Parallel Runtimes: Cilk

Chris Rossbach & Calvin Lin

cs380p

Outline

- Background
- Cilk
    - DAG-based computation
    - Critical Path
    - Work-stealing
    - Continuation-passing

Review: Decomposition

Review: Decomposition

Domain v. Functional

Review: Decomposition

Domain v. Functional

Domain Decomposition

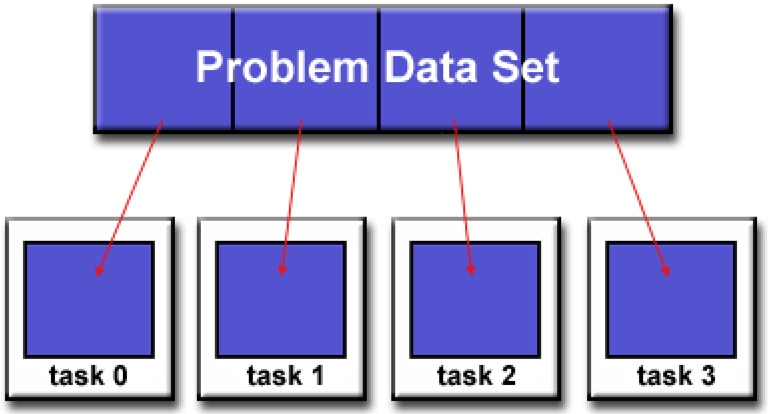
- a.k.a. Data Parallel
- Input domain
- Output Domain

Review: Decomposition

Domain v. Functional

Domain Decomposition

- a.k.a. Data Parallel
- Input domain
- Output Domain



Review: Decomposition

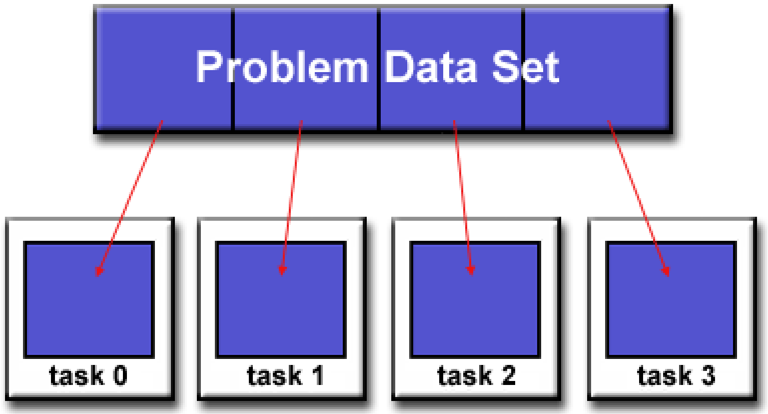
Domain v. Functional

Domain Decomposition

- a.k.a. Data Parallel
- Input domain
- Output Domain

Functional Decomposition

- a.k.a. Task Parallel
- Independent Tasks
- Pipelining



Review: Decomposition

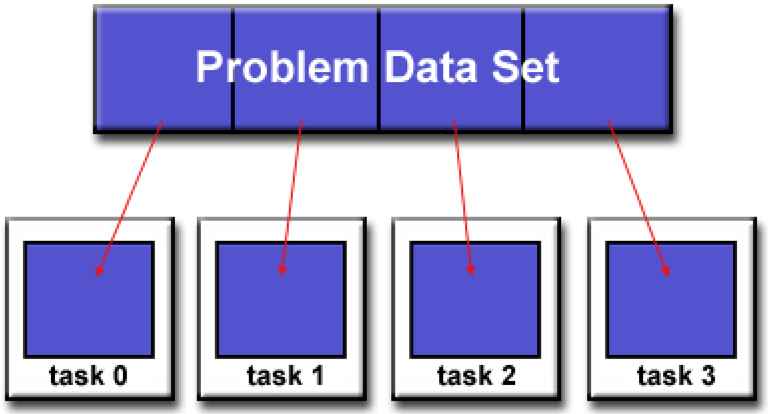
Domain v. Functional

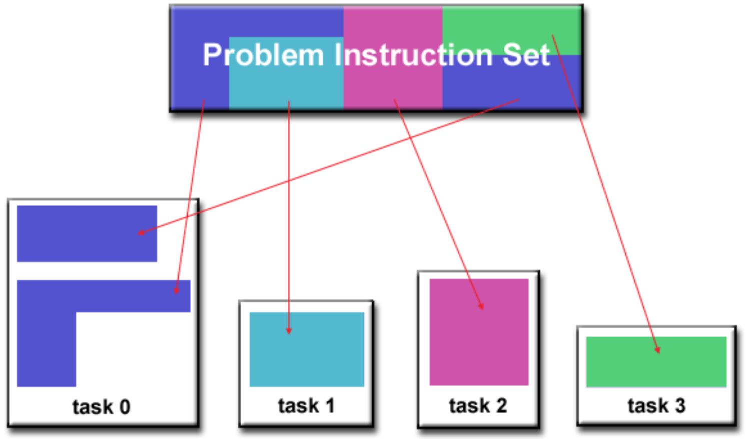
Domain Decomposition

- a.k.a. Data Parallel
- Input domain
- Output Domain

Functional Decomposition

- a.k.a. Task Parallel
- Independent Tasks
- Pipelining





Review: Decomposition

Domain v. Functional

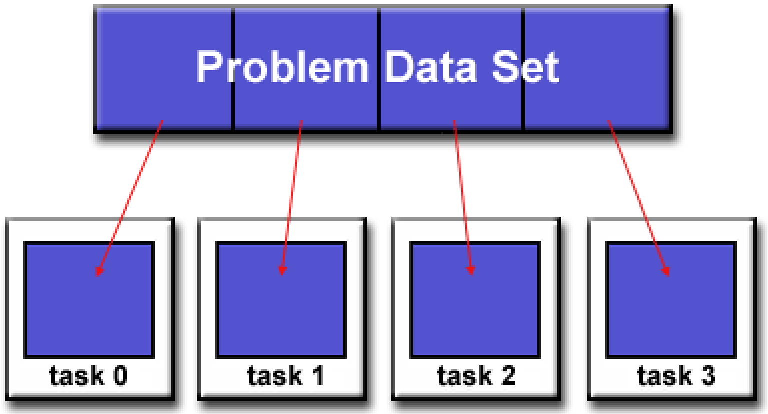
Domain Decomposition

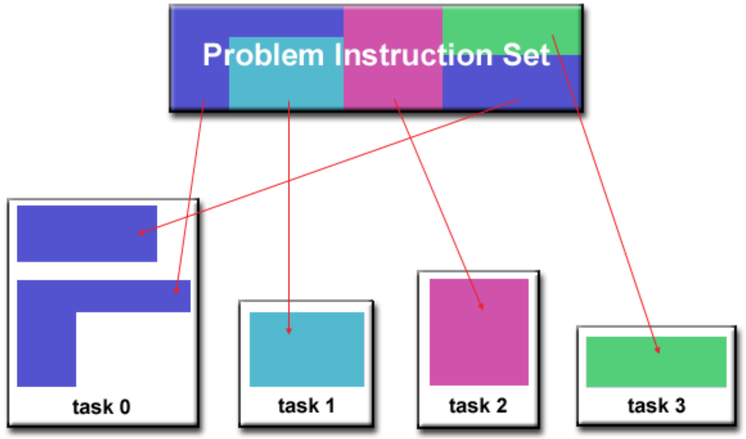
- a.k.a. Data Parallel
- Input domain
- Output Domain

