Today:
− Multithreaded Algs.
Reading Assignments

• Today’s class:
  – Chapter 27.1-27.2

• Reading assignment for next class:
  – Chapter 27.3

• Announcement: Exam #2 on Tuesday, April 1
  – Will cover greedy algorithms, amortized analysis
  – HW 6-9
Scheduling

• The performance depends not just on the work and span. Additionally, the **strands must be scheduled efficiently**.

• The strands must be mapped to static threads, and the operating system schedules the threads on the processors themselves.

• The scheduler must schedule the computation with no advance knowledge of when the strands will be spawned or when they will complete; it must operate **online**.
Greedy Scheduler

• We will assume a greedy scheduler in our analysis, since this keeps things simple. A greedy scheduler assigns as many strands to processors as possible in each time step.

• On P processors, if at least P strands are ready to execute during a time step, then we say that the step is a complete step; otherwise we say that it is an incomplete step.
Greedy Scheduler Theorem

• On an ideal parallel computer with P processors, a greedy scheduler executes a multithreaded computation with work $T_1$ and span $T_\infty$ in time:

$$T_P \leq \frac{T_1}{P} + T_\infty$$

• Given the fact the best we can hope for on P processors is $T_P = \frac{T_1}{P}$ by the work law, and $T_P = T_\infty$ by the span law, the sum of these two gives the lower bounds
Proof (1/3)

• Let’s consider the complete steps. In each complete step, the P processors perform a total of P work.

• Seeking a contradiction, we assume that the number of complete steps exceeds $\frac{T_1}{P}$. Then the total work of the complete steps is at least

\[
P\left(\left\lfloor \frac{T_1}{P} \right\rfloor + 1 \right) = P\left\lfloor \frac{T_1}{P} \right\rfloor + P = T_1 - (T_1 \mod P) + P > T_1
\]

• Since this exceeds the total work required by the computation, this is impossible.
Proof (2/3)

• Now consider an **incomplete step**. Let $G$ be the DAG representing the entire computation. W.l.o.g. assume that each strand takes unit time (otherwise replace longer strands by a chain of unit-time strands).

• Let $G'$ be the subgraph of $G$ that has **yet to be executed** at the start of the incomplete step, and let $G''$ be the subgraph **remaining to be executed** after the completion of the incomplete step.
Proof (3/3)

• A longest path in a DAG must necessarily start at a vertex with in-degree 0. Since an incomplete step of a greedy scheduler executes all strands with in-degree 0 in $G'$, the length of the longest path in $G''$ must be 1 less than the length of the longest path in $G'$.

• Put differently, an incomplete step decreases the span of the unexecuted DAG by 1. Thus, the number of incomplete steps is at most $T_\infty$.

• Since each step is either complete or incomplete, the theorem follows. □
Corollary

• The running time of any multithreaded computation scheduled by a greedy scheduler on an ideal parallel computer with P processors is within a factor of 2 of optimal.

• Proof: Let $T_P^*$ be the running time produced by an optimal scheduler. Let $T_1$ be the work and $T_\infty$ be the span of the computation. We know from work and span laws that:

$$T_P^* \geq \max(T_1/P, T_\infty).$$

• By the theorem,

$$T_P \leq \frac{T_1}{P} + T_\infty \leq 2 \max\left(\frac{T_1}{P}, T_\infty\right) \leq 2T_P^*.$$
Slackness

- The parallel **slackness** of a multithreaded computation executed on an ideal parallel computer with \( P \) processors is the ratio of parallelism by \( P \).

\[
\text{Slackness} = \left( \frac{T_1}{T_\infty} \right) / P
\]

- If the slackness is less than 1, we cannot hope to achieve a linear speedup.
Achieving Near-Perfect Speedup

• Let $T_P$ be the running time of a multithreaded computation produced by a greedy scheduler on an ideal computer with $P$ processors. Let $T_1$ be the work and $T_\infty$ be the span of the computation. If the slackness is big, $P <\ll (T_1 / T_\infty)$, then $T_P$ is approximately $T_1 / P$ [i.e., near-perfect speedup]

• Proof: If $P <\ll (T_1 / T_\infty)$, then $T_\infty <\ll T_1 / P$. Thus, by the theorem, $T_P \leq T_1 / P + T_\infty \approx T_1 / P$. By the work law, $T_P \geq T_1 / P$. Hence, $T_P \approx T_1 / P$, as claimed.

