Parallel Programming Models – Part I

Lecture 04

Outline

- Parallelism and Types of Parallelism
- Parallel Programming Models
  - Models of Coordination
  - Program Parallelization
  - Parallel Programming Patterns
- Summary

PARALLELISM

What is Parallelism?

- **Parallelism**:
  - Average number of units of work that can be performed in parallel per unit time
  - Example: MIPS, MFLOPS, average number of threads (processes) per second
- Limits in exploiting parallelism
  - Program dependencies – data dependencies, control dependencies
  - Runtime – memory contention, communication overheads, thread/process overhead, synchronization (coordination)
- Work = tasks + dependencies

Types of Parallelism

Data parallelism

- **Partition the data** used in solving the problem among the processing units; each processing unit carries out *similar operations* on its part of the data

Task parallelism

- **Partition the tasks** in solving the problem among the processing units

Data Parallelism

- **Same operation** is applied to **different elements** of a data set
  - If operations are independent, elements can be distributed among processors for parallel execution ➔ **data parallelism**

- SIMD computers / instructions are designed to exploit data parallelism

- Example:

```
for (i = 0; i < N; i++)
    a[i] = b[i-1] + c[i]
```

Loop Parallelism – aka Data Parallelism

- Many algorithms perform computations by iteratively traversing a large data structure
  - Commonly expressed as a loop

- **If the iterations are independent**:
  - Iterations can be executed in arbitrary order and in parallel on different processors

Example: Parallel For in OpenMP

- Iterations of the for loop executed in parallel by a group threads

```
// Parallelize the matrix multiplication (result = a x b)
// Each thread will work on one iteration of the outer-most loop
// Variables are shared among threads (a, b, result)
// and each thread has its own private copy (i, j, k)


#pragma omp parallel for num_threads(8)
        shared(a, b, result) private (i, j, k)

for (i = 0; i < size; i++)
    for (j = 0; j < size; j++)
        for (k = 0; k < size; k++)
            result.element[i][j] += a.element[i][k] *
                                    b.element[k][j];
```

Data Parallelism on MIMD

- Common model: **SPMD** (Single Program Multiple Data)
  - One parallel program is executed by all processors in parallel (both shared and distributed address space)
- Example: Scalar product of **x·y** on p processors

```
local_size = size/p;
local_lower = me * local_size;
local_upper = (me+1) * local_size - 1;
local_sum = 0.0;

for (i=local_lower; i<=local_upper; i++)
  local_sum += x[i] * y[i];

Reduce(&local_sum, &global_sum, 0, SUM);
```

Same program executed by **p** processor.

"me" is the processor index (0 to p-1)

Task (Functional) Parallelism

- Independent program parts (**tasks**) can be executed in parallel
  - task (functional) parallelism

- Tasks: single statement, series of statements, loops or function calls

- Further decomposition:
  - A single task can be executed sequentially by one processor, or in parallel by multiple processors

Example: Task Parallelism

- Consider the database query:

Model ="civic" **AND** Year = "2001" **AND** (Color = "green" **OR** Color = "white")

ID#	Model	Year	Color	Dealer	Price
4523	Civic	2002	Blue	MN	$18,000
3476	Corolla	1999	White	IL	$15,000
7623	Camry	2001	Green	NY	$21,000
9834	Prius	2001	Green	CA	$18,000
6734	Civic	2001	White	OR	$17,000
5342	Altima	2001	Green	FL	$19,000
3845	Maxima	2001	Blue	NY	$22,000
8354	Accord	2000	Green	VT	$18,000
4395	Civic	2001	Red	CA	$17,000
7352	Civic	2002	Red	WA	$18,000

Taken from "Introduction to Parallel Computing" [1]

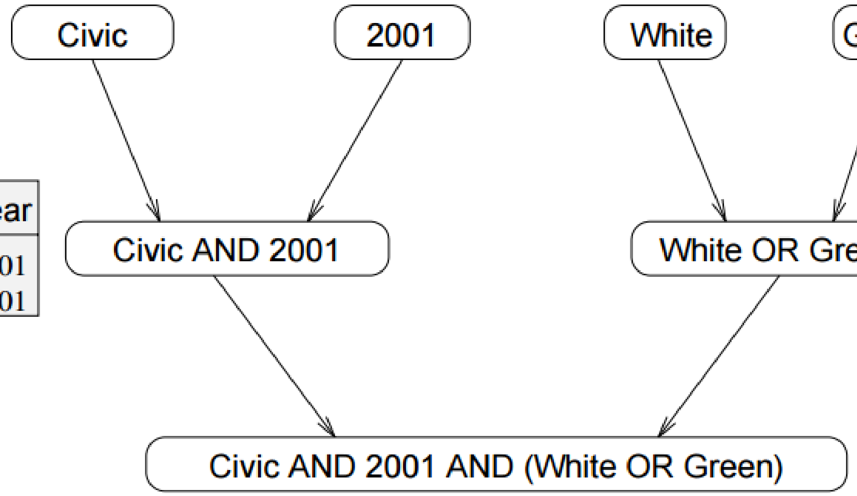
Example: **Decomposition A**

ID#	Model
4523	Civic
6734	Civic
4395	Civic
7352	Civic

ID#	Year
7623	2001
6734	2001
5342	2001
3845	2001
4395	2001

ID#	Color
3476	White
6734	White

ID#	Color
7623	Green
9834	Green
5342	Green
8354	Green



ID#	Model	Year
6734	Civic	2001
4395	Civic	2001

ID#	Color
3476	White
7623	Green
9834	Green
6734	White
5342	Green
8354	Green

ID#	Model	Year	Color
6734	Civic	2001	White

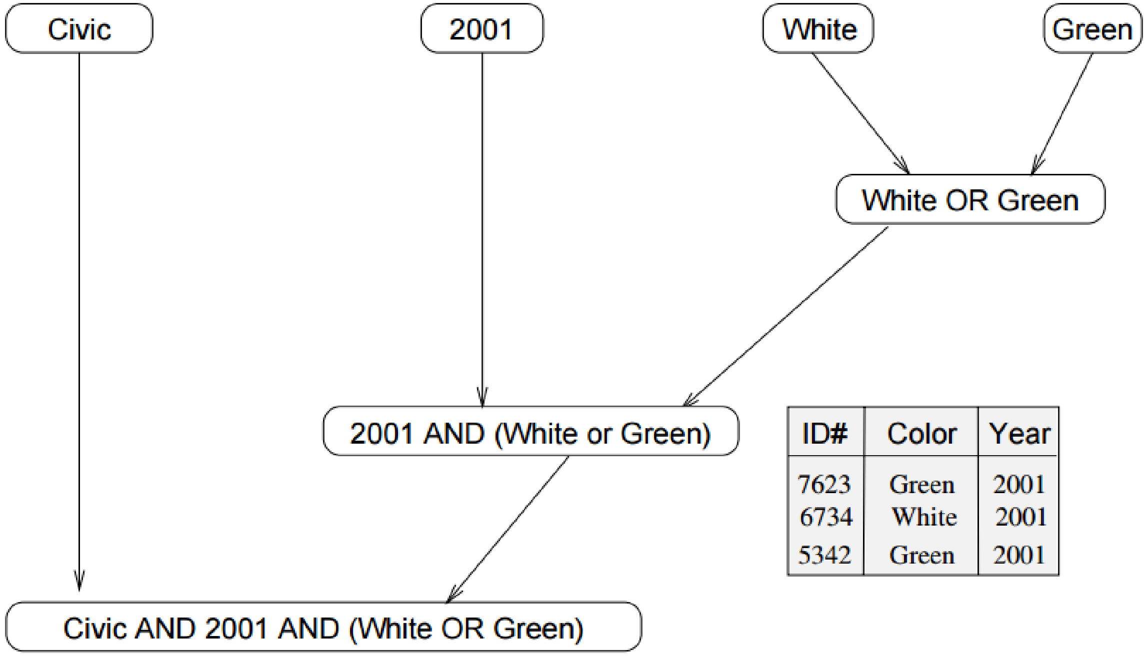
Example: **Decomposition B**

ID#	Model
4523	Civic
6734	Civic
4395	Civic
7352	Civic

ID#	Year
7623	2001
6734	2001
5342	2001
3845	2001
4395	2001

ID#	Color
3476	White
6734	White

ID#	Color
7623	Green
9834	Green
5342	Green
8354	Green



ID#	Color
3476	White
7623	Green
9834	Green
6734	White
5342	Green
8354	Green

ID#	Color	Year
7623	Green	2001
6734	White	2001
5342	Green	2001

ID#	Model	Year	Color
6734	Civic	2001	White

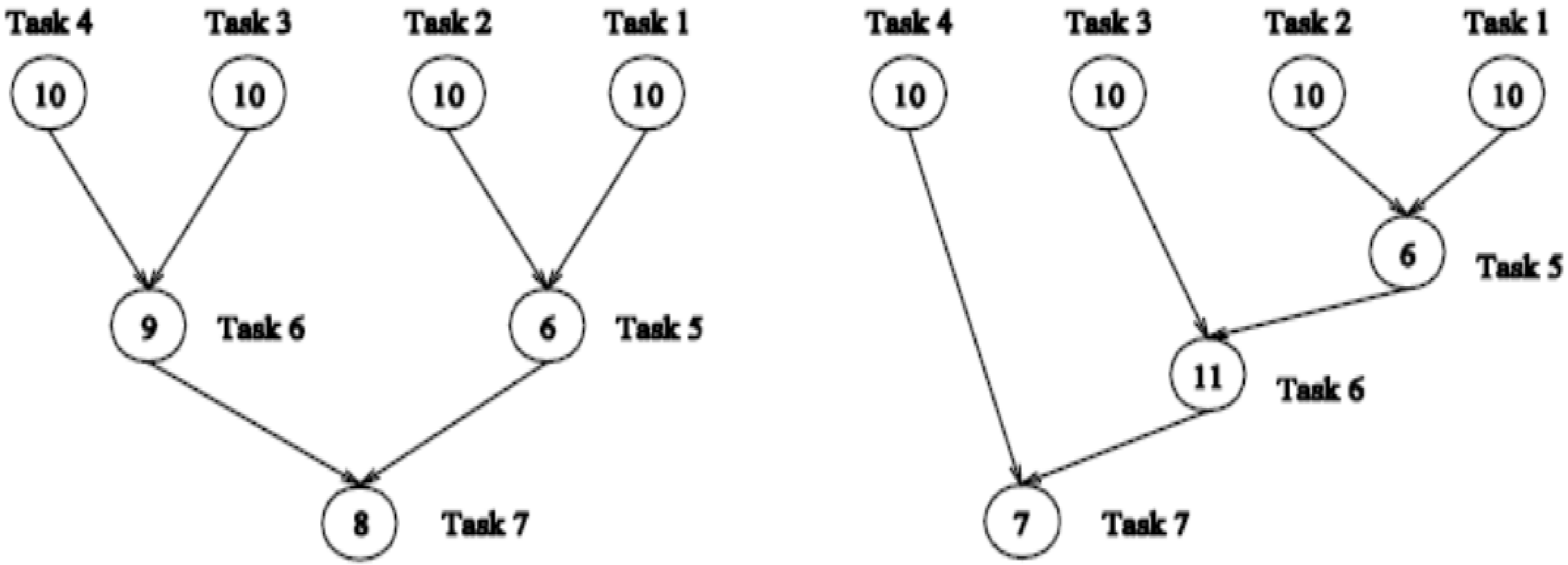
Task Dependence Graph

- Can be used to visualize and evaluate the task decomposition strategy
- A **directed acyclic graph**:
  - Node: Represent each task, node value is the expected execution time
  - Edge: Represent **control dependency** between task