Functional Decomposition

- a.k.a. Task Parallel
- Independent Tasks
- Pipelining

Real Problems: mix/nest





Review: Decomposition

Domain v. Functional

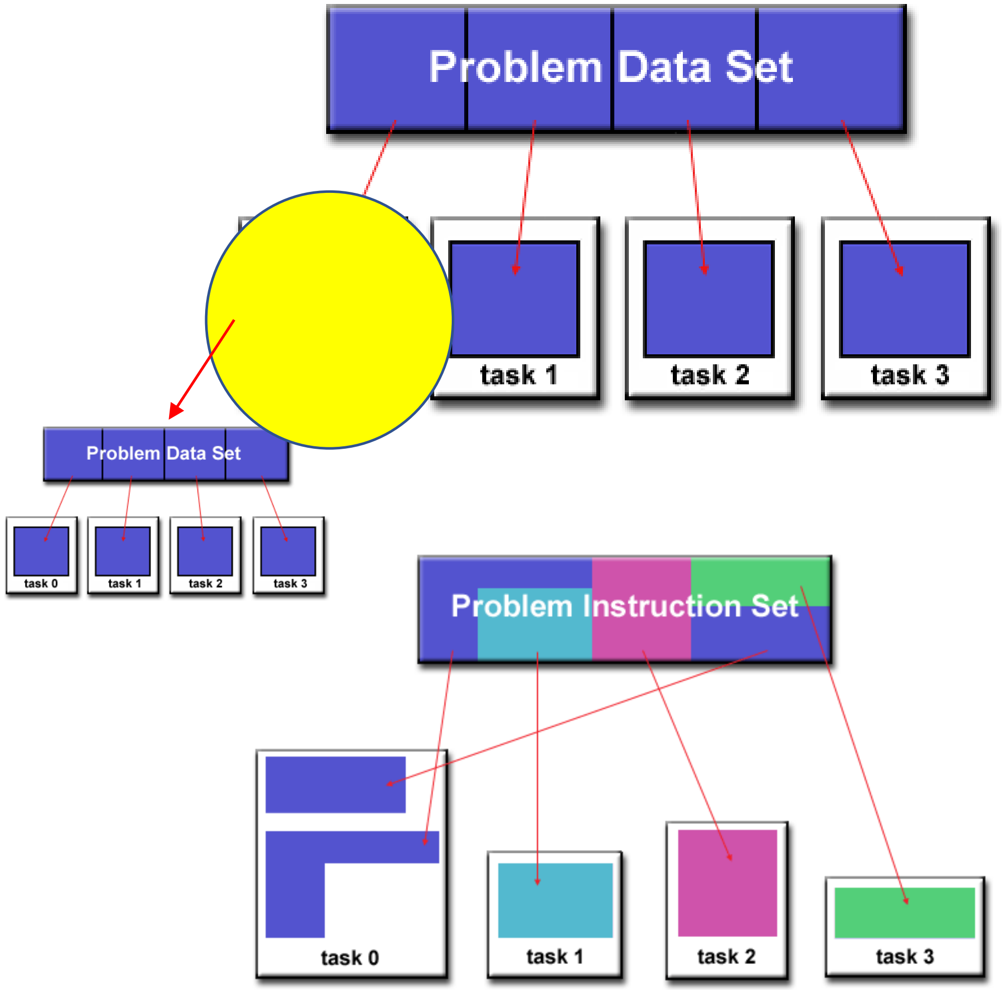
Domain Decomposition

- a.k.a. Data Parallel
- Input domain
- Output Domain

Functional Decomposition

- a.k.a. Task Parallel
- Independent Tasks
- Pipelining

Real Problems: mix/nest



Review: Decomposition

Domain v. Functional

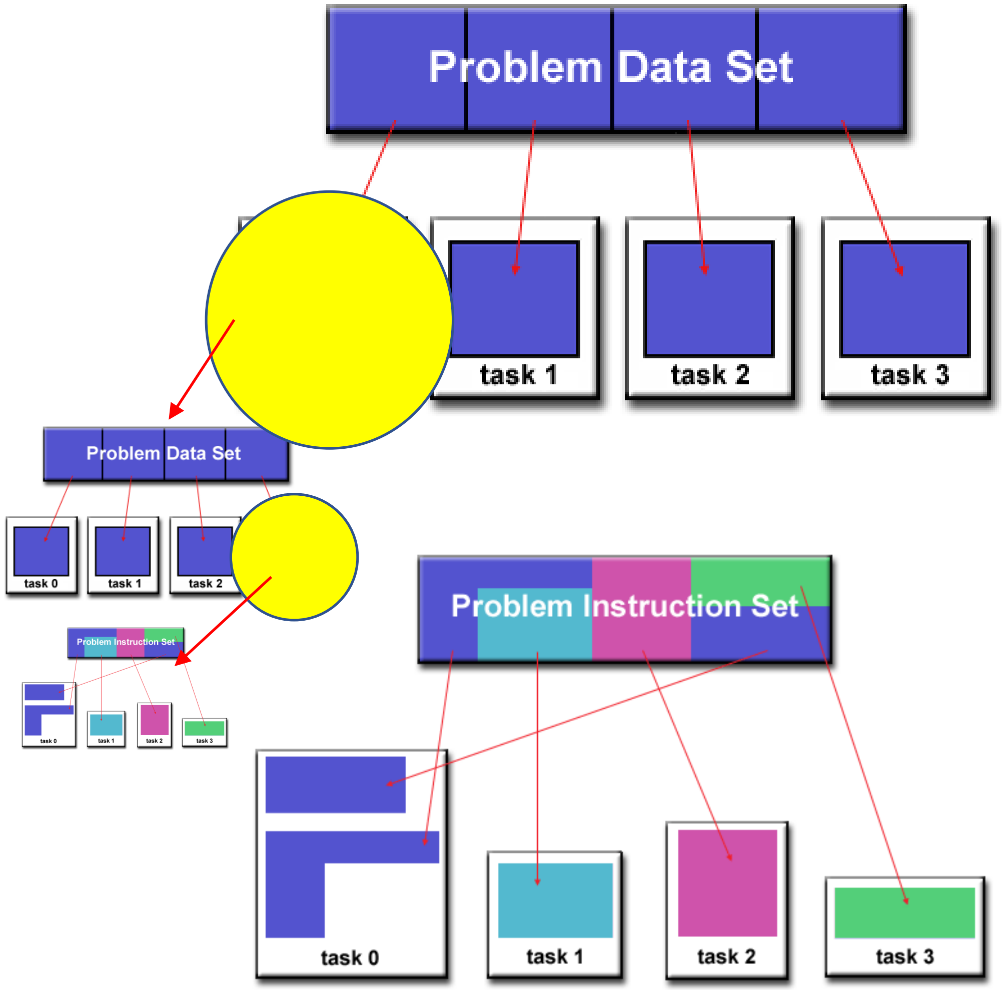
Domain Decomposition

- a.k.a. Data Parallel
- Input domain
- Output Domain

Functional Decomposition

- a.k.a. Task Parallel
- Independent Tasks
- Pipelining

Real Problems: mix/nest



Exercise: Parallelizing Fibonacci

Serial Fibonacci:

