COMP 422, Lecture 5: Programming Shared Address Space Platforms

(Sections 7.1 - 7.4 of textbook & new material on Cilk)

Vivek Sarkar

Department of Computer Science
Rice University

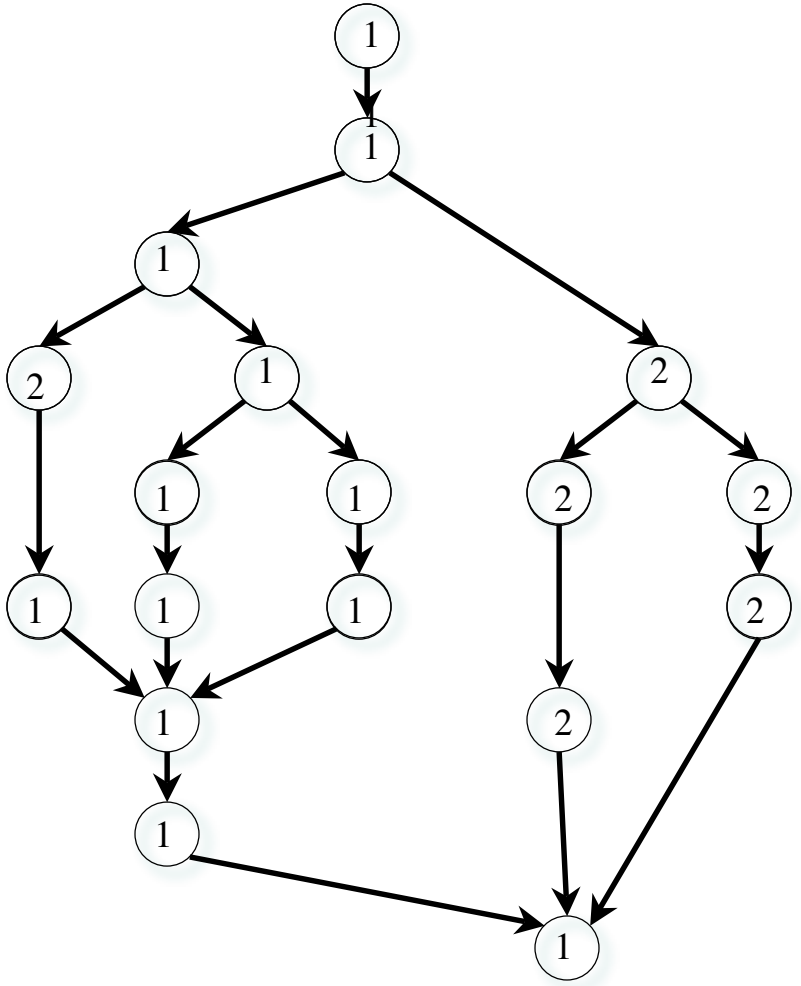
vsarkar@rice.edu

Recap of Lecture 4

- Tasks, Dependence Graphs, Scheduling Theory
  - —Task = indivisible sequential unit of computation
  - —Task Dependency Graph
  - —T_P = execution time on P processors
    - T_1 = sequential time = total work
    - T_∞ = critical path length
  - — Lower & upper bounds: $\max(T_1/P, T_\infty) \leq T_P \leq T_1/P + T_\infty$
- Decomposition Techniques
  - —recursive decomposition
  - data decomposition
  - exploratory decomposition
  - speculative decomposition

Lecture 4 Review Question

- Give lower and upper bounds on the parallel execution time of this task graph on
  - 2 processors, and
  - 100 processors



Acknowledgments for today's lecture

- Slides accompanying course textbook
  - http://www-users.cs.umn.edu/~karypis/parbook/
- Cilk lecture by Charles Leiserson and Bradley Kuszmaul (Lecture 1, Scheduling Theory)
  - http://supertech.csail.mit.edu/cilk/lecture-1.pdf
- John Mellor-Crummey (Rice) --- COMP 422 slides from Spring 2007 (Lecture 4)

Outline of Today's Lecture

- **Thread Basics**
- **Introduction to Cilk**

Overview of Programming Models

- A *thread* is a single stream of control in the flow of a program. A program like:

```
for (row = 0; row < n; row++)
  for (column = 0; column < n; column++)
          c[row][column] =
          dot_product( get_row(a, row),
                       get_col(b, col));
```

can be transformed to:

```
for (row = 0; row < n; row++)
  for (column = 0; column < n; column++)
          c[row][column] =
          create_thread( dot_product(get_row(a, row),
          get_col(b, col)));
```

In this case, one may think of the thread as an instance of a function that returns before the function has finished executing.

Thread Basics

- All memory in the logical machine model of a thread is globally accessible to every thread.
- The stack corresponding to the function call is generally treated as being local to the thread for liveness reasons.
- This implies a logical machine model with both global memory (default) and local memory (stacks).
- It is important to note that such a flat model may result in very poor performance since memory is physically distributed in typical machines.

The POSIX Thread API

- Commonly referred to as Pthreads, POSIX has emerged as the standard threads API, supported by most vendors.
- The concepts discussed here are largely independent of the API and can be used for programming with other thread APIs (NT threads, Solaris threads, Java threads, etc.) as well.

Thread Basics: Creation and Termination

- Pthreads provides two basic functions for specifying concurrency in a program:

```
#include <pthread.h>
int pthread_create (
    pthread_t *thread_handle, const pthread_attr_t *attribute,
    void * (*thread_function)(void *),
    void *arg);
int pthread_join (
    pthread_t thread,
    void **ptr);
```

- The function pthread_create invokes function thread_function as a thread

Thread Basics: Creation and Termination (Example)

```
#include <pthread.h>
#include <stdlib.h>
#define MAX_THREADS 512
void *compute_pi (void *);
....
main() {
      ...
      pthread_t p_threads[MAX_THREADS];
      pthread_attr_t attr;
      pthread_attr_init (&attr);
      for (i=0; i< num_threads; i++) {
          hits[i] = i;
          pthread_create(&p_threads[i], &attr, compute_pi,
                (void *) &hits[i]);
      }
      for (i=0; i< num_threads; i++) {
          pthread_join(p_threads[i], NULL);
          total_hits += hits[i];
      }
      ...
}
```

Thread Basics: Creation and Termination (Example)

```
void *compute_pi (void *s) {
      int seed, i, *hit_pointer;
      double rand_no_x, rand_no_y;
      int local_hits;
      hit_pointer = (int *) s;
      seed = *hit_pointer;
      local_hits = 0;
      for (i = 0; i < sample_points_per_thread; i++) {
          rand_no_x =(double)(rand_r(&seed))/(double)((2<<14)-1);
          rand_no_y =(double)(rand_r(&seed))/(double)((2<<14)-1);
          if (((rand_no_x - 0.5) * (rand_no_x - 0.5) +
               (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)
               local_hits ++;
          seed *= i;
      }
      *hit_pointer = local_hits;
      pthread_exit(0);
}
```

Programming and Performance Notes

- Note the use of the function `rand_r` (instead of superior random number generators such as `drand48`).
- Executing this on a 4-processor SGI Origin, we observe a 3.91 fold speedup at 32 threads. This corresponds to a parallel efficiency of 0.98!
- We can also modify the program slightly to observe the effect of false-sharing.
- The program can also be used to assess the secondary cache line size.

