Intro To Parallel Computing

Shuhao Zhang

Nanyang Technological University

shuhao.zhang@ntu.edu.sg

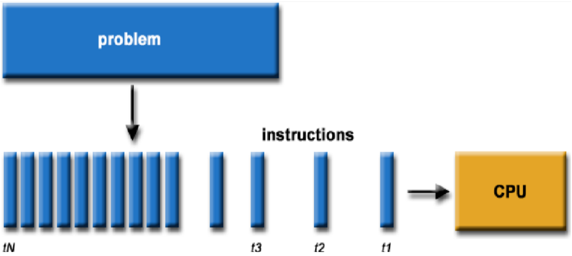
June 3, 2024

Observations

Single Processor: powerful, but has capacity upper bond. Failed to meet *Moore's Law* since early 2000.

Multicore Processor: a collection of processing units to cooperatively solve a problem quickly.

Serial Computing



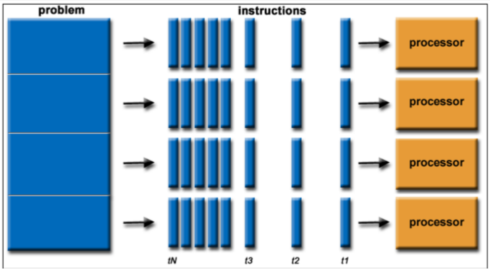
Traditionally, a problem is divided into a discrete series of instructions

- Instructions are executed one after another
- Only one instruction executed at any moment in time

Parallel Computing

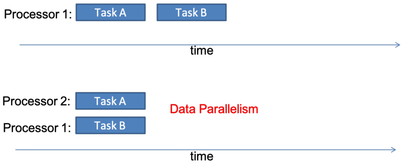
- **Simultaneous use of multiple processing units** to solve a problem fast / solve a larger problem
- **Processing units** could be
  - A single processor with multiple cores
  - A single computer with multiple processors
  - A number of computers connected by a network
  - Combinations of the above
- Ideally, a problem (application) is partitioned into *sufficient* number of *independent* parts for execution on parallel processing elements

Illustration of Parallel Computing



- A problem is divided into 4 pieces (tasks) that can be solved concurrently
- Each task may be processed as multiple instructions same as *serial computing*

Benefit of Concurrency



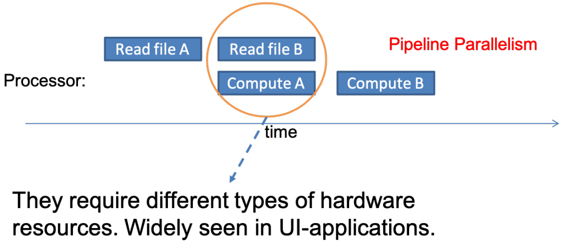
- Better hardware resource utilization: with K processors, ideally we can be K times faster
- Time Complexity: $O(n) \rightarrow O(n/k)$

Observation

It does not change from $O(n)$ to $O(log(n))$ or $O(loglog(n))$. Why it still helps?

Benefit of Concurrency

- Can we get better performance with 1 core only[1]?



[1] FYI: A single-core computer is rarely seen nowadays, but it does exist in the history :)

Benefit of Concurrency: Example Application

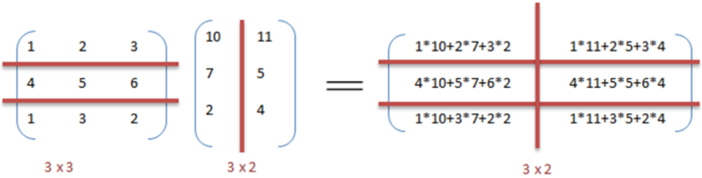
Matrix Multiplication

$$\underbrace{\begin{bmatrix} a & b \\ c & d \end{bmatrix}}_{A} \times \underbrace{\begin{bmatrix} e & f \\ g & h \end{bmatrix}}_{B} = \underbrace{\begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}}_{C}$$

A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2

Benefit of Concurrency: Example Application

Matrix Multiplication



- A[m x n] dot B [n x k] can be finished in $O(n)$ instead of $O(m * n * k)$ when executed in parallel using $m * k$ processors.

