Multithreaded Programming in *Cilk*

Charles E. Leiserson
MIT

Cilk

A C language for programming dynamic multithreaded applications on shared-memory multiprocessors.

- virus shell assembly
- graphics rendering
- *n*-body simulation
- heuristic search
- dense and sparse matrix computations
- friction-stir welding simulation
- artificial evolution

***Cilk**'s provably good runtime system automatically manages low-level aspects of parallel execution, including protocols, load balancing, and scheduling.*

Fibonacci

```
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);
  }
}
```

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.

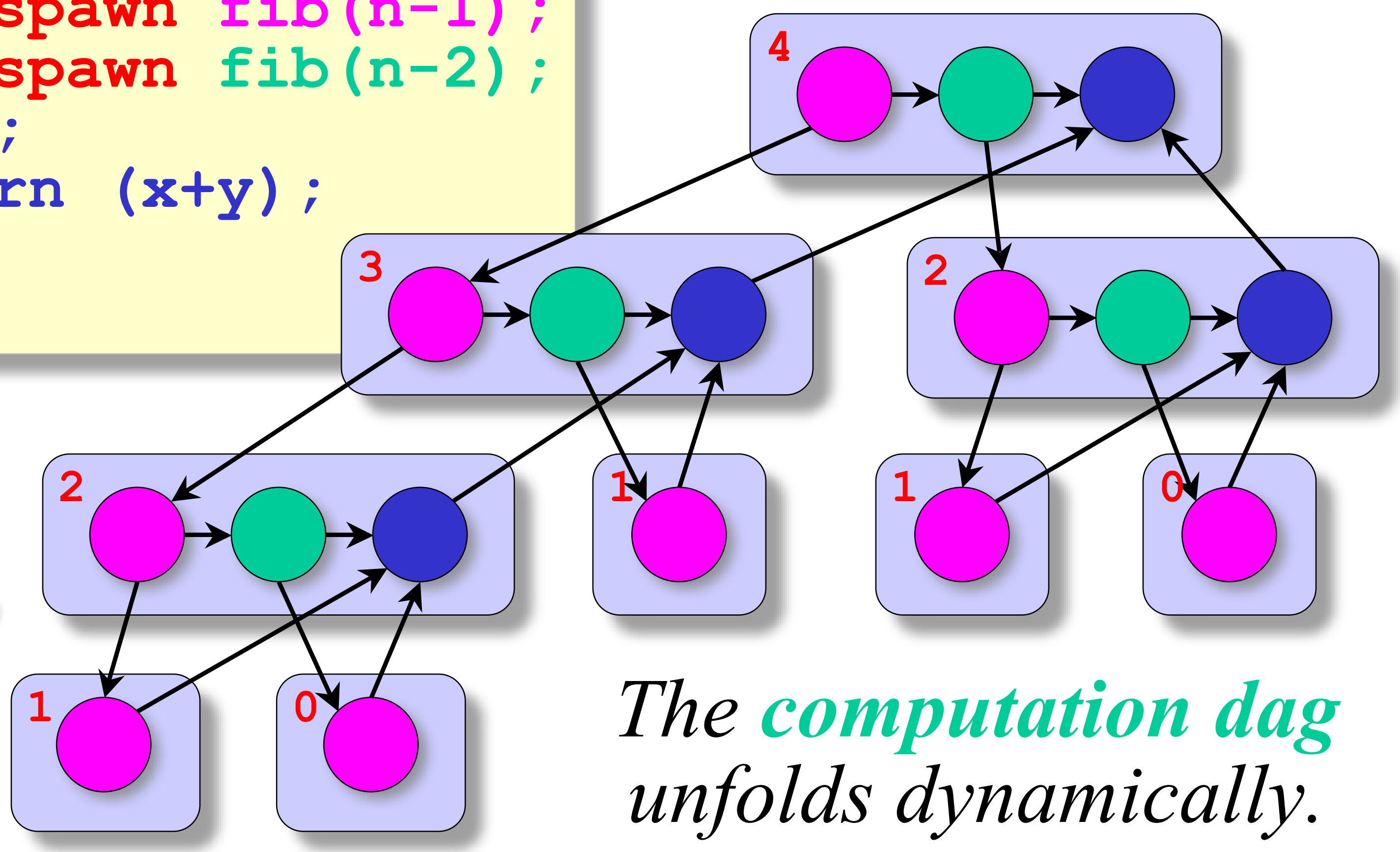
The named ***child*** *Cilk* procedure can execute in parallel with the ***parent*** caller.

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

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"

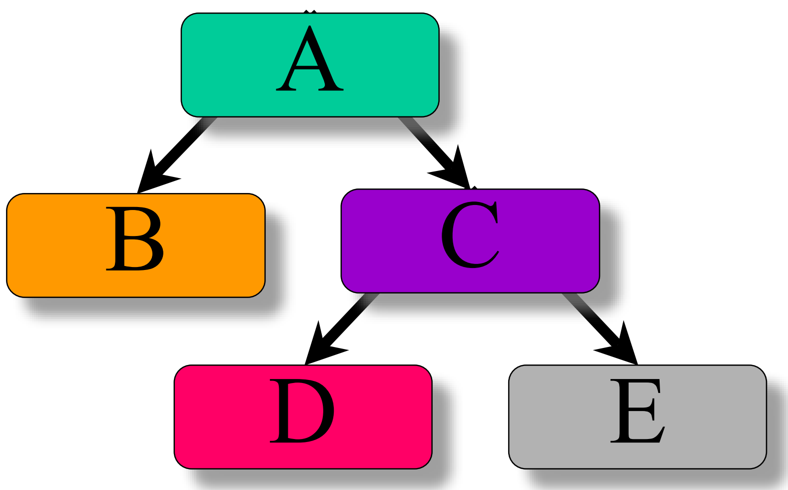
*The **computation dag** unfolds dynamically.*

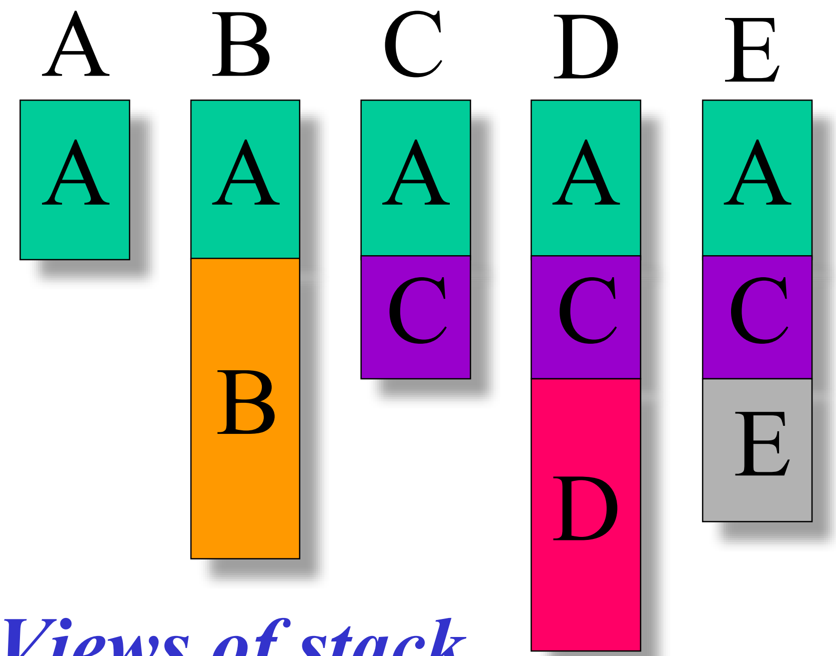
Cilk's Thread Scheduler

- *Cilk*'s randomized "***work-stealing***" scheduler load-balances the computation at run-time.
- A mathematical proof guarantees near-perfect linear speed-up on applications with sufficient parallelism, as long as the architecture has sufficient memory bandwidth.
- A *Cilk* program running on 1 processor typically exhibits less than 2% slowdown compared with its C elision.
- A **`spawn`**/**`return`** in *Cilk* is over 450 times faster than a Pthread **`create`**/**`exit`** and less than 3 times slower than an ordinary C function call on a modern Intel processor.

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.

Debugging

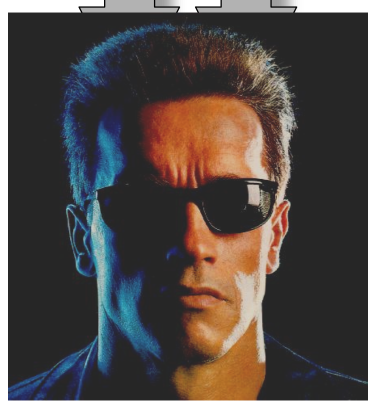
The ***Nondeterminator*** debugging tool provably guarantees to detect and localize data-race bugs.

“Abelian”
Cilk *program*

Input data set

Information localizing a data race.

FAIL



PASS

Every scheduling produces the same result.

A ***data race*** occurs whenever a thread modifies a location and another thread, holding no locks in common, accesses the location simultaneously.

Additional Features

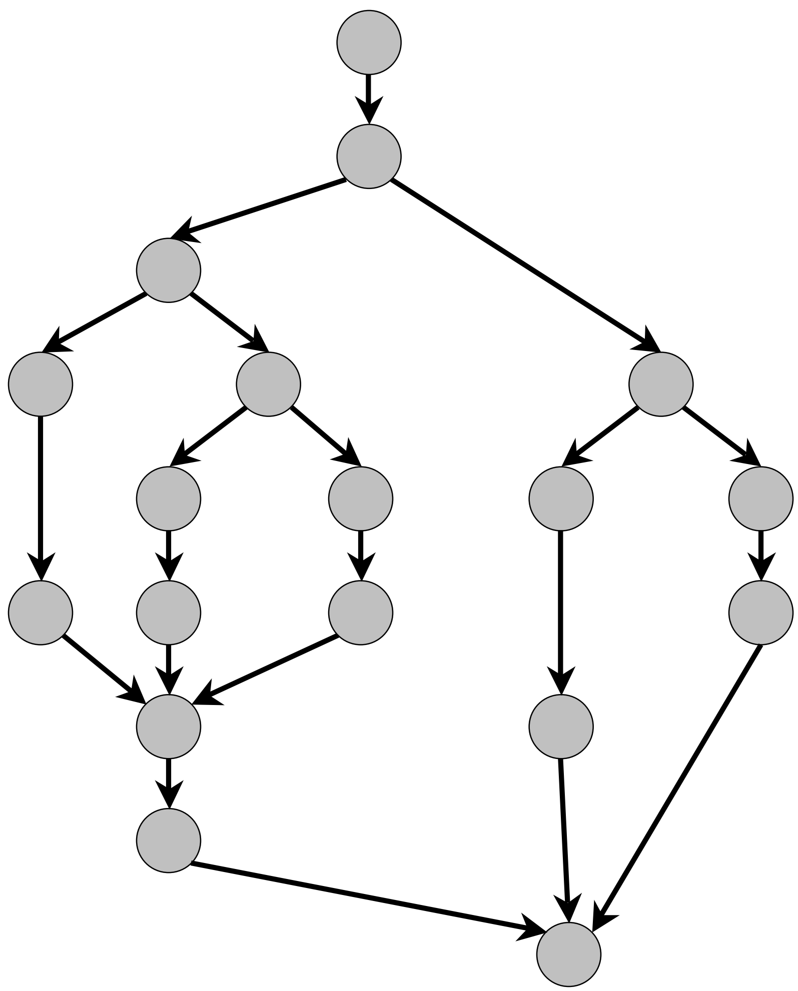
- The `inlet` keyword specifies an internal function that can be called to incorporate a returned result into the parent frame in a nonstandard way when a spawned child returns.
- The `abort` keyword forces all spawned children to terminate abruptly.
- The `SYNCHED` pseudovariable tests whether a `sync` would succeed.
- A *Cilk* library provides ***mutex locks*** for atomicity. (Alas, no ***transactional memory***!)

