2110412 Parallel Comp Arch
Parallel Programming Paradigm

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

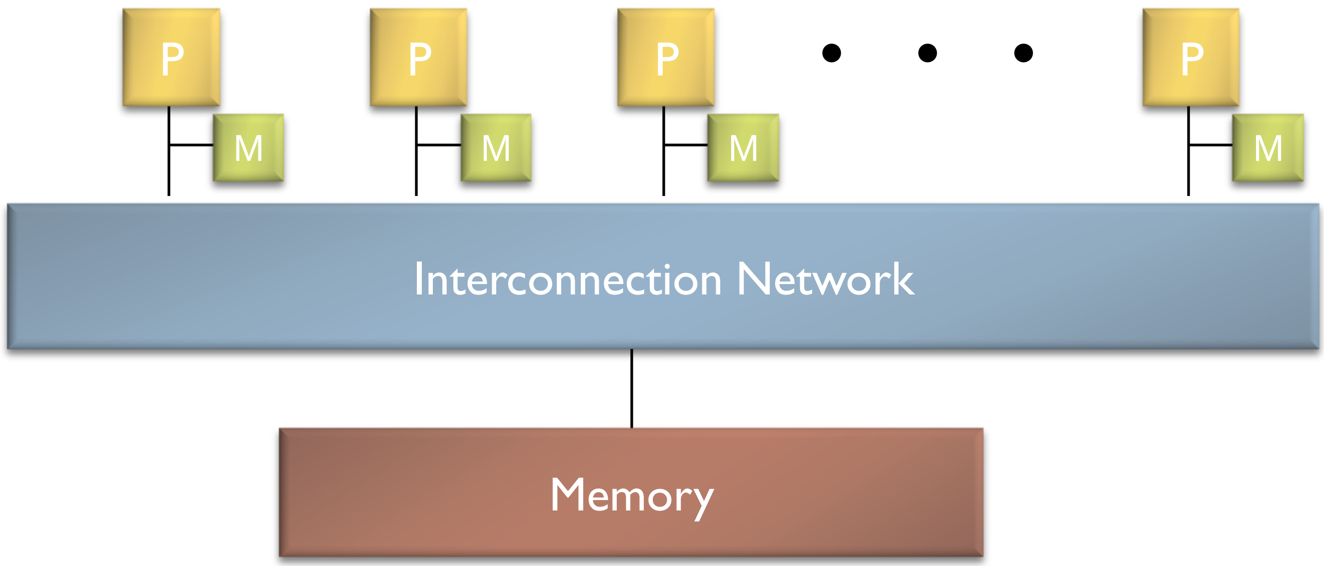
Outline

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

What are the factors for parallel programming paradigm?

- System Architecture
- Parallelism – Nature of Applications
- Development Paradigms
  - Automatic (by Compiler or by Library) : OpenMP
  - Semi-Auto (Directives / Hints) : CUDA
  - Manual : MPI, Multi-Thread Programming

Generic Parallel Architecture



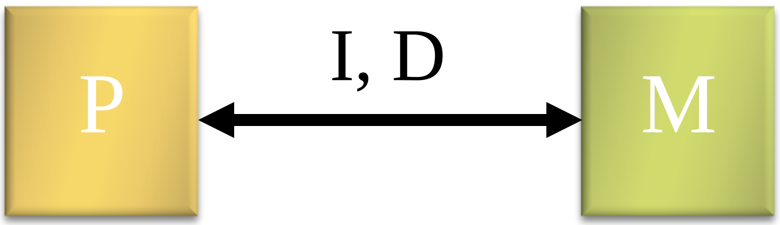
- Where is the memory physically located ?

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** (Multiple Instruction, Single Data).
  - **MIMD** (Multiple Instruction, Multiple Data).

SISD

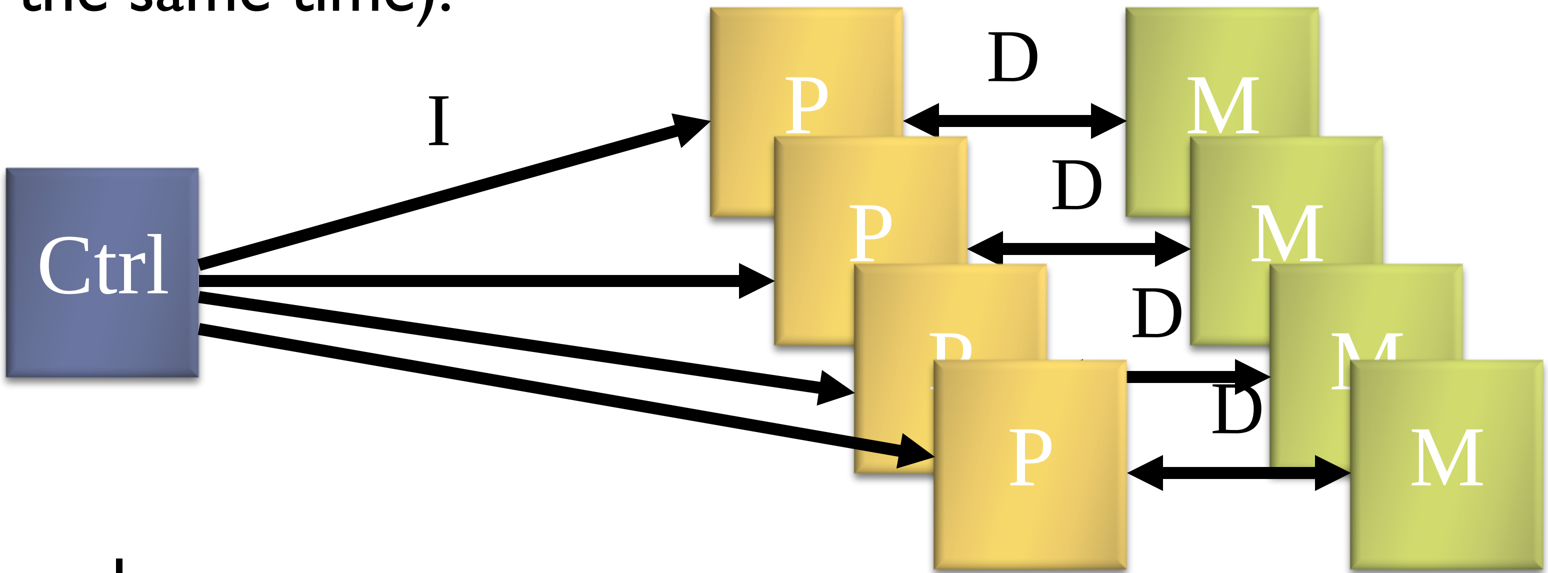
- One instruction stream and one data stream - from memory to processor.



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

SIMD

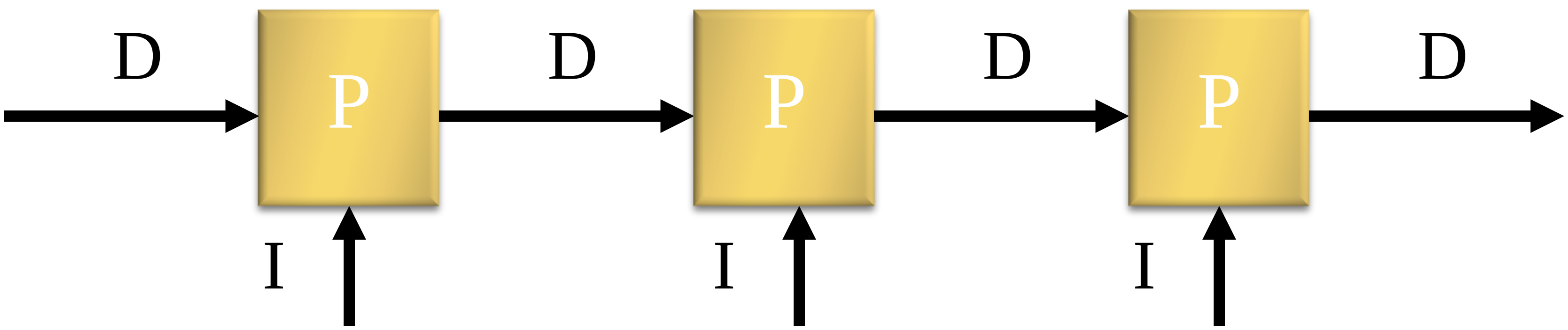
- One control unit tells processing elements to compute (at the same time).