- Properties:
  - Critical Path Length: Maximum (slowest) completion time
  - Degree of concurrency = Total Work / Critical Path Length
    - An indication of amount of work that can be done concurrently

Task Dependence Graph - Example

- Decompositions A and B can be visualized as:

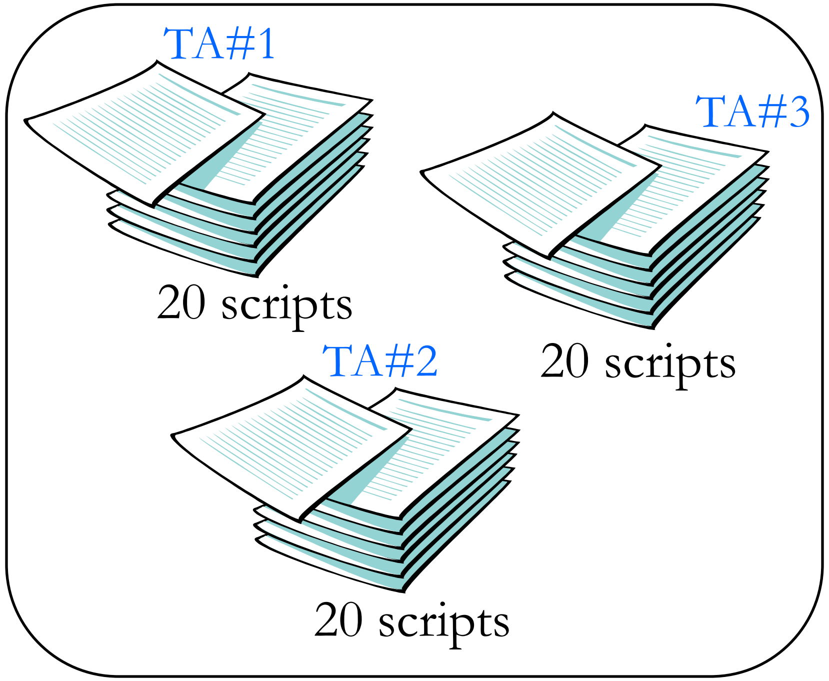


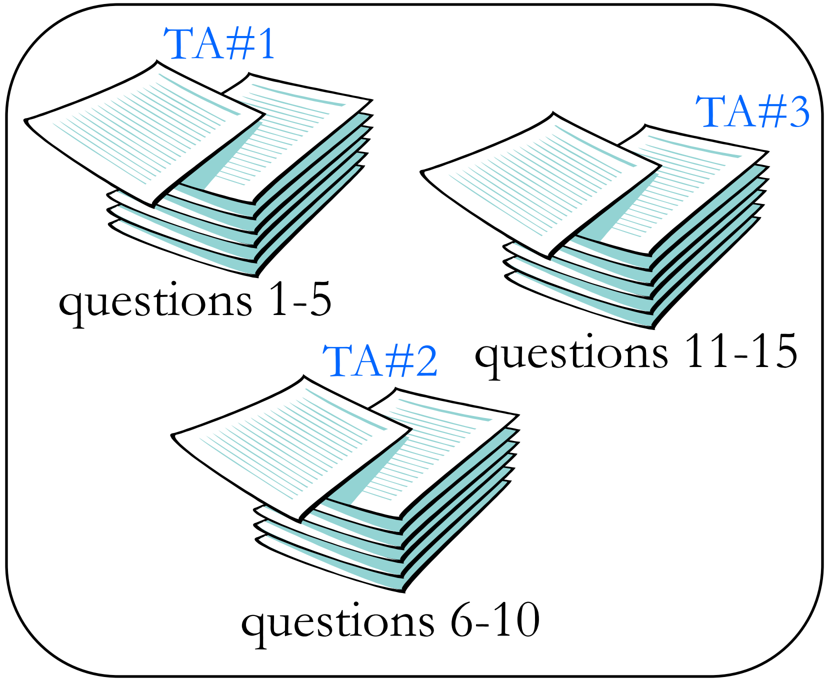
Critical Path = (Task 4 → 6 → 7)
Critical Path Length = **27**
Degree of concurrency = 63 / 27 = **2.33**

Critical Path = (Task 1 → 5 → 6 → 7)
Critical Path Length = **34**
Degree of concurrency = 64 / 34 = **1.88**

Example: Data vs Task Parallelism

- Suppose we have 60 assignment scripts, each with 15 questions to be distributed to 3 TAs for marking:





task or data parallel?

Example: Sum N numbers (lecture 1)

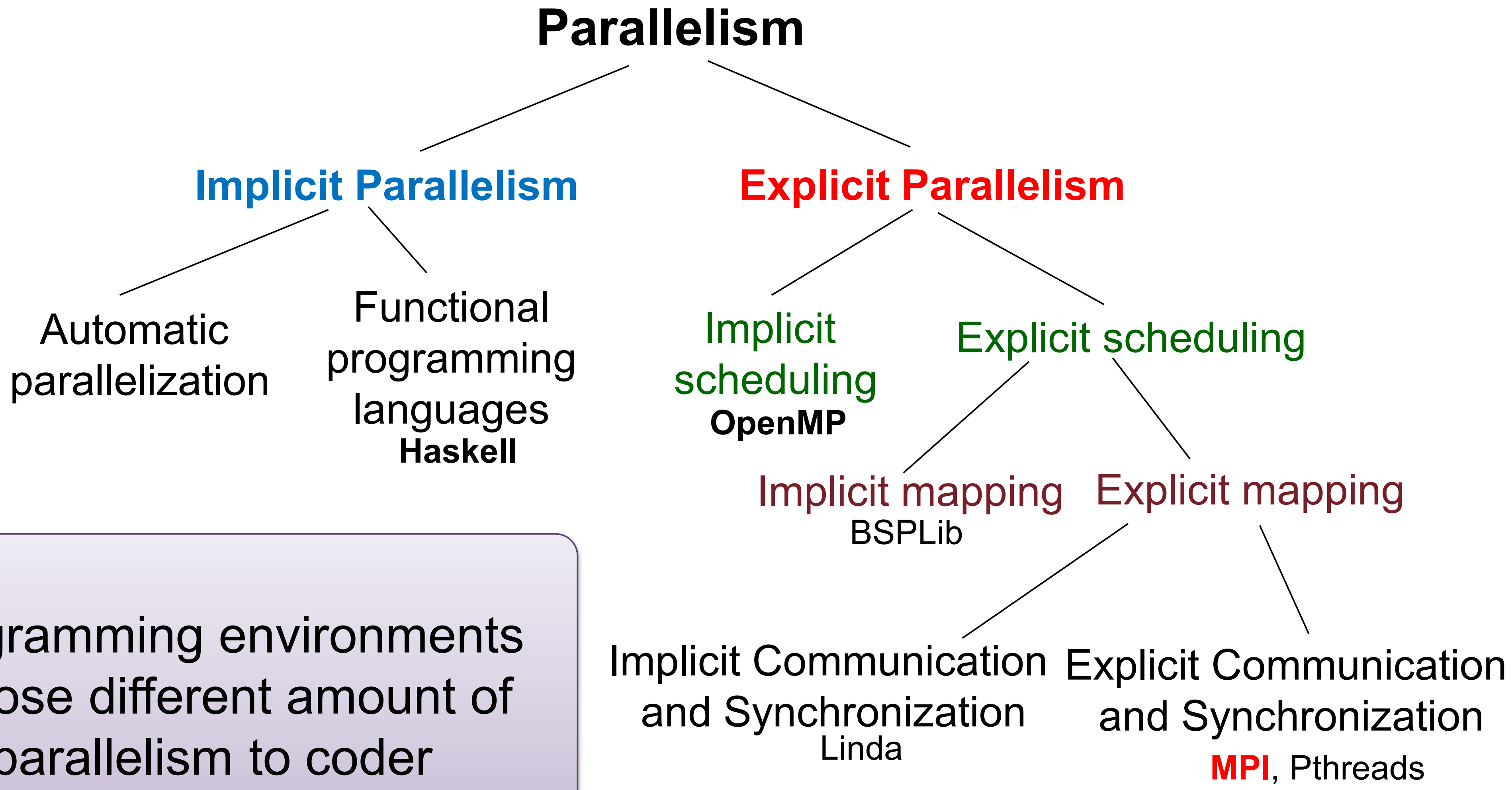
```
my_sum = 0;
my_start = ……;
my_end = ……;
for (my_i = my_start; my_i < my_end; my_i++){
    my_x = Compute_next_value(. . .);
    my_sum += my_x;
}
```

```
if (I'm the master core){
    sum = my_x;
    for each core other than myself {
        receive value from other core
        sum += value;
    }
} else {                //not master core
     send my_x to the master;
}
```

Two versions discussed in lecture 1

Can you distinguish the two forms of parallelism exploited?