Here, “big” means slackness of 10 – i.e., at least 10 times more parallelism than processors
Analyzing multithreaded algs.

• Analyzing work is no different than for serial algorithms

• Analyzing span is more involved...

  – Two computations in series means their spans add

  \[
  \begin{align*}
  \text{Work: } T_1(A \cup B) &= T_1(A) + T_1(B) \\
  \text{Span: } T_\infty(A \cup B) &= T_\infty(A) + T_\infty(B)
  \end{align*}
  \]

  – Two computations in parallel means you take maximum of individual spans

  \[
  \begin{align*}
  \text{Work: } T_1(A \cup B) &= T_1(A) + T_1(B) \\
  \text{Span: } T_\infty(A \cup B) &= \max(T_\infty(A), T_\infty(B))
  \end{align*}
  \]
Analyzing Parallel Fibonacci Computation

- Parallel algorithm to compute Fibonacci numbers:

\[
\text{P-FIB}(n) = \begin{cases} 
  n & \text{if } n \leq 1 \\
  \text{spawn P-FIB}(n-1); & \text{else}
  \text{spawn P-FIB}(n-2); & \text{// parallel execution}
  \text{sync}; & \text{// wait for results of x and y}
  x + y & \text{return}
\end{cases}
\]
Work of Fibonacci

• We want to know the work and span of the Fibonacci computation, so that we can compute the parallelism (work/span) of the computation.

• The work $T_1$ is straightforward, since it amounts to computing the running time of the serialized algorithm:

$$T_1 = T(n-1) + T(n-2) + \Theta(1)$$

$$= \Theta \left( \left( \frac{1+\sqrt{5}}{2} \right)^n \right)$$
Span of Fibonacci

• Recall that the span $T_\infty$ is the longest path in the computational DAG. Since $\text{FIB}(n)$ spawns $\text{FIB}(n-1)$ and $\text{FIB}(n-2)$, we have:

$$T_\infty(n) = \max(T_\infty(n - 1), T_\infty(n - 2)) + \Theta(1)$$

$$= T_\infty(n - 1) + \Theta(1)$$

$$= \Theta(n)$$
Parallelism of Fibonacci

• The parallelism of the Fibonacci computation is:

\[ \frac{T_1(n)}{T_\infty(n)} = \Theta \left( \left( \frac{1+\sqrt{5}}{2} \right)^n / n \right) \]

which grows dramatically as \( n \) gets large.

• Therefore, even on the largest parallel computers, a modest value of \( n \) suffices to achieve near perfect linear speedup, since we have considerable parallel slackness.
Parallel Loops

• Consider multiplying \( n \times n \) matrix \( A \) by an \( n \)-vector \( x \):

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

• Can be calculated by computing all entries of \( y \) in parallel:

\[
\text{MAT-VEC}(A, x)
\]

\[
n = A.\text{rows}
\]

let \( y \) be a new vector of length \( n \)

\[
\text{parallel for } i = 1 \text{ to } n
\]

\[
y_i = 0
\]

\[
\text{parallel for } i = 1 \text{ to } n
\]

\[
\text{for } j = 1 \text{ to } n
\]

\[
y_i = y_i + a_{ij} x_j
\]

return \( y \)

Here, \textit{parallel for} is implemented by the compiler as a divide-and-conquer subroutine using nested parallelism.
Parallel Loops – Implementation

\[ \text{MAT-Vec}(A, x) \]
\[ n = A.\text{rows} \]
let \( y \) be a new vector of length \( n \)
parallel for \( i = 1 \) to \( n \)
\[ y_i = 0 \]
parallel for \( i = 1 \) to \( n \)
for \( j = 1 \) to \( n \)
\[ y_i = y_i + a_{ij}x_j \]
return \( y \)

Here, parallel for is implemented by the compiler as a divide-and-conquer subroutine using nested parallelism

\[ \text{MAT-Vec-Main-Loop}(A, x, y, n, i, i') \]
if \( i == i' \)
for \( j = 1 \) to \( n \)
\[ y_i = y_i + a_{ij}x_j \]
else
\[ \text{mid} = \lfloor (i + i')/2 \rfloor \]
spawn \[ \text{MAT-Vec-Main-Loop}(A, x, y, n, i, \text{mid}) \]
\[ \text{MAT-Vec-Main-Loop}(A, x, y, n, \text{mid} + 1, i') \]
sync
Parallel Loops – Implementation

