Serial vs. Parallel Algorithms

Serial algorithms are suitable for running on sequential uniprocessor computers. These sequential computers are not built anymore. We live in the age of parallel computers, where multiple instructions are executed at the same time. These parallel computers come in different forms:

- **Single multi core chips.** A core is a full-fledged processor. Each core can access a single shared memory.
- **Single multi core chips with accelerators.** An accelerator is a special co-processor that can execute certain (simple) codes in parallel, e.g., a vector processor that executes the same instruction on an array of values.
- **Distributed memory multi-computers,** where each processor’s memory is private, and processors communicate via an interconnection network.

We will concentrate on the first class: multi-core shared memory computers.
Dynamic Multithreading

Programs can specify parallelism through:

1. **Nested Parallelism**, where a function call is "spawned", allowing the caller and spawned function to run in parallel. We also call this **Task Parallelism**.

2. **Loop Parallelism**, where the iterations of the loop can execute in parallel.

These parallel loop iterations and tasks are executed by "virtual processors" or **threads**. Exactly when a thread executes and on which core it executes is not decided by the programmer, but by the run time system, which coordinates, schedules and manages the parallel computing resources. This lightens the task of writing parallel programs, as we don't have to worry about data partitioning (shared memory) and task scheduling.

Parallel constructs

Parallel tasks are created by a **spawn** and at the end of the task's execution synchronized with the parent by a **sync**. Parallel tasks naturally follow the divide-and-conquer paradigm.

Parallel loops are created using **parallel** and **new** constructs (later).

Removing spawn, sync, parallel and new from the program brings back the original sequential code.

There is a growing number of dynamic multi threading platforms. E.g., in cs475 we study OpenMP (open multi processing), built on top of C, C++, or Fortran.
The basics of dynamic multithreading

Fibonacci sequence:

\[ F_0 = 0 \]
\[ F_1 = 1 \]
\[ F_n = F_{n-1} + F_{n-2} \quad n > 1 \]

Simple recursive solution:

\[
\text{Fib}(n) : \\
\text{if } n \leq 1 \\
\quad \text{return } n \\
\text{else} \\
\quad x = \text{Fib}(n-1) \\
\quad y = \text{Fib}(n-2) \\
\quad \text{return } x+y
\]

Why do you not want to compute Fibonacci for large \( n \) this way?

How many nodes in this tree? (order of magnitude)

How would you write an efficient Fibonacci?

---

Run time of \( \text{Fib}(n) \)

\( T(n) \) denotes the run time of \( \text{Fib}(n) \):

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

the two recursive calls and

some constant time split and combine extra work

Claim: \( T(n) = \Theta(F_n) \)

Proof: strong induction.
Base: all constants, OK.
Step: assume

\[
T(m) = \Theta(F_m) \leq a F_m - b \quad a, b \text{ non negative constants, } 0 \leq m < n
\]

Then:

\[
T(n) \leq a F_{n-1} - b + a F_{n-2} - b + \Theta(1) = a(F_{n-1} + F_{n-2}) - b - \Theta(1)
\]

\[
= a F_n - b - \Theta(1) \leq a F_n - b
\]

In fact, we can show that \( T(n) = \Theta(\varphi^n) \) \quad \varphi = (1 + \sqrt{5})/2 \quad (CS420)
Parallel Fibonacci

```
P-Fib(n):
  if n<=1
    return n
  else
    x = spawn P-Fib(n-1) // spawn
    y = P-Fib(n-2) // call
    sync Call: caller waits until the called returns
               Spawn: spawner (parent) may execute at the same time as
                      spawned (child), until spawner encounters a sync
    return x+y
```

spawn: the caller (parent) can compute on in parallel with the called (child); it does not have to, but it may (up to the RTS when and where to schedule tasks)

sync: the parent must wait for all its spawned children to have completed, before proceeding. The sync is required to avoid summing x+y, before x is computed

return: in addition to explicit sync-s, return statement execute a sync implicitly, so that the parent waits for its children to complete

A Multithreaded execution model

The multithreaded computation can be viewed as executing a Directed Acyclic Graph G=(V,E), called computation DAG

Example: computation DAG for P-Fib(4)

Box: Procedure instance
Light: spawned
Dark: called in parent
Circle is strand is sequence of non control instructions
Black: base case or code up to spawn
Grey: call in parent
White: code after synch
Arrows: control spawn, call, sequence, return
Fat arrows: critical path

Assuming each strand takes one time unit (total) work equals 17 time units
span equals 8 (#critical path strands)
A Multithreaded execution model

The multithreaded computation can be viewed as executing a Directed Acyclic Graph $G=(V,E)$, called computation DAG, which is embedded in the call tree.

