Lecture 15-16: Parallel Programming with Cilk

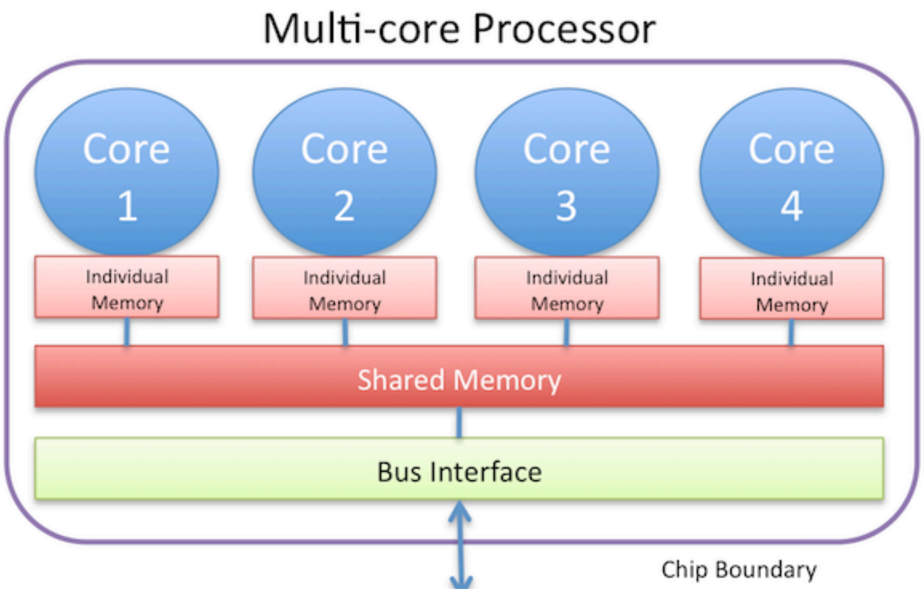
Concurrent and Multicore Programming

Department of Computer Science and Engineering
Yonghong Yan
yan@oakland.edu
www.secs.oakland.edu/~yan

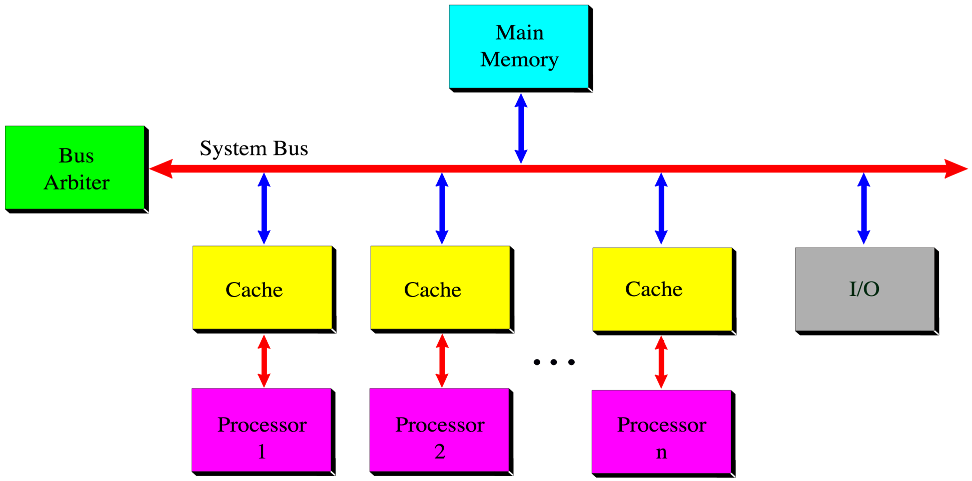
Topics (Part 1)

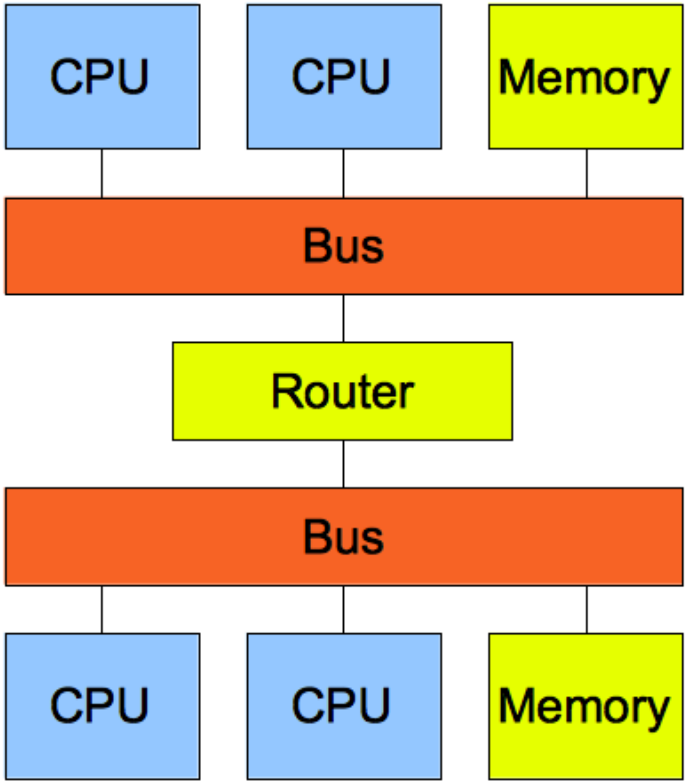
- Introduction
- Principles of parallel algorithm design (Chapter 3)
- Programming on shared memory system (Chapter 7)
  - **OpenMP**
  - **Cilk/Cilkplus**
  - **PThread, mutual exclusion, locks, synchronizations**
- Analysis of parallel program executions (Chapter 5)
  - **Performance Metrics for Parallel Systems**
    - **Execution Time, Overhead, Speedup, Efficiency, Cost**
  - **Scalability of Parallel Systems**
  - **Use of performance tools**

Shared Memory Systems: Multicore and Multisocket Systems



SMP - Symmetric Multiprocessor System





NUMA Architecture

Threading on Shared Memory Systems

- Employ parallelism to compute on shared data
  - boost performance on a fixed memory footprint (strong scaling)
- Useful for hiding latency
  - e.g. latency due to I/O, memory latency, communication latency
- Useful for scheduling and load balancing
  - especially for dynamic concurrency
- Relatively easy to program
  - easier than message-passing? you be the judge!

Programming Models on Shared Memory System

- Library-based models
  - All data are shared
  - Pthreads, Intel Threading Building Blocks, Java Concurrency, Boost, Microsoft .Net Task Parallel Library
- Directive-based models, e.g., OpenMP
  - shared and private data
  - pragma syntax simplifies thread creation and synchronization
- Programming languages
  - CilkPlus (Intel, GCC), and MIT Cilk
  - CUDA (NVIDIA)
  - OpenCL

Toward Standard Threading for C/C++

At last month's meeting of the C standard committee, WG14 decided to form a study group to **produce a proposal for language extensions for C to simplify parallel programming**. This proposal is expected to **combine the best ideas from Cilk and OpenMP, two of the most widely-used and well-established parallel language extensions for the C language family.**

As the chair of this new study group, named CPLEX (C Parallel Language Extensions), I am announcing its organizational meeting:

June 17, 2013 10:00 AM PDT, 2 hours

Interested parties should join the group's mailing list, to which further information will be sent:

http://www.open-std.org/mailman/listinfo/cplex

Questions can be sent to that list, and/or to me directly.

--

Clark Nelson　　Vice chair, PL22.16 (ANSI C++ standard committee)
Intel Corporation　　Chair, SG10 (WG21 study group for C++ feature-testing)
clark.nelson@intel.com　　Chair, CPLEX
(WG14 study group for C parallel language extensions)

OpenMP

- Identify static mapping and scheduling of tasks and cores
  - Before tasking
- No need to create thread manually
- Sequential code migration to parallel code by inserting directives
- Optimization for memory and synchronization are the key
  - Reduce memory contention and parallelism overhead

- **Users achieve both problem decomposition into tasks and mapping tasks to hardware**
  - **OpenMP worksharing 1:1 mapping**

Outline for Cilk/Cilkplus

☛ Introduction and Basic Cilk Programming

- Cilk Work-stealing Scheduler
- Implementation Strategies
- Performance Analysis
- Scheduling Performance Analysis
- More Examples

Cilk/Cilkplus Summary

- A simpler model for writing parallel programs
  - Focusing on problem decomposition
    - What computation can be performed in parallel
  - Runtime perform the mapping
- Extends C/C++ with two main keywords → ***tasking***
  - **spawn**: invoke a function (potentially) in parallel
  - **sync**: wait for a procedure's spawned functions to finish
- Faithful language extension
  - if Cilk/Cilkplus keywords are elided → C/C++ program semantics
- The idea has been adopted by OpenMP with task
  - omp task
  - omp taskwait

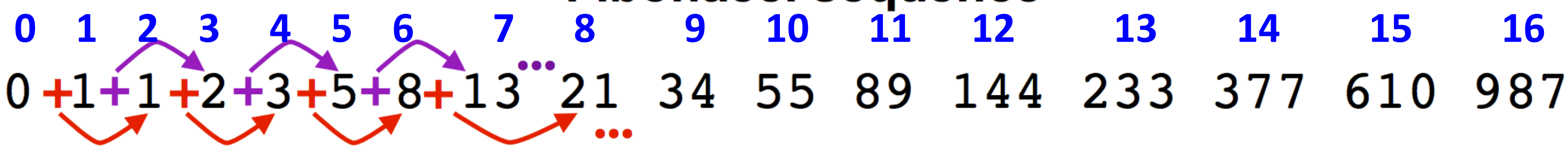
Availability

- Cilk and Cilkplus
  - Cilk is originally developed by MIT Charles E. Leiserson
    - http://supertech.csail.mit.edu/cilk/
  - Cilkplus is commercialized now from Intel: **cilk_spawn** and **cilk_sync**
    - Added **cilk_for**, parallel execution of a for loop
- Availability
  - MIT Cilk
  - Intel compilers, GCC 4.9

- lennon.secs.oakland.edu

Cilk Example

Fibonacci sequence



- **Computing Fibonacci recursively**

```
int fib(int n)  {
  if (n < 2) return n;
  else {
    int n1, n2;
    n1 = fib(n-1);
    n2 = fib(n-2);
   return (n1 + n2);
  }
}
```

https://en.wikipedia.org/wiki/Fibonacci_number

Fibonacci (MIT Cilk)

```
int fib (int n) {
if (n<2) return (n);
  else {
    int x,y;
    x = fib(n-1);
    y = fib(n-2);
    return (x+y);
  }
}
```

C elision

Cilk code

```
cilk int fib (int n) {
  if (n<2) return (n);
  else {
    int x,y;
    x = spawn fib(n-1);
    y = spawn fib(n-2);
    sync;
    return (x+y);
  }
}
```

A Cilk program's ***serial elision*** is always a legal implementation of Cilk semantics. Cilk provides *no* new data types.

Basic Cilk Keywords

```
cilk int fib (int n) {
  if (n<2) return (n);
  else {
    int x,y;
    x = spawn fib(n-1);
    y = spawn fib(n-2);
    sync;
    return (x+y);
  }
}
```

Identifies a function as a *Cilk procedure*, capable of being spawned in parallel.

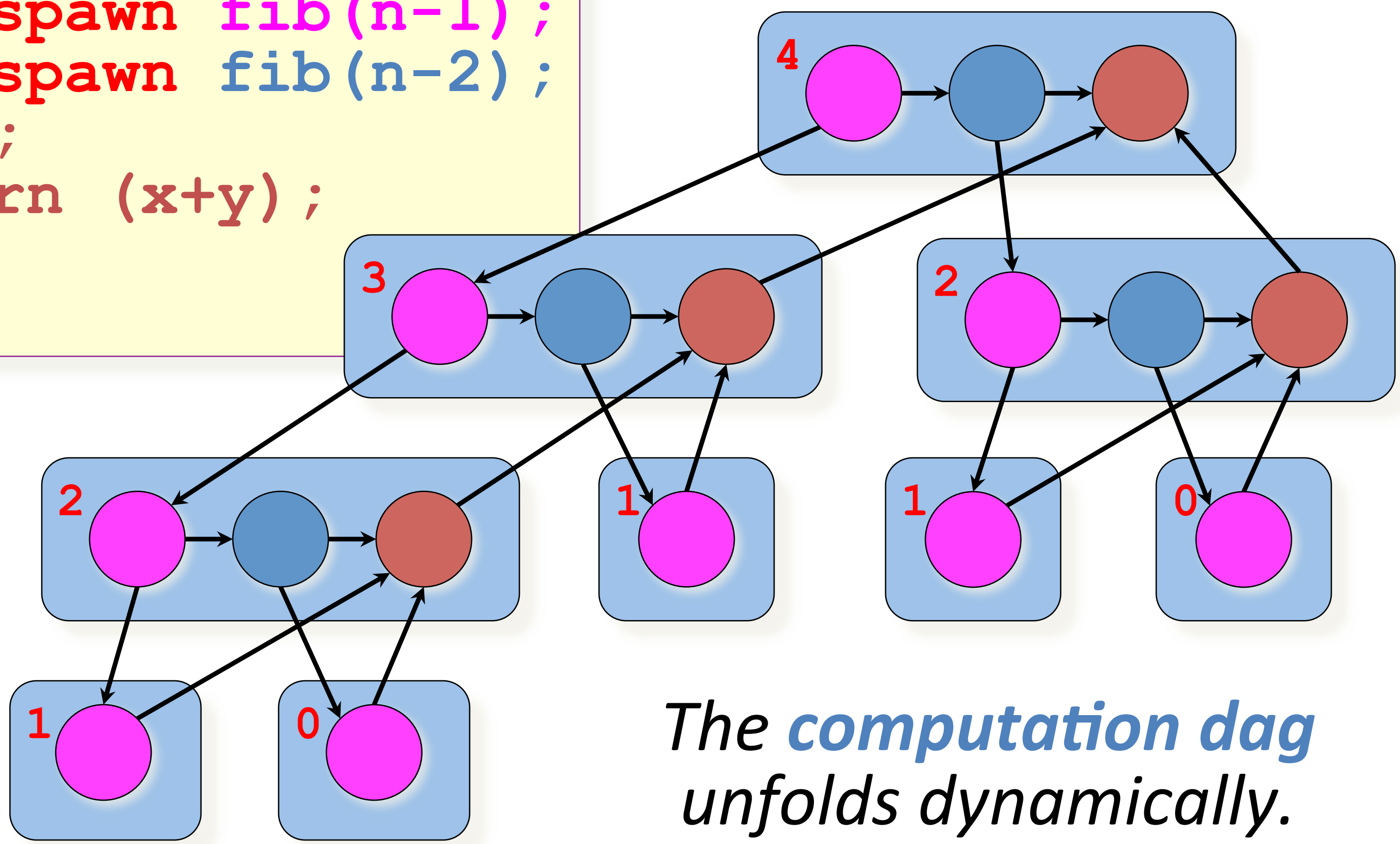
The named ***child*** Cilk procedure can execute in parallel with the ***parent*** caller.

Control cannot pass this point until all spawned children have returned.

