Multithreaded Algorithms

Computational Model

- The vast majority of algorithms are **serial algorithms** suitable for running on a **uniprocessor** computer in which only one instruction executes at a time.

- **Parallel computers**—**computers with multiple processing units**—have become increasingly common, and they span a wide range of prices and performance.

- **Parallel algorithms**, which can run on a **multiprocessor** computer that permits **multiple instructions** to execute **concurrently**.

- There exist many **competing models of parallel computation** that are essentially different. For example, some parallel computers feature **shared memory**, where each processor can directly access any location of memory. Other parallel computers employ **distributed memory**, where 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.**

- With the advent of **multicore technology**, however, every new laptop and desktop machine is now a **shared-memory parallel computer** and the trend appears to be toward shared-memory multiprocessing.

Threads

- One common means of **programming** chip multiprocessors is **Threading**.

- Each thread maintains an associated **program counter** and can execute code independently of the other threads.

- Multithreading refers to a **central processing unit's ability to run many code threads simultaneously**, as long as the operating system allows it.

- There are **some examples of multithreading**, such as a **word processor** that displays a document uses different threads to perform tasks such as **checking the spelling and grammar** of the content and generating a pdf version of the document, **all while typing content in the document**.

- Multiple threads are used to **load material, display animations, play a video,** and so forth in multiple tabs corresponding to an internet browser.

- Although the operating system allows programmers to create and destroy threads, these operations are comparatively slow. Thus, for most applications, **threads persist for the duration of a computation, which is why we call them "static."**

Threads

Static and Dynamic Multithreading

Static threading means that the **programmer needs to specify how many processors to use at each point in advance**. When it comes to **evolving conditions**, this can be **inflexible**.

In **dynamic multithreading models**, the **programmer needs to specify opportunities for parallelism** and a **concurrency platform manages the decisions of mapping these opportunities to actual static threads**.

A **concurrency platform is a software layer that schedules, manages, and coordinates parallel-computing resources.**

Threads

Current parallel-computing practice:

Spawn

Instead of waiting for the child to finish, the **procedure instance** that executes the spawn (**the parent**) **may continue to execute in parallel with the spawned subroutine** (**the child**).

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 sub computations should run concurrently.

Sync

the **procedure must wait for the completion of all of its spawning children**. It is used **when one cannot proceed without pending results.**

Parallel

To indicate that **each iteration can be done in 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.**

The "**parallel**" and "**spawn**" keywords **do not impose parallelism**. They **just indicate that it is possible**.

Dynamic Multithreading

Example: Parallel Fibonacci

Let's take a serial algorithm and make it parallel. Here's an algorithm (non-parallel) for computing Fibonacci numbers:

Algorithm 1: Definition of Fibonacci numbers

$F_0 = 0$

$F_1 = 1$

$F_i = F_{i-1} + F_{i-2}$, for $i \geq 2$

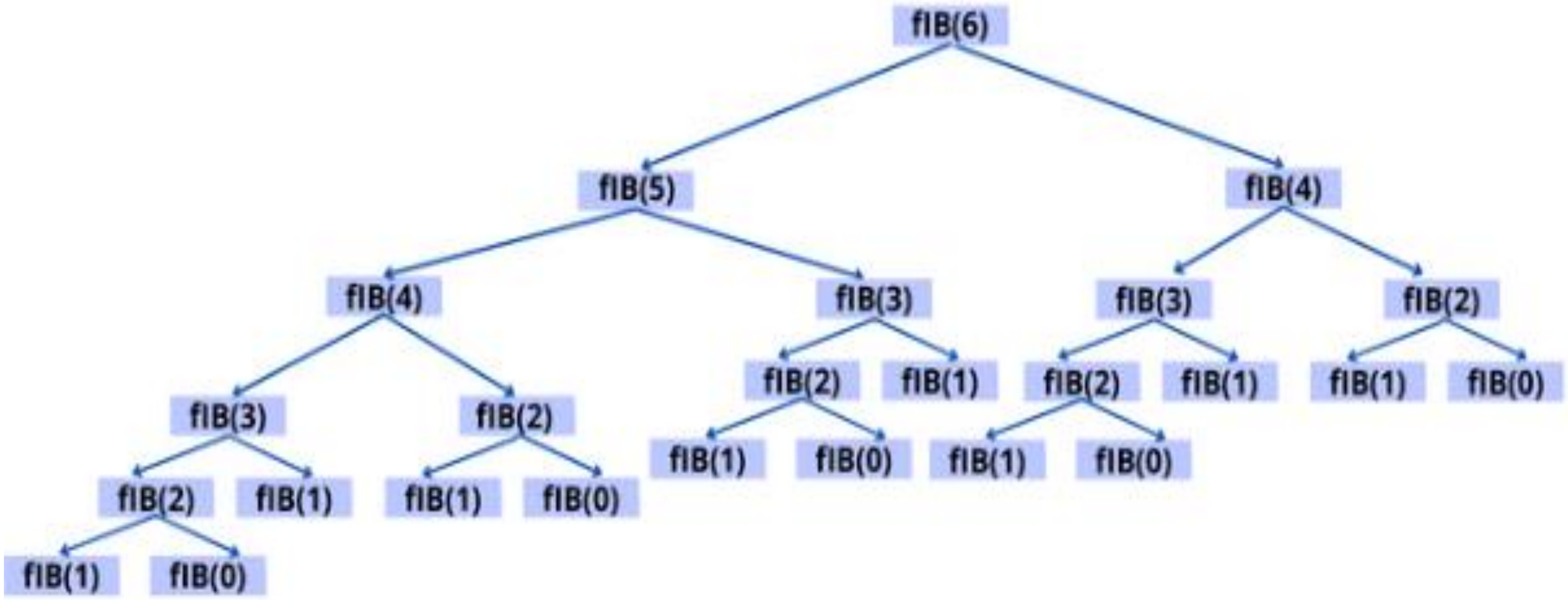
Algorithm 2: Fibonacci numbers (non-parallel)

```
FIB(n)
if n ≤ 1 then
    return n
else
    x = FIB(n-1)
    y = FIB(n-2)
    return (x+y)
end
```

Dynamic Multithreading

Example: Parallel Fibonacci

The recursion tree of the algorithm:



Dynamic Multithreading

Example: Parallel Fibonacci

Let's see what improvement we can get by computing the two recursive calls in parallel using the concurrency keywords:

Algorithm 2: Fibonacci numbers (non-parallel)

```
FIB(n)
if n ≤ 1 then
 |  return n
else
 |  x = FIB(n-1)
 |  y = FIB(n-2)
 |  return (x+y)
end
```

Observe that within FIB(n), the two recursive calls **FIB(n-1) and FIB(n-2),** respectively, are **independent of each other**: they could be called in either order, and the computation performed by one has no way affects the other. Therefore, the **two recursive calls can run in parallel.**

Algorithm 3: Fibonacci numbers (parallel)

```
P-FIB(n)
if n ≤ 1 then
 |  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
end
```

Nested parallelism occurs when the keyword spawn precedes a procedure call, as in line 3. It creates a concurrent process.
The semantics of a spawn differs from an ordinary procedure call in that the procedure instance that executes the spawn—the **parent—may continue to execute in parallel with the spawned subroutine**—its child—instead of waiting for the child to complete, as would normally happen in a serial execution.

Dynamic Multithreading

- Since the P-FIB procedure is recursive, these two subroutine calls **themselves create nested parallelism**, as do their children, thereby creating a potentially vast tree of subcomputations, all executing in parallel.

- The keyword **spawn does not say, however, that a procedure *must execute concurrently*** with its spawned children, only that **it *may***.

- *The concurrency keywords* express the ***logical parallelism of the computation, indicating which parts of the*** computation **may proceed in parallel**.

- At runtime, it is up to a ***scheduler to determine*** which sub computations actually run concurrently by assigning them to **available processors** as the computation unfolds.

- A procedure cannot safely use the values returned by its spawned children until after it executes a **sync statement. The keyword sync indicates that** the procedure must wait as necessary for all its spawned children to complete execution before proceeding to the statement after the **sync.**

Modeling Dynamic Multithreading

Formal model for describing parallel computations

Computation DAG

- A **computation DAG** (directed acyclic graph) will be used to model a multithreaded computation

$$\{G = (V, E)\}:$$

Vertices: vertices (V) in the graph represent the instructions. Think of each vertex as a strand (sequence of instructions containing no parallel control, such as sync, spawn, parallel, return from spawn)

Edges: edges (E) represent the dependencies between strands or instructions.