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

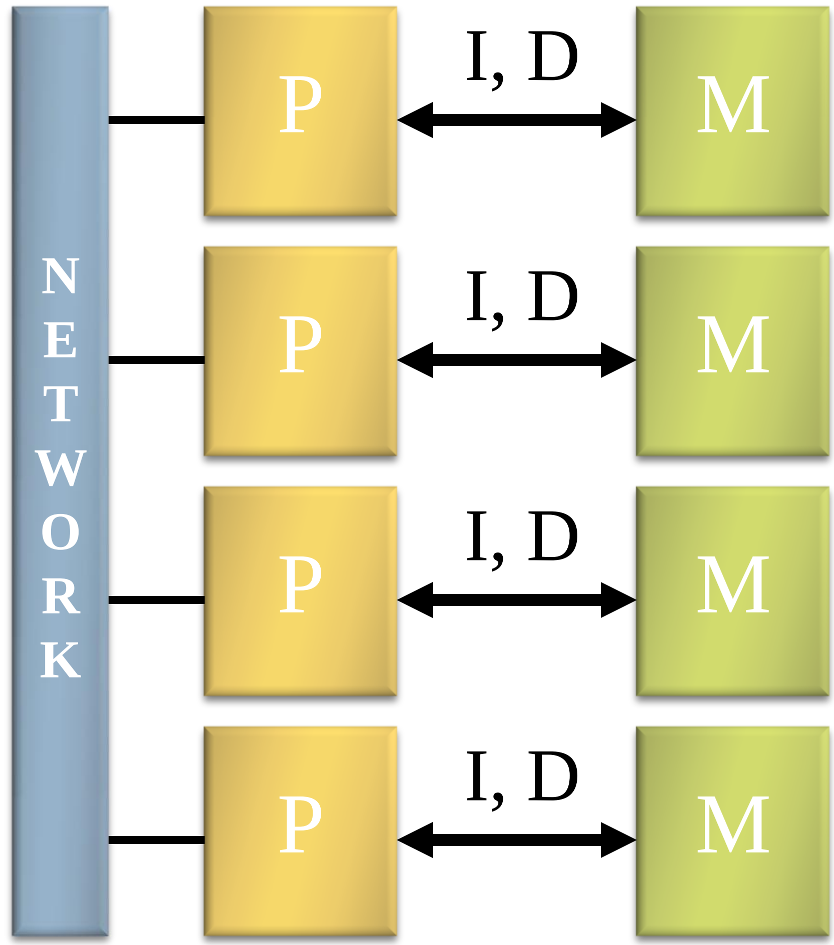
MISD

- 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

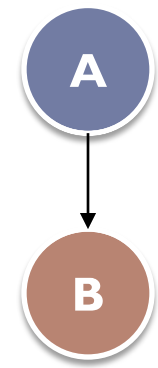


Parallelism

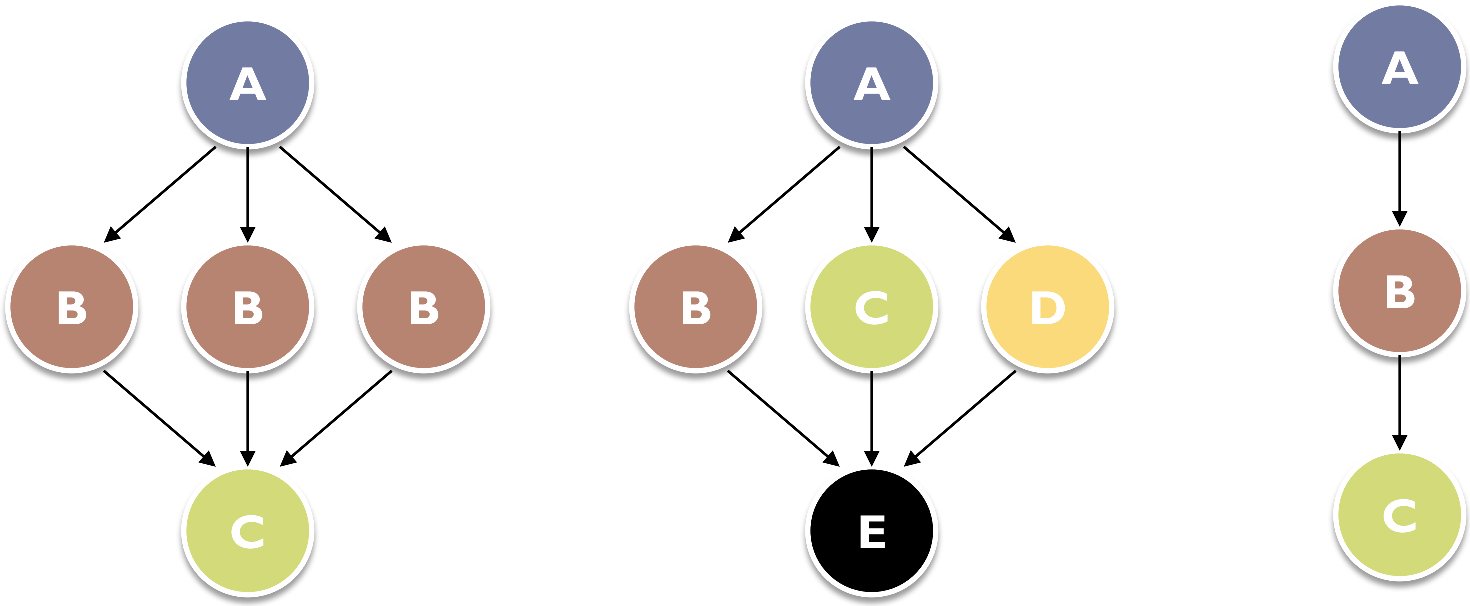
- To understand parallel system, we need to understand how can we utilize parallelism
- There are 3 types of parallelism
  - Data parallelism
  - Functional parallelism
  - Pipelining
- Can be described with data dependency graph

Data Dependency 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

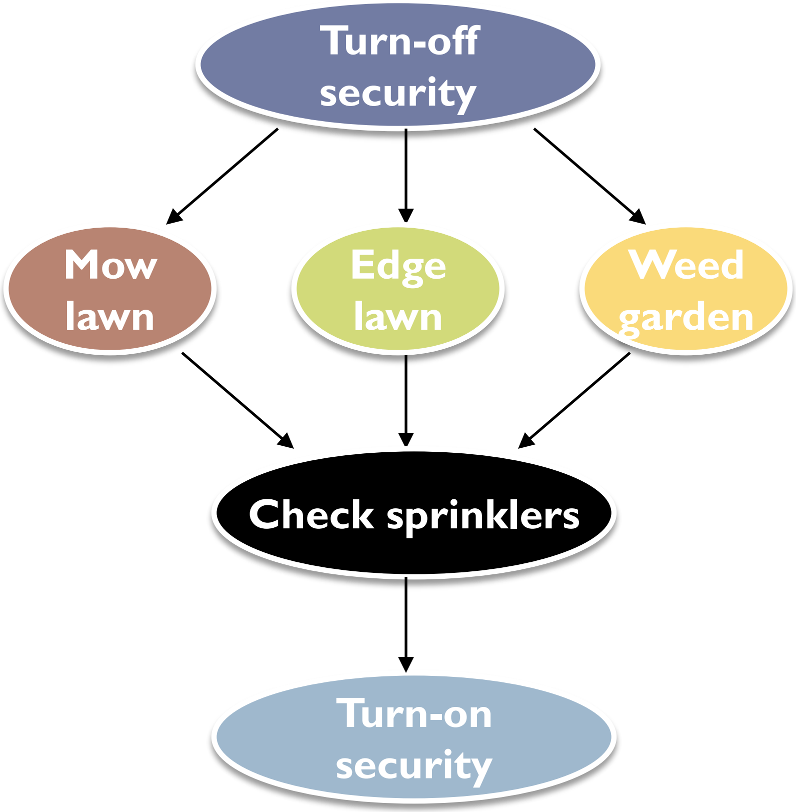


Data Parallelism **Functional Parallelism** **Pipelining**

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

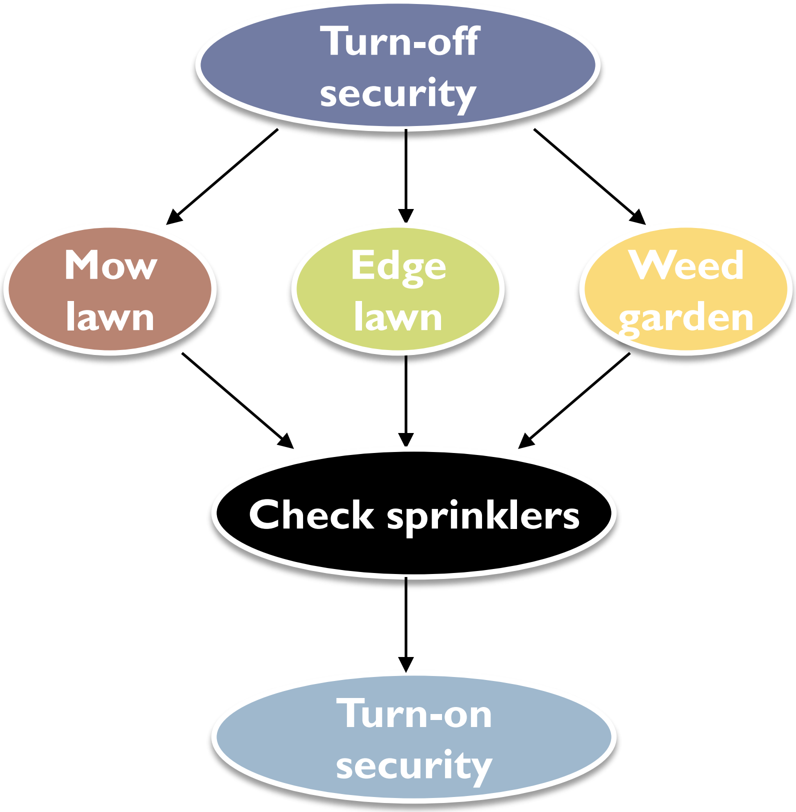
Example: Dependency Graph



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

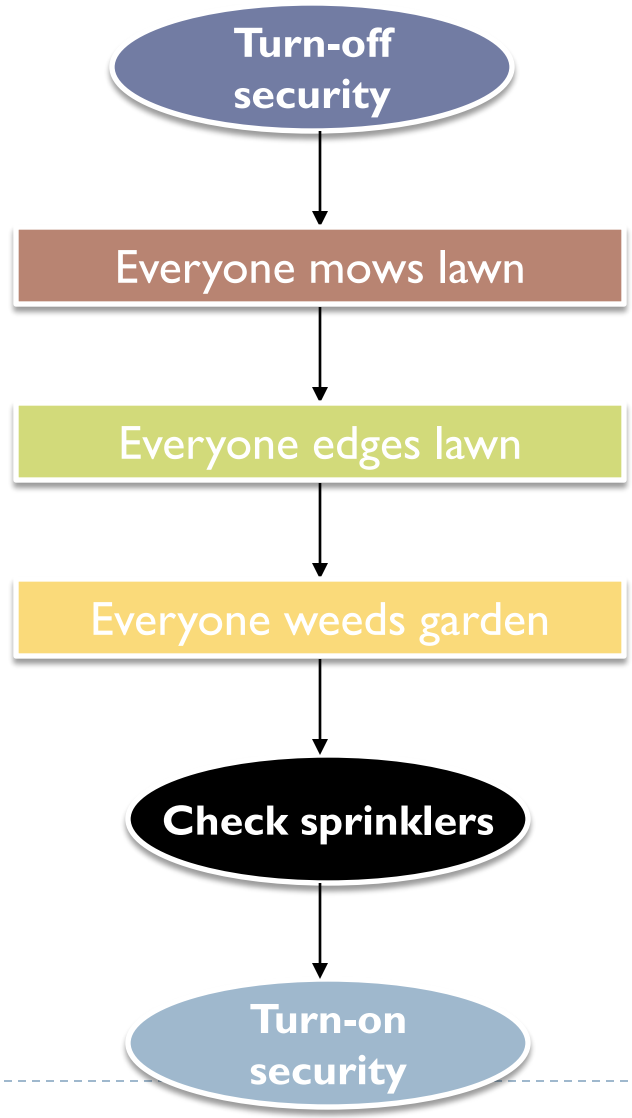
Functional Parallelism

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



Data Parallelism

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



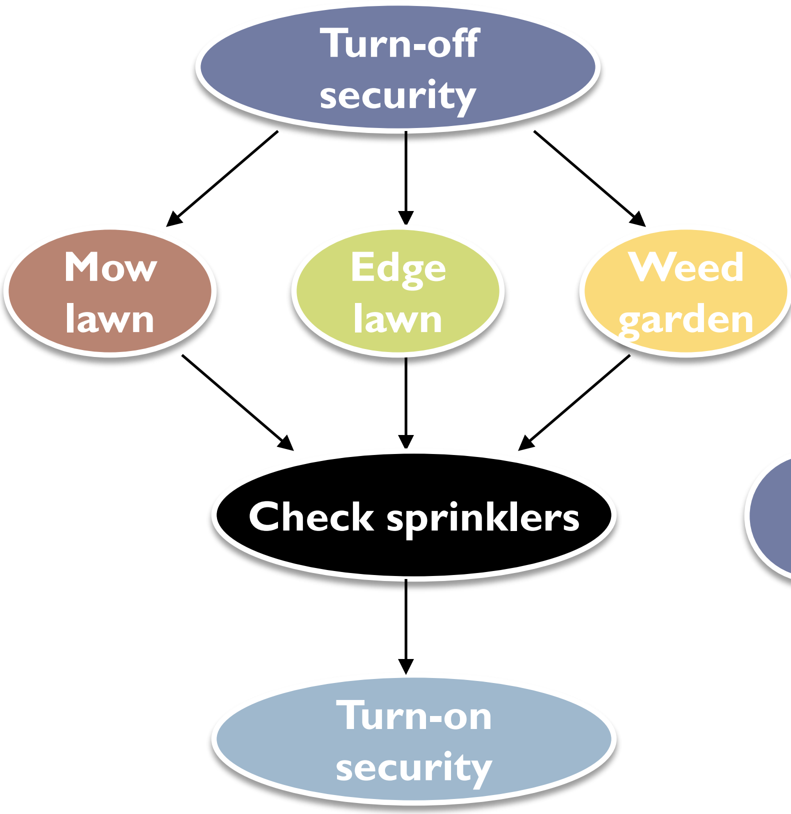
Sample Algorithm

```
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
```