What is “Process"

A program in execution:

- Identified by PID (process ID)
- Comprises:
  - executable program and Program Counter[2]
  - global data
    - OS resources: open files, network connections
  - stack or heap
  - current values of the registers (e.g., General Purpose Register (GPR))
- Own memory address space → exclusive access to its data
- Two or more processes exchange data → need explicit communication mechanism (IPC)

[2] A program counter (PC) is a CPU register in the computer processor which has the address of the next instruction to be executed from memory.

Example of “Fork()"

```c
int main(int argc, char *argv[])
{
        char *name = argv[0];
        int child_pid = fork();
        if (child_pid == 0) {
                printf("Child of %s is %d\n", name, getpid());
                return 0;
                }else{
                printf("My child is %d\n", child_pid);
                return 0;
        }
}
```

Multi-Processes Programming

Several processes at different stages of execution

- Need **context switch**, i.e., switching between processes
- States of the suspended process must be saved → overhead
- two types of multi-processes execution:
  - Time slicing execution – pseudo-parallelism
  - Parallel execution of processes on different resources (e.g., cores)

Inter-process Communication (IPC)

Cooperating processes have to share information:

- Shared memory: Need to protect access when reading/writing to the same space concurrently
- Message passing:
  - Blocking & non-blocking
  - Synchronous & asynchronous

Example of IPC through Shared-Memory

```
#include <sys/ipc.h>
#include <sys/shm.h>
using namespace std;
int main(){
key_t key = ftok("shmfile",65);// ftok to generate unique key
int shmid = shmget(key,1024,0666|IPC_CREAT);// shmget returns an identifier in shmid
char *str = (char*) shmat(shmid,(void*)0,0);// shmat to attach to shared memory
gets(str);//write data
shmdt(str);//detach from shared memory
return 0;
}
```

```
#include <sys/ipc.h>
#include <sys/shm.h>
int main(){
key_t key = ftok("shmfile",65); // ftok to generate unique key
int shmid = shmget(key,1024,0666|IPC_CREAT); // shmget returns an identifier in shmid
char *str = (char*) shmat(shmid,(void*)0,0);// shmat to attach to shared memory
printf("Data read from memory: %s\n",str);//read data
shmdt(str);//detach from shared memory
// destroy the shared memory
shmctl(shmid,IPC_RMID,NULL);
return 0;
}
```

⚠: Such shared-memory IPC is **not available** for Java/Python.

Example of IPC through Message Passing

```java
public class MyServer {
public static void main(String[]
     args){
try{
        ServerSocket ss=new
            ServerSocket(6666);
        Socket s=ss.accept();//
            establishes
            connection
        DataInputStream dis=new
            DataInputStream(s.
            getInputStream());
        String  str=(String)dis.
            readUTF();
        System.out.println("
            message=␣"+str);
        ss.close();
        }catch(Exception e){
            System.out.println(e
            );}
        }
}
```

```java
public class MyClient {
public static void main(String[]
     args) {
try{
        Socket s=new Socket("
            localhost",6666);
        DataOutputStream dout=new
             DataOutputStream(s.
            getOutputStream());
        dout.writeUTF("Hello␣
            Server");
        dout.flush();
        dout.close();
        s.close();
        }catch(Exception e){
            System.out.println(e
            );}
        }
}
```

⚠: Java uses RMI and socket for communication between processes, it is not “shared-memory" but message passing.

Disadvantages of Processes

- Creating a new process is costly
  - Overhead of system calls
  - All data structures must be allocated, initialized and copied
- Communicating between processes costly
  - Communication goes through the OS

Why Threads?

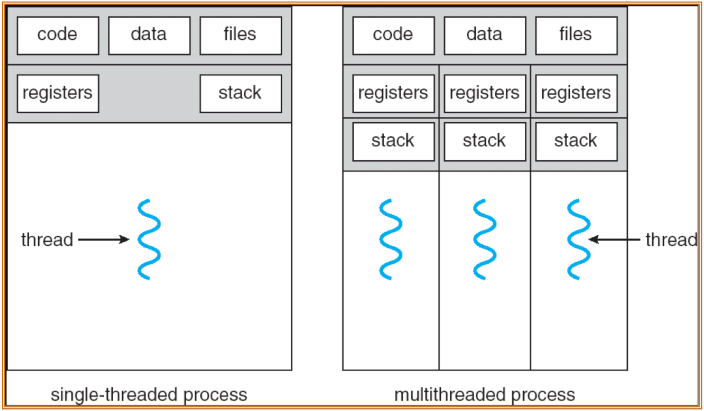
- Extension of process model:
  - A process may consist of multiple independent control flows called “threads"
  - The thread defines a sequential execution stream *within* a process (PC, SP, registers)
- Threads share the address space of the process:
  - All threads belonging to the same process see the same value $\rightarrow$ shared-memory architecture

