Algorithms and Data Structures (CSci 115)

California State University Fresno
College of Science and Mathematics
Department of Computer Science
H. Cecotti

Learning outcomes

- Multithreading*
  - Example with the sequence of Fibonacci
- Memory Optimization for data structures*

* : Not in the final exam

Introduction

- Link between **software** and **hardware**
  - ➢Everything is connected
    - Importance of considering a **holistic** (global) approach
  - ➢→ Direct link with **Operating Systems** (OS)
    - 1 vs. Multi threads
    - Shared vs. Distributed memory
  - ➢→ Think about
    - **where** the application will be deployed
    - How often it will be used, how often the different actions will be used

Rationale

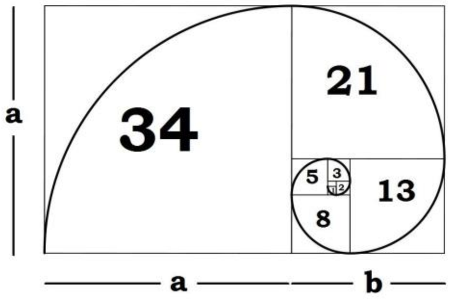
- **Considerations**
  - **Serial algorithms**
    - For running on a uniprocessor computer
      - Only 1 instruction executes at a time
  - **Parallel algorithms**
    - To run on a multiprocessor computer that permits multiple
      - → instructions to execute concurrently
  - **Parallel computer**
    - **Shared memory**
      - Each processor can directly access any location of memory
    - **Distributed memory**
      - Each processor's memory is private and an explicit message must be sent between processors in order for one processor to access the memory of another

Concurrency keywords

- **Concurrency keywords**
  - **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.
  - **Sync**
    - The procedure must wait as necessary for all its spawned children to complete before proceeding to the statement after the sync

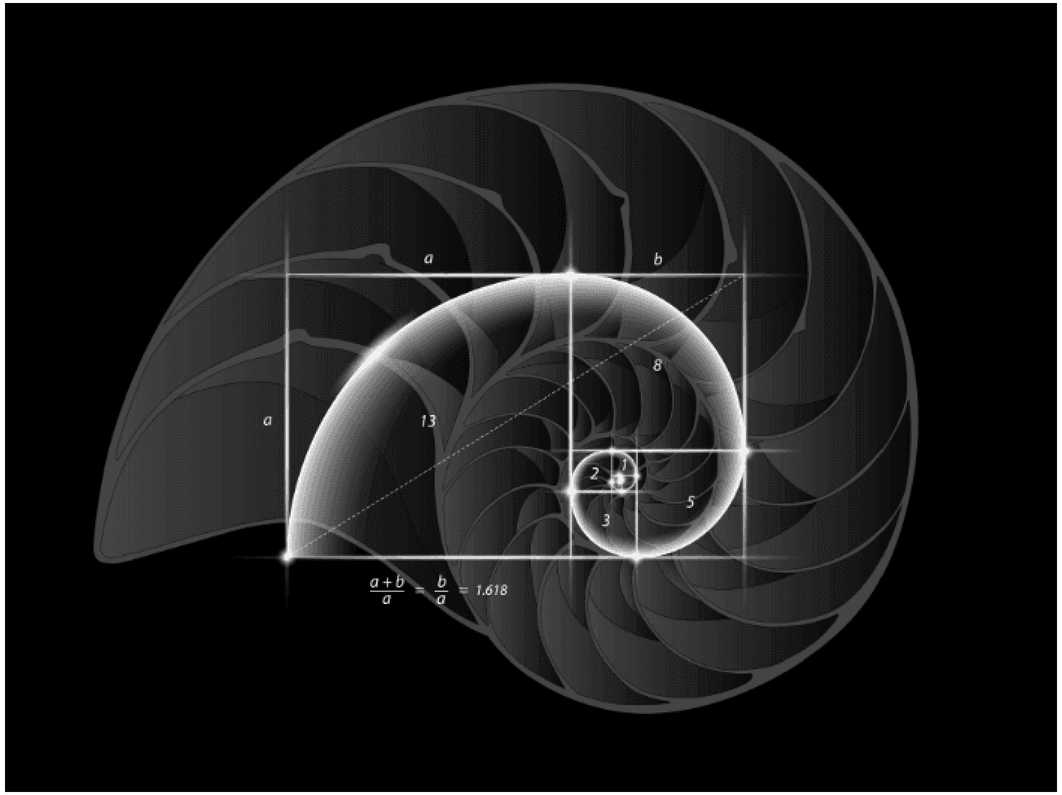
Back to Fibonacci (again)