Programming and Performance Notes

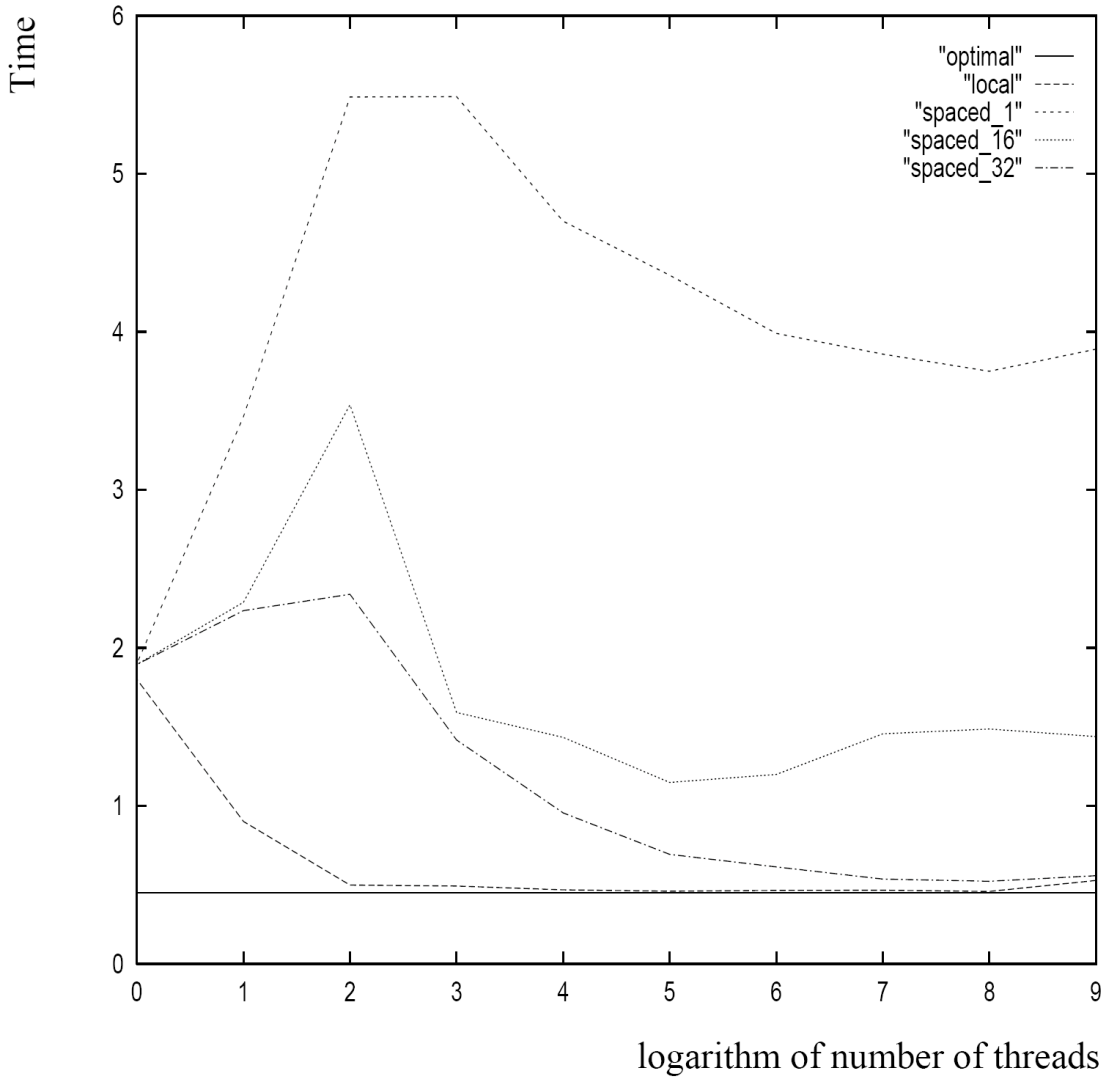


Figure 7.2 Execution time of the `compute_pi` program as a function of number of threads.

Outline of Today's Lecture

- **Thread Basics**
- **Introduction to Cilk**

Introducing Cilk

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

- Cilk constructs
  - —cilk: Cilk function. without it, functions are standard C
  - —spawn: call can execute asynchronously in a concurrent thread
  - —sync: current thread waits for all locally-spawned functions
- Cilk constructs specify logical parallelism in the program
  - —what computations can be performed in parallel
  - —not mapping of tasks to processes

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.

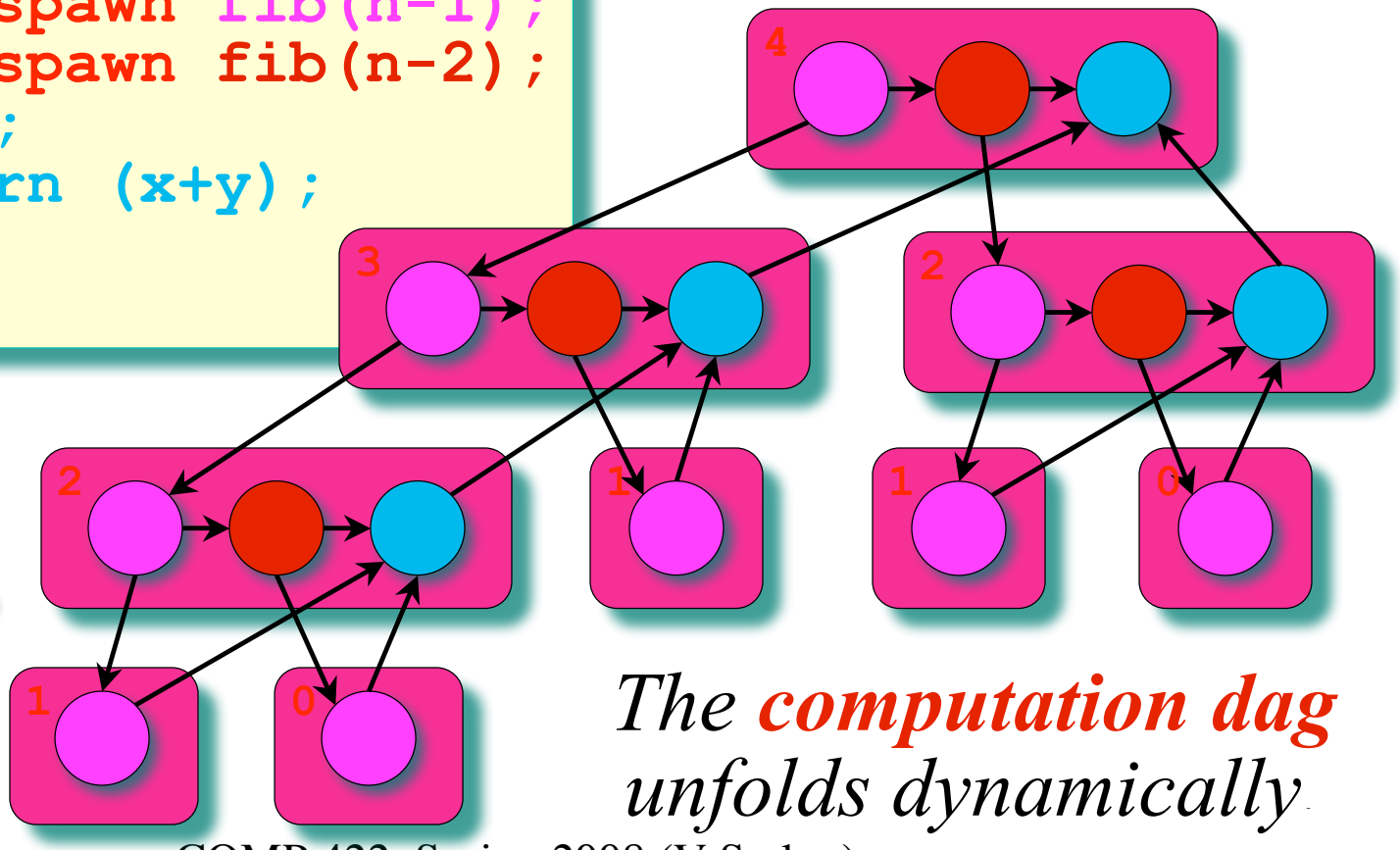
Cilk Language

- Cilk is a faithful extension of C
  - —if Cilk keywords are elided → C program semantics
- Idiosyncrasies
  - —spawn keyword can only be applied to a cilk function
  - —spawn keyword cannot be used in a C function
  - —cilk function cannot be called with normal C call conventions
    - must be called with a spawn & waited for by a sync