Dynamic Multithreading

```
cilk int fib (int n) {
  if (n<2) return (n);
  else {
    int x,y;
    x = spawn fib(n-1);
    y = spawn fib(n-2);
    sync;
    return (x+y);
  }
}
```

Example: `fib(4)`

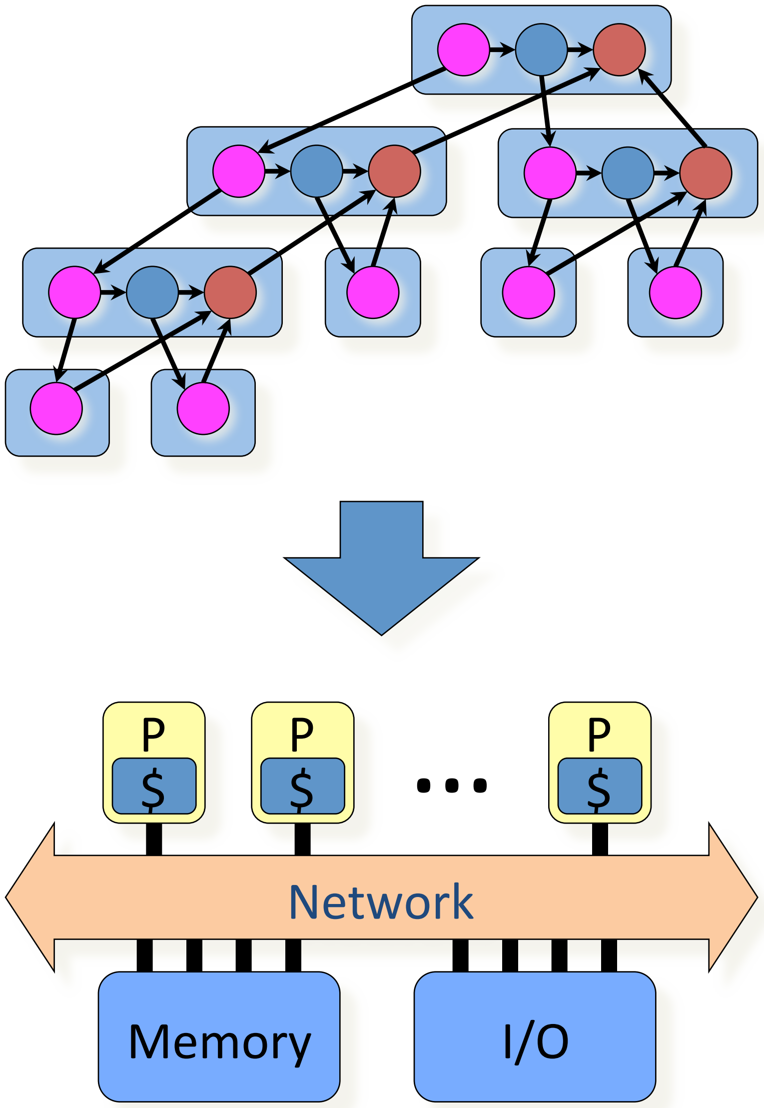


*The **computation dag** unfolds dynamically.*

Mapping Tasks to Hardware

- Cilk allows the programmer to express ***potential*** parallelism in an application.
  - Many tasks

- The Cilk ***scheduler*** maps Cilk tasks onto processors dynamically at runtime
  - A thread in this context is a PE



Outline for Cilk/Cilkplus

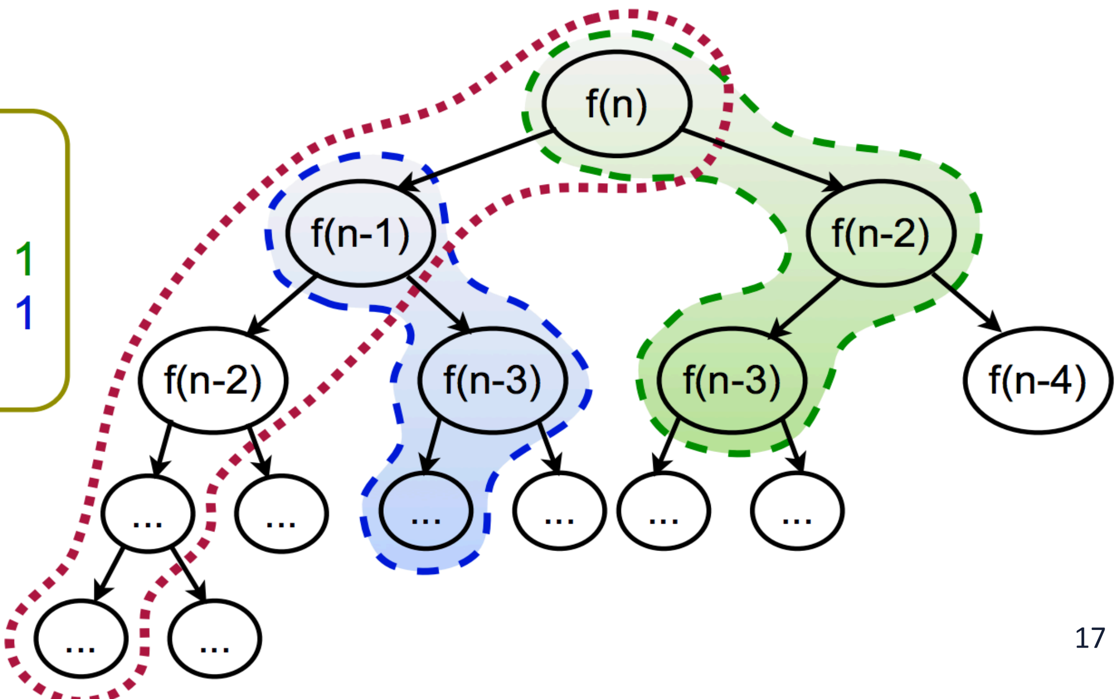
- Introduction and Basic Cilk Programming
- ☛ Cilk Work-stealing Scheduler
- Implementation Strategies
- Performance Analysis
- Scheduling Performance Analysis
- More Examples

Scheduling Tasks in Cilk

- **Lazy parallelism**
  - Put off work for parallel execution until necessary
    - E.g. no need for parallel execution when no enough PEs
- **Work-stealing**
  - Multiple PEs share work (tasks)
    - A PE looks for work in other PEs when it becomes idle
  - Any PE can create work (tasks) via spawn

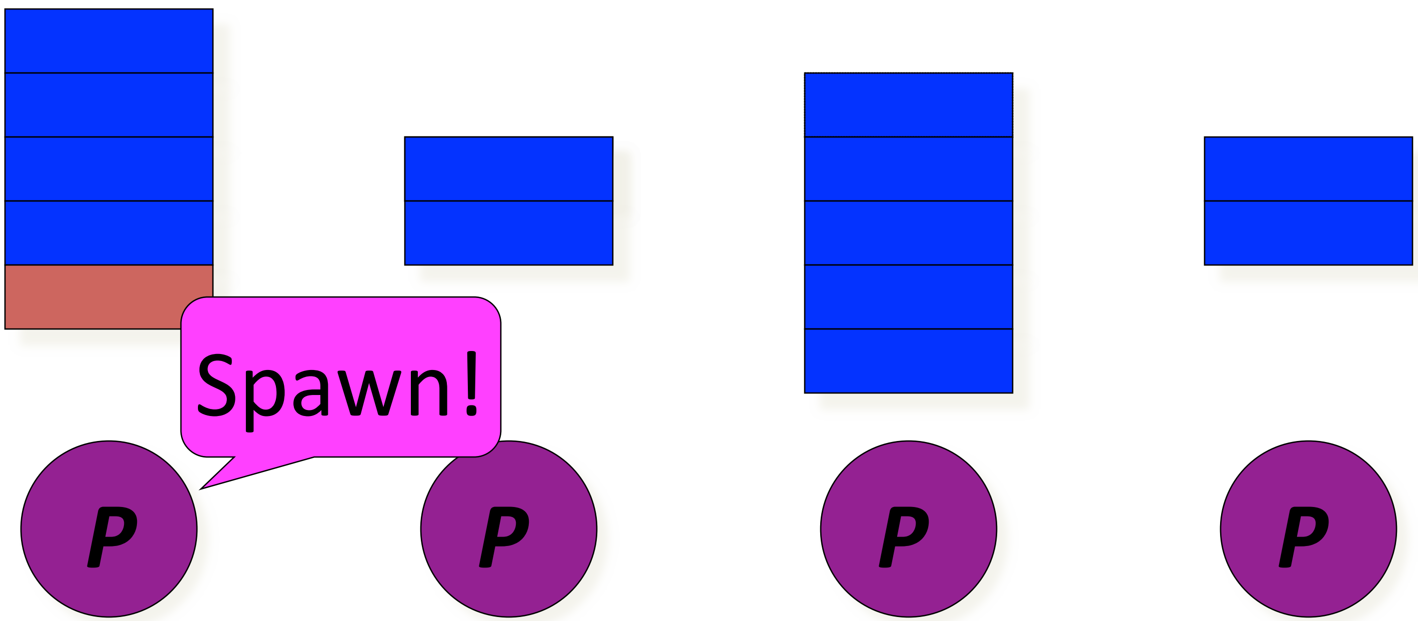
Possible Execution:

thread 1 begins
thread 2 steals from 1
thread 3 steals from 1
etc...



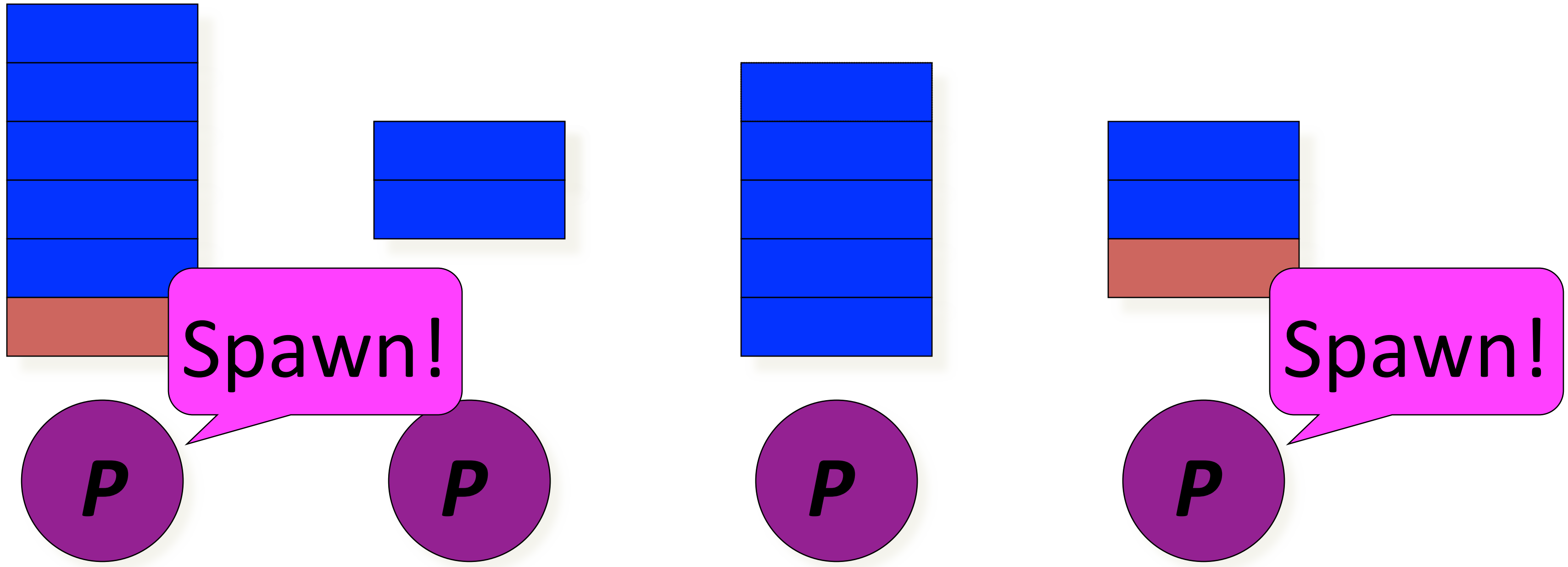
Cilk's Work-Stealing Scheduler

- Each PE maintains a ***work deque*** of ready tasks, and it manipulates the bottom of the deque like a stack.
  - Push and pop



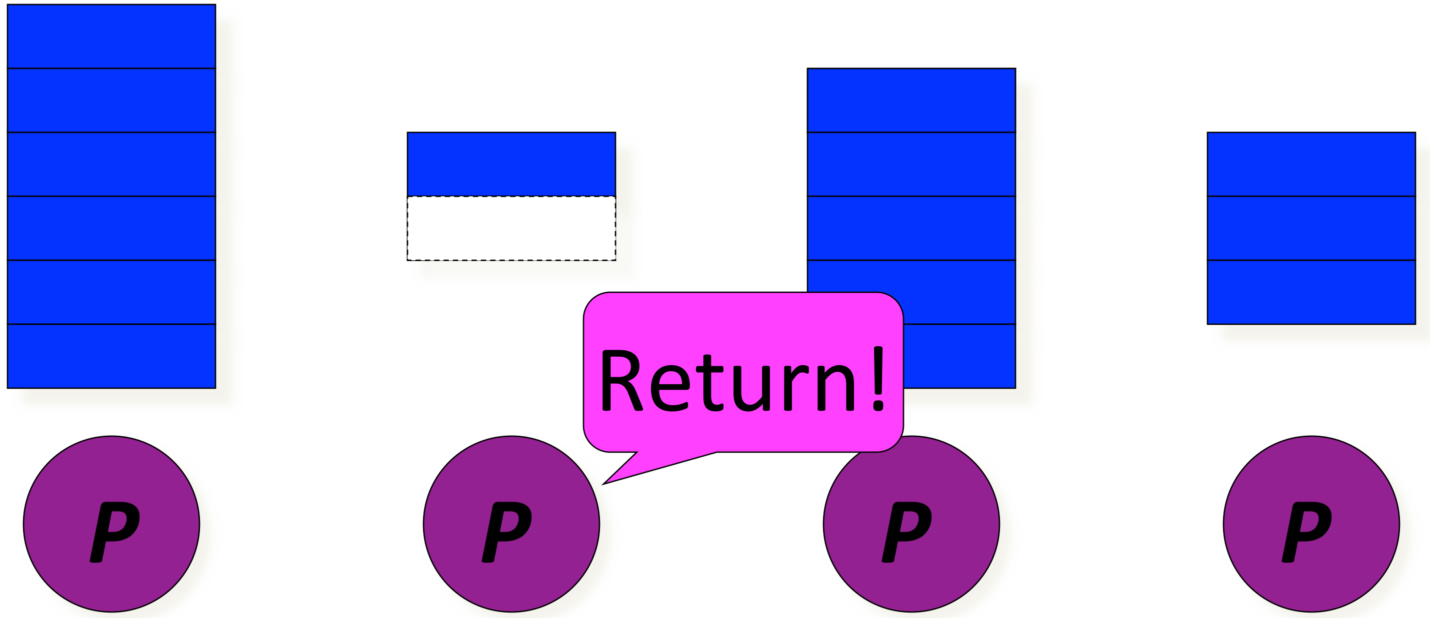
Cilk’s Work-Stealing Scheduler

- Each PE maintains a ***work deque*** of ready tasks, and it manipulates the bottom of the deque like a stack.
  - Push and pop



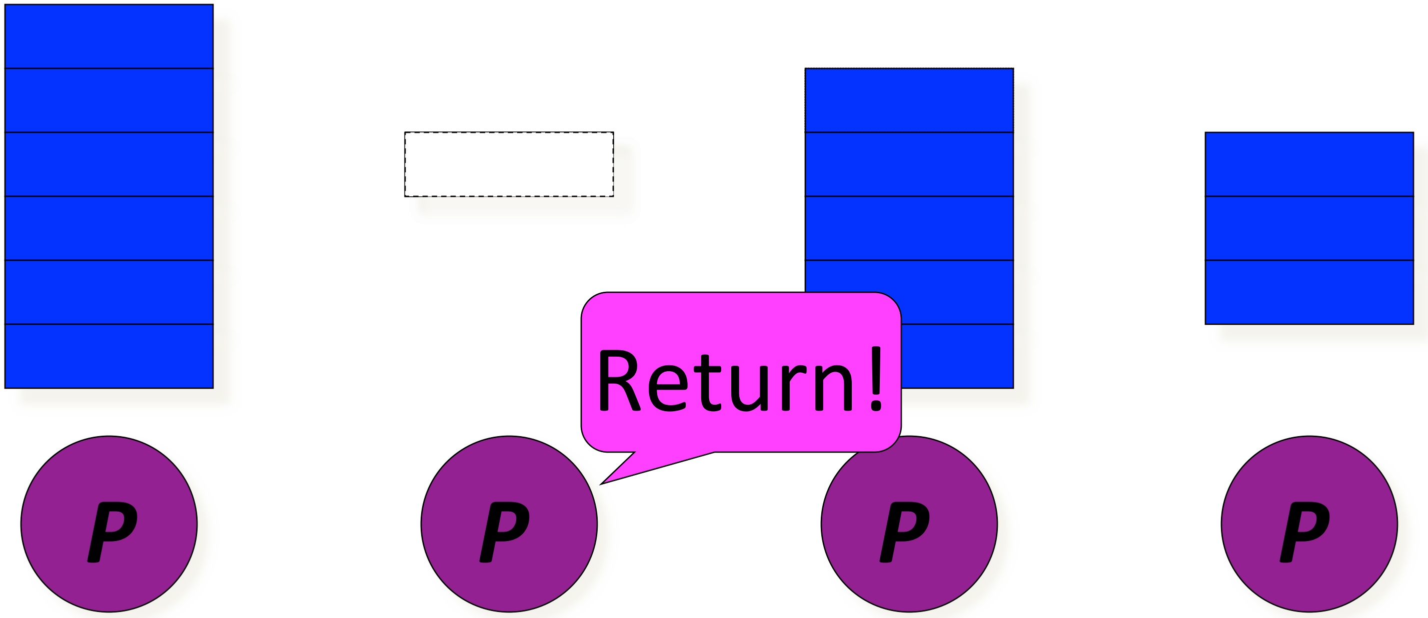
Cilk's Work-Stealing Scheduler

- Each PE maintains a ***work deque*** of ready tasks, and it manipulates the bottom of the deque like a stack.
  - Push and pop