Why Threads? (cont'd)

- Thread generation is faster than process generation
  - No copy of the address space is necessary
- Different threads of a process can be assigned run on different cores of a multicore processor

⚠We draw attention primarily on multi-threading programming in the first half of this module.

Process and thread: Illustration



Taken from Operating System Concepts (7th Edition) by Silberschatz, Galvin & Gagne, published by Wiley

POSIX Threads

```
#include <pthread.h>
void * main(){
        ...
        iret1 = pthread_create( &thread1, NULL, print_message_function,
             (void*) message1);
        iret2 = pthread_create( &thread2, NULL, print_message_function,
             (void*) message2);
        ...
        pthread_join( thread1, NULL);
        pthread_join( thread2, NULL);
        ...
}
```

C++20 Threads

```
void print(int n, const std::string &str) {
  std::string msg = std::to_string(n) + " : " + str;
  std::cout << msg << std::endl;
}
int main() {
  std::vector<std::string> s = {``Parallel", ``Computing"};
  std::vector<std::thread> threads;
  for (int i = 0; i < s.size(); i++) {
    threads.push_back(std::thread(print, i, s[i]));
  }
  for (auto &th : threads) {
    th.join();
  }
  return 0;
}
```

⚠To start a thread in C++ we simply need to create a new thread object and pass the executing code to be called (i.e, a callable object) into the constructor of the object.

Java Threads

```
//creating a Java Thread subclass
public class MyClassThread extends Thread {
      public void run(){
      System.out.println("MyClass running");
   }
}
//To create and start the above thread:
MyClassThread t1 = new MyClassThread ();
t1.start();
//To wait for thread to complete:
t1.join();
```

Notes

We will mostly use Java thread as an example to cover the first half of this course. However, the concepts and techniques apply regardless of specific programming languages.

How to Stop a Thread

- Not recommend:
  - destroy()
  - stop() or std::terminate() in C++
  - stop(Throwable obj)
  - suspend()
- Recommend:
  - Interrupt()

⚠

Java 11 Removes stop() and destroy() Methods as they are “unsafe" or may leave the system in “undetermined" states.

What is Parallelism?

- Parallelism:
  - Average number of units of work that can be performed in parallel per unit time
  - Example: 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

1. 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

2. 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: Loop parallelism

Representative data parallelism: Loop 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

Remark

OpenMP is a widely used "shortcut" to achieve loop parallelism in C/C++. We will cover OpenMP later.

Task (Functional) Parallelism

- Program parts (*tasks*) can be executed in parallel
- 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

Representative Task Parallelism: Pipeline Parallelism

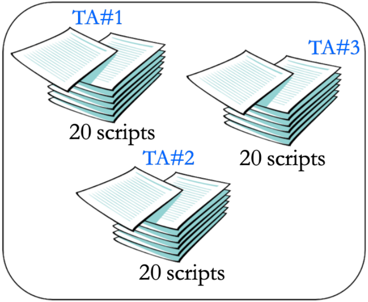
- If a program can be divided into multiple pieces without any dependency among them, we can achieve true task parallelism.
- If there are dependency among them, we can achieve *pipeline parallelism*.

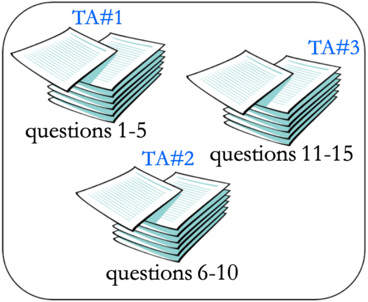
Remark

Note that, there are different ways to "split" a program, so we can end up with multiple alternative plans of task parallelism of the same program → those plans often lead to significantly different execution efficiency.

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?