\textbf{MAT-VEC}(A, x) \\
\textit{n} = A.\textit{rows} \\
let \textit{y} be a new vector of length \textit{n} \\
\textbf{parallel for} \textit{i} = 1 \textbf{ to } \textit{n} \\
\hspace{1em} \textit{y}_{\textit{i}} = 0 \\
\textbf{parallel for} \textit{i} = 1 \textbf{ to } \textit{n} \\
\hspace{2em} \textbf{for} \textit{j} = 1 \textbf{ to } \textit{n} \\
\hspace{3em} \textit{y}_{\textit{i}} = \textit{y}_{\textit{i}} + a_{\textit{ij}}x_{\textit{j}} \\
\textbf{return} \textit{y}

Here, \textit{parallel for} is implemented by the compiler as a divide-and-conquer subroutine using nested parallelism

\textbf{MAT-VEC-MAIN-LOOP}(A, x, y, n, i, i') \\
\textit{if} \textit{i} == \textit{i}' \\
\hspace{1em} \textbf{for} \textit{j} = 1 \textbf{ to } \textit{n} \\
\hspace{2em} \textit{y}_{\textit{i}} = \textit{y}_{\textit{i}} + a_{\textit{ij}}x_{\textit{j}} \\
\textit{else} \textit{mid} = [(\textit{i} + \textit{i}')/2] \\
\hspace{1em} \textbf{spawn} \textbf{MAT-VEC-MAIN-LOOP}(A, x, y, n, i, \textit{mid}) \\
\hspace{2em} \textbf{MAT-VEC-MAIN-LOOP}(A, x, y, n, \textit{mid} + 1, i') \\
\hspace{2em} \textbf{sync}
Parallel Loops – Implementation

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

Here, \textit{parallel for} is implemented by the compiler as a divide-and-conquer subroutine using nested parallelism

\[ \text{MAT-VEC-MAIN-LOOP}(A, x, y, n, i, i') \]
\begin{align*}
\text{if } i &= i' \\
& \quad \text{for } j = 1 \text{ to } n \\
& \quad \quad y_i = y_i + a_{ij}x_j \\
\text{else } \text{mid} &= \lfloor (i + i')/2 \rfloor \\
\text{spawn } & \text{MAT-VEC-MAIN-LOOP}(A, x, y, n, i, \text{mid}) \\
& \text{MAT-VEC-MAIN-LOOP}(A, x, y, n, \text{mid} + 1, i') \\
\text{sync}
\end{align*}

Work: \( T_1(n) = \Theta(n^2) \)

Span:

Parallelism
**Parallel Loops – Implementation**

**MAT-VEC**\((A, x)\)

\(n = A\.rows\)

Let \(y\) be a new vector of length \(n\)

**Parallel for** \(i = 1 \text{ to } n\)

\[ y_i = 0 \]

**Parallel for** \(i = 1 \text{ to } n\)

For \(j = 1 \text{ to } n\)

\[ y_i = y_i + a_{ij}x_j \]

Return \(y\)

Here, **parallel for** is implemented by the compiler as a divide-and-conquer subroutine using nested parallelism

**MAT-VEC-MAIN-LOOP**\((A, x, y, n, i, i')\)

If \(i == i'\)

For \(j = 1 \text{ to } n\)

\[ y_i = y_i + a_{ij}x_j \]

Else

Mid = \(\lfloor (i + i')/2 \rfloor\)

Spawn **MAT-VEC-MAIN-LOOP**\((A, x, y, n, i, \text{mid})\)

**MAT-VEC-MAIN-LOOP**\((A, x, y, n, \text{mid} + 1, i')\)

Sync

**Work:** \(T_1(n) = \Theta(n^2)\)

**Span:** \(T_\infty(n) = \Theta(\log n) + \Theta(\log n) + \Theta(n) = \Theta(n)\)

Parallelism
Parallel Loops – Implementation

\text{MAT-VEC}(A, x)\\n = A.\text{rows}\\let y \text{ be a new vector of length } n\\\text{parallel for } i = 1 \text{ to } n\\ \quad y_i = 0\\\text{parallel for } i = 1 \text{ to } n\\ \quad \text{for } j = 1 \text{ to } n\\ \quad y_i = y_i + a_{ij}x_j\\\text{return } y\\

