2110412 Parallel Comp Arch
Parallel Programming Paradigm

Natawut Nupairoj, Ph.D.
Department of Computer Engineering, Chulalongkorn University

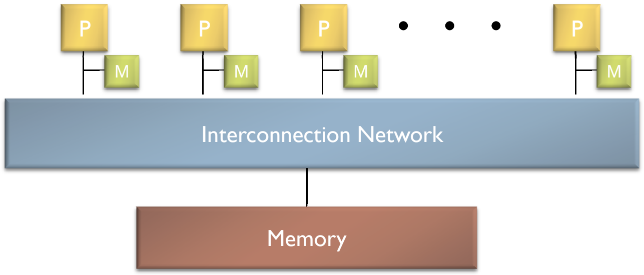
Outline

- Overview
- Parallel Architecture Revisited
- Parallelism
- Parallel Programming Model

What are the factors for parallel programming paradigm?

- System Architecture
- Parallelism – Nature of Applications
- Development Paradigms
  - Automatic
  - Semi-Auto (Directives / Hints)
  - Manual

Generic Parallel Architecture



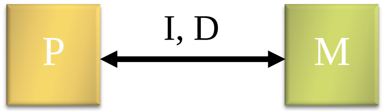
- Where is the memory physically located ?

Flynn's Taxonomy

- Very influential paper in 1966
- Two most important characteristics
  - Number of instruction streams.
  - Number of data elements.
  - **SISD** (Single Instruction, Single Data).
  - **SIMD** (Single Instruction, Multiple Data).
  - **MISD** (Single Instruction, Single Data).
  - **MIMD** (Multiple Instruction, Multiple Data).

SISD

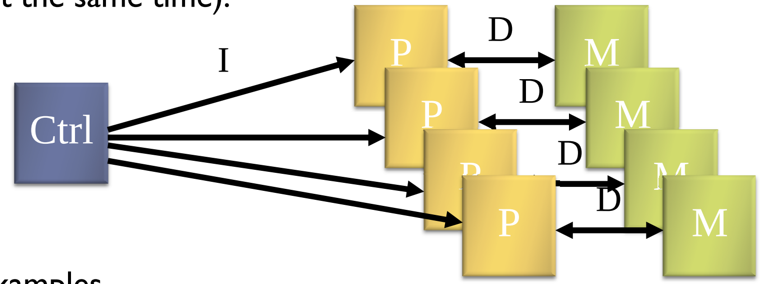
- One instruction stream and one data stream - from memory to processor.



- von Neumann's architecture.
- Example
  - PC.

SIMD

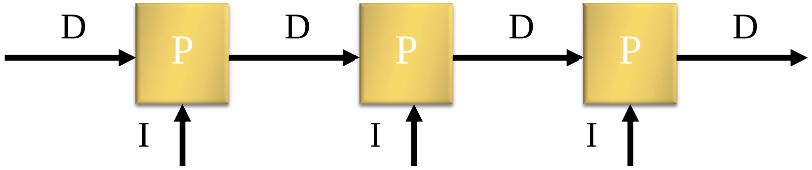
- One control unit tells processing elements to compute (at the same time).



- Examples
  - TMC/CM-1, Maspar MP-1, Modern GPU

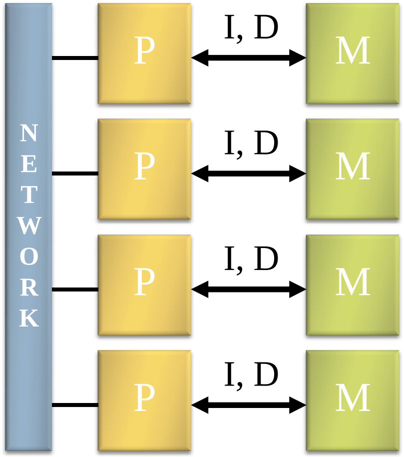
MISD

- No one agrees if there is such a MISD.
- Some say systolic array and pipeline processor are.