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

Exercise: Parallelizing Fibonacci

Serial Fibonacci:

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

Parallel Fibonacci:

```c
void * fib(void * arg) {
    int n = get_input(arg);
    if(n<2) {
        put_result(arg, 1);
    } else {
        pthread_t xtid, ytid;
        pthread_create(&xtid, fib, arg); // n-1
        pthread_create(&ytid, fib, arg); // n-2
        pthread_join(xtid);
        pthread_join(ytid);
        int x = ...
        int y = ...
        put_result(arg, x+y);
    }
}
```

Exercise: Parallelizing Fibonacci

Serial Fibonacci:

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

Parallel Fibonacci:

```c
void * fib(void * arg) {
    int n = get_input(arg);
    if(n<2) {
        put_result(arg, 1);
    } else {
        pthread_t xtid, ytid;
        pthread_create(&xtid, fib, arg); // n-1
        pthread_create(&ytid, fib, arg); // n-2
        pthread_join(xtid);
        pthread_join(ytid);
        int x = ...
        int y = ...
        put_result(arg, x+y);
    }
}
```

Pros/Cons?

Exercise: Parallelizing Fibonacci

Serial Fibonacci:

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

Parallel Fibonacci:

```c
void * fib(void * arg) {
    int n = get_input(arg);
    if(n<2) {
        put_result(arg, 1);
    } else {
        pthread_t xtid, ytid;
        pthread_create(&xtid, fib, arg); // n-1
        pthread_create(&ytid, fib, arg); // n-2
        pthread_join(xtid);
        pthread_join(ytid);
        int x = ...
        int y = ...
        put_result(arg, x+y);
    }
}
```

Pros/Cons?

Challenges:

- Granularity/overheads
- Coupled algorithm, parallel structure
- Each level → more parallelism
- How to balance load?

Cilk

Goal:

Support dynamic, asynchronous, concurrent programs.

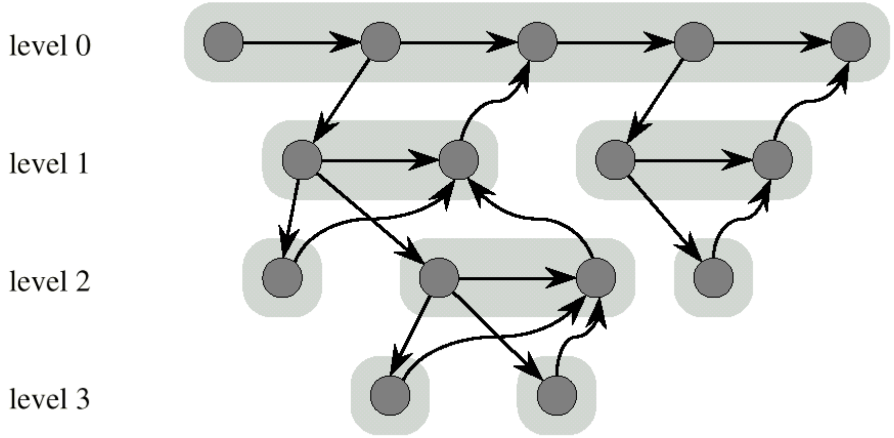
Cilk programmer optimizes:

Total work

Critical path

A Cilk computation:

Dynamic, directed acyclic graph (dag)



Cilk

Goal:

Support dynamic, asynchronous, concurrent programs.

Cilk prograr

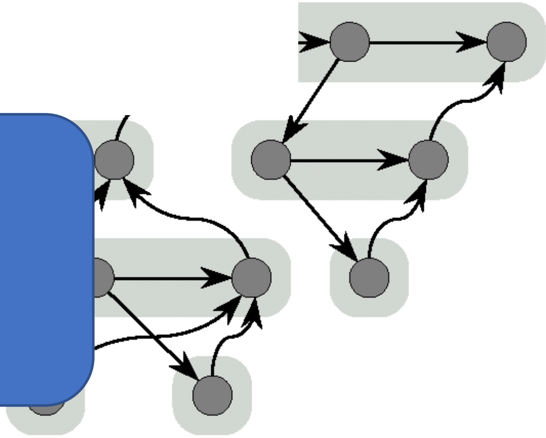
Total worl

Critical pa

A Cilk comp

Dynamic,

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



Key idea(s):

- Programmer writes mostly algorithms
- Programmer *identifies parallelism*
- Runtime figures out mapping to machine

Cilk: Nomenclature

Cilk *program* is a set of procedures

A *procedure* is a *sequence* of threads

Cilk *threads* are:

- represented by nodes in the DAG
- **Non-blocking**: run to completion:
- **no** waiting or suspension; **atomic** units of execution

Cilk: Nomenclature

Cilk *program* is a set of procedures

A *procedure* is a *sequence* of threads

Cilk *threads* are:

- represented by nodes in the DAG
- **Non-blocking**: run to completion:
- **no** waiting or suspension; **atomic** units of execution

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

Cilk: Nomenclature

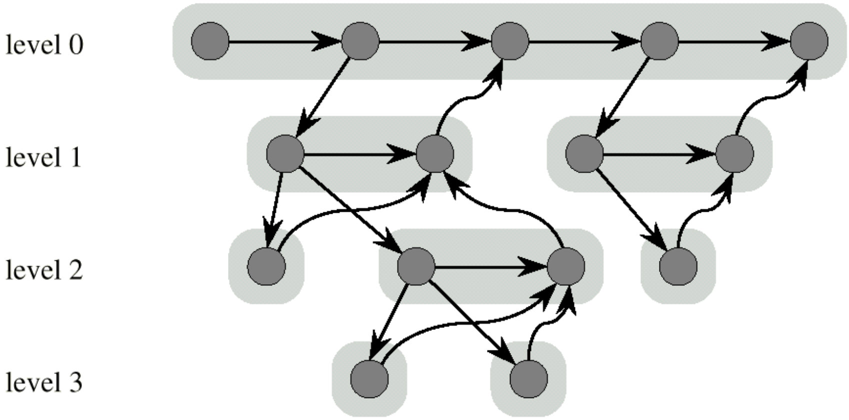
Cilk *program* is a set of procedures

A *procedure* is a *sequence* of threads

Cilk *threads* are:

- represented by nodes in the DAG
- **Non-blocking**: run to completion:
- **no** waiting or suspension; **atomic** units of execution

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



Cilk: Nomenclature

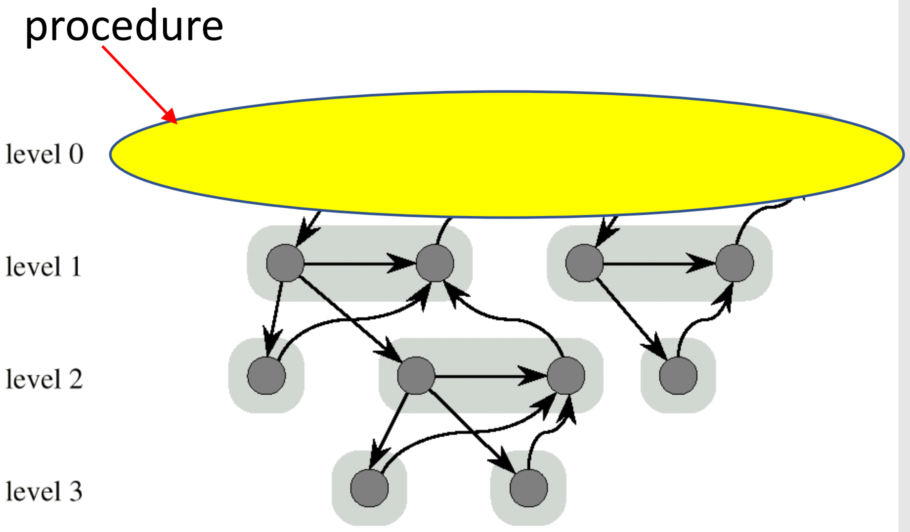
Cilk *program* is a set of procedures

A *procedure* is a *sequence* of threads

Cilk *threads* are:

- represented by nodes in the DAG
- **Non-blocking**: run to completion:
- **no** waiting or suspension; **atomic** units of execution

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



Cilk: Nomenclature

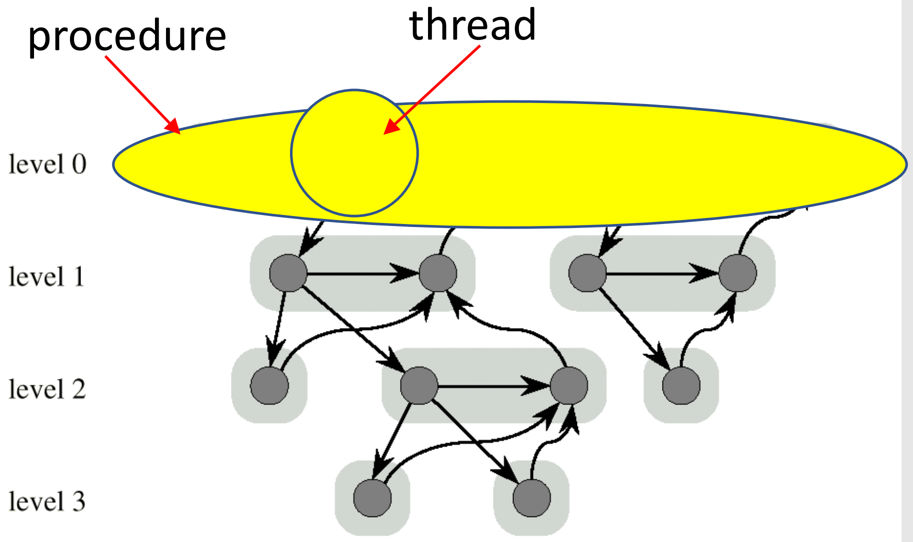
Cilk *program* is a set of procedures

A *procedure* is a *sequence* of threads

Cilk *threads* are:

- represented by nodes in the DAG
- **Non-blocking**: run to completion:
- **no** waiting or suspension; **atomic** units of execution

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



Programming Model

Threads can *spawn* children

- Primary mechanism to create parallel work
- downward edges connect a parent to its children

A child & parent can run concurrently.

- Non-blocking threads ➔ a child cannot return a value to its parent.
- The parent spawns a *successor* that receives values from its children

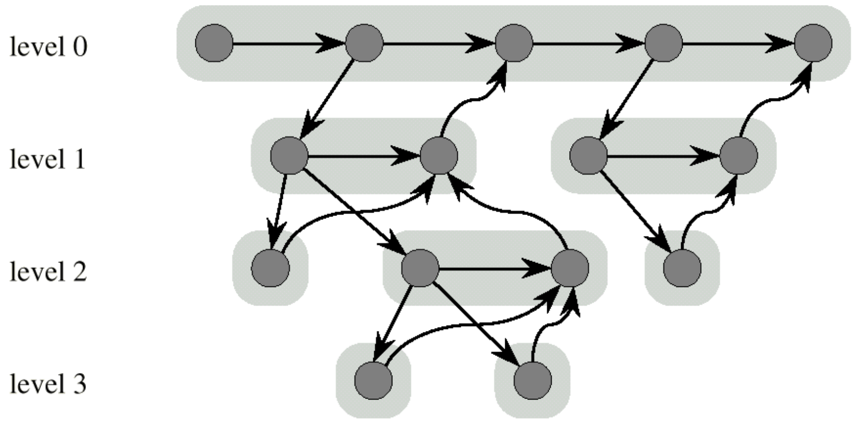
Programming Model

Threads can *spawn* children

- Primary mechanism to create parallel work
- downward edges connect a parent to its children

A child & parent can run concurrently.

- Non-blocking threads ➔ a child cannot return a value to its parent.
- The parent spawns a *successor* that receives values from its children



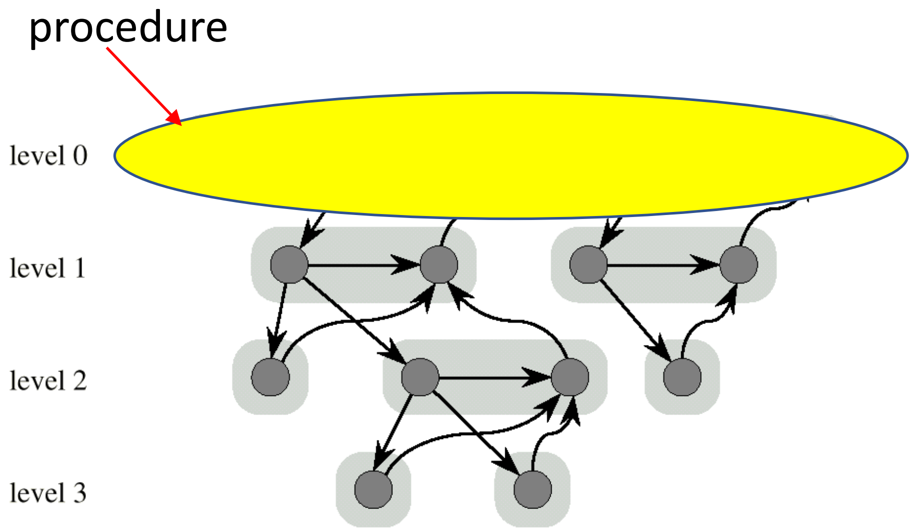
Programming Model

Threads can *spawn* children

- Primary mechanism to create parallel work
- downward edges connect a parent to its children

A child & parent can run concurrently.

- Non-blocking threads ➔ a child cannot return a value to its parent.
- The parent spawns a *successor* that receives values from its children



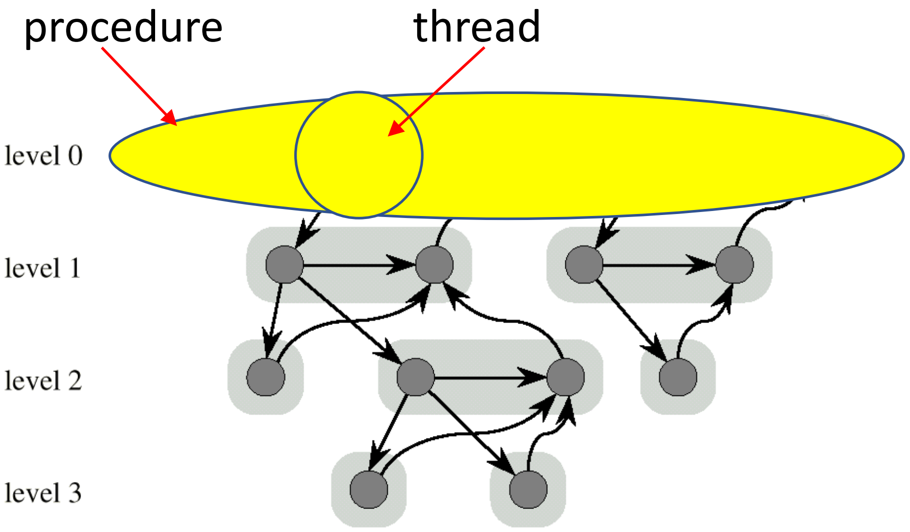
Programming Model

Threads can *spawn* children

- Primary mechanism to create parallel work
- downward edges connect a parent to its children

A child & parent can run concurrently.

