CS528
Cilk

Slides are adopted from

http://supertech.csail.mit.edu/cilk/

Charles E. Leiserson

A Sahu

Dept of CSE, IIT Guwahati

Cilk

- Developed by **Leiserson at CSAIL, MIT**
  - **Chapter 27, Multithreaded Algorithm, Introduction to Algorithm, Coreman, Leiserson and Rivest**
- Initiated a startup: Cilk Plus
  - Added Cilk_for Keyword, Cilk Reduction features
  - Acquired by Intel, Intel uses Cilk Scheduler
- Addition of 6 keywords to standard C
  - Easy to install in linux system
  - With gcc and pthread

Cilk-Installation and testing

- Available @ Course Website
  http://jatinga.iitg.ernet.in/~asahu/cs528/
  - Resources for Cilk: cilktool cilk-5.4.6.tar.gz,
  - How to Install Cilk HowtoInstallCilk.txt
  - Test program and Makefile for cilk cilkmatmultest and
  - Cilk Mannual And Resources at Cilk@MIT, PowerPoint: lecture-1.ppt, lecture-2.ppt, lecture-3.ppt

Cilk

- In 2008, ACM SIGPLAN awarded **Best influential paper of Decade**
  - **The Implementation of the Cilk-5 Multithreaded Language**, PLDI 1998
- PLDI 2008 Best paper Award
  - Reducers and Other Cilk++ Hyperobjects , PLDI 2008

Cilk : Biggest principle

- Programmer should be responsible for
  - Exposing the parallelism,
  - Identifying elements that can safely be executed in parallel
- Work of run-time environment (scheduler) to
  - Decide during execution how to actually divide the work between processors
- Work Stealing Scheduler
  - Proved to be good scheduler
  - Now also in GCC, Intel CC, **Intel acquire Cilk++**

Fibonacci

C elision

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

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);
  }
}
```

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

Basic Cilk Keywords

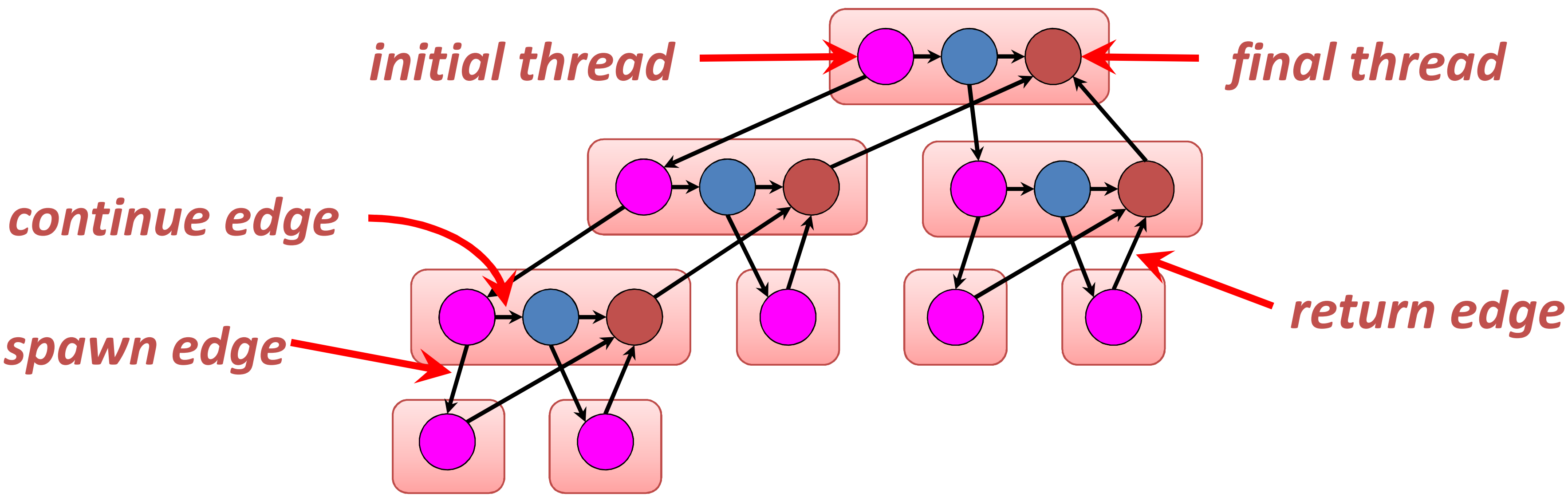
```
cilk int fib (int n) {
  if (n<2) return (n);
  else {
    int x,y;
    x = spawn fib(n-1);
    y = spawn fib(n-2);
    sync;
    return (x+y);
  }
}
```

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.

Multithreaded Computation

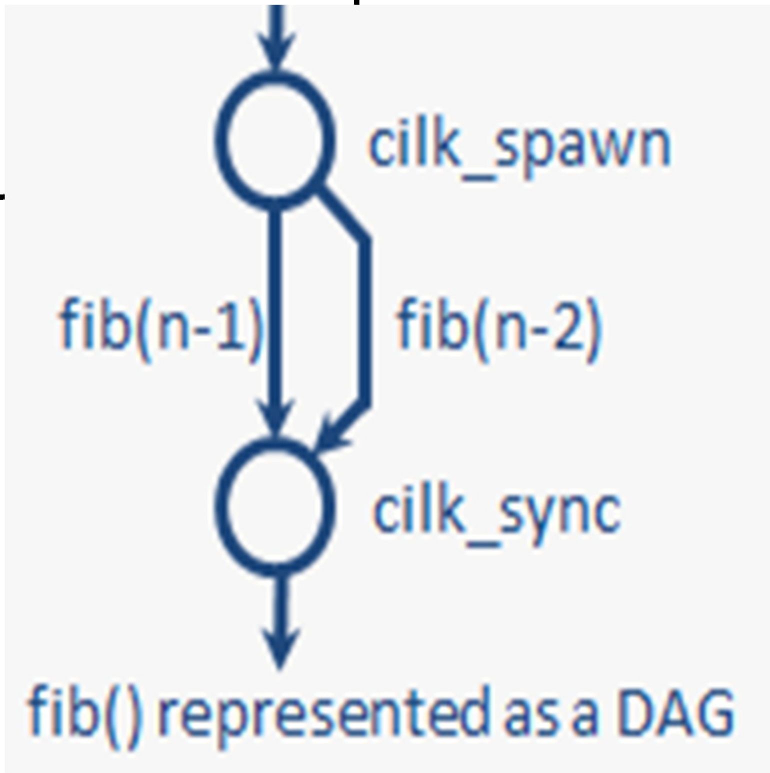


- The dag $G = (V, E)$ represents a parallel instruction stream.
- Each vertex v 2 V represents a ***(Cilk) thread***: a maximal sequence of instructions not containing parallel control (**`spawn`**, **`sync`**, **`return`**).
- Every edge e 2 E is either a ***spawn*** edge, a ***return*** edge, or a ***continue*** edge.

Fib: Cilk++ Version

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

Not available in Cilk



For loop in Cilk++

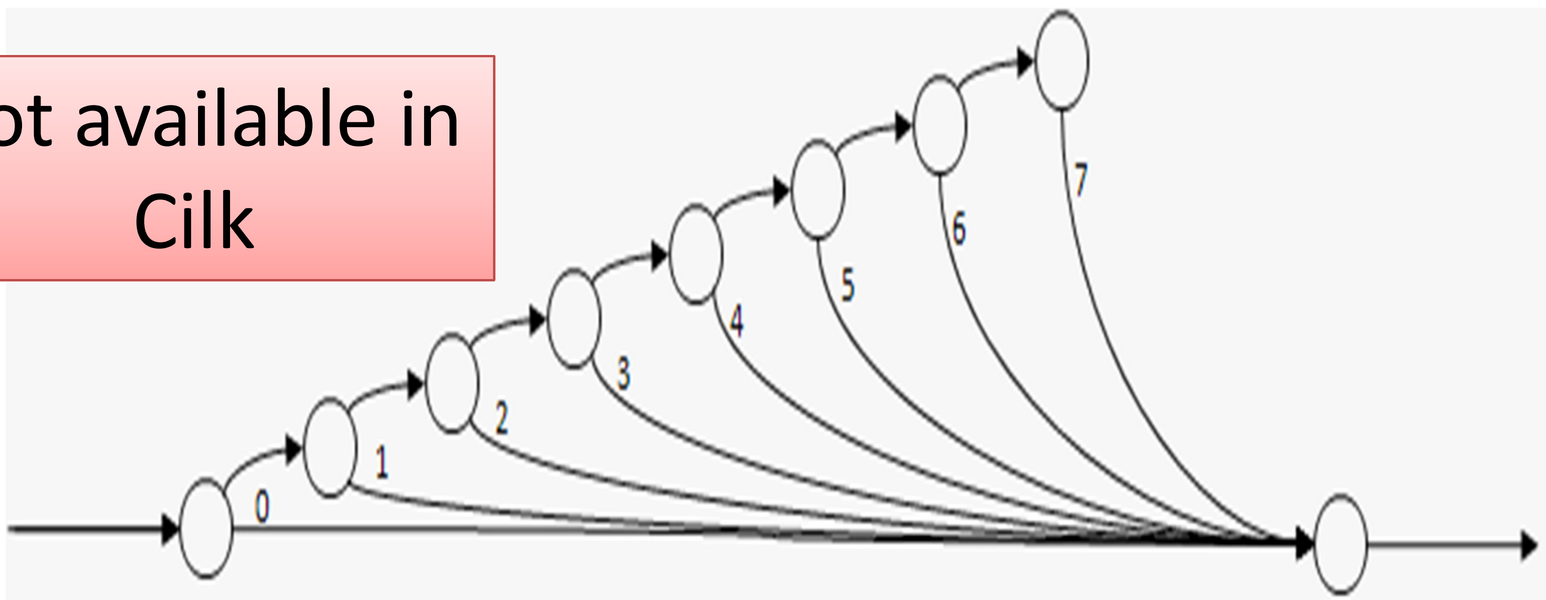
```
for (int i = 0; i < 8; ++i)
    do_work(i);
```

Serial