Modeling Dynamic Multithreading

Formal model for describing parallel computations

Edge Classification

Different types of edges:

Continuation edge (u,v)

Connects a thread **u** to its successor **v** within the same procedure instance

Spawn edge (u,v)

(u,v) is called a spawn edge when a thread **u** spawns a new thread **v** in parallel.

Return edge (v,x)

when a thread v returns to its calling process and x is the thread following the parallel control, the graph includes the return edge (v,x).

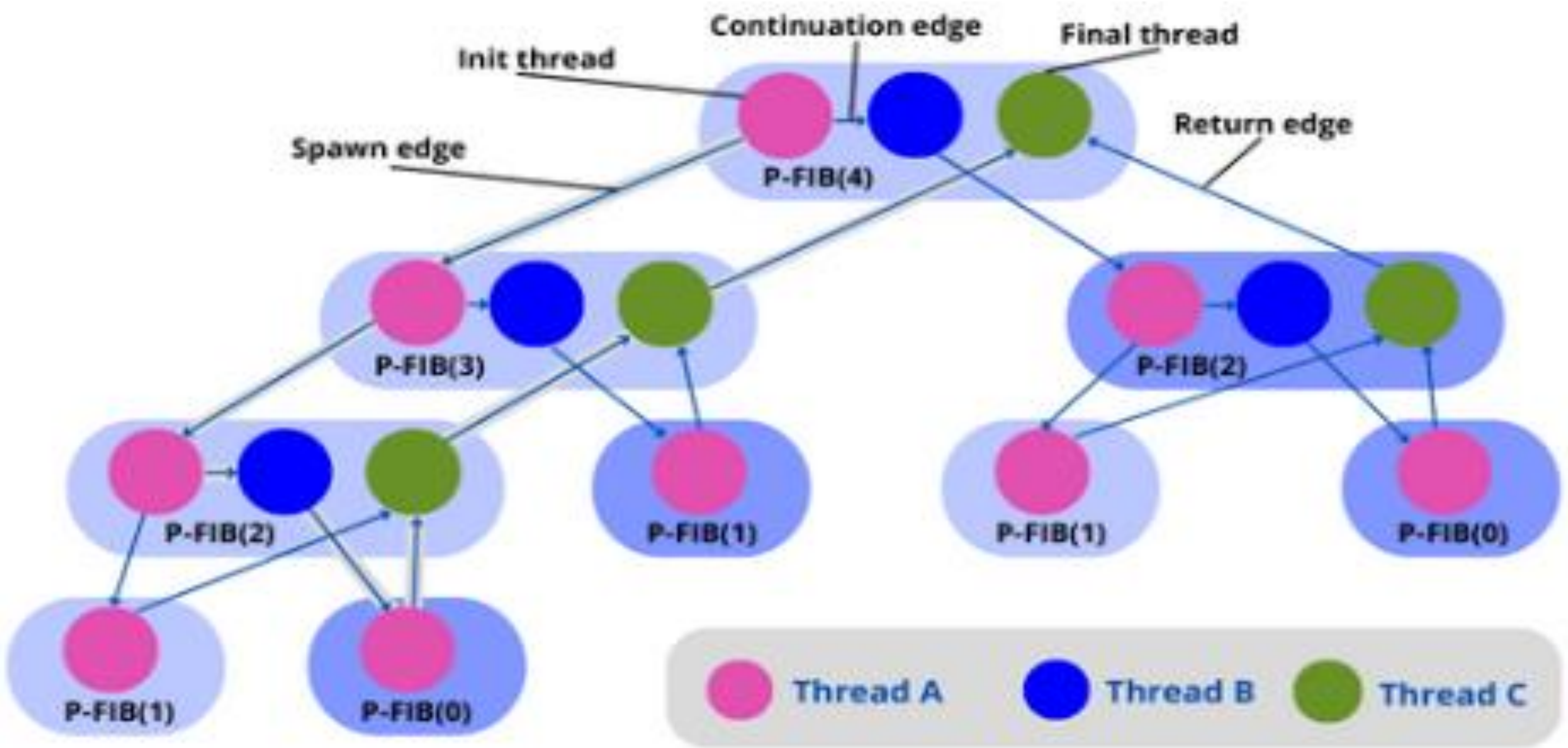
Modeling Dynamic Multithreading

Computation DAG: Parallel algorithm to compute Fibonacci numbers using threads:

Algorithm 4: Fibonacci numbers (parallel)

```
P-FIB(n)
if n ≤ 1 then
    return n // Thread A
else
    x = spawn P-FIB(n-1)
    y = spawn P-FIB(n-2) // Thread B
    sync
    return x + y // Thread C
end
```

computation DAG for P-FIB()



As we can see in the figure above, the purple thread in the block number {4} spawns another purple thread in the block number {3}. Then the purple thread in the block number {3} points to the purple thread in the block number {2}. The purple thread in the block number {2} points to the purple thread in the block number {2}. Then we point to the green thread in the block number {2} using the return edge. We continue pointing to the green thread in block number {3} and {4} until we reach the final thread

Dynamic Multithreading: Performance Measures

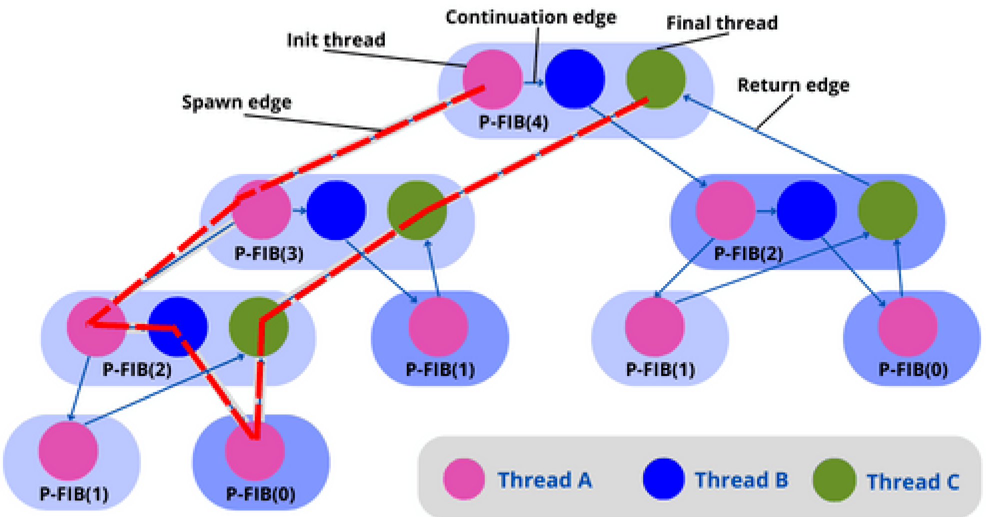
- **Span** S or $T_\infty(n)$. Number of vertices on the longest directed path from start to finish in the computation DAG. (The critical path).

- **Work** W or T_1 (n). Total time to execute the entire computation on one processor. Defined as the number of vertices in the computation DAG

- Tp(n) = Total time to execute entire computation with p processors

- Speed up = T1/Tp. How much faster it is.

- Parallelism = T1/ T_∞ . The maximum possible speed up.

