Parallelizing Programs

- Goal: speed up programs using multiple processors/cores
(Sequential) Matrix Multiplication

double A[n][n], B[n][n], C[n][n] // assume n x n
for i = 0 to n-1
    for j = 0 to n-1
        double sum = 0.0
        for k = 0 to n-1
            sum += A[i][k] * B[k][j]
        C[i][j] = sum

Question: how can this program be parallelized?
Steps to parallelization

• First: find parallelism
  – Concerned about what can *legally* execute in parallel
  – At this stage, expose as much parallelism as possible
  – Partitioning can be based on data structures or by function

Note: other steps are architecture dependent
Finding Parallelism in Matrix Multiplication

• Can we parallelize the inner loop?
Finding Parallelism in Matrix Multiplication

• Can we parallelize the inner loop?
  – No, because \textit{sum} would be written concurrently
Finding Parallelism in Matrix Multiplication

- Can we parallelize the inner loop?
  - No, because $sum$ would be written concurrently

- Can we parallelize the outer loops?
Finding Parallelism in Matrix Multiplication

• Can we parallelize the inner loop?
  – No, because \textit{sum} would be written concurrently

• Can we parallelize the outer loops?
  – Yes, because the read and write sets are independent
    • Read set for process \((i,j)\) is \(sum, A[i][k=0:n-1], B[k=0:n-1][j]\)
    • Write set for process \((i,j)\) is \(sum, C[i][j]\)
  – Note: we have the option to parallelize just one of these loops
Terminology

• *co* statement: creates parallelism

  co i := 0 to n-1

  Body

  oc

• Meaning: *n* instances of body are created and executed concurrently until the end of the *co* (i.e., at the *oc*)

• Implementation: fork *n* threads, join them at the *oc*

• Can also be written *co b1 // b2 // … // bn oc*
Terminology

• *Process* statement: also creates parallelism
  
  \[
  \text{process } i := 0 \text{ to } n-1 \{ \\
  \quad \text{Body} \\
  \}
  \]

• Meaning: \( n \) instances of body are created and executed in parallel until the end of the *process*

• Implementation: fork \( n \) threads

• No synchronization at end

Need to understand what processes/threads are!
Processes

• History: OS had to coordinate many activities
  – Example: deal with multiple users (each running multiple programs), incoming network data, I/O interrupts

• Solution: Define a model that makes complexity easier to manage
  – Process (thread) model
What’s a process?

• Informally: program in execution
• Process encapsulates a physical processor
  – everything needed to run a program
    • code (“text”)
    • registers (PC, SP, general purpose)
    • stack
    • data (global variables or dynamically allocated)
    • files
• NOTE: a process is sequential
Examples of Processes

• Shell: creates a process to execute command
  lectura:> ls foo
  (shell creates process that executes “ls”)
  lectura:> ls foo & cat bar & more
  (shell creates three processes, one per command)

• OS: creates a process to manage printer
  – process executes code such as:
    wait for data to come into system buffer
    move data to printer buffer
Creating a Process

• Must somehow specify code, data, files, stack, registers

• Ex: UNIX
  – Use the fork() system call to create a process
  – Makes an exact duplicate of the current process
    • (returns 0 to indicate child process)
  – Typically exec() is run on the child

We will not be doing this (systems programming)
Example of Three Processes

OS switches between the three processes ("multiprogramming")
Review: Run-time Stack

A(int x) {
    int y = x;
    if (x == 0) return;
    else return A(y-1) + 1;
}

B() {
    int z;
    A(1);
}

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>z</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Decomposing a Process

• Process: everything needed to run a program

• Consists of:
  – Thread(s)
  – Address space
Thread

• Sequential stream of execution
• More concretely:
  – program counter (PC)
  – register set
  – stack
• Sometimes called lightweight process
Address Space

• **Consists of:**
  – code
  – contents of main memory (data)
  – open files

• **Address space can have > 1 thread**
  – threads share memory, files
  – threads have separate stacks, register set
One Thread, One Address Space

code

data

files

main thread
(stack, registers)

address space
Many Threads, One Address Space

code
data
files
main thread
thread 0
thread 1
thread 2
thread 3
thread 4
thread 5

address space

each thread: stack, regs
Thread States

- **Ready**
  - eligible to run, but another thread is running

- **Running**
  - using CPU

- **Blocked**
  - waiting for something to happen
Thread State Graph

- **Ready**
  - Scheduled to **Running**
  - Pre-empted (timer) to **Blocked**

- **Blocked**
  - I/O event or wait for thread

- **Running**
  - I/O complete or thread we were waiting for is done

- **Pre-empted (timer)**
  - Scheduled to **Ready**
Scheduler

• Decides which thread to run
  – (from ready list only)
• Chooses from some algorithm
• From our point of view, the scheduler is something we cannot control
  – We have no idea which thread will be run, and our programs must not depend on execution order of two ready threads
Context Switching

• Switching between 2 threads
  – change PC to current instruction of new thread
    • might need to restart old thread in the future
  – must save exact state of first thread

• What must be saved?
  – registers (including PC and SP)
  – what about stack itself?
Multiple Threads, One Machine

Machine

PC
SP
R1
R2

Code
Data
Files

Stack
Stack

PC, SP, R1, R2
PC, SP, R1, R2

Address Space
Thread
Thread
Why Save Registers?