Cilk's Work-Stealing Scheduler

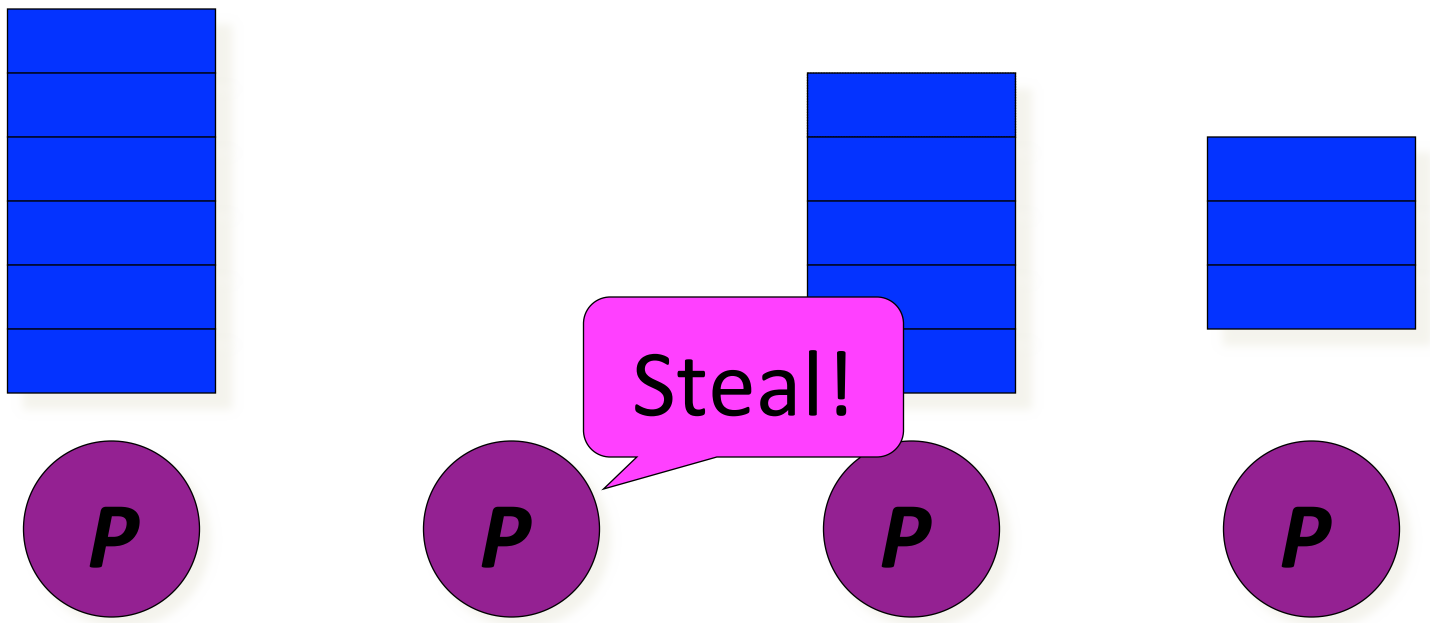
- Each PE maintains a ***work deque*** of ready tasks, and it manipulates the bottom of the deque like a stack.
  - Push and pop



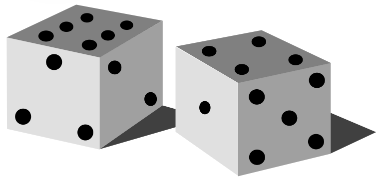
Cilk's Work-Stealing Scheduler

- Each PE maintains a ***work deque*** of ready tasks, and it manipulates the bottom of the deque like a stack.
  - Push and pop

https://en.wikipedia.org/wiki/Double-ended_queue

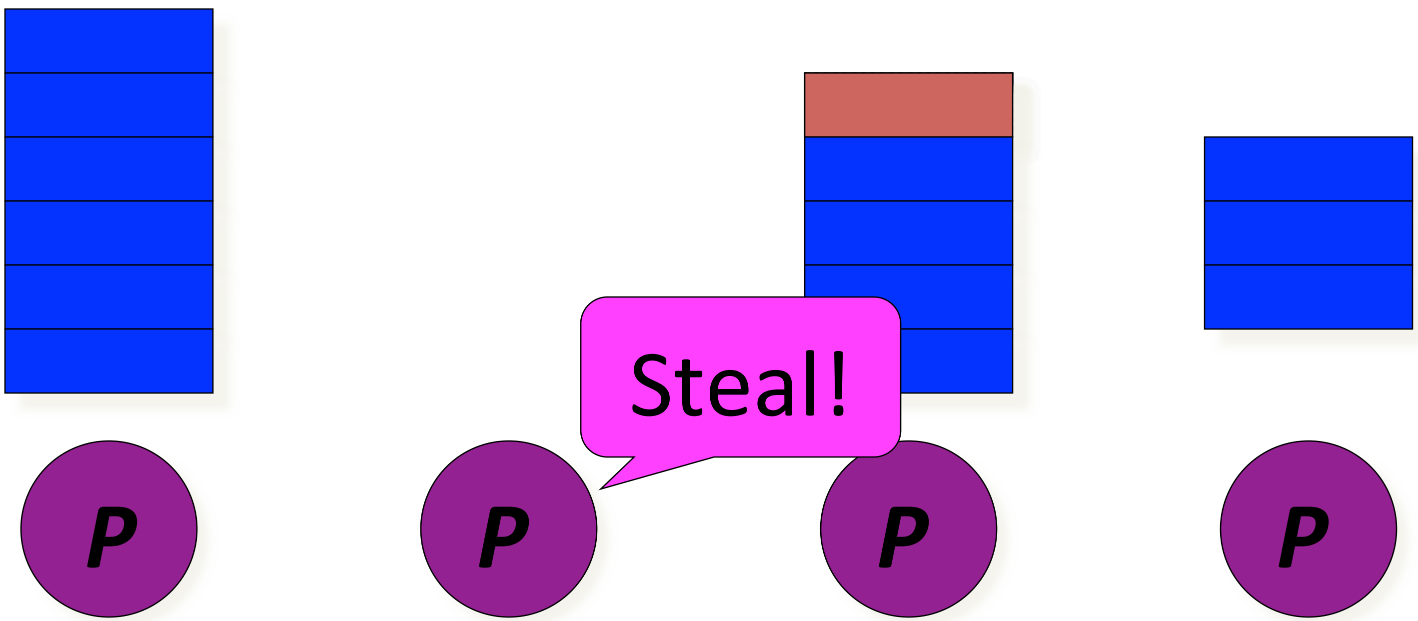


When a processor runs out of work, it ***steals*** a task from the top of a ***random*** victim's deque.

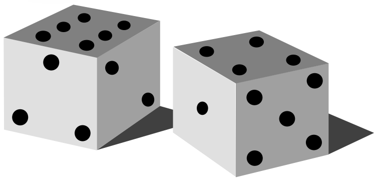


Cilk's Work-Stealing Scheduler

- Each PE maintains a ***work deque*** of ready tasks, and it manipulates the bottom of the deque like a stack.
  - Push and pop

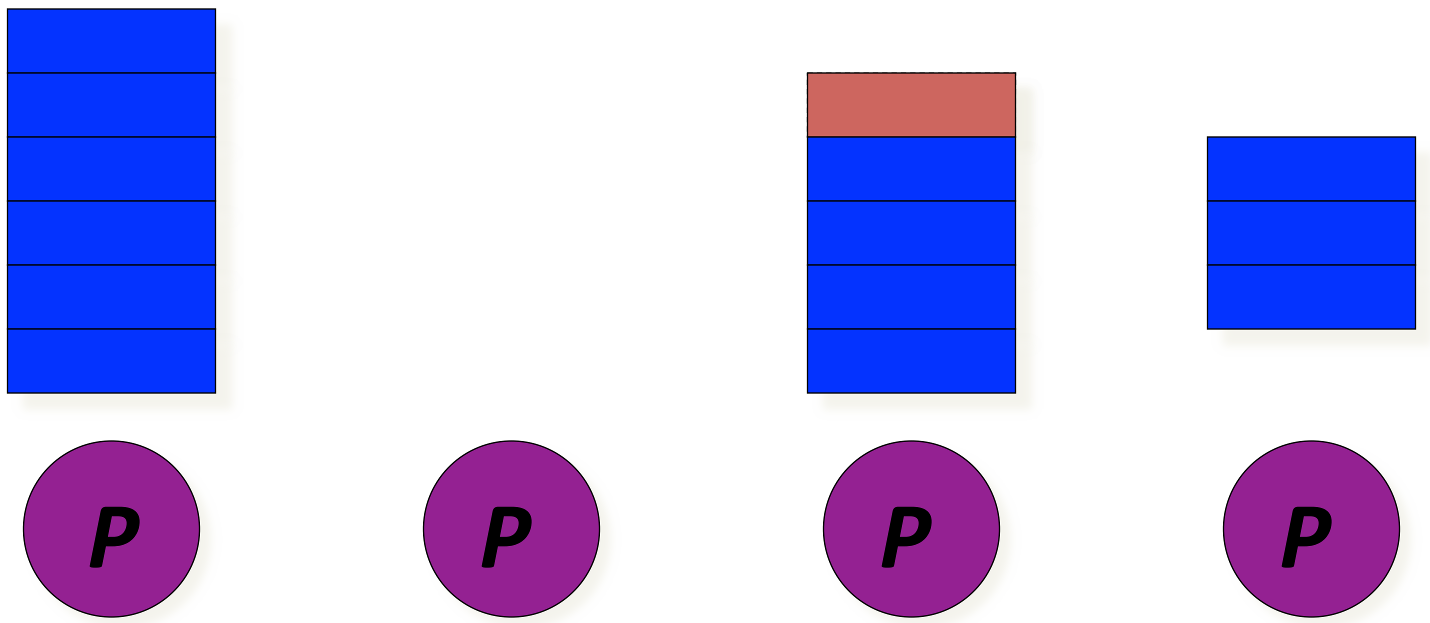


When a processor runs out of work, it ***steals*** a task from the top of a ***random*** victim's deque.

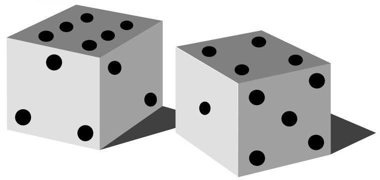


Cilk's Work-Stealing Scheduler

- Each PE maintains a ***work deque*** of ready tasks, and it manipulates the bottom of the deque like a stack.
  - Push and pop

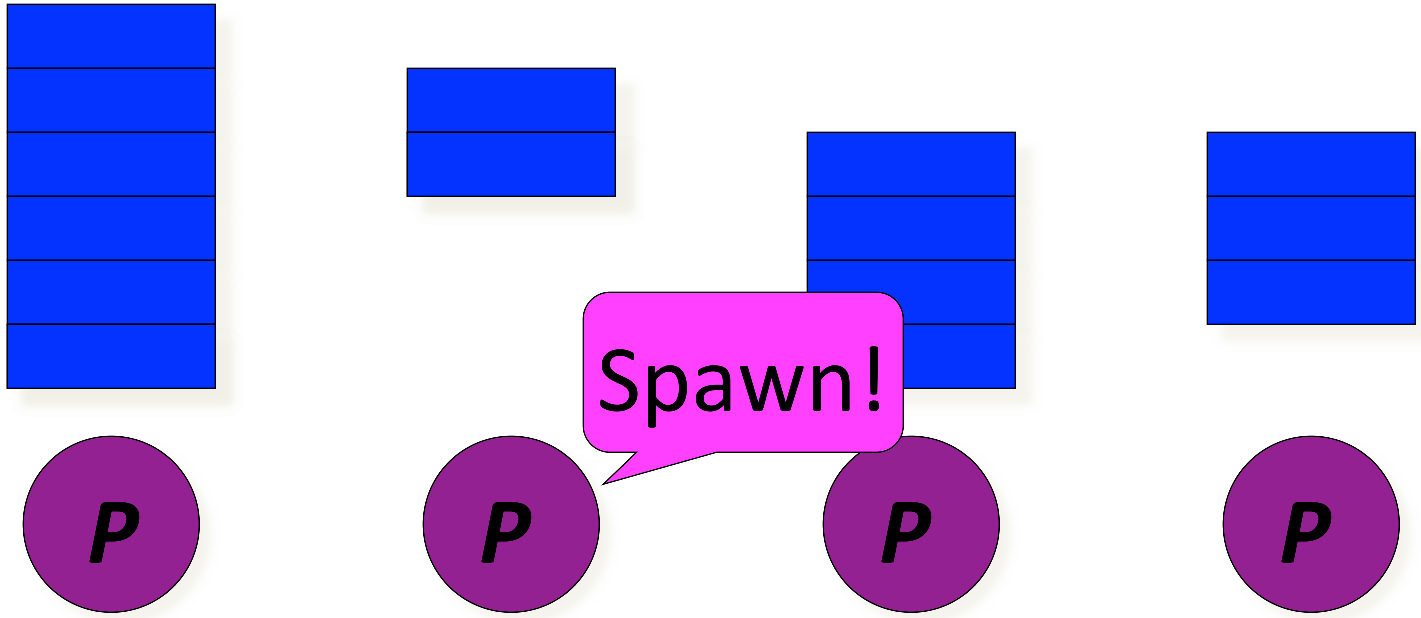


When a processor runs out of work, it ***steals*** a task from the top of a ***random*** victim's deque.

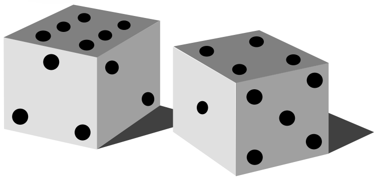


Cilk's Work-Stealing Scheduler

- Each PE maintains a ***work deque*** of ready tasks, and it manipulates the bottom of the deque like a stack.
  - Push and pop



When a processor runs out of work, it ***steals*** a task from the top of a ***random*** victim's deque.



Dynamic Multithreading

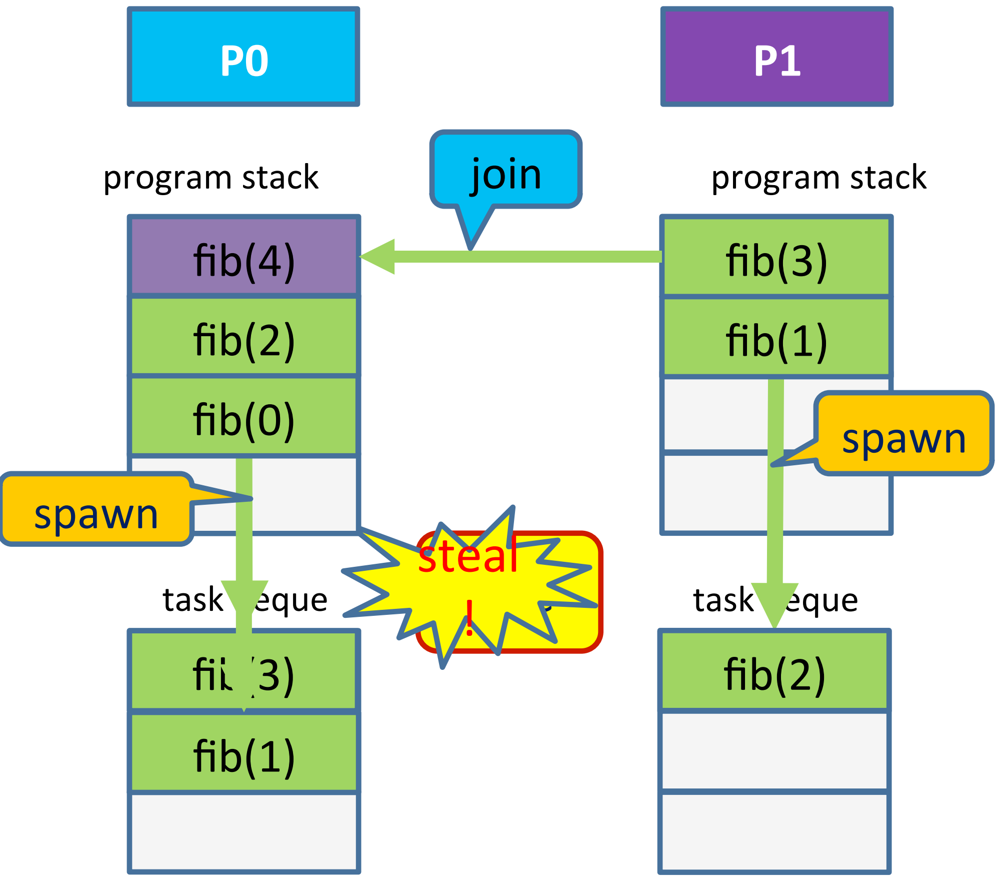
```
int fib (int n) {
  if (n<2) return (n);
  else {
    int x,y;

    x = _Cilk_spawn fib(n-1);

    y = _Cilk_spawn fib(n-2);

    _Cilk_sync;

    return (x+y);
  }
}
```

Dynamic Multithreading

```
int fib (int n) {
  if (n<2) return (n);
  else {
    int x,y;

    x = cilk_spawn fib(n-1);

    y = cilk_spawn fib(n-2);

    cilk_sync;

    return (x+y);
  }
}
```

Continuation

Workstealing State on both Program Stack an Dequeu



- At a fork point, add tasks to the tail of the current worker's deque and execute the other task.
- If idle, steal work from the ***head*** of a ***random*** other worker's deque.
- When at an incomplete join point, pop work off the ***tail*** of the worker's ***own*** deque (reverse fork).
- If worker's own deque is empty, either stall at join, or do a random steal.