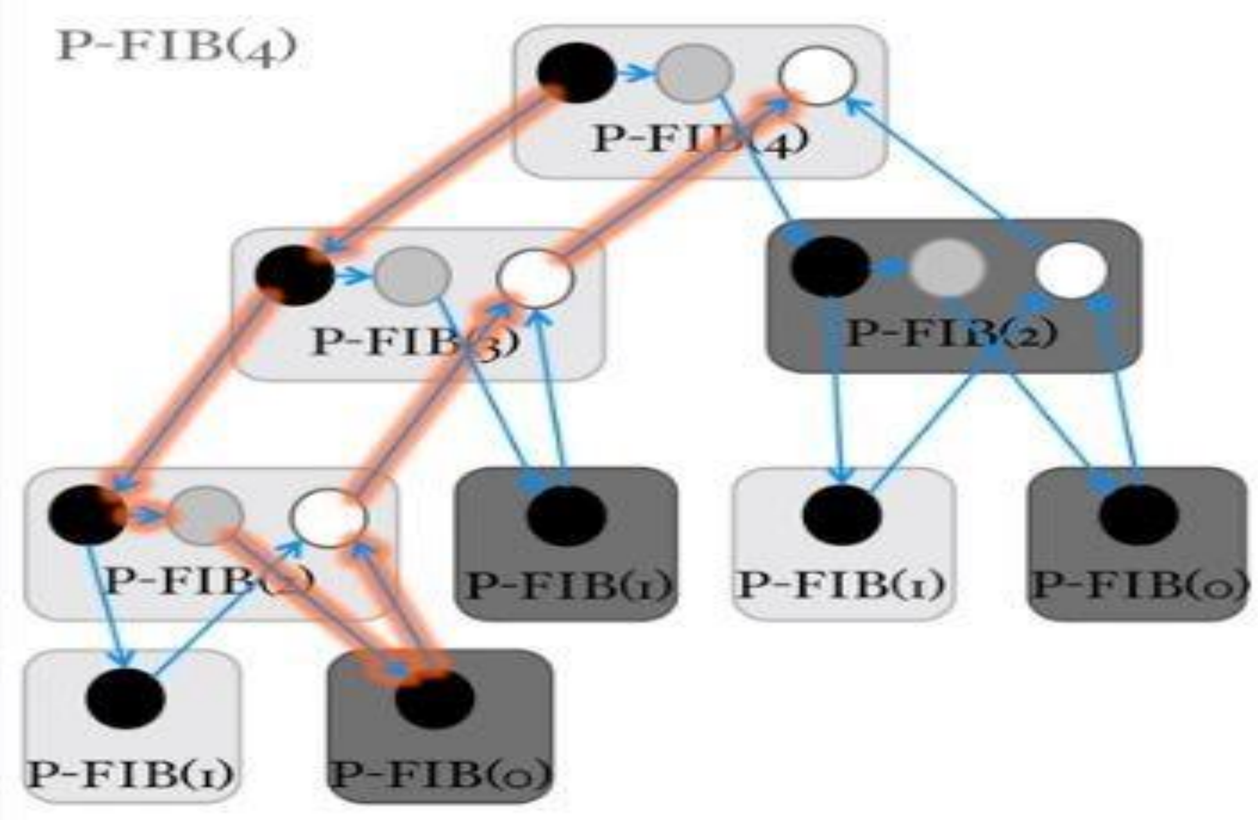


Dynamic Multithreading: Performance Measures

- 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

= 17 time units

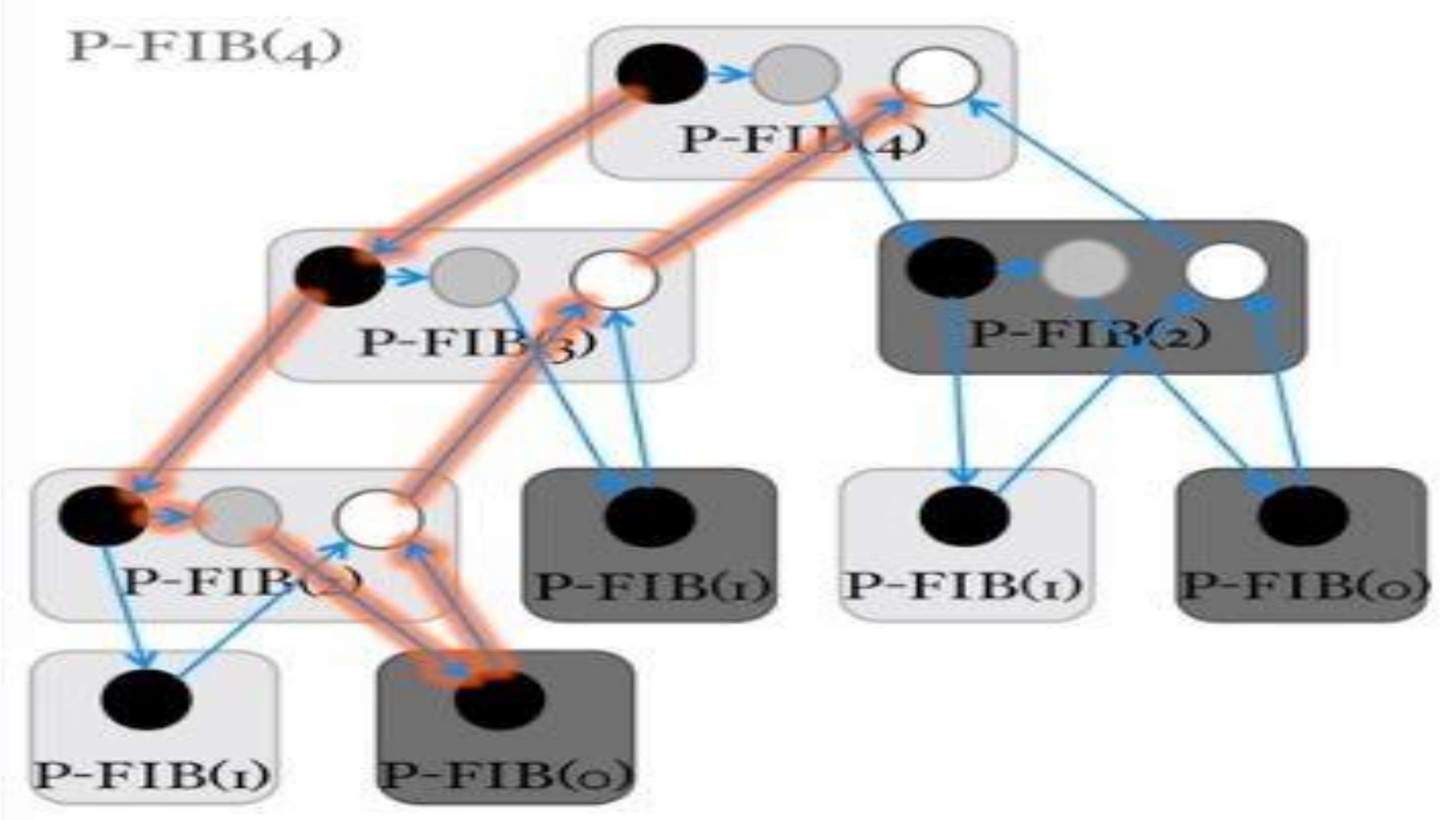


Dynamic Multithreading: Performance Measures