- code for Thread 0
  foo( )
  x := x+1
  x := x*2

  Assembly code:
  R1 := R1 + 1  /* !! */
  R1 := R1 * 2

  Suppose context switch occurs after line “!!”

- code for Thread 1
  bar( )
  y := y+2
  y := y-3

  Assembly code:
  R1 := R1 + 2
  R1 := R1 - 3
Matrix Multiplication, $n^2$ threads

double A[n][n], B[n][n], C[n][n]  // assume n x n

co i = 0 to n-1  
  co j = 0 to n-1  
    double sum = 0.0
    for k = 0 to n-1
      sum += A[i][k] * B[k][j]
      C[i][j] = sum
  
We already argued the two outer “for” loops were parallelizable
Steps to parallelization

• Second: control granularity
  – Must trade off advantages/disadvantages of fine-granularity
    • Advantages: better load balancing, better scalability
    • Disadvantages: process/thread overhead and communication
  – Combine small processes into larger ones to coarsen granularity
    • Try to keep the load balanced
Matrix Multiplication, \( n \) threads

double A[n][n], B[n][n], C[n][n]  // assume \( n \times n \)

co i = 0 to n-1  
  
for j = 0 to n-1  
  
  double sum = 0.0
  
  for k = 0 to n-1
  
    sum += A[i][k] * B[k][j]

  C[i][j] = sum

This is plenty of parallelization

if the number of cores is \( \leq n \)
Matrix Multiplication, $p$ threads

double A[n][n], B[n][n], C[n][n]  // assume n x n

do t = 0 to p-1  
  
    startrow = t * n / p; endrow = (t+1) * n/p - 1

for i = startrow to endrow
  
    for j = 0 to n-1  
      
      double sum = 0.0

      for k = 0 to n-1
        
        sum += A[i][k] * B[k][j]

      C[i][j] = sum

Steps to parallelization

• Third: distribute computation and data
  – Assign which processor does which computation
    • The co statement does not do this
  – If memory is distributed, decide which processor stores which data (why is this?)
    • Data can be replicated also
  – Goals: minimize communication and balance the computational workload
    • Often conflicting goals
Steps to parallelization

- Fourth: synchronize and/or communicate
  - If shared-memory machine, synchronize
    - Both mutual exclusion and sequence control
    - Locks, semaphores, condition variables, barriers, reductions
  - If distributed-memory machine, communicate
    - Message passing
    - Usually communication involves implicit synchronization
Parallel Matrix Multiplication---Distributed-Memory Version

```java
process worker [i = 0 to p-1] {
    double A[n][n], B[n][n], C[n][n] // wasting space!
    startrow = i * n / p; endrow = (i+1) * n/p – 1
    if (i == 0) {
        for j = 1 to p-1 {
            sr = j * n / p; er = (j+1) * n/p – 1
            send A[sr:er][0:n-1], B[0:n-1][0:n-1] to process j
        }
    } else {
        receive A[startrow:endrow][0:n-1], B[0:n-1][0:n-1] from 0
    }
}
```
Parallel Matrix Multiplication---Distributed-Memory Version

for i = startrow to endrow
   for j = 0 to n-1 {
      double sum = 0.0
      for k = 0 to n-1
         sum += A[i][k] * B[k][j]
      C[i][j] = sum
   }
   // here, need to send my piece back to master
   // how do we do this?
} // end of process statement
Adaptive Quadrature: Sequential Program

double f() {
    ....
}
double area(a, b)
    c := (a+b)/2
    compute area of each half and area of whole
    if (close)
        return area of whole
    else
        return area(a,c) + area (c,b)
Adaptive Quadrature: Recursive Program

double f() {
    ....
}

double area(a, b)
    c := (a+b)/2
    compute area of each half and area of whole
    if (close)
        return area of whole
    else
        leftArea = area(a,c) // rightArea = area (c,b) oc
        return leftArea + rightArea
Challenge with Adaptive Quadrature

• For efficiency, must control granularity (step 2)
  – Without such control, granularity will be too fine
  – Can stop thread creation after “enough” threads created
    • Hard in general, as do not want cores idle either
  – Thread implementation can perform work stealing
    • Idle cores take a thread and execute that thread, but care must be taken to avoid synchronization problems and/or efficiency problems
Steps to parallelization

• Fifth: assign processors to tasks (only if using task and data parallelism)
  – Must also know dependencies between tasks
  – Usually task parallelism used if limits of data parallelism are reached
Steps to parallelization

• Sixth: parallelism-specific optimizations
  – Examples: message aggregation, overlapping communication with computation
Steps to parallelization

• Seventh: acceleration
  – Find parts of code that can run on GPU/Cell/etc., and optimize those parts
  – Difficult and time consuming
    • But may be quite worth it