Representation of Parallelism



Programming environments expose different amount of parallelism to coder

MODELS OF COORDINATION

Overheads of Parallelism

- Given enough parallel work, overheads are the biggest barrier to getting desired speedup (improvement in performance)
  - cost of starting a parallel task
  - manage and coordinate large number of inter-processor/task interactions
- Overheads can be in the range of milliseconds (= millions of flops) on some systems

Models of Coordination (Communication)

- Shared address space
- Data parallel
- Message passing

Shared Address Space

- Communication abstraction
  - Tasks communicate by reading/writing to shared variables
  - Ensure mutual exclusion via use of locks
  - Logical extension of uniprocessor programming
- Requires hardware support to implement efficiently
  - Any processor can load and store from any address
  - Even with NUMA, costly to scale
  - Matches shared memory systems – UMA, NUMA, etc.

Data Parallel

- Historically: same operation on each element of an array
  - SIMD, vector processors
- Basic structure: map a function onto a large collection of data
  - Functional: side-effect free execution
  - No communication among distinct function invocations
    - Allows invocations to be scheduled in parallel
  - Stream programming model
- Modern performance-oriented data-parallel languages do not strictly enforce this structure
  - CUDA, OpenCL, ISPC

Message passing

- Tasks operate within their own private address spaces
  - Tasks communicate by explicitly sending/receiving messages
- Popular software library: MPI (message passing interface)
- Hardware does not implement system-wide loads and stores
  - Can connect commodity systems together to form large parallel machine
- Matches distributed memory systems
  - Programming model for clusters, supercomputers, etc.

Correspondence with Hardware Implementations

- Common to implement message passing abstractions on machines that implement a shared address space in hardware
  - “Sending message” = copying memory from message library buffers
  - “Receiving message” = copy data from message library buffers
- Possible to implement shared address space abstraction on machines that do not support it in HW
  - Less efficient software solutions
  - Mark all pages with shared variables as invalid
  - Page-fault handler issues appropriate network requests

Summary of Coordination Models

- Shared address space: very little structure
  - All threads can read and write to all shared variables
  - Drawback: not all reads and writes have the same cost (and that cost is not apparent in program text)
- Data-parallel: very rigid computation structure
  - Programs perform same function on different data elements in a collection
- Message passing: highly structured communication
  - All communication occurs in the form of messages

PROGRAM PARALLELIZATION

Program Parallelization

- Parallelization: Transform sequential into parallel computation
  - Define parallel tasks of the appropriate granularity

- Granularity of computation can be:

Fine-Grain	A sequence of **instructions**
	A sequence of **statements** where each statement consists of several instructions
Coarse-Grain	A **function / method** which consists of several statements

General Design Approach for Parallelization

- Consider machine independent issues first, then machine specific aspects of design
- Task/Channel model:
  - **Task** consists of:
    - Code and Data needed for computation
    - Execute in parallel
    - Mapped onto physical processors
  - Tasks interact by sending messages through **channels**
    - A channel is a message queue (buffer) that connects one task's output port to another task's input port

Foster’s Design Methodology

1. Partitioning

- First partition a problem into many smaller pieces, or tasks

2. Communication

- Provides data required by the partitioned tasks (cost of parallelism)

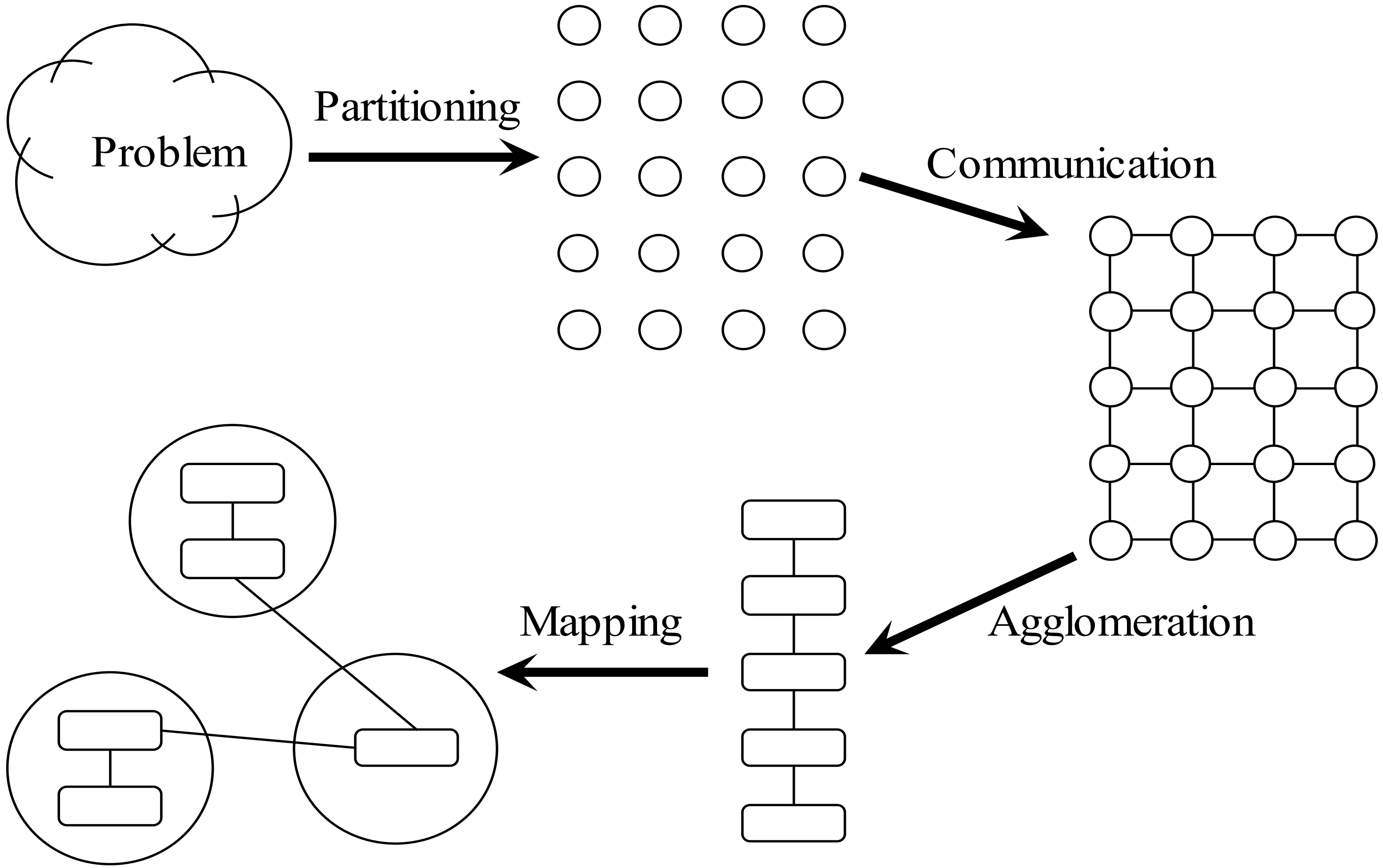
3. Agglomeration

- Decrease communication and development costs, while maintaining flexibility

4. Mapping

- Map tasks to processors (cores), with the goals of minimizing total execution time

Foster's Methodology



1. Partitioning

- Divide **computation** and **data** into independent pieces to discover maximum parallelism
  - Different way of thinking about problems – reveals structure in a problem, and hence opportunities for optimization

Data Centric - Domain decomposition

- Divide data into pieces of approximately equal size
- Determine how to associate computations with the data

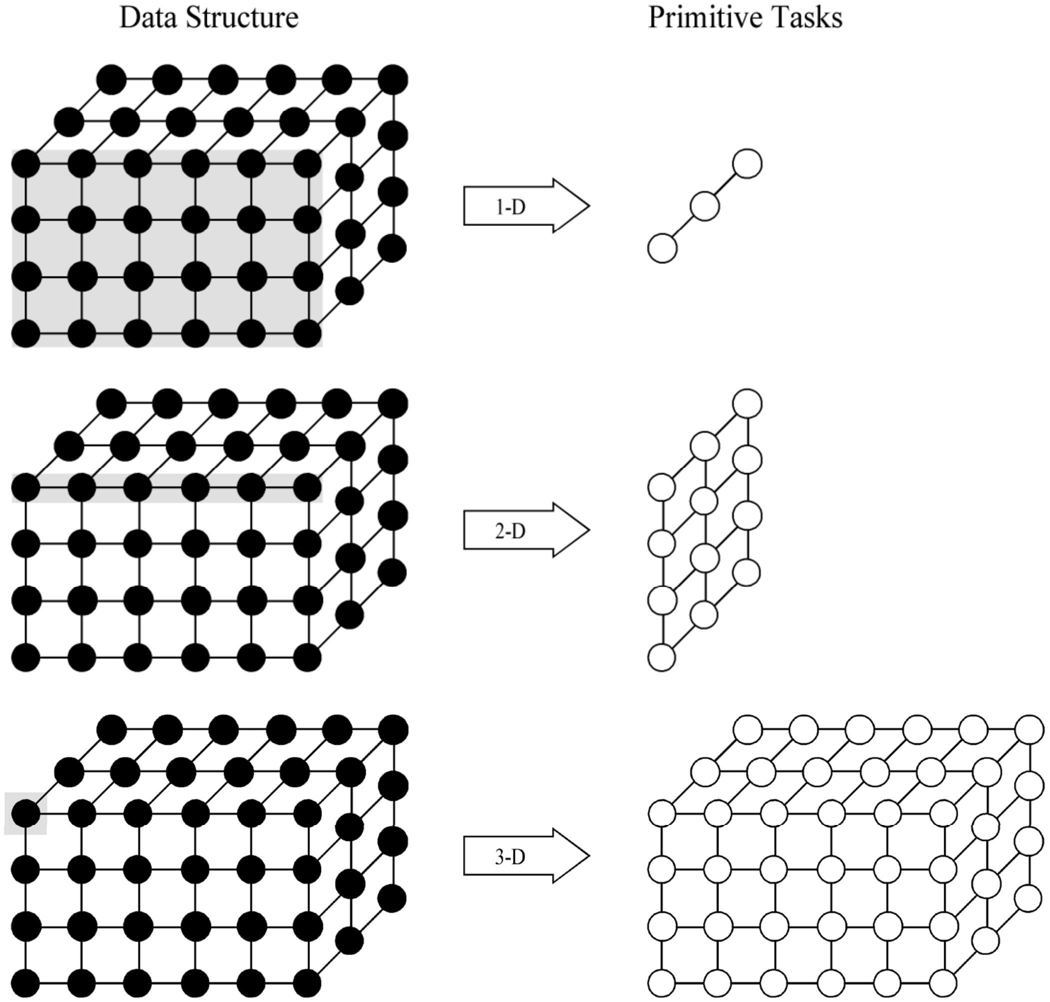
Data parallelism

Computation Centric - Functional decomposition

- Divide computation into pieces
- Determine how to associate data with the computations