OUTLINE

- Overview
- *Cilk* Performance
- A Chess Lesson
- Alpha-Beta Search
- *Cilk*'s Scheduler
- Conclusion

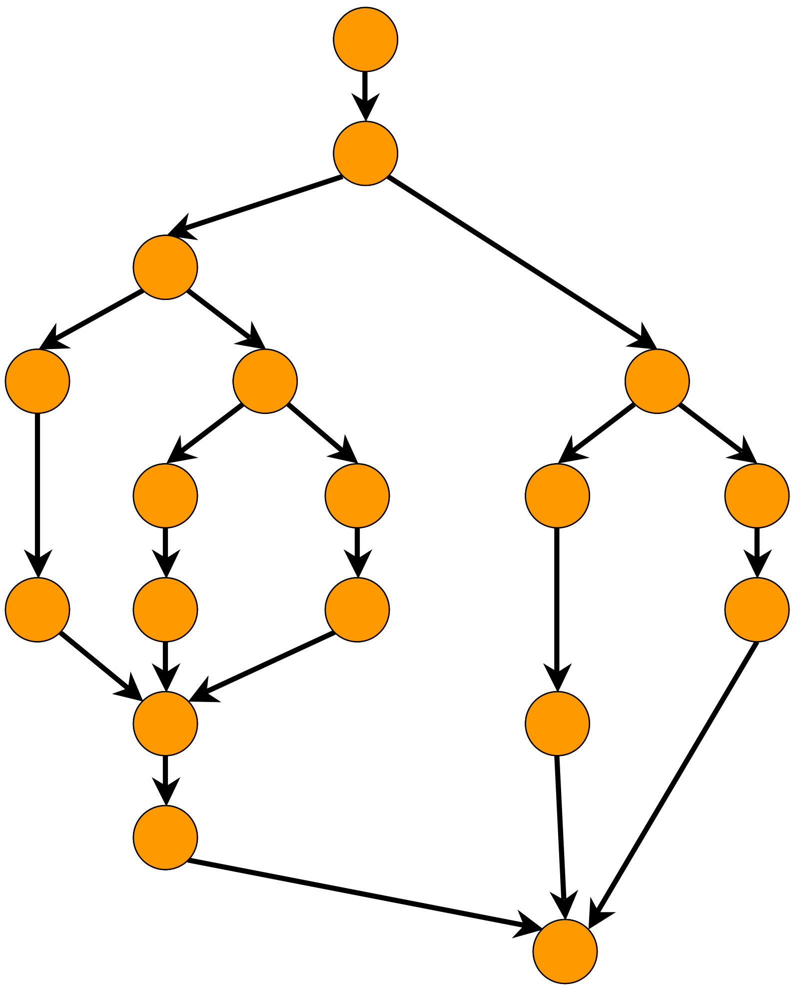
Algorithmic Complexity Measures

T_P = execution time on P processors



Algorithmic Complexity Measures

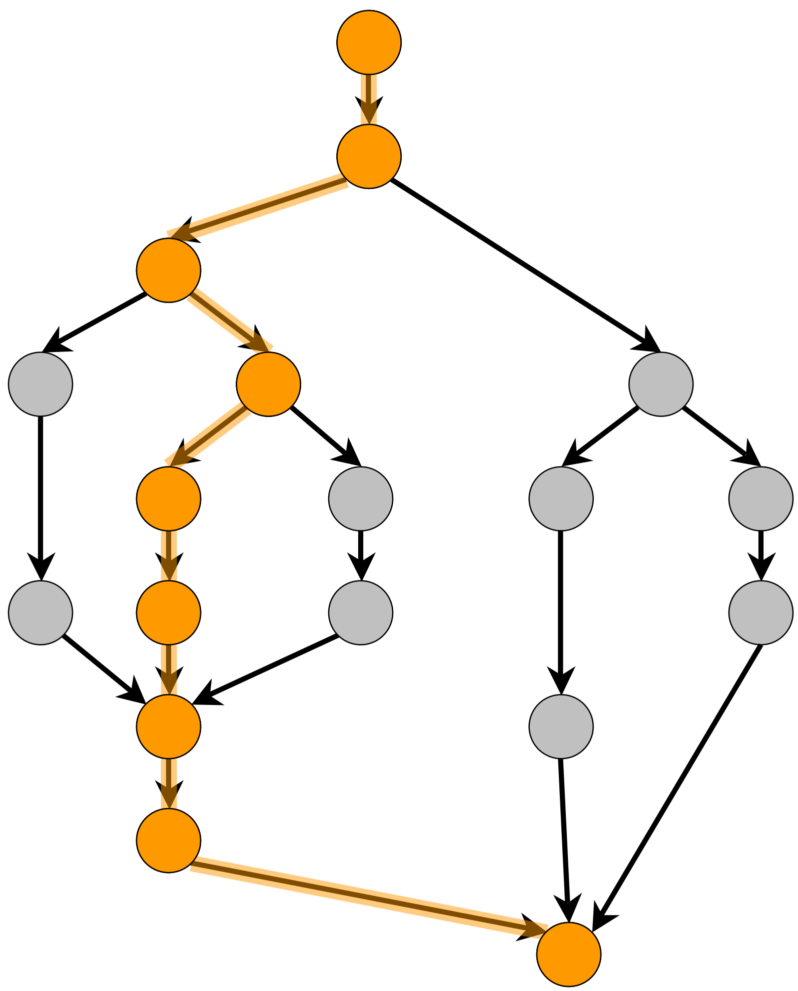
T_P = execution time on P processors



T_1 = ***work***

Algorithmic Complexity Measures

T_P = execution time on P processors

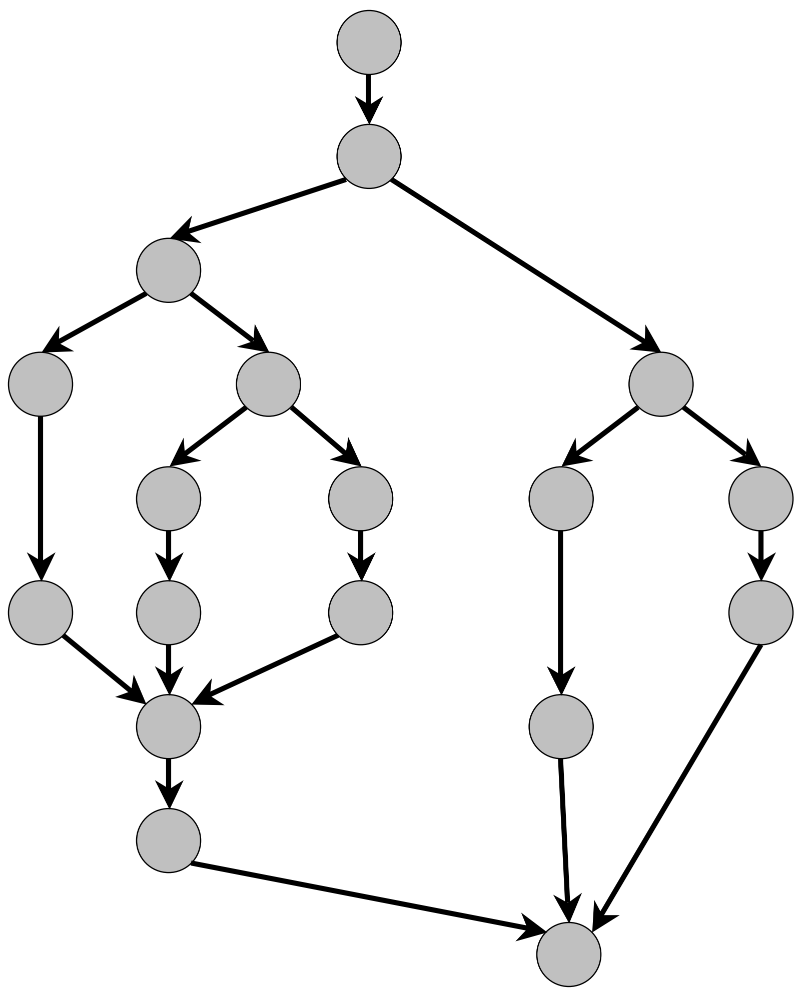


T_1 = ***work***

T_∞ = ***span***

Algorithmic Complexity Measures

T_P = execution time on P processors



T_1 = ***work***

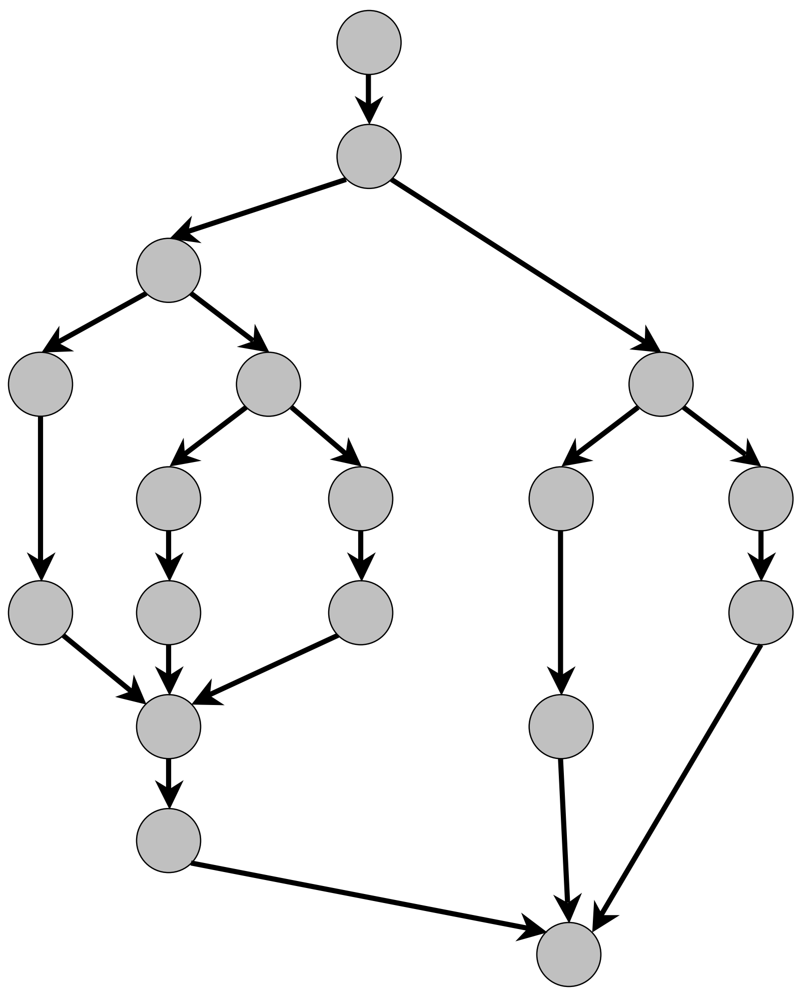
T_∞ = ***span***

Lower Bounds

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

Algorithmic Complexity Measures

T_P = execution time on P processors



T_1 = ***work***