- Fibonacci
  - *Golden ratio: 1.61803*
  - Sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...
  - One of the first examples given for recursive functions
  - Dynamic programming
    - Memoization
  - Fibonacci heaps




```
FIB(n)
if n ≤ 1
    return n
else x = FIB(n − 1)
    y = FIB(n − 2)
    return x + y
```

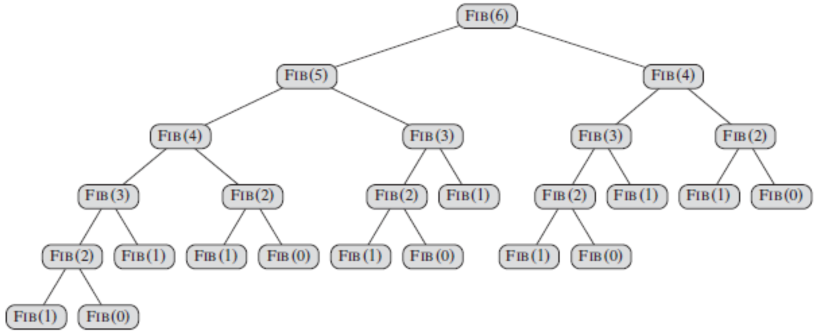
- Sequence:



Back to Fibonacci (again)

- Exploration of dynamic multithreading
  - ➢ With the sequence of Fibonacci
- Let T(n): the running time of Fibonacci(n)
  - ➢ Since this procedure contains 2 recursive calls and a constant amount of extra work
    - o $T(n) = T(n-1) + T(n-2) + \theta(1)$
    - o → $T(n) = \theta(F_n) = \theta(((1+\text{sqrt}(5))/2)^n)$
  - ➢ **Exponential growth**
    - o Particularly bad way to calculate Fibonacci numbers.
    - o How would you calculate the Fibonacci numbers?

```
FIB(n)
if n ≤ 1
    return n
else x = FIB(n − 1)
    y = FIB(n − 2)
    return x + y
```



Back to Fibonacci

- Implementation with matrixes

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

- To calculate F_n in O(log n) steps
  - by repeated squaring of the matrix
  - you can calculate the Fibonacci numbers with a **serial** algorithm.
- To illustrate the principles of parallel programming
  - Use the naive (bad) algorithm

Fibonacci Example

- 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;

Computation DAG

- Multi-threaded computation
  - Using the help of a computation **Directed Acyclic Graph** (DAG) G=(V,E).
    - V: instructions.
    - E: dependencies between instructions.
- An edge (u,v) is in E
  - the instruction u must execute **before** instruction v.
- A computation DAG G=(V,E)
  - It consists of :
    - The vertex set V: the threads of the program.
    - The edge set E contains an edge (u,v) if and only if the thread u need to execute before thread v.
  - **If** (ExistEdge(u,v))
    - **then** they are said to be (logically) in **series**
  - **If** (there is no thread)
    - **then** they are said to be (logically) in **parallel**

Strand and Threads

- **Definitions**
  - 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: a thread.

