Concurrent Programming

Session 7-1: Parallel Algorithms

Computer Engineering Department
Iran University of Science and Technology
Tehran, Iran

Instructor: Hadi Salimi
Presenter: Arash Khosraviani
Distributed Systems Lab.
Computer Engineering Department,
Iran University of Science and Technology,
hsalimi@iust.ac.ir

Motivation

- Computer world move toward parallelism
- Parallelism tools is improving these days
  - **Hardware:** Multi Core CPUs, GPUs, Computer Network …
  - **OS**: Support Multitasking and Multithreading, Network Management Capabilities, …
  - **Software Development Tools**: PThread, Cilk++, OpenMP, TBB, MPI, …
- So Algorithms should be paralleled too!

Problem Decomposition

- The first step in developing a parallel algorithm is to decompose the problem into tasks that can be executed concurrently
- A given problem may be decomposed into tasks in many different ways
- In general, the number of tasks in a decomposition exceeds the number of processing elements available

Decomposition Techniques

- So how does one decompose a task into various subtasks?
  - ***recursive decomposition***
  - ***data decomposition***
  - ***exploratory decomposition***
  - ***speculative decomposition***

Recursive Decomposition (R.D.)

- Generally suited to problems that are solved using the divide-and-conquer strategy
- A given problem is first decomposed into a set of sub-problems.
- These sub-problems are recursively decomposed further until a desired granularity is reached

Warm up: Fibonacci

```
FIB(n)
if n < 2
    then return n
return FIB(n− 1) + FIB(n− 2)
```

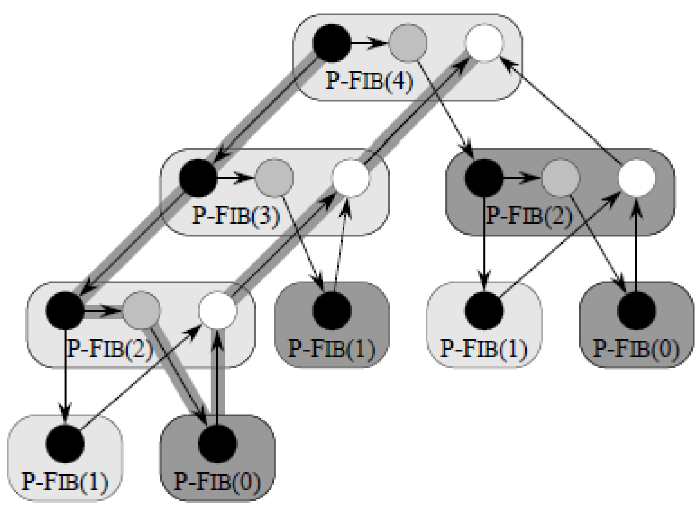
- Can Anybody make it Parallel?
  It's so simple!

Warm up: Parallel Fibonacci

```
P-FIB(n)
if n < 2
   then return n
X = P-FIB(n− 1)      ← one processing element do this
Y = P-FIB(n− 2)      ← the other do this
return X + Y
```

- We can make it parallel simply using Pthreads, Cilk++, OpenMP

Warm up: Parallel Fibonacci (cont.)



R.D. Capable Algorithms

- We are able to decompose the recursive algorithms easily, cause they are decomposed by default!
  - Quick Sort and Merge Sort
  - Eight Queen
  - Hanoi tower
  - …
- But we can make parallel some other part of these algorithms too.

Parallel Merge Sort

```
Merge-Sort(A, beg, end)
if beg < end
    then mid = ⌊(beg + end)/2⌋
Merge-Sort(A, beg, mid)
Merge-Sort(A, mid + 1, end)
P-Merge(A, beg, mid, mid+1, end)
return
```

Each processing element do one of these.