Here, \text{parallel for} is implemented by the compiler as a divide-and-conquer subroutine using nested parallelism

\text{MAT-VEC-MAIN-LOOP}(A, x, y, n, i, i')\\\text{if } i == i'\\ \quad \text{for } j = 1 \text{ to } n\\ \quad y_i = y_i + a_{ij}x_j\\\text{else } mid = [(i + i')/2]\\ \quad \text{spawn } \text{MAT-VEC-MAIN-LOOP}(A, x, y, n, i, mid)\\ \quad \text{MAT-VEC-MAIN-LOOP}(A, x, y, n, mid + 1, i')\\ \quad \text{sync}\\

Work: \quad T_1(n) = \Theta(n^2)\\Span: \quad T_\infty(n) = \Theta(\lg n) + \Theta(\lg n) + \Theta(n)\\\quad = \Theta(n)\\Parallelism = \Theta(n^2)/\Theta(n) = \Theta(n)
Race Conditions

• A multithreaded algorithm is deterministic if and only if does the same thing on the same input, no matter how the instructions are scheduled.

• A multithreaded algorithm is nondeterministic if its behavior might vary from run to run.

• Often, a multithreaded algorithm that is intended to be deterministic fails to be.
Determinacy Race

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

```
RACE-EXAMPLE()
    x = 0
    parallel for i = 1 to 2
        x = x+1
    print x
```
Determinacy Race

• When a processor increments $x$, the operation is not indivisible, but composed of a sequence of instructions:
  1) Read $x$ from memory into one of the processor’s registers
  2) Increment the value of the register
  3) Write the value in the register back into $x$ in memory
Determinacy Race

x = 0
assign r1 = 0
incr r1, so r1=1
assign r2 = 0
incr r2, so r2 = 1
write back x = r1
write back x = r2
print x  // now prints 1 instead of 2
Example: Using work, span for design

• Consider a program prototyped on 32-processor computer, but aimed to run on supercomputer with 512 processors
• Designers incorporated an optimization to reduce run time of benchmark on 32-processor machine, from $T_{32} = 65$ to $T'_{32} = 40$
• But, can show that this optimization made overall runtime on 512 processors slower than the original! Thus, optimization didn’t help.

• Analysis for 32 processors:
  Original:
  \[
  \begin{align*}
  T_1 &= 2048 \\
  T_\infty &= 1 \\
  T_P &= T_1/p + T_\infty \\
  \Rightarrow T_{32} &= 2048/32 + 1 = 65
  \end{align*}
  \]
  Optimized:
  \[
  \begin{align*}
  T'_1 &= 1024 \\
  T'_\infty &= 8 \\
  T'_P &= T'_1/p + T'_\infty \\
  \Rightarrow T'_{32} &= 1024/32 + 8 = 40
  \end{align*}
  \]

• Analysis for 512 processors:
  Original:
  \[
  \begin{align*}
  T_1 &= 2048 \\
  T_\infty &= 1 \\
  T_P &= T_1/p + T_\infty \\
  \Rightarrow T_{512} &= 2048/512 + 1 = 5
  \end{align*}
  \]
  Optimized:
  \[
  \begin{align*}
  T'_1 &= 1024 \\
  T'_\infty &= 8 \\
  T'_P &= T'_1/p + T'_\infty \\
  \Rightarrow T'_{512} &= 1024/512 + 8 = 10
  \end{align*}
  \]

\textit{Difference depends on whether or not span dominates}
In-Class Exercise

Prof. Karan measures her deterministic multithreaded algorithm on 4, 10, and 64 processors of an ideal parallel computer using a greedy scheduler. She claims that the 3 runs yielded $T_4 = 80$ seconds, $T_{10} = 42$ seconds, and $T_{64} = 10$ seconds. Are these runtimes believable?
Multithreaded Matrix Multiplication

First, parallelize Square-Matrix-Multiply:

\[ \text{P-SQUARE-MATRIX-MULTIPLY}(A, B) \]
\[ n = A.\text{rows} \]
let \( C \) be a new \( n \times n \) matrix
\[ \text{parallel for } i = 1 \text{ to } n \]
\[ \quad \text{parallel for } j = 1 \text{ to } n \]
\[ \quad c_{ij} = 0 \]
\[ \quad \text{for } k = 1 \text{ to } n \]
\[ \quad \quad c_{ij} = c_{ij} + a_{ik} \cdot b_{kj} \]
return \( C \)
Multithreaded Matrix Multiplication