- Non-blocking threads ➔ a child cannot return a value to its parent.
- The parent spawns a *successor* that receives values from its children



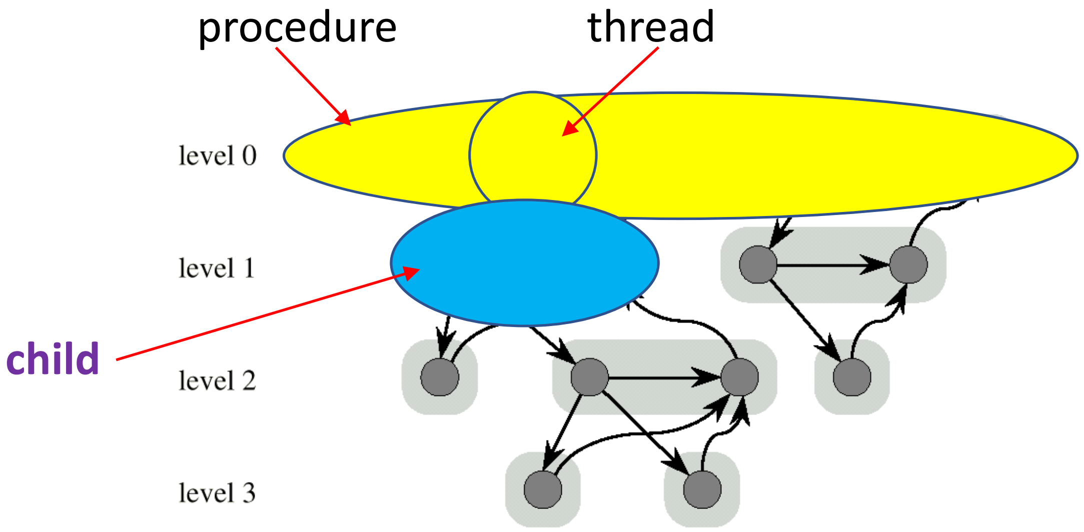
Programming Model

Threads can *spawn* children

- Primary mechanism to create parallel work
- downward edges connect a parent to its children

A child & parent can run concurrently.

- Non-blocking threads ➔ a child cannot return a value to its parent.
- The parent spawns a *successor* that receives values from its children



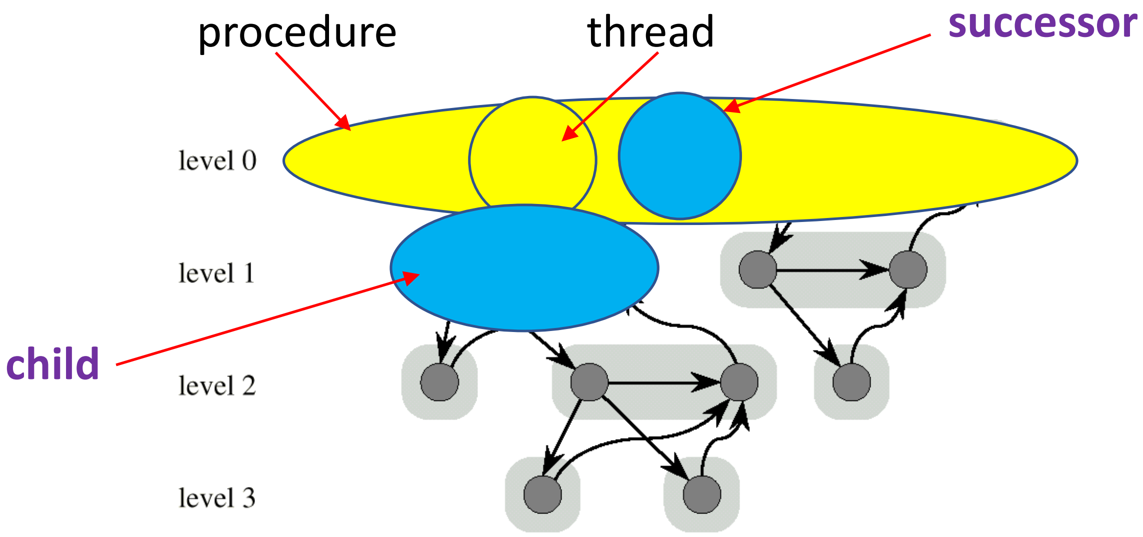
Programming Model

Threads can *spawn* children

- Primary mechanism to create parallel work
- downward edges connect a parent to its children

A child & parent can run concurrently.

- Non-blocking threads ➔ a child cannot return a value to its parent.
- The parent spawns a *successor* that receives values from its children



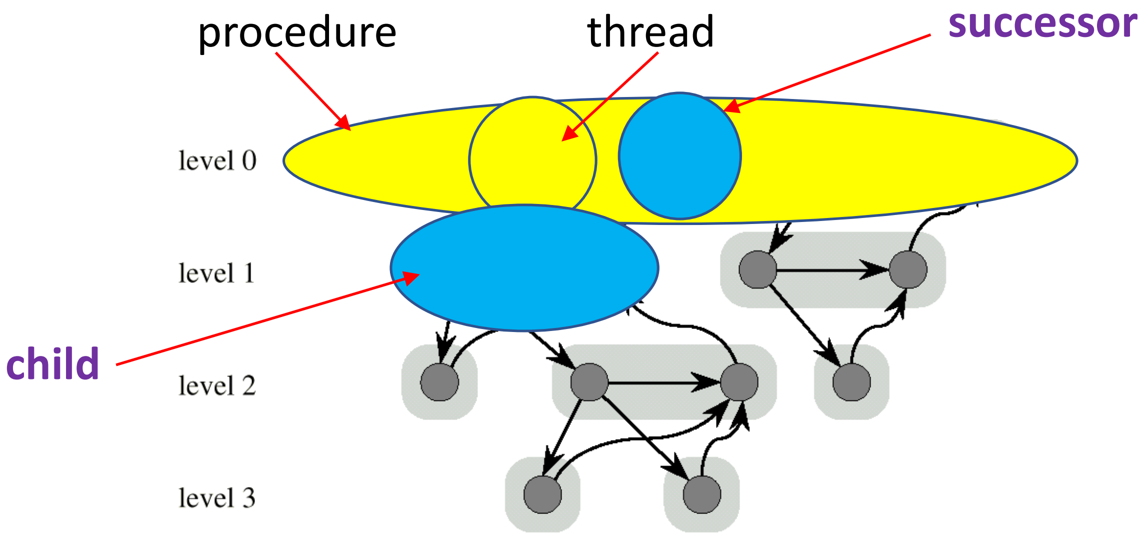
Programming Model

Thread & successor: parts of the same Cilk procedure.