Parallel Merge

```
P-MERGE(T, p1, r1, p2, r2, A, p3)
n1 = r1 - p1 + 1
n2 = r2 - p2 + 1
if n1 < n2
      exchange p1 with p2
      exchange r1 with r2
      exchange n1 with n2
if n1 == 0
      return
else q1 = [(p1 + r1)/2]
      q2 = BINARY-SEARCH(T [q1], T, p2, r2)
      q3 = p3 + (q1 - p1) + (q2 – p2)
      A[q3] = T[q1]
      P-MERGE(T, p1, q1 – 1, p2, q2 – 1, A, p3)
      P-MERGE(T, q1 + 1, r1, q2, r2, A, q3 + 1)
```

Sorted s ***Sorted*** ***Result array A[p3…r3]***

Parallel Merge

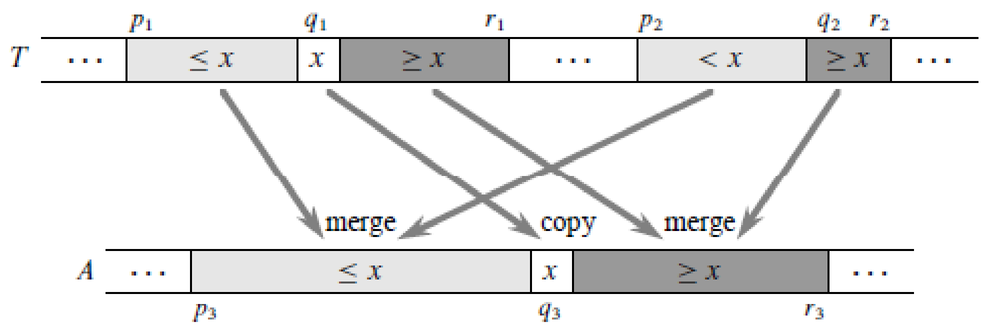
```
P-MERGE(T, p1, r1, p2, r2, A, p3)
n1 = r1 - p1 + 1
n2 = r2 - p2 + 1
if n1 < n2
      exchange p1 with p2
      exchange r1 with r2
      exchange n1 with n2
if n1 == 0
      return
else q1 = [(p1 + r1)/2]
      q2 = BINARY-SEARCH(T [q1], T, p2, r2)
      q3 = p3 + (q1 - p1) + (q2 – p2)
      A[q3] = T[q1]
      P-MERGE(T, p1, q1 – 1, p2, q2 – 1, A, p3)
      P-MERGE(T, q1 + 1, r1, q2, r2, A, q3 + 1)
```

without loss of generality, larger array should be first

Both empty?

Each processing element do one of these.

Parallel Merge (cont.)



Another R.D. Example

- The problem of finding the minimum number in a given list (or indeed any other associative operation such as sum, AND, etc.) can be fashioned as a divide-and-conquer algorithm.

Serial Minimum

```
MIN (A, n)
min = A[1];
for i = 2 to n
    if A[i] < min
        min = A[i];
return min;
```

Parallel Minimum

```
REC-MIN (A, n)
if n == 1
    min = A[1]  ;
else
    lmin = REC-MIN(A[1], n/2)
    rmin = REC-MIN(A[n/2], n - n/2 )
if lmin  < rmin
    min = lmin
else
    min = rmin
return min
```

Parallel Minimum (cont.)

- As you see we can rewrite loop to have recursive decomposition
- Now if have n/2 processing element, we can have a more efficient algorithm

Parallel Minimum (cont.)

```
P-MIN (A, n)
p = index of this processor
gap = 1
for step = 1 to log(n)
      if (p % gap) == 0
            first = A[2*p – 1]
            second = A[2*p – 1 + gap]
            if first < second
                  A[2*p – 1] = first
            else
                  A[2*p – 1] = second
      gap = gap * 2
return A[1]
```

If this processor needs to execute in this step

Each processing element, individually do one iteration of this loop

Data Decomposition

- Identify the data on which computations are performed
- Partition this data across various tasks
- This partitioning induces a decomposition of the problem
- Data can be partitioned in various ways. this critically impacts performance of a parallel algorithm

Serial Matrix-Vector Multiplication

```
MAT-VEC (A, X, Y, n)
for i = 1 to n
    for j = 1 to n
        Y[i] = Y[i] + A[i][j]. x[j]
return Y
```