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);
  }
}
```

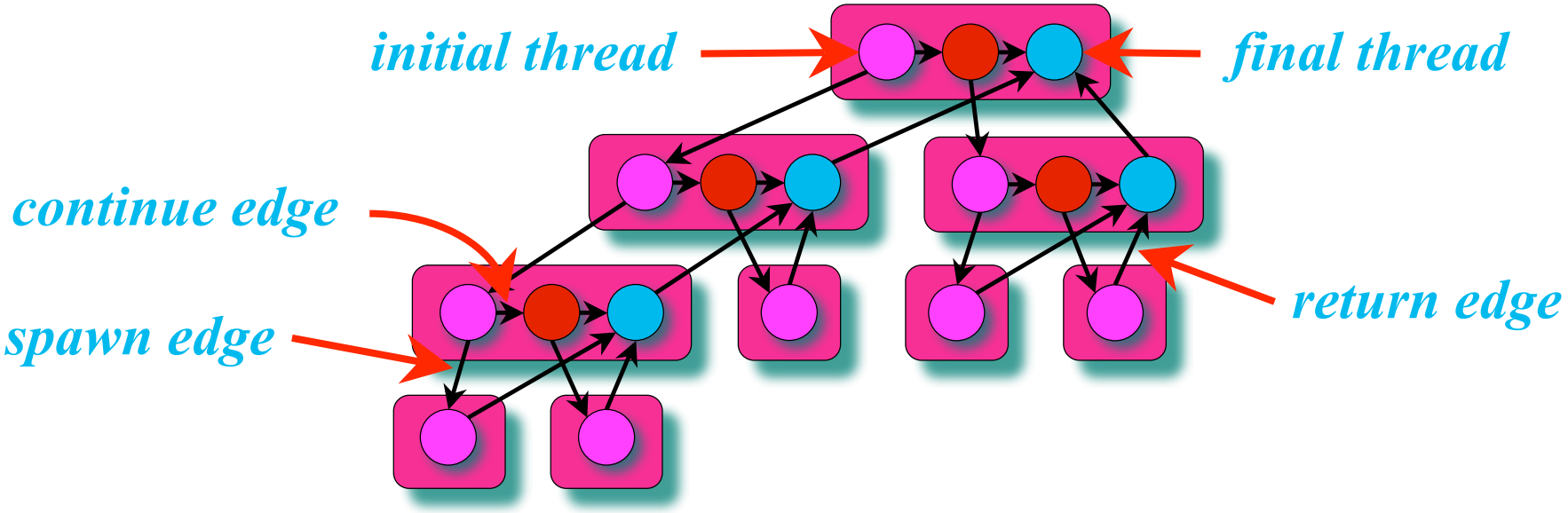
Example: `fib(4)`



"Processor oblivious"

The computation dag unfolds dynamically

Multithreaded Computation



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

Cilk Terminology

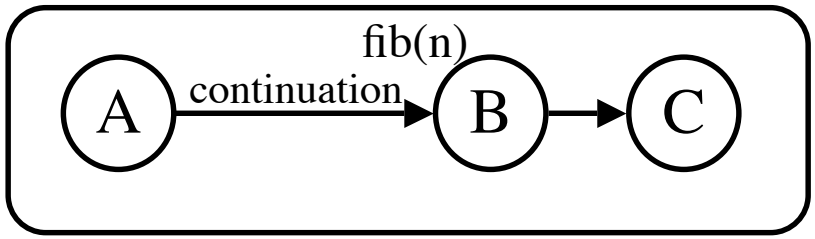
- **Parallel control = spawn, sync, return from spawned function**
- **Thread = maximal sequence of instructions not containing parallel control (task in earlier terminology)**

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

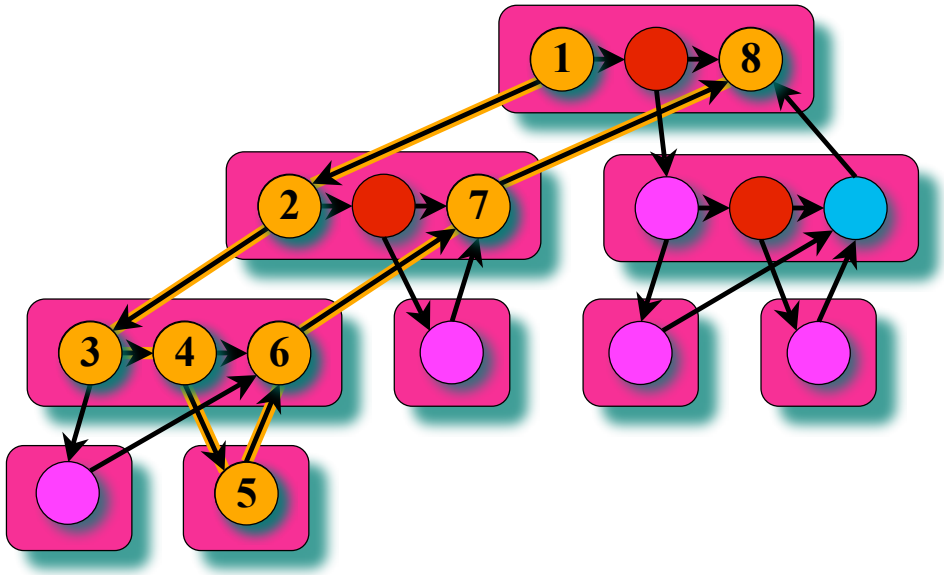
Thread A: if statement up to first spawn

Thread B: computation of n-2 before 2nd spawn

Thread C: n1+ n2 before the return



Example: `fib(4)`

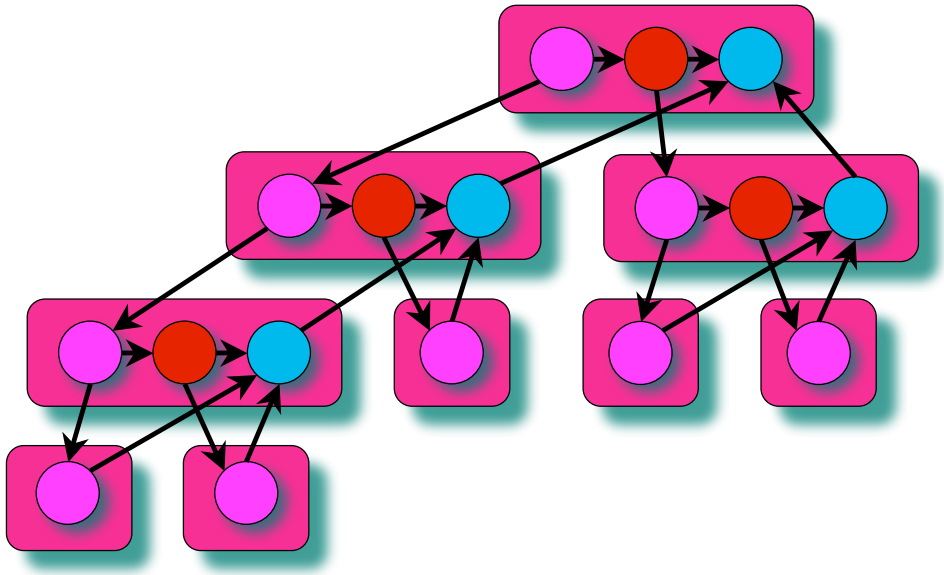


Assume for simplicity that each Cilk thread in **`fib()`** *takes unit time to execute.*

Work: $T_1 = 17$

Span: $T_\infty = 8$

Example: `fib(4)`



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

Work: $T_1 = 17$

Span: $T_\infty = 8$

Parallelism: $T_1 / T_\infty = 2.125$

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;
}
```

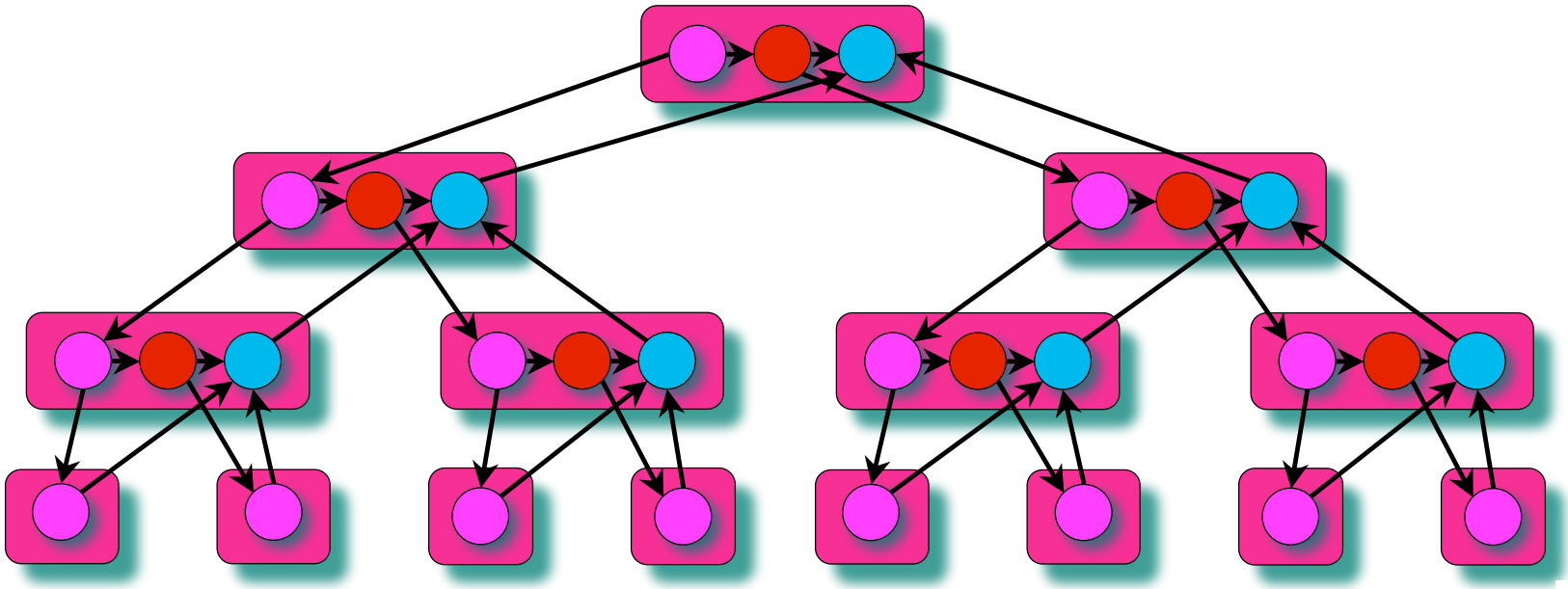
Parallelization strategy:

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

Side benefit:
D&C 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;
  }
}
```



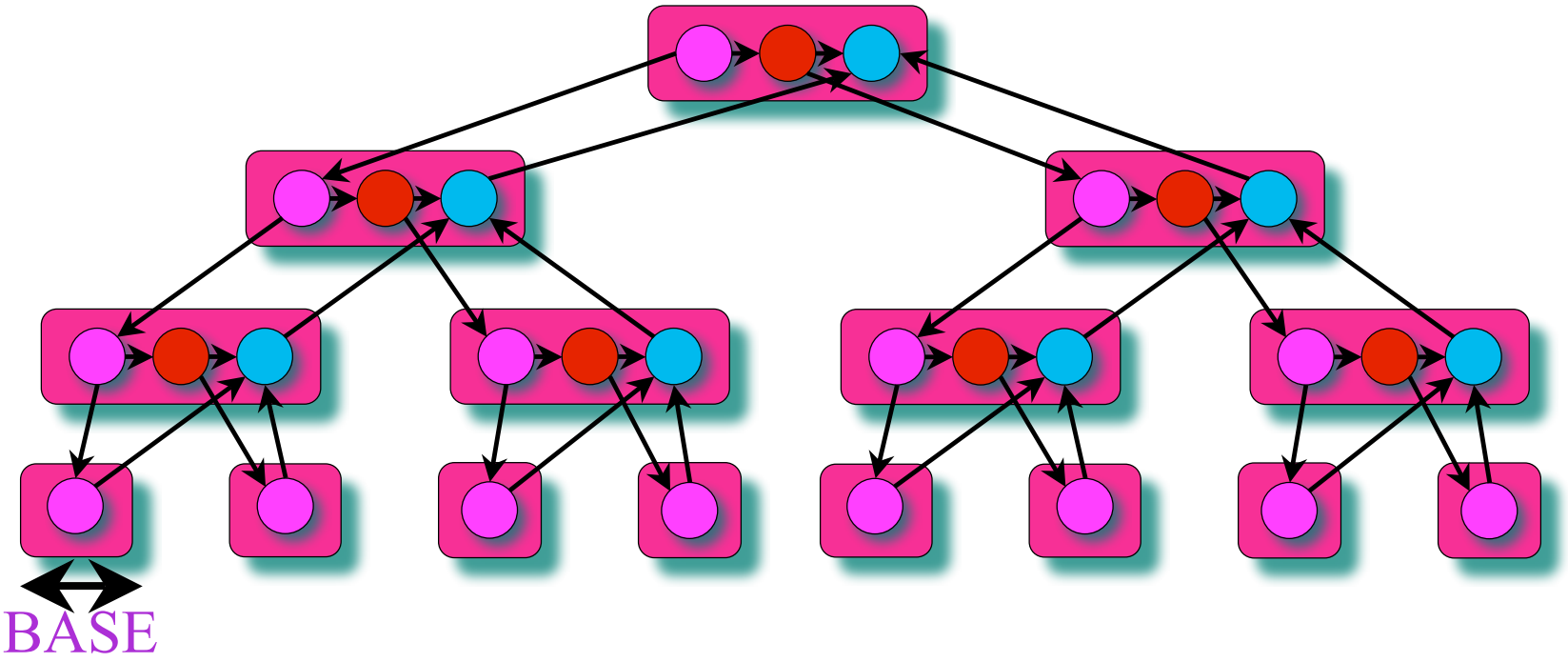
Vector Addition Analysis

To add two vectors of length *n*, where BASE = Θ(1):

Work: $T_1 = \Theta(n)$

Span: $T_\infty = \Theta(\lg n)$

Parallelism: $T_1 / T_\infty = \Theta(n/\lg n)$



Another Parallelization

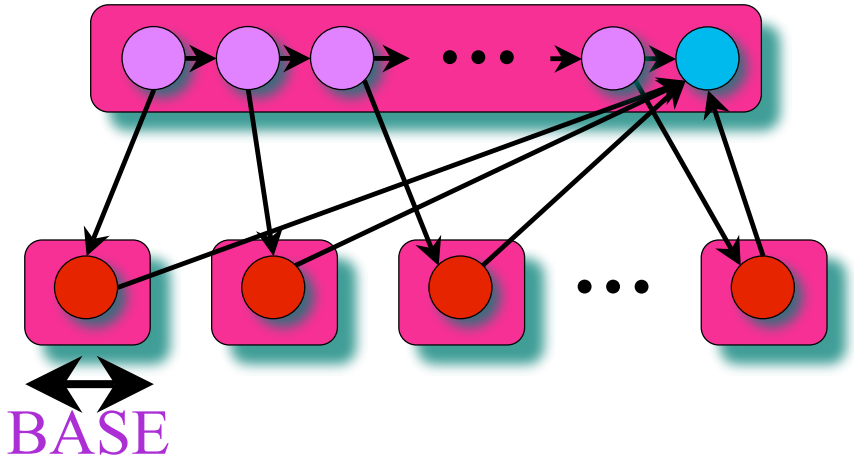
C

```
void vadd1 (real *A, real *B, int n){
  int i; for (i=0; i<n; i++) A[i]+=B[i];
}
void vadd (real *A, real *B, int n){
  int j; for (j=0; j<n; j+=BASE) {
    vadd1(A+j, B+j, min(BASE, n-j));
  }
}
```

Cilk