```
for (int i = 0; i < 8; ++i)
    cilk_spawn do_work(i);
cilk_sync;
```

Parallel

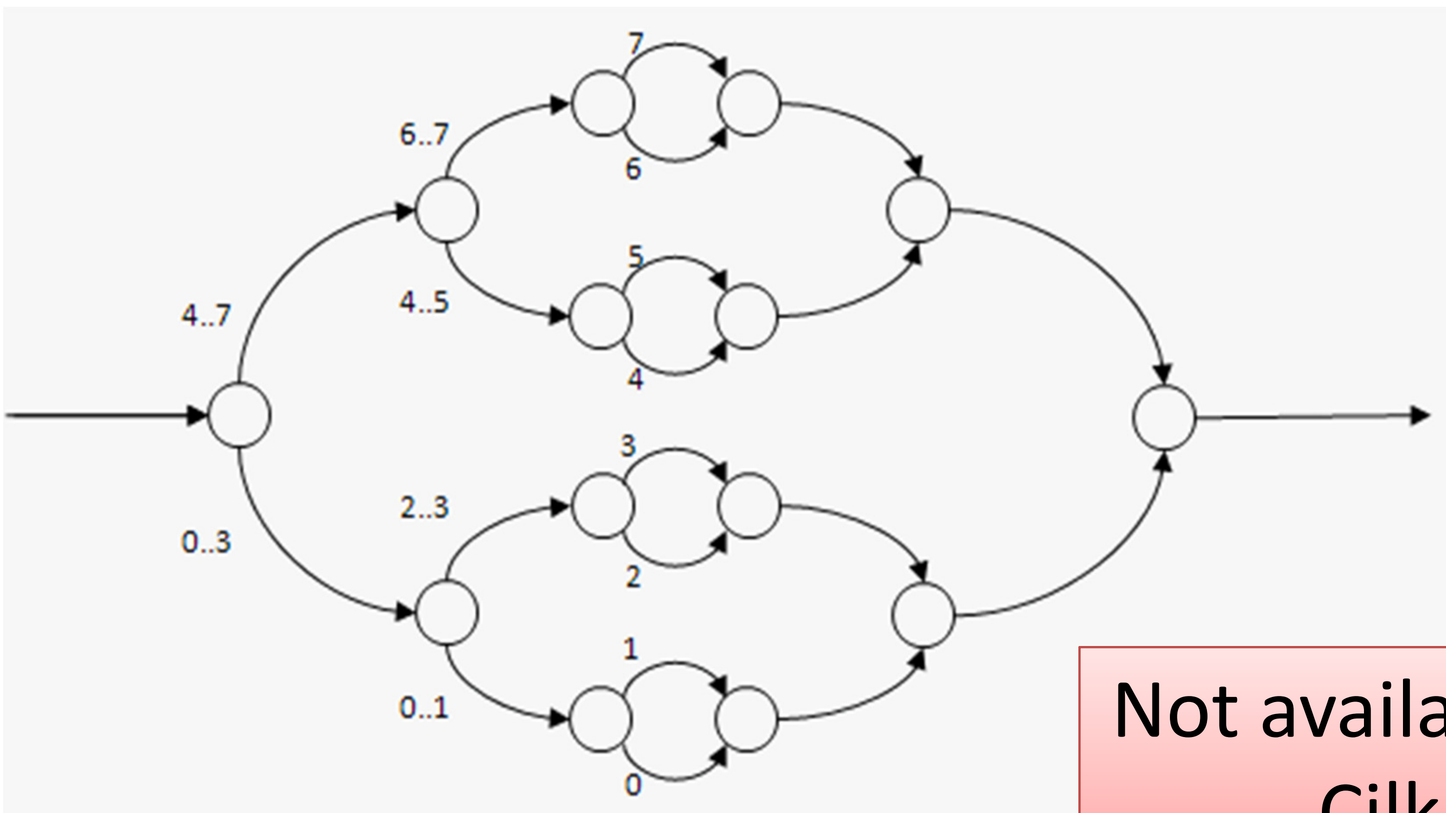
Not available in Cilk



Loop_for in Cilk++