First, parallelize Square-Matrix-Multiply:

P-SQUARE-MATRIX-MULTIPLY(A, B)

\( n = A \text{.rows} \)

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

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

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

\[ c_{ij} = 0 \]

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

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

return \( C \)
Multithreaded Matrix Multiplication

First, parallelize Square-Matrix-Multiply:

\[ \text{P-SQUARE-MATRIX-MULTIPLY}(A, B) \]

\[ n = \text{A.rows} \]

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

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

\[
\begin{align*}
\text{parallel for } j &= 1 \text{ to } n \\
\quad c_{ij} &= 0 \\
\text{for } k &= 1 \text{ to } n \\
\quad c_{ij} &= c_{ij} + a_{ik} \cdot b_{kj}
\end{align*}
\]

return \( C \)

Work: \( T_1(n) = \Theta(n^3) \)

Span:

Parallelism:
Multithreaded Matrix Multiplication

First, parallelize Square-Matrix-Multiply:

P-SQUARE-MATRIX-MULTIPLY(A, B)

\( n = A . \text{rows} \)

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

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

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

\( c_{ij} = 0 \)

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

\( c_{ij} = c_{ij} + a_{ik} \cdot b_{kj} \)

return \( C \)

Work: \( T_1(n) = \Theta(n^3) \)

Span: \( T_\infty(n) = \Theta(\lg n) + \Theta(\lg n) + \Theta(n) = \Theta(n) \)

Parallelism:
Multithreaded Matrix Multiplication

First, parallelize Square-Matrix-Multiply:

\[
\text{P-SQUARE-MATRIX-MULTIPLY}(A, B)
\]
\[
n = A.\text{rows}
\]
let \( C \) be a new \( n \times n \) matrix

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

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

\[
c_{ij} = 0
\]

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

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

return \( C \)

Work: \( T_1(n) = \Theta(n^3) \)

Span: \( T_\infty(n) = \Theta(\lg n) + \Theta(\lg n) + \Theta(n) = \Theta(n) \)

Parallelism = \( \Theta(n^3)/\Theta(n) = \Theta(n^2) \)
Now, let’s try divide-and-conquer

• Remember: Basic divide and conquer method:

To multiply two $n \times n$ matrices, $A \times B = C$, divide into sub-matrices:

\[
\begin{pmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{pmatrix}
\cdot
\begin{pmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{pmatrix}
=
\begin{pmatrix}
C_{11} & C_{12} \\
C_{21} & C_{22}
\end{pmatrix}
\]

\[
C_{11} = A_{11}B_{11} + A_{12}B_{21}
\]

\[
C_{12} = A_{11}B_{12} + A_{12}B_{22}
\]

\[
C_{21} = A_{21}B_{11} + A_{22}B_{21}
\]

\[
C_{22} = A_{21}B_{12} + A_{22}B_{22}
\]
Parallelized Divide-and-Conquer Matrix Multiplication

P-MATRIX-MULTIPLY-RECURSIVE(C, A, B):

\( n = A.\text{rows} \)

if \( n = 1 \):
\[ c_{11} = a_{11} b_{11} \]

else:

allocate a temporary matrix \( T[1 \ldots n, 1 \ldots n] \)

partition \( A, B, C, \) and \( T \) into \((n/2) \times (n/2)\) submatrices

spawn P-MATRIX-MULTIPLY-RECURSIVE \((C_{11}, A_{11}, B_{11})\)

spawn P-MATRIX-MULTIPLY-RECURSIVE \((C_{12}, A_{11}, B_{12})\)

spawn P-MATRIX-MULTIPLY-RECURSIVE \((C_{21}, A_{21}, B_{11})\)

spawn P-MATRIX-MULTIPLY-RECURSIVE \((C_{22}, A_{21}, B_{12})\)

spawn P-MATRIX-MULTIPLY-RECURSIVE \((T_{11}, A_{12}, B_{21})\)

spawn P-MATRIX-MULTIPLY-RECURSIVE \((T_{12}, A_{12}, B_{22})\)

spawn P-MATRIX-MULTIPLY-RECURSIVE \((T_{21}, A_{22}, B_{21})\)

spawn P-MATRIX-MULTIPLY-RECURSIVE \((T_{22}, A_{22}, B_{22})\)

sync

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

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

\[ c_{ij} = c_{ij} + t_{ij} \]
Parallelized Divide-and-Conquer Matrix Multiplication