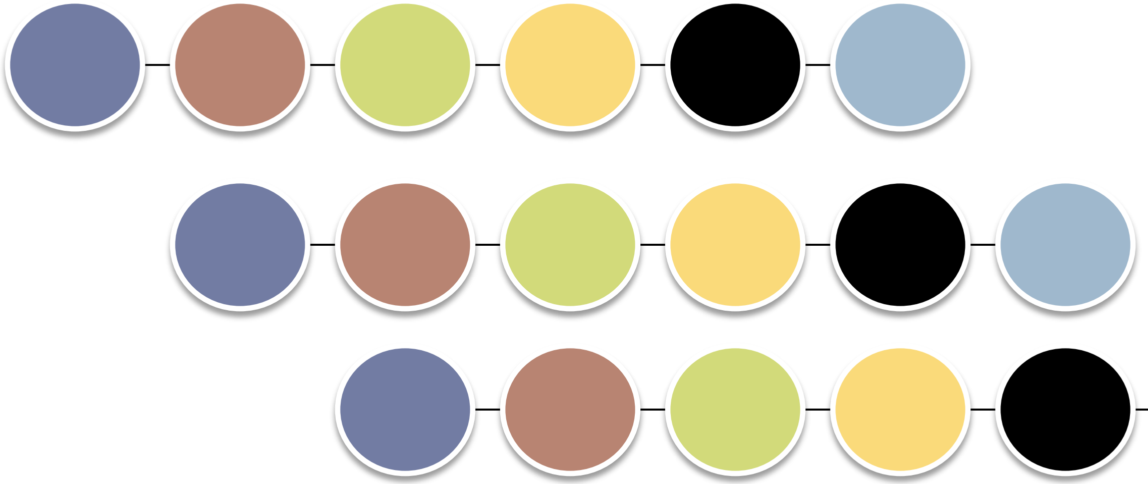
Pipelining

- 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 does not work for single data element !!!
- Pipelining is best for
  - Limited functional units
  - Each data unit cannot be partitioned

Example: Pipelining and Landscape Maintenance

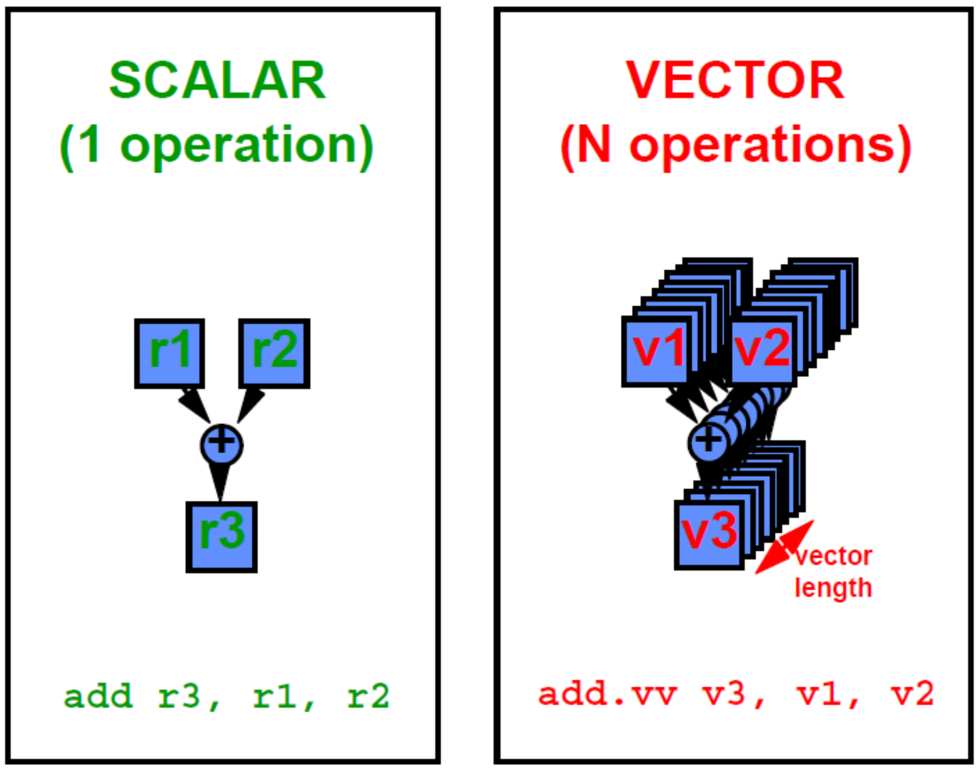


- Does not work for a single house
- Multiple houses are not good either!



Vector Processing

- Data parallelism technique
  - Perform the same function on multiple data elements (aka. "vector")
  - Many scientific applications are matrix-oriented



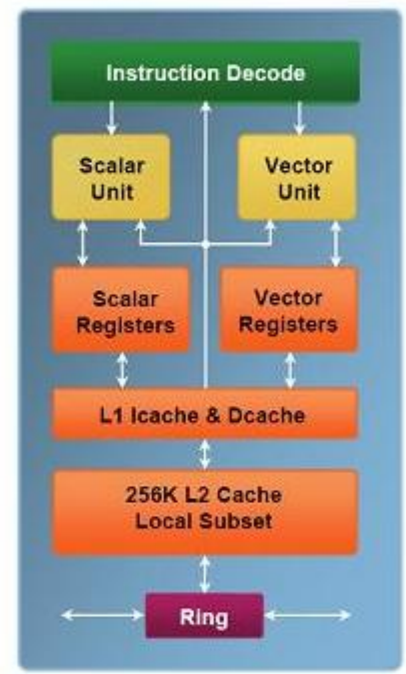
Example: SAXPY (DAXPY) problem

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

Y(0:63) = a*X(0:63) + Y(0:63)

LV V1,R1             ; R1 contains based address for “X[*]”
LV V2,R2             ; R2 contains based address for “Y[*]”
MULSV 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
- Vector Processors may also be pipelined

Vector Processing

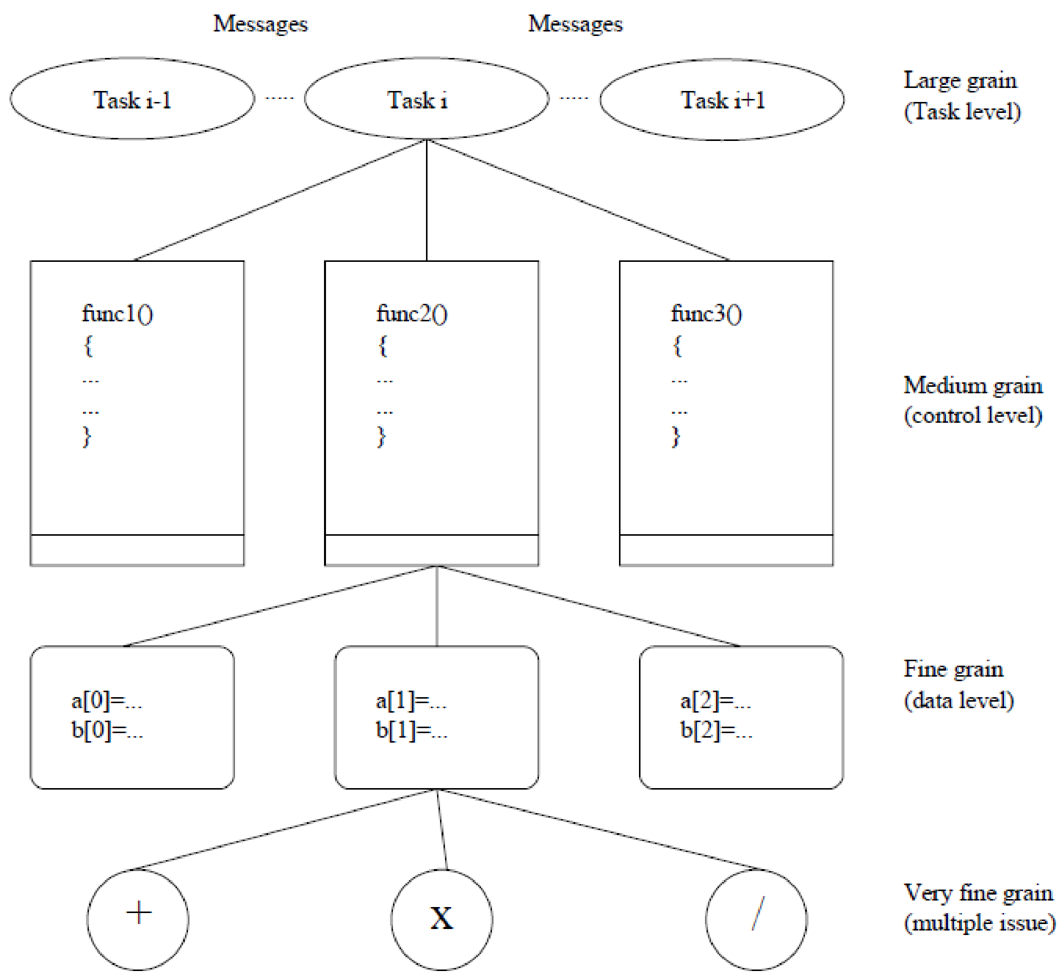
- Problems that can be efficiently formulated in terms of vectors
  - Long-range weather forecasting
  - Petroleum explorations
  - Medical diagnosis
  - Aerodynamics and space flight simulations
  - Artificial intelligence and expert systems
  - Mapping the human genome
  - Image processing
- Very famous in the past e.g. Cray Y-MP
- Not obsolete yet!
  - IBM Cell Processor
  - Intel Larrabee GPU