```
cilk_for (int i=0;i<8;++i){
     do_work(i);
}// No sync required; auto sync
```

Parallel



Not available in Cilk

Scheduler: Load balancing

- Centralized load balancing
  - Master sever and many worker
  - **Master assign work/task to worker**
- Distributed load balancing
  - **All are peers worker, they collaborate among them self and balance the load**
  - Receiver initiated (**Example Cilk RTS**)
    - Free/lightly loaded worker ask for task
  - Sender initiated
    - Highly loaded worker task transfer task to free worker

Cilk Run Time Scheduler

- Distributed load balancing
  - Receiver initiated
- Work stealing : Free processor steal a task of busy processor
- When ever a process spawns a new process,
  - This processor starts executing the spawned one
  - Parent goes to waiting/suspend mode
  - Parent can be transferred to other processor

Cilk Run Time Scheduler

- Distributed load balancing
  - Receiver initiated
- Work stealing : Free processor steal a task of busy processor
- When ever a process spawns a new process,
  - This processor starts executing the spawned one
  - Parent goes to waiting/suspend mode
  - Parent can be transferred to other processor

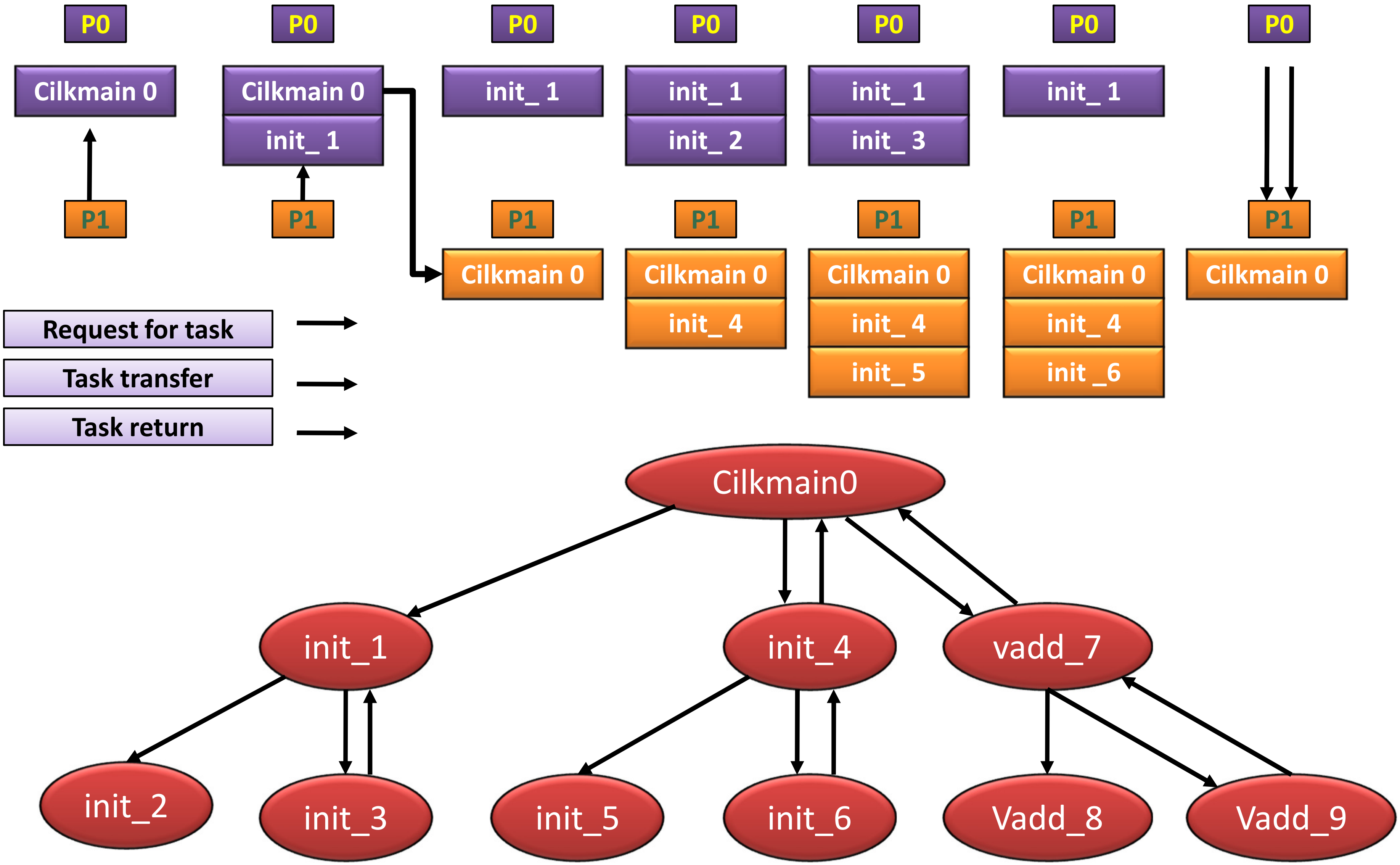
Work stealing

- Work stealing algorithm is receiver initiated algorithm
- Technique commonly used for load balancing
- Thief processor (Idle processor)
  - Steal work from other processor
  - Victim is selected randomly
- Victim processor (From a set of busy processor)
  - Work is stolen from these processor

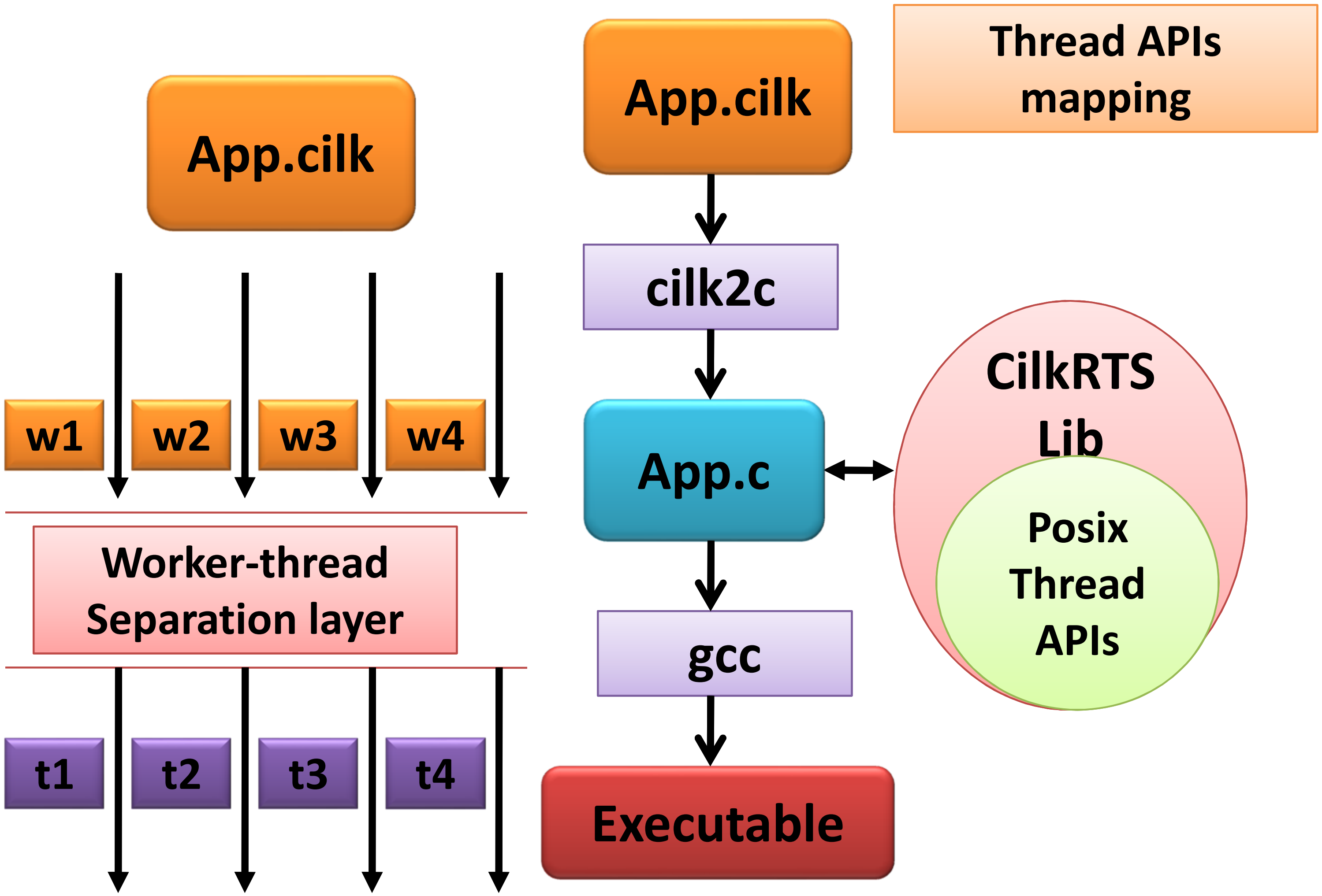
Work stealing

- Optimal algorithm for load balancing
  - If select victim randomly algorithm is Optimal **Proved**
- Basic assumption in work stealing
  - All the memory access are take same time
  - **UMA (Uniform Memory Access): shared memory**
  - Can be feasible iff
    - Task transfer time is same for all pair of processors
    - Communication bandwidth is same for all pair of processors

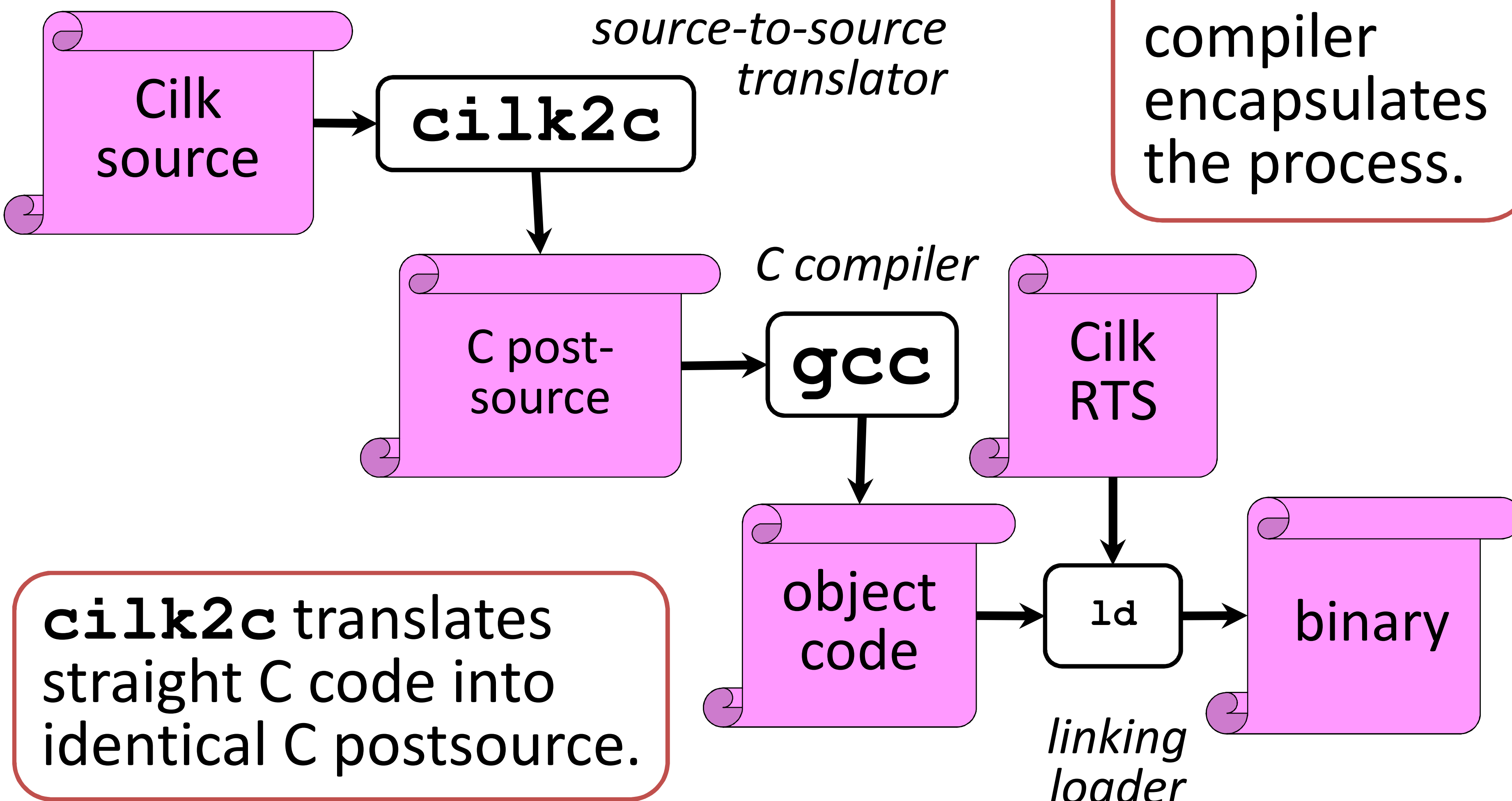
Cilk Run Time Scheduler



Cilk: flow


App.cilk
w1
w2
w3
w4
Worker-thread Separation layer
t1
t2
t3
t4
App.cilk
Thread APIs mapping
cilk2c
CilkRTS Lib
Posix Thread APIs
App.c
gcc
Executable

Compiling Cilk Program



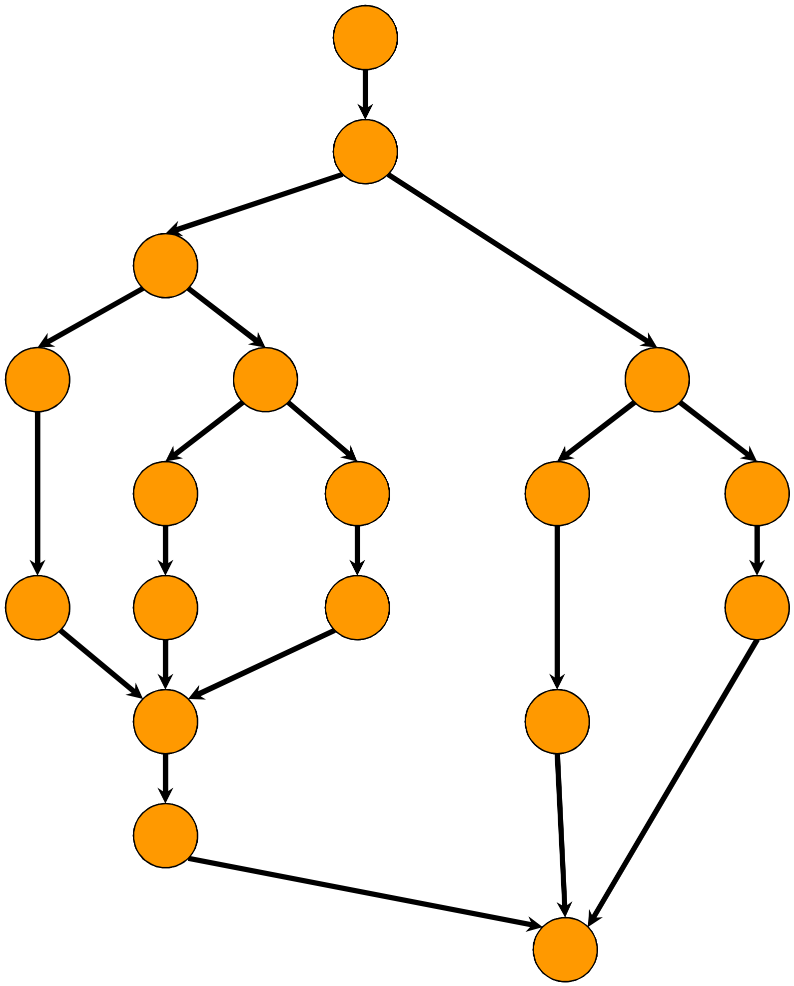
The **`cilkc`** compiler encapsulates the process.

`cilk2c` translates straight C code into identical C postsource.

