Week 12

Dynamic Multi-Threading

Cormen et. al., Chapter 27

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. Also called Shared Memory Multiprocessors.
- **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.

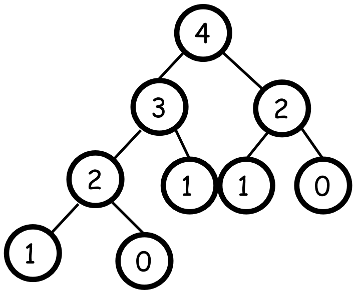
There is a growing number of dynamic multi threading platforms. E.g., in cs475 we study OpenMP (open multi processing), built on top of C, C++, or Fortran.

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:

```
Fib(n) :
   if n<=1
      return n
   else
     x = Fib(n-1)
     y = Fib(n-2)
     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 Fib(n)

T(n) denotes the run time of 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
$T(m) = \Theta(F_m) \leq aF_m - b$ a,b non negative constants, $0 \leq m < n$
Then: $T(n) \leq aF_{n-1} - b + aF_{n-2} - b + \Theta(1) = a(F_{n-1} + F_{n-2}) - b - (b - \Theta(1))$
$= aF_n - b - (b - \Theta(1)) \leq aF_n - b$

In fact, we can show that $T(n) = \Theta(\varphi^n)$ $\varphi = (1+\text{Sqrt}(5))/2$ (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
     return x+y
```

Because return x+y needs (depends on) results of both spawn and call

Why do we sync here?

In a sequential call, the caller waits until the called returns, whereas in a parallel spawn, the spawner (parent) may execute at the same time as spawned (child), until the spawner encounters a sync.

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)

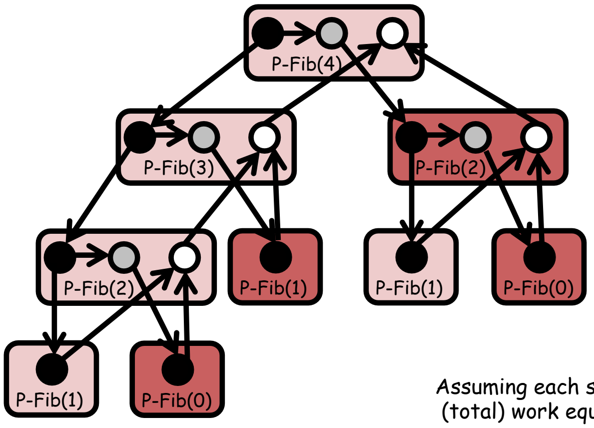
sync: the parent must wait for all its spawned children to have completed, before proceeding. The sync is required to avoid summing x+y, before x is computed

return: in addition to explicit sync-s, a return statement executes 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 a **strand: a** sequence of non control instructions
Black: base case or code up to spawn
Grey: call in parent
White: code after sync

Arrows: control:
spawn, call, sequence, return

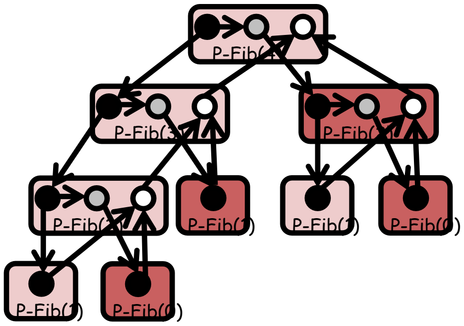
Fat arrows: critical path: the longest path through the computation

Work: total number of strands
Span: number of strands on a 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.

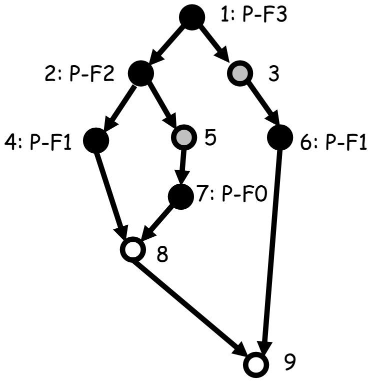


Edge (u,v): u executes before v
(u,v) indicates a dependency:
  if a node (strand) has two successors,
    one of them is spawned
  if a strand has multiple predecessors,
    they sync before execution continues
If there is a path from u to v, they execute in series, otherwise they execute in parallel