Child-Stealing

- At a spawn, the child task in pushed onto the worker's deque.
  - A task data structure is allocated on the heap
  - Everything needed to run the child is stored in the task data structure
  - A pointer to the task data structure is pushed onto the deque
- The worker then executes the fork continuation immediately.
- An idle worker can steal the child task.
- If the child task is not stolen, it is run by the original worker when it reaches the join point.
- *Typically*, the scheduler stalls at the join point if there are stolen children that have not completed.

Continuation Stealing

- At a spawn, the *continuation* in pushed onto the worker's deque.
  - Registers are saved on the *stack.*
  - A pointer to the current stack frame is pushed onto the deque
- The worker then executes the child immediately, as if it were a normal call.
- An idle worker can steal the continuation task.
- Upon completing the child, if the continuation (parent) has not been stolen, the original worker continues as if returning from a normal function call.
- The *join continuation* is run by whichever worker completes its task *last*.
  - Typically, no worker stalls at the join point.
  - The worker running after the join might be different than the one entering it.

Advantages of Child stealing over continuation Stealing

Both are types of work stealing. Continuation stealing has a number of **practical** advantages, however:

- Child stealing libraries can be implemented without special compiler support; continuation stealing typically requires compiler support.
- At each fork and spawn point, a continuation stealing implementation might switch to a different worker thread, confusing code that depends on thread-local storage.

Advantages of continuation stealing over Child Stealing

Conversely continuation stealing has many **theoretical** advantages of continuation stealing:

- Queue size bounded by recursion depth & stack space bound to P times serial stack usage vs. unbounded queue size for child stealing.
- On a single worker, continuation stealing produces identical execution to serial code; child stealing produces a scrambled execution order.
- Naturally lends itself to non-stalling join points making it closer to an ideal greedy scheduler.
- Certain features are easier to implement efficiently on top of a continuation-stealing scheduler, for example: associative reductions.

Advantages of continuation stealing over Child Stealing

Conversely continuation stealing has many **theoretical** advantages of child stealing:

- Queue size bounded by … depth & stack space bound to P times serial stack … nded queue size for child stealing.
- On a single wo… produces identical execution to … oduces a scrambled execution or…
- Naturally len… oints making it closer to an ideal gr…
- Certain feature… t efficiently on top of a continuation-ste… xample: associative reductions.

Outline for Cilk/Cilkplus

- Introduction and Basic Cilk Programming
- Cilk Work-stealing Scheduler
- ☛ Implementation Strategies
- Performance Analysis
- Scheduling Performance Analysis
- More Examples

Compiling spawn — Fast Clone

Cilk source

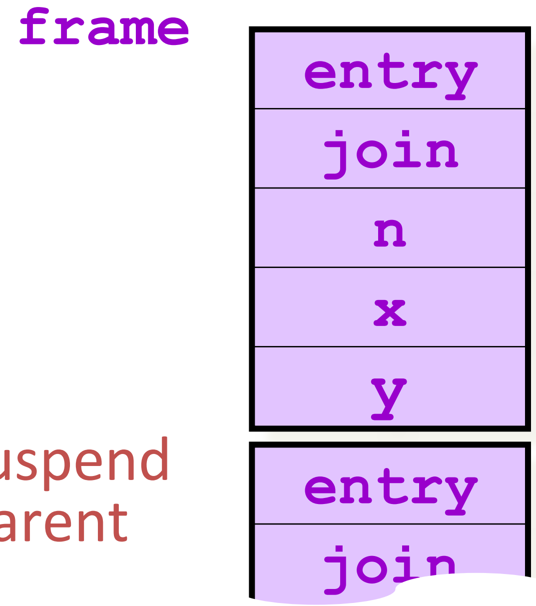
```
x = spawn fib(n-1);
```

cilk2c

C post-source

```
frame->entry = 1;
frame->n = n;
push(frame);

x = fib(n-1);

if (pop()==FAILURE) {
   frame->x = x;
   frame->join--;
   h clean up &
    return to scheduler i
}
```

- suspend parent (first three lines)
- run child (`x = fib(n-1);`)
- resume parent remotely (the `if` block)



Cilk deque

Compiling `sync` — Fast Clone

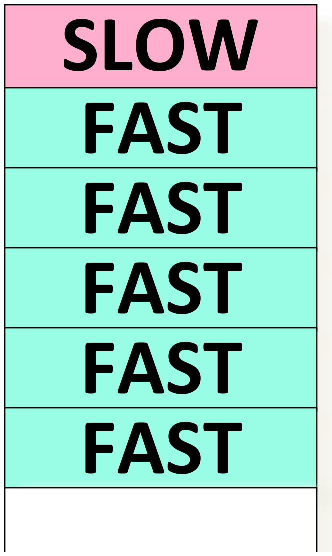
Cilk source

```
sync;
```

cilk2c

C post-source

```
;
```



No synchronization overhead in the fast clone!

Compiling the Slow Clone

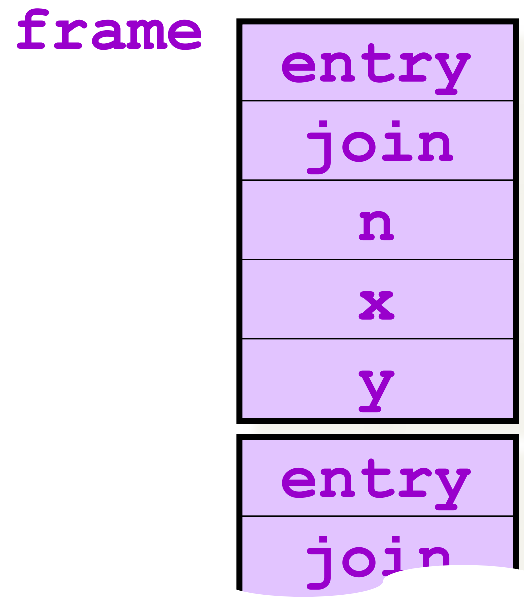
```
void fib_slow(fib_frame *frame) {
  int n,x,y;
  switch (frame->entry) {
    case 1: goto L1;
    case 2: goto L2;
    case 3: goto L3;
    }
          ⋮
  frame->entry = 1;
  frame->n = n;
  push(frame);
  x = fib(n-1);
  if (pop()==FAILURE) {
    frame->x = x;
    frame->join--;
    h clean up &
     return to scheduler i
  }

  if (0) {
    L1:;
    n = frame->n;
  }
          ⋮
}
```

restore program counter

same as fast clone

restore local variables if resuming

continue



Cilk deque

Project Accounts

- On orion.ec.oakland.edu
  - Need VPN to access from home

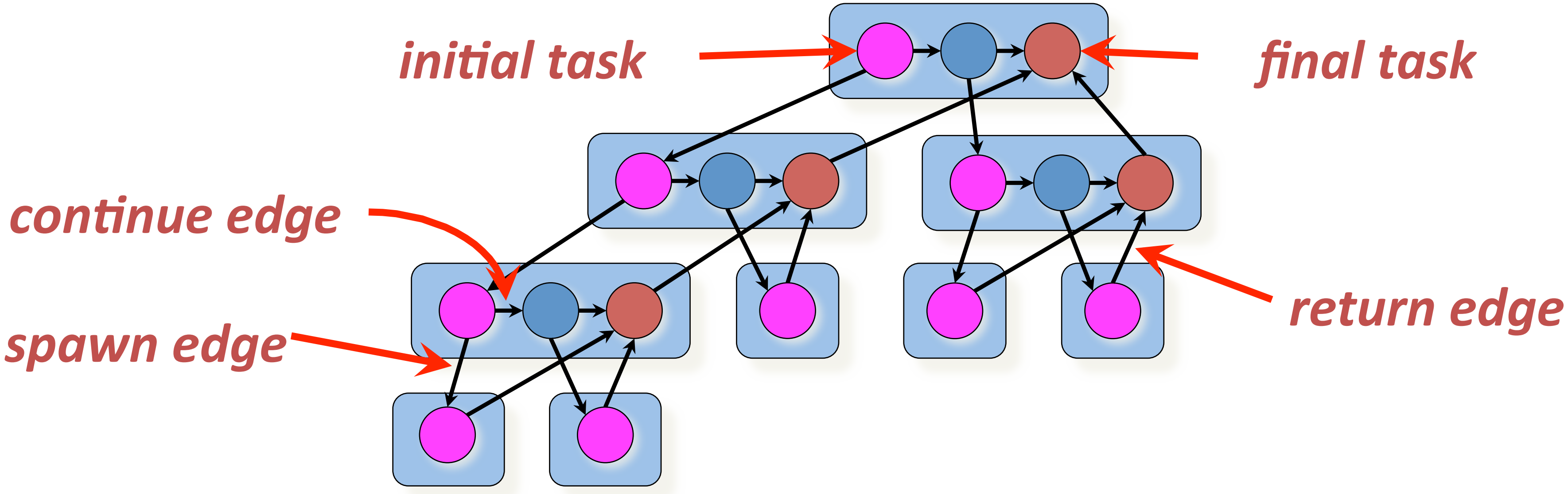
- Accont is the same as your netid
  - Password: <first four letters of your netid>1234
  - Change it the first time you login

- Follow development setup steps to clone the OpenMP runtime repo and examples repo

Outline for Cilk/Cilkplus

- Introduction and Basic Cilk Programming
- Cilk Work-stealing Scheduler
- Implementation Strategies
- ☛ Performance Analysis
- Scheduling Performance Analysis
- More Examples

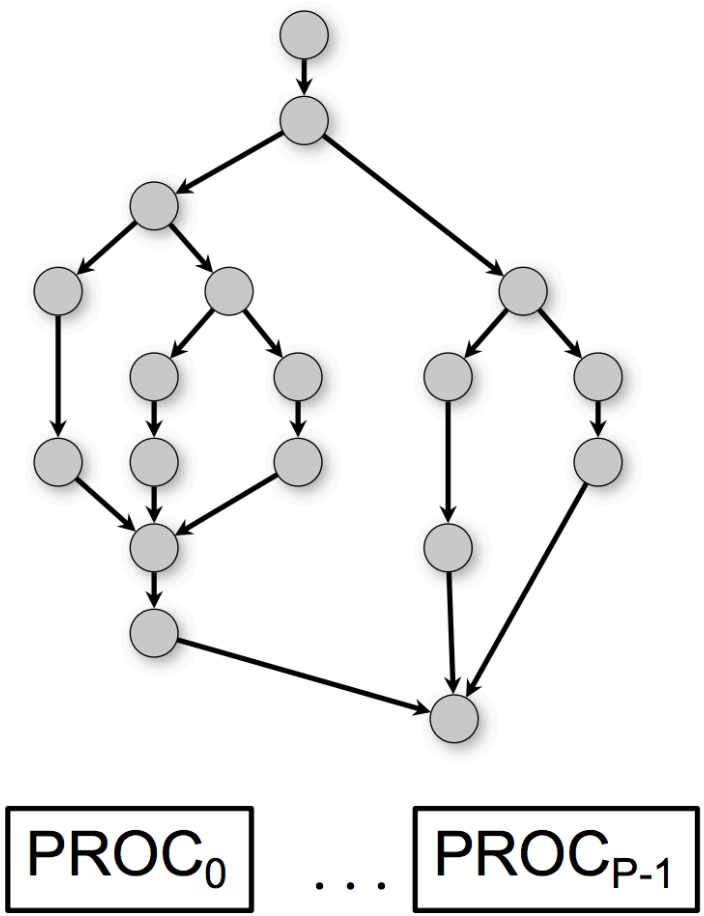
Multithreaded Computation



- The dag $G = (V, E)$ represents a parallel instruction stream.
- Each vertex v of V represents a ***(Cilk) task***: a maximal sequence of instructions not containing parallel control (**spawn**, **sync**, **return**).
- Every edge e of E is either a ***spawn*** edge, a ***return*** edge, or a ***continue*** edge.

Algorithmic Complexity Analysis

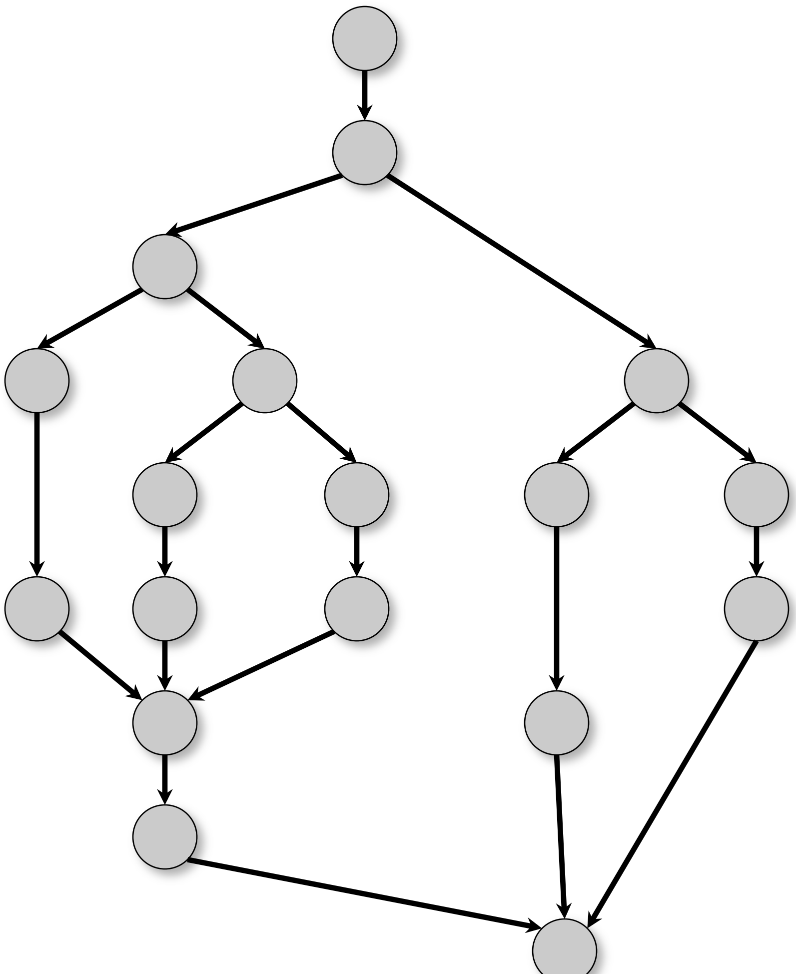
T_P = execution time on P processors



- Computation graph abstraction:
  - node = arbitrary sequential computation
  - edge = dependence (successor node can only execute after predecessor node has completed)
  - Directed Acyclic Graph (DAG)
- Processor abstraction:
  - P identical processors
  - each processor executes one node at a time