```
$ cilkc fib.cilk –o fib
$ ./fib  -proc 4  5  // run using  4 threads
```

Algorithmic Complexity Measures

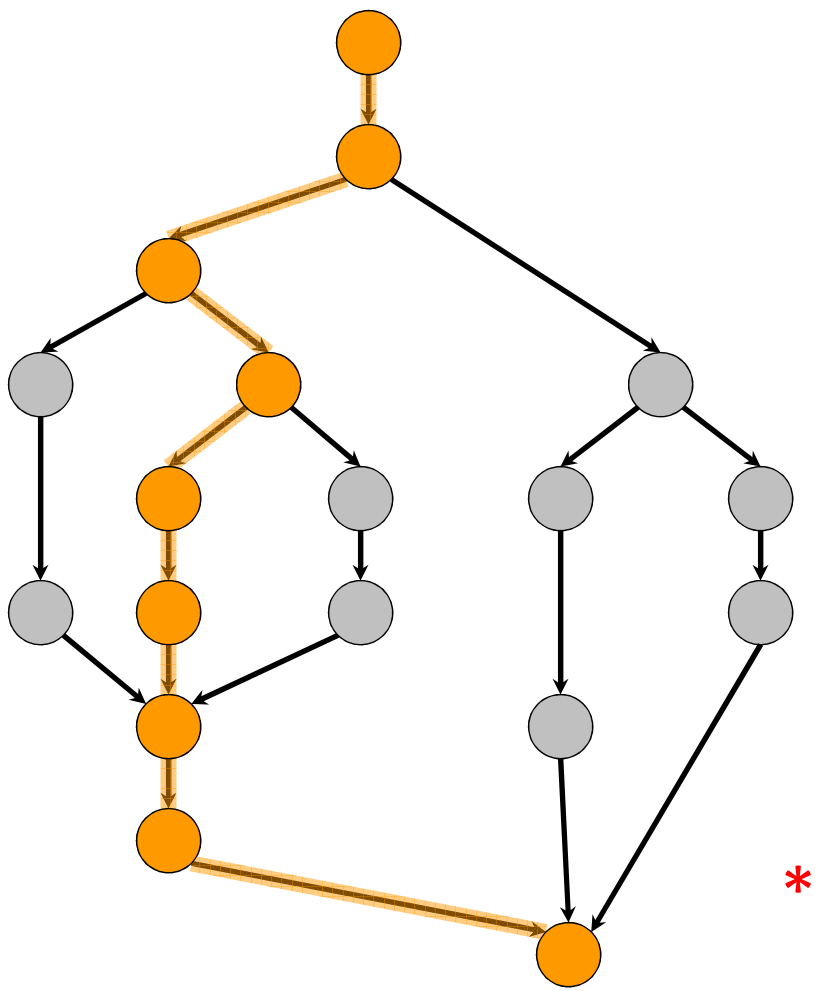
T_P = execution time on P processors



T_1 = ***work***

Algorithmic Complexity Measures

T_P = execution time on P processors



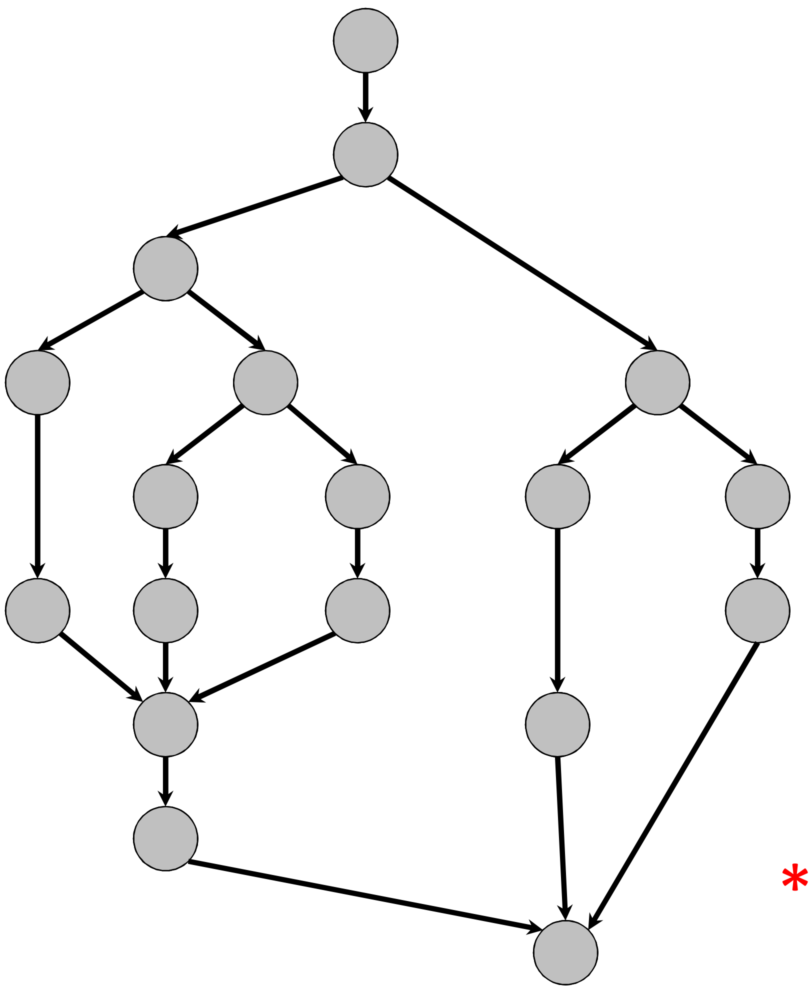
T_1 = ***work***

T_∞ = ***span****

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

Algorithmic Complexity Measures

T_P = execution time on P processors



T_1 = ***work***

T_∞ = ***span****

LOWER BOUNDS

- $T_P \geq T_1/P$
- $T_P \geq T_\infty$

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

Speedup

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

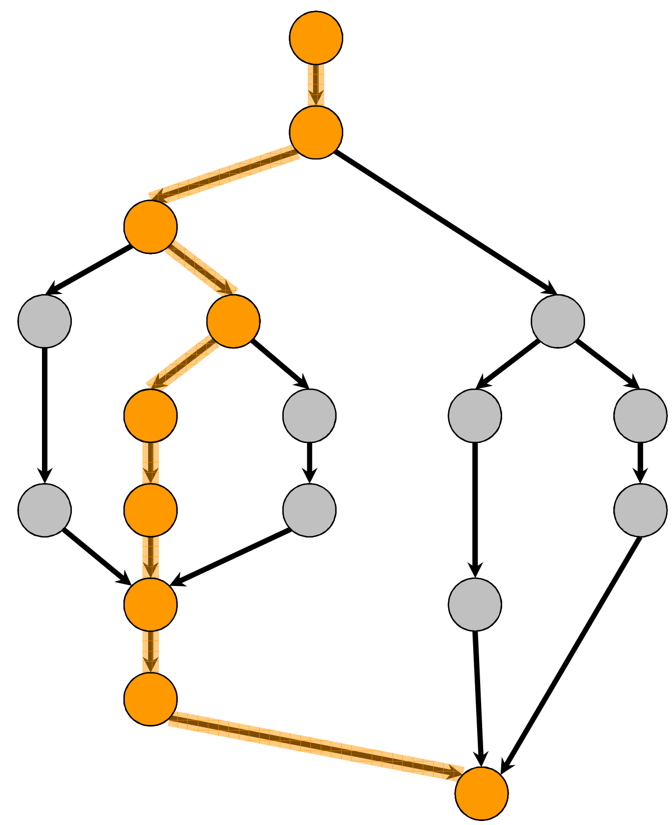
If $T_1/T_P = \Theta(P) < P$, we have ***linear speedup***;
$= P$, we have ***perfect linear speedup***;
$> P$, we have ***superlinear speedup***,
which is not possible in our model, because of the lower bound $T_P \geq T_1/P$.

Parallelism