Functional parallelism

Example: Domain Decompositions



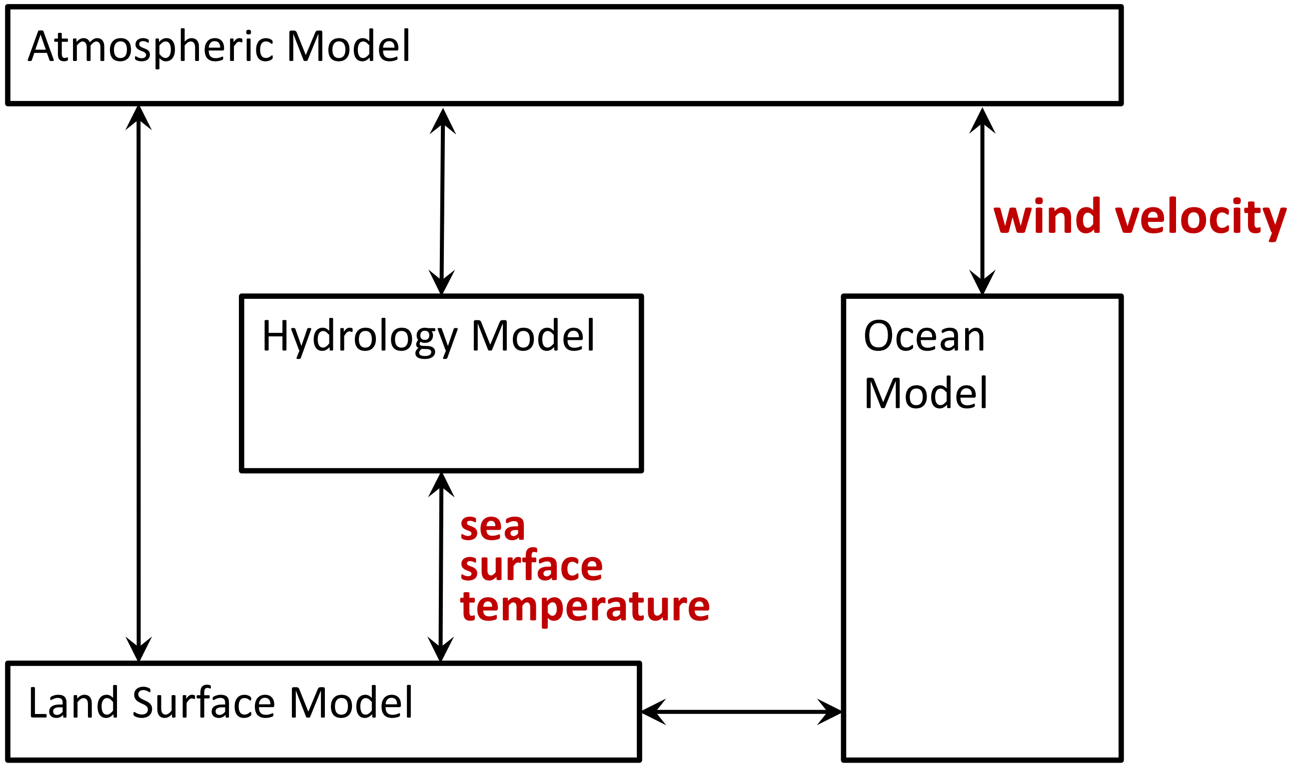
24 tasks, each task with 3 grid points

6 tasks, each task with 12 grid points

1 task with 72 grid points

3-D Matrix (common data structure)

Example: Functional Decomposition



Computer Model of Climate

Partitioning Rules of Thumb

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

2. Communication (Coordination)

- Tasks are intended to execute in parallel
  - but generally not executing independently
  - → Need to determine data passed among tasks

- **Local communication**
  - Task needs data from a small number of other tasks (“neighbors”)
  - Create channels illustrating data flow

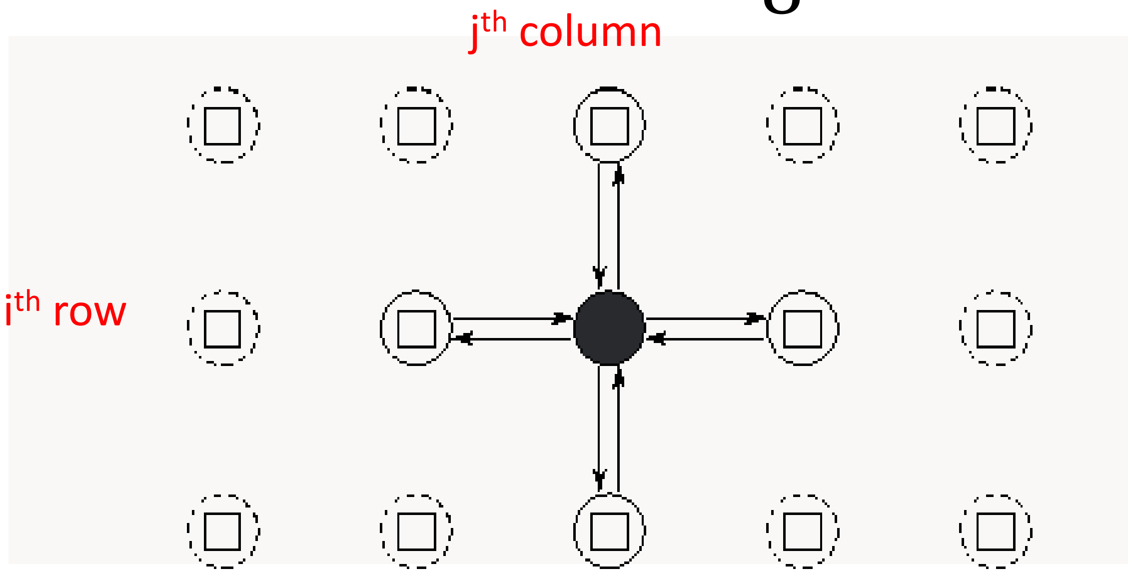
- **Global communication**
  - Significant number of tasks contribute data to perform a computation
  - Don’t create channels for them early in design

- Ideally, distribute and overlap computation and communication

Local Communication

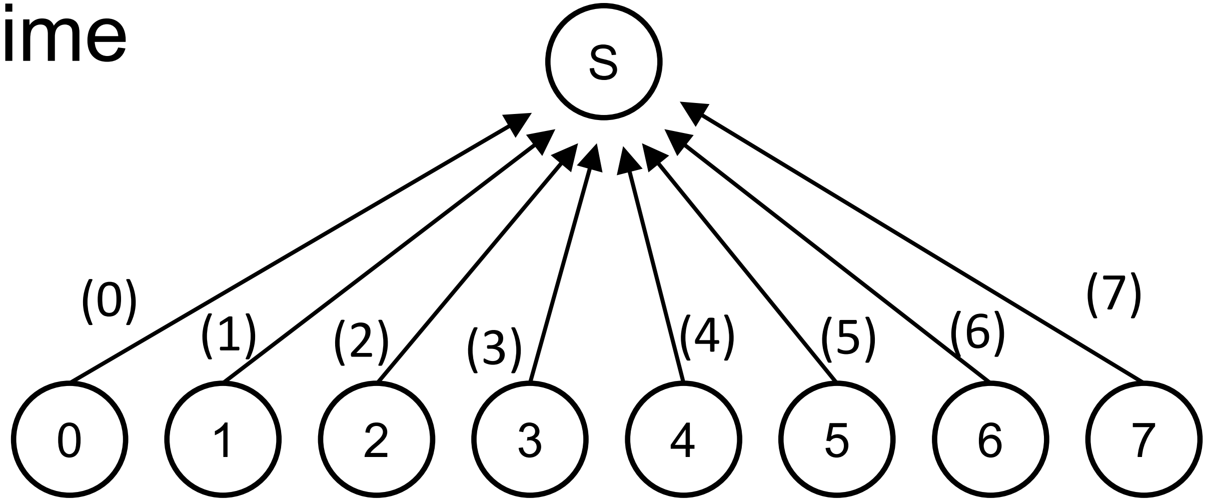
- 2-D Finite Difference Computation
- 2-D grid: at time t+1, requires five points (values at time t) to update each element

$$X_{i,j}^{(t+1)} = \frac{4X_{i,j}^{(t)} + X_{i-1,j}^{(t)} + X_{i+1,j}^{(t)} + X_{i,j-1}^{(t)} + X_{i,j+1}^{(t)}}{8}$$



Global Communication

- Unoptimized sum N numbers distributed among N (= 8) tasks need O(N) time



Centralised Summation Algorithm

- Algorithm is:
  - Centralised – does not distribute computation and communication
  - Sequential – does not allow overlap of computation and communication operations

Communication Rules of Thumb

- Communication operations balanced among tasks
- Each task communicates with only a small group of neighbors
- Tasks can perform communication in parallel
- Overlap computation with communication

3. Agglomeration

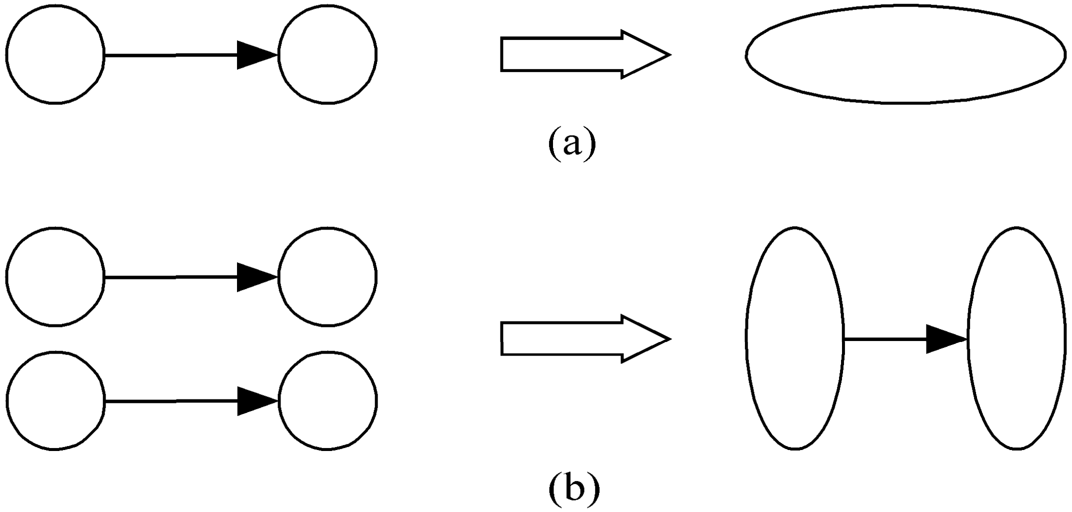
- Combine tasks into larger tasks
  - Number of tasks >= number of cores