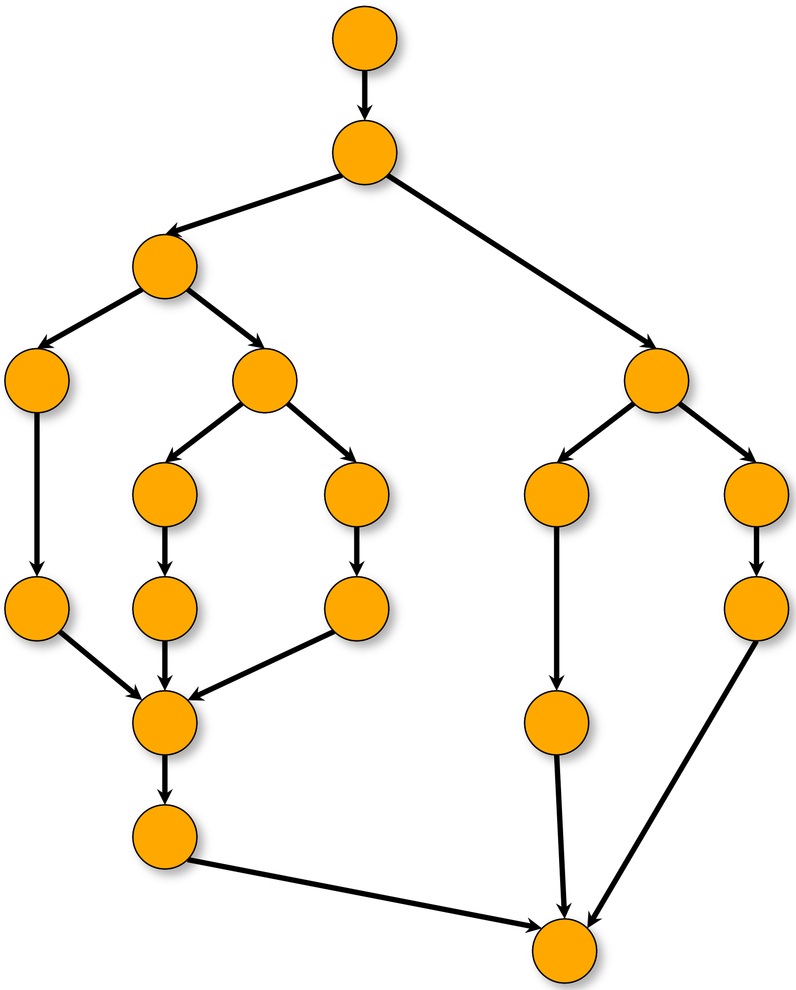
Algorithmic Complexity Analysis

T_P = execution time on P processors



Algorithmic Complexity Analysis

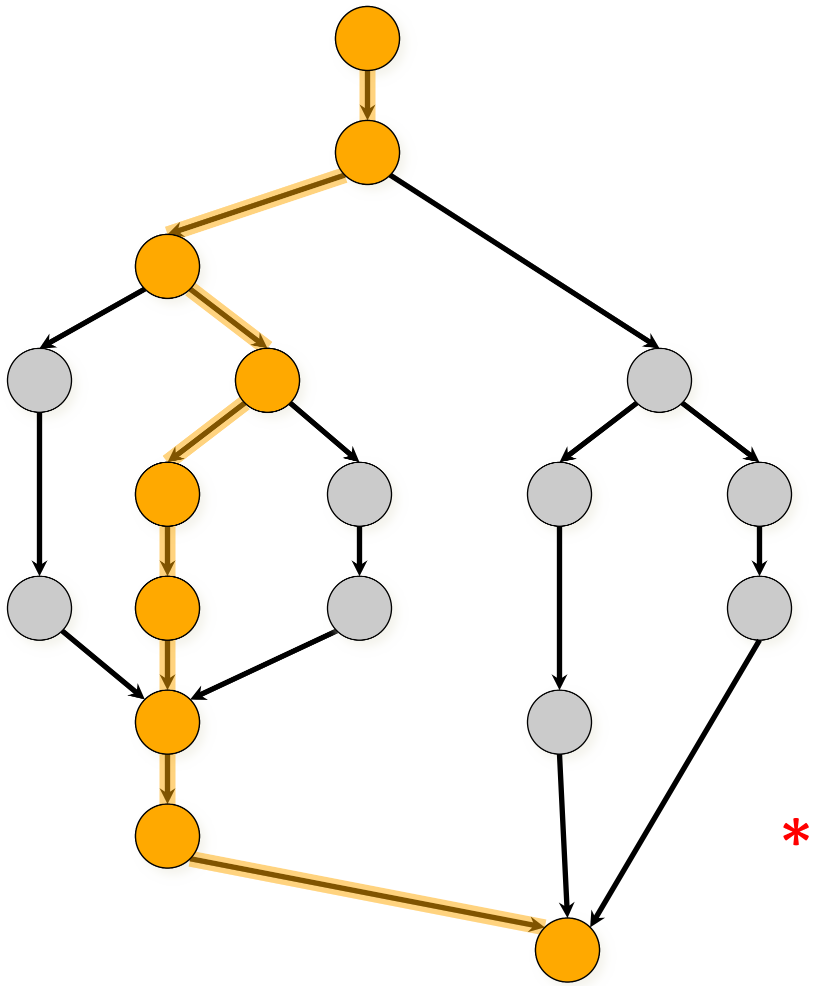
T_P = execution time on P processors



T_1 = ***work***

Algorithmic Complexity Analysis

T_P = execution time on P processors



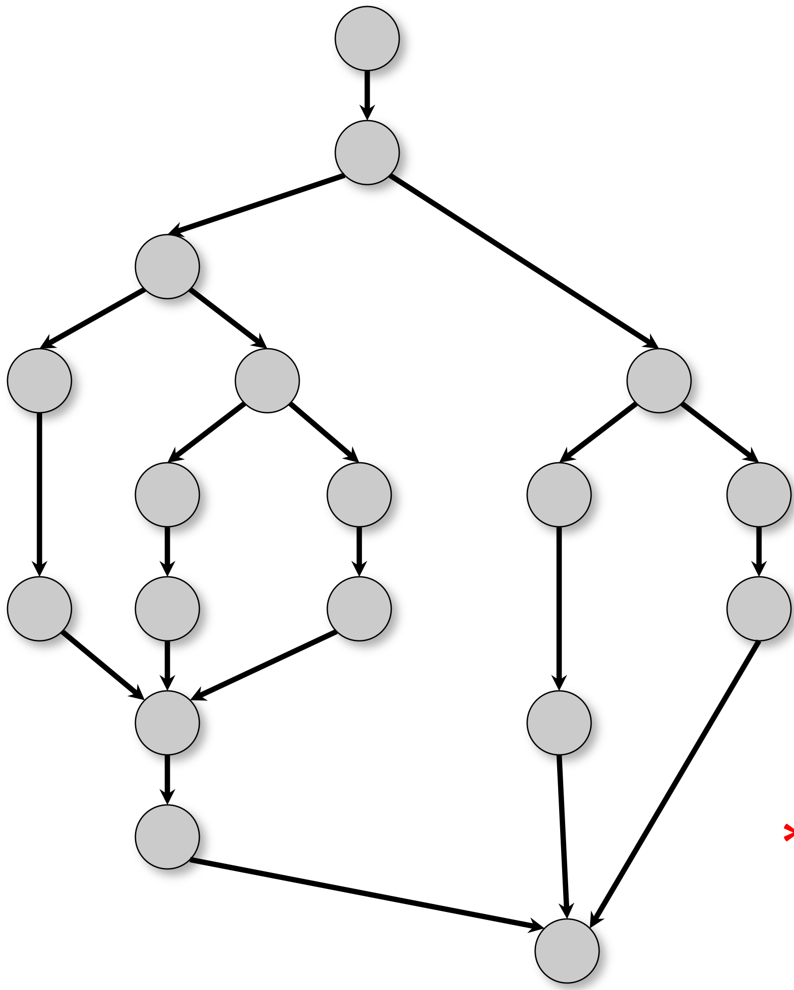
T_1 = ***work***

T_∞ = ***span****

* Also called ***critical-path length*** or ***computational depth***.

Algorithmic Complexity Analysis

T_P = execution time on P processors



T_1 = ***work***

T_∞ = ***span****

LOWER BOUNDS

- $T_P >= T_1/P$
- $T_P >= T_\infty$

*Also called ***critical-path length*** or ***computational depth***.

Speedup

Definition: T_1/T_P = ***speedup*** on P processors.

If $T_1/T_P = \Theta(P)$, we have ***linear speedup***;
$= P$, we have ***perfect linear speedup***;
$> P$, we have ***superlinear speedup***, which is not possible in our model, because of the lower bound $T_P >= T_1/P$.

Parallelism and Parallel Slackness

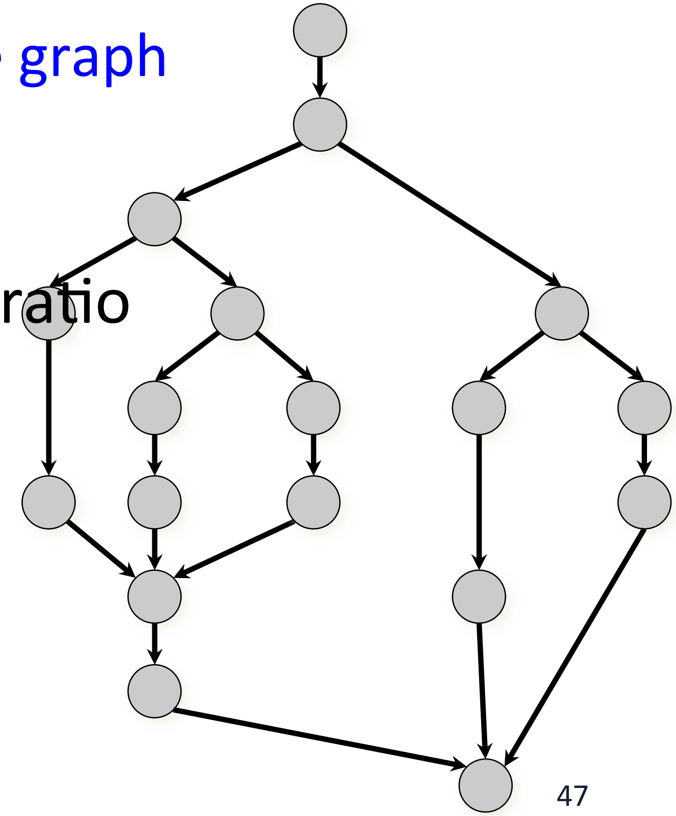
- We have the lower bound $T_P >= T_\infty$ and $T_P >= T_1/P$
- The maximum possible speedup given T_∞ and T_1, i.e. the ***parallelism***
  - Independent of P, only depend on the graph

$$P = T_1/T_\infty$$

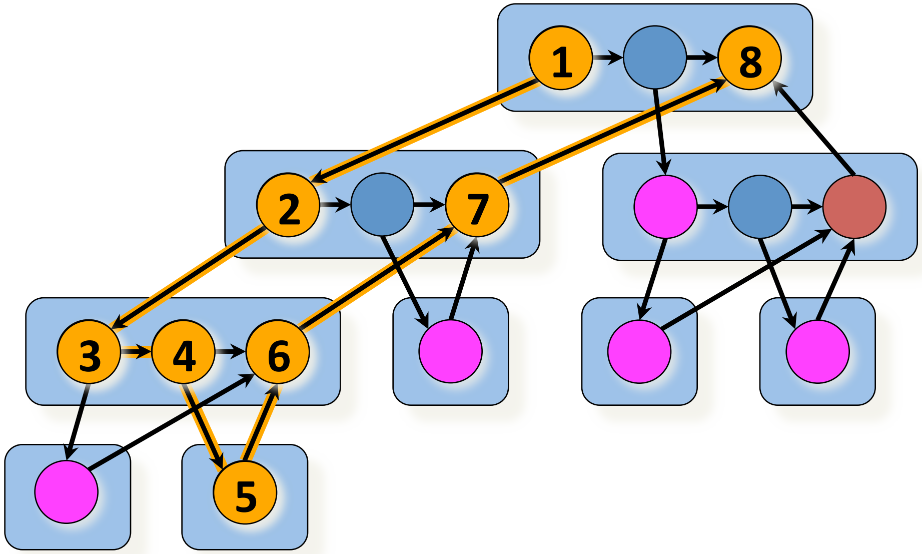
- ***Parallel slackness (Efficiency)*** as the ratio

$$(T_1/T_\infty)/P$$

  - The larger the efficiency, the less the impact of T_∞ on performance



Example: fib(4)



Assume for simplicity that each Cilk task in `fib()` *takes unit time to execute.*

Work: $T_1 = 17$

Span: $T_\infty = 8$

Parallelism: $T_0/T_\infty = 2.125$

Using many more than 2 processors makes little sense.

Parallelizing Vector Addition

C

```
void vadd (real *A, real *B, int n){
  int i; for (i=0; i<n; i++) A[i]+=B[i];
}
```

Parallelizing Vector Addition

C

```
void vadd (real *A, real *B, int n){
  int i; for (i=0; i<n; i++) A[i]+=B[i];
}
```

C

```
void vadd (real *A, real *B, int n){
  if (n<=BASE) {
    int i; for (i=0; i<n; i++) A[i]+=B[i];
  } else {
    vadd (A, B, n/2);
    vadd (A+n/2, B+n/2, n-n/2);
  }
}
```

Parallelization strategy:

1. Convert loops to recursion.

Parallelizing Vector Addition

C

```
void vadd (real *A, real *B, int n){
  int i; for (i=0; i<n; i++) A[i]+=B[i];
}
```

Cilk

```
cilk void vadd (real *A, real *B, int n){
  if (n<=BASE) {
    int i; for (i=0; i<n; i++) A[i]+=B[i];
  } else {
    spawn vadd(A, B, n/2);
    spawn vadd(A+n/2, B+n/2, n-n/2);
    sync;
  }
}
```

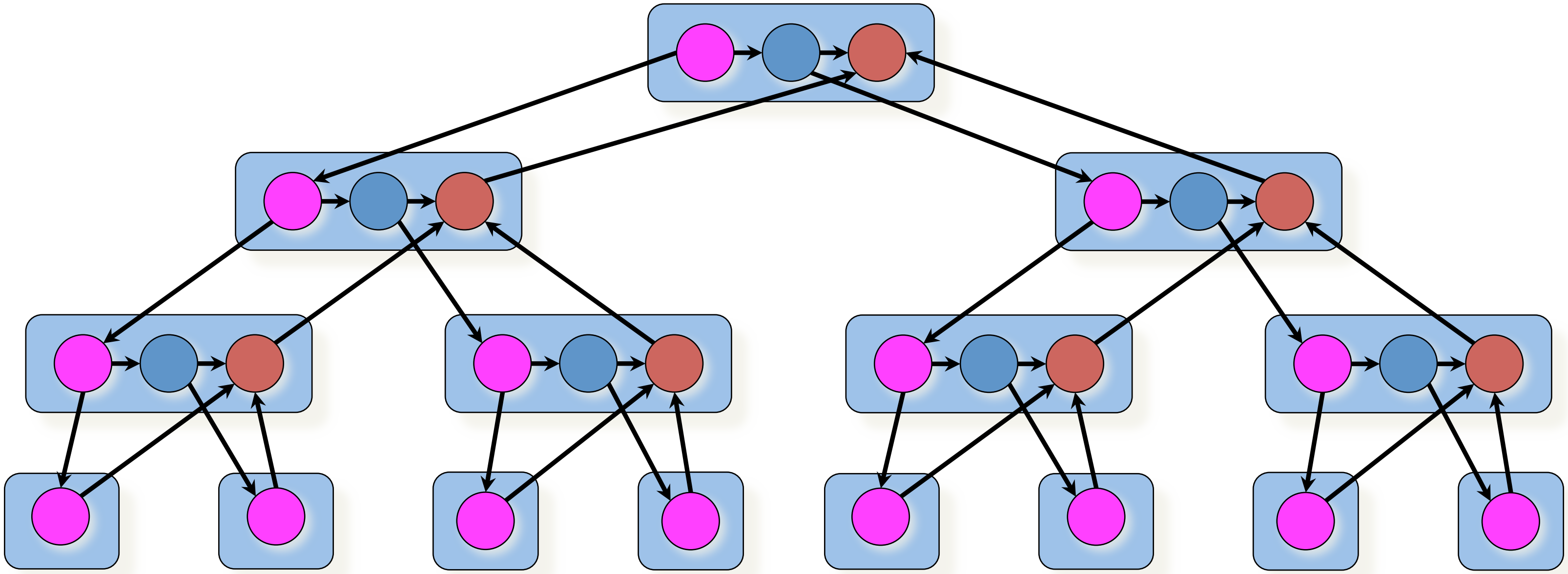
Parallelization strategy:

1. Convert loops to recursion.
2. Insert Cilk keywords.

Side benefit:
divide and conquer is generally good for caches!

