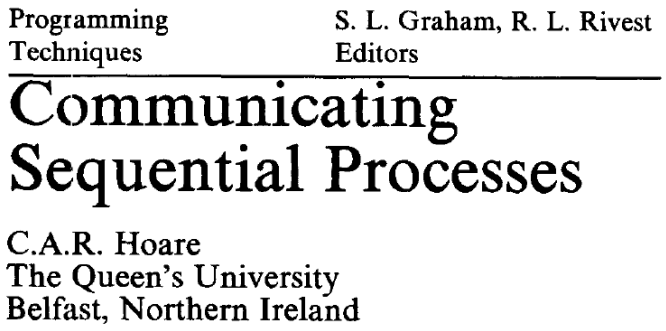
and Daniel Orozco

University of Delaware
http://www.udel.edu

Computer Architecture and
Parallel Systems Laboratory
http://www.capsl.udel.edu

MPI

- Stands for **M**essage **P**assing application programmer **I**nterface.
- It is a specification. There is not one MPI.
- The specification describes primitives that can be used to communicate and program.
- Inspired by the Communicating Sequential Processes paper.


Programming Techniques
S. L. Graham, R. L. Rivest Editors

Communicating Sequential Processes

C.A.R. Hoare
The Queen's University
Belfast, Northern Ireland

Why Learn MPI?

- MPI is the *de facto* standard to program MIMD systems.
- It can be used in SMP systems as well.
- Very versatile, can run on:
  - Symmetric or asymmetric systems
  - Local networks or over the internet
  - On serial processors

Why Learn MPI?

- Comparatively easy to use.
- All communication is explicit.
- Easy to learn
- Reasonably good performance
- Most importantly: Everyone already uses it

What is MPI?: An example

- MPI is like exchanging emails with your advisor
- Your advisor gets hundreds of emails per day
  - If he doesn't know an email is coming, he can't respond
- But if he is expecting an email, he'll read it

MPI: Execution Model

- There are P processes that are created at the beginning.
- All processes execute the same program.
- Processes communicate and synchronize using **send and receive** operations.
- Operations can be **blocking** or **nonblocking.**

The MPI Programming Model

- Write ONE program that everybody runs.
- Initialize the MPI library:
  - MPI_Init
- Clean the MPI library at the end:
  - MPI_Finalize

The Basics

MPI_Init

Initialize the MPI execution environment

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

Input Parameters

argc

Pointer to the number of arguments

argv

Pointer to the argument vector

MPI_Finalize

Terminates MPI execution environment

Synopsis

```
int MPI_Finalize( void )
```

The Basics

MPI_Comm_rank

Determines the rank of the calling process in the communicator

Synopsis

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

Input Argument

comm

communicator (handle)

Output Argument

rank

rank of the calling process in the group of comm (integer)

The Basics

MPI_Comm_size

Determines the size of the group associated with a communicator

Synopsis

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

Input Parameter

comm

communicator (handle)

Output Parameter

size

number of processes in the group of comm (integer)

Code Example

Hello World!

```
1  #include <mpi.h>
2
3  main(int argc, char *argv[])
4  {
5    int npes, myrank;
6
7    MPI_Init(&argc, &argv);
8    MPI_Comm_size(MPI_COMM_WORLD, &npes);
9    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
10    printf("From process %d out of %d, Hello World!\n",
11           myrank, npes);
12  MPI_Finalize();
13  }
```

```
./mpicc hello.c -o hello
./mpirun -np 3 hello
From process 2 out of 3, Hello World!
From process 1 out of 3, Hello World!
From process 0 out of 3, Hello World!
```

Sending and Receiving Data

- Now let's actually do something useful
- MPI, at it simplest, is a series of matched sends and receives
- Host A sends a message to Host B. Host B receives the message
- These sends and receives are blocking by default
- What is blocking?

A Visual Example

Working

Working

A Visual Example

Should I use red paint?

Working

A Visual Example

...

Working

A Visual Example

Yay, I got an ack. Back to work…

Yes. Use red Ack

A Visual Example

Working

Working

Now with code?

First, let’s learn us some syntax!

MPI_Send

Performs a blocking send

Synopsis

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest,
        int tag, MPI_Comm comm)
```

Input Parameters

buf

initial address of send buffer (choice)

count

number of elements in send buffer (nonnegative integer)

datatype

datatype of each send buffer element (handle)

dest

rank of destination (integer)

tag

message tag (integer)

comm

communicator (handle)

MPI_Recv

Blocking receive for a message

Synopsis

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source,
        int tag, MPI_Comm comm, MPI_Status *status)
```

Output Parameters

buf

initial address of receive buffer (choice)

status

status object (Status)

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)

comm

communicator (handle)

Now for Examples

The right way

```
MPI_Comm_rank (comm, &myrank);
if (my rank == 0) {
  MPI_Send (sendbuf, count, MPI_INT, 1,
tag, comm);
  MPI_Recv (recvbuf, count, MPI_INT, 1,
tag, comm,    &status);
}
else if (my rank == 1) {
  MPI_Recv (recvbuf, count, MPI_INT, 0,
tag, comm, &status);
  MPI_Send (sendbuf, count, MPI_INT, 0,
tag, comm);
}
```

the wrong way

```
MPI_Comm_rank (comm, &my rank);
if (my rank == 0) {
MPI_Recv (recvbuf, count, MPI_INT, 1, tag,
comm, &status);
MPI_Send (sendbuf, count, MPI_INT, 1, tag,
comm);
}
else if (my rank == 1) {
MPI_Recv (recvbuf, count, MPI_INT, 0, tag,
comm, &status);
MPI_Send (sendbuf, count, MPI_INT, 0, tag,
comm);
}
```

Huh?