T_∞ = ***span***

Lower Bounds

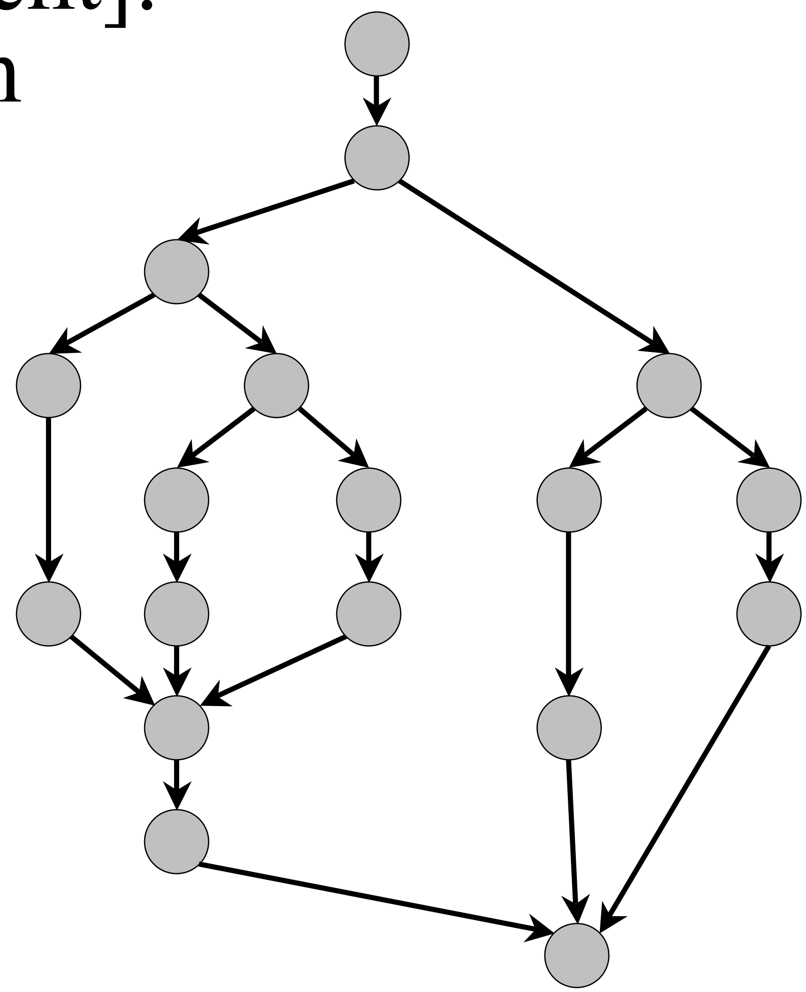
- $T_P \geq T_1/P$
- $T_P \geq T_\infty$

T_1/T_P = ***speedup***

T_1/T_∞ = ***parallelism***

Greedy Scheduling

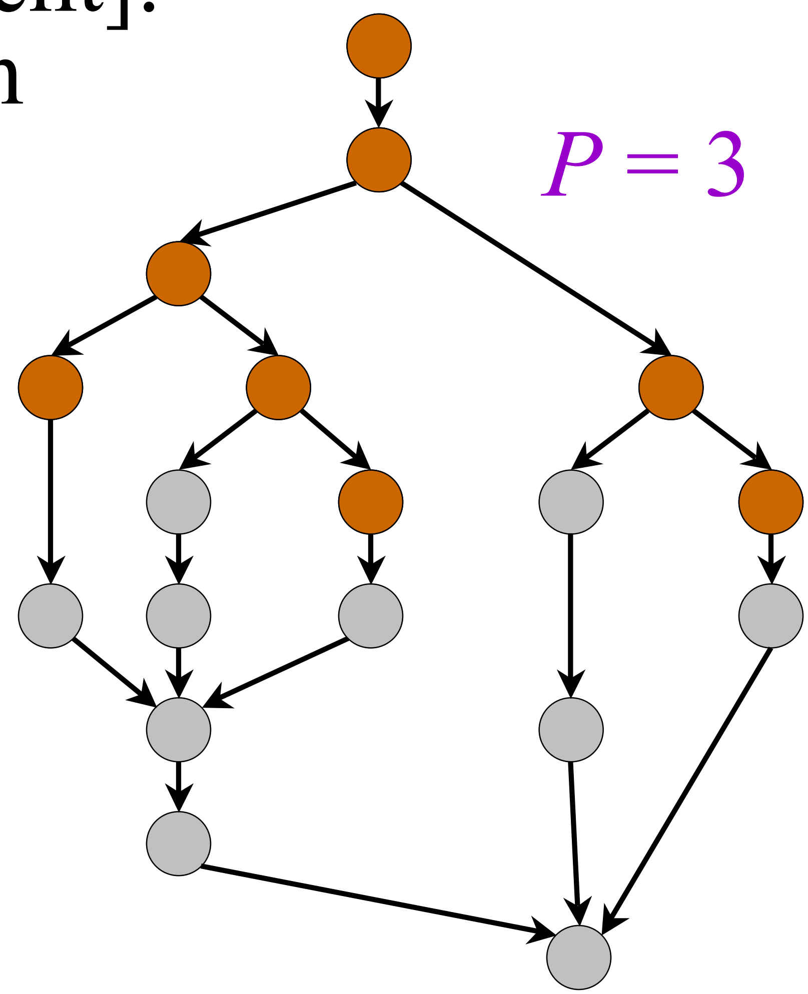
Theorem [Graham & Brent]:
There exists an execution with $T_P \leq T_1/P + T_\infty$.



Greedy Scheduling

Theorem [Graham & Brent]: There exists an execution with $T_P \leq T_1/P + T_\infty$.

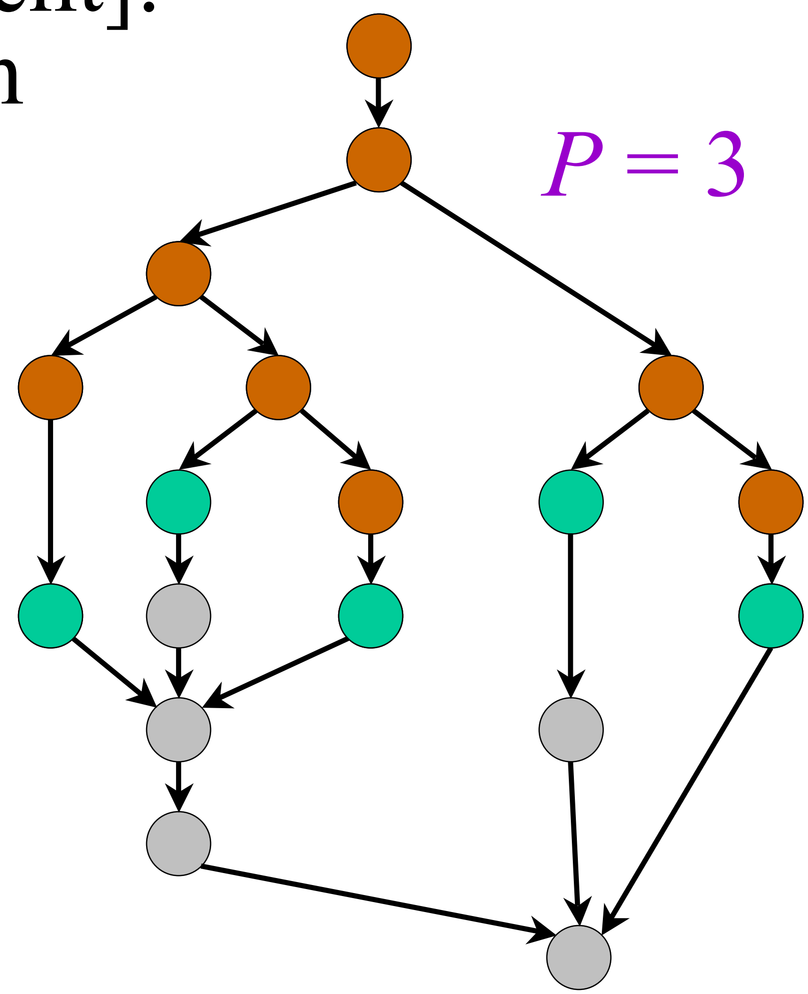
Proof. At each time step, …



Greedy Scheduling

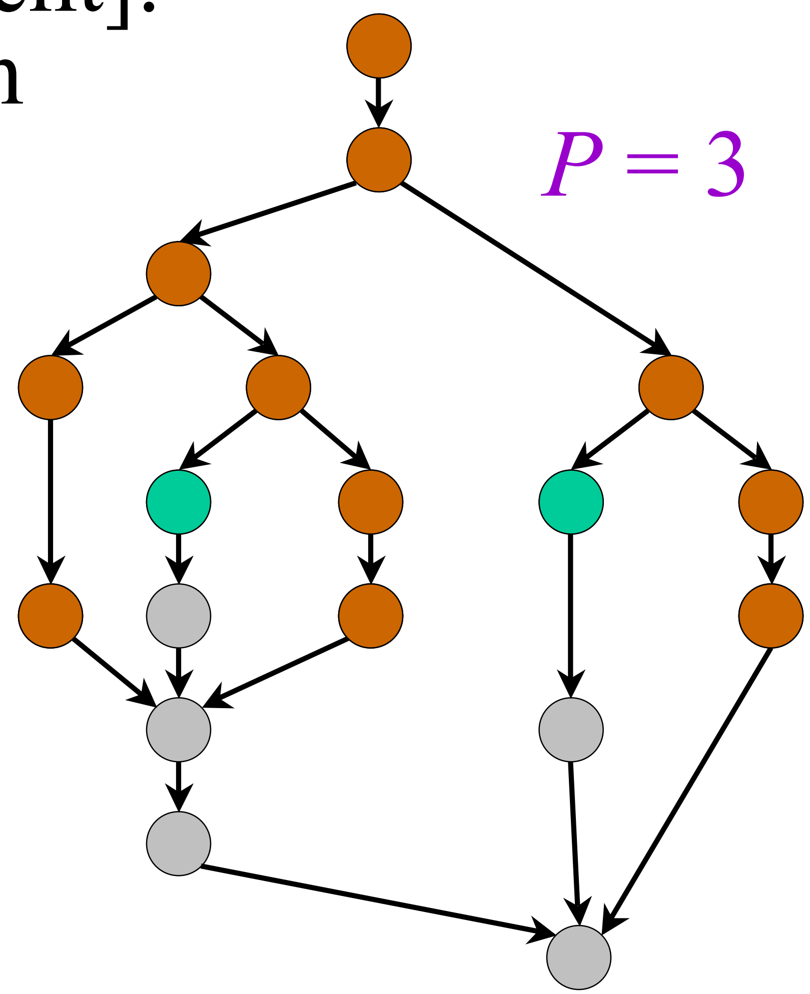
Theorem [Graham & Brent]:
There exists an execution with $T_P \leq T_1/P + T_\infty$.

Proof. At each time step, if at least P tasks are ready, …



Greedy Scheduling

Theorem [Graham & Brent]: There exists an execution with $T_P \leq T_1/P + T_\infty$.