Thread vs. Task

- Thread is the execution unit. (Think about a student)
- Task is the work unit. (Think about an assignment)

Thread and Task can be bundled

```java
class SummerThread extends Thread {
    int[] array;
    int lower;
    int upper;
    int sum = 0;
    public SummerThread(int[] array, int lower, int upper) {
        this.array = array;
        this.lower = lower;
        this.upper = upper;
    }
    public void run() {
        for (int i = lower; i < upper; i++) {
            sum += array[i];
        }
    }
    public int getSum() {
        return sum;
    }
}
```

Thread and Task can be bundled

```
public class BundledExample {
    public static void main(String[] args) throws Exception {
        int[] array = {1, 2, 3, 4, 5, 6, 7, 8, 9};
        SummerThread sthread1 = new SummerThread(array, 0, array.length
             / 2 + 1);
        SummerThread sthread2 = new SummerThread(array, array.length / 2
              + 1, array.length);
        sthread1.start();
        sthread2.start();
        try {
            sthread1.join();
            sthread2.join();
            System.out.println("The␣sum␣is:␣" + (sthread1.getSum() +
                 sthread2.getSum()));
        } catch (InterruptedException e) {
            System.out.println("A␣thread␣didn't␣finish!");
        }
    }
}
```

⚠

Thread is initialized with ‘tasks’ (i.e., summing of a subset of an array).

Thread and Task can be separated

```java
class Summer implements Runnable {
    int[] array;
    int lower;
    int upper;
    int sum = 0;
    public Summer(int[] array, int lower, int upper) {
        this.array = array;
        this.lower = lower;
        this.upper = upper;
    }
    public void run() {
        for (int i = lower; i < upper; i++) {
            sum += array[i];
        }
    }
    public int getSum() {
        return sum;
    }
}
```

Thread and Task can be separated

```
public class SeparatedExample {
    public static void main(String[] args) throws Exception {
        int[] array = {1, 2, 3, 4, 5, 6, 7, 8, 9};
        Summer summer1 = new Summer(array, 0, array.length / 2 + 1);
        Summer summer2 = new Summer(array, array.length / 2 + 1, array.
             length);
        Thread thread1 = new Thread(summer1);
        Thread thread2 = new Thread(summer2);
        thread1.start();
        thread2.start();
        try {
            thread1.join();
            thread2.join();
            System.out.println("The sum is: " + (summer1.getSum() +
                 summer2.getSum()));
        } catch (InterruptedException e) {
            System.out.println("A thread didn't finish!");
        }
    }
}
```

Program Parallelization

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

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

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)

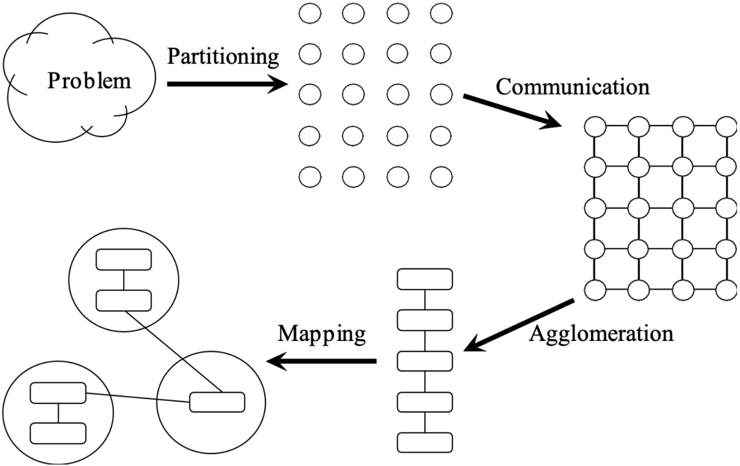
3. Agglomeration

- Decrease communication and development costs, while maintaining flexibility

4. Mapping

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

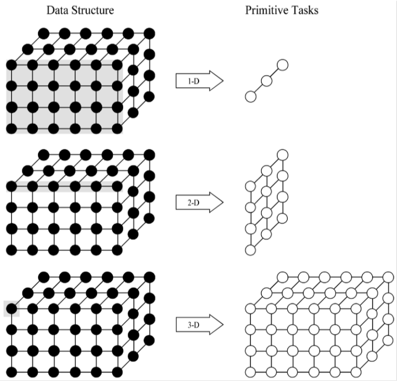
Foster's Design Methodology (cont'd)