The **span** is the longest time to execute the strands along any path of the computational directed acyclic graph.

Span =the number of vertices on a longest or critical path

Span = 8 time units



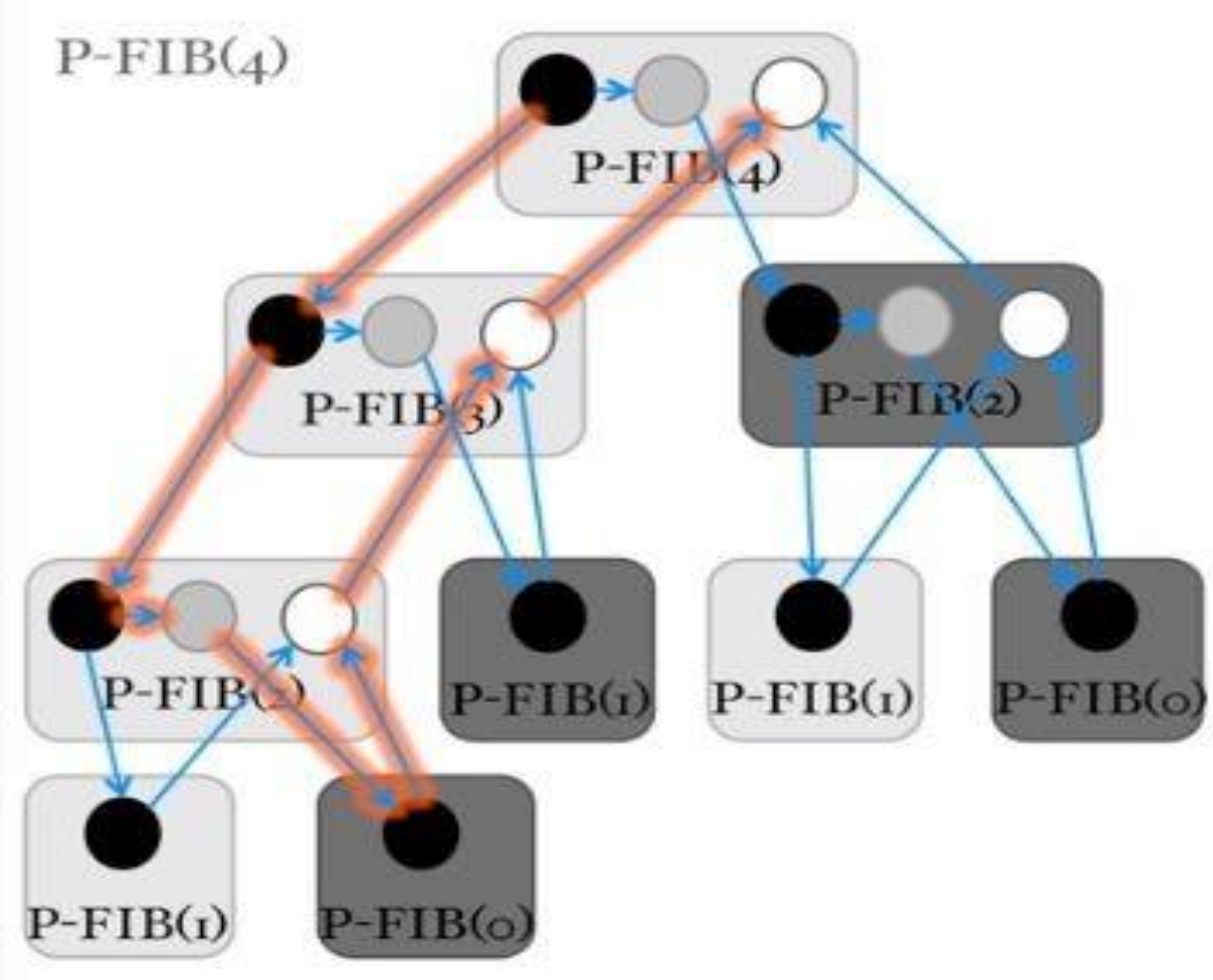
Dynamic Multithreading: Performance Measures