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



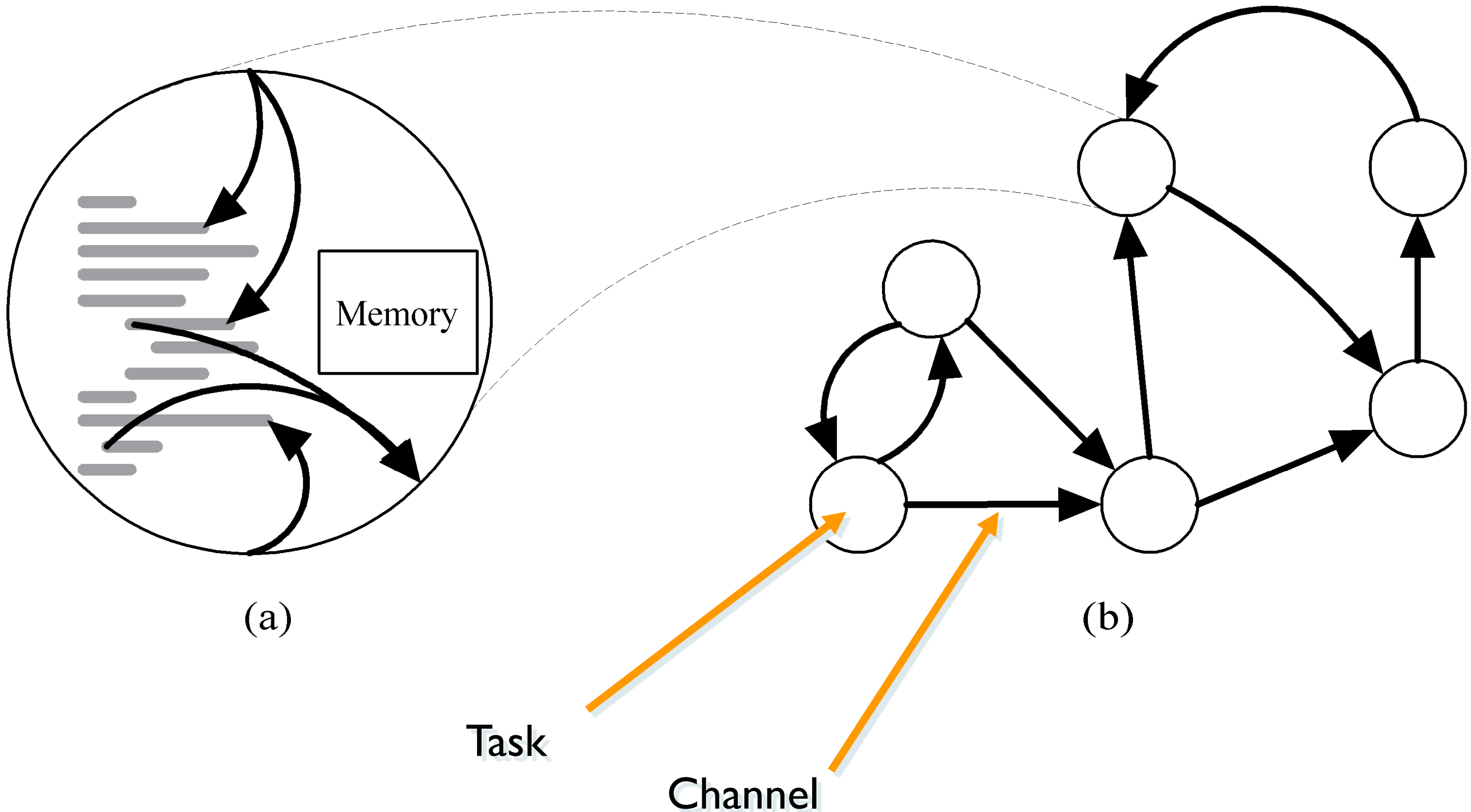
Parallel Programming Models

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

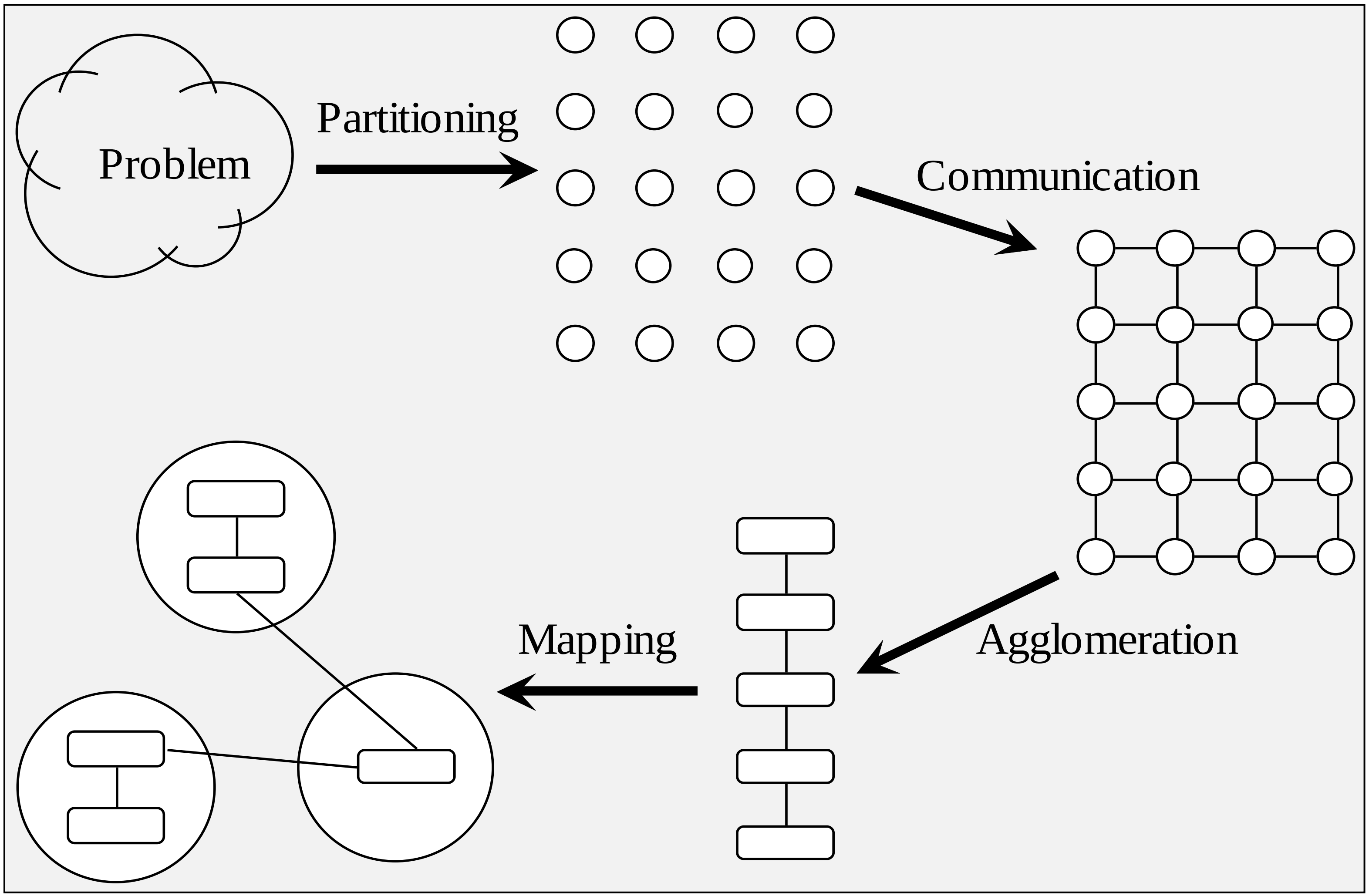
Parallel Algorithm Design

- Parallel computation = set of tasks
- Task - A program unit with its local memory and a collection of I/O ports
  - local memory contains program instructions and data
  - send local data values to other tasks via output ports
  - receive data values from other tasks via input ports
  - Tasks interact by sending messages through channels
- Channel: - A message queue that connects one task's output port with another task's input port
  - sender is never blocked
  - receiver is blocked if the data value is not yet sent