Vector Addition

```
cilk void vadd (real *A, real *B, int n){
  if (n<=BASE) {
    int i; for (i=0; i<n; i++) A[i]+=B[i];
  } else {
    spawn vadd (A, B, n/2);
    spawn vadd (A+n/2, B+n/2, n-n/2);
    sync;
  }
}
```



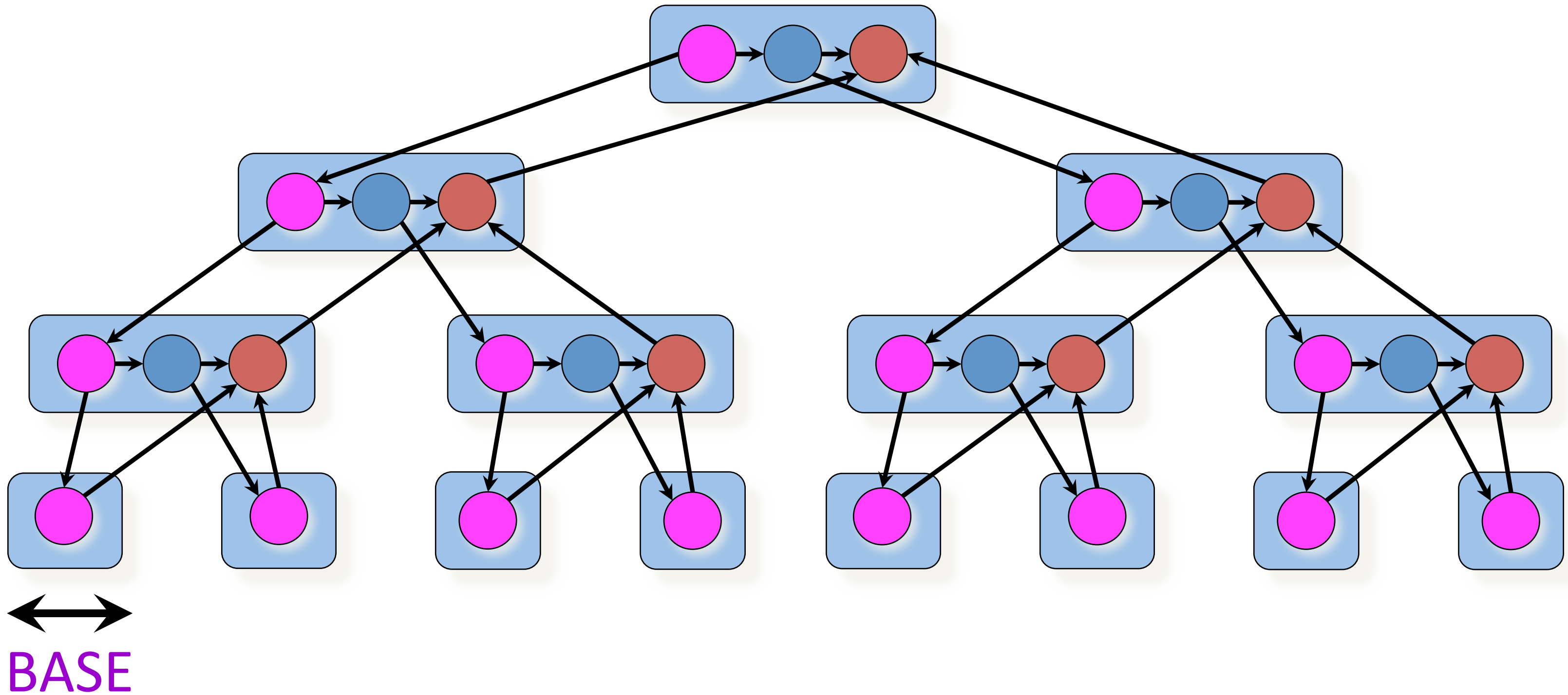
Vector Addition Analysis

To add two vectors of length n, where BASE = $\Theta(1)$:

Work: T_1 = ? $\Theta(n)$

Span: T_∞ = ? $\Theta(\log n)$

Parallelism: T_1/T_∞ = ? $\Theta(n/\log n)$



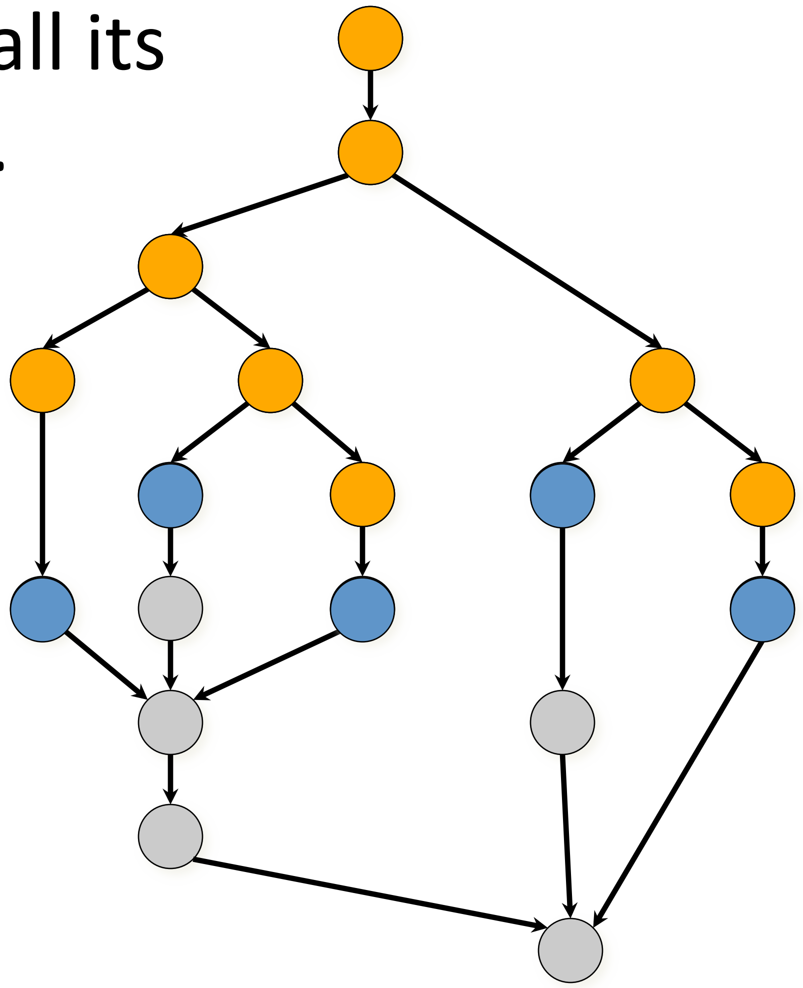
Outline for Cilk/Cilkplus

- Introduction and Basic Cilk Programming
- Cilk Work-stealing Scheduler
- Implementation Strategies
- Performance Analysis
- ☛ Scheduling Performance Analysis
- More Examples

Analysis: Greedy Scheduling

IDEA: Do as much as possible on every step.

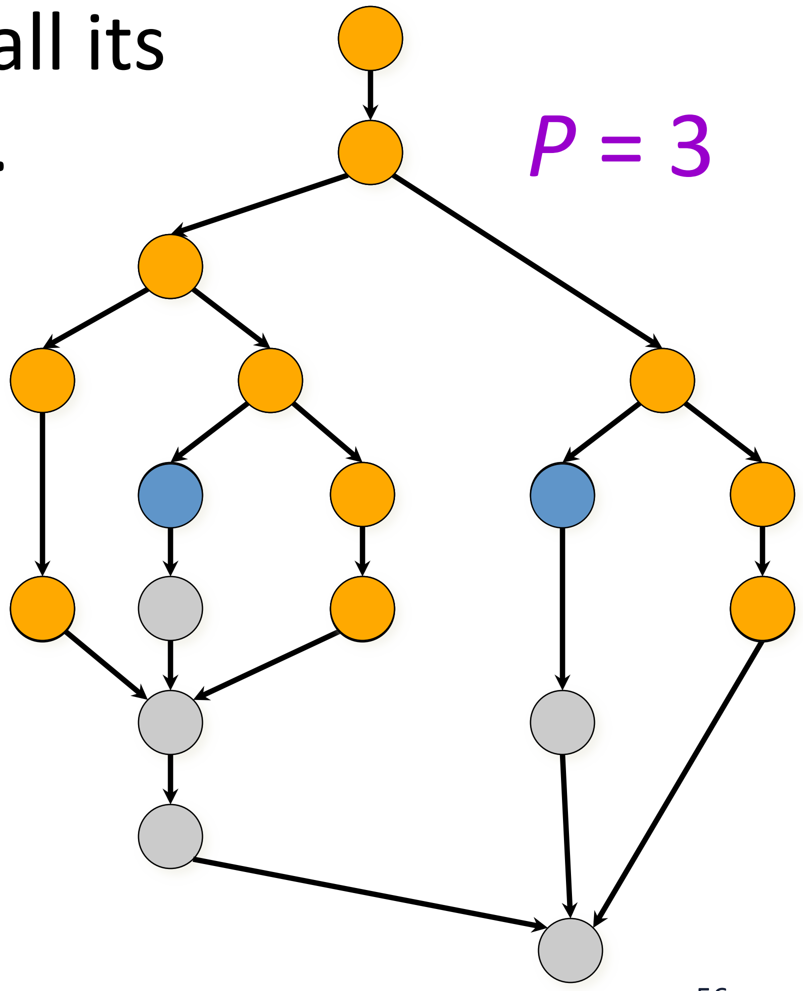
Definition: A task is ***ready*** if all its predecessors have ***executed***.



Greedy Scheduling

IDEA: Do as much as possible on every step.

Definition: A task is ***ready*** if all its predecessors have ***executed***.



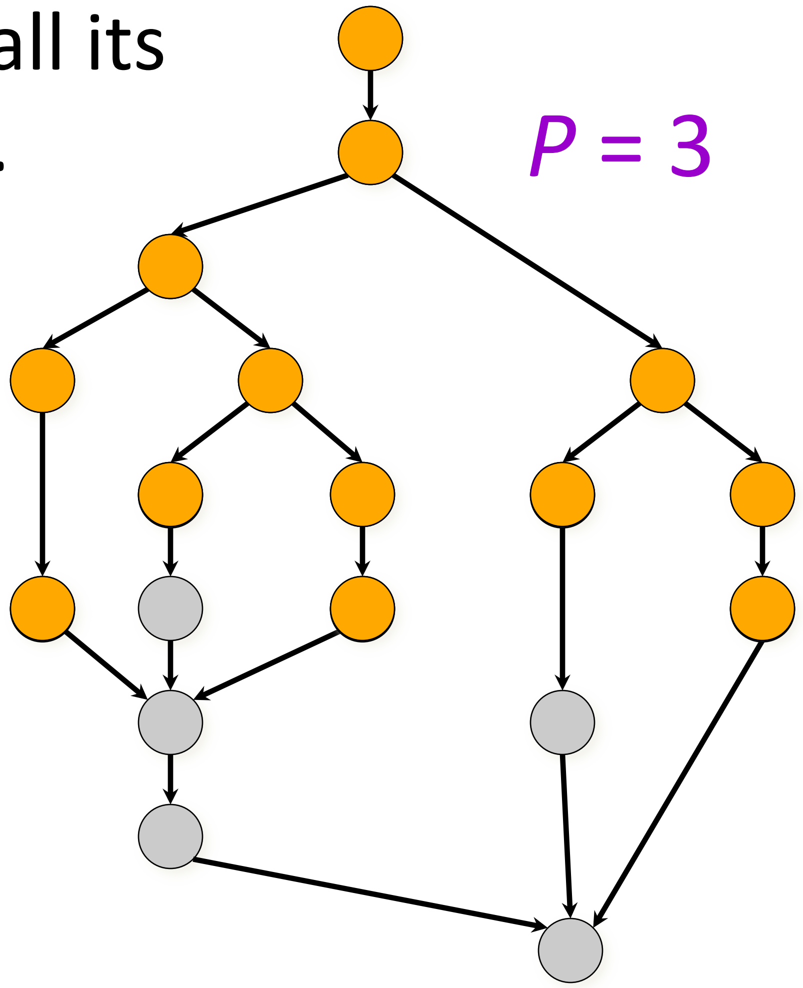
Complete step

- >= P tasks ready.
- Run any P.

Greedy Scheduling

IDEA: Do as much as possible on every step.

Definition: A task is ***ready*** if all its predecessors have ***executed***.



$P = 3$

Complete step

- >= P tasks ready.
- Run any on P.

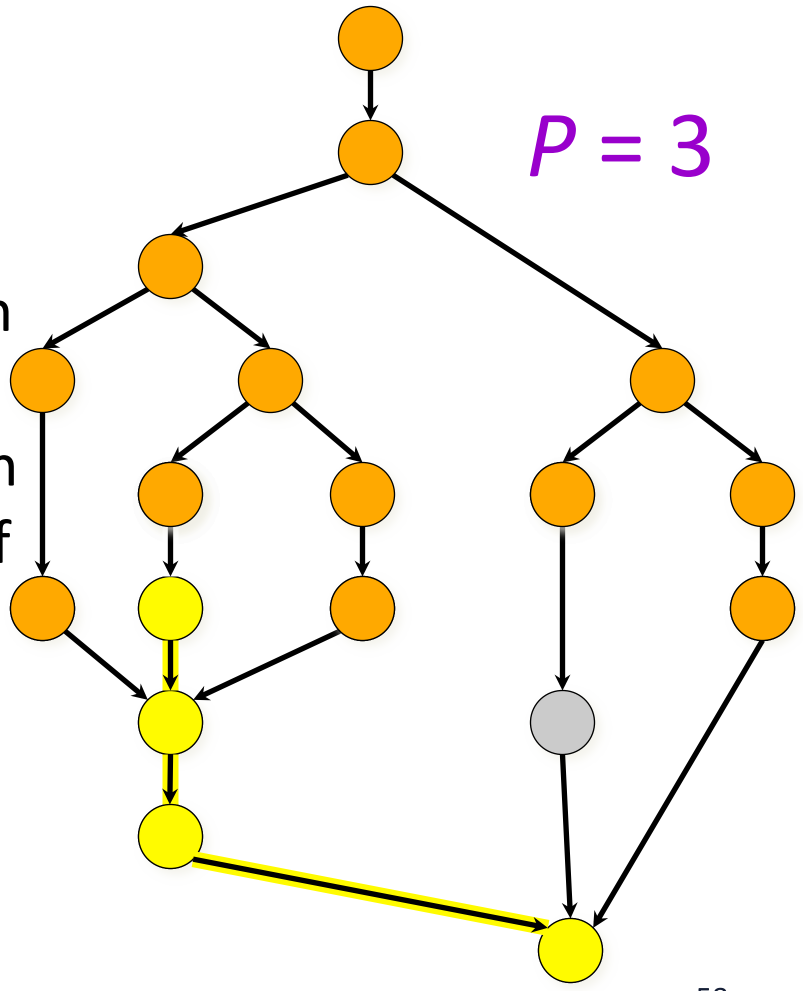
Incomplete step

- < P tasks ready.
- Run all of them.

Greedy-Scheduling Theorem

Theorem [Graham ’68 & Brent ’75]. Any greedy scheduler achieves

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



***Proof*.**

- # complete steps <= T_1/P, since each complete step performs P work.
- # incomplete steps <= T_∞, since each incomplete step reduces the span of the unexecuted dag by 1.