```
cilk void vadd1 (real *A, real *B, int n){
  int i; for (i=0; i<n; i++) A[i]+=B[i];
}
cilk void vadd (real *A, real *B, int n){
  int j; for (j=0; j<n; j+=BASE) {
    spawn vadd1(A+j, B+j, min(BASE, n-j));
  }
sync;
}
```

Analysis



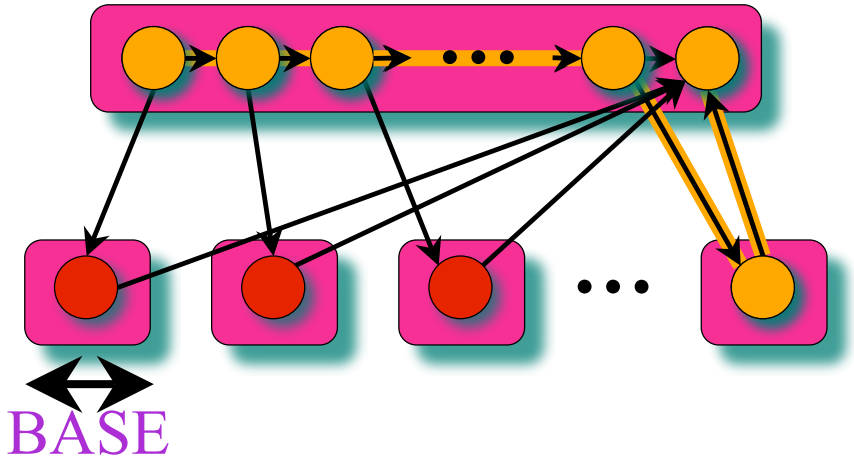
To add two vectors of length *n*, where BASE = Θ(1):

Work: $T_1 = \Theta(n)$

Span: $T_\infty = \Theta(n)$

Parallelism: $T_1 / T_\infty = \Theta(1)$

Optimal Choice of BASE



To add two vectors of length *n* using an optimal choice of BASE to maximize parallelism:

Work: $T_1 = \Theta(n)$

Span: $T_\infty = \Theta(\text{BASE} + n/\text{BASE})$

Choosing $\text{BASE} = \sqrt{n}$ **==>** $T_\infty = \Theta(\sqrt{n})$

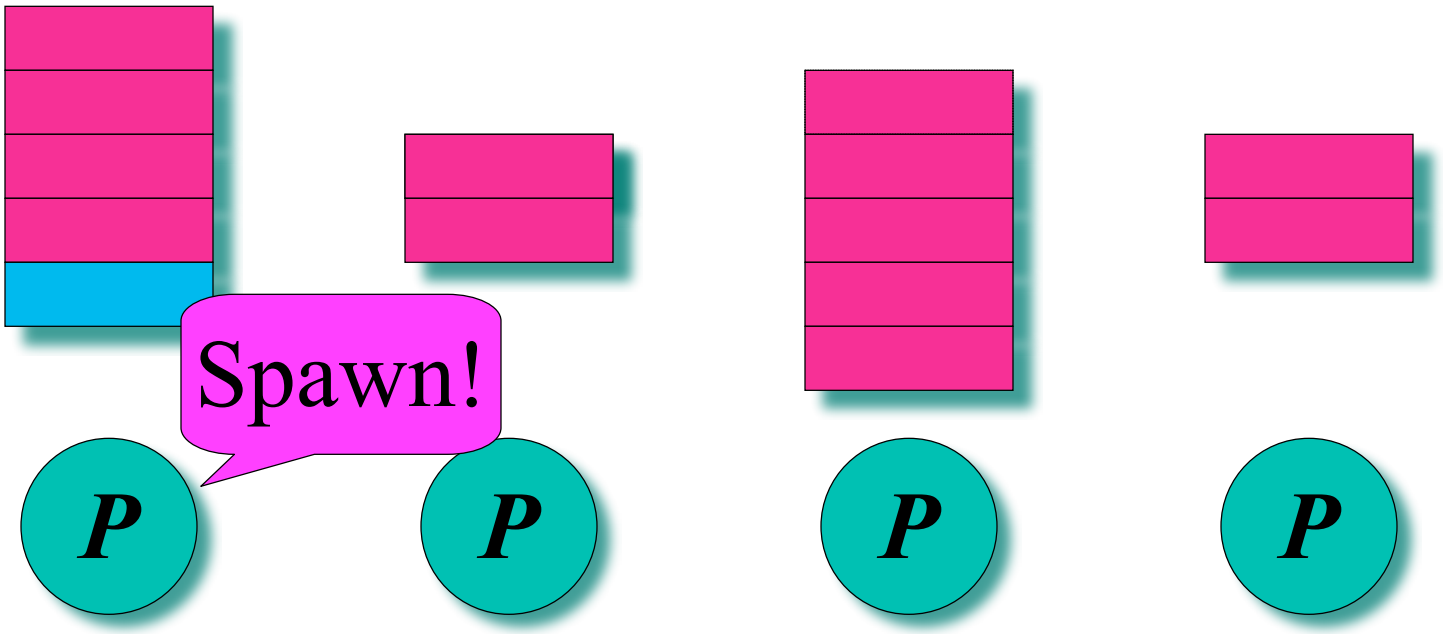
Parallelism: $T_1 / T_\infty = \Theta(\sqrt{n})$

Task Scheduling in Cilk

- **Alternative strategies**
  - —**work-sharing: thread scheduled to run in parallel at every spawn**
    - **benefit: maximizes parallelism**
    - **drawback: cost of setting up new threads is high → should be avoided**
  - —**work-stealing: processor looks for work when it becomes idle**
    - **lazy parallelism: put off work for parallel execution until necessary**
    - **benefits: executes with precisely as much parallelism as needed**
      minimizes the number of threads that must be set up