P-MATRIX-MULTIPLY-RECURSIVE(C, A, B):

$n = A$.rows

if $n == 1$:

\[ c_{11} = a_{11}b_{11} \]

else:

allocate a temporary matrix $T[1 ... n, 1 ... n]$

partition $A$, $B$, $C$, and $T$ into $(n/2) \times (n/2)$ submatrices

spawn P-MATRIX-MULTIPLY-RECURSIVE ($C_{11}, A_{11}, B_{11}$)

spawn P-MATRIX-MULTIPLY-RECURSIVE ($C_{12}, A_{11}, B_{12}$)

spawn P-MATRIX-MULTIPLY-RECURSIVE ($C_{21}, A_{21}, B_{11}$)

spawn P-MATRIX-MULTIPLY-RECURSIVE ($C_{22}, A_{21}, B_{12}$)

spawn P-MATRIX-MULTIPLY-RECURSIVE ($T_{11}, A_{12}, B_{21}$)

spawn P-MATRIX-MULTIPLY-RECURSIVE ($T_{12}, A_{12}, B_{22}$)

spawn P-MATRIX-MULTIPLY-RECURSIVE ($T_{21}, A_{22}, B_{21}$)

spawn P-MATRIX-MULTIPLY-RECURSIVE ($T_{22}, A_{22}, B_{22}$)

sync

parallel for $i = 1$ to $n$

parallel for $j = 1$ to $n$

\[ c_{ij} = c_{ij} + t_{ij} \]

\[
\begin{pmatrix}
  c_{11} & c_{12} \\
  c_{21} & c_{22}
\end{pmatrix}
= 
\begin{pmatrix}
  a_{11} & a_{12} \\
  a_{21} & a_{22}
\end{pmatrix}
\cdot
\begin{pmatrix}
  b_{11} & b_{12} \\
  b_{21} & b_{22}
\end{pmatrix}
= 
\begin{pmatrix}
  a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\
  a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22}
\end{pmatrix}
\]
Parallelized Divide-and-Conquer Matrix Multiplication

\[\text{P-MATRIX-MULTIPLY-RECURSIVE}(C, A, B):\]
\[n = A.\text{rows}\]
\[\text{if } n == 1:\]
\[c_{11} = a_{11}b_{11}\]
\[\text{else:}\]
\[\text{allocate a temporary matrix } T[1 \ldots n, 1 \ldots n]\]
\[\text{partition } A, B, C, \text{ and } T \text{ into } (n/2) \times (n/2) \text{ submatrices}\]
\[\text{spawn } \text{P-MATRIX-MULTIPLY-RECURSIVE} \left( C_{11}, A_{11}, B_{11} \right)\]
\[\text{spawn } \text{P-MATRIX-MULTIPLY-RECURSIVE} \left( C_{12}, A_{11}, B_{12} \right)\]
\[\text{spawn } \text{P-MATRIX-MULTIPLY-RECURSIVE} \left( C_{21}, A_{21}, B_{11} \right)\]
\[\text{spawn } \text{P-MATRIX-MULTIPLY-RECURSIVE} \left( C_{22}, A_{21}, B_{12} \right)\]
\[\text{spawn } \text{P-MATRIX-MULTIPLY-RECURSIVE} \left( T_{11}, A_{12}, B_{21} \right)\]
\[\text{spawn } \text{P-MATRIX-MULTIPLY-RECURSIVE} \left( T_{12}, A_{12}, B_{22} \right)\]
\[\text{spawn } \text{P-MATRIX-MULTIPLY-RECURSIVE} \left( T_{21}, A_{22}, B_{21} \right)\]
\[\text{P-MATRIX-MULTIPLY-RECURSIVE} \left( T_{22}, A_{22}, B_{22} \right)\]
\[\text{sync}\]
\[\text{parallel for } i = 1 \text{ to } n\]
\[\text{parallel for } j = 1 \text{ to } n\]
\[c_{ij} = c_{ij} + t_{ij}\]

\[\text{Work:}\]
\[T_1(n) = 8T_1 \left( \frac{n}{2} \right) + \Theta(n^2)\]
\[= \Theta(n^3)\]