Edge Classification

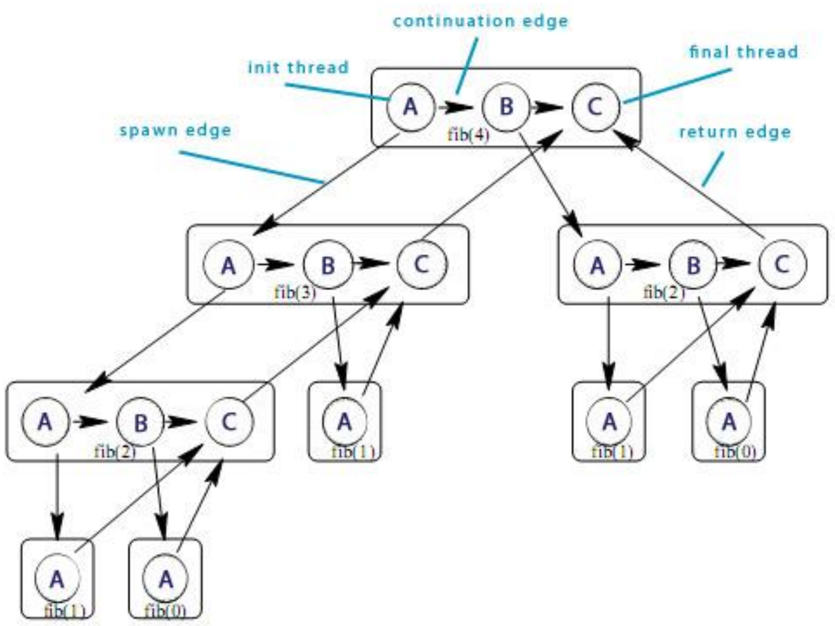
- A continuation edge (u,v) connects a thread u to its successor v within the same procedure instance.
- **If** (a thread u spawns a new thread v)
  - **Then** (u,v) is called a **spawn** edge.
- **If** (a thread v returns to its calling procedure **and** x is the thread following the parallel control)
  - **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**

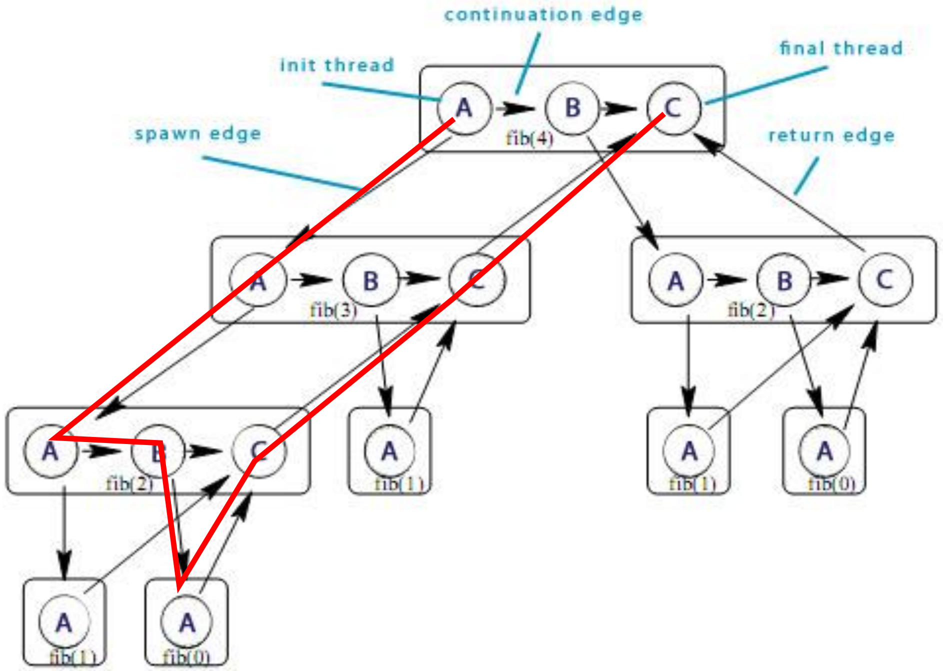
Performance Measures

- Definitions
  - Work of a multithreaded computation:
    - the total time to execute the entire computation on 1 processor.
    - Work
      - sum of the times taken by each thread
  - The span
    - longest time to execute the threads along any path of the computational DAG



Performance Measures

- In Fibonacci(4)
  - 17 vertices = 17 threads.
  - 8 vertices on longest path.
- Assuming
  - unit time for each thread
- We obtain:
  - Work = 17 time units
  - Span = 8 time units



Performance Measures

- The actual running time of a multithreaded computation depends on:
  - its **work** and **span**
  - on how many processors/cores are available
  - 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_∞ running time on unlimited processors