Because we have the lower bound $T_P \geq T_\infty$, the maximum possible speedup given T_1 and T_∞ is

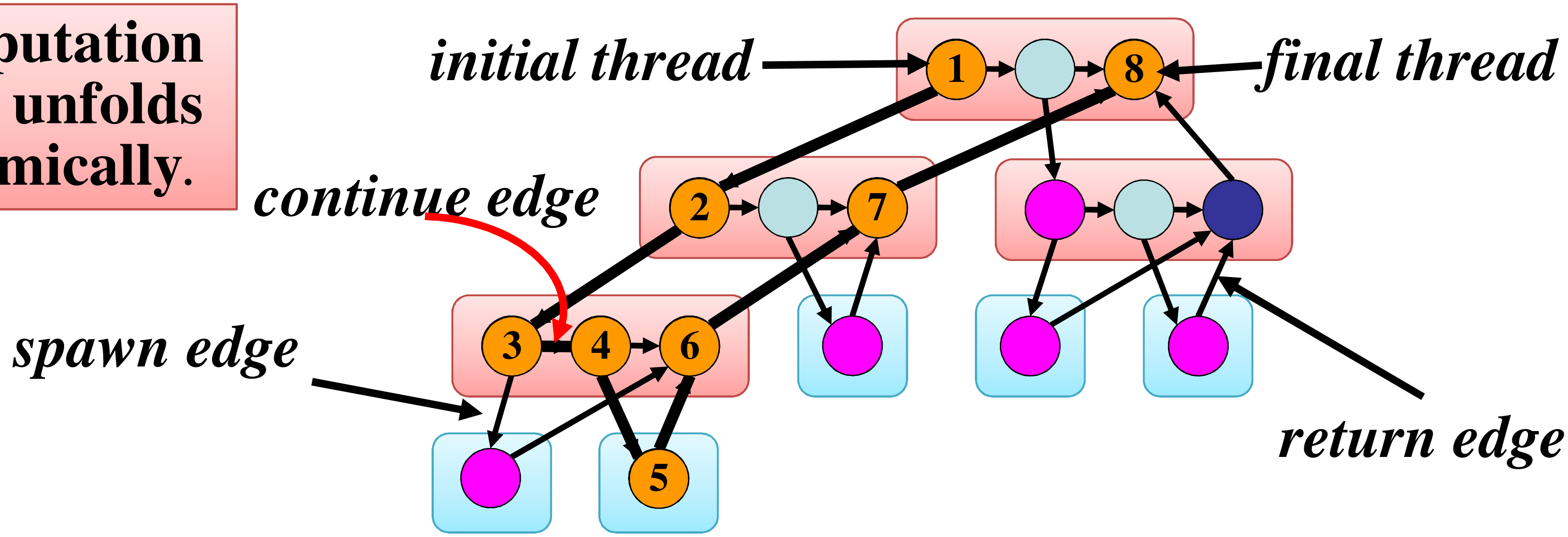
T_1 / T_∞ = ***parallelism***

= the average amount of work per step along the span.



CILK Example: Fib(4)

Computation DAG unfolds dynamically.



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

Work: $T_1 = 17$

Span: $T_\infty = 8$

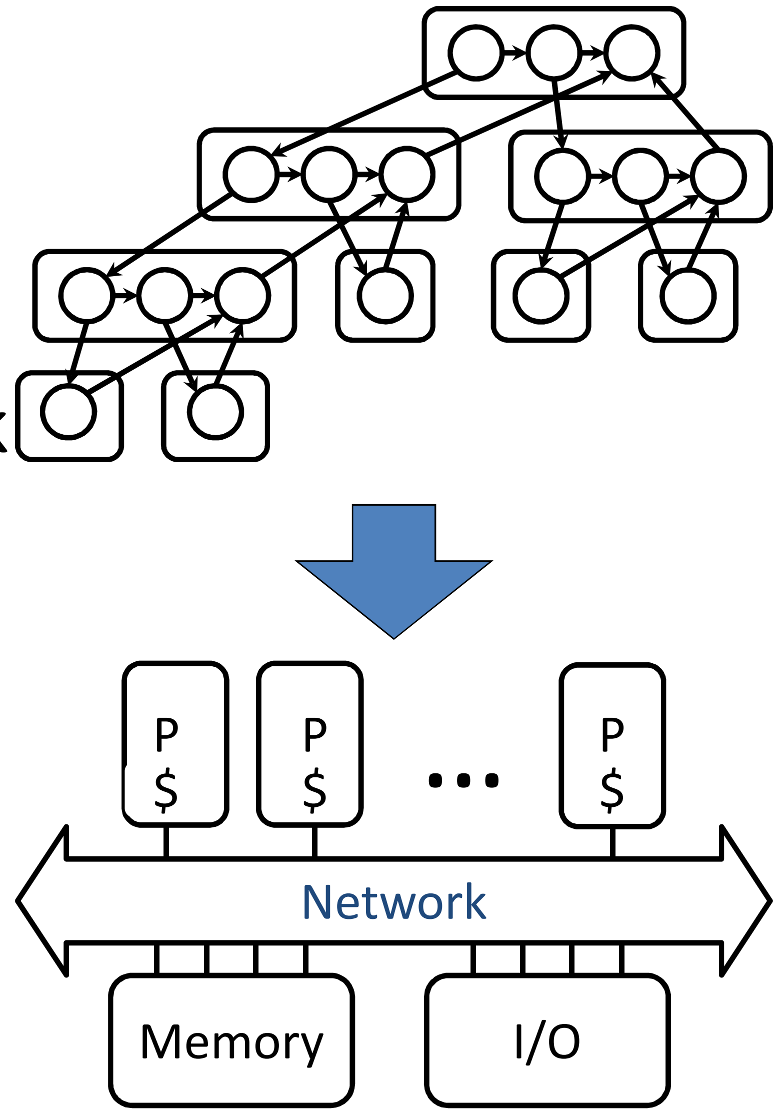
Parallelism: $T_1/T_\infty = 2.125$

Using many more than **2** processors makes little sense.

Ref1:The Cilk System for Parallel Multithreaded Computing, MIT Phd Thesis
Ref2:The Implementation of the Cilk-5 Multithreaded Language, 1998 ACM SIGPLAN

Scheduling

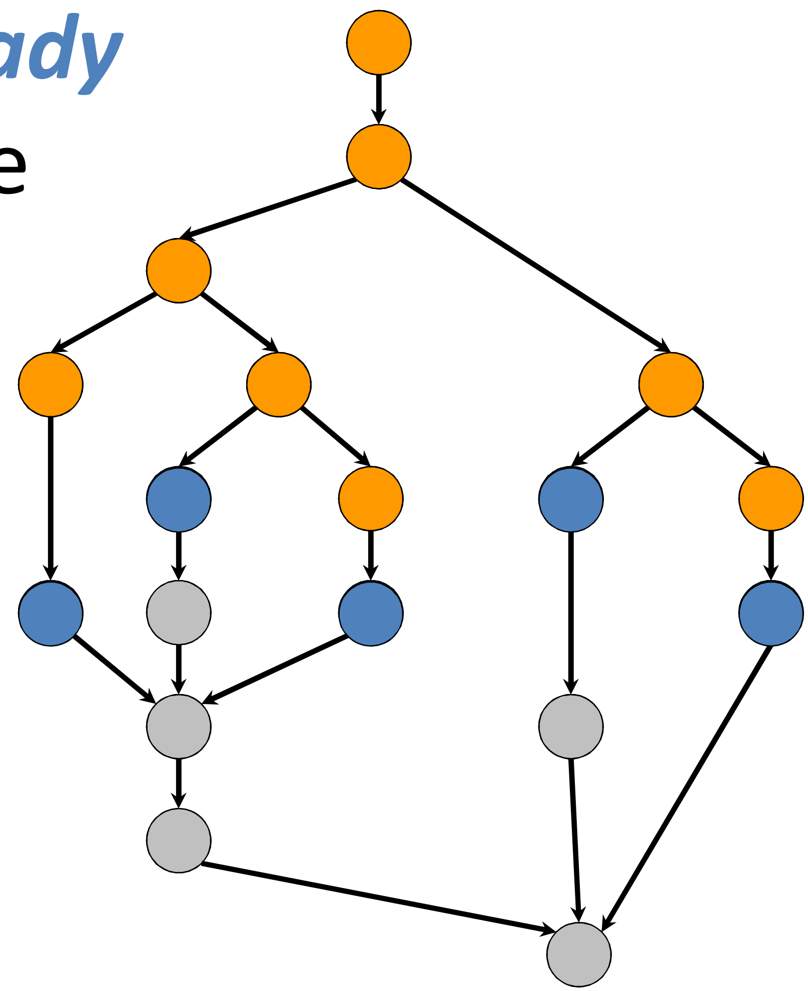
- Cilk allows the programmer to express *potential* parallelism in an application.
- The Cilk ***scheduler*** maps Cilk threads onto processors dynamically at runtime.
- Since ***on-line*** schedulers are complicated, we'll illustrate the ideas with an ***off-line*** scheduler.



