Introduction to Multithreaded Algorithms

CCOM5050: Design and Analysis of Algorithms

Chapter VII Selected Topics

April 23, 2012

E. Orozco
Multithreaded Algorithms

Learning objectives: at the end of this chapter students are expected to

1. understand the importance of parallel computation.

2. identify the abstract model of dynamic multithreading programming as a concurrency platform.

3. identify dynamic multithreading programming as an extension of sequential programming.

4. identify work, span, speedup, and parallelism as metrics for analyzing multithreading algorithms.

5. compute and interpret work, span, speedup, and parallelism of a given multithreaded algorithm on a particular input.

6. analyze simple multithreaded algorithms.

7. compute the running time of simple multithreaded algorithms.

8. represent a multithreaded computation as a directed acyclic graph.

9. identify concurrent for loops.

10. identify nested parallelism.

11. write simple parallel algorithms using spawn, sync, parallel for, and return control structures.
Multithreaded Algorithms

• Key concepts
  – Abstract model of multithreaded computation
  – Parallel control:
    • Spawn
    • Sync
    • return
  – Nested parallelism
  – Concurrent for loops
  – Performance metrics:
    • work
    • Span
    • speedup
    • Parallelism
  – Directed acyclic graph (dag)
  – Race condition
Dynamic multithreaded programming (DMP)

- **Thread: software abstraction of a virtual processor**
  - Threads share a common memory.
  - Maintains its own stack and program counter.
  - Execute code independently of the others.
  - OS loads a thread onto a processor, executes it and switch to another one when needed.
Dynamic multithreaded programming (DMP)

- **Thread: software abstraction of a virtual processor**
  - Threads share a common memory.
  - Maintains its own stack and program counter.
  - Execute code independently of the others.
  - OS loads a thread onto a processor, executes it and switch to another one when needed.

- **Concurrency platform:**
  - Coordinates,
  - Schedule, ➔ parallel computing resources
  - Manages
Dynamic multithreaded programming (DMP)

- **Thread**: software abstraction of a virtual processor
  - Threads share a common memory.
  - Maintains its own stack and program counter.
  - Execute code independently of the others.
  - OS loads a thread onto a processor, executes it and switch to another one when needed.

- **Concurrency platform**:
  - Coordinates,
  - Schedule, \(\Rightarrow\) parallel computing resources
  - Manages

- **DMP allows the programmer to specify parallelism without worrying about communication protocols and load balancing.**
Dynamic multithreaded programming (DMP)

- **Thread**: software abstraction of a virtual processor
  - Threads share a common memory.
  - Maintains its own stack and program counter.
  - Execute code independently of the others.
  - OS loads a thread onto a processor, executes it and switch to another one when needed.

- **Concurrency platform**:
  - Coordinates,
  - Schedule, ➔ parallel computing resources
  - Manages

- **DMP allows the programmer to specify parallelism without worrying about communication protocols and load balancing.**

- **Concurrency platform for DMP**:
  - Contains a scheduler: automatic load-balance computation.
  - Support nested parallelism: allows a subroutine to be spawned.
  - Contains parallel loops: the iteration of the loop can execute concurrently.
Dynamic multithreaded programming (DMP)

• Programmer needs to specify only the logical parallelism within a computation:
  
  – concurrency platform schedules and load-balance the computation among the threads.
Reasons for choosing Multithreading as a parallel computation model

• It is a \textit{simple extension} of sequential programming:
  – A parallel algorithm can be described by adding the keywords \textit{parallel}, \textit{spawn}, and \textit{sync}.

• It provides a theoretically clean way of quantifying parallelism by the notions of \textit{work} and \textit{span}.

• \textit{Divide-and-conquer} paradigm: Nested parallelism.

• Multithreading platforms:
  – Cilk
  – OpenMP
  – Task Parallel Library
  – Threading Building Blocks [Intel TBB]

• Grand challenge problems are being solved with multithreading algorithms.