Definitions

- **Work law**
  - An ideal parallel computer with P processors can do at most P units of work.
  - Total work to do is T_1 → $PT_p >= T_1$
  - The work law: $T_p >= T_1/P$
- **Span law**
  - 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
    - → The span law is:
      - $T_p >= T_\infty$
- 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_∞

Scheduling

- The performance depends not just on the work and span
- In addition,
  - the strands must be
    - scheduled efficiently.
    - mapped to static threads
  - and the OS 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
  - **$T_P <= T_1 / P + T_\infty$**

- As the best we can hope for on P processors is:
  - $T_P = T_1 / P$ by the work law
  - $T_P = T_\infty$ by the span law
  - the sum of these 2 lower bounds

Proof (part 1)

- 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

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

- As this exceeds the total work required by the computation
  - This is impossible ☺

Proof (part 2)

- Now consider an incomplete step.
  - Let G be the DAG representing the entire computation.
  - Without loss of generality, we 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
- Let G'' be the subgraph
  - remaining to be executed **after** the completion of the incomplete step.

Proof (part 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’**.
- Idea
  - An incomplete step **decreases** the span of the unexecuted DAG by **1**.
  - → 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:
  - The 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
  - Then $T_P^* >= \max(T_1 /P, T_\infty)$
  - By the theorem,
    - $T_P <= T_1 /P + T_\infty <= 2 \max(T_1 /P, T_\infty) <= 2 T_P^*$

Slackness

- **Definitions**
  - 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)
  - **then** we cannot hope to achieve a linear speedup.

Speedup

- Let T_P be the running time of a multi-threaded 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.
- Proof:
  - If $P << (T_1 / T_\infty)$
  - then $T_\infty << T_1 / P$
  - → By the theorem, we have
    - $T_P <= T_1 /P + T_\infty \approx T_1 /P$
  - By the work law, we have
    - $T_P >= T_1 /P$
  - → $T_P \approx T_1 /P$, as claimed.

Work of Fibonacci

- We want to know the **work** and **span** of the Fibonacci computation
  - to compute the parallelism (work/span) of the computation.
- The work T_1 is straight forward
  - since it amounts to compute the running time of the serialized algorithm.
- $T_1 = \theta(((1+sqrt(5))/2)^n)$

Span of Fibonacci

- Recall that the span T_∞ 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 $T_\infty(n) = \theta(n)$.

Parallelism of Fibonacci

- The parallelism of the Fibonacci computation is
  - ➢ $T_1(n)/T_\infty(n) = \theta(((1+sqrt(5))/2)^n / n)$
  - ➢ which grows dramatically as n gets large.

- → Even on the largest parallel computers
  - ➢ A modest value of n suffices to achieve near perfect linear speedup

Data structure

- **The idea**
  - Keep heavily accessed data members near each other physically in memory a system's caching
  - If it’s declared together, it s easier to access it together
    - In the same function
- **Organization:**
  - Field reordering
    - In a struct with multiple elements,
      - keep the items that will be accessed together, together in the order of elements in the struct

Software prefetching

- **Definition**
  - Cache prefetching:
    - Technique used by computer processors to boost execution performance
    - How?: by fetching instructions/data
      - **from** their original storage in "**slow memory**"
      - **to** a "**faster local memory"** before it is actually needed
- **Approaches**
  - Not too early
    - Data may be evicted before use
  - Not too late
    - Data not fetched in time for use
  - Greedy

Conclusion

- To take advantage of the architecture:
  - Multi-threaded/multi processors/cores architectures
  - Memory available
  - Disk size available
- **Remark**
  - What is your job? What is the job of the compiler?
    - Modern compilers deal with many optimization
  - Your priority:
    - Depends on the project:
      - Team work, Maintenance, Code easy to read, update, maintain
      - Optimization…
- **Next session**:
  - Final conclusion of the CSc115 course + Revisions for the final

Questions ?

- Reading & Acknowledgement
  - Multithreaded Algorithms ,Introduction to Algorithms, 3rd Edition, Chapter 26.