Today:
- Multithreaded Algs.

COSC 581, Algorithms

March 13, 2014

Many of these slides are adapted from several online sources

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_∞ in time:

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

- Given the fact the best we can hope for on P processors is $T_P = {}^{T_1}/_P$ 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 ${T_1}/{P}$. Then the total work of the complete steps is at least

$$\begin{aligned} P(\lfloor T_1/P \rfloor + 1) &= P\lfloor T_1/P \rfloor + P \\ &= T_1 - (T_1 \mod P) + P \\ &> T_1 \end{aligned}$$

- 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_∞ .
- 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_∞ 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,

$$T_P \leq {}^{T_1}\!/_{P} + T_\infty \leq 2\max\left({}^{T_1}\!/_{P}, T_\infty\right) \leq 2T_P^*$$

Slackness

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

- Slackness = $(T_1 \ / \ T_\infty)$ / 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_∞ be the span of the computation. If the slackness is big, $P << (T_1 / T_\infty)$, then

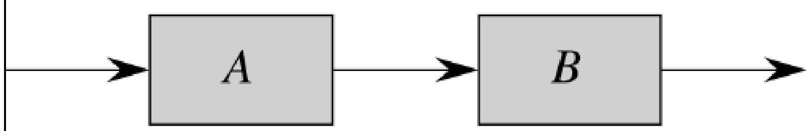
  T_P is approximately T_1 / P [i.e, near-perfect speedup]

- Proof: If $P << (T_1 / T_\infty)$, then $T_\infty << T_1 / P$. 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.

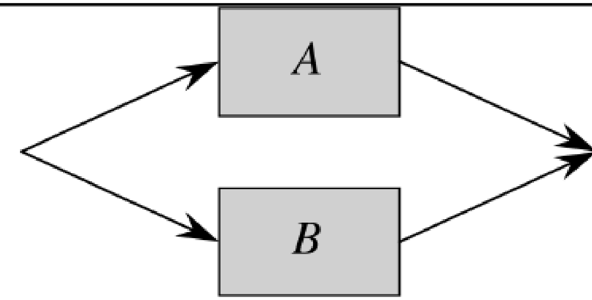
- Analyzing work is no different than for serial algorithms
- Analyzing span is more involved…
  - Two computations in series means their spans *add*



Work: $T_1(A \cup B) = T_1(A) + T_1(B)$

Span: $T_\infty(A \cup B) = T_\infty(A) + T_\infty(B)$

  - Two computations in parallel means you take *maximum* of individual spans



Work: $T_1(A \cup B) = T_1(A) + T_1(B)$

Span: $T_\infty(A \cup B) = \max(T_\infty(A), T_\infty(B))$

Analyzing Parallel Fibonacci Computation

- Parallel algorithm to compute Fibonacci numbers:

```
P-FIB(n)
    if n ≤ 1   return n;
    else x = spawn P-FIB (n-1);   // parallel execution
         y = spawn P-FIB (n-2) ;  // parallel execution
         sync;  // wait for results of x and y
         return x + y;
```

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:

$T_1 = T(n\text{-}1) + T(n\text{-}2) + \theta(1)$

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

Span of Fibonacci

- Recall that the span T_∞ is the longest path in the computational DAG. Since FIB(n) spawns

  FIB(n-1) and FIB(n-2),

  we have:

$$
\begin{aligned}
T_\infty(n) &= \max(T_\infty(n-1), T_\infty(n-2)) + \Theta(1) \\
&= T_\infty(n-1) + \Theta(1) \\
&= \Theta(n)
\end{aligned}
$$

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* x *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:

MAT-VEC(*A*, *x*)
n = *A.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

Parallel Loops – Implementation

MAT-VEC(A, x)
n = $A.rows$
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

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 $mid = \lfloor (i + i')/2 \rfloor$
  spawn MAT-VEC-MAIN-LOOP(A, x, y, n, i, mid)
  MAT-VEC-MAIN-LOOP(A, x, y, n, $mid + 1$, i')
  sync

Parallel Loops – Implementation

MAT-VEC(A, x)
n = $A.rows$
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

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 $mid = \lfloor (i + i')/2 \rfloor$
    spawn MAT-VEC-MAIN-LOOP(A, x, y, n, i, mid)
    MAT-VEC-MAIN-LOOP(A, x, y, n, $mid + 1$, i')
    sync

Work:

Span:

Parallelism

Parallel Loops – Implementation

MAT-VEC(A, x)
n = $A.rows$
let y be a new vector of length n
parallel for $i = 1$ **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

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 $mid = \lfloor (i + i')/2 \rfloor$
    spawn MAT-VEC-MAIN-LOOP(A, x, y, n, i, mid)
    MAT-VEC-MAIN-LOOP(A, x, y, n, $mid + 1$, i')
    sync

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 $i = 1$ **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

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 mid = $\lfloor (i + i')/2 \rfloor$
    spawn MAT-VEC-MAIN-LOOP(A, x, y, n, i, mid)
    MAT-VEC-MAIN-LOOP(A, x, y, n, $mid + 1$, i')
    sync

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

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

Parallelism

Parallel Loops – Implementation

MAT-VEC(*A*, *x*)
n = *A.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

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 *mid* = $\lfloor (i + i')/2 \rfloor$
    spawn MAT-VEC-MAIN-LOOP(*A*, *x*, *y*, *n*, i, mid)
    MAT-VEC-MAIN-LOOP(*A*, *x*, *y*, *n*, $mid + 1$, i')
    sync

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

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

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:

$T_1 = 2048$

$T_\infty = 1$

$T_P = {}^{T_1}/_P + T_\infty$

$\Rightarrow T_{32} = 2048/32 + 1 = 65$

Optimized:

$T'_1 = 1024$

$T'_\infty = 8$

$T'_P = {}^{T'_1}/_P + T'_\infty$

$\Rightarrow T'_{32} = 1024/32 + 8 = 40$

- Analysis for 512 processors:

Original:

$T_1 = 2048$

$T_\infty = 1$

$T_P = {}^{T_1}/_P + T_\infty$

$\Rightarrow T_{512} = 2048/512 + 1 = 5$

Optimized:

$T'_1 = 1024$

$T'_\infty = 8$

$T'_P = {}^{T'_1}/_P + T'_\infty$

$\Rightarrow T'_{512} = 1024/512 + 8 = 10$

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:

P-SQUARE-MATRIX-MULTIPLY(A, B)
n=A.rows
let C be a new n x 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:

P-SQUARE-MATRIX-MULTIPLY(A, B)
n=A.rows
let C be a new n x 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:

Span:

Parallelism:

Multithreaded Matrix Multiplication

First, parallelize Square-Matrix-Multiply:

P-SQUARE-MATRIX-MULTIPLY(A, B)
n=A.rows
let *C* be a new n x 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:

Parallelism:

Multithreaded Matrix Multiplication

First, parallelize Square-Matrix-Multiply:

P-SQUARE-MATRIX-MULTIPLY(A, B)
n=A.rows
let C be a new n x 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:

P-SQUARE-MATRIX-MULTIPLY(A, B)
n=A.rows
let C be a new n x 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* x *n* matrices, *A* x *B = C*, divide into sub-matrices:

$$\begin{vmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{vmatrix} \cdot \begin{vmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{vmatrix} = \begin{vmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{vmatrix}$$

$C_{11} = A_{11}B_{11} + A_{12}B_{21}$

$C_{12} = A_{11}B_{12} + A_{12}B_{22}$

$C_{21} = A_{21}B_{11} + A_{22}B_{21}$

$C_{22} = A_{21}B_{12} + A_{22}B_{22}$

Parallelized Divide-and-Conquer Matrix Multiplication

P-MATRIX-MULTIPLY-RECURSIVE(C, A, B):
n = A.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) x (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})
    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

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) x (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})
    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:

Span:

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.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) x (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})
    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:

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.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) x (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})
    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:

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

Parallelized Divide-and-Conquer Matrix Multiplication

P-MATRIX-MULTIPLY-RECURSIVE(C, A, B):
n = A.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) x (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})
    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: $\Theta\left({n^3}/{lg^2 n}\right)$

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

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:

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} * \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

$$= \begin{pmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_5 + P_1 - P_3 - P_7 \end{pmatrix}$$

Formulas for Strassen's Algorithm

$P_1 = A_{11} * (B_{12} - B_{22})$

$P_2 = (A_{11} + A_{12}) * B_{22}$

$P_3 = (A_{21} + A_{22}) * B_{11}$

$P_4 = A_{22} * (B_{21} - B_{11})$

$P_5 = (A_{11} + A_{22}) * (B_{11} + B_{22})$

$P_6 = (A_{12} - A_{22}) * (B_{21} + B_{22})$

$P_7 = (A_{11} - A_{21}) * (B_{11} + B_{12})$

Multi-threaded version of Strassen's Algorithm

$P_1 = A_{11} * (B_{12} - B_{22})$

$P_2 = (A_{11} + A_{12}) * B_{22}$

$P_3 = (A_{21} + A_{22}) * B_{11}$

$P_4 = A_{22} * (B_{21} - B_{11})$

$P_5 = (A_{11} + A_{22}) * (B_{11} + B_{22})$

$P_6 = (A_{12} - A_{22}) * (B_{21} + B_{22})$

$P_7 = (A_{11} - A_{21}) * (B_{11} + B_{12})$

First, create 10 matrices, each of which is $n/2$ x $n/2$.

Work = $\Theta(n^2)$

Span = $\Theta(\lg n)$, using doubly-nested **parallel for** loops

Formulas for Strassen's Algorithm

$P_1 = A_{11} * (B_{12} - B_{22})$

$P_2 = (A_{11} + A_{12}) * B_{22}$

$P_3 = (A_{21} + A_{22}) * B_{11}$

$P_4 = A_{22} * (B_{21} - B_{11})$

$P_5 = (A_{11} + A_{22}) * (B_{11} + B_{22})$

$P_6 = (A_{12} - A_{22}) * (B_{21} + B_{22})$

$P_7 = (A_{11} - A_{21}) * (B_{11} + B_{12})$

First, create 10 matrices, each of which is $n/2$ x $n/2$.

Work = $\Theta(n^2)$

Then, recursively compute 7 matrix products

Then add together, using doubly-nested parallel for loops

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} * \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

$$= \begin{pmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_5 + P_1 - P_3 - P_7 \end{pmatrix}$$

Work = $\Theta(n^2)$

Span = $\Theta(\lg n)$,

Resulting Runtime for Multithreaded Strassens' Alg

Work:

$$T_1(n) = \Theta(1) + \Theta(n^2) + 7T_1\left(\tfrac{n}{2}\right) + \Theta(n^2)$$
$$= 7T_1\left(\tfrac{n}{2}\right) + \Theta(n^2)$$
$$= \Theta\left(n^{\lg 7}\right)$$

Span:

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

Parallelism: $\Theta\left({n^{\lg 7}}/{\lg^2 n}\right)$

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