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 Parallelism – Domain Decomposition:
    - Divide data into pieces of approximately equal size
    - Determine how to associate computations with the data

Example: Domain Decomposition



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)

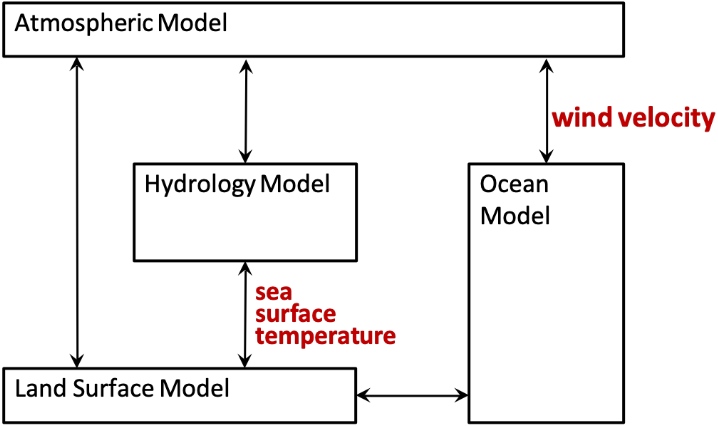
1. Partitioning

- 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 Parallelism – Domain Decomposition
  - Functional Parallelism – Functional Decomposition:
    - Divide computation into pieces
    - Determine how to associate data with the computations

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 Parallelism – Domain Decomposition
  - Functional Parallelism – Functional Decomposition:
    - Divide computation into pieces
    - Determine how to associate data with the computations

Example: Functional Decomposition



Computer Model of Climate

1. Partitioning: Partitioning Rules of Thumb

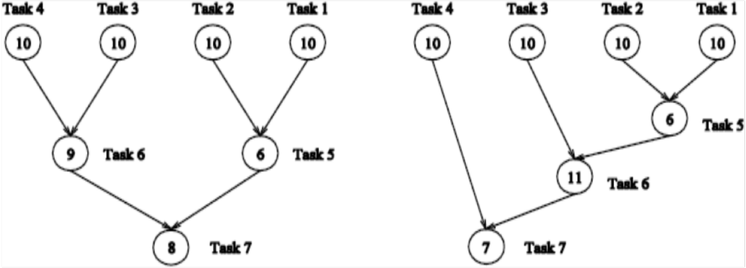
- At least 10x more primitive tasks than processors in target computer
  - Fine-grained primitive tasks → More effective usage of hardware resources
- Minimize redundant computations and redundant data storage (best to eliminate if any)
- Primitive tasks roughly of the same size
- Number of tasks as an increasing function of problem size

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

- Two different TDGs for the same program:

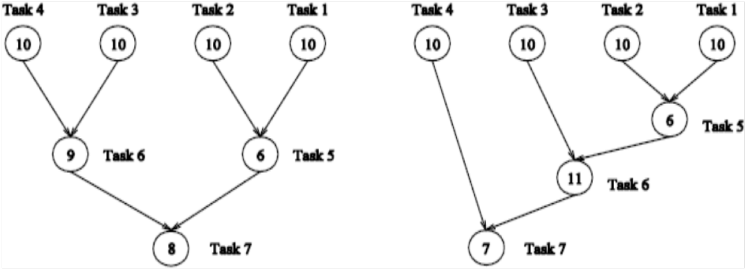


Critical Path Length: Maximum (slowest) completion time
Degree of concurrency = Total Work / Critical Path Length

- An indication of amount of work that can be done concurrently

Task Dependence Graph - Example

- Two different TDGs for the same program:



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**

2. Communication (Coordination)

- Tasks are intended to execute in parallel
  - but generally not executing independently
  - need to determine data passed among tasks
- Ideally, distribute and overlap computation and communication