MIMD

- Multiprocessor, each executes its own instruction/data stream.
- May communicate with one another once in a while.
- Examples
  - IBM SP, SGI Origin, HP Convex, Cray …
  - Cluster
  - Multi-Core CPU

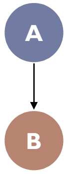


Parallelism

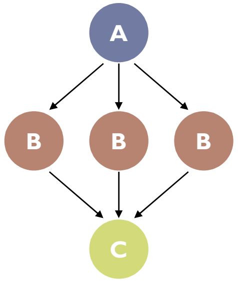
- To understand parallel system, we need to understand how can we utilize parallelism
- There are 3 types of parallelism
  - Data parallelism
  - Functional parallelism
  - Pipelining
- Can be described with data dependence graph

Data Dependence Graph

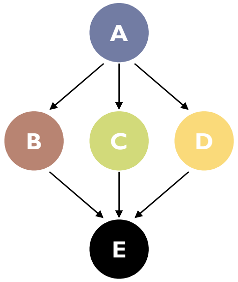
- A directed graph representing the dependency of data and order of execution
- Each vertex is a task
- Edge from A to B
  - Task A must be completed before task B
  - Task B is dependent on task A
- Tasks that are independent from one another can be perform concurrently



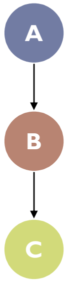
Parallelism Structure



Data Parallelism



Functional Parallelism

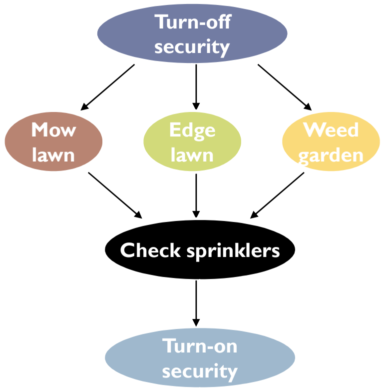


Pipelining

Example

- Weekly Landscape Maintenance
  - Mow lawn, edge lawn, weed garden, check sprinklers
  - Cannot check sprinkler until all other 3 tasks are done
  - Must turn off security system first
  - And turn it back on before leaving

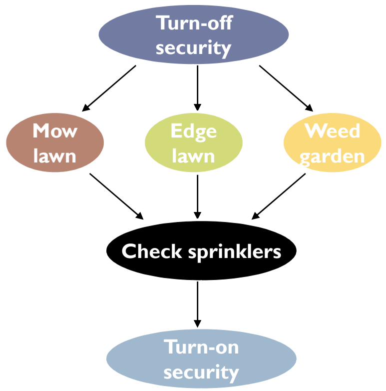
Example: Dependency Graph


Turn-off security
Mow lawn
Edge lawn
Weed garden
Check sprinklers
Turn-on security

- What can you do with a team of 8 people?

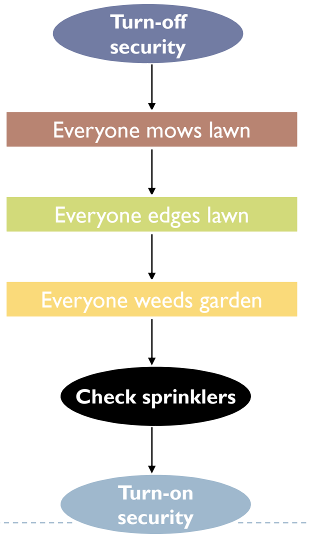
Functional Parallelism

- Apply different operations to different (or same) data elements
- Very straight forward for this problem
- However, we have 8 people?


Turn-off security
Mow lawn
Edge lawn
Weed garden
Check sprinklers
Turn-on security

Data Parallelism

- Apply the same operation to different data elements
- Can be processor array and vector processing
- Complier can help!!!


Turn-off security
Everyone mows lawn
Everyone edges lawn
Everyone weeds garden
Check sprinklers
Turn-on security

Sample Algorithm

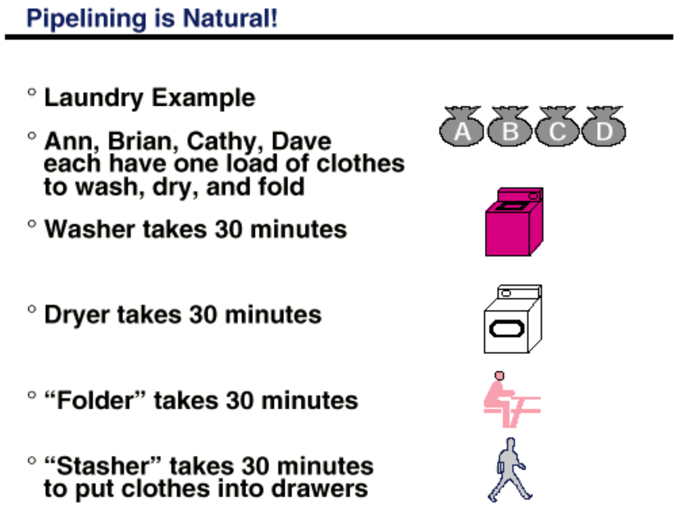
```
for i := 0 to 99 do
      a[i] := b[i] + c[i]
endfor

for i := 1 to 99 do
      a[i] := a[i-1] + c[i]
endfor

for i := 1 to 99 do
      for j := 0 to 99 do
            a[i,j] := a[i-1,j] + c[i,j]
      endfor
endfor
```

