CS4460 Advanced Algorithms
Batch 08, L4S2

Lecture 12
Multithreaded Algorithms – Part 2

N. H. N. D. de Silva
Dept. of Computer Science & Eng
University of Moratuwa
Outline: Multithreaded Algorithms

• Part 1: last session
  • Introduction, Dynamic Multithreading
  • Model for Multithreaded Execution
  • Performance Measures
  • Analyzing Multithreaded Algorithms

• Part 2: this session
  – Parallel Loops
  – Race Conditions
  – Examples
Parallel Loops

- Uses the "parallel" keyword preceding the "for" keyword
  - Iterations of loops can be concurrent

- Example
  - Multiplying matrix $A = (a_{ij})$ by vector $x = (x_j)$ to get a vector $y = (y_i)$

\[
\begin{align*}
\text{MAT-VEC}(A, x) \\
1 & \quad n = A.\text{rows} \\
2 & \quad \text{let } y \text{ be a new vector of length } n \\
3 & \quad \text{parallel for } i = 1 \text{ to } n \\
4 & \quad \quad y_i = 0 \\
5 & \quad \text{parallel for } i = 1 \text{ to } n \\
6 & \quad \quad \text{for } j = 1 \text{ to } n \\
7 & \quad \quad \quad y_i = y_i + a_{ij}x_j \\
8 & \quad \text{return } y \\
\end{align*}
\]

\[
y_i = \sum_{j=1}^{n} a_{ij}x_j
\]
Parallel Loops

• “parallel for” keywords in lines 3 and 5
  – indicate that the iterations of the respective loops may be run concurrently

• A compiler can implement each parallel for loop as a divide-and-conquer subroutine using nested parallelism
  – For e.g., the parallel for loop in lines 5–7 can be implemented with the call MAT-VEC-MAIN-LOOP (A, x, y, n,1, n)
```plaintext
MAT-VEC-MAIN-LOOP(A, x, y, n, i, i')
1   if i == i'
2     for j = 1 to n
3         y_i = y_i + a_ij x_j
4   else mid = [(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 for” loop in lines 5–7 of MAT-VEC (A, x) can be implemented with the call MAT-VEC-MAIN-LOOP (A, x, y, n, i, i'), where the compiler produces the auxiliary subroutine MAT-VEC-MAIN-LOOP as above.
• It recursively spawns the first half of the iterations of the loop to execute in parallel with the second half of the iterations and then executes a sync
Computation DAG for MAT-VEC-MAIN-LOOP (A, x, y, 8, 1, 8)

Figure 27.4
Computation DAG for
MAT-VEC-MAIN-LOOP (A, x, y, 8, 1, 8)

Figure 27.4 A dag representing the computation of MAT-VEC-MAIN-LOOP(A, x, y, 8, 1, 8). The two numbers within each rounded rectangle give the values of the last two parameters (i and i' in the procedure header) in the invocation (spawn or call) of the procedure. The black circles represent strands corresponding to either the base case or the part of the procedure up to the spawn of MAT-VEC-MAIN-LOOP in line 5; the shaded circles represent strands corresponding to the part of the procedure that calls MAT-VEC-MAIN-LOOP in line 6 up to the sync in line 7, where it suspends until the spawned subroutine in line 5 returns; and the white circles represent strands corresponding to the (negligible) part of the procedure after the sync up to the point where it returns.

Note: As before, each group of strands belonging to the same procedure is surrounded by a rounded rectangle, lightly shaded for spawned procedures and heavily shaded for called procedures. Spawn edges and call edges point downward, continuation edges point horizontally to the right, and return edges point upward.
Analysis: Work $T_1(n)$

• Work $T_1(n)$ of MAT-VEC on an $n \times n$ matrix
  – Compute the running time of its serialization
  – Replace the parallel for with normal for loops
  – $T_1(n) = \Theta(n^2)$

• Ignores the overhead of recursive spawning in implementing the parallel loops, but the overhead does not increase the work asymptotically

• In some multithreading platforms, leaves of recursion may be coarsened by executing several iterations in a single leaf → reduces overhead
Analysis: Span $T_{\infty}(n)$

- Span $T_{\infty}(n)$ of MAT-VEC on an $n \times n$ matrix
  - Depth of recursion logarithmic in $n$
  - For a parallel loop with $n$ iterations, the $i$-th iteration has span $\text{iter}_{\infty}(i)$
  - $T_{\infty}(n) = \Theta(\lg n) + \max \text{iter}_{\infty}(i)$
  - Span of doubly nested loops lines 5-7 is $\Theta(n)$
    - Each iteration of outer parallel-for contains $n$ iterations of inner (serial) for loop → this dominates
  - Overall span is $T_{\infty}(n) = \Theta(n)$

What is the parallelism?
Race Conditions

• A multithreaded algorithm is
  – **deterministic** if it always does the same thing on the same input (no matter the scheduling)
  – **non-deterministic** if its behavior can vary from run to run

• If an algorithm contains a “**determinacy race**”, it will not be deterministic even though it was intended to be so
Race Conditions

• A “determinacy race” occurs when two logically parallel instructions access the same memory location and at least one of them performs a write.

• Example