- Connected by horizontal arcs

Children's returned values:

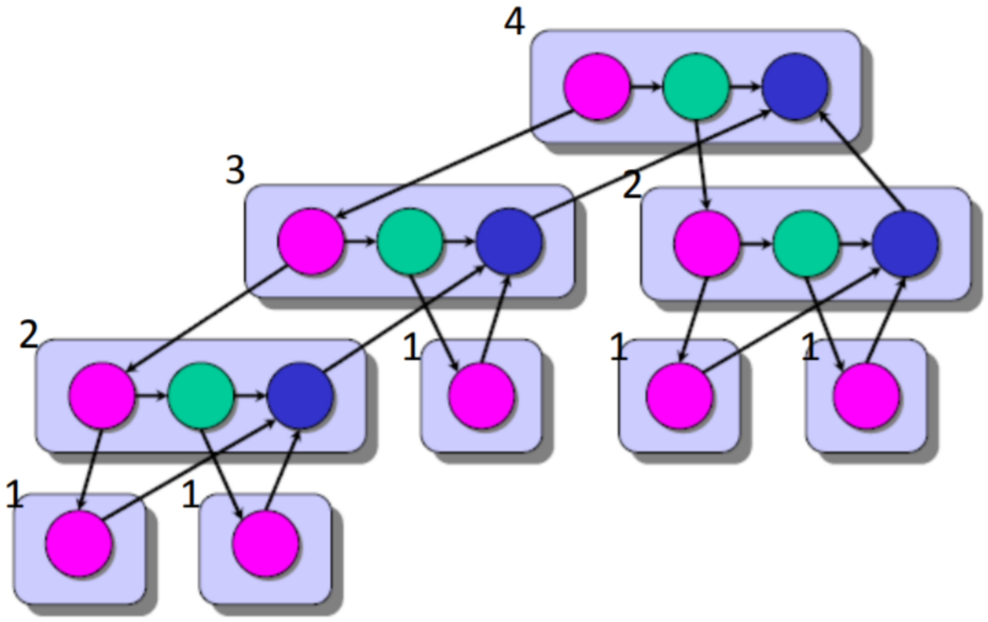
- Received before their successor begins
- They constitute data dependencies.
- Connected by curved arcs



Execution Model

```
cilk fib (int n) {
 if (n < 2) return 1;
   else {
      int rst = 0;
      rst += spawn fib (n-1);
      rst += spawn fib (n-2);
    sync;
     return rst;
  }
}
```

Compute: fib (4)



Source: http://supertech.csail.mit.edu/cilk/lecture-1.pdf

Explicit Continuation Passing

Nonblocking threads ➔ parent cannot block on children's results.

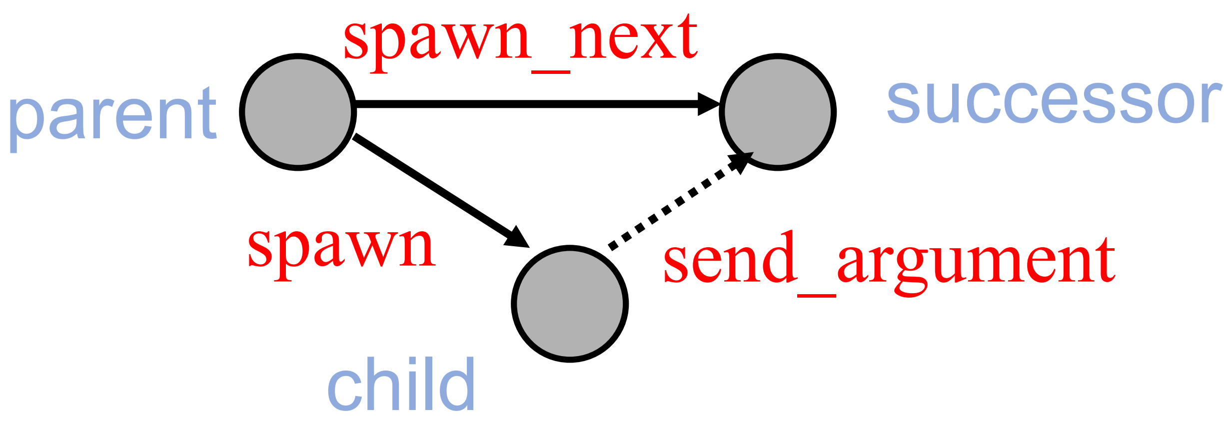
Parent spawns a successor thread.

Called *explicit continuation passing*.

Cilk primitive to *send a value* from a closure to another:

send_argument(k, value)

sends value to the argument slot of a waiting closure specified by continuation k.



Environment: Closures and Continuations

A *closure* is a data structure that has:

a pointer to the C function for T

a slot for each argument

(inputs & continuations)

join counter: # of missing arg values

Closure is ready when join counter == 0.

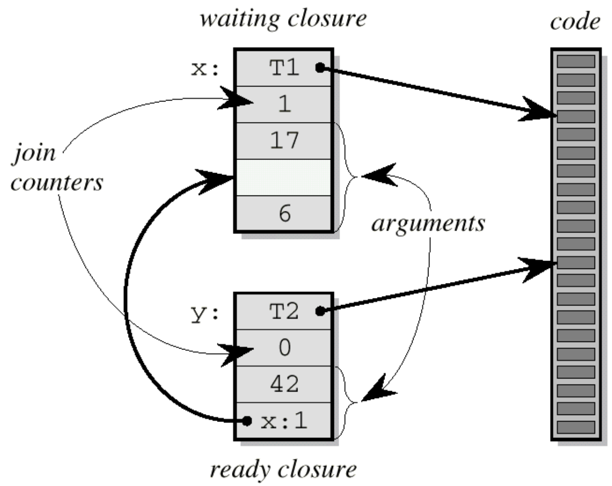
A closure is waiting otherwise.

Closures allocated from a runtime heap

- *Continuation* is a data type,

```
cont int x;
```

- Global reference to an *empty slot of a closure*.
- Implemented as 2 items:
  - pointer to closure; (what thread)
  - int value: slot number. (what input)



Execution Time & Scheduling

Execution time of a Cilk program using P cores **depends on**:

Work (T_1): time for Cilk program with 1 processor to complete.

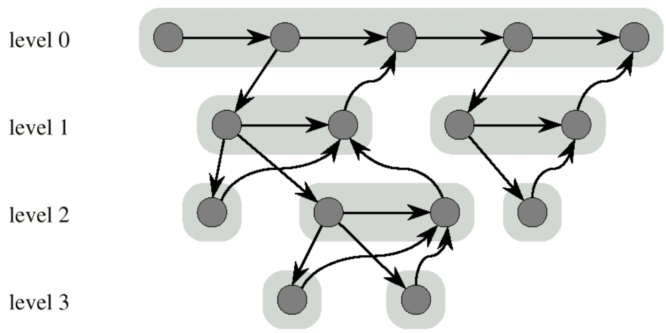
Critical path (T_∞): the time to execute the longest directed path in the DAG.

$T_P >= T_1 / P$

$T_P >= T_\infty$

Parallelism = T_1 / T_∞ **or** *(Work/Depth)*

- Cilk uses run time scheduling: work stealing.
- For "fully strict" programs
  - asymptotic optimality for:
  - space, time, & communication



Nonblocking Threads: Pros, Cons

Nonblocking Threads: Pros, Cons

Shallow call stack.

