Multithreaded Algorithms

Andreas Klappenecker
Motivation

We have discussed serial algorithms that are suitable for running on a uniprocessor computer. We will now extend our model to parallel algorithms that can run on a multiprocessor computer.
There exist many competing models of parallel computation that are essentially different. For example, one can have shared or distributed memory.

Since multicore processors are ubiquitous, we focus on a parallel computing model with shared memory.
Dynamic Multithreading

Programming a shared-memory parallel computer can be difficult and error-prone. In particular, it is difficult to partition the work among several threads so that each thread approximately has the same load.

A concurrency platform is a software layer that coordinates, schedules, and manages parallel-computing resources. We will use a simple extension of the serial programming model that uses the concurrency instructions `parallel`, `spawn`, and `sync`. 
Spawn: If spawn proceeds a procedure call, then the procedure instance that executes the spawn (the parent) may continue to execute in parallel with the spawned subroutine (the child), instead of waiting for the child to complete.

The keyword spawn does not say that a procedure must execute concurrently, but simply that it may.

At runtime, it is up to the scheduler to decide which subcomputations should run concurrently.
The keyword `sync` indicates that the procedure must wait for all its spawned children to complete.
Parallel

Many algorithms contain loops, where all iterations can operate in parallel. If the `parallel` keyword proceeds a for loop, then this indicates that the loop body can be executed in parallel.
Fibonacci Numbers
Definition

The Fibonacci numbers are defined by the recurrence:

\[ F_0 = 0 \]

\[ F_1 = 1 \]

\[ F_i = F_{i-1} + F_{i-2} \]

for \( i > 1 \).
Computing the Fibonacci numbers can be done with the following algorithm:

\[
\text{Fibonacci}(n) \\
\quad \text{if } n < 2 \text{ then return } n; \\
\quad x = \text{Fibonacci}(n-1); \\
\quad y = \text{Fibonacci}(n-2); \\
\quad \text{return } x + y;
\]
Let $T(n)$ denote the running time of $Fibonacci(n)$. Since this procedure contains two recursive calls and a constant amount of extra work, we get

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

which yields $T(n) = \Theta(F_n) = \Theta \left( \frac{1 + \sqrt{5}}{2} \right)^n$.

Since this is an exponential growth, this is a particularly bad way to calculate Fibonacci numbers.

How would you calculate the Fibonacci numbers?
Fibonacci Numbers

\[
\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}
\]

This allows you to calculate \( F_n \) in \( O(\log n) \) steps by repeated squaring of the matrix. This is how you can calculate the Fibonacci numbers with a serial algorithm.

To illustrate the principles of parallel programming, we will use the naive (bad) algorithm, though.
Parallel algorithm to compute Fibonacci numbers:

\[ \text{Fibonacci}(n) \]

- if \( n < 2 \) then return \( n \);
- \( x = \text{spawn} \ \text{Fibonacci}(n-1);  \) // parallel execution
- \( y = \text{spawn} \ \text{Fibonacci}(n-2);  \) // parallel execution
- \( \text{sync};  \) // wait for results of \( x \) and \( y \)
- return \( x + y \);
Multithreaded computation can be better understood with the help of a computation directed acyclic graph \( G=(V,E) \).

The vertices \( V \) in the graph are the instructions.

The edges \( E \) represent dependencies between instructions.

An edge \((u,v)\) is in \( E \) means that the instruction \( u \) must execute before instruction \( v \).

[Problem: Somewhat too detailed. We will group the instructions into threads.]
A sequence of instructions containing no parallel control (spawn, sync, return from spawn, parallel) can be grouped into a single *strand*.

A strand of maximal length will be called a *thread*. 
A computation directed acyclic graph $G=(V,E)$ consists a vertex set $V$ that comprises the threads of the program.

The edge set $E$ contains an edge $(u,v)$ if and only if the thread $u$ needs to execute before thread $v$.

If there is an edge between thread $u$ and $v$, then they are said to be (logically) in series. If there is no thread, then they are said to be (logically) in parallel.
Edge Classification

A continuation edge \((u,v)\) connects a thread \(u\) to its successor \(v\) within the same procedure instance.

When a thread \(u\) spawns a new thread \(v\), then \((u,v)\) is called a spawn edge.

When a thread \(v\) returns to its calling procedure and \(x\) is the thread following the parallel control, then the return edge \((v,x)\) is included in the graph.
Fibonacci Example

Parallel algorithm to compute Fibonacci numbers:

Fibonacci(n)
    if n < 2 then return n;   // thread A
    x = spawn Fibonacci(n-1);
    y = spawn Fibonacci(n-2); // thread B
    sync;
    return x + y; // thread C
Fibonacci(4)
Performance Measures

The work of a multithreaded computation is the total time to execute the entire computation on one processor.

Work = sum of the times taken by each thread
Performance Measures

The span is the longest time to execute the threads along any path of the computational directed acyclic graph.
Performance Measure Example

In Fibonacci(4), we have

17 vertices = 17 threads.

8 vertices on longest path.

Assuming unit time for each thread, we get

work = 17 time units

span = 8 time units
The actual running time of a multithreaded computation depends not just on its work and span, but also on how many processors (cores) are available, and how the scheduler allocates strands to processors.