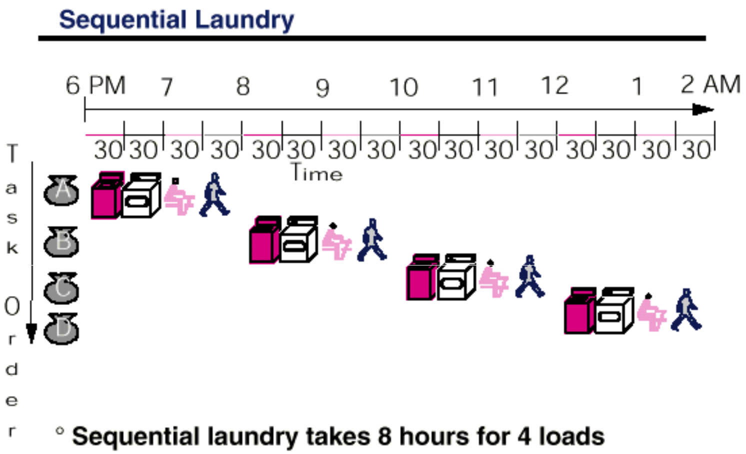
Pipelining

- Improve the execution speed
- Divide long tasks into small steps or "stages"
- Each stage executes independently and concurrently
- Move data toward workers (or stages)
- Pipelining is natural !!!
  (from David Patterson's lecture note)



Pipelining is Natural!

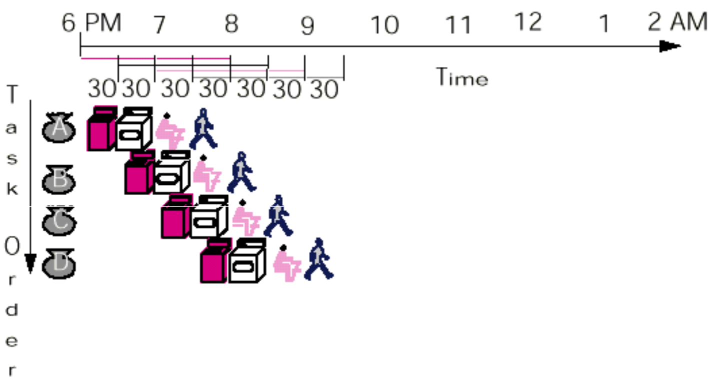
- Laundry Example
- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold
- Washer takes 30 minutes
- Dryer takes 30 minutes
- "Folder" takes 30 minutes
- "Stasher" takes 30 minutes to put clothes into drawers



Sequential Laundry

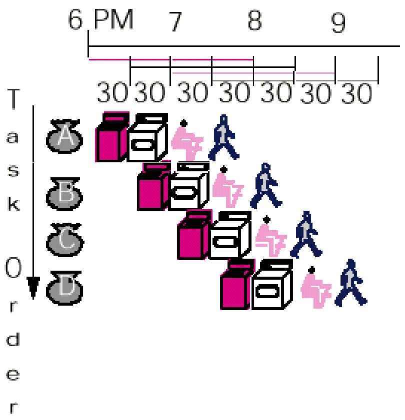
- Sequential laundry takes 8 hours for 4 loads
- If they learned pipelining, how long would laundry take?

Pipelined Laundry: Start work ASAP



° Pipelined laundry takes 3.5 hours for 4 loads!

Pipelining Lessons



- Pipelining doesn't help **latency** of single task.
- It helps **throughput** of entire workload.
- **Multiple** tasks operating simultaneously using different resources.
- Potential speedup = **number pipe stages**

Example

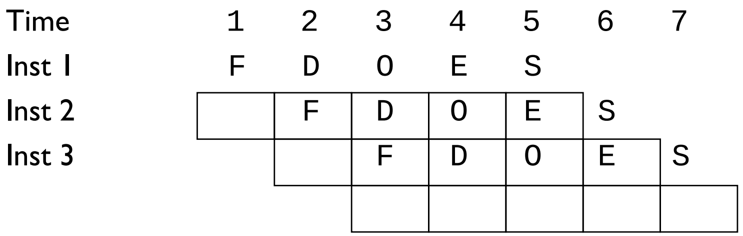
- Pipelining does not work for single data element !!!
- Pipelining is best for
  - Limited functional units
  - Each data unit cannot be partitioned
- For single house, pipelining is useless
- For multiple houses, still not good