• Hardware:
  – Super computers: Cray
  – Multi-core.
Basics of dynamic multithreading

**Example:** Multithreading the computation of Fibonacci numbers

\[ F_0 = 0, \]
\[ F_1 = 1, \]
\[ F_i = F_{i-1} + F_{i-2} \quad \text{for } i \geq 2. \]

**Sequential version**

\[ \text{FIB}(n) \]
1. if \( n \leq 1 \) return \( n \)
2. return \( n \)
3. else
4. \( x = \text{FIB}(n-1) \)
5. \( y = \text{FIB}(n-2) \)
6. return \( x + y \)
Basics of dynamic multithreading

Example: Recursive computation of the Fibonacci numbers.

The tree of recursive calls to compute FIB(6).
Basics of dynamic multithreading

**Example**: Recursive computation of the Fibonacci numbers.

**Running time analysis:**

\[ T(n) = T(n-1) + T(n-2) + \Theta(1). \]

**Exercise**: Show that \( T(n) = \Theta( \Phi^n) \),

where \( \Phi = (1 + \sqrt{5})/2 \) is the **golden ratio**.

Poorly efficient algorithm...

But illustrates the key concepts for analyzing multithreaded algorithms.
Basics of dynamic multithreading

Example: Recursive computation of the Fibonacci numbers.

Multithreading the computation of Fibonacci numbers
Recursive calls to FIB(n-1) and FIB(n-2) in lines 4 and 5 are independent to each other.

```python
FIB(n)
1 if n ≤ 1
2 return n
3 else
4 x = FIB(n - 1)
5 y = FIB(n - 2)
6 return x + y
```

Task dependency graph for FIB( n )
Basis of dynamic multithreading

Example: Multithreading the computation of Fibonacci numbers

P-FIB( n )
1. if n ≤ 1
2. return n
3. else
4. x = spawn P-FIB( n – 1 )
5. y = P-FIB( n – 2 )
6. sync
7. return x + y

Task dependency graph for FIB( n )
Basics of dynamic multithreading

*Nested parallelism*
occurs when the keyword `spawn` proceeds a procedure call.
A model of multithreaded execution

- **Multithreaded computation**: set of runtime instructions executed by a processor on behalf of a multithreaded program.

- A directed acyclic graph $G=(V,E)$, called a **computation dag**, is used to represent multithreaded computation.
  - Vertices in $V$ represent instructions.
  - Edges in $E$ represent dependencies between instructions. Edge $(u, v)$ in $E$ means instruction $u$ must execute before instruction $v$.
    - **Continuation edge** $(u, u')$, drawn horizontally, connects a strand $u$ to its successor $u'$ within the same procedure instance.
    - **Spawn edge** $(u, v)$, pointing downward, means strand $u$ spawns strand $v$.
    - **Call edges**, pointing downward, represent normal procedure calls.
    - **Return edge** $(u, x)$, pointing upward, means that a strand $u$ returns to its calling procedure and $x$ is the strand immediately following the next **sync** in the calling procedure.
A model of multithreaded execution

- **Multithreaded computation**: set of runtime instructions executed by a processor on behalf of a multithreaded program.

- A directed acyclic graph $G=(V,E)$, called a *computation dag*, is used to represent multithreaded computation.
  - Vertices in $V$ represent instructions.
  - Edges in $E$ represent dependencies between instructions:
    - Edge $(u, v) \in E$ means instruction $u$ must execute before instruction $v$.
    - A **strand** represents a chain of instructions that does not contain any parallel control (spawn, sync, or return from a spawn).
    - A computation starts with a single *initial strand* and ends with a single *final strand*.
    - If there is a directed path from strand $u$ to strand $v$, $u$ and $v$ are **logically in series**. Otherwise, they are **logically in parallel**.

- A **multithreaded computation** is a dag of strands embedded in a tree of procedure instances.
A model of multithreaded execution

**Example**: Recursive computation of the Fibonacci numbers.

The tree of recursive calls to compute FIB(4).
A model of multithreaded execution

- Circles represent strands.
- Black circle: base case or the part of the procedure up to the spawn of \( \text{P-FIB}(n-1) \) in line 4.
- Gray circle: part of the procedure that calls \( \text{P-FIB}(n-2) \) in line 5 up to the \text{sync} in line 6, where it suspends until the spawn of \( \text{P-FIB}(n-1) \) returns.
- White circle: part of the procedure after the \text{sync} where it sums \( x \) and \( y \) up to the point where it returns the result.
- Rectangles contain strands belonging to the same procedure.
  - Light blue for spawned procedures.
  - Gray for called procedures.