- Why didn't the wrong way work?
  - …
- Deadlock
- Both processes are waiting for a message to arrive.

Can we work around that?

Yes!

Non-blocking communication is one way

MPI_Isend and MPI_Irecv

- You can look online for the syntax
  - And it is really long…
- The simple logic is
  - Start a send/recv
  - Do some work
  - Only check when you need the data

A Visual Example

Should I use red paint?

Working

A Visual Example

Working with the green

Working

A Visual Example

Working with the green

Yes, use the red paint. Ack

A Visual Example

Working with the green

Working

A Visual Example

Oh, cool. I can use red

Working

A Visual Example

Working with the red

Working

A Code Example

The Wrong Way done Right!

```
MPI_Comm_rank (comm, &my rank);
if (my rank == 0) {
MPI_IRecv (recvbuf, count, MPI_INT, 1, tag, comm, &recv_request);
MPI_ISend (sendbuf, count, MPI_INT, 1, tag, comm, &send_request);
MPI_Wait(&recv_request, &status);
MPI_Wait(&send_request, &status);
}
else if (my rank == 1) {
MPI_IRecv (recvbuf, count, MPI_INT, 0, tag, comm, &recv_request);
MPI_ISend (sendbuf, count, MPI_INT, 0, tag, comm, &send_request);
MPI_Wait(&recv_request, &status);
MPI_Wait(&send_request, &status);

}
```

The Last Must-Have Tool

- What if we need to ensure one phase is complete before starting the next
  - Finish washing your hands before you leave the restroom
- How to guarantee that in MPI?
  - A series of blocking sends and receives?
  - Something else?

Or just use Barrier

- A Barrier halts execution of code until all processes have signaled that they have reached a barrier
- Many ways to implement a barrier
  - We may discuss these during the course
- Only use a Barrier if you need to
  - Hurts performance due to idle processes

Barrier!

MPI_Barrier

Blocks until all processes in the communicator have reached this routine.

Synopsis

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

Input Parameter

comm

communicator (handle)

Code Example

Let's see how to use a Barrier

```
MPI_Comm_rank (comm, &myrank);
if (my rank == 0) {
  //p0 does something in stage 1
   MPI_Barrier(comm);
  //p0 does something in stage 2
  MPI_Barrier(comm);
}
else if (my rank == 1) {
   //p1 does something in stage 1
   MPI_Barrier(comm);
 //p1 does something in stage 2
   MPI_Barrier(comm);
}
```

Advanced MPI

- Other types of Send/Recv
  - Buffered: Copies data to another buffer
  - MPI_Ssend: Won't return until recv is completed
  - MPI_Rsend: May only be used if matching recv is already active
  - MPI_SendRecv: Combines Send and Receive into a single command

Collective Operations

- An alternative to point to point operations.
- Involve communication and synchronization between many processes.
- The two most common are:
  - MPI_Bcast(…, root, ...):
    - All processes call the same function.
    - All processes receive data from process root.
  - MPI_Reduce(…, root, …)
    - A reduction operation is done with data from each process and the result is given to the root process.

Collective Example: Intuitive

- TA normally wants to inform you we have a huge project.
  - So TA “Broadcast” the information in a mass email.
- TA will also collect the “quiz” after the midterm exam
  - a reduction is performed where all of you put your quizzes in the basket of the root process(TA).

Dot Product

```
#include "mpi.h”
main(int argc, char* argv[]) {
//Step 1:initialize vector a and b
float loc_dot=0.0f;
float dot=0.0f;
float a[N],b[N],loc_dots[N];
for (i = 0;i<N;i++) {
    a[i] = i;
    b[i] = i+1;}
//Step2 initialize MPI
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
MPI_Comm_size(MPI_COMM_WORLD,&p);
//Step3: Each processor computes a local dot product
    loc_n = N/p;
    bn = (my_rank)*loc_n;
    en = bn + loc_n;
    loc_dot = 0.0;
    for (i = bn;i <en; i++) {
     loc_dot = loc_dot + a[i]*b[i];
    }
// Step4: collect result
MPI_Reduce(&loc_dot, dot, 1, MPI_FLOAT,
MPI_SUM, 0, MPI_COMM_WORLD);

if(my_rank==0)
  printf( "dot product = %f",dot);

/* mpi is terminated. */
     MPI_Finalize();
}
```

Thoughts on Advanced MPI

- Be very careful
- Using collectives may kills performance
  - If only because they are blocking
- There may be special cases where the specialized send and recv are useful
- But unless you are in HPC, use whatever is most intuitive

Can I use MPI with…

- Fortran?
  - Yes
- C++?
  - Yes, and it is actually a very intuitive interface that I really like
- Java?
  - Not easily
- Python?
  - Yes
- Matlab?
  - Not easily

References

- The CSP paper
  - http://portal.acm.org/citation.cfm?id=359576.359585
- A Reference for MPI
  - http://www.mcs.anl.gov/research/projects/mpi/www/www3/