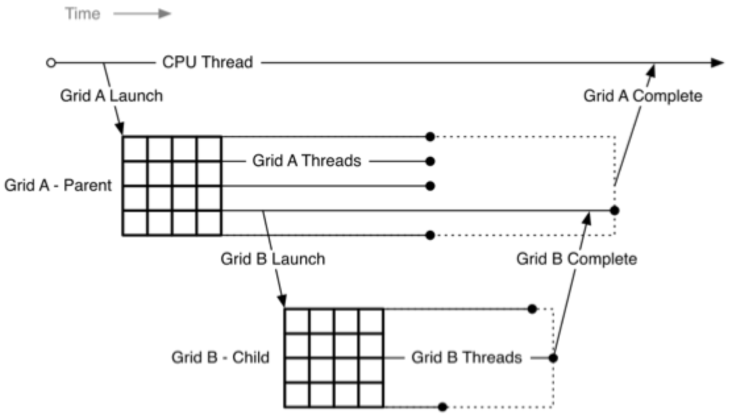
Coordination/Communication Models

- No communication
- Shared address space
- Message passing

Communication Models: No Communication

- 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

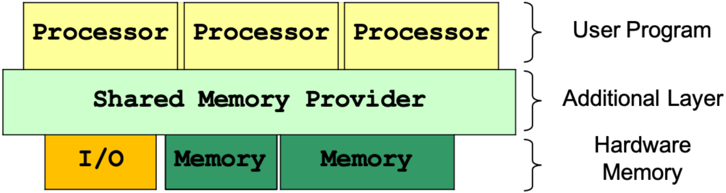
CUDA Execution



Communication Models: 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.

Shared-Memory Communication



Examples

Typical forms of shared-memory communication include “broadcast”, “reduction”, etc.

Communication Models: Message Passing

- Tasks operate within their own private address spaces
  - Tasks communicate by *explicitly sending/receiving messages*
- Popular software library: MPI (Mostly for C++), RMI & Sockets (Mostly for Java)
- 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

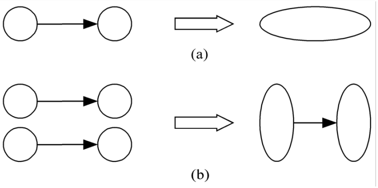
- No communication:
  - Programs perform same function on different data elements in a collection
- Shared address space:
  - All threads can read and write to all shared variables
  - Drawback: not all reads and writes have the same cost (and that cost is not apparent in program text), and may lead to implicit conflict (dangerous!)
- Message passing:
  - All communication occurs in the form of explicit messages

3. Agglomeration/Scheduling

- Combine tasks into larger tasks
  - Still, make sure Number of tasks $\geq$ number of cores
- Goals:
  - Improve performance (cost of task creation + communication)
  - Maintain scalability of program
  - Simplify programming

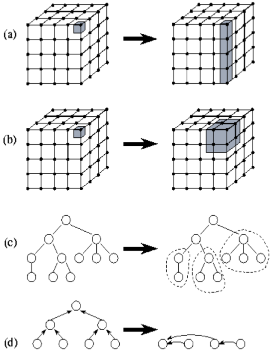
Motivation of Agglomeration

- Eliminate communication between primitive tasks agglomerated into consolidated task
- For example, 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

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

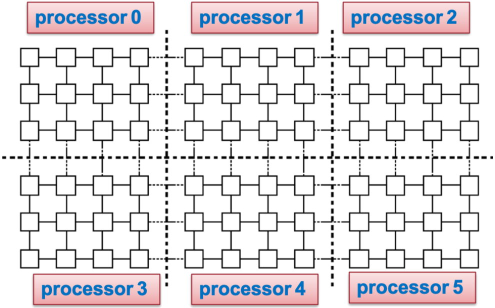


Figure: 12 x 6 Grid Problem

Remark

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

Mapping Example (Cont'd)

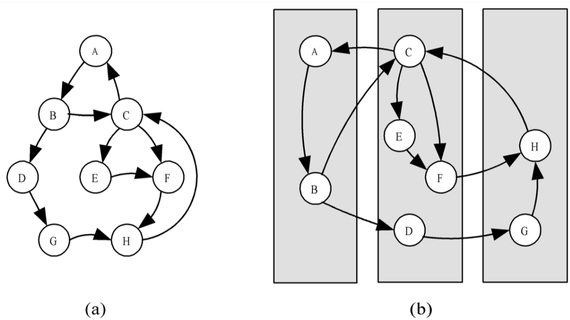


Figure: Mapping a Task Dependency Graph to Three Processors