Performance of Work-Stealing

Theorem: On *P* processors, Cilk's work-stealing scheduler achieves an expected running time of

$$T_P = T_1/P + O(T_\infty)$$

work term

Critical path term

Critical Path Overhead

- Critical path overhead = smallest constant C_∞ such that

$$T_p \le \frac{T_1}{P} + c_\infty T_\infty$$

$$T_p \le \left(\frac{T_1}{T_\infty P} + c_\infty\right) T_\infty = \left(\frac{\overline{P}}{P} + c_\infty\right) T_\infty$$

Let $\overline{P}$ = T_1/T_∞ = parallelism = max speedup on ∞ processors

Parallel slackness assumption

$$\overline{P}/P >> c_\infty$$ **thus** $$\frac{T_1}{P} >> c_\infty T_\infty$$

$$T_p \approx \frac{T_1}{P}$$ **linear speedup**

"critical path overhead has little effect on performance when sufficient parallel slackness exists"

Work Overhead

$$c_1 = \frac{T_1}{T_s}$$

work overhead

$$T_p \leq c_1 \frac{T_s}{P} + c_\infty T_\infty$$

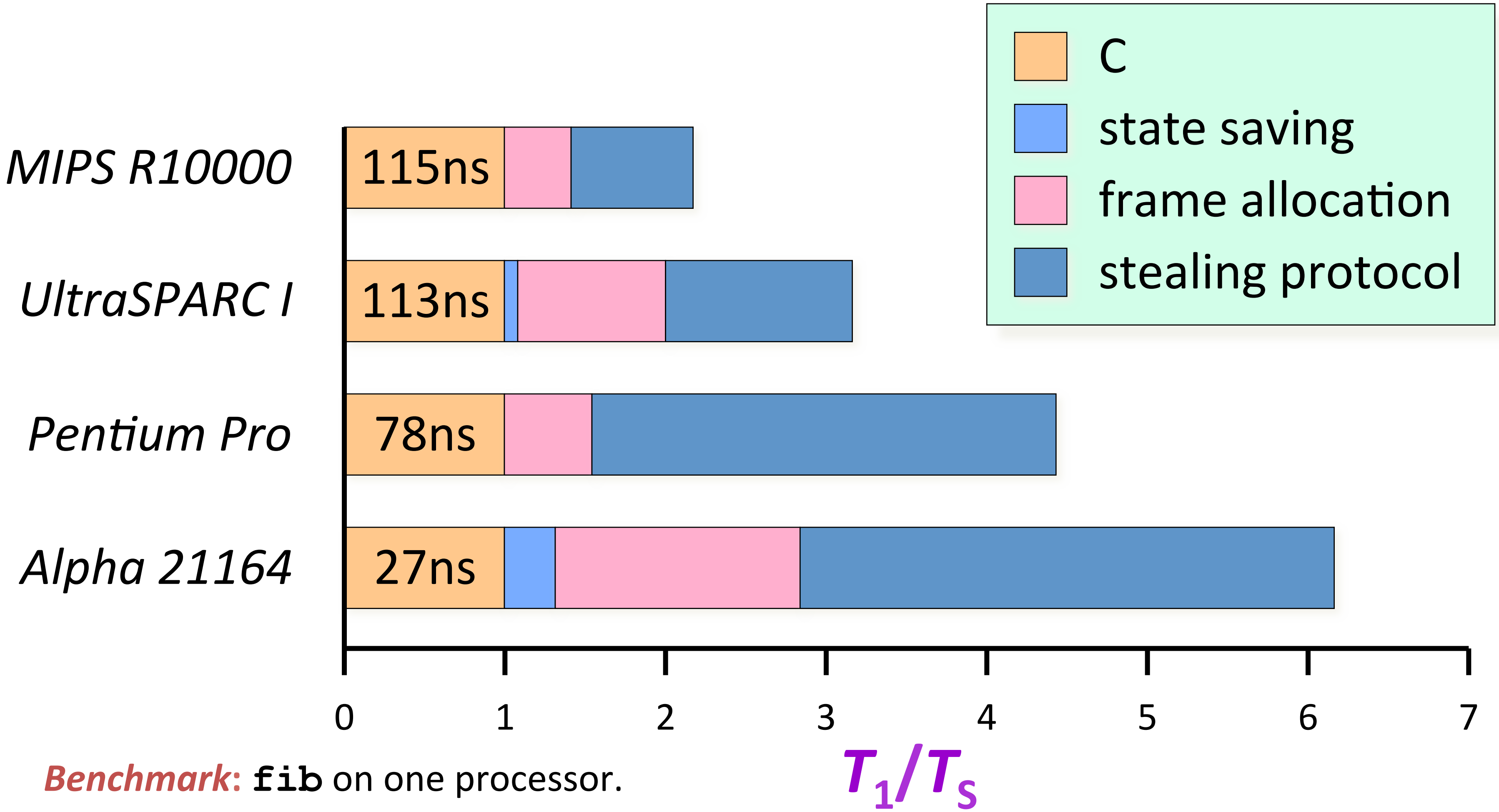
"Minimize work overhead (c_1) at the expense of a larger critical path overhead (c_∞), because work overhead has a more direct impact on performance"

$$T_p \approx c_1 \frac{T_s}{P}$$

assuming parallel slackness

You can reduce C_1 by increasing the granularity of parallel work

Breakdown of Work Overhead



***Benchmark*: `fib` on one processor.**

The average cost of a **spawn** in Cilk-5 is only 2–6 times the cost of an ordinary C function call, depending on the platform.

Outline for Cilk/Cilkplus

- Introduction and Basic Cilk Programming
- Cilk Work-stealing Scheduler
- Implementation Strategies
- Performance Analysis
- Scheduling Performance Analysis

☛ More Examples

Square-Matrix Multiplication

$$\begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix}$$

$C \qquad\qquad A \qquad\qquad B$

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

Assume for simplicity that $n = 2^k$.

Recursive Matrix Multiplication

Divide and conquer —

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$= \begin{bmatrix} A_{11}B_{11} & A_{11}B_{12} \\ A_{21}B_{11} & A_{21}B_{12} \end{bmatrix} + \begin{bmatrix} A_{12}B_{21} & A_{12}B_{22} \\ A_{22}B_{21} & A_{22}B_{22} \end{bmatrix}$$

8 multiplications of $(n/2) \times (n/2)$ matrices.
1 addition of $n \times n$ matrices.

Matrix Multiplication

```
cilk void MultA(*C, *A, *B, n) {
  // C = C + A * B
  h base case & partition matrices i
  spawn MultA(C11,A11,B11,n/2);
  spawn MultA(C12,A11,B12,n/2);
  spawn MultA(C22,A21,B12,n/2);
  spawn MultA(C21,A21,B11,n/2);
  sync;
  spawn MultA(C21,A22,B21,n/2);
  spawn MultA(C22,A22,B22,n/2);
  spawn MultA(C12,A12,B22,n/2);
  spawn MultA(C11,A12,B21,n/2);
  sync;
  return;
}
```

Work of Multiply

```
cilk void MultA(*C, *A, *B, n) {
  // C = C + A * B
  h base case & partition matrices i
  spawn MultA(C11,A11,B11,n/2);
  spawn MultA(C12,A11,B12,n/2);
  spawn MultA(C22,A21,B12,n/2);
  spawn MultA(C21,A21,B11,n/2);
  sync;
  spawn MultA(C21,A22,B21,n/2);
  spawn MultA(C22,A22,B22,n/2);
  spawn MultA(C12,A12,B22,n/2);
  spawn MultA(C11,A12,B21,n/2);
  sync;
  return;
}
```

Work: $T_1(n) = \Theta(n^3)$

Span of Multiply

maximum

maximum

```
cilk void MultA(*C, *A, *B, n) {
  // C = C + A * B
  h base case & partition matrices i
  spawn MultA(C11,A11,B11,n/2);
  spawn MultA(C12,A11,B12,n/2);
  spawn MultA(C22,A21,B12,n/2);
  spawn MultA(C21,A21,B11,n/2);
  sync;
  spawn MultA(C21,A22,B21,n/2);
  spawn MultA(C22,A22,B22,n/2);
  spawn MultA(C12,A12,B22,n/2);
  spawn MultA(C11,A12,B21,n/2);
  sync;
  return;
}
```

Span: $T_\infty(n) = 2\,T_\infty(n/2) + \Theta(1)$

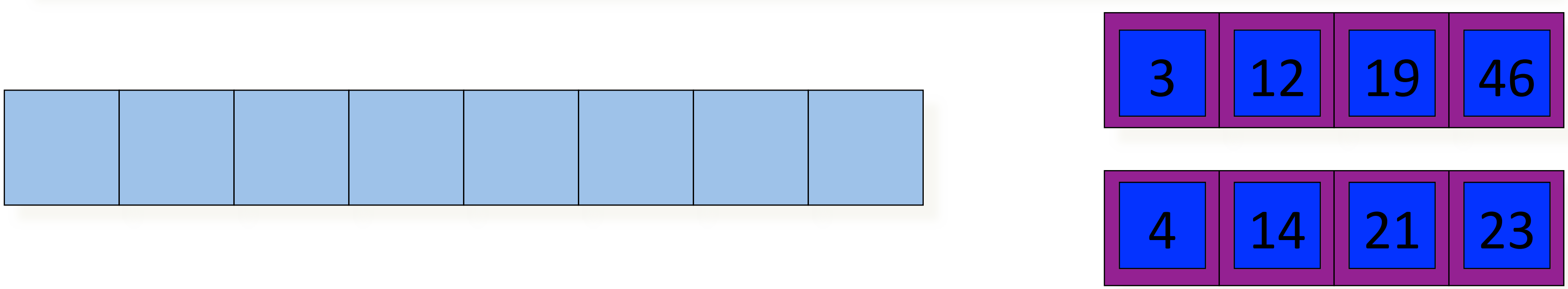
$= \Theta(n)$

Parallelism: $= T_1/T_\infty = \Theta(n^3) / \Theta(n) = \Theta(n^2)$

Merging Two Sorted Arrays

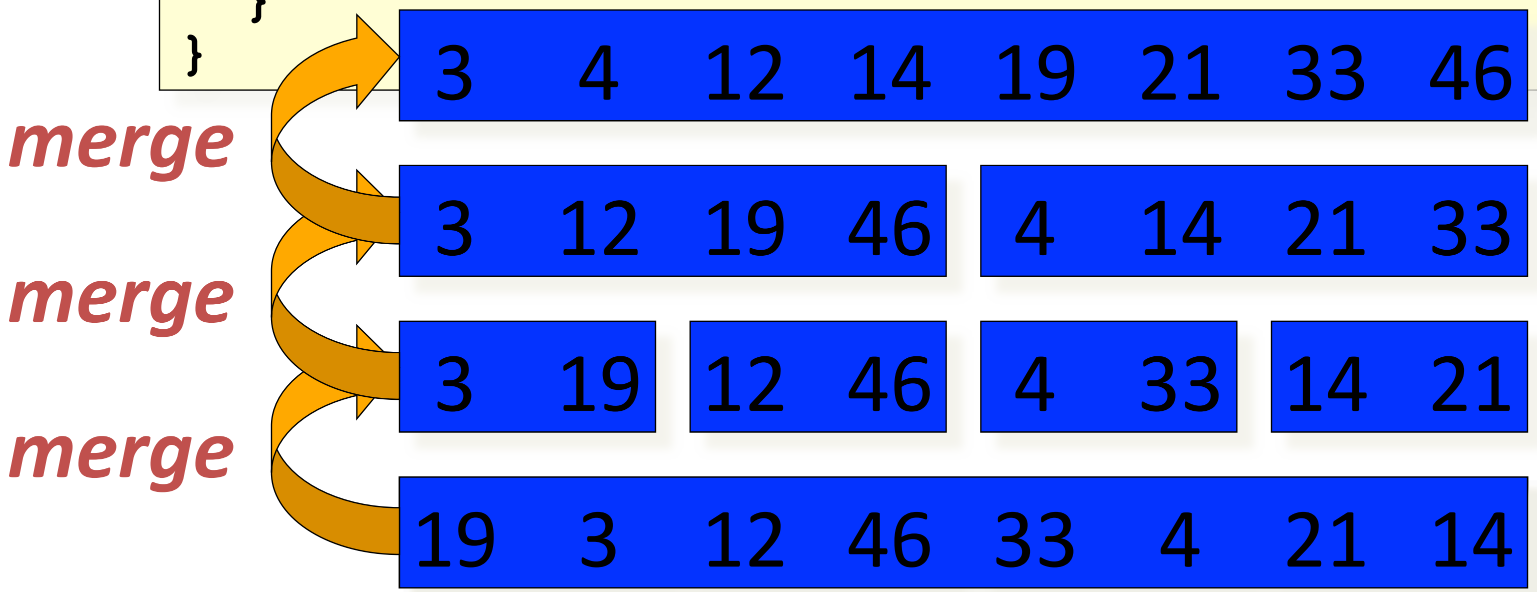
```
void Merge(int *C, int *A, int *B, int na, int nb) {
  while (na>0 && nb>0) {
    if (*A <= *B) {
      *C++ = *A++; na--;
    } else {
      *C++ = *B++; nb--;
    }
  }
  while (na>0) {
    *C++ = *A++; na--;
  }
  while (nb>0) {
    *C++ = *B++; nb--;
  }
}
```

Time to merge n elements = $\Theta(n)$.



Merge Sort

```
cilk void MergeSort(int *B, int *A, int n) {
  if (n==1) {
    B[0] = A[0];
  } else {
    int *C;
    C = (int*) Cilk_alloca(n*sizeof(int));
    spawn MergeSort(C, A, n/2);
    spawn MergeSort(C+n/2, A+n/2, n-n/2);
    sync;
    Merge(B, C, C+n/2, n/2, n-n/2);
  }
}
```



Work of Merge Sort

```
cilk void MergeSort(int *B, int *A, int n) {
  if (n==1) {
    B[0] = A[0];
  } else {
    int *C;
    C = (int*) Cilk_alloca(n*sizeof(int));
    spawn MergeSort(C, A, n/2);
    spawn MergeSort(C+n/2, A+n/2, n-n/2);
    sync;
    Merge(B, C, C+n/2, n/2, n-n/2);
  }
}
```

Work: $T_1(n) = 2\,T_1(n/2) + \Theta(n)$

$= \Theta(n \lg n)$

Span of Merge Sort

```
cilk void MergeSort(int *B, int *A, int n) {
  if (n==1) {
    B[0] = A[0];
  } else {
    int *C;
    C = (int*) Cilk_alloca(n*sizeof(int));
    spawn MergeSort(C, A, n/2);
    spawn MergeSort(C+n/2, A+n/2, n-n/2);
    sync;
    Merge(B, C, C+n/2, n/2, n-n/2);
  }
}
```

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

$= \Theta(n)$

Parallelism: $\dfrac{T_1(n)}{T_\infty(n)} = \Theta(\lg n)$

Tableau Construction

Problem: Fill in an n x n tableau A, where

$$A[i, j] = f\big(A[i, j-1], A[i-1, j], A[i-1, j-1]\big).$$

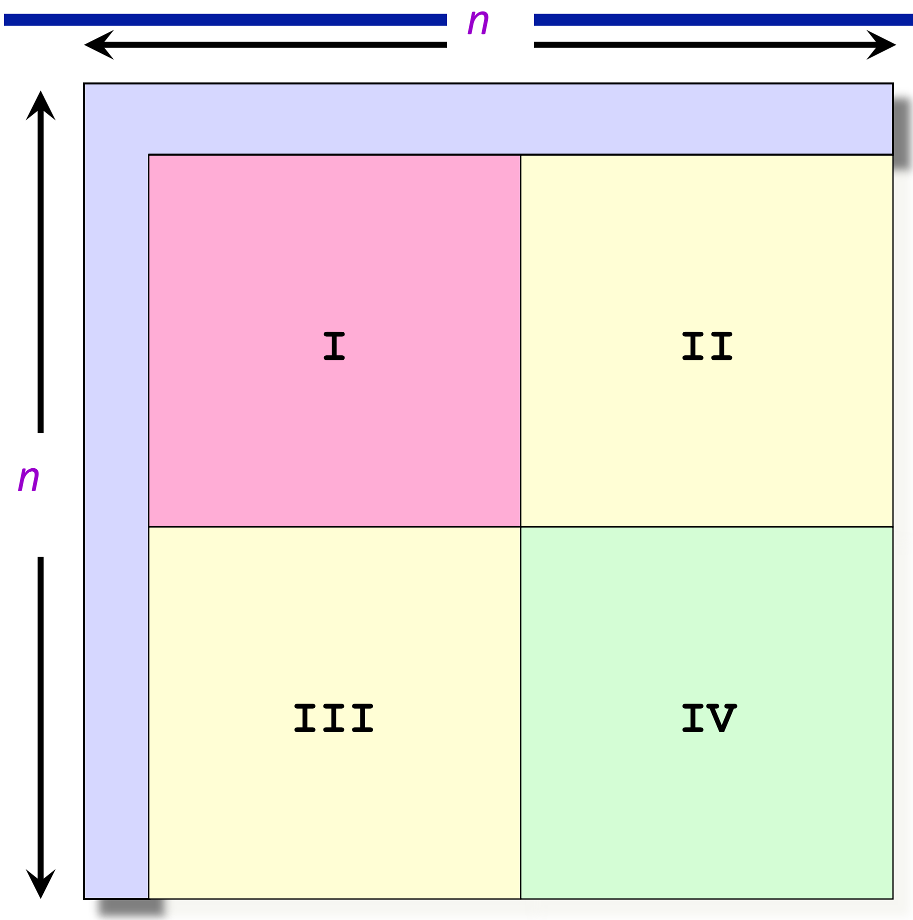
00	01	02	03	04	05	06	07
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

Dynamic programming

- Longest common subsequence
- Edit distance
- Time warping

Work: $\Theta(n^2)$.