\[\text{Parallelism:}\]
\[\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}\]
Parallelized Divide-and-Conquer Matrix Multiplication

\[ \text{P-MATRIX-MULTIPLY-RECURSIVE}(C, A, B): \]
\[ n = A.\text{rows} \]
\[ \text{if } n \geq 1: \]
\[ c_{11} = a_{11}b_{11} \]
\[ \text{else:} \]
\[ \text{allocate a temporary matrix } T[1 \ldots n, 1 \ldots n] \]
\[ \text{partition } A, B, C, \text{and } T \text{ into } (n/2) \times (n/2) \text{ submatrices} \]
\[ \text{spawn } \text{P-MATRIX-MULTIPLY-RECURSIVE} \ (C_{11}, A_{11}, B_{11}) \]
\[ \text{spawn } \text{P-MATRIX-MULTIPLY-RECURSIVE} \ (C_{12}, A_{11}, B_{12}) \]
\[ \text{spawn } \text{P-MATRIX-MULTIPLY-RECURSIVE} \ (C_{21}, A_{21}, B_{11}) \]
\[ \text{spawn } \text{P-MATRIX-MULTIPLY-RECURSIVE} \ (C_{22}, A_{21}, B_{12}) \]
\[ \text{spawn } \text{P-MATRIX-MULTIPLY-RECURSIVE} \ (T_{11}, A_{12}, B_{21}) \]
\[ \text{spawn } \text{P-MATRIX-MULTIPLY-RECURSIVE} \ (T_{12}, A_{12}, B_{22}) \]
\[ \text{spawn } \text{P-MATRIX-MULTIPLY-RECURSIVE} \ (T_{21}, A_{22}, B_{21}) \]
\[ \text{spawn } \text{P-MATRIX-MULTIPLY-RECURSIVE} \ (T_{22}, A_{22}, B_{22}) \]
\[ \text{sync} \]
\[ \text{parallel for } i = 1 \text{ to } n \]
\[ \text{parallel for } j = 1 \text{ to } n \]
\[ c_{ij} = c_{ij} + t_{ij} \]

**Work:**
\[
T_1(n) = 8T_1\left(\frac{n}{2}\right) + \Theta(n^2) \\
= \Theta(n^3)
\]

**Span:**
\[
T_\infty(n) = T_\infty\left(\frac{n}{2}\right) + \Theta(lg \ n) \\
= \Theta(lg^2 n)
\]

**Parallelism:**
\[
\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \\
= \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}
\]
Parallelized Divide-and-Conquer Matrix Multiplication

P-MATRIX-MULTIPLY-RECURSIVE(C, A, B):

\[ n = A\.\text{rows} \]

if \( n == 1 \):

\[ c_{11} = a_{11}b_{11} \]

else:

allocate a temporary matrix \( T[1 \ldots n, 1 \ldots n] \)
partition A, B, C, and T into \( (n/2) \times (n/2) \) submatrices
spawn P-MATRIX-MULTIPLY-RECURSIVE \((C_{11},A_{11},B_{11})\)
spawn P-MATRIX-MULTIPLY-RECURSIVE \((C_{12},A_{11},B_{12})\)
spawn P-MATRIX-MULTIPLY-RECURSIVE \((C_{21},A_{21},B_{11})\)
spawn P-MATRIX-MULTIPLY-RECURSIVE \((C_{22},A_{21},B_{12})\)
spawn P-MATRIX-MULTIPLY-RECURSIVE \((T_{11},A_{12},B_{21})\)
spawn P-MATRIX-MULTIPLY-RECURSIVE \((T_{12},A_{12},B_{22})\)
spawn P-MATRIX-MULTIPLY-RECURSIVE \((T_{21},A_{22},B_{21})\)
spawn P-MATRIX-MULTIPLY-RECURSIVE \((T_{22},A_{22},B_{22})\)

sync

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

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

\[ c_{ij} = c_{ij} + t_{ij} \]

Work:

\[ T_1(n) = 8T_1 \left( \frac{n}{2} \right) + \Theta(n^2) = \Theta(n^3) \]

Span:

\[ T_\infty(n) = T_\infty \left( \frac{n}{2} \right) + \Theta(\lg n) = \Theta(\lg^2 n) \]