```plaintext
RACE-EXAMPLE()
1  x = 0
2  parallel for i = 1 to 2
3   x = x + 1
4  print x
```

After initializing `x` to 0 in line 1, RACE-EXAMPLE creates two parallel strands, each of which increments `x` in line 3.

It seems that RACE-EXAMPLE should always print the value 2 (serialization does so), it could instead print the value 1.
Figure 27.5 Illustration of the determinacy race in RACE-EXAMPLE. (a) A computation dag showing the dependencies among individual instructions. The processor registers are $r_1$ and $r_2$. Instructions unrelated to the race, such as the implementation of loop control, are omitted. (b) An execution sequence that elicits the bug, showing the values of $x$ in memory and registers $r_1$ and $r_2$ for each step in the execution sequence.
Race Conditions

• The problem with determinacy races
  – Most orderings produce correct results
    • E.g., In RACE-EXAMPLE, the execution order of steps \(1,2,3,7,4,5,6,8\) or \(1,4,5,6,2,3,7,8\)

• For our purposes, we will ensure strands that operate in parallel are independent
  – i.e., they have no determinacy races
  – E.g., in parallel-for, all the iterations should be independent
Matrix Multiplication

• Requirement: Compute $C = A \cdot B$ where all are square matrices of $n \times n$

1. Sequential Algorithms
   a) Obvious/basic algorithm
   b) Simple divide-and-conquer approach
   c) Strassen’s method

2. Multithreaded Algorithms
   – For each of (a), (b) and (c) above
1(a) Basic Matrix Multiplication

**SQUARE-MAT-MULT** \((A, B, n)\)

let \(C\) be a new \(n \times n\) matrix

for \(i = 1\) to \(n\)

for \(j = 1\) to \(n\)

\[c_{ij} = 0\]

for \(k = 1\) to \(n\)

\[c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}\]

return \(C\)

Takes \(\Theta(n^3)\) time
1(b) Simple Divide-\&-Conquer

- Assume $n$ is a power of 2
- Partition $A$, $B$, $C$ into four $n/2 \times n/2$ matrices

\[
A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}.
\]

Rewrite $C = A \cdot B$ as
\[
\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix},
\]
giving the four equations
\[
\begin{align*}
C_{11} &= A_{11} \cdot B_{11} + A_{12} \cdot B_{21}, \\
C_{12} &= A_{11} \cdot B_{12} + A_{12} \cdot B_{22}, \\
C_{21} &= A_{21} \cdot B_{11} + A_{22} \cdot B_{21}, \\
C_{22} &= A_{21} \cdot B_{12} + A_{22} \cdot B_{22}.
\end{align*}
\]

Use these four equations to get a divide-and-conquer algorithm
1(b) Simple Divide-\&-Conquer

\texttt{REC-MAT-MULT}(A, B, n)
let $C$ be a new $n \times n$ matrix
if $n == 1$
  $c_{11} = a_{11} \cdot b_{11}$
else partition $A$, $B$, and $C$ into $n/2 \times n/2$ submatrices
  $C_{11} = \text{REC-MAT-MULT}(A_{11}, B_{11}) + \text{REC-MAT-MULT}(A_{12}, B_{21})$
  $C_{12} = \text{REC-MAT-MULT}(A_{11}, B_{12}) + \text{REC-MAT-MULT}(A_{12}, B_{22})$
  $C_{21} = \text{REC-MAT-MULT}(A_{21}, B_{11}) + \text{REC-MAT-MULT}(A_{22}, B_{21})$
  $C_{22} = \text{REC-MAT-MULT}(A_{21}, B_{12}) + \text{REC-MAT-MULT}(A_{22}, B_{22})$
return $C$

- Conquering makes 8 recursive calls $\rightarrow 8 \ T(n/2)$
- Combining takes $\Theta(n^2)$ time to add $n/2 \times n/2$ matrices four times
1(b) Simple Divide-&-Conquer

Recurrence is

\[ T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 8T(n/2) + \Theta(n^2) & \text{if } n > 1. \end{cases} \]

Can use master method to show that it has solution \( T(n) = \Theta(n^3) \).
Asymptotically, no better than the obvious method.