Directed acyclic graph representing \( \text{P-FIB}(4) \)
A model of multithreaded execution

- Circles represent strands.
- , base case or the part of the procedure up to the spawn of P-FIB(n-1) in line 4.
- , part of the procedure that calls P_FIB(n-2) in line 5 up to the sync in line 6, where it suspends until the spawn of P-FIB(n-1) returns.
- , part of the procedure after the sync where it sums x and y up to the point where it returns the result.
- Rectangles contain strands belonging to the same procedure
  - , for spawned procedures
  - , for called procedures.

Directed acyclic graph representing P-FIB(4)
An ideal parallel computer for multithreaded computation

• Consists of
  – **Set of homogenous processors**: all processors in the machine have equal computer power.
  
  – **Sequentially consistent shared memory**: shared memory produces the same results as if at each step, exactly one instruction from one of the processors is executed.

• Performance assumption:
  – **No schedule overhead**: scheduling cost is ignored.
Performance measures for multithreaded algorithms

- **Work**: total time to execute the entire computation on a single processor
  
  - \( \text{Work} = \Sigma t_i \), where \( t_i \) is the time taken by strand \( i \).
  
  - \( \text{Work} = \# \text{vertices in the dag if each strand takes unit time.} \)

- **Span**: longest time to execute the strands along any path in the dag.
  
  - \( \text{Span} = \# \text{vertices on a longest or critical path in the dag if each strand takes unit time.} \)
Performance measures for multithreaded algorithms

- **For the computation dag for P-FIB(4):**
  - Work = 17 time units
  - Span = 8 time units

Directed acyclic graph representing P-FIB(4)
Performance measures for multithreaded algorithms

• The running time of a multithreaded computation: 
  \[ T_p = \text{depends on work, span, } p, \text{ scheduler} \]

  where \( p \) = \# processor available.

• \( T_1 = \text{work} \) on a single processor.

• \( T_\infty \): running time if we could run each strand on its own processor (i.e., unlimited number of processors)

• \( \text{Span} = T_\infty \)
Performance measures for multithreaded algorithms

- **Speedup** = $T_1 / T_p$
  - Speedup $\leq p$
  - Linear speedup $= \Theta(p)$
  - Perfect linear speedup $= p$.

- **Parallelism** = $T_1 / T_\infty$
  
  - **Interpretations of parallelism:**
    - As an upper bound:
      - maximum possible speedup that can be achieved on any number of processors.
    - As a limit on the speedup:
      - provides a limit on the possibility of attaining perfect linear speedup.
Performance measures for multithreaded algorithms

Example: Computation of P-FIB(4)

- Assume each strand takes unit time.

- Work = 17,

- Span = 8,

- Parallelism = $T_1 / T_\infty = 17 / 8 = 2.125$.

  - Interpretation:
    - No matter how many processors are used for the computation of P-FIB(4), the speedup will not be more than two.
Performance measures for multithreaded algorithms

Class exercise:
Draw the computation dag that results from executing P-FIB(5). Assuming that each strand in the computation takes unit time, what are the work, span, and parallelism of the computation?
Analysis of multithreaded algorithms

• Analysis of the work:
  – running time of a serial algorithm.
    » For P-FIB\((n)\), \(T_1(n) = \Theta( \Phi^n)\)
Analysis of multithreaded algorithms

• Analysis of the work:
  – running time of a serial algorithm.
    » For P-FIB(n), $T_1(n) = \Theta(\Phi^n)$

• Analysis of the span:

  \[
  \text{Work : } T_1(A\cup B) = T_1(A) + T_1(B) \\
  \text{Span : } T_\infty(A\cup B) = T_\infty(A) + T_\infty(B)
  \]

  \[
  \text{Subcomputations in series} \\
  \text{Subcomputations in parallel}
  \]

  \[
  \text{Work : } T_1(A\cup B) = T_1(A) + T_1(B) \\
  \text{Span : } T_\infty(A\cup B) = \max(T_\infty(A), T_\infty(B))
  \]
Analysis of multithreaded algorithms