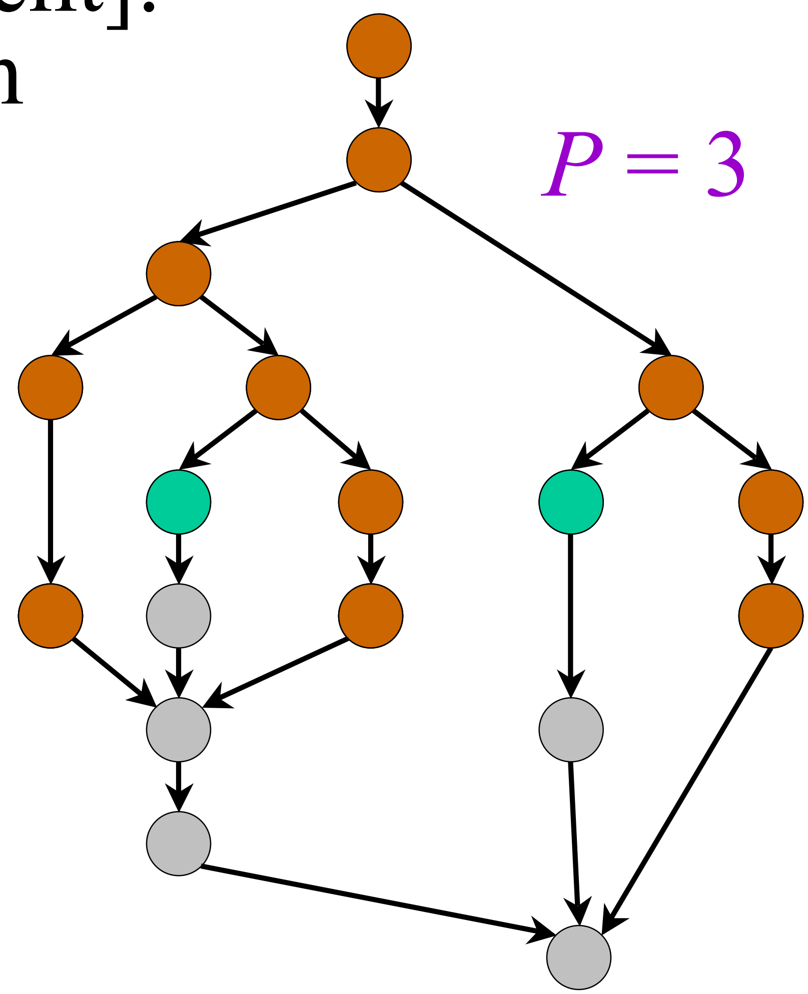


Proof. At each time step, if at least P tasks are ready, execute P of them.

Greedy Scheduling

Theorem [Graham & Brent]: There exists an execution with $T_P \le T_1/P + T_\infty$.

Proof. At each time step, if at least P tasks are ready, execute P of them. If fewer than P tasks are ready, …

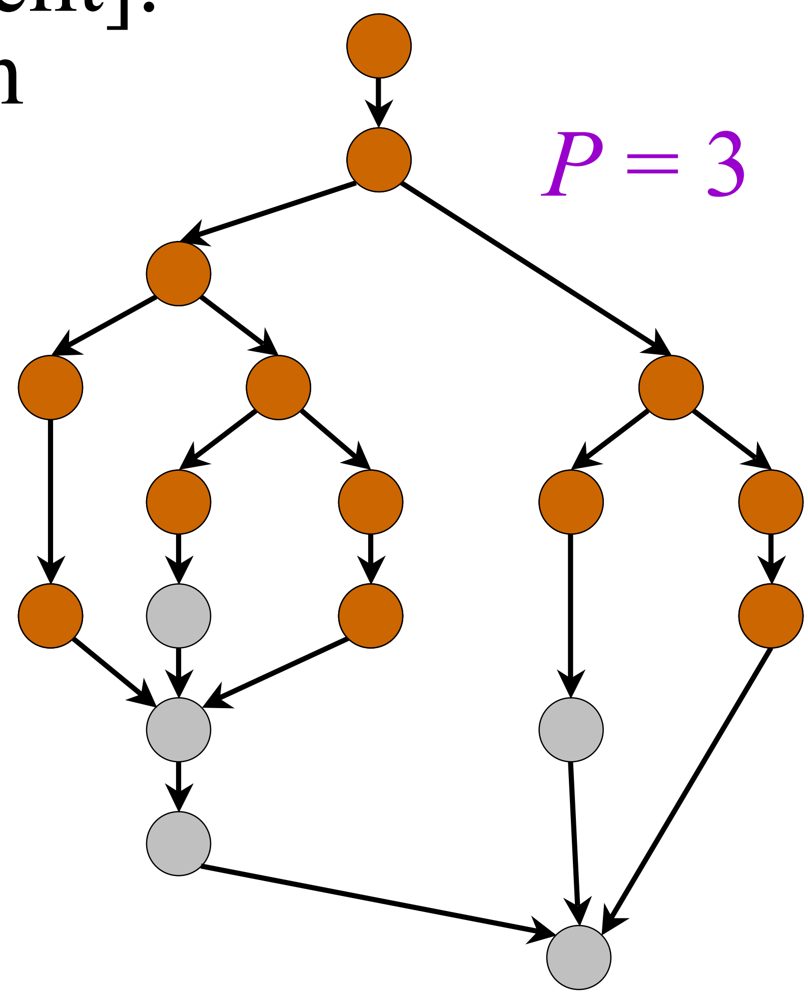


Greedy Scheduling

Theorem [Graham & Brent]: There exists an execution with $T_P \leq T_1/P + \boxed{T_\infty}$.

Proof. At each time step, if at least P tasks are ready, execute P of them. If fewer than P tasks are ready, execute all of them.

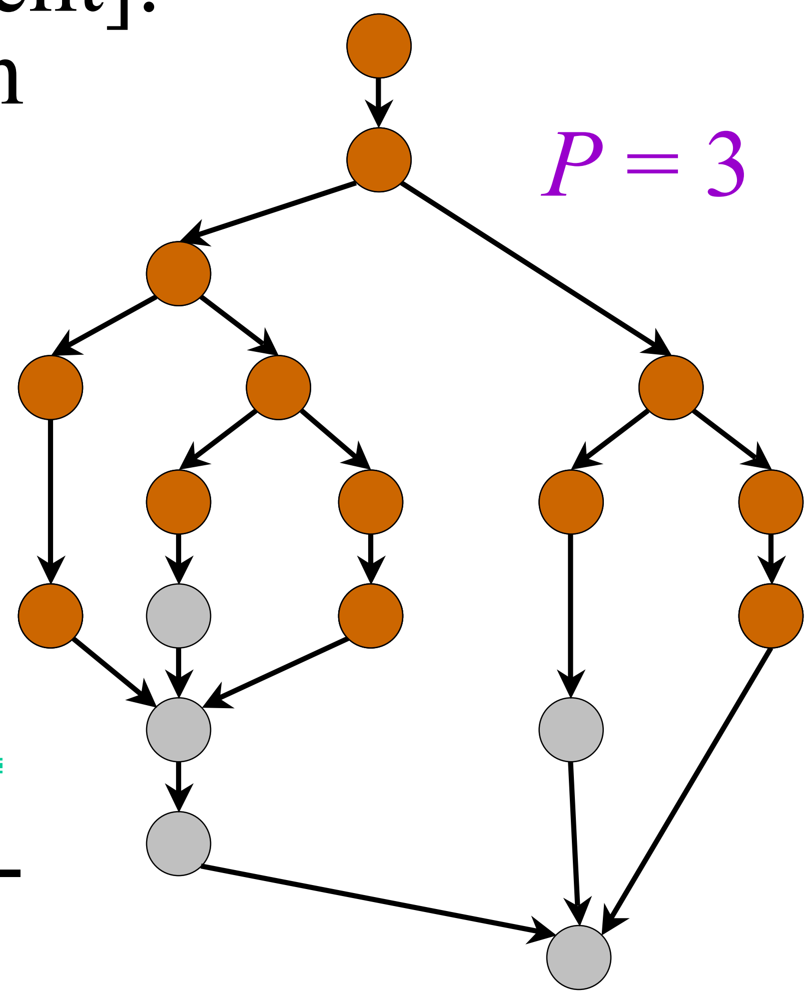
$P = 3$



Greedy Scheduling

Theorem [Graham & Brent]: There exists an execution with $T_P \le T_1/P + T_\infty$.

Proof. At each time step, if at least P tasks are ready, execute P of them. If fewer than P tasks are ready, execute all of them.



Corollary: Linear speed-up when $P \ll T_1/T_\infty$.

Cilk Performance

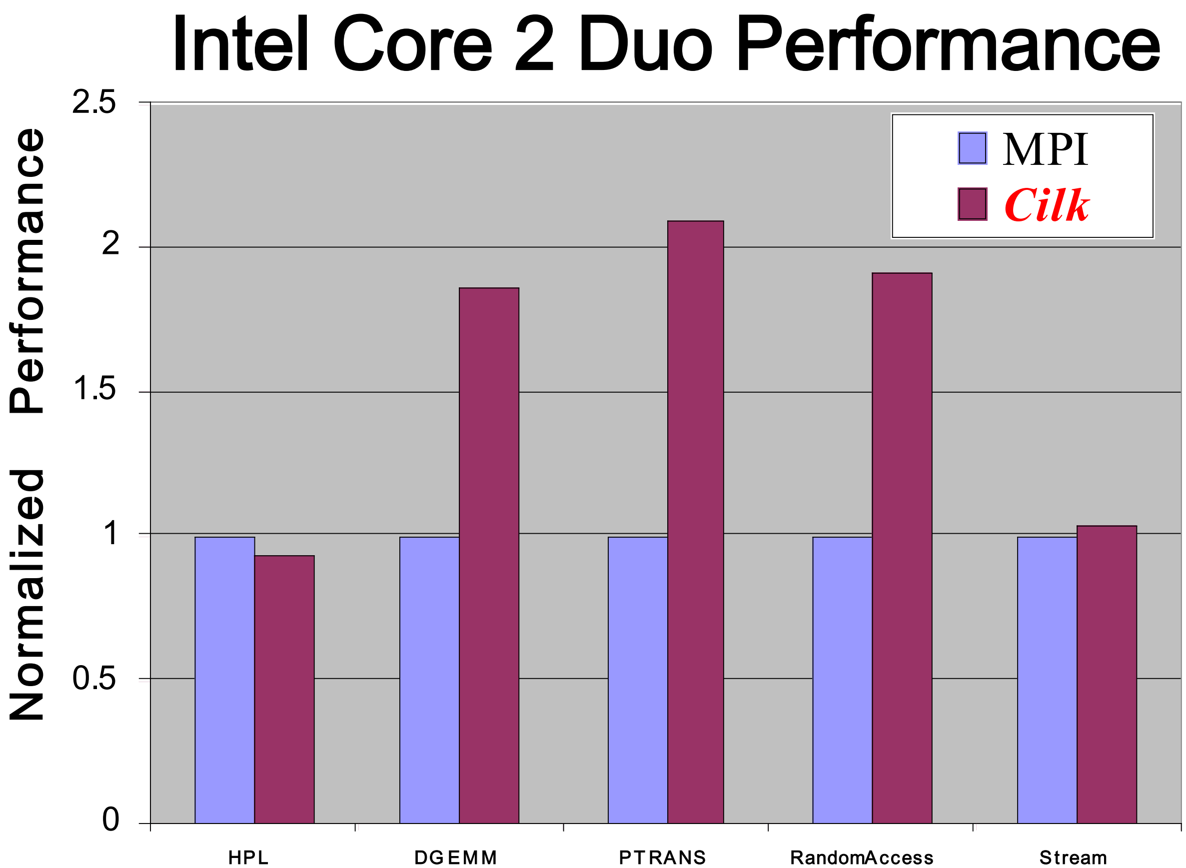
- *Cilk*'s "work-stealing" scheduler achieves
  - $T_P = T_1/P + O(T_\infty)$ expected time (provably);
  - $T_P \approx T_1/P + T_\infty$ time (empirically).
- Linear speedup if $P \ll T_1/T_\infty$ (parallelism).
- Instrumentation in *Cilk* allows the user to determine accurate measures of T_1 and T_∞.
- The average cost of a **`spawn`** in *Cilk* is only 2–6 times the cost of an ordinary C function call, depending on the platform.