Simplify runtime system:

Completed threads leave C runtime stack empty.

Portable runtime implementation

Nonblocking Threads: Pros, Cons

Shallow call stack.

Simplify runtime system:

- Completed threads leave C runtime stack empty.

Portable runtime implementation

Con: programmer deals with continuation passing.

Stealing Work: The Ready Deque

Work-stealing:

- Process with no work selects a victim
- Gets shallowest thread in victim's spawn tree.

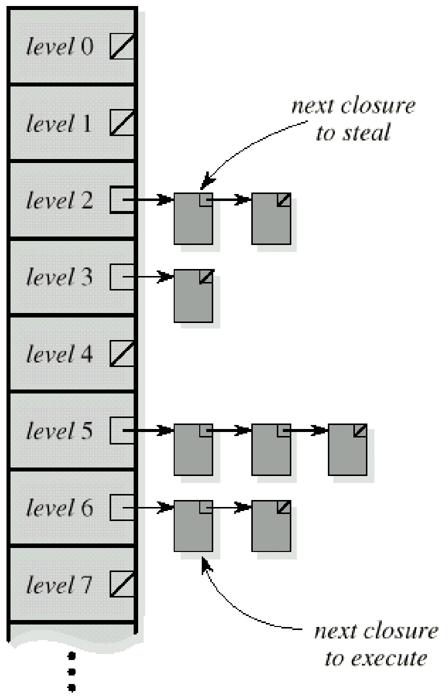
Thieves choose victim processor *randomly*.

Each closure has a level:

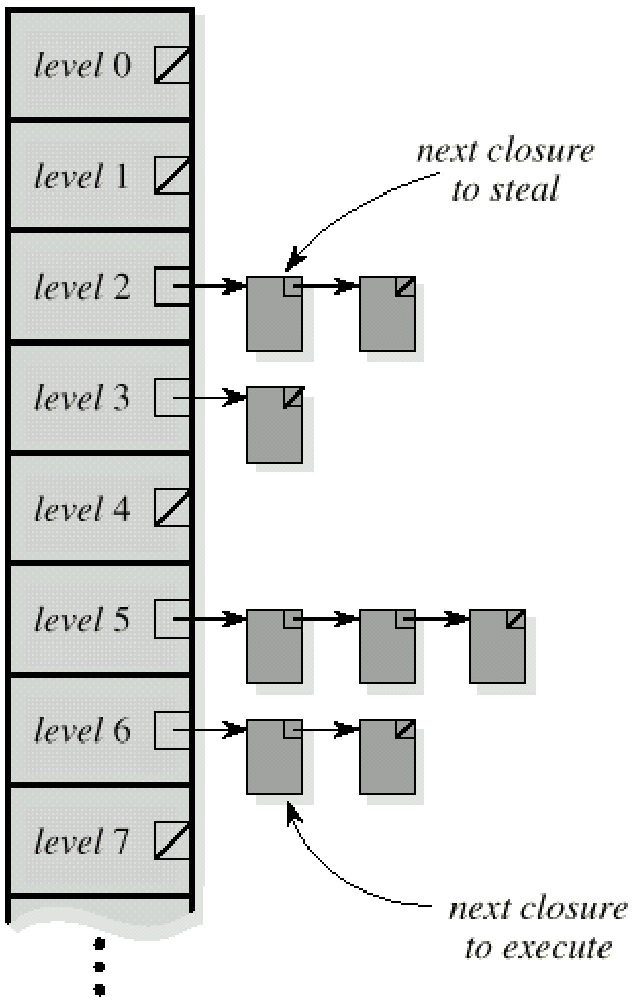
- level(child) = level(parent) + 1
- level(successor) = level(parent)

Each processor keeps a ready deque:

- Contains ready closures
- The **L**th element contains the list of all ready closures whose level is **L**.



Ready Deque



if (! readyDeque .isEmpty())

take deepest thread

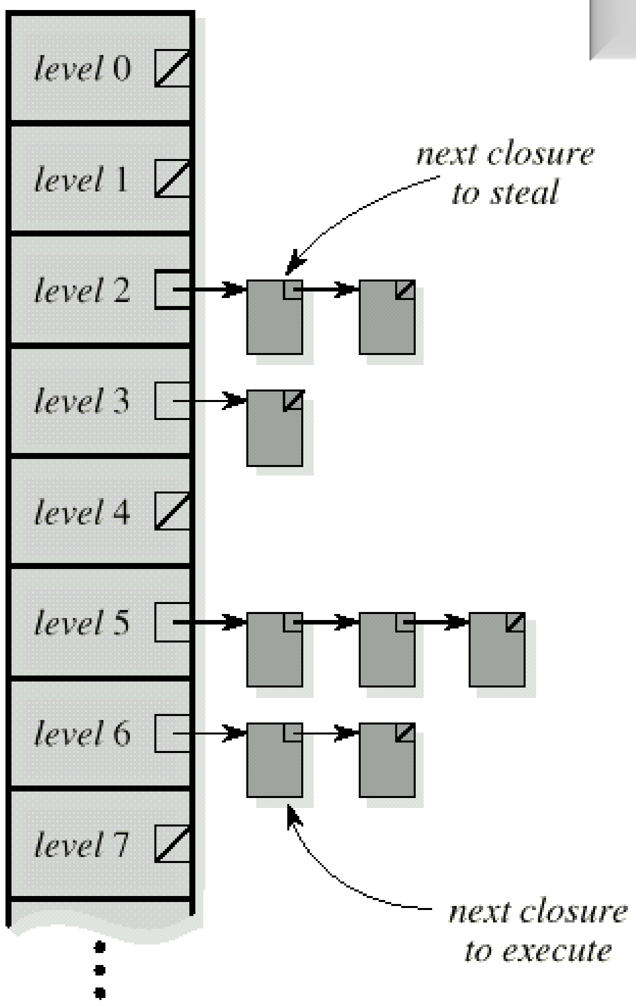
else

steal shallowest thread from

readyDeque of *randomly*

selected victim

Ready Deque



if (! readyDeque .isEmpty())

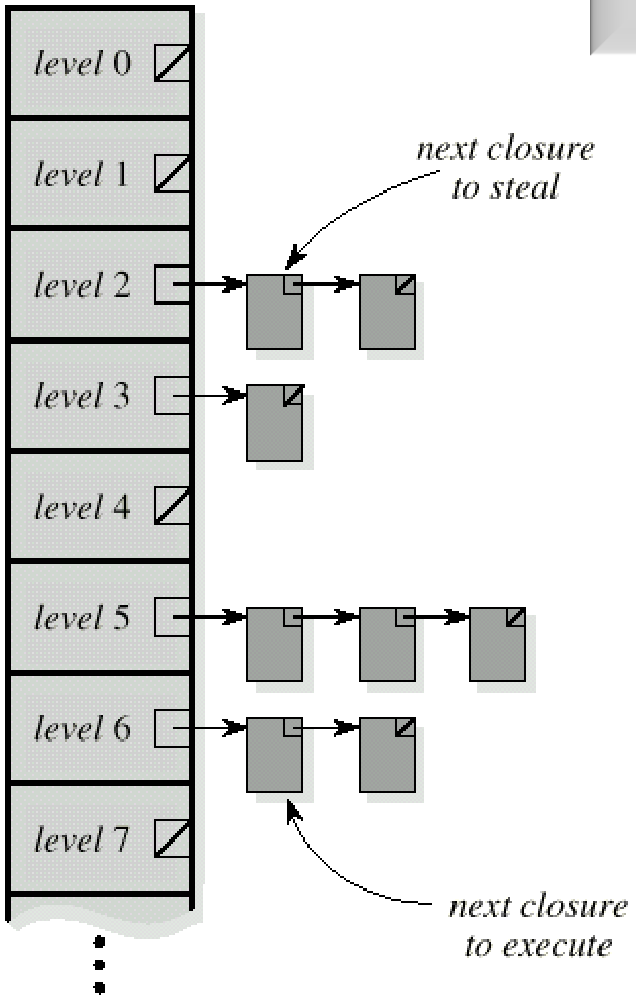
take deepest thread

else

steal shallowest thread from readyDeque of *randomly selected* victim

Ready Deque

Why steal shallowest closure?