**Edge** $(u,v)$: $u$ executes before $v$.
- $(u,v)$ indicates a dependency:
  - if a node (strand) has two successors, one of them is spawned.
  - if a strand has multiple predecessors, they sync before execution continues.
- If there is a path from $u$ to $v$, they execute in series, otherwise they execute in parallel.
- Spawn and call edges point downward.
- Horizontal (continuation) edges indicate that the parent may keep computing while spawn executes. Return edges point up.
- Execution starts in a single initial strand (which one?) and ends in a single final strand (which one?)

Impact of schedule

Unfolded DAG for PF-3

<table>
<thead>
<tr>
<th>Schedule 1</th>
<th>Schedule 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td>3 5 7</td>
</tr>
<tr>
<td>P1</td>
<td>1 2 4 6 8 9</td>
</tr>
<tr>
<td>time</td>
<td>1 2 3 4 5 6</td>
</tr>
</tbody>
</table>

Idle time: number of empty slots (processor not busy) in schedule:
- schedule 1: 3
- schedule 2: 5

2 Processors
Performance Measures

**Work**: the total time to execute the program sequentially. Assuming 1 time unit per strand, this is the number of nodes in the DAG.

**Span**: longest time to execute the strands along any path in the tree, i.e., the critical path length of the DAG.

The run time of the program depends also on schedule and number of processors.

Performance Measures: time

**Work**: the total time to execute the program sequentially. Assuming 1 time unit per strand, this is the number of nodes in the DAG.

**Span**: longest path length of the DAG.

\[ T_1 : \text{the time to execute the program with 1 processor (}T_1=\text{work}) \]

\[ T_P : \text{the time to execute the program with } P \text{ processors} \]

As we have seen, different schedules can sometimes take different time, but we always assume greedy scheduling: if there are \((\geq 1)\) strands ready and a processor is available, a strand will be executed. (Which strand depends on the scheduler.)

Simplifying assumption: We are assuming no time cost for communication between the strands or memory accesses.

We call this model of computation ideal.
Work Law and Span Law

Work Law:
in one step P processors can do at most P units of work:

\[ T_P \leq T_1/P \]

Span Law:

\( T_\infty \): the time to execute the program with unlimited # processors

\( T_\infty = \text{span} \) is less or equal the time to execute the program with a fixed #processors P

\[ T_\infty \leq T_P \text{ or } T_P \geq T_\infty \]

Performance Measures: parallelism and speedup

\( S_P \): speedup with P processors: \( T_1 / T_P \).

(Average) Parallelism: \( T_1 / T_\infty \) (sometimes called \( \pi \) (pi)):

- average amount of work that can be done per time step
- or, maximum possible speedup (when no idle time exists)

With P processors you can only go P times faster than with 1 processor:

\[ S_P \leq P \]

linear speedup: \( S_P = fP \) \((0 < f \leq 1)\)

ideal speedup: \( f=1 \) or \( S_P = P \)

(no idle time, all processors busy all the time)

When \( P > \pi \) there will be idle time and thus no perfect speedup.
Exercise

Fill in

$T_1$:

$T_\infty$:

Is there idle time for: $P=1$, $P=2$, $P=3$?

Create a schedule for $P=3$:

$P_3$

$P_2$

$P_1$

Time

$T_3$, $S_3$

Is $T_3 < T_\infty$? Explain

Bound on $T_P$

We consider greedy schedulers only.

If there are at least $P$ strands available in a time step, all processors execute, and we call this a complete step.

If there are fewer than $P$ strands available in a time step, some processors will be idle, and we call that an incomplete step.