• Example: Analysis of P-FIB(n)

\[ T_1(n) \text{ of P-FIB}(n) = \Theta(\Phi^n), \]

Span :
\[ T_\infty(n) = \max(T_\infty(n-1), T_\infty(n-2)) + \Theta(1) \]
\[ = T_\infty(n-1) + \Theta(1) \]
\[ = \Theta(n). \]
Analysis of multithreaded algorithms

- Example: Analysis of P-FIB(n):
  
  \[ T_1(n) \text{ of P-FIB(n)} = \Theta(\Phi^n), \]

  \[
  \begin{align*}
  \text{Span} & : T_{\infty}(n) = \max(T_{\infty}(n-1), T_{\infty}(n-2)) + \Theta(1) \\
  & = T_{\infty}(n-1) + \Theta(1) \\
  & = \Theta(n).
  \end{align*}
  \]

  Parallelism = \( T_1(n) / T_{\infty}(n) = \Theta(\Phi^n / n) \)
Parallel Loops

Example: matrix-vector multiplication.
Let $A = (a_{ij})$ be an $nxn$ matrix and let $x = (x_i)$ be an $n$-vector. The product of $A$ times $x$ is other $n$-vector $y = (y_i)$ defined by

$$y_i = \sum_{j=1}^{n} a_{ij}x_j$$

Serial code for matrix-vector multiplication

```
MAT-VEC-SEQ( A, x )
1 n = A.rows
2 Let y be a new vector of length n
3 for i = 1 to n
4 y_i = 0
5 for i = 1 to n
6 for j = 1 to n
7 y_i = y_i + a_ij x_j
8 return y
```

Parallel code for matrix-vector multiplication

```
MAT-VEC( A, x )
1 n = A.rows
2 Let y be a new vector of length n
3 parallel for i = 1 to n
4 y_i = 0
5 parallel for i = 1 to n
6 for j = 1 to n
7 y_i = y_i + a_ij x_j
8 return y
```
Parallel Loops

Example: *matrix-vector multiplication.*
Let $A = (a_{ij})$ be an $n \times n$ matrix and let $x = (x_i)$ be an $n$-vector. The product of $A$ times $x$ is other $n$-vector $y = (y_i)$ defined by

$$y_i = \sum_{j=1}^{n} a_{ij}x_j$$

MAT-VEC( $A$, $x$ )
1  $n = A$.rows
2  Let $y$ be a new vector of length $n$
3  parallel for $i = 1$ to $n$
4    $y_i = 0$
5  parallel for $i = 1$ to $n$
6    for $j = 1$ to $n$
7    $y_i = y_i + a_{ij}x_j$
8  return $y$

Parallel code for matrix-vector multiplication
Parallel Loops

**Example:** *matrix-vector multiplication.*
A divide-and-conquer parallel implementation of MAT-VEC( A, x )

MAT-VEC-MAIN-LOOP( A, x, y, n, i, i’ )
1 if i == i’
2 for j = 1 to n
3 y_i = y_j + a_{ij} x_j
4 else mid = floor((i+ i’)/2)
5 spawn MAT-VEC-MAIN-LOOP( A, x, y, n, i, mid)
6 MAT-VEC-MAIN-LOOP( A, x, y, n, mid+ 1, i’ )
7 sync
Parallel Loops

**Example:** matrix-vector multiplication.
A divide-and-conquer parallel implementation of MAT-VEC( A, x )

DAG representing the computation of MAT-VEC-LOOP( A, x, y, 1, 8)
Parallel Loops

Example: *matrix-vector multiplication.*

Running time analysis of MAT-VEC( A, x )

Analysis of work:
\[ T_1(n) = \Theta(n^2) + h(n) = \Theta(n^2) + \Theta(n) = \Theta(n^2) \]

where \( h(n) \) is the overhead for recursive spawning in implementing the parallel loops.

Why is it that \( h(n) = \Theta(n^2) \) ?

Analysis of span:
\[ T_\infty (n) = \Theta(\lg n) + \max_{1 \leq i \leq n} \text{iter}_\infty(i) \]
Parallel Loops

Example: matrix-vector multiplication.