Cilk Benchmarks (c. 1997)

Program	Size	T_1	T_∞	T_1/T_∞	T_1/T_S	T_8	T_1/T_8
`blockedmul`	1024	29.9	.0046	6783	1.05	4.29	7.0
`notempmul`	1024	29.7	.0156	1904	1.05	3.9	7.6
`strassen`	1024	20.2	.5662	36	1.01	3.54	5.7
`queens`	22	150.0	.0015	96898	0.99	18.8	8.0
`cilksort*`	4.1M	5.4	.0048	1125	1.21	0.9	6.0
`knapsack`	30	75.8	.0014	54143	1.03	9.5	8.0
`lu`	2048	155.8	.4161	374	1.02	20.3	7.7
`cholesky*`	1.02M	1427.0	3.4	420	1.25	208	6.9
`heat`	2M	62.3	.16	384	1.08	9.4	6.6
`fft`	1M	4.3	.002	2145	0.93	0.77	5.6
`barnes-hut`	65536	124.0	.15	853	1.02	16.5	7.5

All benchmarks were run on a Sun Enterprise 5000 SMP with 8 167-megahertz UltraSPARC processors. All times are in seconds, repeatable to within 10%.

HPC Challenge Benchmarks



Performance of the ***Cilk*** code rivals or exceeds that of the hand-coded MPI.

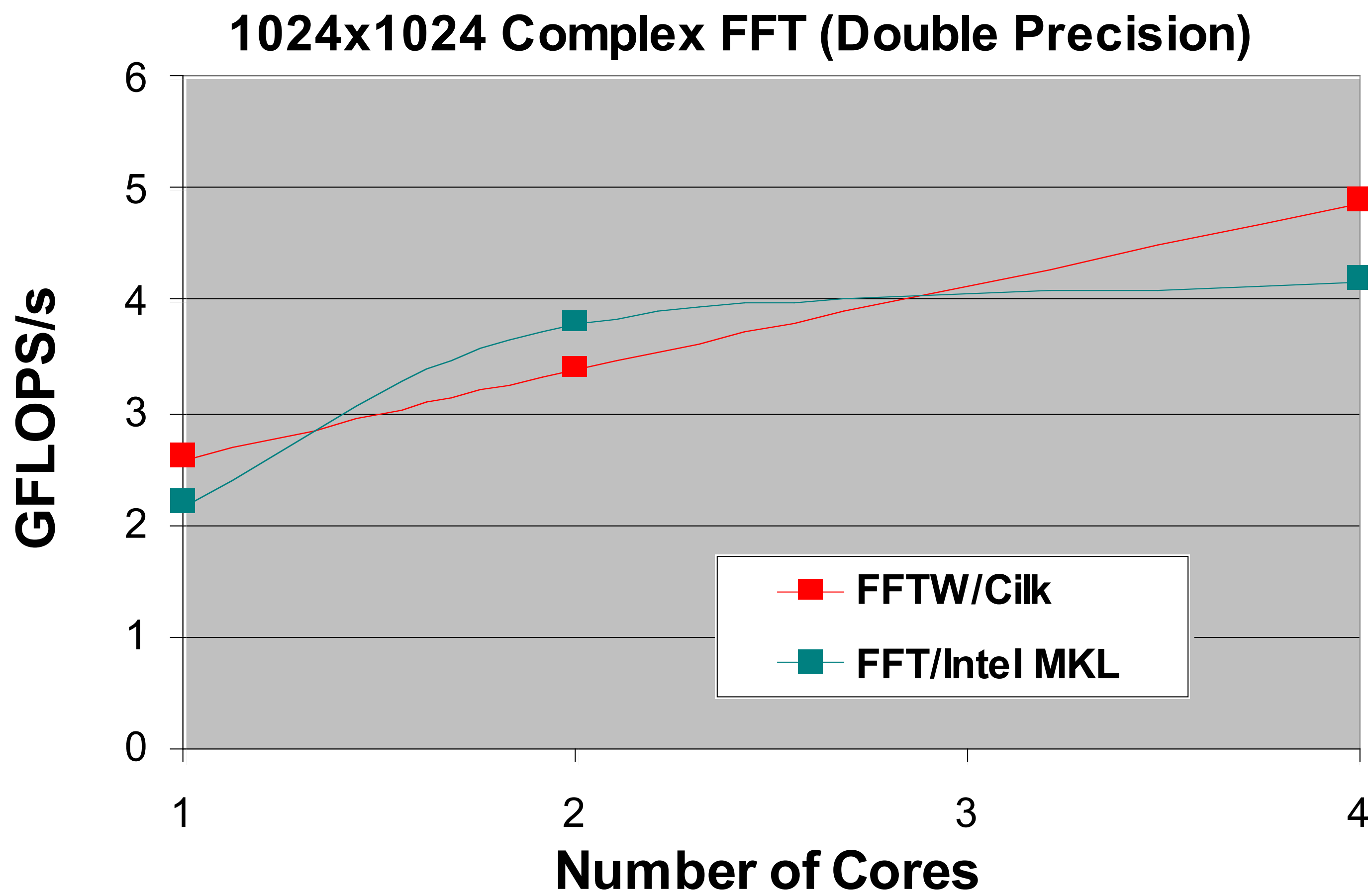
Cilk won the 2006 HPC Challenge Class 2 Award for ***Best Overall Productivity***. ***Cilk***ifying all six benchmarks required 137 keywords.

Productivity

Benchmark	*MPI SLoC*	*Cilk SLoC*	*Dist. to Multicore*
STREAM	658	58	11
PTRANS	2261	81	13
RandomAccess	1883	123	18
HPL	15608	348	41
DGEMM	184 †	97	19
FFTE	230	1747	35

† MPI DGEMM uses the HPL parallel matrix multiplication.

Fast Fourier Transform



Performance of ***Cilk***'s FFTW competes with Intel's hand-tuned MKL FFT.

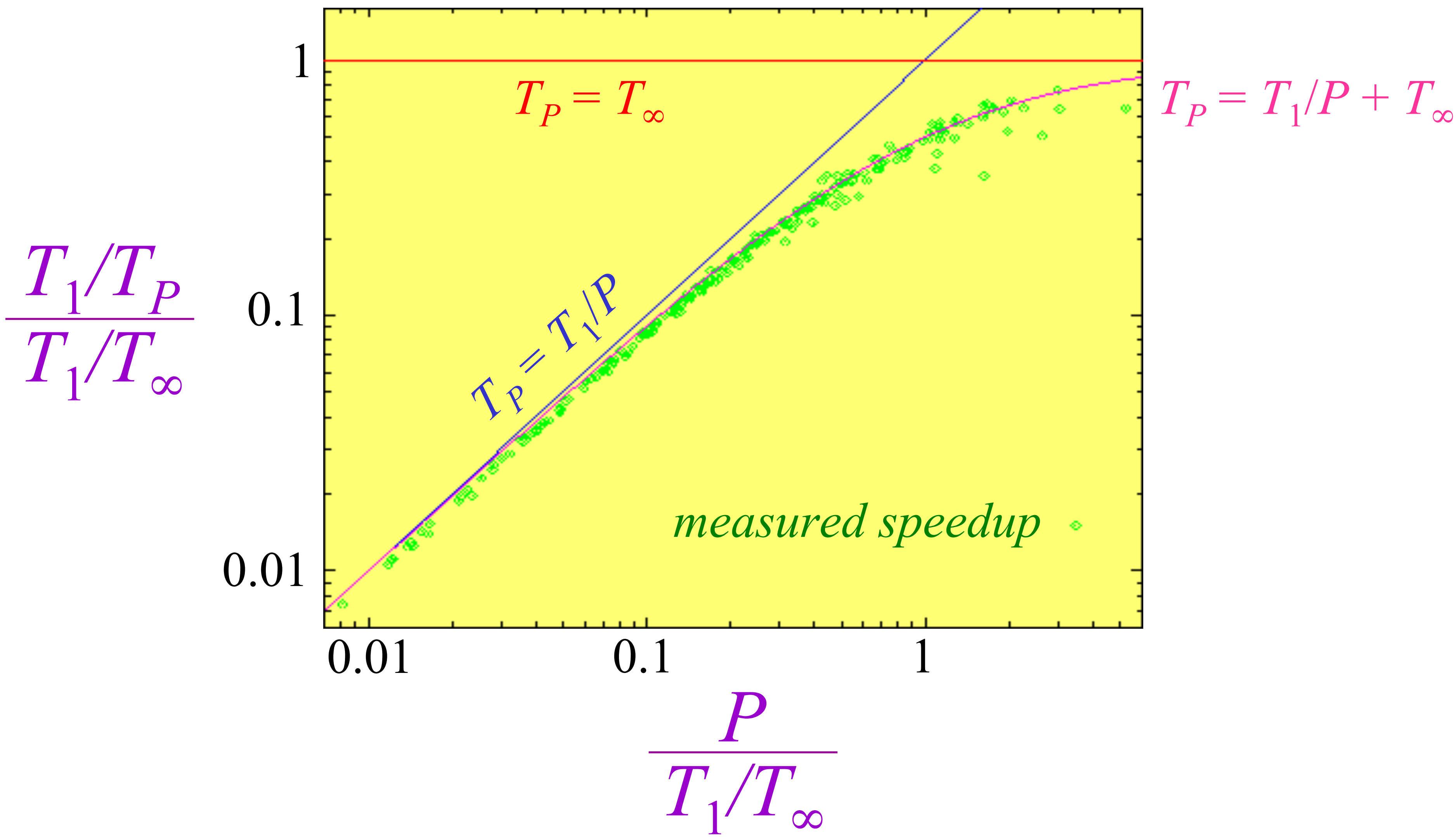
OUTLINE

- Overview
- *Cilk* Performance
- A Chess Lesson
- Alpha-Beta Search
- *Cilk*'s Scheduler
- Conclusion

Cilk Chess Programs

- ★*Socrates* placed 3rd in the 1994 International Computer Chess Championship running on NCSA's 512-node Connection Machine CM5.
- ★*Socrates 2.0* took 2nd place in the 1995 World Computer Chess Championship running on Sandia National Labs' 1824-node Intel Paragon.
- *Cilkchess* placed 1st in the 1996 Dutch Open running on a 12-processor Sun Enterprise 5000. It placed 2nd in 1997 and 1998 running on Boston University's 64-processor SGI Origin 2000.
- *Cilkchess* tied for 3rd in the 1999 WCCC running on NASA's 256-node SGI Origin 2000.