Task/Channel Model



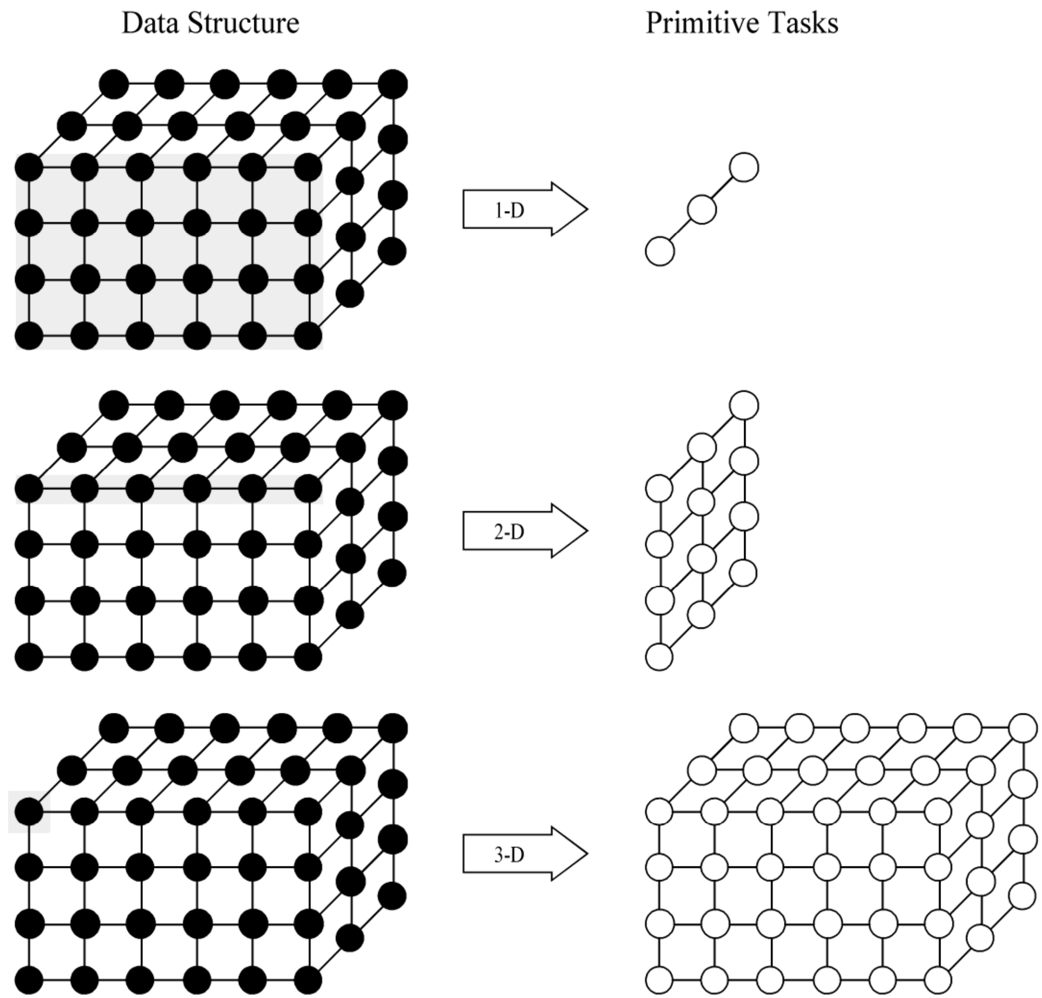
Foster's Methodology



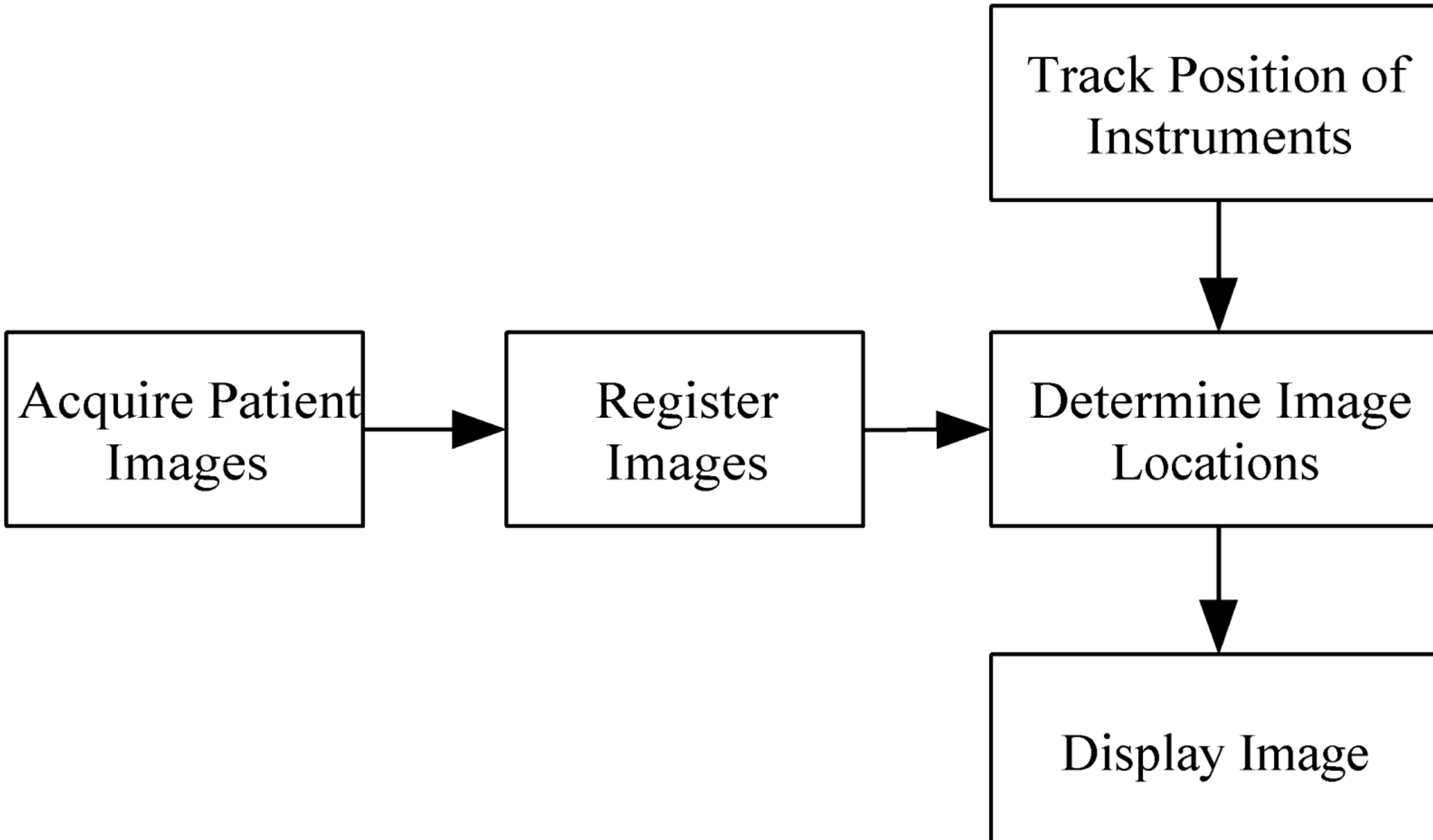
Partitioning

- To discover as much parallelism as possible
- Dividing computation and data into pieces
- Domain decomposition (Data-Centric Approach)
  - Divide data into pieces
  - Determine how to associate computations with the data
- Functional decomposition (Computational-Centric)
  - Divide computation into pieces
  - Determine how to associate data with the computations
  - Most of the time = Pipelining

Example Domain Decompositions



Example Functional Decomposition



Partitioning Checklist

- At least 10x more primitive tasks than processors in target computer
- Minimize redundant computations and redundant data storage
- Primitive tasks roughly the same size
- Number of tasks an increasing function of problem size

Communication

- Local communication
  - Task needs values from a small number of other tasks
- Global communication
  - Significant number of tasks contribute data to perform a computation

Communication Checklist

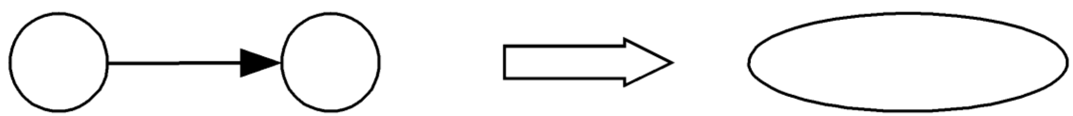
- Communication operations balanced among tasks
- Each task communicates with only small group of neighbors
- Tasks can perform communications concurrently
- Task can perform computations concurrently

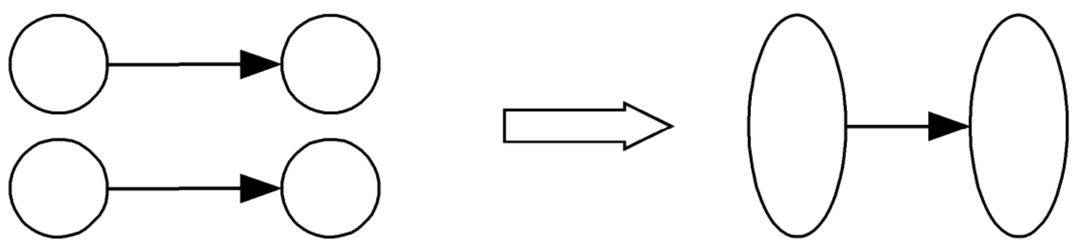
Agglomeration

- After 2 steps, our design still cannot execute efficiently on a real parallel computer
- Grouping tasks into larger tasks to reduce overheads
- Goals
  - Improve performance
  - Maintain scalability of program
  - Simplify programming
- In MPI programming, goal often to create one agglomerated task per processor

Agglomeration Can Improve Performance

- Eliminate communication between primitive tasks agglomerated into consolidated task
- Combine groups of sending and receiving tasks


(a)


(b)

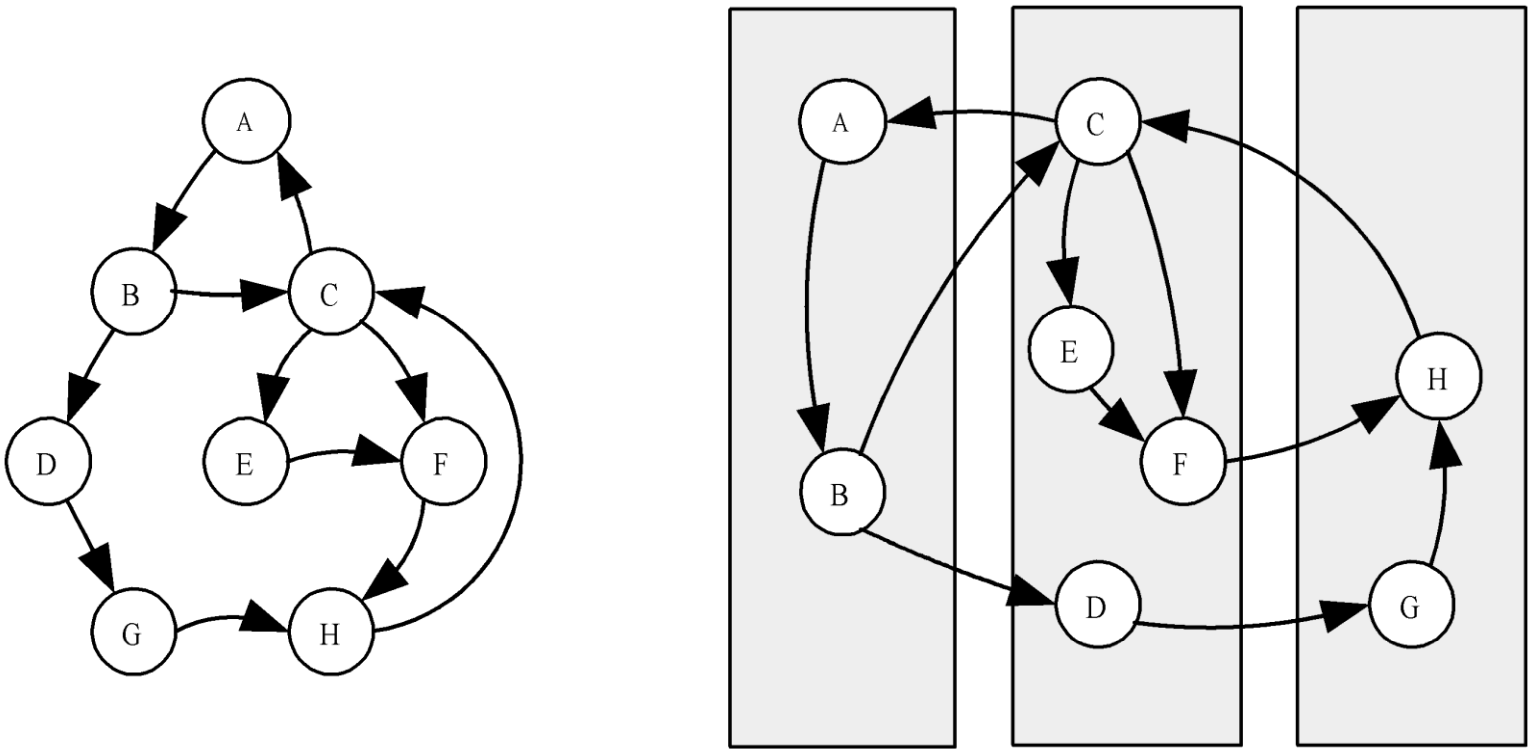
Agglomeration Checklist

- Locality of parallel algorithm has increased
- Replicated computations take less time than communications they replace
- Data replication doesn't affect scalability
- Agglomerated tasks have similar computational and communications costs
- Number of tasks increases with problem size
- Number of tasks suitable for likely target systems
- Tradeoff between agglomeration and code modifications costs is reasonable