From the work law we know that at best $T_P = T_1/P$

From the span law we know that at best $T_P = T_\infty$
Theorem: bound on $T_P$

**Theorem:** $T_P \leq T_1/P + T_{\infty}$

**Proof:**
- There can be at most $\lfloor T_1/P \rfloor$ complete steps, otherwise there would be more than $T_1$ work.
- There can be at most $T_{\infty}$ (critical path length) incomplete steps. This happens when all steps are incomplete in which case in every step the remaining critical path length is decreased.
Steps are either complete or incomplete, therefore: $T_P \leq T_1/P + T_{\infty}$

QED

We define **slackness** as $(T_1/T_{\infty})/P$. Slackness is the factor by which program parallelism $\pi$ exceeds machine parallelism $P$. More slackness, more idle time. If all steps are incomplete, the number of processors is larger than the maximum parallelism in the program.

---

**Exercises**

1. Draw a task graph $G$ with 8 nodes, where every step is incomplete for $P=2$. Determine $T_2$, $T_{\infty}$

2. Use the schedule for $P=3$ from the previous exercise
   - Determine #incomplete steps, #complete steps
   - Determine $T_1/P$, $T_{\infty}$, $T_P$

   - Verify the theorem for this case
Exercises cont'

3 - How many critical paths?

- Critical path length?

- For $P = 1, 2$ and $3$ draw a schedule and check that for each incomplete step the critical path length decreases.

- Does the critical path length ONLY decrease on incomplete steps?

Corollary:

$T_P$ of any computation scheduled by a greedy scheduler is within a factor of 2 of optimal

Proof: Let $T_P^*$ be the run time of an optimal schedule

Work law: $T_P^* \geq T_1 / P$

Span law: $T_P^* \geq T_\infty$

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

For any $P$ processor computation we have the theorem:

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

$\leq 2 \max(T_1 / P, T_\infty)$

$\leq 2 T_P^*$

QED

Also, if $P \ll T_1 / T_\infty$ i.e., $P \ll n$ (<< much less than) then most steps are complete and thus $T_P \sim T_1 / T_\infty$ (~ is about), i.e. $S_{\sim P}$ i.e. the speedup is near perfect. The scheduling algorithm has thus a low impact on the performance.
Composing computations

We can compose two computations $A$ and $B$ in series or in parallel.

In series: $A$ is followed by $B$

- Work: $T_1(A \cup B) = T_1(A) + T_1(B)$
- Span: $T_\infty(A \cup B) = T_\infty(A) + T_\infty(B)$

In parallel: $A$ and $B$ execute in parallel

- Work: $T_1(A \cup B) = T_1(A) + T_1(B)$
- Span: $T_\infty(A \cup B) = \max(T_\infty(A), T_\infty(B))$

Critique of the ideal execution model

Why are the previous observations highly (unrealistically) optimistic?

1. Communication between strands is NOT free of time cost. Determining that a strand is ready for execution, and starting it on an available processor, takes time.

2. Accessing memory is not free; it takes (A LOT OF) time. In modern computers memory accesses take many processor clock cycles.

This is why modern computers have a complex cache architecture.
Analyzing multithreaded computation: Fibonacci

\[ P\text{-Fib}(n): \]
\[ \text{if } n \leq 1 \]
\[ \quad \text{return } n \]
\[ \text{else} \]
\[ \quad x = \text{spawn } P\text{-Fib}(n-1) \quad // \text{spawn} \]
\[ \quad y = P\text{-Fib}(n-2) \quad // \text{call} \]
\[ \quad \text{sync} \]
\[ \quad \text{return } x+y \]

- Work: we already did this: execution time of Fib:
  \[ T_1(n) = \Theta(\phi^n) \quad \phi = (1+\sqrt{5})/2 \]
- Span: spawn \( P\text{-Fib}(n-1) \) and call \( P\text{-Fib}(n-2) \) can run in parallel:
  \[ T_\infty(n) = \max(T_\infty(n-1),T_\infty(n-2)) + \Theta(1) = T_\infty(n-1) + \Theta(1) \]
  \[ T_\infty(n) = \Theta(n) \]
- Parallelism: \( n = T_1(n)/ T_\infty(n) = \Theta(\phi^n / n) \) which grows fast, and so near perfect speedup can be achieved. BUT!! WHAT IS WRONG HERE? (hint: It is often easy to write inefficient parallel algorithms 😞)