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

2 Processors

```
Schedule 1
P2       3  5  7
P1     1  2  4  6  8  9
-----------------------
time  1  2  3  4  5  6
```

```
Schedule 2
P2       3  6
P1     1  2  4  5  7  8  9
-----------------------
time  1  2  3  4  5  6  7
```

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

Performance Measures

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

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

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

Intuitive interpretation of work and span:
work models **sequential** execution time
span models fully **parallel** execution time

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. This is the fully parallel execution time (if there are always enough processors to execute a task immediately)

T_1: the time to execute the program with 1 processor (T_1=work)
T_P: the time to execute the program with P processors

As we have seen, different schedules can sometimes take different time, but we always assume **greedy scheduling:** if there are (≥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**. **WHY IDEAL?**

Because we assume no time cost for
memory access or communication between
the processors executing the strands

Work Law and Span Law

Work Law:

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

$T_P \geq T_1/P$

Span Law:

T_∞: the time to execute the program with unlimited #processors (T_∞ = span) is less or equal the time to execute the program with a fixed #processors P

$T_\infty \leq T_P$ 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_∞ (sometimes called Π (pi)):

- average amount of work that can be done per time step

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

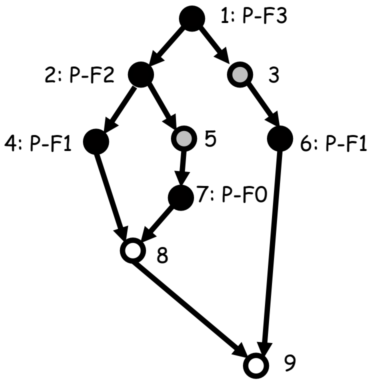
$S_P \leq P$

linear speedup: $S_P = f\, P$ $(0 < f \leq 1)$

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 hence non-ideal speedup

Exercise



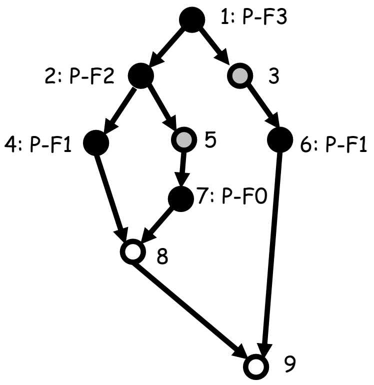
Unfolded DAG for PF-3

Fill in

T_1:
T_∞:
Π :

Is there idle time for:
P=1 P=2 P=3 ?
P_3
P_2
P_1
Create a schedule for P=3
Time/speedup p=3
T_3: S_3:
Is $T_4 < T_3$? explain

Exercise



Unfolded DAG for PF-3

T_1: 9
T_∞: 6 (nodes on critical path: 1,2,5,7,8,9)
π: 9/6 = 1.5

Is there idle time for:
P=1 P=2 P=3 ?
P=3: YES
P=2: YES
P=1: NO (never for P1)
Create a schedule for P=3

```
          6
      3   5
1     2   4   7   8   9
```

T_3: 6 S_3: 9/6=1.5
Is $T_4 < T_3$? NO The fourth processor is unnecessary. Never are there more than 3 parallel strands.

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 ($T_P \geq T1/P$)we know that at best $T_P = T_1/P$
From the span law ($T_P \geq T_\infty$) 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_∞ (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

Corollary of theorem $T_P \le T_1/P + T_\infty$

T_P of any computation scheduled by a greedy scheduler is within a factor of 2 of optimal schedule for p processors, no matter which greedy schedule

Proof: Let T^*_P be the run time of an optimal schedule
  Work law: $T^*_P \ge T_1/P$ Span law: $T^*_P \ge T_\infty$
  therefore $T^*_P \ge \max(T_1/P, T_\infty)$

For any P processor computation we have the theorem:

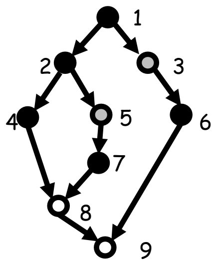
$$\begin{aligned} T_P &\le T_1/P + T_\infty \\ &\le 2\max(T_1/P, T_\infty) \\ &\le 2T^*_P \end{aligned}$$

QED

In other words: the scheduling algorithm has a low impact on the performance.