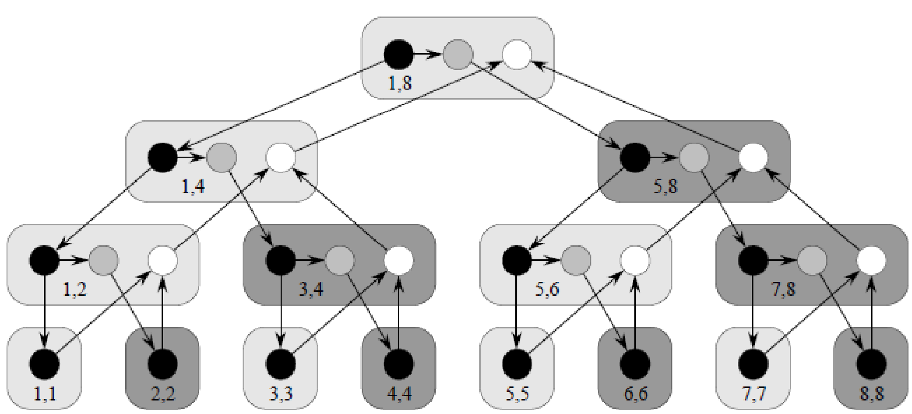
For this problem, we should decompose the matrix data. of course we can do others!

Parallel Matrix-Vector Multiplication

```
P-MAT-VEC (A, X, Y, n, beg, end)
if beg == end
    for j = 1 to n
        Y[beg] = Y[beg] + A[beg][j].X[j]
else mid = [(beg + end)/2]
P-MAT-VEC(A, X, Y, n, beg, mid)
P-MAT-VEC(A, X, Y, n, mid+1, end)
return
```

Each processing element do one of these.

Parallel Matrix-Vector Multiplication (cont.)



Matrix Multiplication

- Consider the problem of multiplying two ***n*** x ***n*** matrices ***A*** and ***B*** to yield matrix ***C***. The output matrix ***C*** can be partitioned into four tasks as follows

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

$$C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$$

$$C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$$

$$C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$$

$$C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$$

Parallel Matrix Multiplication

```
P-MATRIX-MULTIPLY(C, A, B)
n = A.rows
if n == 1
    c[1][1] = a[1][1].b[1][1]
else
    let T be a new n*n matrix
    partition A, B, C and T into n/2*n/2
    submatrices
P-MATRIX-MULTIPLY(C[1][1],A[1][1],B[1][1])
P-MATRIX-MULTIPLY(C[1][2],A[1][1],B[1][2])
P-MATRIX-MULTIPLY(C[2][1],A[2][1],B[1][1])
```

Decompose Matrix data

Parallel Matrix Multiplication (cont.)

```
P-MATRIX-MULTIPLY(C[2][2],A[2][1],B[1][2])
P-MATRIX-MULTIPLY(T[1][1],A[1][2],B[2][1])
P-MATRIX-MULTIPLY(T[1][2],A[1][2],B[2][2])
P-MATRIX-MULTIPLY(T[2][1],A[2][2],B[2][1])
P-MATRIX-MULTIPLY(T[2][2],A[2][2],B[2][2])
for i = 1 to n
    for j = 1 to n
        C[i][j] = C[i][j] + T[i][j]
```

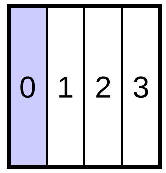
Add two temporary data

Cilk++ Matrix Multiplication

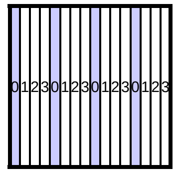
- Despite making Matrix Multiplication algorithm parallel, if we use Cilk++ it's more simple to do it
- Of course it is important how we decompose the matrices

```
P-MATRIX-MULTIPLY (A, B)
n = A.rows
let C be a new n*n matrix
parallel for i = 1 to n
    parallel for j = 1 to n
        C[i][j] = 0
        for k = 1 to n
            C[i][j] = C[i][j] + A[i][k]*B[k][j]
return C
```

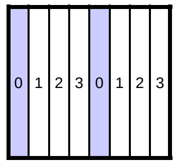
Different Parallel Data Layouts for Matrices



1) 1D Column Blocked Layout

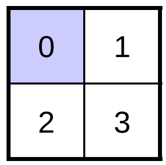


2) 1D Column Cyclic Layout

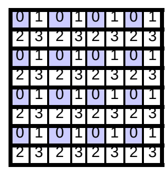


3) 1D Column Block Cyclic Layout

4) Row versions of the previous layouts



5) 2D Row and Column Blocked Layout



6) 2D Row and Column Block Cyclic Layout

Different Parallel Data Layouts for Matrices

1) 1D Co

Layout

Which one is the best?