*Constant factors and recurrences:* When setting up recurrences, can absorb constant factors into asymptotic notation, but cannot absorb a constant number of subproblems. Although we absorb the 4 additions of \( n/2 \times n/2 \) matrices into the \( \Theta(n^2) \) time, we cannot lose the 8 in front of the \( T(n/2) \) term. If we absorb the constant number of subproblems, then the recursion tree would not be “bushy” and would instead just be a linear chain.
1(c) Strassen’s Method

• Idea
  – Make recursion tree less bushy
  – Perform only 7 recursive multiplications of $n/2 \times n/2$ matrices
  – Cost of this is several new additions of $n/2 \times n/2$ matrices (but only a constant number)
  – Can still absorb the constant factor for matrix additions into the $\Theta(n^2)$ term
1(c) Strassen’s Method

The algorithm:

1. As in the recursive method, partition each of the matrices into four $n/2 \times n/2$ submatrices. Time: $\Theta(1)$.
2. Create 10 matrices $S_1, S_2, \ldots, S_{10}$. Each is $n/2 \times n/2$ and is the sum or difference of two matrices created in previous step. Time: $\Theta(n^2)$ to create all 10 matrices.
3. Recursively compute 7 matrix products $P_1, P_2, \ldots, P_7$, each $n/2 \times n/2$.
4. Compute $n/2 \times n/2$ submatrices of $C$ by adding and subtracting various combinations of the $P_i$. Time: $\Theta(n^2)$.

Recurrence will be:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 7T(n/2) + \Theta(n^2) & \text{if } n > 1. \end{cases}$$

By the master method, solution is $T(n) = \Theta(n^{\log_7 7})$.

Homework: Read the details on Strassen’s method.
2(a) Parallel Basic Algorithm

P-SQUARE-MATRIX-Multiply \((A, B)\)

1. \(n = A.\text{rows}\)
2. let \(C\) be a new \(n \times n\) matrix
3. parallel for \(i = 1\) to \(n\)
4. parallel for \(j = 1\) to \(n\)
5. \(c_{ij} = 0\)
6. for \(k = 1\) to \(n\)
7. \(c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}\)
8. return \(C\)
2(a) Parallel Basic Algorithm

To analyze this algorithm, observe that since the serialization of the algorithm is just SQUARE-MATRIX-MULTIPLY, the work is therefore simply $T_1(n) = \Theta(n^3)$, the same as the running time of SQUARE-MATRIX-MULTIPLY. The span is $T_\infty(n) = \Theta(n)$, because it follows a path down the tree of recursion for the parallel for loop starting in line 3, then down the tree of recursion for the parallel for loop starting in line 4, and then executes all $n$ iterations of the ordinary for loop starting in line 6, resulting in a total span of $\Theta(lg n) + \Theta(lg n) + \Theta(n) = \Theta(n)$. 
2(b) Parallel Divide-&-Conquer

\begin{verbatim}
P-MATRIX-MULTIPLY-RECURSIVE(C, A, B)
1 n = A.rows
2 if n == 1
3 \hspace{1em} c_{11} = a_{11} b_{11}
4 else let T be a new n x n matrix
5 \hspace{1em} partition A, B, C, and T into n/2 x n/2 submatrices
6 \hspace{2em} A_{11}, A_{12}, A_{21}, A_{22}; B_{11}, B_{12}, B_{21}, B_{22}; C_{11}, C_{12}, C_{21}, C_{22};
7 \hspace{2em} and T_{11}, T_{12}, T_{21}, T_{22}; respectively
8 spawn P-MATRIX-MULTIPLY-RECURSIVE(C_{11}, A_{11}, B_{11})
9 spawn P-MATRIX-MULTIPLY-RECURSIVE(C_{12}, A_{11}, B_{12})
10 spawn P-MATRIX-MULTIPLY-RECURSIVE(C_{21}, A_{21}, B_{11})
11 spawn P-MATRIX-MULTIPLY-RECURSIVE(C_{22}, A_{21}, B_{12})
12 spawn P-MATRIX-MULTIPLY-RECURSIVE(T_{11}, A_{12}, B_{21})
13 spawn P-MATRIX-MULTIPLY-RECURSIVE(T_{12}, A_{12}, B_{22})
14 spawn P-MATRIX-MULTIPLY-RECURSIVE(T_{21}, A_{22}, B_{21})
15 spawn P-MATRIX-MULTIPLY-RECURSIVE(T_{22}, A_{22}, B_{22})
16 sync
17 parallel for i = 1 to n
18 \hspace{1em} parallel for j = 1 to n
19 \hspace{2em} c_{ij} = c_{ij} + t_{ij}
\end{verbatim}
2(b) Parallel Divide-&-Conquer