Exercise

Use the schedule for P=3 from the previous exercise.
- Determine #incomplete steps, #complete steps
- Determine T_1/P, T_∞, T_P
- Verify the theorem for this case



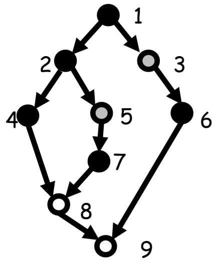
Exercise

Use the schedule for P=3 from the previous exercise.

- #incomplete steps: 5, #complete steps: 1

- T_1/P: 9/3=3 , T_∞: 6, T_3: 6

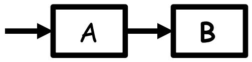
- Verify the theorem for this case
  $T_3 = 6$
  $T_\infty = 6$
  Theorem $T_P \leq T_1/P + T_\infty$
  $T_3 = 6 <= 3 + 6 = 9$



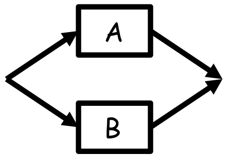
Composing computations

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, as compared to executing a strand of arithmetic instructions. In modern computers instruction execution takes 1 clock cycle, whereas memory accesses take many processor clock cycles; we call this phenomenon the

   MEMORY WALL.

   This is why modern computers have a complex cache architecture.

Parallel (Recursive) Fibonacci

```
P_Fib(n):
  if n<=1
     return n
  else
     x = spawn P_Fib(n-1) // spawn
     y = P_Fib(n-2)           // call
     sync
     return x+y
```

-Work (slide 6): execution time of Fib exponential: $\Theta(\varphi^n)$

-Span: spawn P_Fib(n-1) and call P_Fib(n-2) can run in parallel:

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

- Parallelism: $\pi = T_1(n) / T_\infty(n) = \Theta(\varphi^n / n)$ which grows fast, so near perfect speedup can be achieved.

BUT: WHAT IS THE PROBLEM?

This is the **very inefficient** recursive implementation!
It is easy to write an inefficient highly parallel program ☺

Parallel Loops

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

It is legal to parallelize a for loop if the iterations are **independent** of each other, i.e., an iteration does not use values computed in previous iterations.

Example of legal parallelization:

    for i in 0 to n-1:   C[i] = A[i]+B[i]

can be made

    parallel for i in 0 .. n-1:   C[i] = A[i]+B[i]

Example of illegal parallelization:

    for i in 0 to n-1:   A[i] = A[i-1]+B[i]

Cannot be made parallel

Here iteration i uses a value computed in iteration i-1. We call this:

iteration i is dependent on iteration i-1.

So iteration i-1 must be executed before iteration i. If we don't do this, and insist on executing the iterations in parallel we will create a data race and (see pp 787-790 of the text)

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

Each Y_i can be computed in parallel by an independent a loop iteration i:

```
Mat-Vec(A,x):
   n = A.rows
   y float[n]
   parallel for i = 1 to n
    y[i] = 0
   # for each row i compute the in-product(row i, X)
   parallel for i = 1 to n    # parallel for rows of A
       for new j = 1 to n     # sequential for j
           y[i] = y[i] + a[i,j] * x[j]
   return y
```

Because all inner j loops update j, j cannot be shared, Thus, each spawned iteration needs a private copy of j. This is expressed using the ***new*** keyword. Parallel for is often called "forall"

Mat-Vec Performance

- If p, the number of processors is very large (e.g., data center scale), a parallel for can be compiled into a divide and conquer tree of spawned processes much like merge sort (see below). But in practice, even if p is very large, each processor can independently (i.e., in constant time). determine "its chunk."
- The textbook analysis (pages 785-787) update:
  - Though it is possible to always compile parallel for using spawn-sync this is unnecessarily inefficient
  - parallel for is actually a more powerful and additional parallelization construct (see exercises in PQ9 DMT)
- **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 $\Theta(n^2)$
- **Span:** All the computation is in the (sequential) leaves. The leaves run in parallel and take $\Theta(n)$ time. Hence the span is $\Theta(n)$.