Mapping

- Process of assigning tasks to processors
- Centralized multiprocessor: mapping done by operating system
- Distributed memory system: mapping done by user
- Conflicting goals of mapping
  - Maximize processor utilization
  - Minimize interprocessor communication

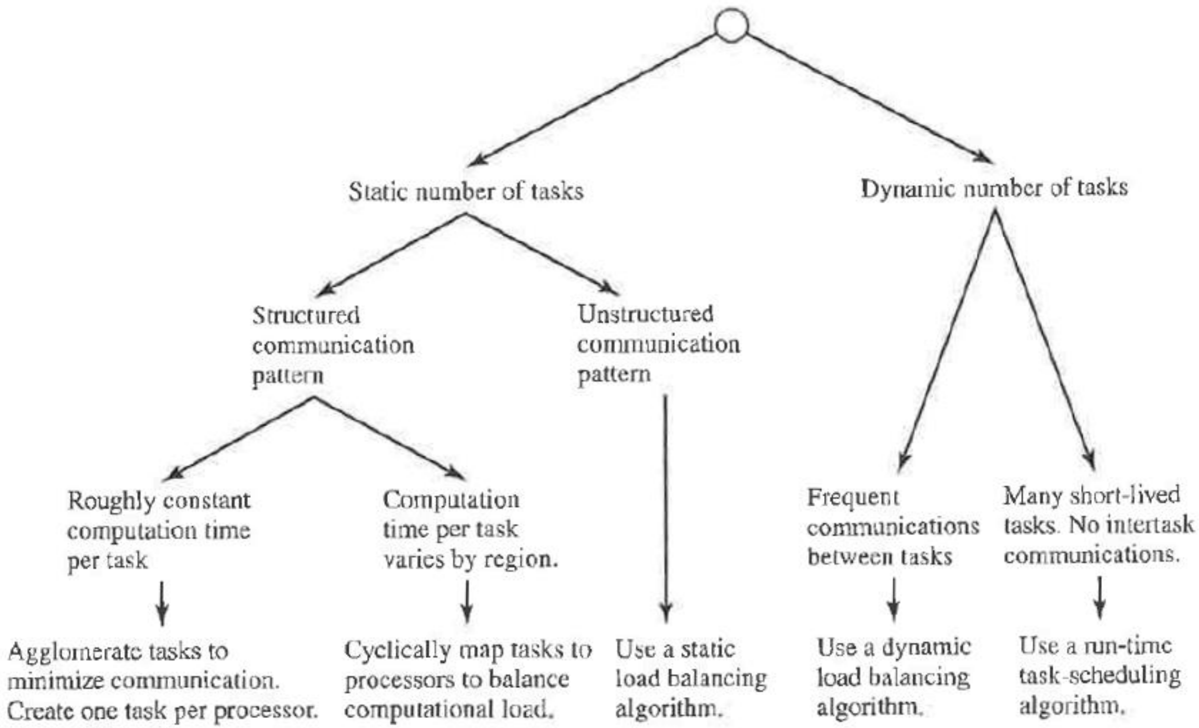
Mapping Example



(a) (b)

Optimal Mapping

- Finding optimal mapping is NP-hard
- Must rely on heuristics



Mapping Decision Tree

- Static number of tasks
  - Structured communication
    - Constant computation time per task
      - Agglomerate tasks to minimize comm
      - Create one task per processor
    - Variable computation time per task
      - Cyclically map tasks to processors
  - Unstructured communication
      - Use a static load balancing algorithm
- Dynamic number of tasks

Mapping Strategy

- Static number of tasks
- Dynamic number of tasks
  - Frequent communications between tasks
    - Use a dynamic load balancing algorithm
  - Many short-lived tasks
    - Use a run-time task-scheduling algorithm

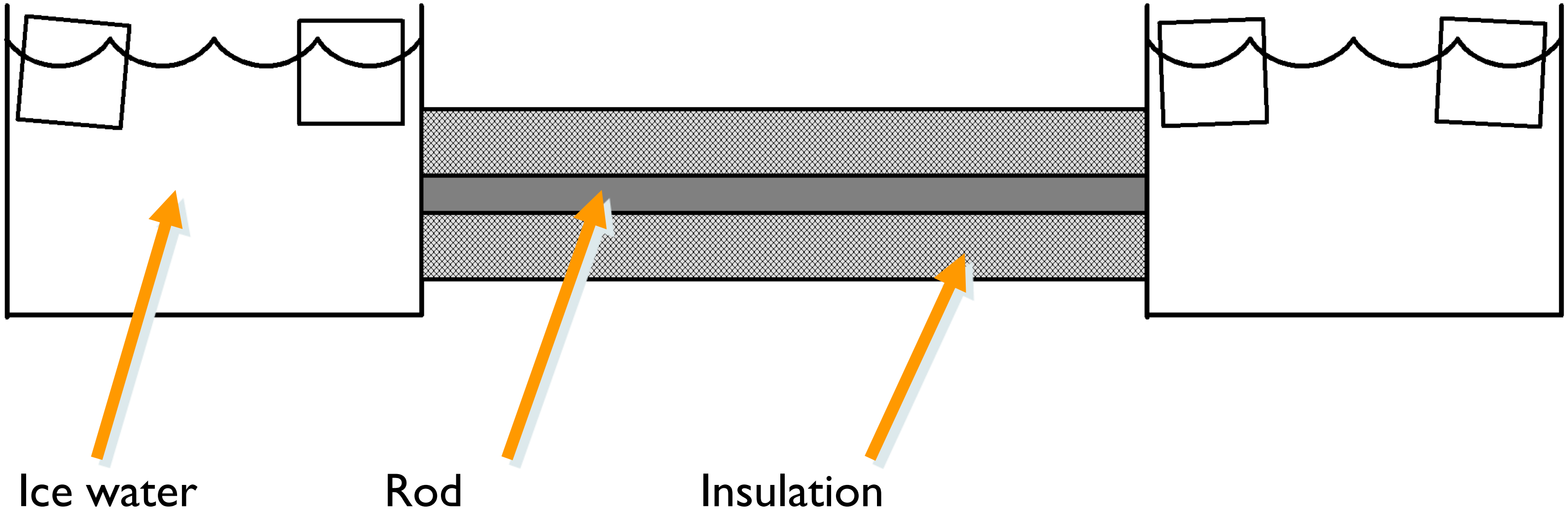
Mapping Checklist

- Considered designs based on one task per processor and multiple tasks per processor
- Evaluated static and dynamic task allocation
- If dynamic task allocation chosen, task allocator is not a bottleneck to performance
- If static task allocation chosen, ratio of tasks to processors is at least 10:1