★Socrates Normalized Speedup



★Socrates Speedup Paradox

Original program

$T_{32} = 65$ seconds

Proposed program

$T'_{32} = 40$ seconds

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

$T_1 = 2048$ seconds
$T_\infty = 1$ second

$T_{32} = 2048/32 + 1$
$= 65$ seconds

$T_{512} = 2048/512 + 1$
$= 5$ seconds

$T'_1 = 1024$ seconds
$T'_\infty = 8$ seconds

$T'_{32} = 1024/32 + 8$
$= 40$ seconds

$T'_{512} = 1024/512 + 8$
$= 10$ seconds

OUTLINE

- Overview
- *Cilk* Performance
- A Chess Lesson
- **Alpha-Beta Search**
- ***Cilk*'s Scheduler**
- **Conclusion**

Cilk's Inlet Mechanism

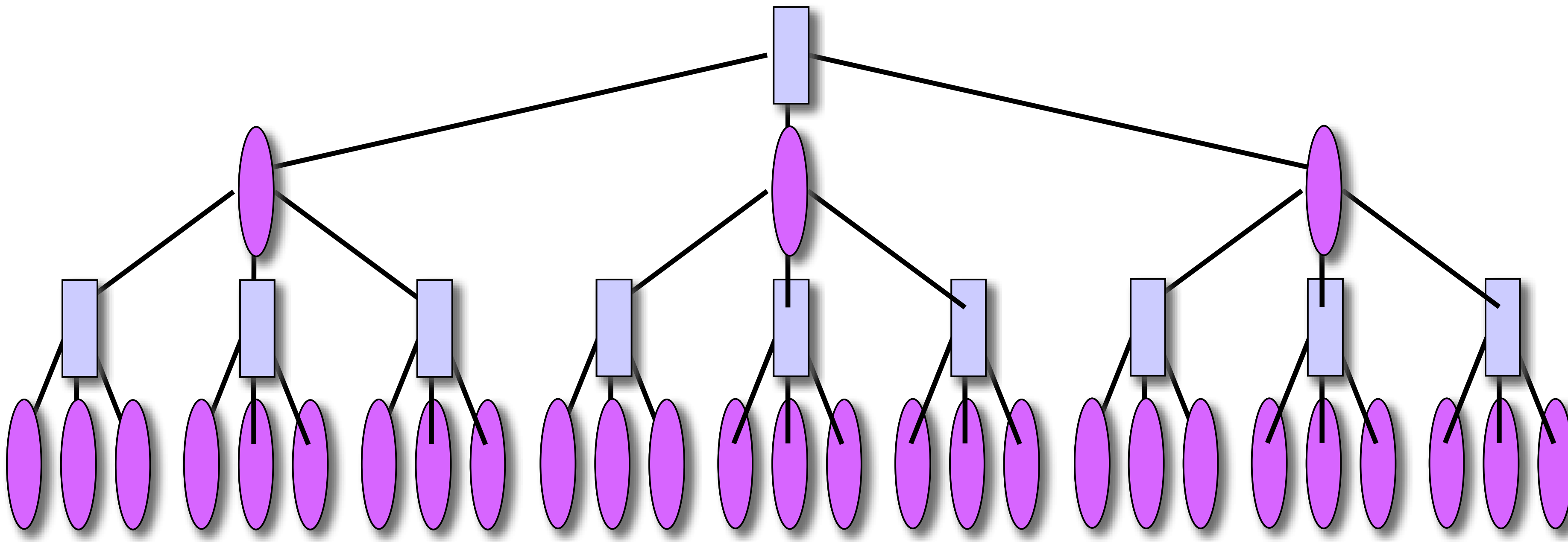
Returned values can be incorporated into the parent frame using a delayed internal function called an ***inlet***.

```
int max = -INF;
int max_index = 0;

inlet void update_max ( int x, int index )
{
  if (x > max)
  {
    max = x;
    max_index = index;
  }
}
:
for (i=0; i<1000000; ++i)
{
  update_max ( spawn bar(i), i );
}
sync;
```

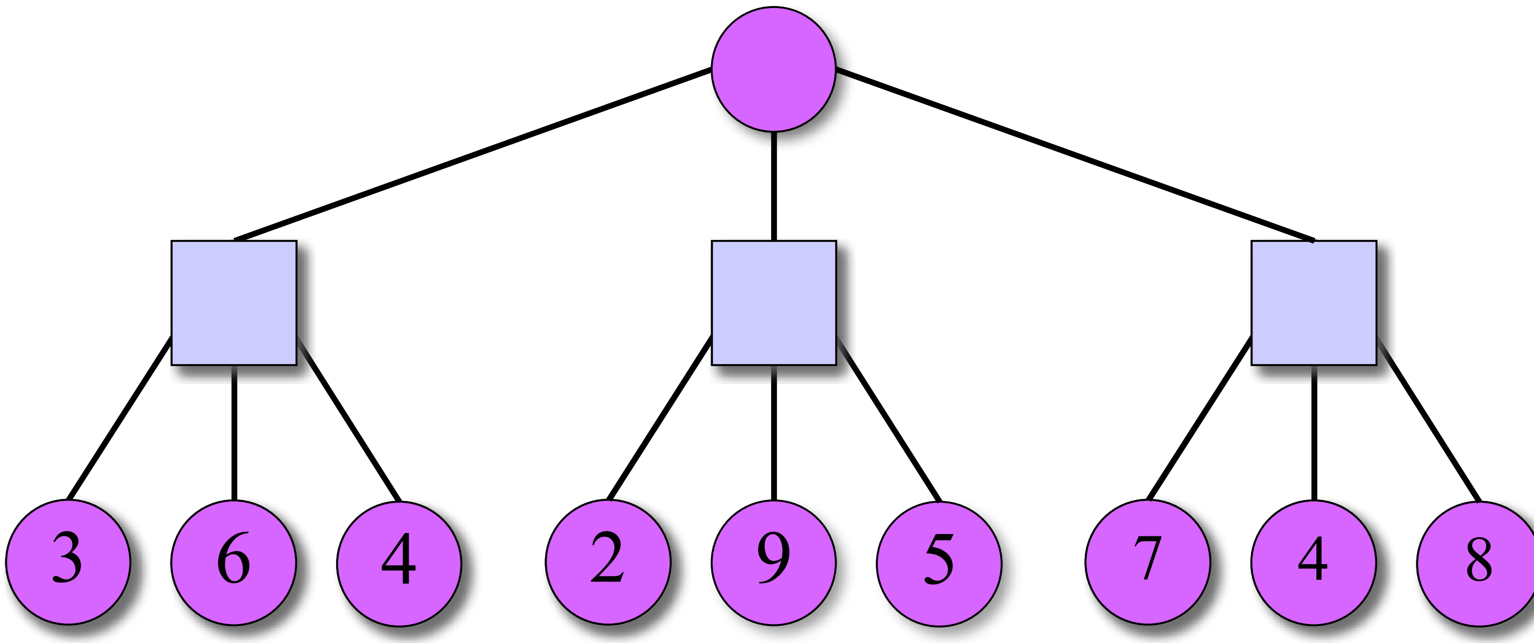
Cilk provides ***implicit atomicity*** among the threads belonging to the same frame.

Min-Max Search



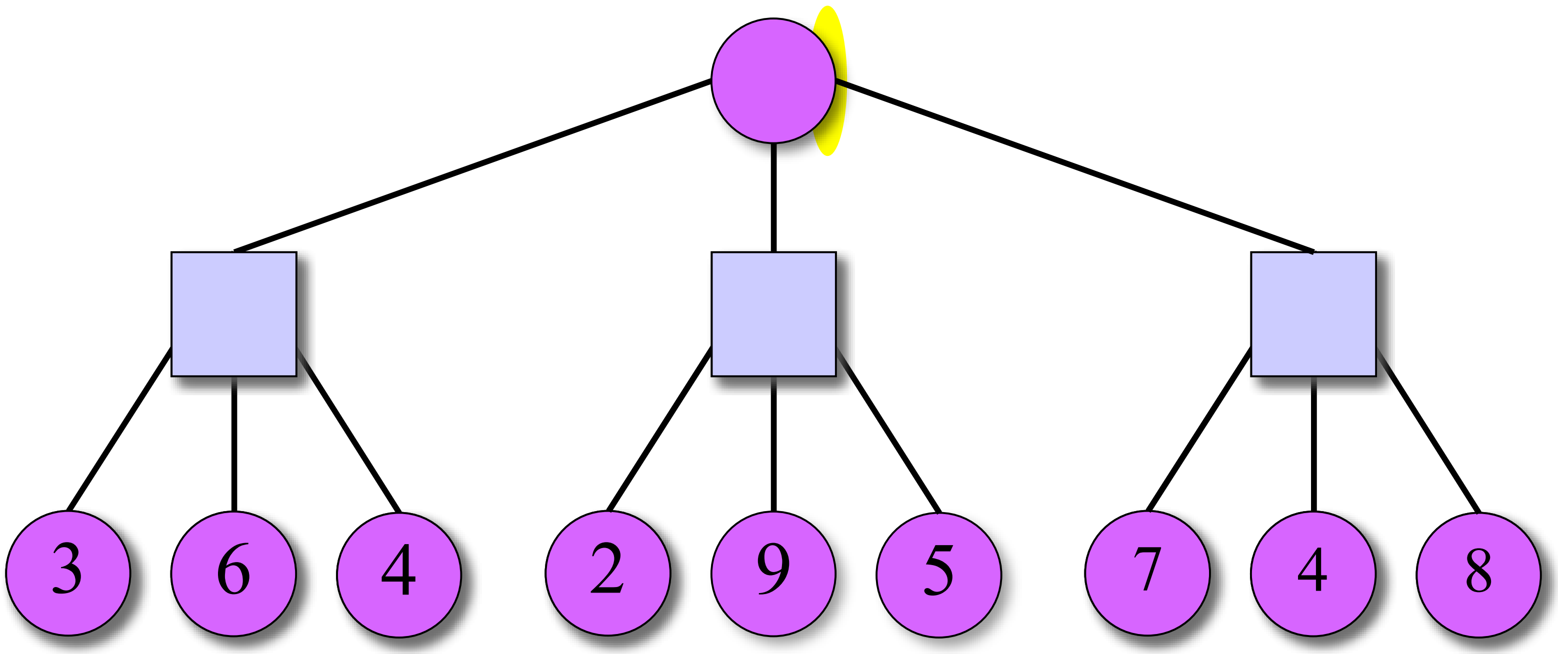
- Two players: MAX ■ and MIN ●.
- The game tree represents all moves from the current position within a given search depth.
- At leaves, apply a static evaluation function.
- MAX chooses the maximum score among its children.
- MIN chooses the minimum score among its children.