Recursive SCAN

SCAN (also called "Prefix sum"): given an array A, compute an array of X, where the i-th element of X is the sum of the first i elements of A. Divide into halves; (recursively) compute prefix sums and add the sum of the first half to each element of the second half.

```
Scan(lo, hi, A):
if lo = hi return A[lo]
else
  mid = (hi-lo)/2
  X[1:mid] = Scan(lo, mid)
  X[mid+1:hi] = Scan(mid+1, hi)
  X[mid+1:hi] = X[mid]+X[mid+1:hi] #for loop
return X
```

Work complexity: $W(n) = 2^* W(n/2) + n/2$ is $\Theta(n \lg n)$ (Master Theorem).
More than standard $\Theta(n)$ iterative scan:

```
X[1] = A[1]
for i = 2 to n: X[i] = X[i-1]+A[i]
```

But the iterative scan has a dependency, so cannot be parallelized.

Parallelized recursive Scan

```
P_Scan(lo, hi, A):
if lo = hi X[lo]= A[lo]
else
  mid = (hi-lo)/2
  X[1:mid] = spawn P_scan(lo, mid, A)
  X[mid+1:hi] = P_scan(mid+1, hi, A)
  sync
  X[mid+1:hi] = X[mid] + X[mid+1:hi]
```

This can be executed in parallel: all iterations are independent!!

- Use parallel spawn-sync across recursion, and parallel for to update X[mid+1:hi]
- Work: Θ(n lg n)
- Span: Θ(lg n)

We can scan n numbers in Θ(lg n) time with Θ(n) processors

Even better parallelization

- See https://en.wikipedia.org/wiki/Prefix_sum
- Tree based (work out on excel spreadsheet)
  - One pass up the tree to compute a reduction (and also save all partial sums contributing to that)
  - Second pass down the tree to update/repair the elements with the prefix results of everything to the left of the node
- Work is reduced to Θ(n)
- Span: Θ(lg n)

We can scan n numbers in Θ(lg n) time with Θ(n) processors

Back to Fibonacci

- Problem:
  - Compute an array of the first n Fibonacci numbers
  - Computing (only) the nth one is a special case of this problem.
- Lower bound:
  - Is it Θ(n), the size of the output?
  - No, processors can write outputs in parallel
- Recall the memo-Fib: in each iteration, update one value using the previous and pre-previous

```
F[0] = F[1] = 1;
for i in range(2:n)
    F[i] = F[i-1] + F[i-2]
    F[i-1] = F[i-1] //redundant copy
```

Key idea: reduction (pun intended)

How to reduce the Fib to a scan?

- Draw from the memory efficient memo-Fib: copy
  - new to previous
  - previous to pre-previous
- Maintain a 2-element vector
- Use a matrix notation, to express F[i] and F[i-1] in terms of previous vector F[i-1] and F[i-2]:

F[i] = 1 * F[i-1] + 1 * F[i-2]
F[i-1] = 1 * F[i-1] + 0 * F[i-2]

$$\begin{aligned}\begin{pmatrix} F_i \\ F_{i-1} \end{pmatrix} &= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{pmatrix} F_{i-1} \\ F_{i-2} \end{pmatrix} \\ &= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} * \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{pmatrix} F_{i-2} \\ F_{i-3} \end{pmatrix} \\ &= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} * \cdots * \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{pmatrix} F_1 \\ F_0 \end{pmatrix}\end{aligned}$$

Main question: is extra work worth it?

- Parallel algorithm does more work (Θ(lg n) factor)
- Uses Θ(n) processors
- Achieves faster time (Θ(lg n) time as opposed to Θ(n) so speedup is Θ(n/lg n)
- Can we do better?
  - Yes, see in CS475/CS575 (Brent's theorem)
- In practice best parallel algorithm may be "too fast" because
  - Both p and n grow asymptotically
  - But p grows much slower than n
- Chunking algorithm
  - Each process has a chunk of $\frac{n}{p}$ elements
  - Sequentially reduces its chunk
  - Processors cooperate to parallel scan result
  - Scan local array sequentially
  - Factor of two extra work
  - Work Law limit: ideal speedup possible

Is Scan an important problem?

Parallelize something that's inherently sequential!!!

It is representative of problems in signal/image processing (recursive filtering/convolution)

The sequential algorithm/program uses the standard incremental approach with (O(n)) work

The parallel scan breaks the sequential dependence, of the sequential span. It has a shorter span, but it performs more work

The parallel and sequential algorithms can be combined to achieve an efficient parallel implementation (CS475: Brent's theorem)

- top of the call tree parallel, bottom sequential.