Pipelining in Modern Processor

- The current trend is "super-pipelined".
  - The more stage, the better performance.
  - Not always true !!!
- Instruction cycle is divided into five stages:

Fetch → Decode → Operand Fetch → Execute → Store

Pipelining Execution



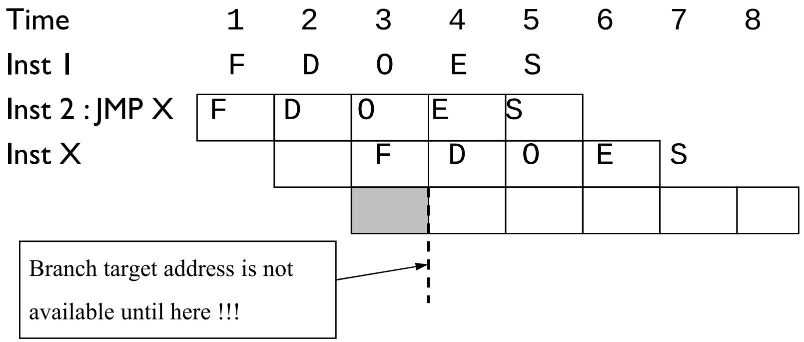
Time	1	2	3	4	5	6	7
Inst 1	F	D	0	E	S		
Inst 2		F	D	0	E	S	
Inst 3			F	D	0	E	S

Performance of Pipeline

- What do we gain ?
- Suppose we execute 1000 instructions on non-pipelined and pipelined CPUs
- Clock speed = 500 MHz (1 clock = 2 ns.)
- non-pipelined CPU:
  - total time = 2ns/cycle x 5 cycles/inst x 1000 instr.
    = 10 ms.
- Perfect pipelined CPU:
  - total time = 2ns/cycle x (1 cycle/inst x 1000 instr. + 4 cycles drain)
    = 2.008 ms.

Nothing is perfect !!!

- Problem with branch.
- Don't know what to fetch next until decoded.



Time	1	2	3	4	5	6	7	8
Inst 1	F	D	0	E	S			
Inst 2 : JMP X	F	D	0	E	S			
Inst X			F	D	0	E	S	

Branch target address is not available until here !!!

Stalled Pipe

- When pipelining is not smooth, we called it is "stalled"
- Branch and others ?
  - Subroutine calling
  - Memory accessing
  - Multi-cycle execution
  - Interrupt
  - Context switching
- These are common, thus, pipeline should not be too deep

Vector Processing

- Data parallelism technique
  - Perform the same function on multiple data elements (aka. “vector”)
  - Example: SAXPY (DAXPY) problem

```
for i := 0 to 63 do
     Y[i] := a*X[i] + Y[i]
endfor
```

Vector Processing

```
LV V1,R1          ; R1 contains based address for “X[*]”
LV V2,R2          ; R2 contains based address for “Y[*]”
ADDSV V3,R3,V1    ; a*X -- R3 contains the value of “a”
ADDV V1,V3,V2     ; a*X + Y
SV R2,V1          ; write back to “Y[*]”
```

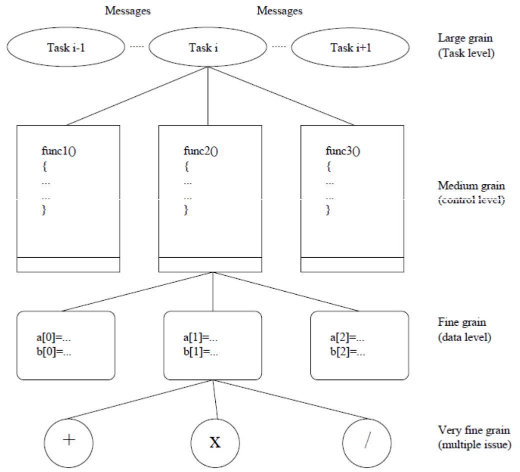
- No stall, reduce Flynn bottleneck problem.