Parallelism:

\[ \Theta \left( \frac{n^3}{\lg^2 n} \right) \]
Multithreading Strassen’s Alg

• Remember how Strassen works?
Strassen’s Matrix Multiplication

Strassen observed [1969] that the product of two matrices can be computed in general as follows:

\[
\begin{pmatrix}
C_{11} & C_{12} \\
C_{21} & C_{22}
\end{pmatrix}
= \begin{pmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{pmatrix}
\times
\begin{pmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{pmatrix}
= \begin{pmatrix}
P_5 + P_4 - P_2 + P_6 \\
P_3 + P_4 \\
P_5 + P_1 - P_3 - P_7
\end{pmatrix}
+ \begin{pmatrix}
P_1 + P_2
\end{pmatrix}
\]
Formulas for Strassen’s Algorithm

\[ P_1 = A_{11} \times (B_{12} - B_{22}) \]
\[ P_2 = (A_{11} + A_{12}) \times B_{22} \]
\[ P_3 = (A_{21} + A_{22}) \times B_{11} \]
\[ P_4 = A_{22} \times (B_{21} - B_{11}) \]
\[ P_5 = (A_{11} + A_{22}) \times (B_{11} + B_{22}) \]
\[ P_6 = (A_{12} - A_{22}) \times (B_{21} + B_{22}) \]
\[ P_7 = (A_{11} - A_{21}) \times (B_{11} + B_{12}) \]
Multi-threaded version of Strassen’s Algorithm

\[ P_1 = A_{11} \times (B_{12} - B_{22}) \]
\[ P_2 = (A_{11} + A_{12}) \times B_{22} \]
\[ P_3 = (A_{21} + A_{22}) \times B_{11} \]
\[ P_4 = A_{22} \times (B_{21} - B_{11}) \]
\[ P_5 = (A_{11} + A_{22}) \times (B_{11} + B_{22}) \]
\[ P_6 = (A_{12} - A_{22}) \times (B_{21} + B_{22}) \]
\[ P_7 = (A_{11} - A_{21}) \times (B_{11} + B_{12}) \]

First, create 10 matrices, each of which is \( \frac{n}{2} \times \frac{n}{2} \).

Work = \( \Theta(n^2) \)

Span = \( \Theta(\lg n) \), using doubly-nested parallel for loops
Formulas for Strassen’s Algorithm

\[ P_1 = A_{11} \times (B_{12} - B_{22}) \]
\[ P_2 = (A_{11} + A_{12}) \times B_{22} \]
\[ P_3 = (A_{21} + A_{22}) \times B_{11} \]
\[ P_4 = A_{22} \times (B_{21} - B_{11}) \]
\[ P_5 = (A_{11} + A_{22}) \times (B_{11} + B_{22}) \]
\[ P_6 = (A_{12} - A_{22}) \times (B_{21} + B_{22}) \]
\[ P_7 = (A_{11} - A_{21}) \times (B_{11} + B_{12}) \]

First, create 10 matrices, each of which is \( n/2 \times n/2 \).

Work = \( \Theta(n^2) \)

Then, recursively compute 7 matrix products.
Then add together, using doubly-nested parallel for loops

\[
\begin{pmatrix}
C_{11} & C_{12} \\
C_{21} & C_{22}
\end{pmatrix}
= 
\begin{pmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{pmatrix}
\begin{pmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{pmatrix}
\]

\[
P_5 + P_4 - P_2 + P_6
= 
P_3 + P_4
\]

\[
P_1 + P_2
= 
P_5 + P_1 - P_3 - P_7
\]

Work = \Theta(n^2)

Span = \Theta(\log n),
Resulting Runtime for Multithreaded Strassens’ Alg

Work:

\[ T_1(n) = \Theta(1) + \Theta(n^2) + 7T_1\left(\frac{n}{2}\right) + \Theta(n^2) \]
\[ = 7T_1\left(\frac{n}{2}\right) + \Theta(n^2) \]
\[ = \Theta(n^{\lg 7}) \]

Span:

\[ T_\infty(n) = T_\infty\left(\frac{n}{2}\right) + \Theta(\lg n) \]
\[ = \Theta(\lg^2 n) \]

Parallelism: \[ \Theta\left(\frac{n^{\lg 7}}{\lg^2 n}\right) \]
Reading Assignments

• Reading assignment for next class:
  – Chapter 27.3

• Announcement: Exam #2 on Tuesday, April 1
  – Will cover greedy algorithms, amortized analysis
  – HW 6-9