Running time on P processors is indicated by subscript P
- $T_1$ running time on a single processor
- $T_P$ running time on P processors
- $T_\infty$ running time on unlimited processors
An ideal parallel computer with $P$ processors can do at most $P$ units of work. Total work to do is $T_1$.

Thus, $PT_p \geq T_1$

The work law is

$$T_p \geq T_1/P$$
A P-processor ideal parallel computer cannot run faster than a machine with unlimited number of processors.

However, a computer with unlimited number of processors can emulate a P-processor machine by using simply $P$ of its processors. Therefore,

$$T_p \geq T_\infty$$

which is called the span law.
The **speed up** of a computation on $P$ processors is defined as $T_1 / T_p$

The **parallelism** of a multithreaded computation is given by $T_1 / T_\infty$
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

$$T_P \leq T_1 / P + T_\infty$$

[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 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

$$P(\lfloor T_1/P \rfloor + 1) = P\lfloor T_1/P \rfloor + P$$

$$= T_1 - (T_1 \mod P) + P$$

$$> T_1$$

as this exceeds the total work required by the computation, this is impossible.
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. q.e.d.
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.: The $T_P^*$ be the running time produced by an optimal scheduler. Let $T_1$ be the work and $T_\infty$ be the span of the computation. Then $T_P^* \geq \max(T_1 / P, T_\infty)$. By the theorem, 

$$T_P \leq T_1 / P + T_\infty \leq 2 \max(T_1 / P, T_\infty) \leq 2 T_P^*$$
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.
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 $T_1 / P$.

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.
Back to Fibonacci
Parallel algorithm to compute Fibonacci numbers:

Fibonacci(n)
    if n < 2 then return n;
    x = spawn Fibonacci(n-1);  // parallel execution
    y = spawn Fibonacci(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 compute the running time of the serialized algorithm.

$$T_1 = \Theta\left( \left(\frac{1+\sqrt{5}}{2}\right)^n \right)$$
Span of Fibonacci

Recall that the span $T_\infty$ in the longest path in the computational DAG. Since Fibonacci(n) spawns

- Fibonacci(n-1)
- Fibonacci(n-2)

we have

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

which yields $T_\infty(n) = \theta(n)$. 
The parallelism of the Fibonacci computation is

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

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.
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.
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 \( 2 \) do

\[ x = x + 1 \]

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
Matrix Multiplication
Matrix Multiplication

Recall that one can multiply nxn matrices serially in time $\theta(n^{\log 7}) = O(n^{2.81})$ using Strassen’s divide-and-conquer method.

We will use multithreading for a simpler divide-and-conquer algorithm.
Simple Divide-and-Conquer

To multiply two $n \times n$ matrices, we perform 8 matrix multiplications of $n/2 \times n/2$ matrices and one addition of $n \times n$ matrices.
Matrix-Add(C, T, n):
   // Adds matrices C and T in-place, producing C = C + T
   // n is power of 2 (for simplicity).
   if  n == 1:
      C[1, 1] = C[1, 1] + T[1, 1]
   else:
      partition C and T into (n/2)x(n/2) submatrices
      spawn Matrix-Add(C_{11}, T_{11}, n/2)
      spawn Matrix-Add(C_{12}, T_{12}, n/2)
      spawn Matrix-Add(C_{21}, T_{21}, n/2)
      spawn Matrix-Add(C_{22}, T_{22}, n/2)
      sync
Matrix Multiplication

Matrix-Multiply(C, A, B, n):
// Multiplies matrices A and B, storing the result in C.
// n is power of 2 (for simplicity).
if n == 1:
    C[1, 1] = A[1, 1] · B[1, 1]
else:
    allocate a temporary matrix T[1...n, 1...n]
    partition A, B, C, and T into (n/2)x(n/2) submatrices
    spawn Matrix-Multiply(C_{11},A_{11},B_{11}, n/2)
    spawn Matrix-Multiply(C_{12},A_{11},B_{12}, n/2)
    spawn Matrix-Multiply(C_{21},A_{21},B_{11}, n/2)
    spawn Matrix-Multiply(C_{22},A_{21},B_{12}, n/2)
    spawn Matrix-Multiply(T_{11},A_{12},B_{21}, n/2)
    spawn Matrix-Multiply(T_{12},A_{12},B_{22}, n/2)
    spawn Matrix-Multiply(T_{21},A_{22},B_{21}, n/2)
    spawn Matrix-Multiply(T_{22},A_{22},B_{22}, n/2)
    sync
Matrix-Add(C, T, n)
The work $T_1(n)$ of matrix multiplication satisfies the recurrence

$$T_1(n) = 8 \ T_1(n/2) + \Theta(n^2) = \Theta(n^3)$$

by case 1 of the Master theorem.
Span of Matrix Multiplication

The span $T_\infty(n)$ of matrix multiplication is determined by
- the span for partitioning $\Theta(1)$
- the span of the parallel nested for loops at the end $\Theta(\log n)$
- the maximum span of the 8 matrix multiplications

$$T_\infty(n) = T_\infty(n/2) + \Theta(\log n)$$

This recurrence does not fall under any of the cases of the Master theorem. One can show that $T_\infty(n) = \Theta((\log n)^2)$
Parallelism of Matrix Mult

The parallelism of matrix multiplication is given by

\[ \frac{T_1(n)}{T_\infty(n)} = \Theta(n^3 / (\log n)^2) \]

which is very high.