Greedy Scheduling

IDEA: Do as much as possible on every step.

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



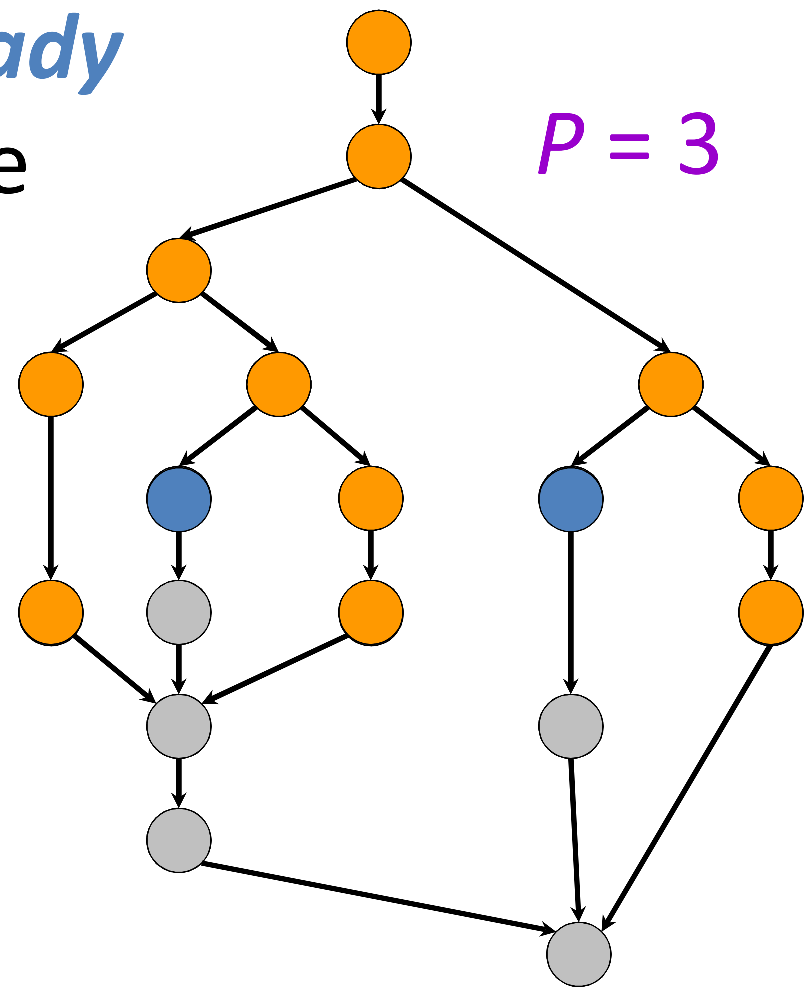
Greedy Scheduling

IDEA: Do as much as possible on every step.

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

Complete step

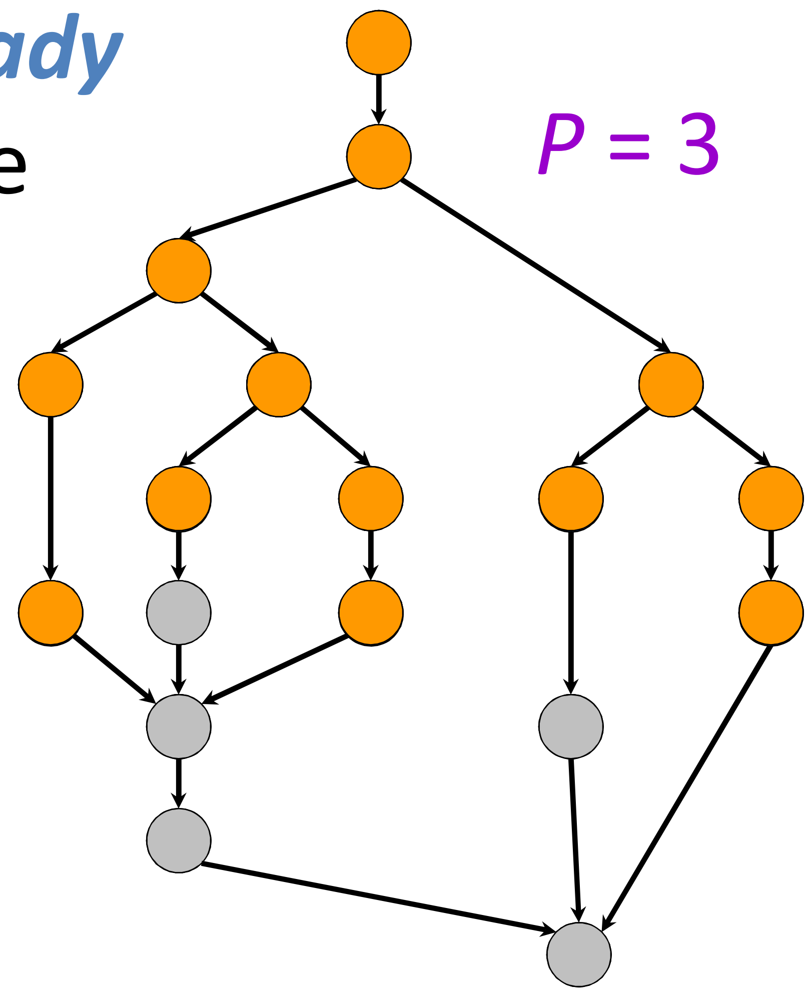
- $\geq P$ threads ready.
- Run any P.



Greedy Scheduling

IDEA: Do as much as possible on every step.

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



Complete step

- $\geq P$ threads ready.
- Run any P.

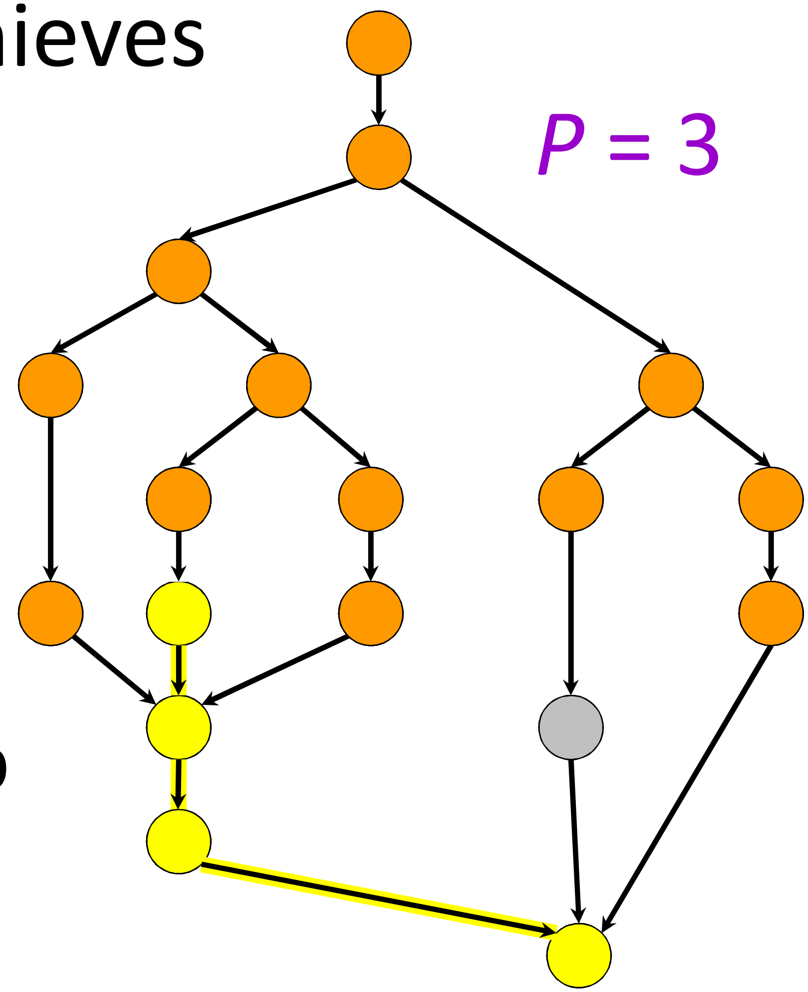
Incomplete step

- $< P$ threads 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 $\le T_1/P$, since each complete step performs P work.
- \# incomplete steps $\le T_\infty$, since each incomplete step reduces the span of the unexecuted dag by 1. ■

Optimality of Greedy

Corollary. Any greedy scheduler achieves within a factor of 2 of optimal.

Proof. Let $T_P{}^*$ be the execution time produced by the optimal scheduler.

Since $T_P{}^* \geq \max\{T_1/P, T_\infty\}$ (lower bounds), we have

$$\begin{aligned} T_P &\leq T_1/P + T_\infty \\ &\leq 2\max\{T_1/P, T_\infty\} \\ &\leq 2T_P{}^* . \ \blacksquare \end{aligned}$$

Linear Speedup

Corollary. Any greedy scheduler achieves near-perfect linear speedup whenever $T_1/T_\infty \gg P$

Proof. Since $T_1/T_\infty \gg P$ ➔ $T_\infty \ll T_1/P$, the Greedy Scheduling Theorem gives us

$$T_P \leq T_1/P + T_\infty$$
$$\approx T_1/P\ .$$

Thus, the speedup is $T_1/T_P \approx P$. ■

Definition. The quantity $(T_1/\ T_\infty)/P$ is called the ***parallel slackness***.

Cilk Performance

- Cilk's "work-stealing" scheduler achieves
  - $T_P = T_1/P + O(T_\infty)$ expected time (provably);
  - $T_P \approx T_1/P + T_\infty$ time (empirically).
- Near-perfect linear speedup if $P \ll T_1/T_\infty$.
- Instrumentation in Cilk allows the user to determine accurate measures of T_1 and T_∞ .
- 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.