      runs with same efficiency as serial program on uniprocessor
- **Cilk uses work-stealing rather than work-sharing**

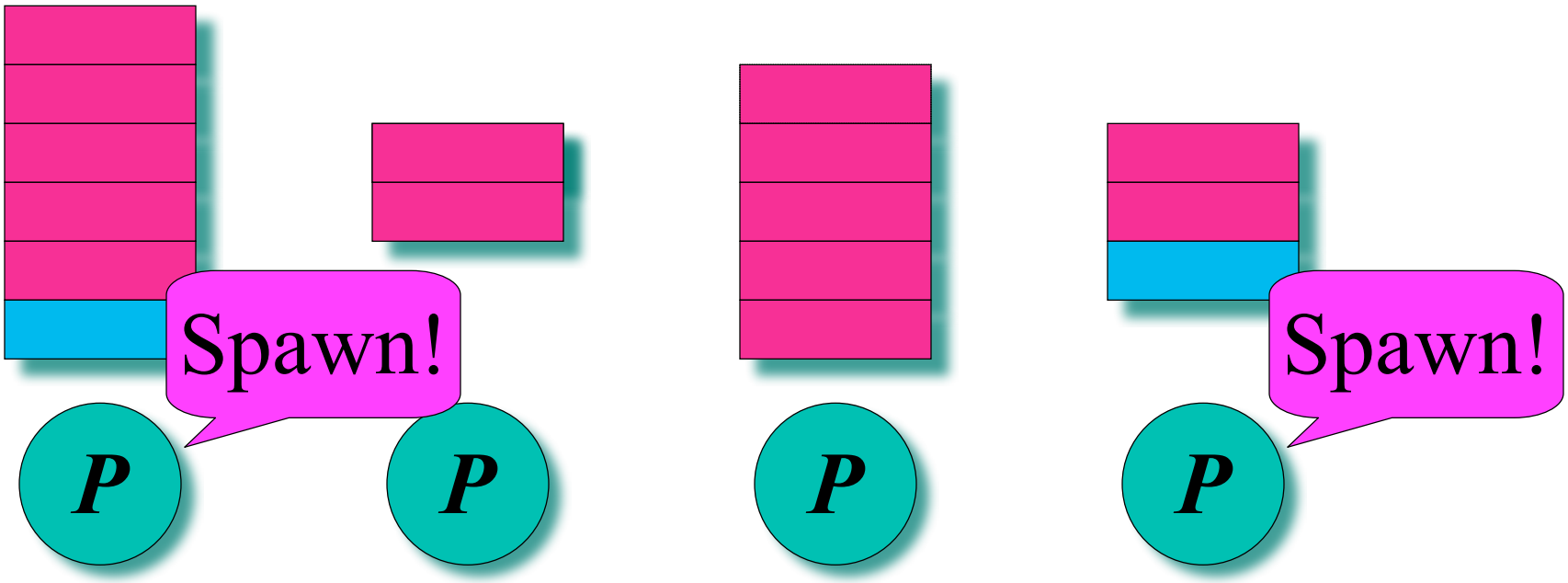
Cilk's Work-Stealing Scheduler

Each processor maintains a ***work deque*** of ready threads, and it manipulates the bottom of the deque like a stack.



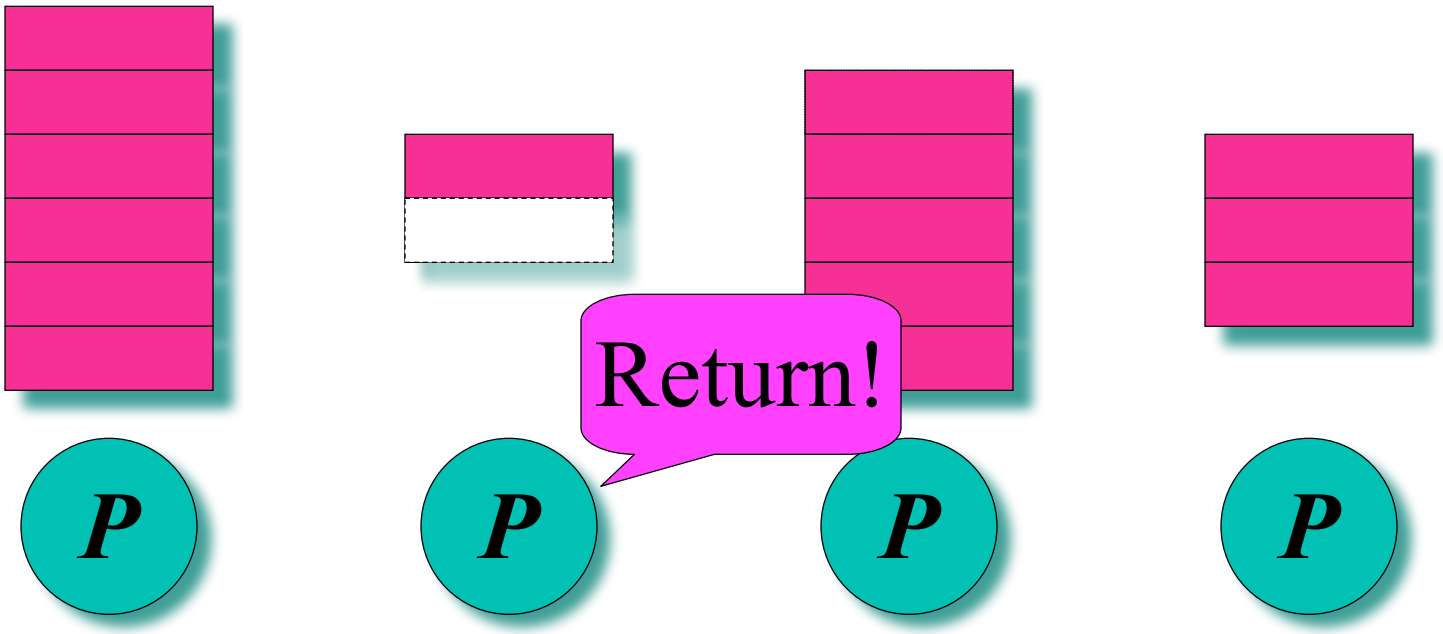
Cilk's Work-Stealing Scheduler

Each processor maintains a ***work deque*** of ready threads, and it manipulates the bottom of the deque like a stack.



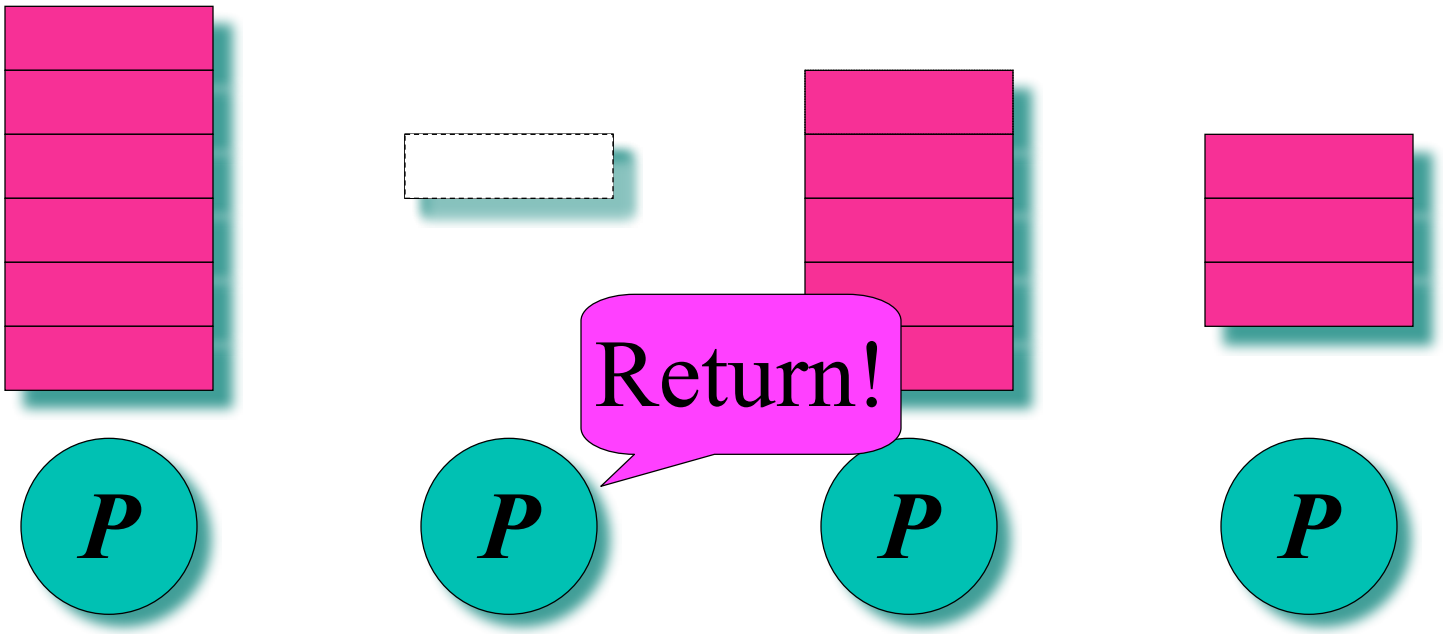
Cilk's Work-Stealing Scheduler

Each processor maintains a ***work deque*** of ready threads, and it manipulates the bottom of the deque like a stack.



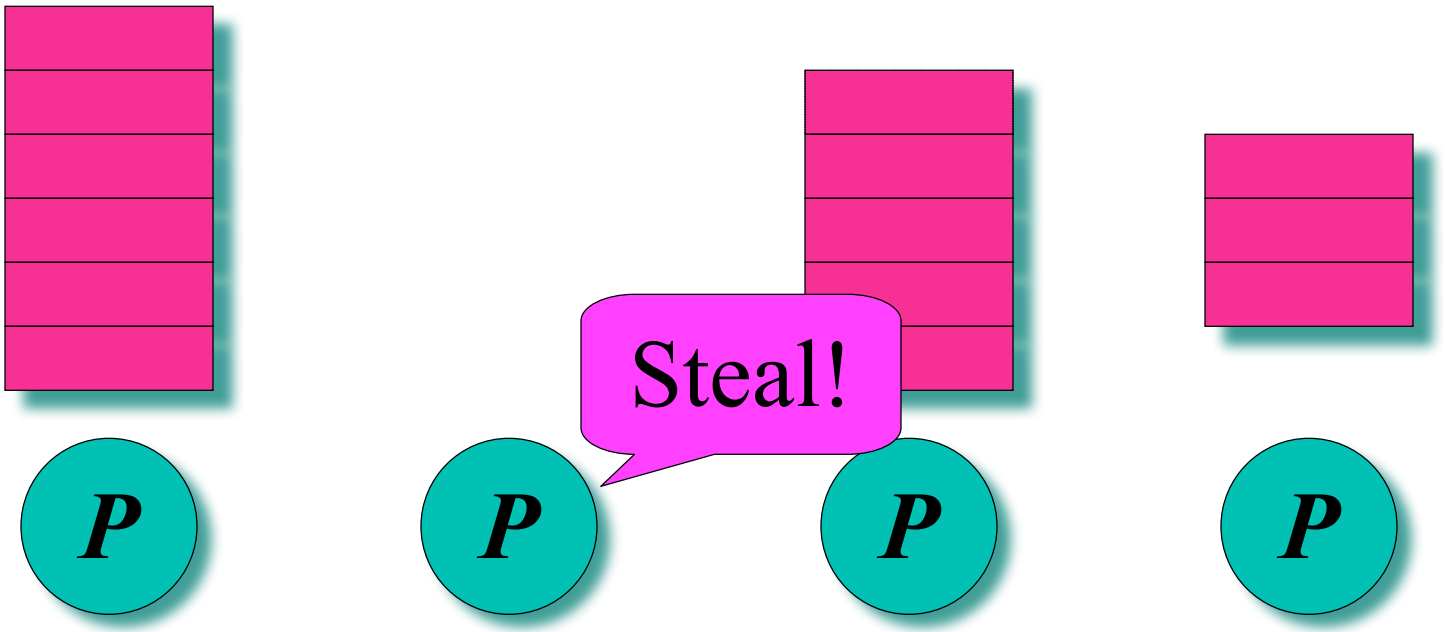
Cilk's Work-Stealing Scheduler

Each processor maintains a ***work deque*** of ready threads, and it manipulates the bottom of the deque like a stack.

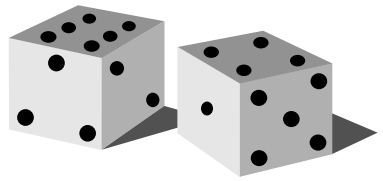


Cilk's Work-Stealing Scheduler

Each processor maintains a ***work deque*** of ready threads, and it manipulates the bottom of the deque like a stack.

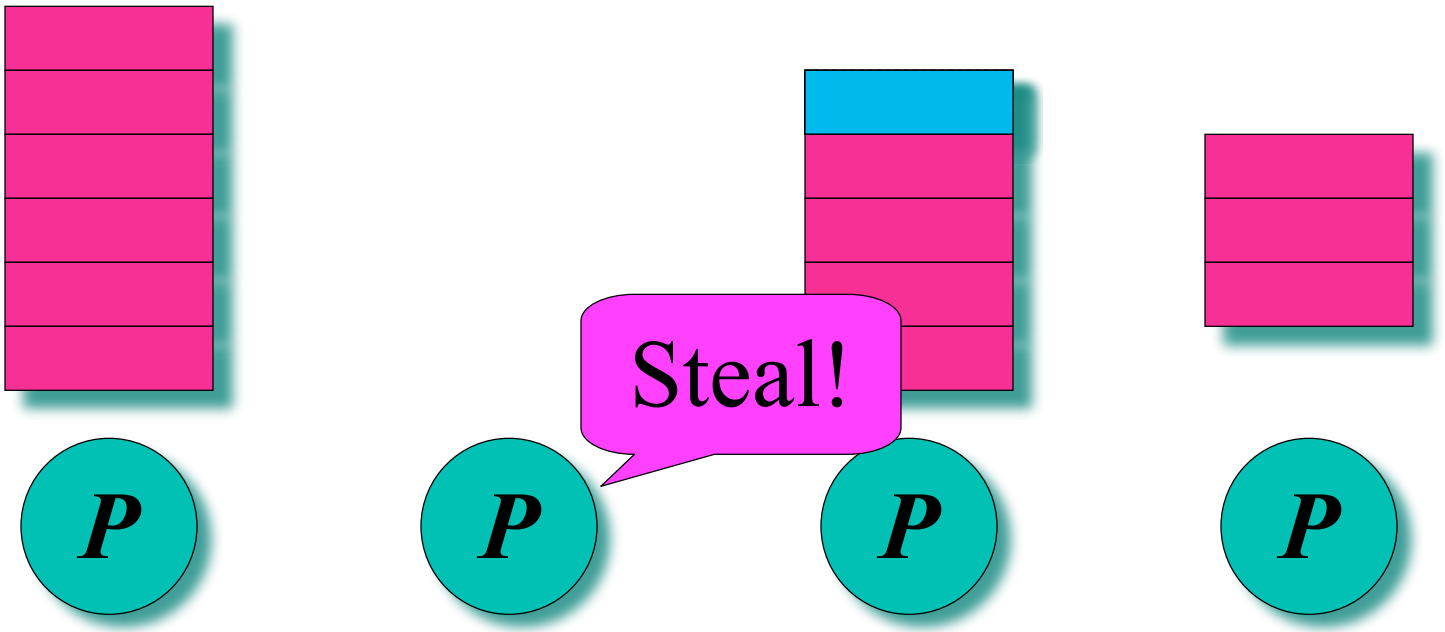


When a processor runs out of work, it ***steals*** a thread from the top of a ***random*** victim's deque.

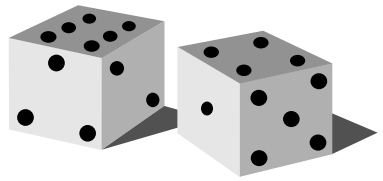


Cilk's Work-Stealing Scheduler