- Goals:
  - Improve performance (cost of task creation + communication)
  - Maintain scalability of program
  - Simplify programming

Motivation of Agglomeration

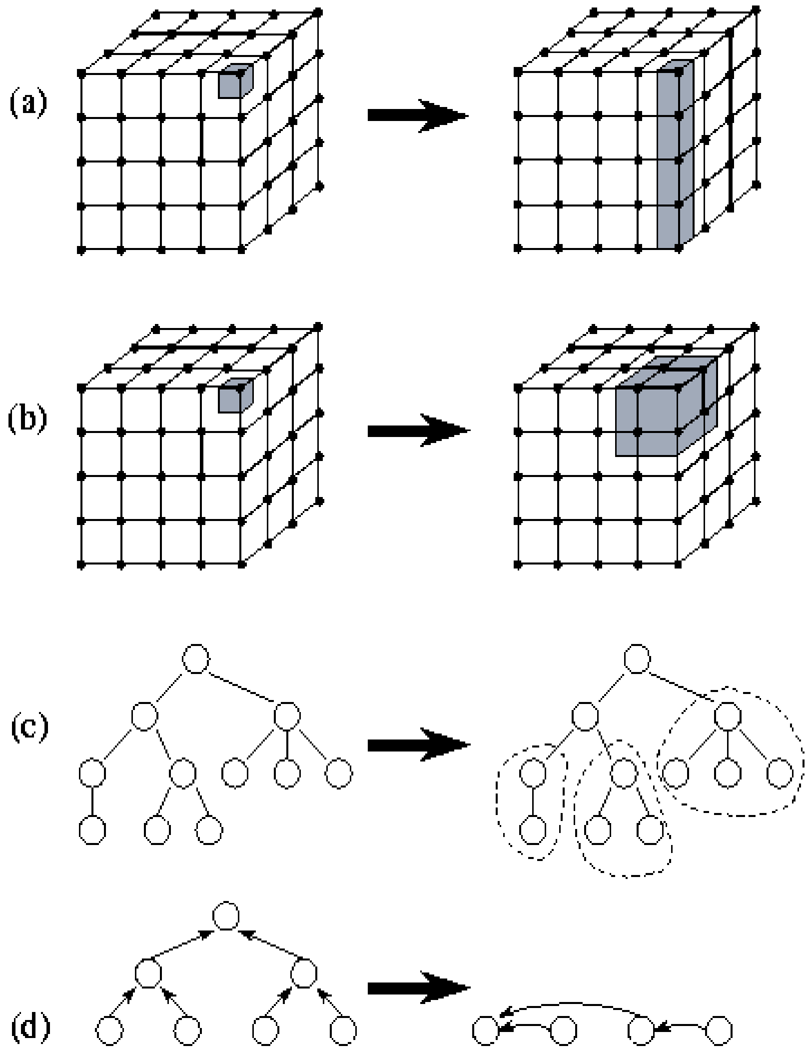
- Eliminate communication between primitive tasks agglomerated into consolidated task

- Eg. Combine groups of sending and receiving tasks



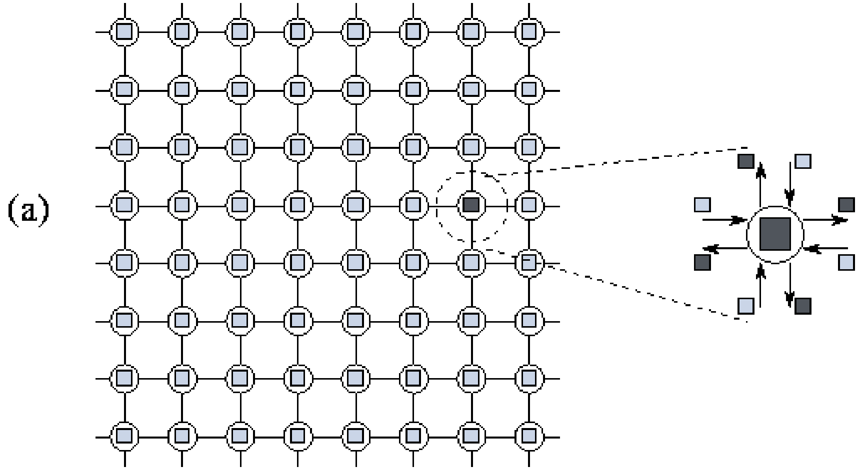
Reduce number of sends and receives

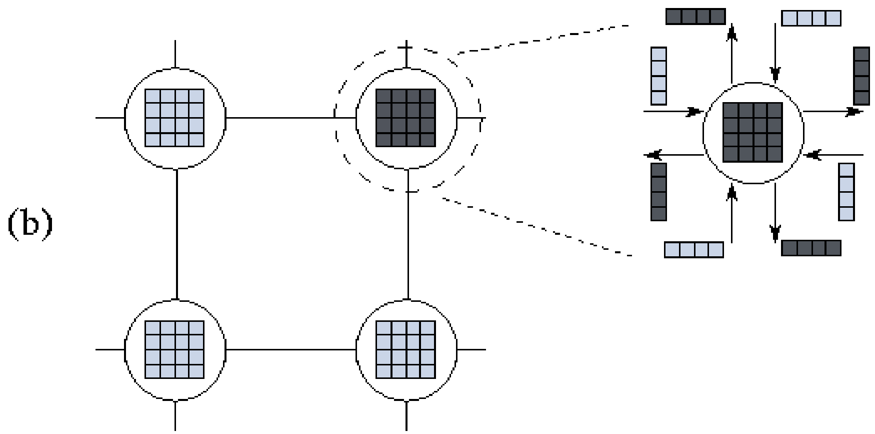
Examples of Agglomeration



- Reduce dimension of decomposition from 3 to 2
- 3-D decomposition (adjacent tasks are combined)
- Divide-and-conquer – sub-tree are coalesced
- Tree algorithm – nodes are combined

Task Granularity: Impact on Communication





2-D 8 x 8 Grid Problem

a. Fine-grain Task Partition

- One grid point per task:
- **? tasks**
- **? communications**
- **? data transfers**

b. Coarse-grain Task Partition

- Each task is a 4 x 4 grid with a total of 16 grid points:
- **? tasks**
- **? communications**
- **? data transfers**

Agglomeration Rules of Thumb

- Locality of parallel algorithm has increased
- Number of tasks increases with problem size
- Number of tasks suitable for likely target systems
- Tradeoff between agglomeration and code modifications costs is reasonable

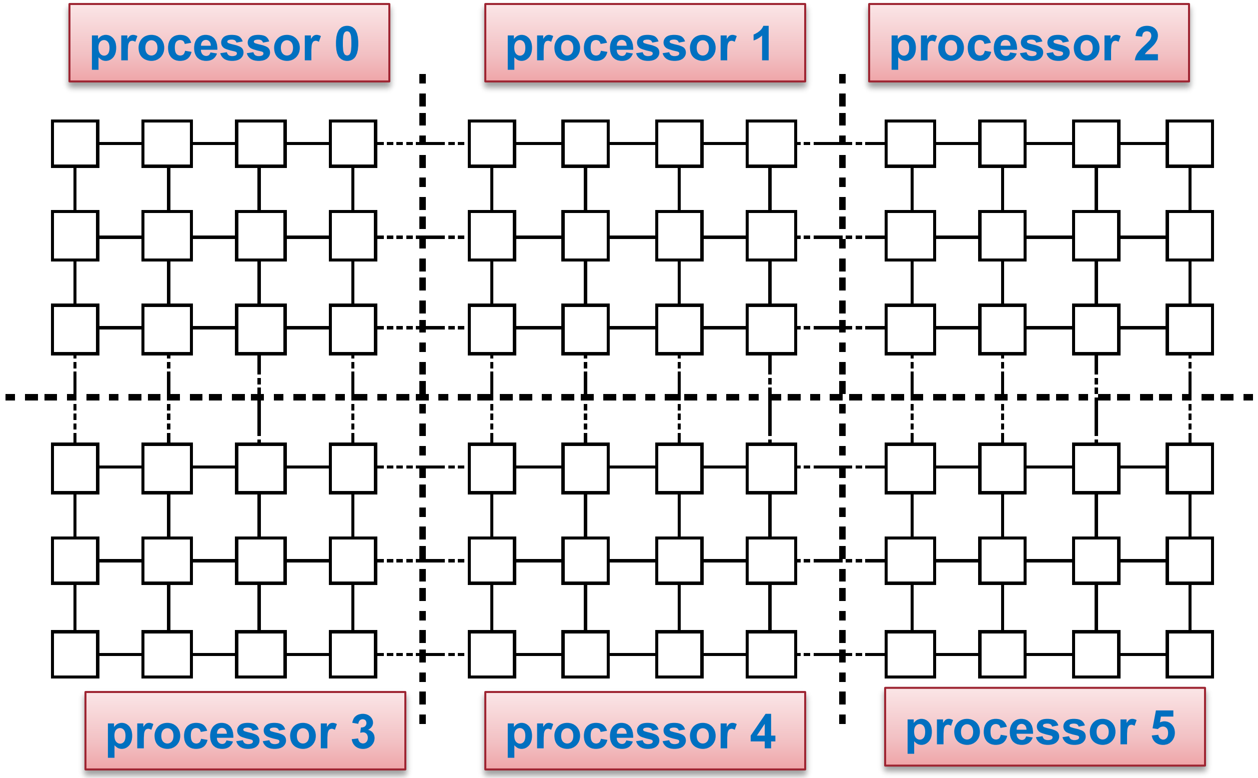
4. Mapping

- **Assignment of tasks** to execution units

- Conflicting goals:
  - **Maximize processor utilization** – place tasks on different processors to increase parallelism
  - **Minimize inter-processor communication** – place tasks that communicate frequently on the same processor to increase locality