Reductions: First step to a realistic parallel Fibonacci

How fast can we add up an array of \( n \) numbers?
- Hint divide and conquer
  - divide into halves;
  - (recursively) add them up
  - add the result

\[ \text{Reduction(lo, hi):} \]
\[ \text{if } lo = hi \]
\[ \quad \text{return } A[lo] \]
\[ \text{else} \]
\[ \quad \text{mid} = (hi-lo)/2 \]
\[ \quad x = \text{Reduction(lo, mid)} \]
\[ \quad y = \text{Reduction(mid+1, hi)} \]
\[ \quad \text{return } x+y \]

- Reduction has (work) complexity \( \Theta(n) \)
Parallel Reductions: spawn-sync one child

\[ P\text{-Reduction}(lo, hi): \]
\[ \text{if } lo = hi \]
\[ \text{return } A[lo] \]
\[ \text{else} \]
\[ \text{mid} = (hi-lo)/2 \]
\[ x = \text{spawn } P\text{-Reduction}(lo, mid): \]
\[ y = P\text{-Reduction(mid+1, hi):} \]
\[ \text{sync} \]
\[ \text{return } x+y \]

- Work remains $\Theta(n)$
- Span is $\Theta(\log n)$
- Can be used for any associative operator, not just addition.

Scan: The second step

Instead of a single result, compute an array (prefix sum):
- i-th element of the output is the sum of the first i elements of the input array
- divide into halves; (recursively) add them up, add the result

\[ S\text{-Scan}(lo, hi, X): \]
\[ \text{if } lo = hi \text{ return } X[lo] \]
\[ \text{else} \]
\[ \text{mid} = (h-lo)/2 \]
\[ Y[1:mid] = S\text{-scan}(lo, mid) \]
\[ Y[mid+1:hi] = S\text{-scan}(mid+1, hi) \]
\[ /\text{just a for loop} \]
\[ \text{return } Y \]

- Work complexity? $\Theta(n)$

Now parallelize with spawn sync
Digressions: Parallel Loops

The `parallel` keyword before a for statement indicates that all the iterations of the for loop can execute in parallel.

Example: matrix vector product $Y_i = \sum_{j=1}^{n} a_{ij} x_j$ for $i = 1..n$

Each $Y_i$ can be computed by spawning a loop iteration $i$. However, each spawned iteration needs a private version of $j$ ($j$ cannot be shared, why?) This is expressed using the `new` keyword:

Mat-Vec($A, x$):

- $n=A.rows$
- $y$ float[$n$]
- parallel for $i = 1$ to $n$
  - $y_i = 0$
- parallel for $i = 1$ to $n$ // parallel loop
  - for new $j = 1$ to $n$ // sequential loop
    - $y_i = y_i + a_{ij} x_j$
- return $y$

Mat-Vec Performance

The parallel for can be compiled into a divide and conquer tree of spawned processes much like merge sort. (There are many other ways to compile this type of loop.)

1. Work: each internal node [lo,up] does constant spawn, compute, call work. There are $n-1$ of these nodes, so set up work is $\theta(n)$. Each of the $n$ leaves does $\theta(n)$ work. Hence the work is $\theta(n^2)$

2. Span: To set up the tree takes span $\theta(\log(n))$. The bulk of the computation is in the (internally sequential) leaves. All leaves can run in parallel and take span $\theta(n)$, which dominates building the tree. Hence the span is $\theta(n)$.
Parallelizing the scan

```python
S-Scan(lo, hi, X):
if lo = hi return X[lo]
else
    mid = (hi-lo)/2
    Y[1:mid] = spawn S-scan(lo, mid)
    Y[mid+1:hi] = S-scan(mid+1, hi)
    sync
    //just a for loop
return Y
```

- Use parallel `spawn-sync` across recursion, and `parallel for` for the result-update
- **Work:** Remains $\Theta(n)$
- **Span:** Now $\Theta(lg n)$