Each processor maintains a ***work deque*** of ready threads, and it manipulates the bottom of the deque like a stack.

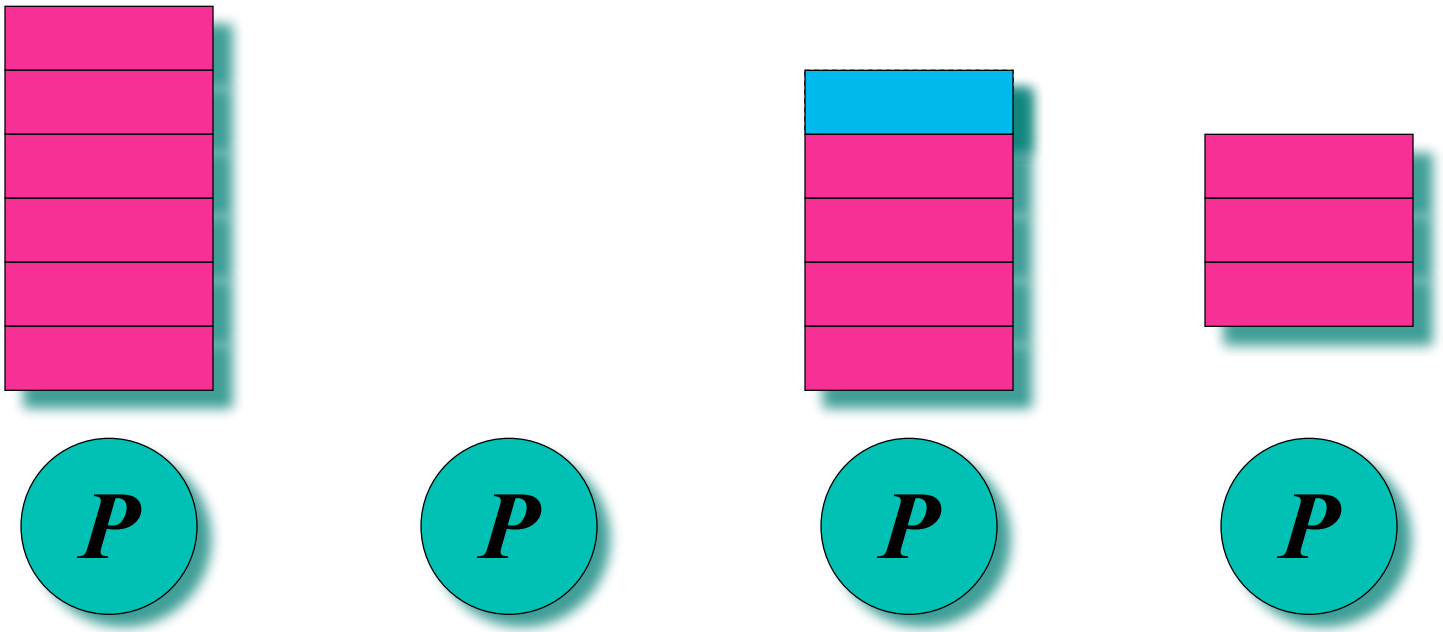


When a processor runs out of work, it ***steals*** a thread from the top of a ***random*** victim's deque.

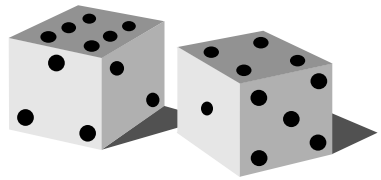


Cilk's Work-Stealing Scheduler

Each processor maintains a ***work deque*** of ready threads, and it manipulates the bottom of the deque like a stack.

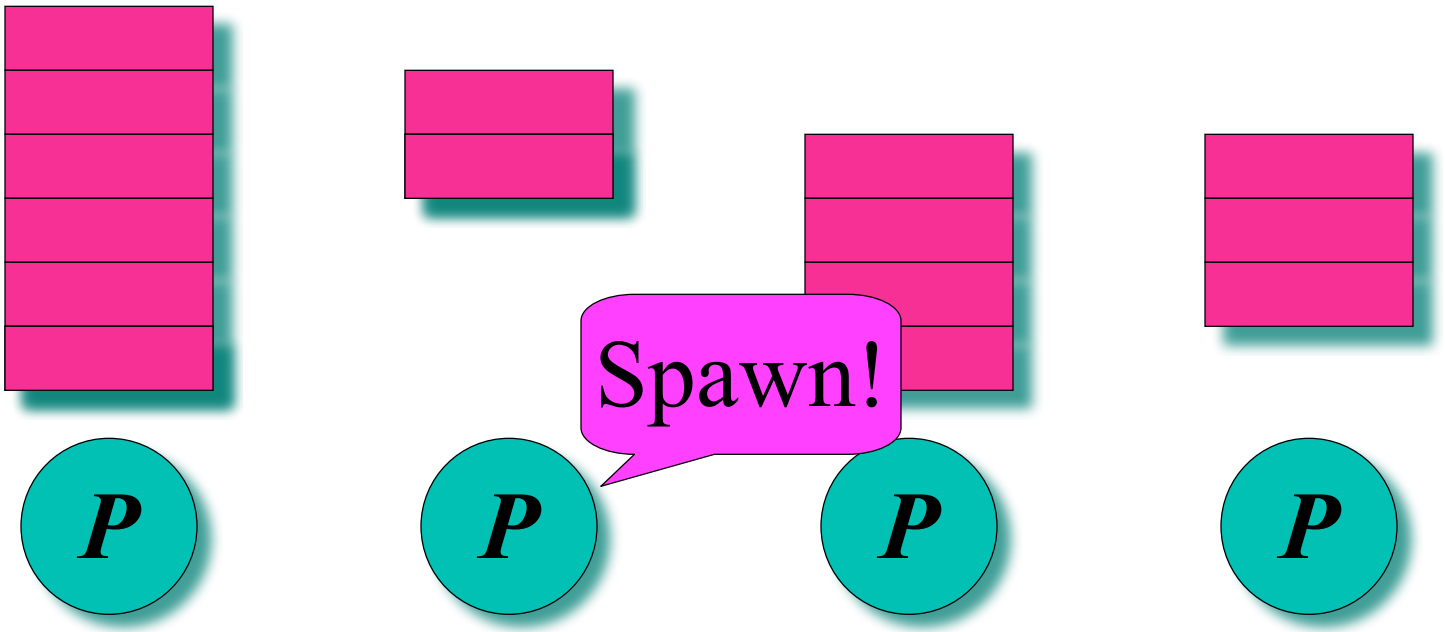


When a processor runs out of work, it ***steals*** a thread from the top of a ***random*** victim's deque.

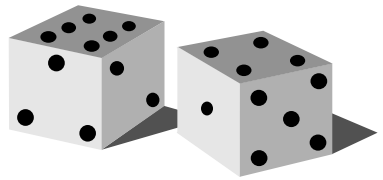


Cilk's Work-Stealing Scheduler

Each processor maintains a ***work deque*** of ready threads, and it manipulates the bottom of the deque like a stack.



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



Performance of Work-Stealing

Theorem: Cilk's work-stealing scheduler achieves an expected running time of

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

on P processors.

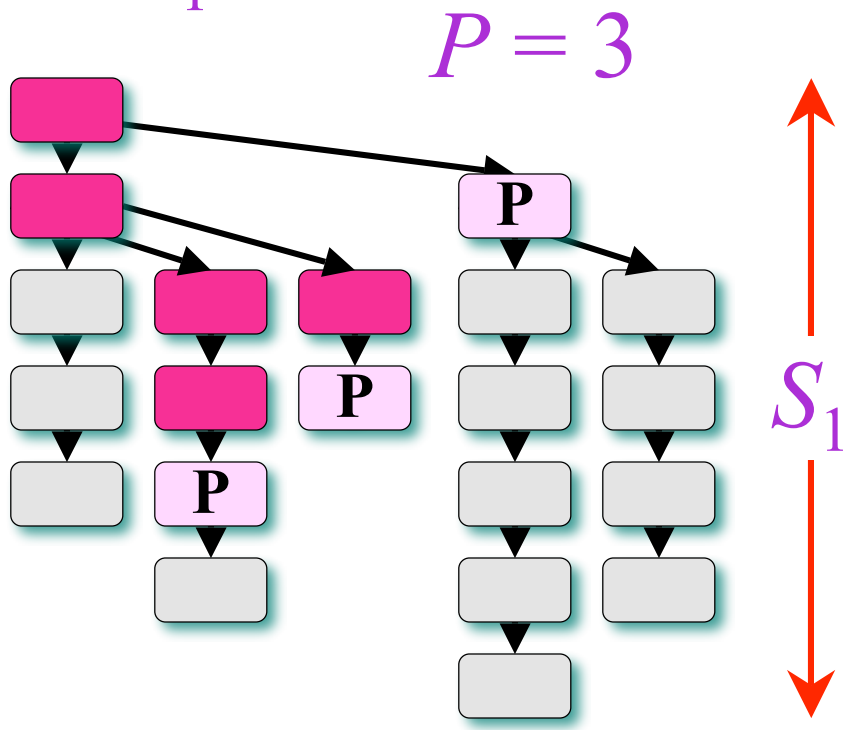
Pseudoproof. A processor is either ***working*** or ***stealing***. The total time all processors spend working is T_1. Each steal has a $1/P$ chance of reducing the span by 1. Thus, the expected cost of all steals is $O(P\,T_\infty)$. Since there are P processors, the expected time is