- Mapping may be performed by:
  - OS for centralized multiprocessor
  - User for distributed memory systems

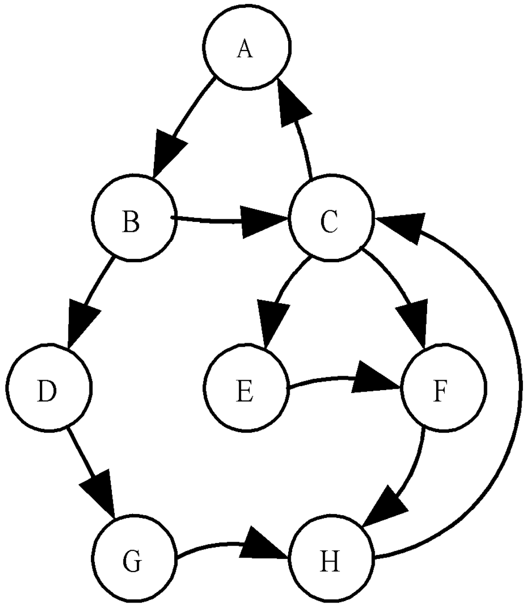
Mapping Example



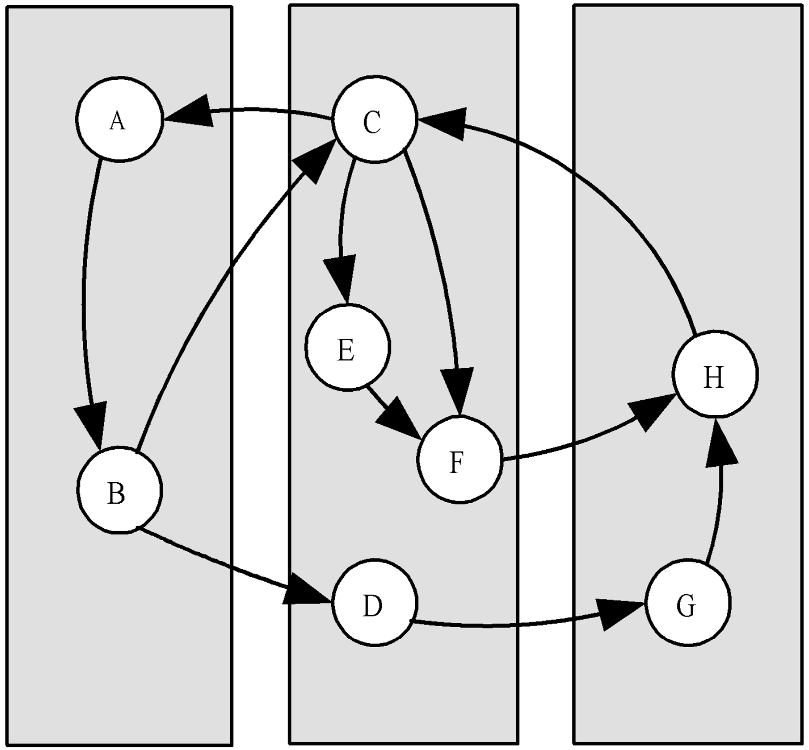
12 x 6 Grid Problem

- Same amount of work on each processor and to minimize off-processor communications

Mapping Example



a. Task/Channel Graph



b. Mapping on Three Processors

Mapping Rules of Thumb

- Finding optimal mapping is NP hard in general
  - Must rely on heuristic

- Consider designs based on one task per processor and multiple tasks per processor

- Evaluate static and dynamic task allocation
  - If dynamic task allocation chosen, task allocator should not be a bottleneck to performance
  - If static task allocation chosen, ratio of tasks to processors is at least 10:1

Foster's Design Methodology

1. Partitioning

- First partition a problem into many smaller pieces, or tasks

2. Communication

- Provides data required by the partitioned tasks (cost of parallelism)

3. Agglomeration

- Decrease communication and development costs, while maintaining flexibility

4. Mapping

- Map tasks to processors (cores), with the goals of minimizing total execution time

Sequential Algorithm

↓ decompose

Tasks

↓ schedule

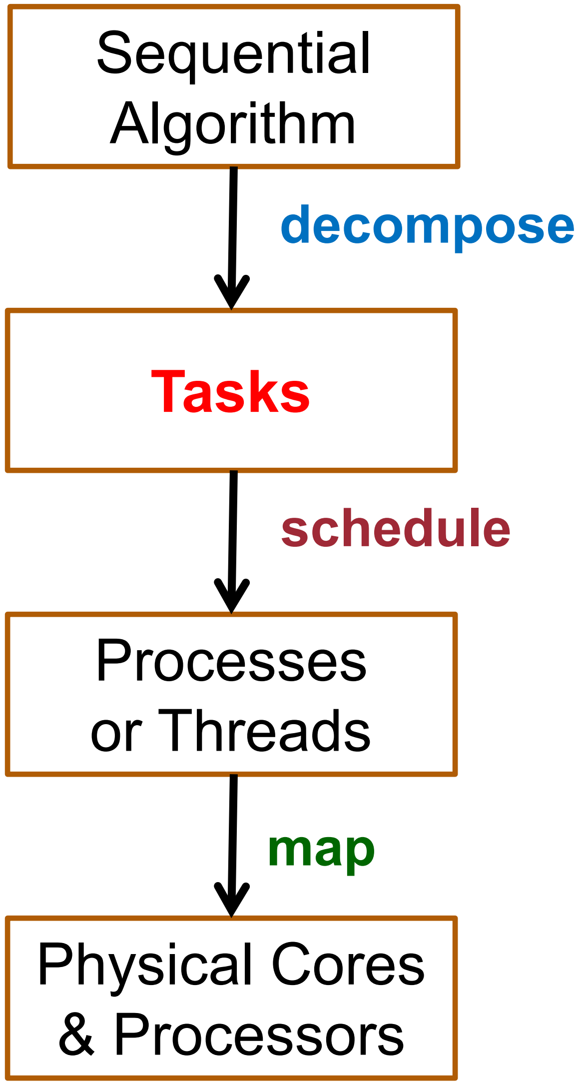
Processes or Threads

↓ map

Physical Cores & Processors

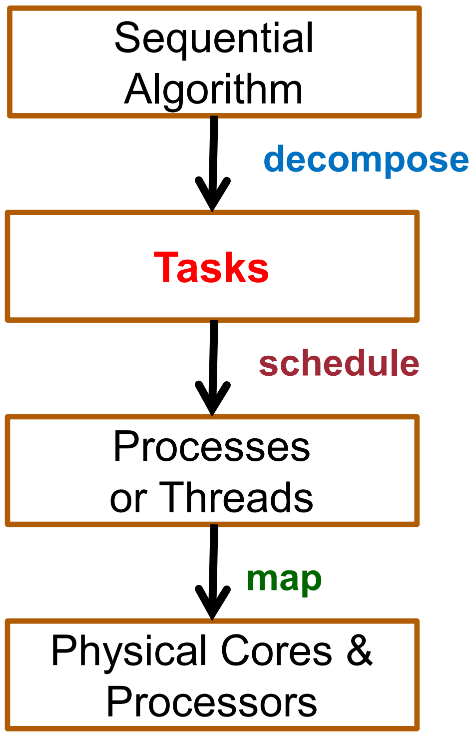
Program Parallelization: Steps

- 3 main steps:
  - Decomposition of the computations
  - Scheduling (assignment of tasks to processes (or threads))
  - Mapping of processes (or threads) to physical processors (or cores)



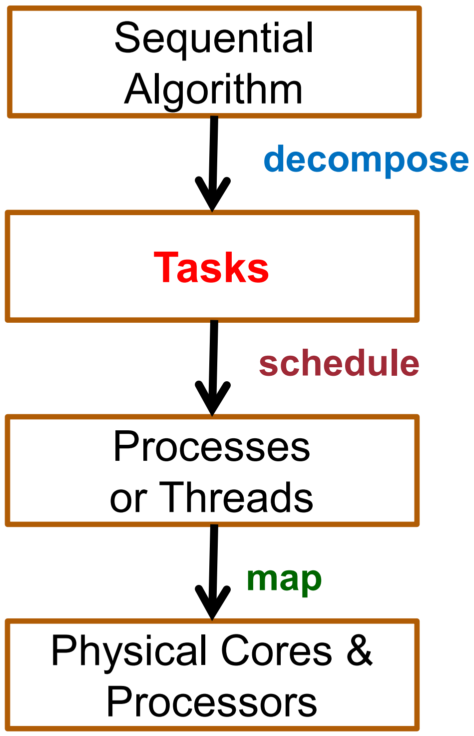
Decomposition

- **What?**
  - Generates **enough tasks** to keep all cores busy at all times, i.e., **number of tasks ≥ number of cores**
  - Granularity is large compared to the scheduling and mapping time, i.e., **size of task >> overhead of parallelism**
- **When?**
  - **Static** at program start or compile time
  - **Dynamic** during program execution



Scheduling

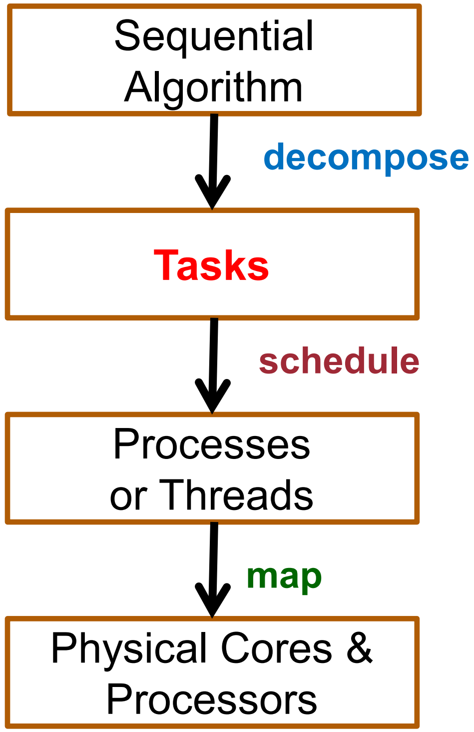
- **What?**
  - Find an efficient task execution order to optimize a given objective function, e.g., overall completion time
  - Good load balancing among tasks:
    - Computations
    - Memory accesses (shared address space)
    - Communication operations (distributed address space)
- **When?**
  - Static scheduling
  - Dynamic scheduling