Running time analysis of MAT-VEC( A, x )

Analysis of work:

\[ T_1(n) = \Theta(n^2) + h(n) = \Theta(n^2) + \Theta(n) = \Theta(n^2) \]

where \( h(n) \) is the overhead for recursive spawning in implementing the parallel loops.

Why is it that \( h(n) = \Theta(n^2) \)?

Analysis of span:

\[ T_\infty(n) = \Theta(\lg n) + \max_{1 \leq i \leq n} \text{iter}_\infty(i) \]

\[ = \Theta(\lg n) + \Theta(n) \]

\[ = \Theta(n) \]

Analysis of parallelism:

\[ parallelism = \Theta(n^2) / \Theta(n) = \Theta(n) \]
Multithreaded Algorithms Assessment

1. **Understanding the multithreaded model**: Students will be given a sequential algorithm *such as* \(\text{FIB}(\ n)\) and will be asked to answer questions such as:
   a. What parts of the algorithm present opportunities for parallelization?
   b. What type of parallelism, if any, is suitable for this algorithm (nested parallelism, loop parallelism)? Why?
   c. Write a multithreaded version of the algorithm.

5. **Understanding the performance of multithreaded algorithms**: Students will be given a multithreaded algorithm *such as* \(\text{P-FIB}(\ n)\) or \(\text{MAT-VEC-MAIN-LOOP}(\ A,\ x,\ y,\ i,\ i')\) and will be asked to answer questions such as:
   a. Identify whether this algorithm is or not race-condition free.
   b. Draw a dag for \(\text{MAT-VEC-MAIN-LOOP}(\ A,\ x,\ y,\ n,\ 1,\ 4)\)
   c. What is the work?
   d. What is the span?
   e. What is the speedup?
   f. What is the parallelism?

3. **Understanding the performance of multithreaded algorithms**: Students are asked to consider the \(\text{P-TRANSPOSE}(\ A)\) procedure to transpose an \(n \times n\) matrix \(A\) in place and are asked to analyze the work, span, and parallelism of this algorithm.

\[
\text{P-TRANSPOSE}(\ A) \\
1 \quad n = A.\ rows \\
2 \quad \text{parallel for} \ j = 2 \ \text{to} \ n \\
3 \quad \text{parallel for} \ i = 1 \ \text{to} \ j - 1 \\
4 \quad \text{exchange} \ a_{ij} \ \text{with} \ a_{ji}
\]
Multithreaded Algorithms Assessment

Students will be given the opportunity to work in teams (2 to 3 students) in one of the following projects:

• **Design of a multithreaded algorithm**: Students will be asked to provide pseudo code for an *efficient* multithreaded algorithm that transposes an $nxn$ matrix in place by using divide-and-conquer to divide the matrix recursively into four $n/2 \times n/2$ submatrices. They should also provide an analysis of their algorithm.

• **Parallel-loop to nested-loop conversion**: Students are given a multithreaded algorithm for performing pairwise addition on $n$-element arrays $A[1..n]$ and $B[1..n]$, storing the sum in $C[1..n]$ and are asked to rewrite the parallel loop in SUM-ARRAYS procedure using nested parallelism (**spawn** and **sync**). Also, they have to provide an analysis of the parallelism of the implementation.

```
SUM-ARRAYS( A, B, C)
1  parallel for i = 1 to A.length
2    C[i] = A[i] + B[i]
```
Multithreaded Algorithms Assessment

3. **Sequential to parallel conversion**: Students are given the LU-DECOMPOSITION procedure and are asked to provide pseudocode for a corresponding multithreaded version. They are asked to make it as parallel as possible, and analyze its work, span, and parallelism.

```
LU-DECOMPOSITION( A )
1    n = A.rows
2    Let L and U be new nxn matrices
3    Initialize U with 0s below the diagonal
4    Initialize L with 1s on the diagonal and 0s above the diagonal
5    for k = 1 to n
6        u_{kk} = a_{kk}
7        for i = k + 1 to n
8            l_{ik} = a_{ik} / u_{kk}    // l_{ik} holds v_i
9            u_{ki} = a_{ki}        // u_{ki} holds w_i^T
10       for i = k + 1 to n
11          for j = k + 1 to n
12              a_{ij} = a_{ij} - l_{ik} u_{kj}
13    return L and U
```