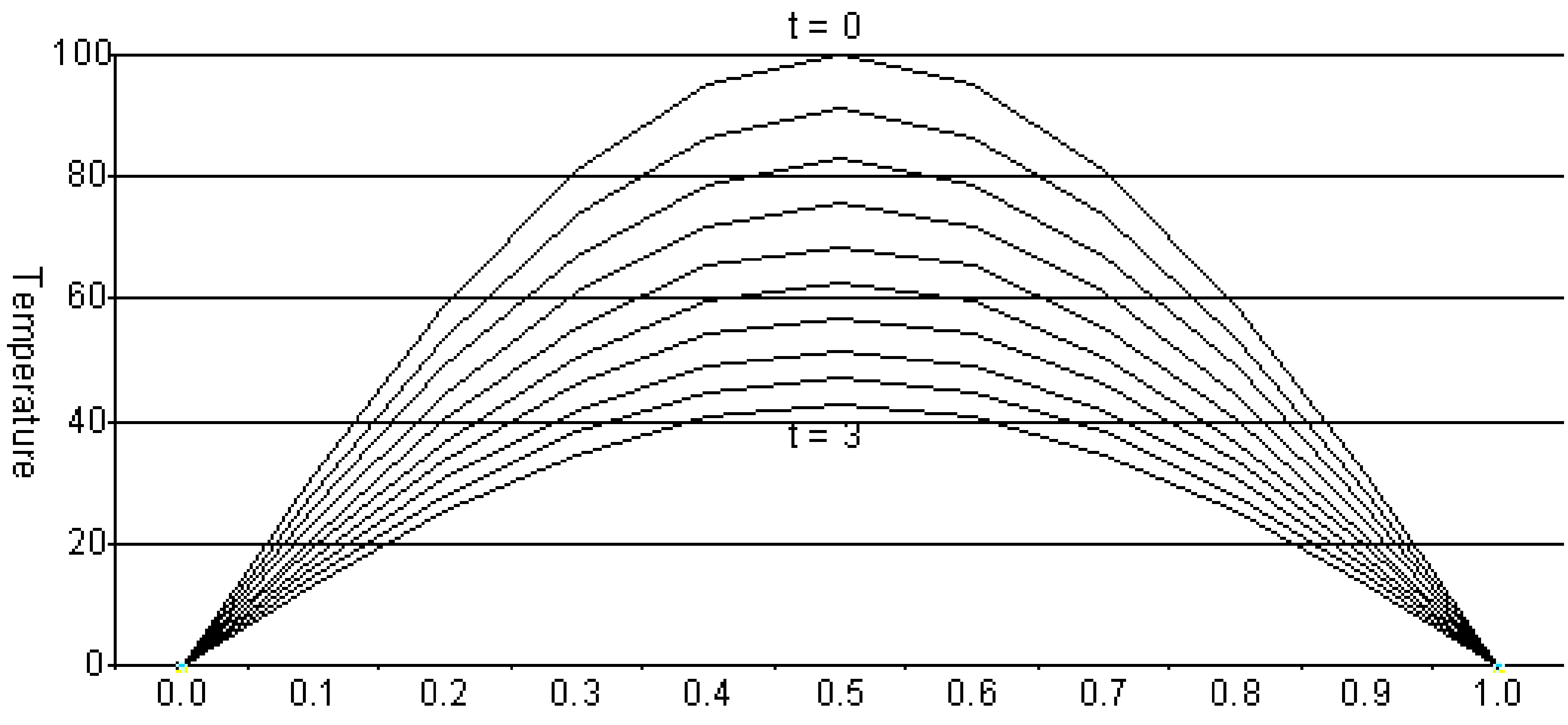
Case Studies

- Boundary value problem
- The n-body problem

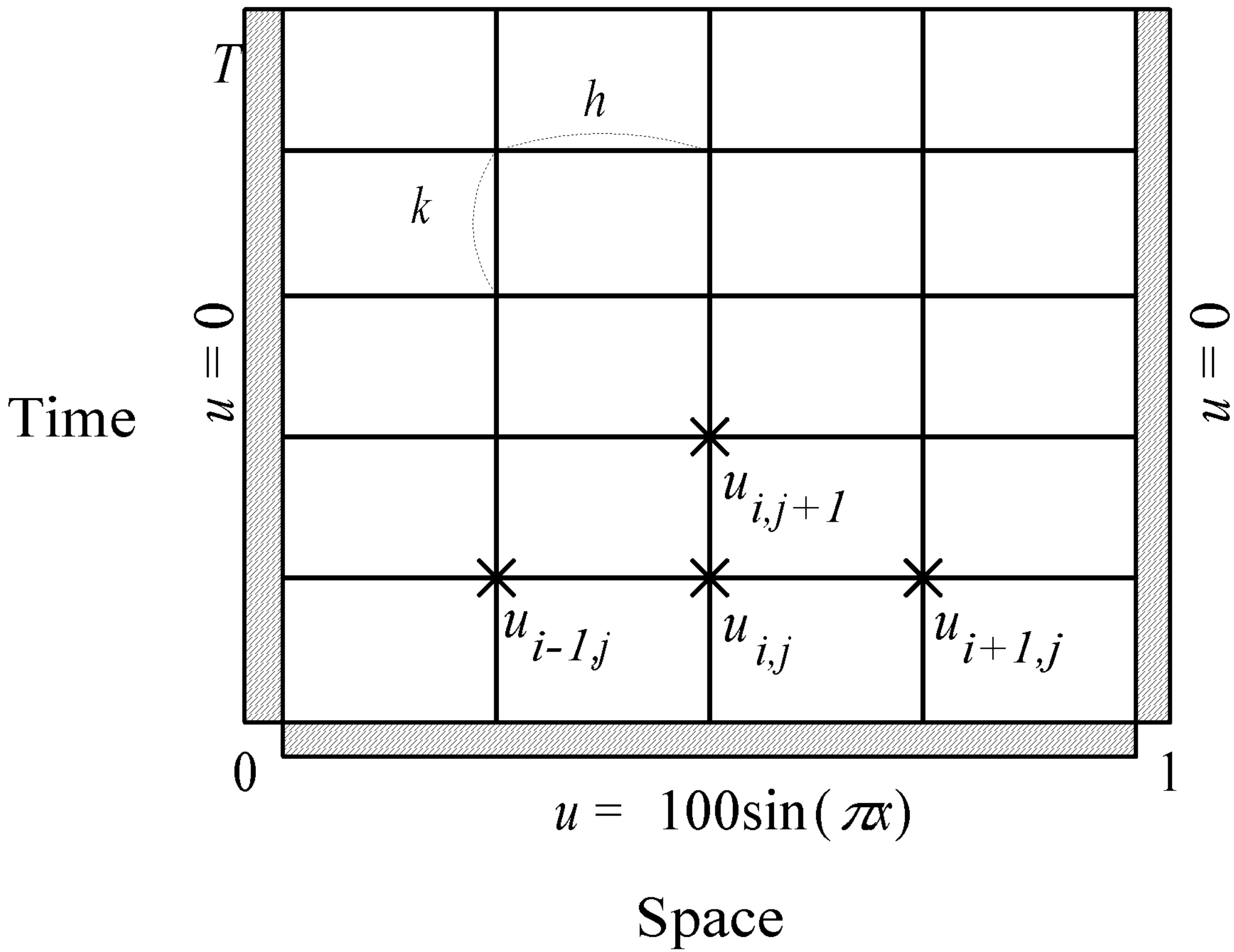
Boundary Value Problem



Rod Cools as Time Progresses



Finite Difference Approximation



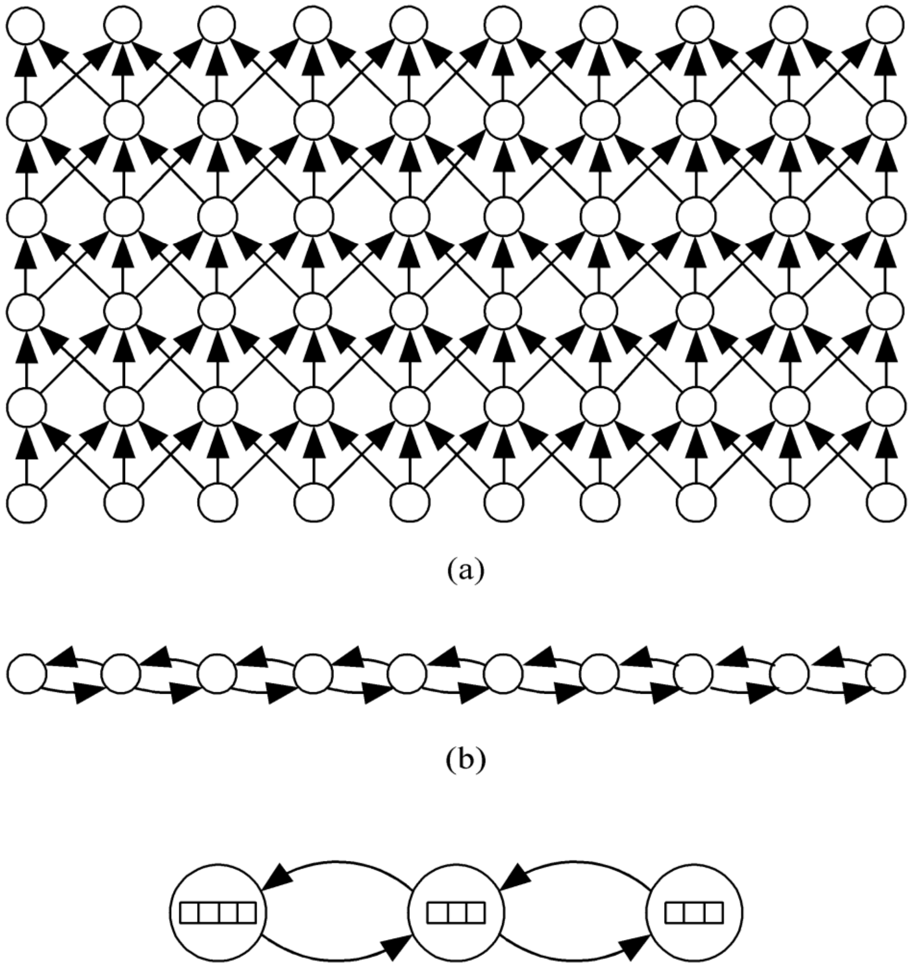
Partitioning

- One data item per grid point
- Associate one primitive task with each grid point
- Two-dimensional domain decomposition

Communication

- Identify communication pattern between primitive tasks
- Each interior primitive task has three incoming and three outgoing channels

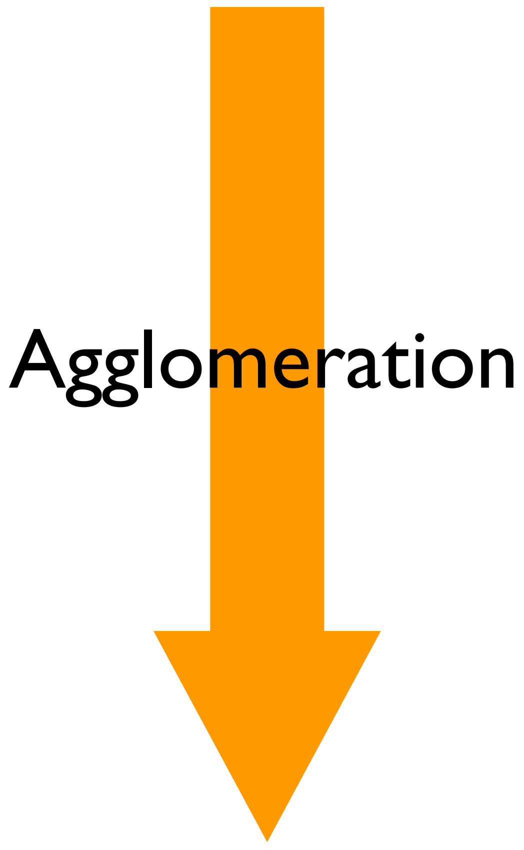
Agglomeration and Mapping



(a)

(b)

(c)



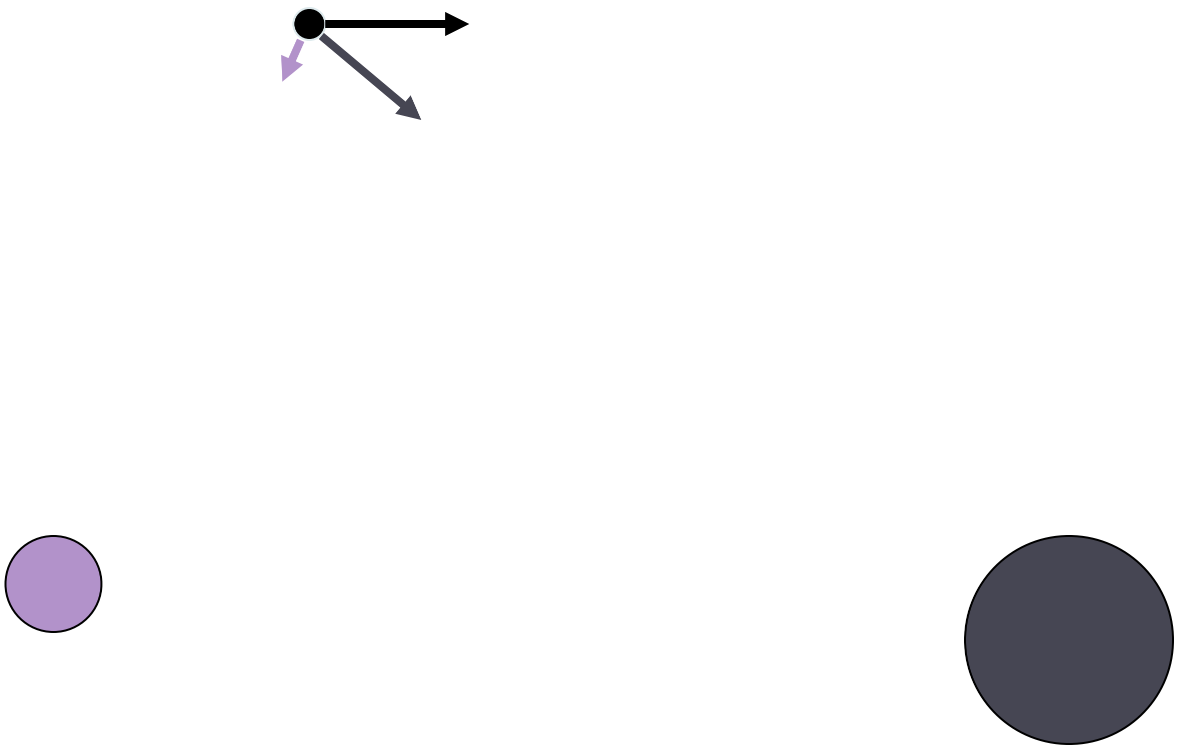
Sequential execution time

- χ – time to update element
- n – number of elements
- m – number of iterations
- Sequential execution time: $m\ (n\text{-}1)\ \chi$