Level of Parallelism

- Levels of parallelism are classified by grain size (or granularity)
  - Very-fine-grain (instruction-level or ILP)
  - Fine-grain (data-level)
  - Medium-grain (control-level)
  - Coarse-grain (task-level)
- Usually mean the number of instructions performed between each synchronization

---

Level of Parallelism



Parallel Programming Models

- Architecture
  - SISD - no parallelism
  - SIMD - instructional-level parallelism
  - MIMD - functional/program-level parallelism
  - SPMD - Combination of MIMD and SIMD

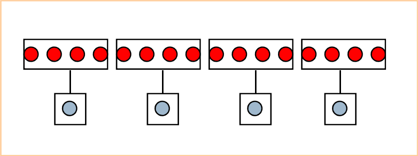
Parallel Programming Models

- Data
  - Private or shared ?
  - How to access data (shared vs. message passing)
- Operations
  - How can we handle atomic operations ?
- Cost
  - How much does it cost (for accessing data, synchronization, etc.)

Example

- Global summation

$$\sum_{k=0}^{n-1} f(A[k])$$



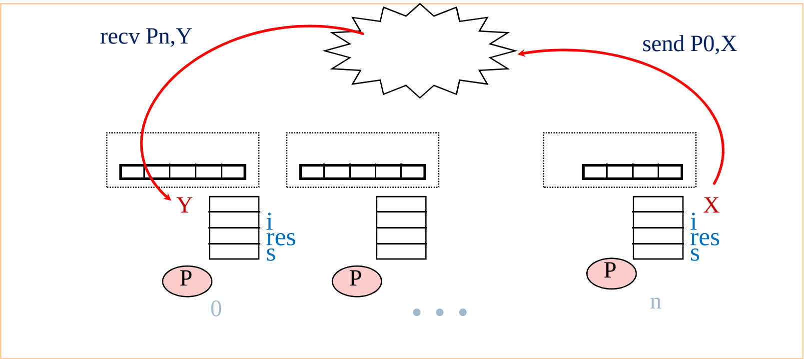
- Decomposition

$$\sum_{k=j}^{j+m-1} f(A[k])$$

- Assign n/p numbers to each of p procs
  - Each process computes f(A[k]) and performs partial sum
  - One process collects the partial sums and computes global sum

---

Model 1: Message Passing



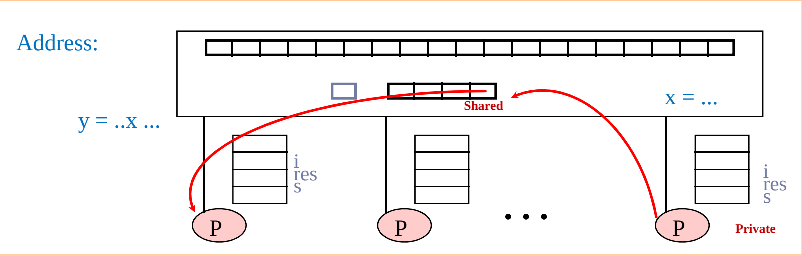
- No shared data
- Explicit data transfer (both sender and receiver must call the send/recv functions)

Global Sum in Message Passing

```
partial_sum = 0;
for each data A[k]
  partial_sum += f(A[k]);
end for

if my_id == 0 then
  for each proc j (excluding 0)
    recv(j, psum);
    global_sum += psum
  end for
else
        send(proc, partial_sum);
end if
```

Model 2: Shared Memory



- Private & shared variables
- Communicate & synchronize via shared variables (semaphore, locks)
- Similar to multi-thread programming

Global Sum in Shared Memory

Thread 1

[s = 0 initially]
local_s1= 0
for i = 0, n/2-1
    local_s1 = local_s1 + f(A[i])
s = s + local_s1

Thread 2

[s = 0 initially]
local_s2 = 0
for i = n/2, n-1
    local_s2= local_s2 + f(A[i])
s = s +local_s2

RACE CONDITION!

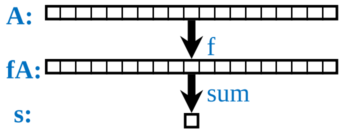
What could go wrong?

Solution? Mutual exclusion with locks

Model 3: Data Parallel

- SIMD style
  - Single instruction for all data
  - Shift data around
  - Pro: easy to understand
  - Con: inapplicable with irregular problem

A = array of all data
fA = f(A)
s = sum(fA)



Message Passing vs. Shared Memory

- Message passing
  - Data distribution among local address spaces needed
  - No explicit shared structures
  - Communication is explicit
  - Synchronization implicit in communication
- Shared Memory
  - Private and shared data
  - Synchronization done by using shared variables