$$(T_1 + O(P\,T_\infty))/P = T_1/P + O(T_\infty)\ .\ \blacksquare$$

Space Bounds

Theorem. Let S_1 be the stack space required by a serial execution of a Cilk program. Then, the space required by a P-processor execution is at most $S_P <= PS_1$.

Proof (by induction). The work-stealing algorithm maintains the ***busy-leaves property***: *every extant procedure frame with no extant descendents has a processor working on it.* ■



Linguistic Implications

Code like the following executes properly without any risk of blowing out memory:

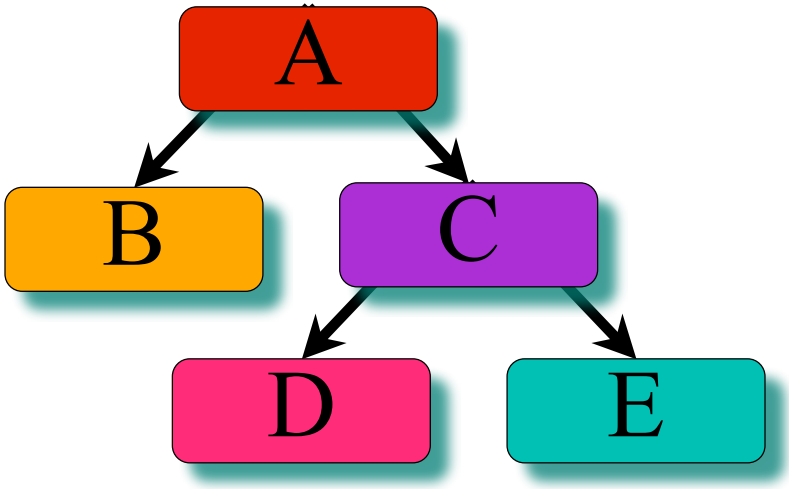
```
for (i=1; i<1000000000; i++) {
  spawn foo(i);
}
sync;
```

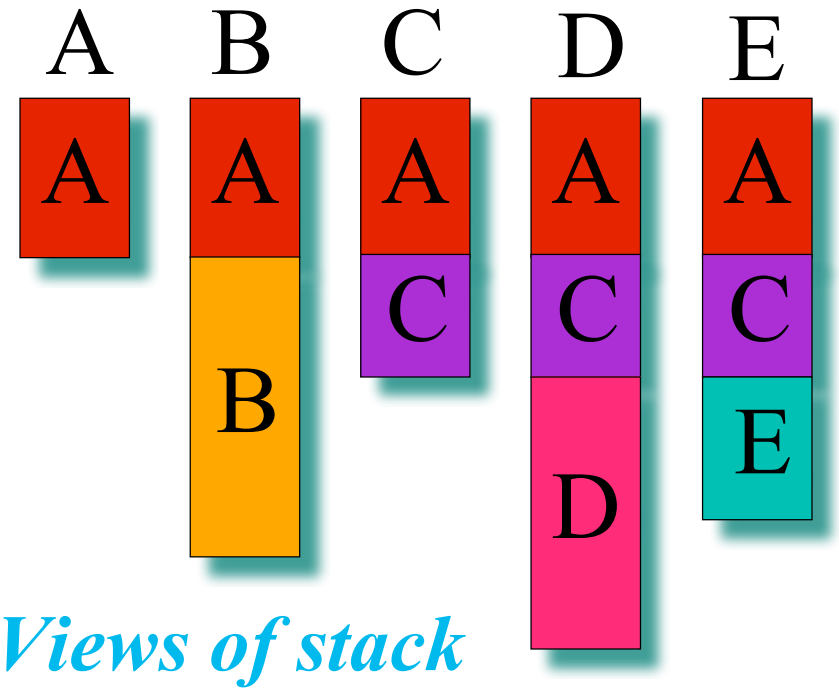
MORAL

Better to steal parents than children!

Cactus Stack

Cilk supports C's rule for pointers: A pointer to stack space can be passed from parent to child, but not from child to parent. (Cilk also supports **`malloc`**.)





Views of stack

Cilk's ***cactus stack*** supports several views in parallel.

Summary of Today's Lecture

- Thread Basics
- Introduction to Cilk

Reading List for Next Lecture (Jan 22nd)

- Pages 18 - 27 (Sections 2.6 - 2.9) and 37 - 42 (Sections 5.1 - 5.4) of Cilk Reference Manual
  - —http://supertech.csail.mit.edu/cilk/manual-5.4.6.pdf