Mapping

- **What?**
  - **Assignment of processes or threads** to execution units

  - Focuses on performance:
    - *Equal utilization* of execution units
    - *Minimal communication* between the processors



Automatic Parallelization

- Parallelizing compilers perform decomposition and scheduling

- **Drawbacks:**
  - Dependence analysis is difficult for pointer-based computations or indirect addressing
  - Execution time of function calls or loops with unknown bounds is difficult to predict at compile time

Functional Programming Languages

- Describe the computations of a program as the evaluation of mathematical functions without side effects

- Advantages:
  - New language constructs are not necessary to enable a parallel execution

- Challenge:
  - Extract the parallelism at the right level of recursion

PARALLEL PROGRAMMING PATTERNS

Overview

- A **parallel programming pattern** provides a coordination structure for tasks:
  - Similar to design pattern from Software Engineering

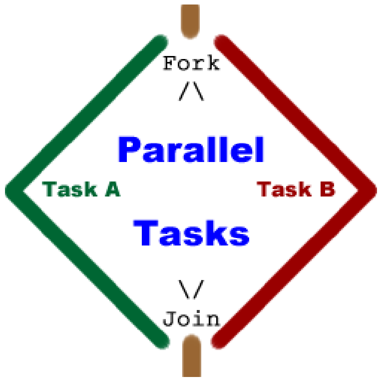
- Examples
  - Fork–Join
  - Parbegin-Parend
  - SPMD and SIMD
  - Master-Slave (Worker)
  - Client-Server
  - Pipelining
  - Task pool
  - Producer-consumer

Fork-Join

- Task T creates a number of child tasks T1,..., Tm with a **fork** statement
  - Child tasks work in parallel and execute

  a given program part or function
  - Task T can execute the same or

  a different program part or function

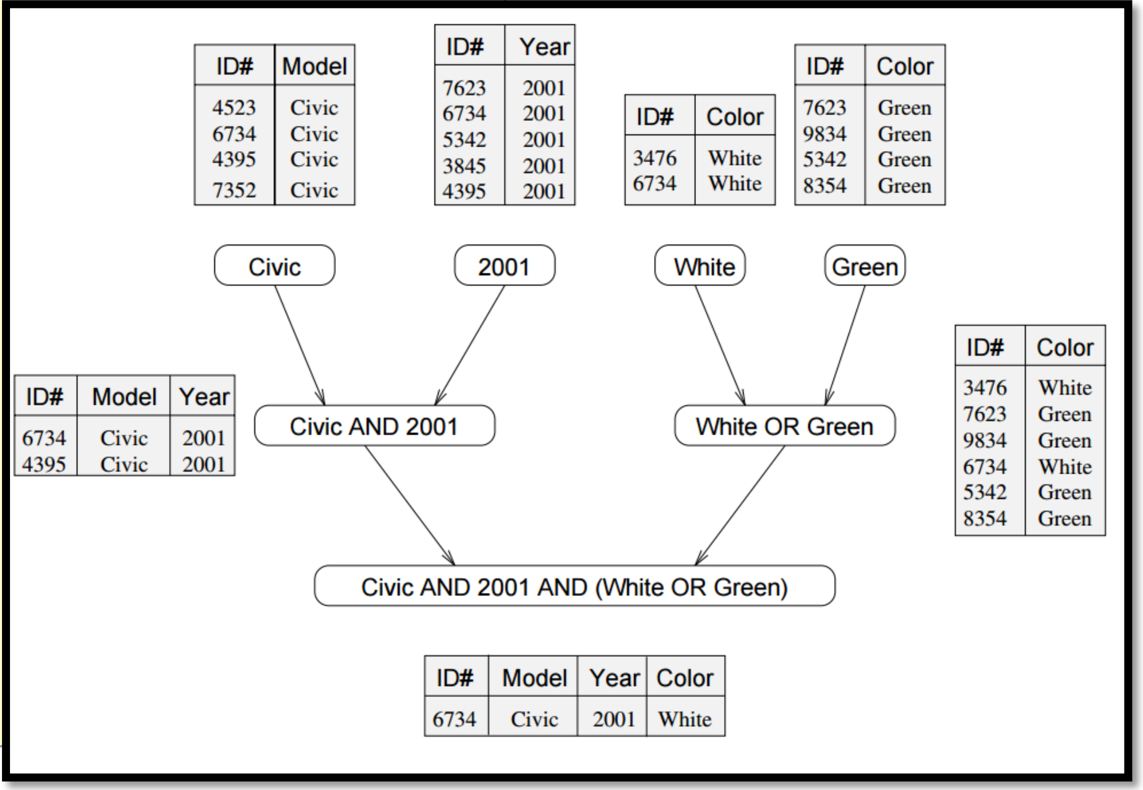


- Task T waits for the termination of T1,..., Tm by using a **join** call

- **Implementation**:
  - Language construct or a library function such as Pthreads, OpenMP and MPI-2

Example: Database Query (A)

```
P1 = Fork {
    P3 = Fork { return Model = "civic" }
    P4 = Fork { return Year = "2001" }
    Join P3, P4
    Return P3 AND P4
}
P2 = Fork {
    P5 = Fork { return Color = "green" }
    P6 = Fork { return Color = "white" }
    Join P5, P6
    Return P5 OR P6
}
Join P1, P2
Return P1 AND P2
```



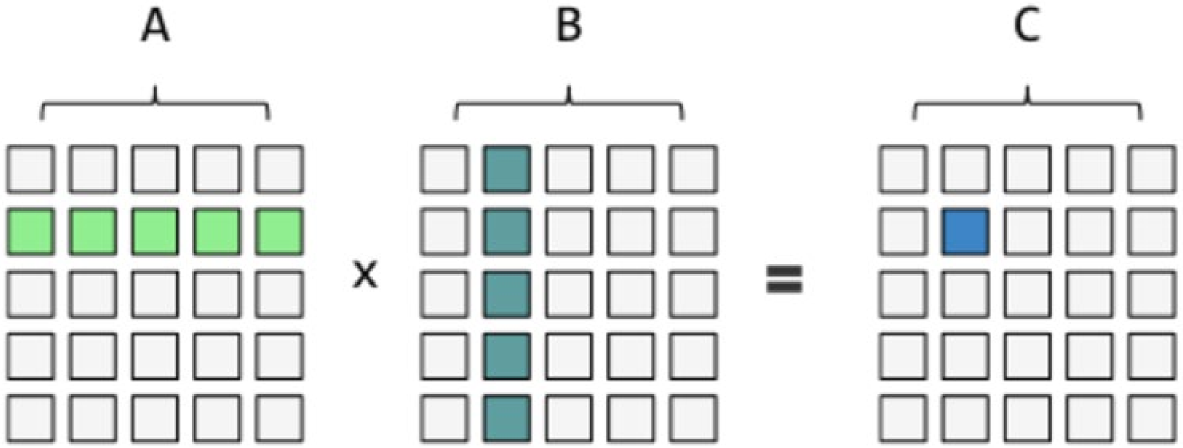
Parbegin–Parend

- Programmer specifies a sequence of statement (function calls) to be executed by a set of processors in parallel
  - When an executing thread reaches a **parbegin–parend construct**, a set of threads is created and the statements of the construct are assigned to these threads for execution

- The statements following the **parbegin–parend construct** are only executed after all these threads have finished their work

- Implementation:
  - a language construct (OpenMP) or compiler directives

Matrix Multiplication




```
for i ← 0 to n-1
    for j ← 0 to n-1
        c[i, j] ← 0
        for k ← 0 to n-1
            c[i, j] ← c[i, j] + a[i, k] x b[k, j]
```

Example: Parallel For in OpenMP

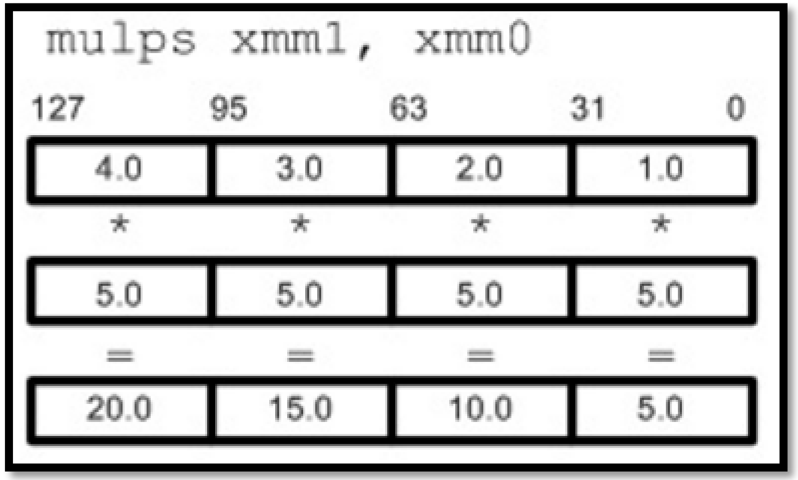
- Iterations of the for loop executed in parallel by a group threads

```
// Parallelize the matrix multiplication (result = a x b)
// Each thread will work on one iteration of the outer-most loop
// Variables are shared among threads (a, b, result)
// and each thread has its own private copy (i, j, k)


#pragma omp parallel for shared(a, b, result)
                                private (i, j, k)

for (i = 0; i < size; i++)
    for (j = 0; j < size; j++)
        for (k = 0; k < size; k++)
            result.element[i][j] += a.element[i][k] *
                                    b.element[k][j];
```

SIMD

- **Single instructions** are executed synchronously by the different threads on different data