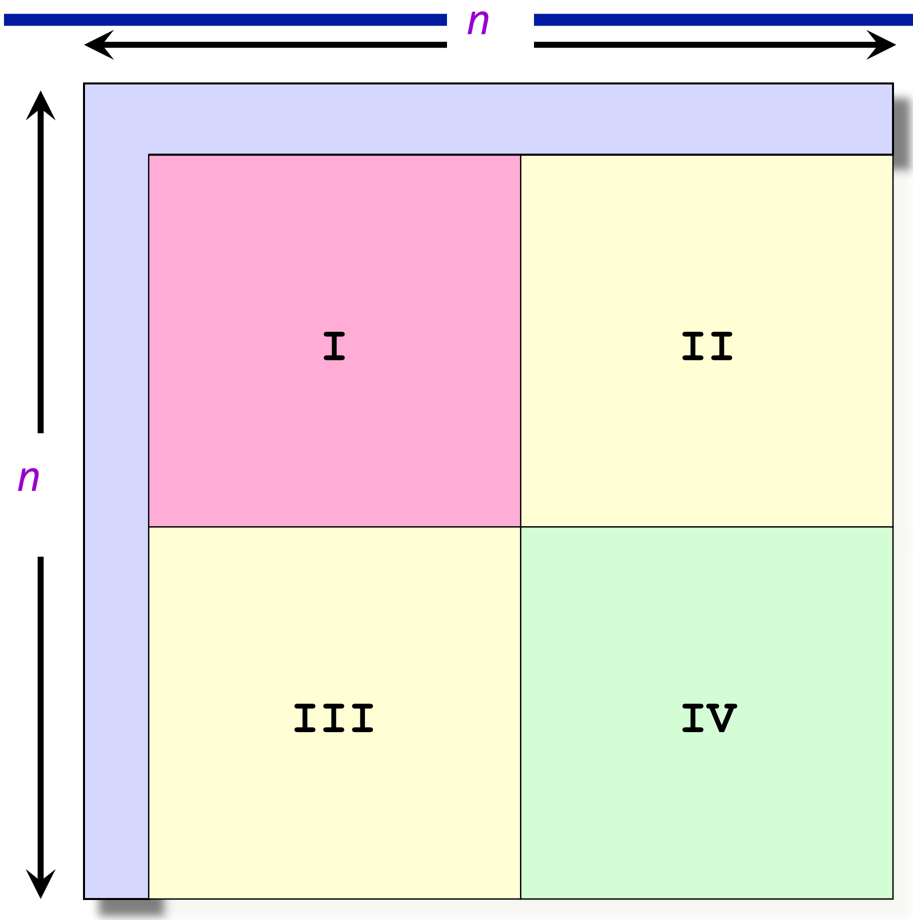
Recursive Construction



Cilk code

```
spawn I;
sync;
spawn II;
spawn III;
sync;
spawn IV;
sync;
```

Recursive Construction



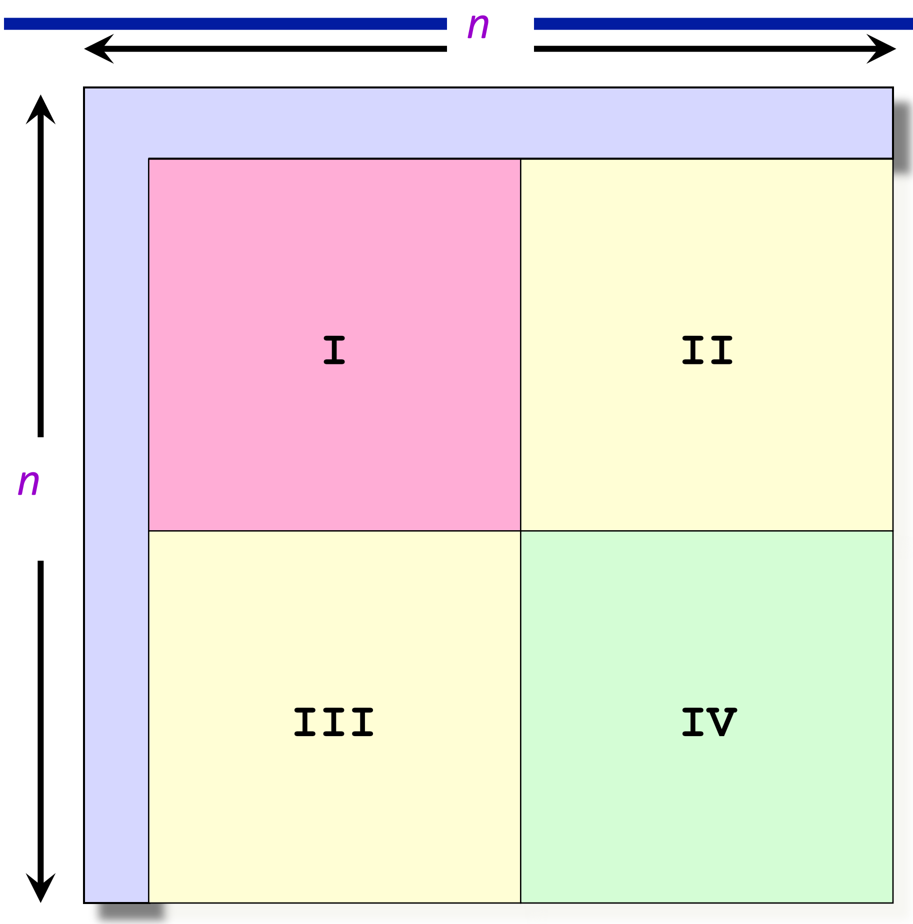
Cilk code

```
spawn I;
sync;
spawn II;
spawn III;
sync;
spawn IV;
sync;
```

Work: $T_1(n) = 4T_1(n/2) + \Theta(1)$

$= \Theta(n^2)$

Recursive Construction



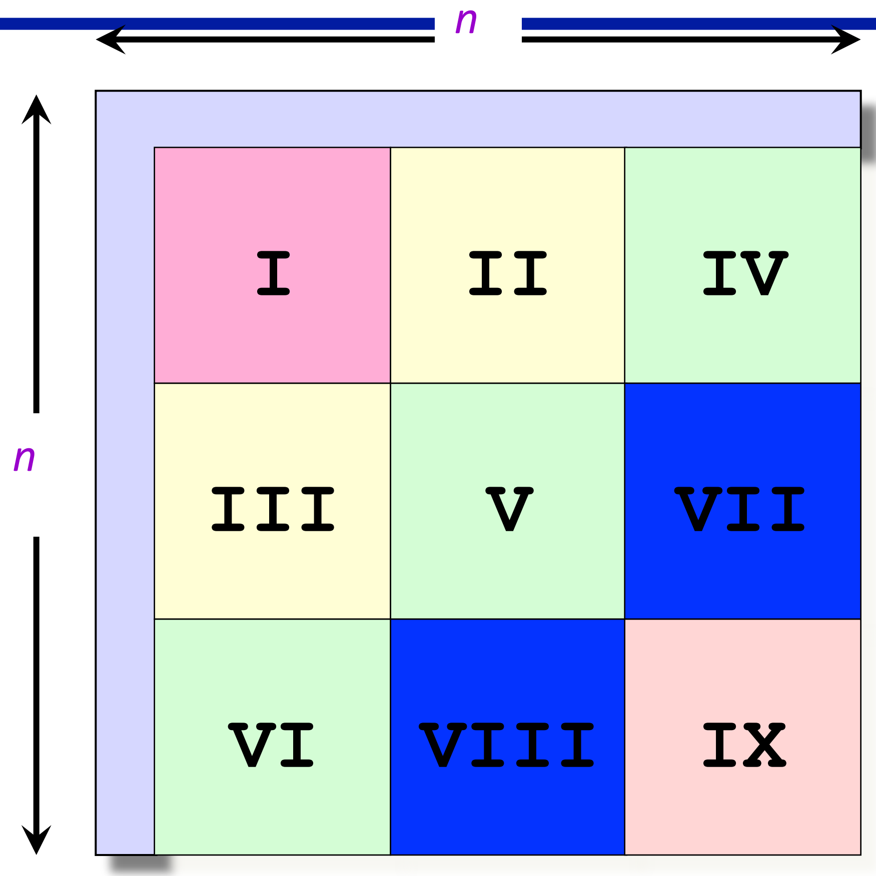
Cilk code

```
spawn I;
sync;
spawn II;
spawn III;
sync;
spawn IV;
sync;
```

Span: $T_\infty(n) = 3T_\infty(n/2) + \Theta(1) \quad = \Theta(n^{\lg 3})$

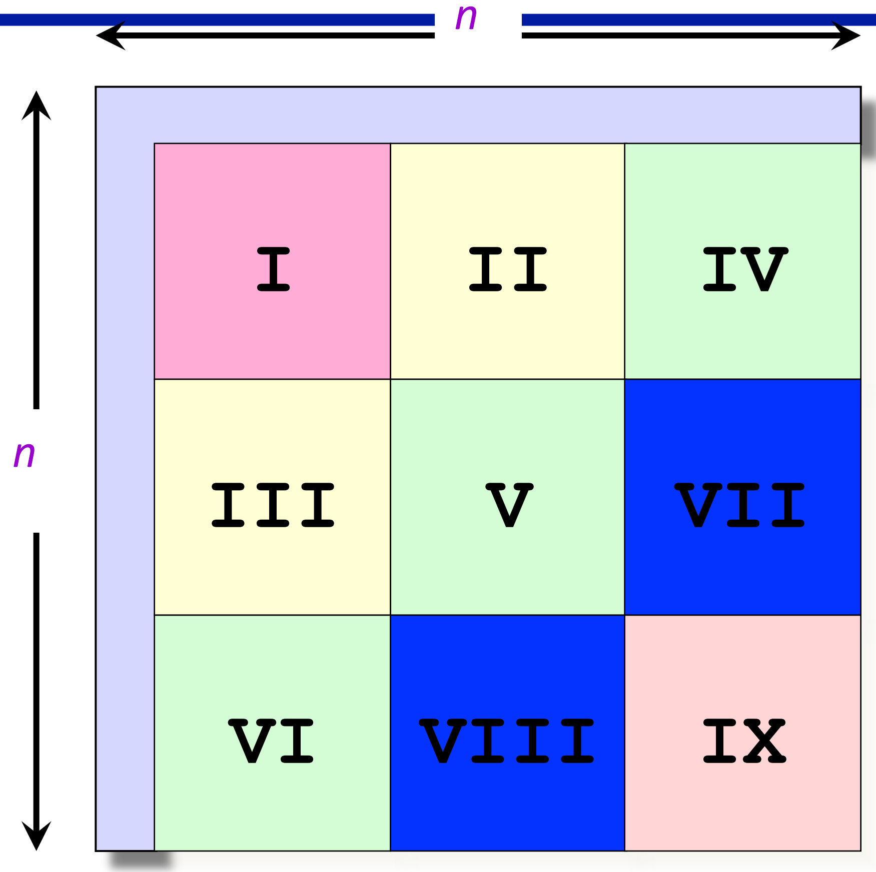
Parallelism: $\dfrac{T_1(n)}{T_\infty(n)} \approx \Theta(n^{0.42})$

A More-Parallel Construction




```
spawn I;
sync;
spawn II;
spawn III;
sync;
spawn IV;
spawn V;
spawn VI
sync;
spawn VII;
spawn VIII;
sync;
spawn IX;
sync;
```

A More-Parallel Construction

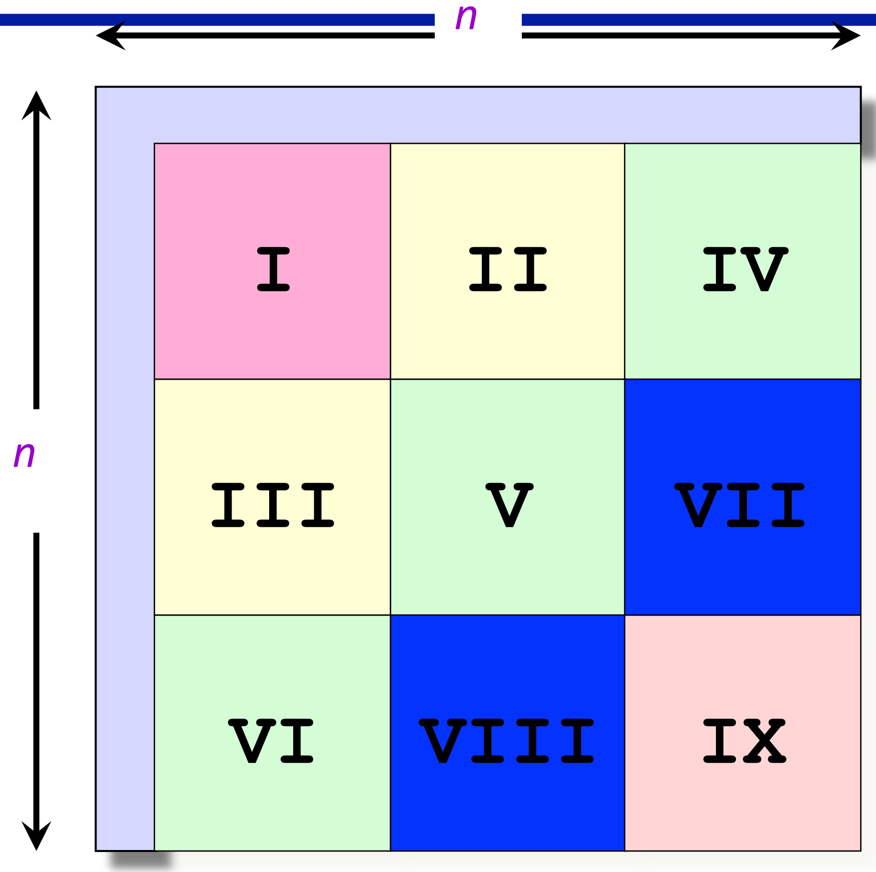



```
spawn I;
sync;
spawn II;
spawn III;
sync;
spawn IV;
spawn V;
spawn VI
sync;
spawn VII;
spawn VIII;
sync;
spawn IX;
sync;
```

Work: $T_1(n) = 9T_1(n/3) + \Theta(1)$

$= \Theta(n^2)$

A More-Parallel Construction




```
spawn I;
sync;
spawn II;
spawn III;
sync;
spawn IV;
spawn V;
spawn VI
sync;
spawn VII;
spawn VIII;
sync;
spawn IX;
sync;
```

Span: $T_\infty(n) = 5T_\infty(n/3) + \Theta(1) = \Theta(n^{\log_3 5})$

Analysis of Revised Construction

Work: $T_1(n) = \Theta(n^2)$

Span: $T_\infty(n) = \Theta(n^{\log_3 5})$

$\approx \Theta(n^{1.46})$

Parallelism: $\dfrac{T_1(n)}{T_\infty(n)} \approx \Theta(n^{0.54})$

More parallel by a factor of

$\Theta(n^{0.54})/\Theta(n^{0.42}) = \Theta(n^{0.12})$.

References

- “Introduction to Parallel Computing” by Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. Addison Wesley, 2003
- Charles E. Leiserson. Cilk LECTURE 1. Supercomputing Technologies Research Group. Computer Science and Artificial Intelligence Laboratory. http://bit.ly/mit-cilk-lec1
- Charles Leiserson, Bradley Kuzmaul, Michael Bender, and Hua-wen Jing. MIT 6.895 lecture notes - Theory of Parallel Systems. http://bit.ly/mit-6895-fall03
- Intel Cilk++ Programmer’s Guide. Document # 322581-001US.