Alpha-Beta Pruning



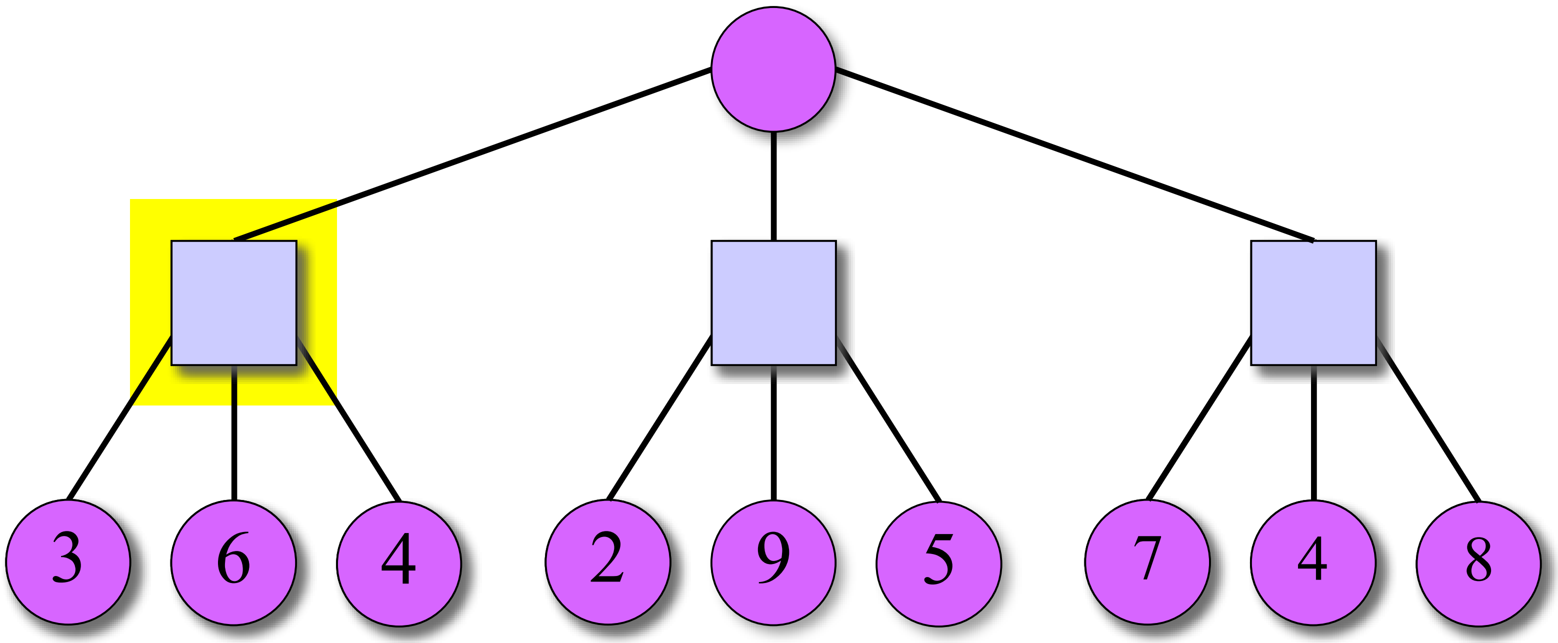
IDEA: If MAX ■ discovers a move so good that MIN ● would never allow that position, MAX's other children need not be searched — ***beta cutoff***.

Alpha-Beta Pruning



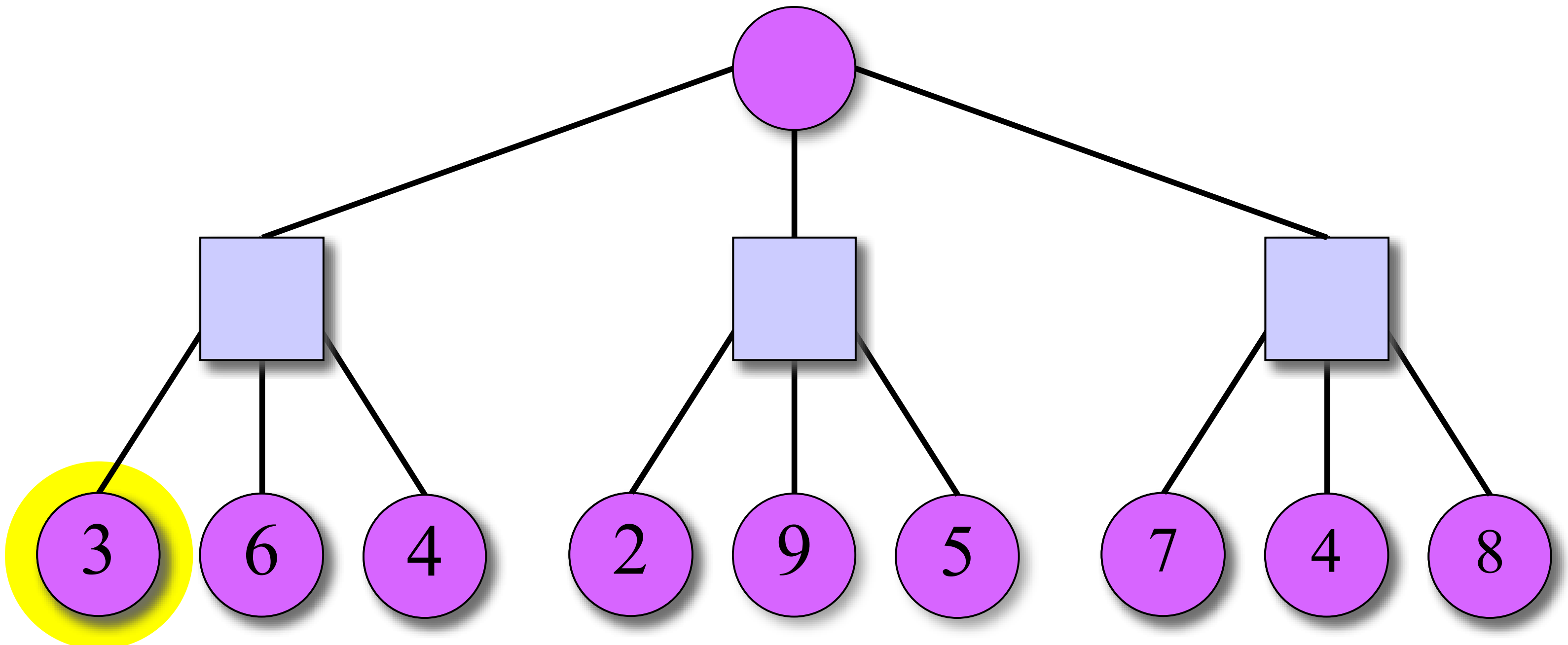
IDEA: If MAX ■ discovers a move so good that MIN ● would never allow that position, MAX's other children need not be searched — ***beta cutoff***.

Alpha-Beta Pruning



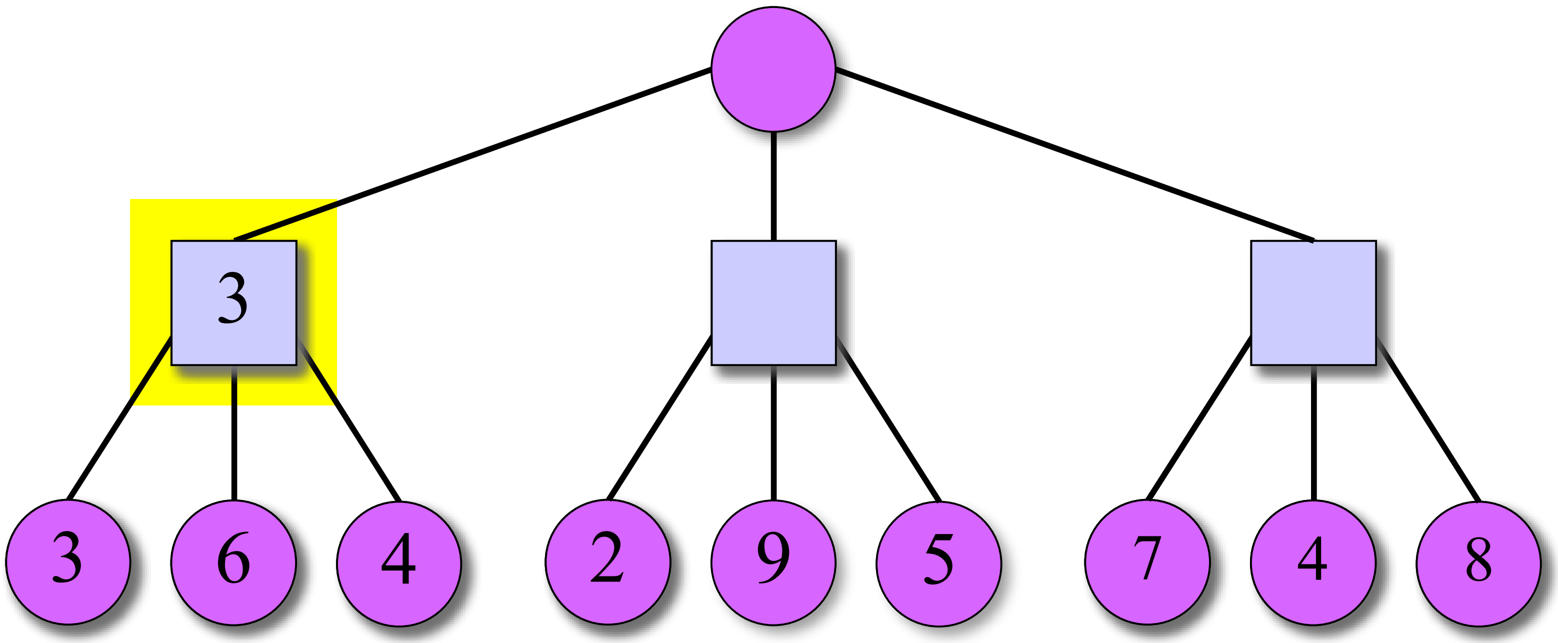
IDEA: If MAX ◼ discovers a move so good that MIN ● would never allow that position, MAX's other children need not be searched — ***beta cutoff***.

Alpha-Beta Pruning



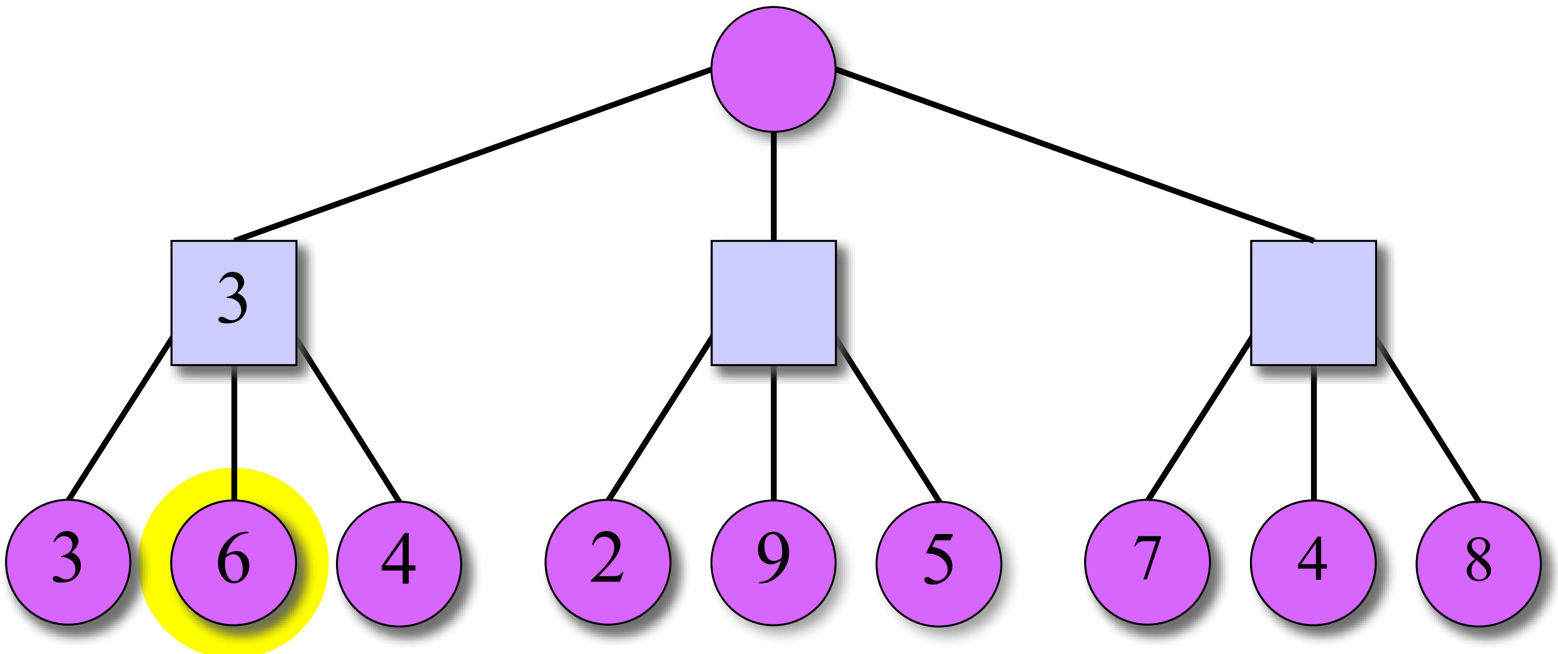
IDEA: If MAX ■ discovers a move so good that MIN ● would never allow that position, MAX's other children need not be searched — ***beta cutoff***.