**Work**

We first analyze the work $M_1(n)$ of the P-MATRIX-MULTIPLY-RECURSIVE procedure, echoing the serial running-time analysis of its progenitor SQUARE-MATRIX-MULTIPLY-RECURSIVE. In the recursive case, we partition in $\Theta(1)$ time, perform eight recursive multiplications of $n/2 \times n/2$ matrices, and finish up with the $\Theta(n^2)$ work from adding two $n \times n$ matrices. Thus, the recurrence for the work $M_1(n)$ is

$$M_1(n) = 8M_1(n/2) + \Theta(n^2)$$

$$= \Theta(n^3)$$
2(b) Parallel Divide-&-Conquer

To determine the span $M_\infty(n)$ of P-MATRIX-MULTIPLY-RECURSIVE, we first observe that the span for partitioning is $\Theta(1)$, which is dominated by the $\Theta(\lg n)$ span of the doubly nested **parallel for** loops in lines 15–17. Because the eight parallel recursive calls all execute on matrices of the same size, the maximum span for any recursive call is just the span of any one. Hence, the recurrence for the span $M_\infty(n)$ of P-MATRIX-MULTIPLY-RECURSIVE is

$$M_\infty(n) = M_\infty(n/2) + \Theta(\lg n).$$

(27.7)

This recurrence does not fall under any of the cases of the master theorem, but it does meet the condition of Exercise 4.6-2. By Exercise 4.6-2, therefore, the solution to recurrence (27.7) is $M_\infty(n) = \Theta(\lg^2 n)$.

Parallelism = $M_1(n) / M_\infty(n) = \Theta(n^3 / \lg^2 n) \rightarrow \text{very high}$
2(c) Parallel Strassen’s Method

• Use same outline and nested parallelism
  1. Divide input matrices A, B and output matrix C into \( n/2 \times n/2 \) sub-matrices
    • Work = span = \( \Theta(1) \)

2. Create 10 matrices \( S_1, S_2, \ldots, S_{10} \) each of which is \( n/2 \times n/2 \) and is the sum or difference of 2 matrices created in Step 1.
   • Using doubly nested \textit{parallel-for} loops, we get work = \( \Theta(n^2) \) and span = \( \Theta(\lg n) \)
2(c) Parallel Strassen’s Method

3. Using the sub-matrices created in Step 1 and the 10 matrices created in Step 2, recursively spawn the computation of seven $n/2 \times n/2$ matrix products $P_1, P_2, \ldots, P_7$

4. Compute the desired $C_{11}, C_{12}, C_{21}, C_{22}$ of the result matrix $C$ by adding and subtracting various combinations of the $P_i$ matrices

• Using doubly nested `parallel-for` loops, can compute all four sub-matrices with, work = $\Theta(n^2)$ and span = $\Theta(\lg n)$
2(c) Parallel Strassen’s Method

To analyze this algorithm, we first observe that since the serialization is the same as the original serial algorithm, the work is just the running time of the serialization, namely, $\Theta(n^{1 \lg 2})$. As for P-MATRIX-MULTIPLY-RECURSIVE, we can devise a recurrence for the span. In this case, seven recursive calls execute in parallel, but since they all operate on matrices of the same size, we obtain the same recurrence (27.7) as we did for P-MATRIX-MULTIPLY-RECURSIVE, which has solution $\Theta(\lg^2 n)$. Thus, the parallelism of multithreaded Strassen’s method is $\Theta(n^{1 \lg 2} / \lg^2 n)$, which is high, though slightly less than the parallelism of P-MATRIX-MULTIPLY-RECURSIVE.
References

• The lecture slides are based on the slides prepared by Prof. Sanath Jayasena for this class in previous years.

• Mainly: CLRS book, 3e
  – Part VII: Selected Topics
  – Chapter 27: Multithreaded Algorithms

• Other resources (on LMS)
  – Document “A Minicourse on Dynamic Multithreaded Algorithms”
  – Slide sets: Cilk and Design and Analysis of Dynamic Multithreaded Algorithms and Strassen’s Matrix Multiplication Algorithm
Conclusion

• We discussed Part 2 of Multithreaded Algorithms
  – Parallel loops
  – Race conditions
  – Examples: Multithreaded matrix-vector and matrix-matrix multiplication

• Conclusion of all lectures