if (! readyDeque .isEmpty())

take deepest thread

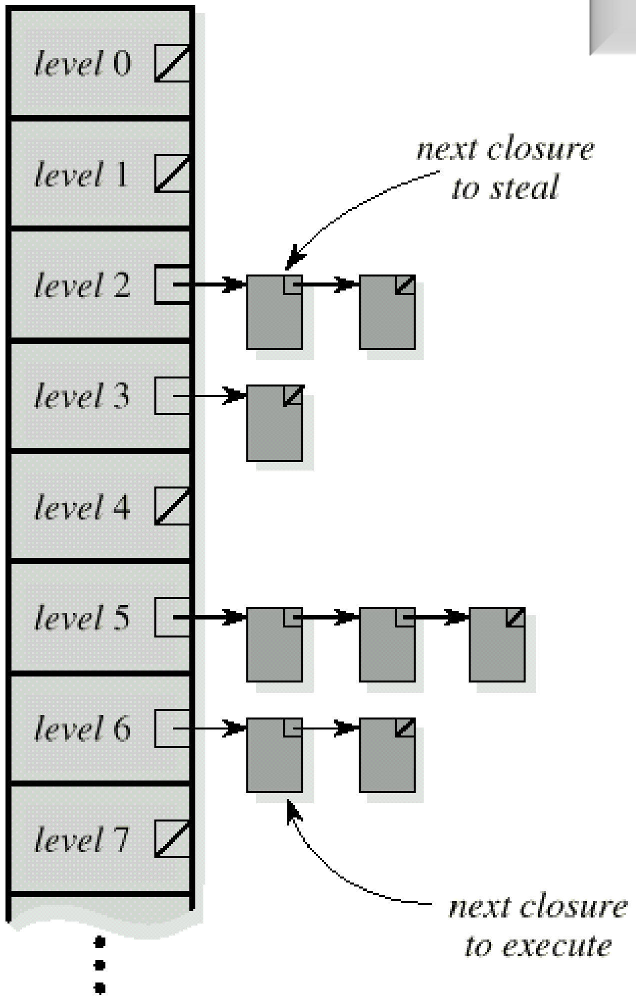
else

steal shallowest thread from readyDeque of *randomly selected* victim

Ready Deque

Why steal shallowest closure?

They *probably* produce more work → reduce communication.



if (! readyDeque .isEmpty())

take deepest thread

else

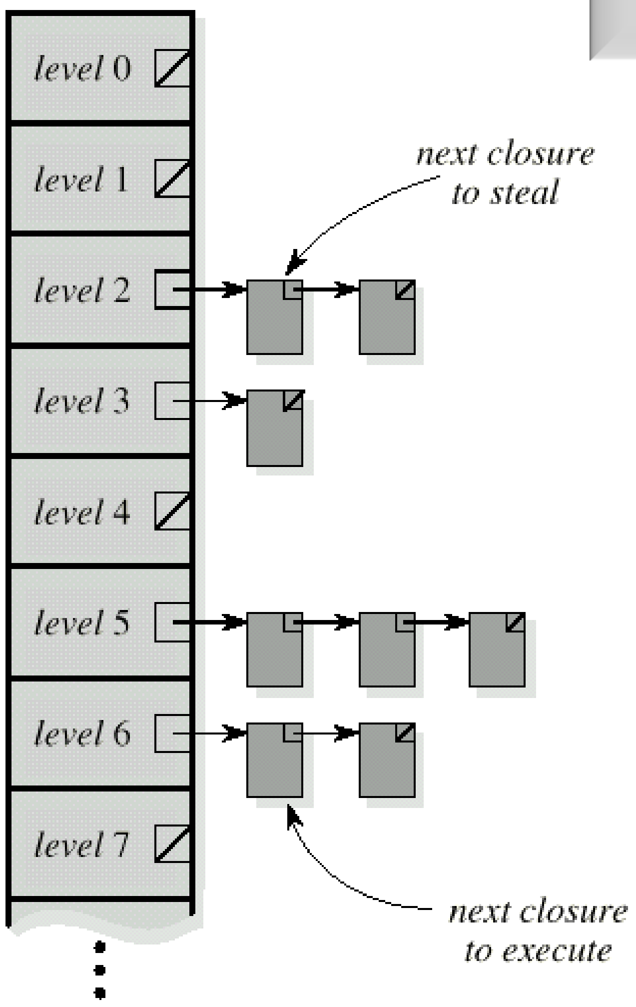
steal shallowest thread from readyDeque of *randomly selected* victim

Ready Deque

Why steal shallowest closure?

They *probably* produce more work → reduce communication.

Shallow threads *more likely to be* on critical path.



if (! readyDeque .isEmpty())

take deepest thread

else

steal shallowest thread from readyDeque of *randomly selected* victim

Cilk Language

Cilk is an extension of C

Cilk programs are:

- preprocessed to C
- linked with a runtime library

- Declaring a thread:

thread T (<args>) { <stmts> }

- T is preprocessed
  - C function of 1 argument
  - return type void.
- The 1 argument: points to *closure*

Cilk Language

Cilk is an extension of C

Cilk programs are:

- preprocessed to C
- linked with a runtime library

- Declaring a thread:

thread T (<args>) { <stmts> }

- T is preprocessed
  - C function of 1 argument
  - return type void.
- The 1 argument: points to *closure*

Serial Elision: remove cilk keywords→serial program

Cilk Language

Cilk is an extension of C

Cilk programs are:

- preprocessed to C
- linked with a runtime library

- Declaring a thread:

thread T (<args>) { <stmts> }

- T is preprocessed
  - C function of 1 argument
  - return type void.
- The 1 argument: points to *closure*

Serial Elision: remove cilk keywords→serial program

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

Cilk Language

Cilk is an extension of C

Cilk programs are:

- preprocessed to C
- linked with a runtime library

- Declaring a thread:

thread T (<args>) { <stmts> }

- T is preprocessed
  - C function of 1 argument
  - return type void.
- The 1 argument: points to *closure*

Serial Elision: remove cilk keywords→serial program

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

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

Concluding Remarks

Cilk illustrates a number of important (recurring) ideas:

- DAG-based parallel execution model
- Critical-path heuristic for available parallelism
- Continuation passing
- Work-stealing scheduling

Discussion/Food For Thought:

- Is continuation passing style (CPS) difficult?
- Why/why not?

Title

Content