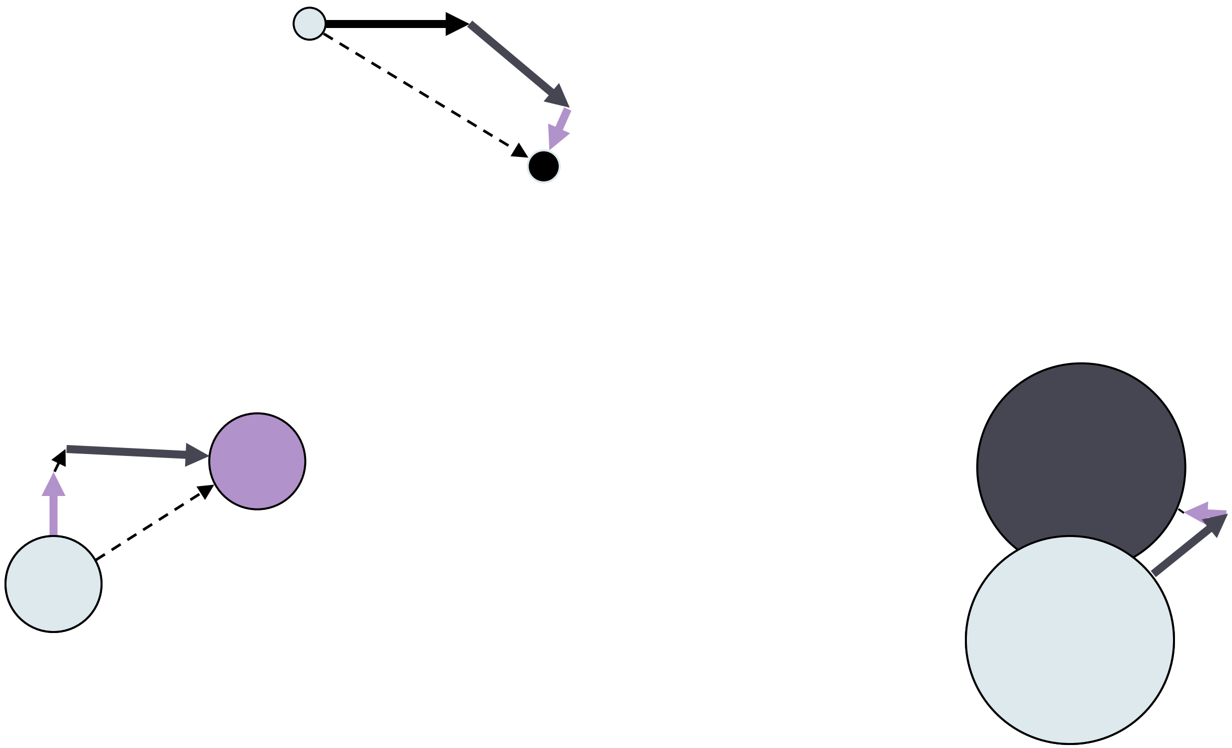
Parallel Execution Time

- p – number of processors
- λ – message latency
- Parallel execution time $m(\chi\lceil(n-1)/p\rceil+2\lambda)$

The n-body Problem



The n-body Problem



Partitioning

- Domain partitioning
- Assume one task per particle
- Task has particle's position, velocity vector
- Iteration
  - Get positions of all other particles
  - Compute new position, velocity

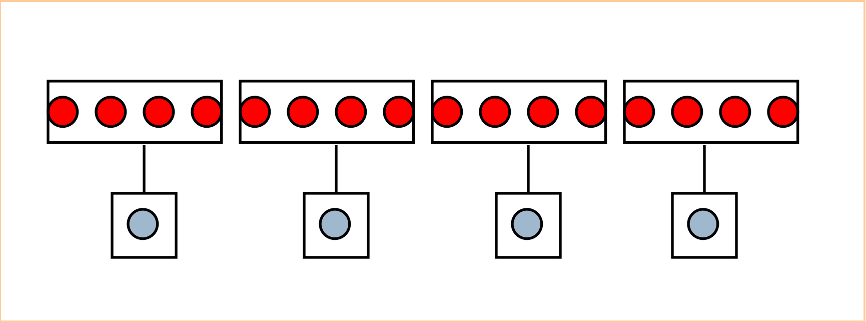
Parallel Programming Models

- 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

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

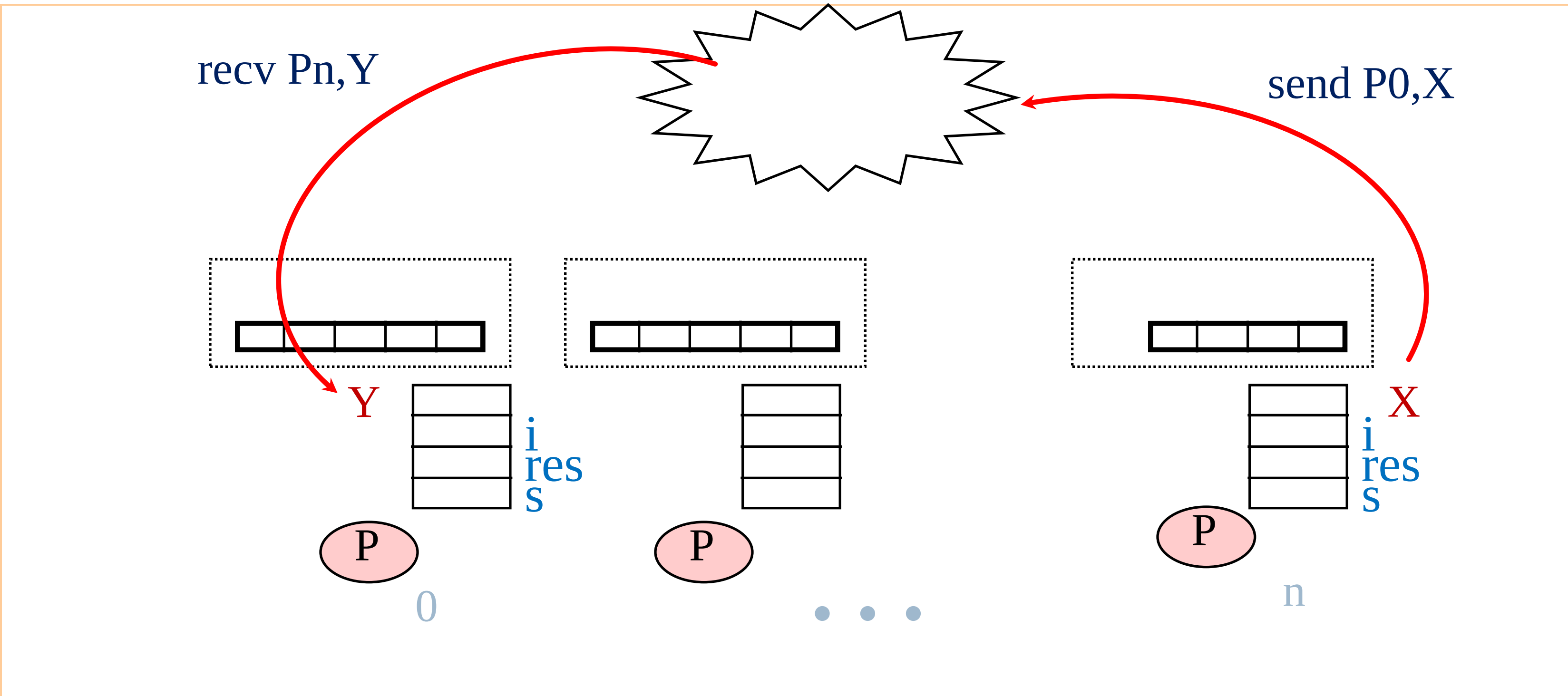


- Decomposition

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

- 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

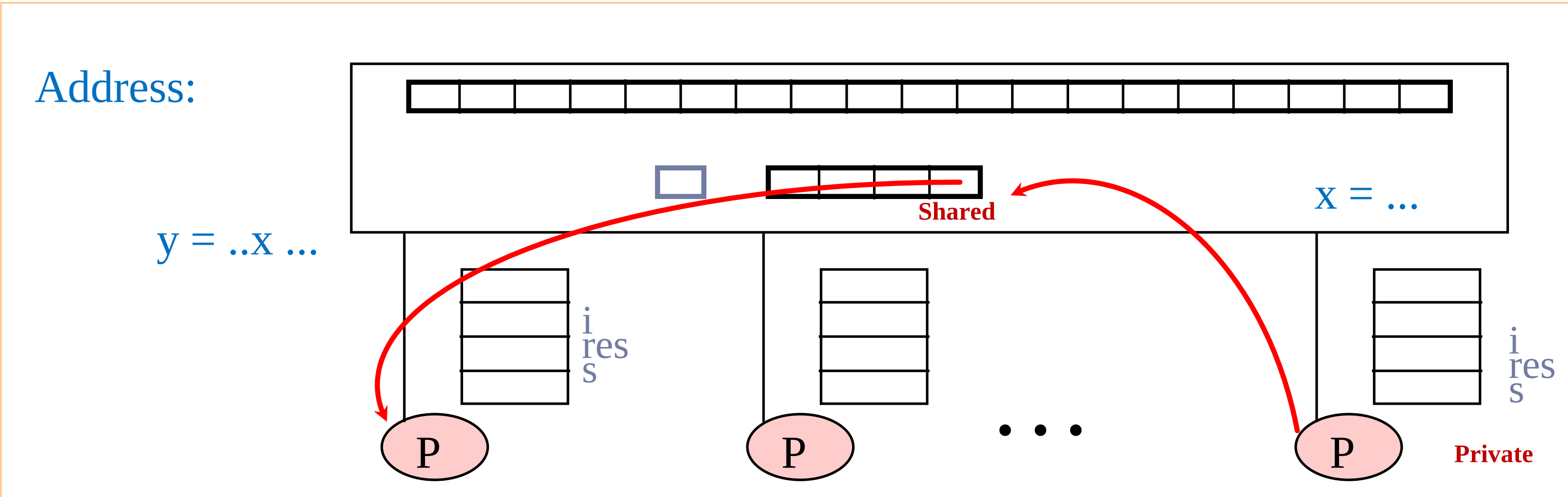


- No shared data
- Explicit data transfer (both sender and receiver must call the send/recv functions)

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!

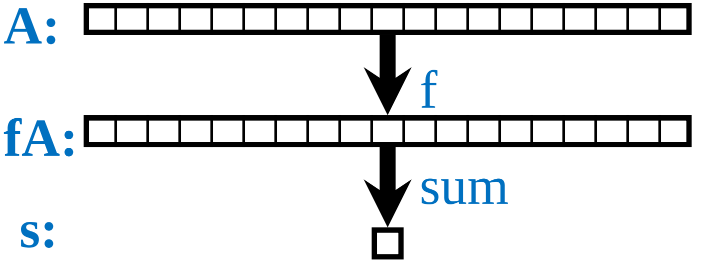
What could go wrong?

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