It depends on your system architecture
It depends on your compiler
It depends on your CPU
It depends on your Cache size
It depends on …

ayouts

3) 1D Co

Block

5) 2D Row and Column Blocked Layout

Exploratory Decomposition

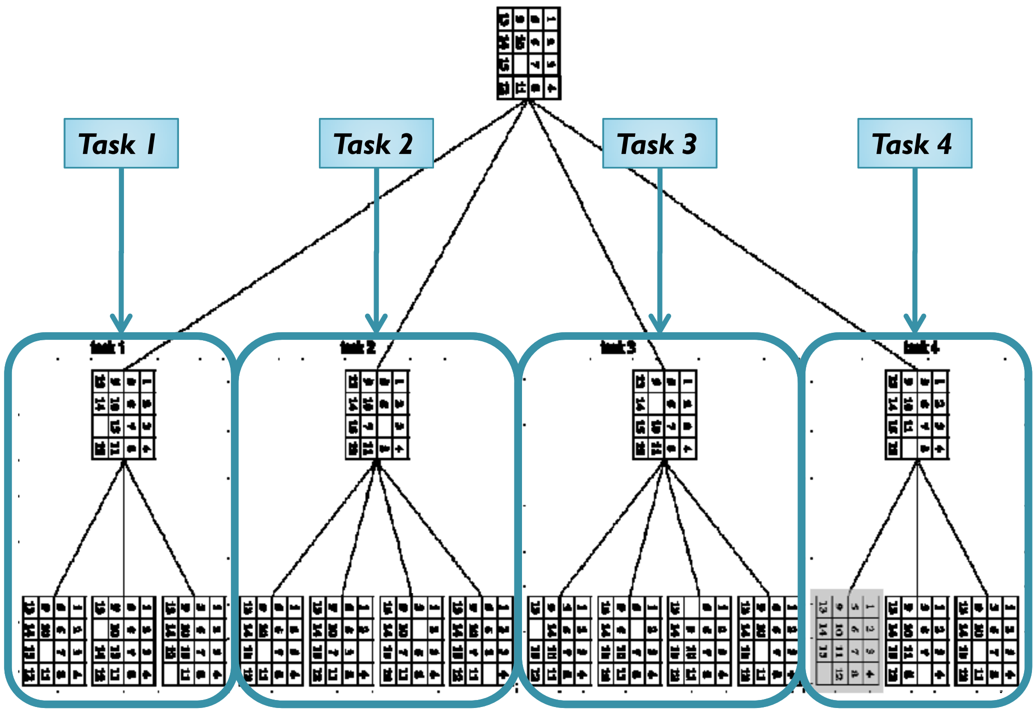
- In many cases, the decomposition of the problem goes hand-in-hand with its execution.
- These problems typically involve the exploration (search) of a state space of solutions.
- Problems in this class include a variety of discrete optimization problems (0/1 integer programming, QAP, etc.), theorem proving, game playing, etc.

15 puzzle

- A simple application of exploratory decomposition is in the solution to a 15 puzzle (a tile puzzle)

- The state space can be explored by generating various successor states of the current state and to view them as independent tasks

15 puzzle (cont.)



Speculative Decomposition

- In some applications, dependencies between tasks are not known a-priori.
- For such applications, it is impossible to identify independent tasks.
- There are generally two approaches to dealing with such applications: conservative approaches, which identify independent tasks only when they are guaranteed to not have dependencies, and, optimistic approaches, which schedule tasks even when they may potentially be erroneous.
- Conservative approaches may yield little concurrency and optimistic approaches may require roll-back mechanism in the case of an error.

Speculative Decomposition: Example

- A classic example of speculative decomposition is in discrete event simulation.
- The central data structure in a discrete event simulation is a time-ordered event list.
- Events are extracted precisely in time order, processed, and if required, resulting events are inserted back into the event list.
- Therefore, an optimistic scheduling of other events will have to be rolled back.