- Implementation:
  - SSE Instruction on intel processor



`xmm` registers are 128 bits long

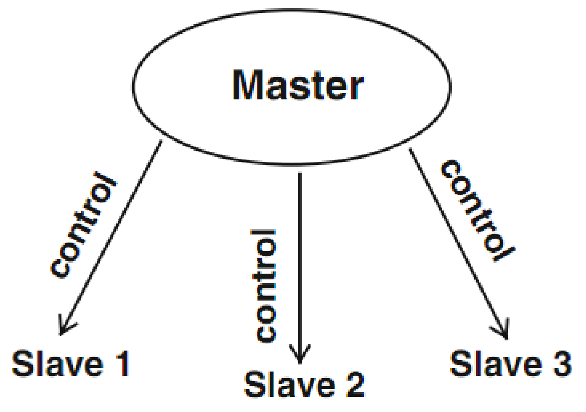
SSE instruction treats the xmm registers as 4 individual 32-bit floating point value

SPMD

- **Same** **program** executed on different processors but operate on different data
  - All **threads have equal rights** and different threads work asynchronously with each other
  - Different threads may execute different parts of the parallel program because of
    - Different speeds of the executing processors
    - Control statement in program, e.g., If statement
- No implicit synchronization
  - Synchronization can be achieved by explicit synchronization operations
- Implementation:
  - Example: MPI (Tutorial 1 "hello world" and more in future lectures)

Master–Slave (or Master–Worker)

- A single program (master) controls the execution of the program
  - Master executes the main function
  - Assigns work to slave threads to perform computations



- **Master task**:
  - Generally responsible for coordination and perform initializations, timings, and output operations

- **Slave task**:
  - Wait for instruction from master task

Matrix Multiplication – Master-Slave

```
int main(int argc, char ** argv)
{
        int nprocs;
        MPI_Init(&argc,&argv);
        MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
        MPI_Comm_rank(MPI_COMM_WORLD, &myid);
        size = 2048;
        // One master (rank = 0) and nprocs-1 slaves
        if (myid == 0)   {
            master();
        } else {
            slave();
        }
        MPI_Finalize();
        return 0;
}
```

Matrix Multiplication – Master-Slave

```
void master()
{
        matrix a, b, result;

        // Allocate memory for matrices
        allocate_matrix(&a);
        allocate_matrix(&b);
        allocate_matrix(&result);

        // Initialize matrix elements
        init_matrix(a);
        init_matrix(b);

        // Distribute data to slaves
        master_distribute(a, b);

        // Gather results from slaves
        master_receive_result(result);

        // Print the result matrix
        print_matrix(result);
}
```

Matrix Multiplication – Master-Slave

```
void slave()
{
        int rows_per_slave = size / slaves ;
        float row_a_buffer[rows_per_slave][size];
        matrix b;
        float result[rows_per_slave][size];

         // Receives data
        slave_receive_data(&b, row_a_buffer);

         // Performs computations
         slave_compute(b, row_a_buffer, result);

         // Sends the results to master
         slave_send_result(result);

}
```

Client–Server

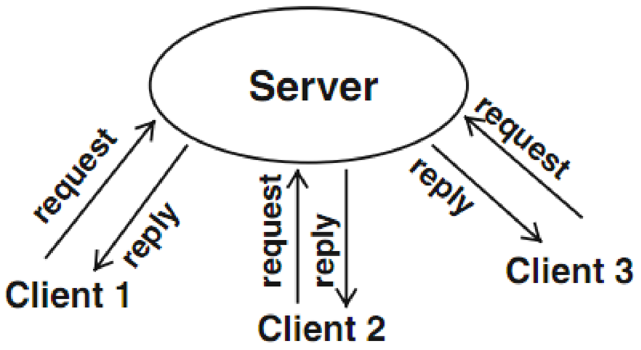
- **MPMD** (multiple program multiple data) model

- Server compute requests from multiple client tasks concurrently
  - Can use multiple threads to compute a single request

- A task can generate requests to other tasks (client role) and process requests from other tasks (server role)

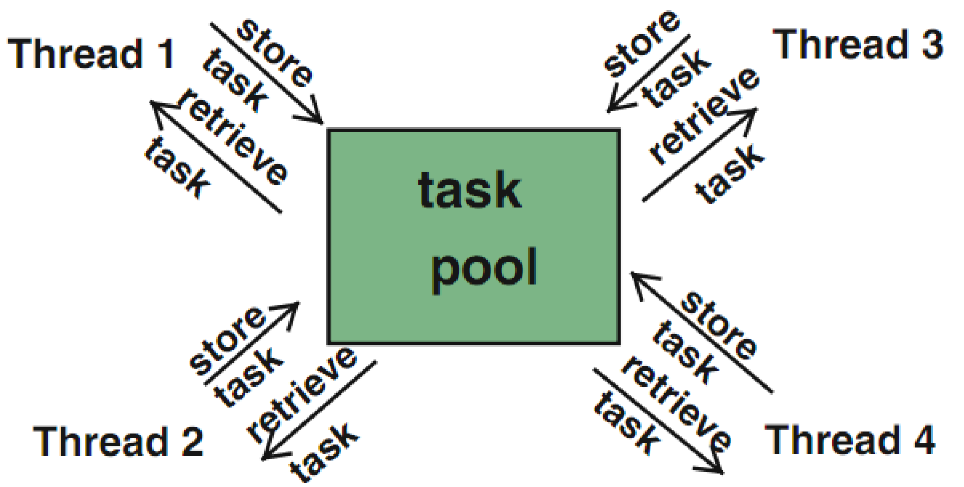
- Useful in heterogeneous systems

such as cloud and grid computing



Task (Work) Pools

- A common data structure from which threads can access to retrieve tasks for execution



- Number of threads is fixed
  - Threads are created statically by the main thread
- During the processing of a task, a thread can generate new tasks and insert them into the task pool

Task (Work) Pools

- Access to the task pool must be synchronized to avoid race conditions

- Execution of a parallel program is completed when
  - Task pool is empty
  - Each thread has terminated the processing of its last task

- Advantages:
  - Useful for adaptive and irregular applications
    - Tasks can be generated dynamically
  - Overhead for thread creation is independent of the problem size and the number of tasks

- Disadvantages:
  - For fine-grained tasks, the overhead of retrieval and insertion of tasks becomes important

Example: Java Thread Pool Executor

```
class ThreadPoolExample {

    public static void main(String[] args) {
        ExecutorService executor =
                    Executors.newFixedThreadPool(5);

        for (int i = 0; i < 10; i++) {
            Runnable Task = new Task( ..... );
            executor.execute( Task );
        }
        .......
    }
}
```

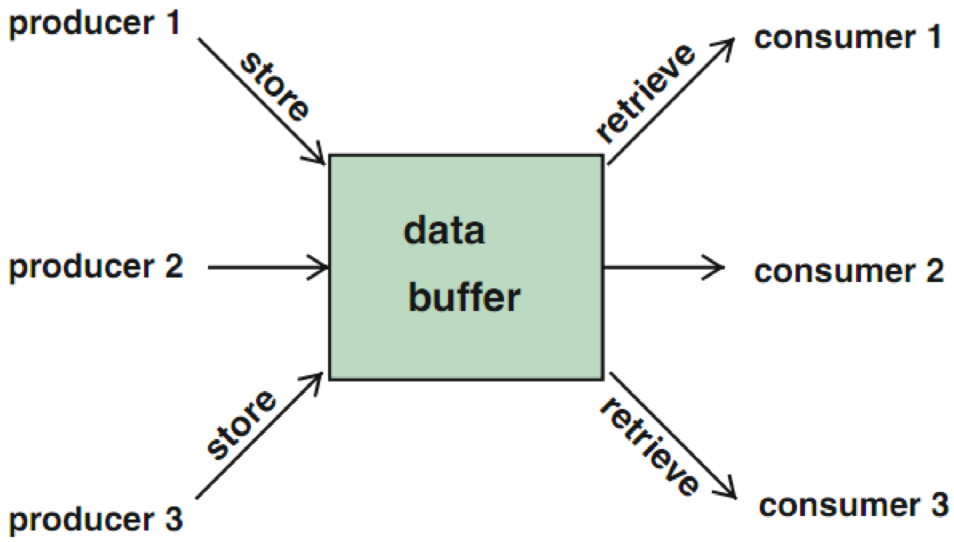
5 threads

10 tasks added to the pool.

- The executor will assign task to the 5 threads:
  - After a thread finishes its task, another task from the pool will be assigned

Producer–Consumer

- Producer threads produce data which are used as input by consumer threads



- Synchronization has to be used to ensure a correct coordination between producer and consumer threads

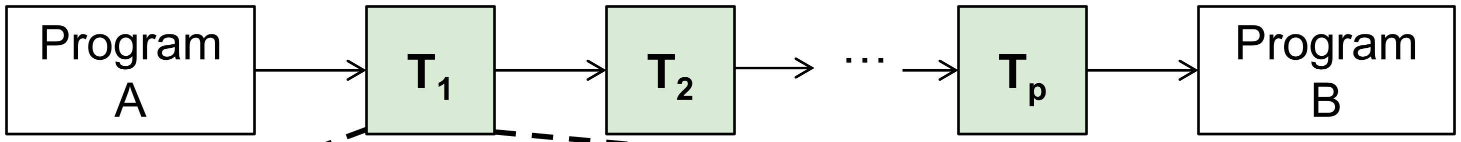
Producer–Consumer: Shared Buffers

```
void produce() {
    synchronized (buffer) {
        while (buffer is full)
            buffer.wait();
        Store an item to buffer;
        if (buffer was empty)
            buffer.notify();
    }
}
```

```
void consume() {
    synchronized (buffer) {
        while (buffer is empty)
            buffer.wait();
        Retrieve an item from buffer;
        if (buffer was full)
            buffer.notify();
    }
}
```

Pipelining

- Data in the application is partitioned into a stream of data elements that flows through the each of the pipeline tasks one after the other to perform different processing steps
  - A form of functional parallelism: **Stream parallelism**




```
initialize
while (more data) {
  receive data element from previous stage
  perform operation on data element
  send data element to next stage
}
finalize
```

Question

Assume you need to compute prefix sums for multiple arrays for image convolution operation

1. Devise an algorithm to compute the prefix sums in a pipeline programming model (you may use pseudo-code)
2. How many operations do you need for each stage of the pipeline? What are these operations?
3. Enumerate and justify your assumptions about array size and processing units

Summary

- Models of Communication
- Types and representation of parallelism
- Three steps in program parallelization
- Main parallel programming patterns

References

- **Main Reference Book**
  - Chapter 3

- **Introduction to Parallel Computing**
  - by Grama, Gupta, Karypis, Kumar
  - http://www-users.cs.umn.edu/~karypis/parbook/