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



Dynamic Multithreading: Performance Measures

- 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**

- $\mathbf{T_1}$ running time on a single processor
- $\mathbf{T_P}$ running time on P processors
- $\mathbf{T_\infty}$ running time on unlimited processors, also called the span, ie if we run each strand on its own processor.

Dynamic Multithreading: Performance Measures

- An ideal parallel computer with P processors can do at most P units of work, and thus in T_p time, it can perform at most PT_p work.

  we have PT_p >= T1

- Dividing by P yields the ***work law:*** **$T_p >= T_1/P$**

Dynamic Multithreading: Performance Measures

- A computer with unlimited number of processors can emulate a P-processor machine by using just P of its processors. Therefore,

  $$T_p >= T_\infty$$

  which is called the **span law.**

  - T_P running time on P processors
  - T_∞ running time on unlimited processors

Dynamic Multithreading: Performance Measures

- The speed up of a computation on P processors is defined as T_1 / T_p

  i.e. how many times faster the computations on P processors than on 1 processor (How much faster it is).

- Thus, speedup on P processors can be at most P.

Dynamic Multithreading: Performance Measures

- The parallelism (max possible speed up) of a multithreaded computation is given by T_1 / T_∞

We can view the parallelism from three perspectives.

- □ As a ratio, the parallelism denotes the average amount of work that can be performed in parallel for each step along the critical path.
- □ As an upper bound, the parallelism gives the maximum possible speedup that can be achieved on any number of processors.
- □ Finally, and perhaps most important, the parallelism provides a limit on the possibility of attaining perfect linear speedup. Specifically, once the number of processors exceeds the parallelism, the computation cannot possibly achieve perfect linear speedup.

Dynamic Multithreading: Performance Measures

- Consider the computation P-FIB(4) and assume that each strand takes unit time.

Since the work is $T1 = 17$ and the span is $T_{\infty} = 8$, the parallelism is $T1/T_{\infty} =$

$17/8 = 2.125$.

- Consequently, achieving much more than double the speedup is impossible, no matter how many processors we employ to execute the computation.

Dynamic Multithreading: Performance Measures

- The performance depends not just on the work and span. Additionally, the strands must be scheduled efficiently onto the processors of the parallel machines.
- 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.

Dynamic Multithreading: Performance Measures

- 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.

Dynamic Multithreading: Performance Measures

- 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.

Dynamic Multithreading: Performance Measures