Alpha-Beta Pruning



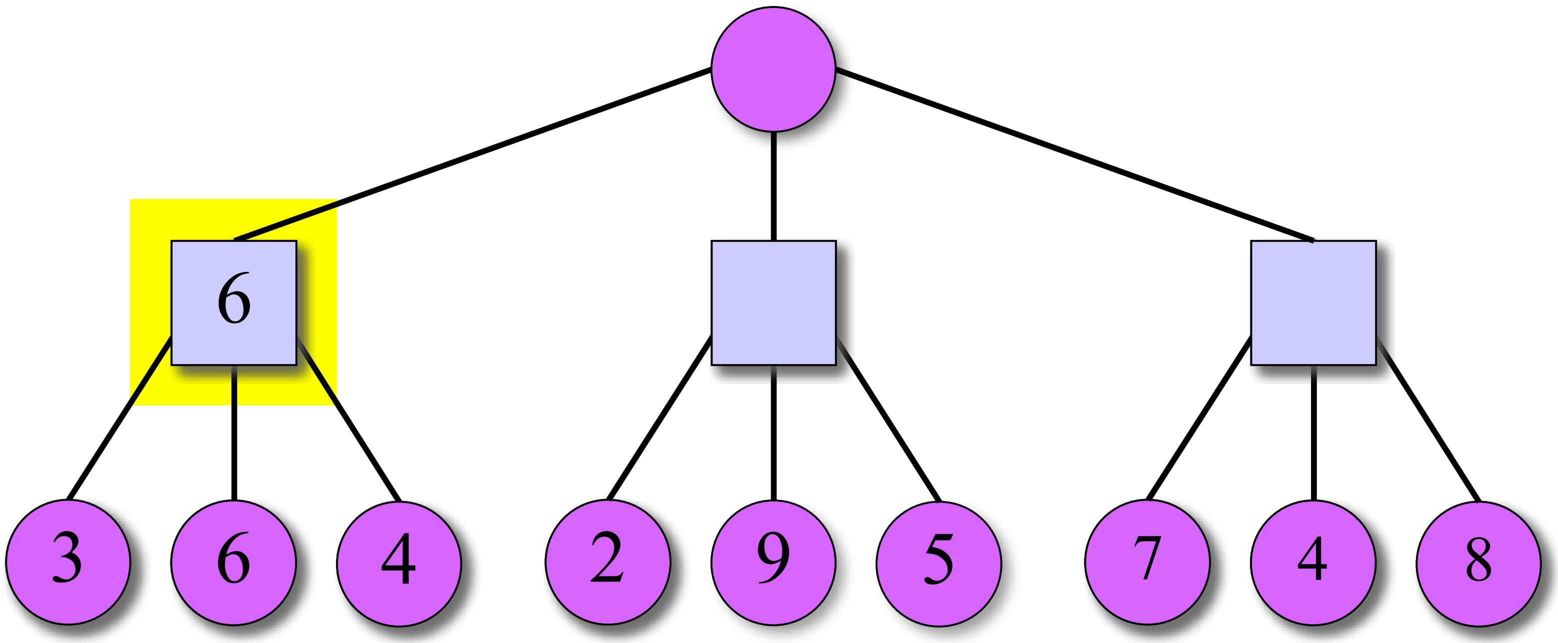
IDEA: If MAX ■ discovers a move so good that MIN ● would never allow that position, MAX's other children need not be searched — ***beta cutoff***.

Alpha-Beta Pruning



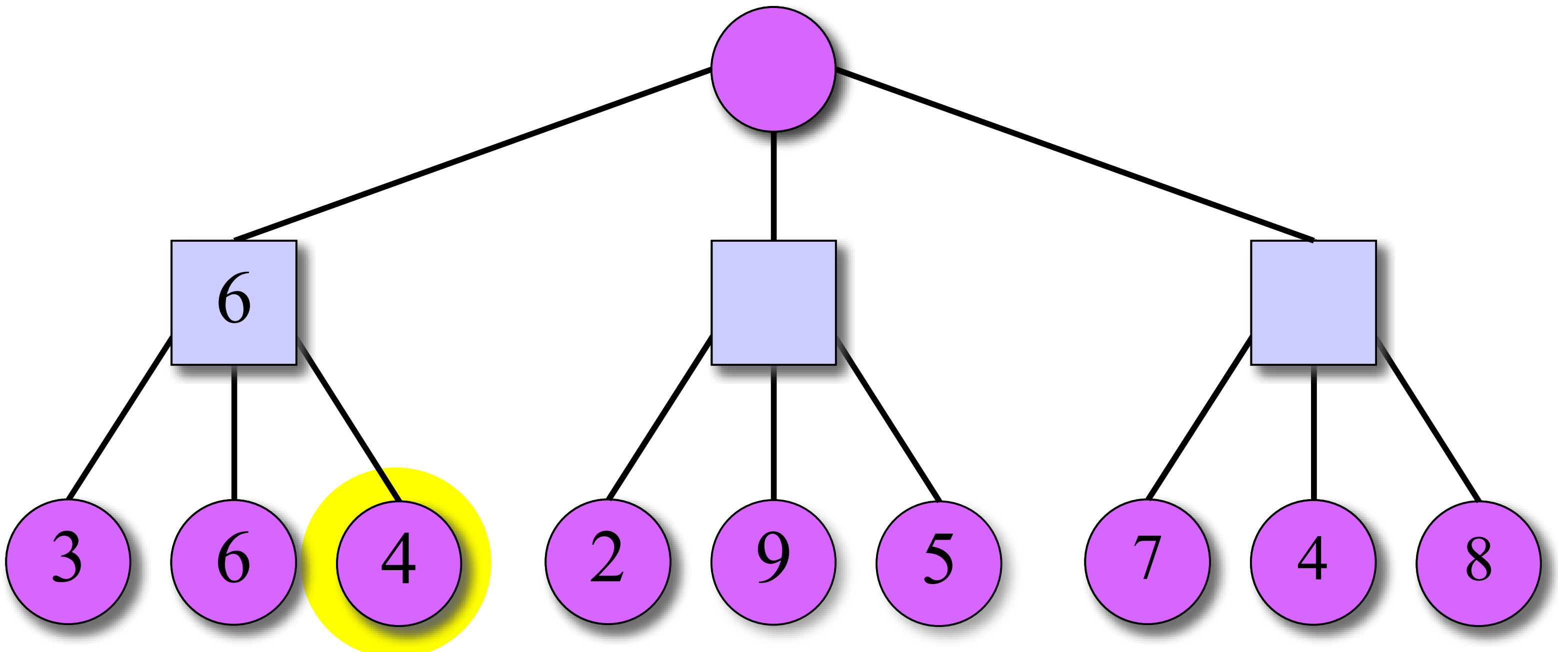
IDEA: If MAX ■ discovers a move so good that MIN ● would never allow that position, MAX's other children need not be searched — ***beta cutoff***.

Alpha-Beta Pruning



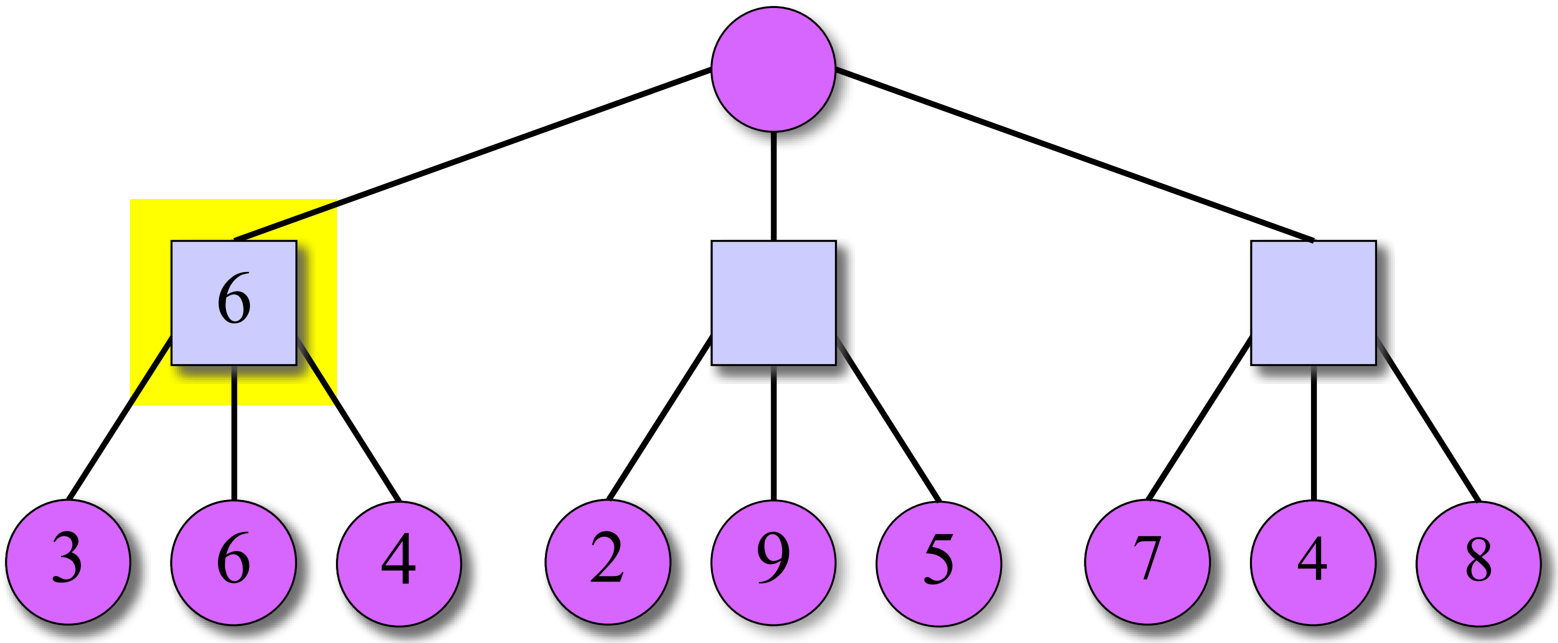
IDEA: If MAX ■ discovers a move so good that MIN ● would never allow that position, MAX's other children need not be searched — ***beta cutoff***.

Alpha-Beta Pruning



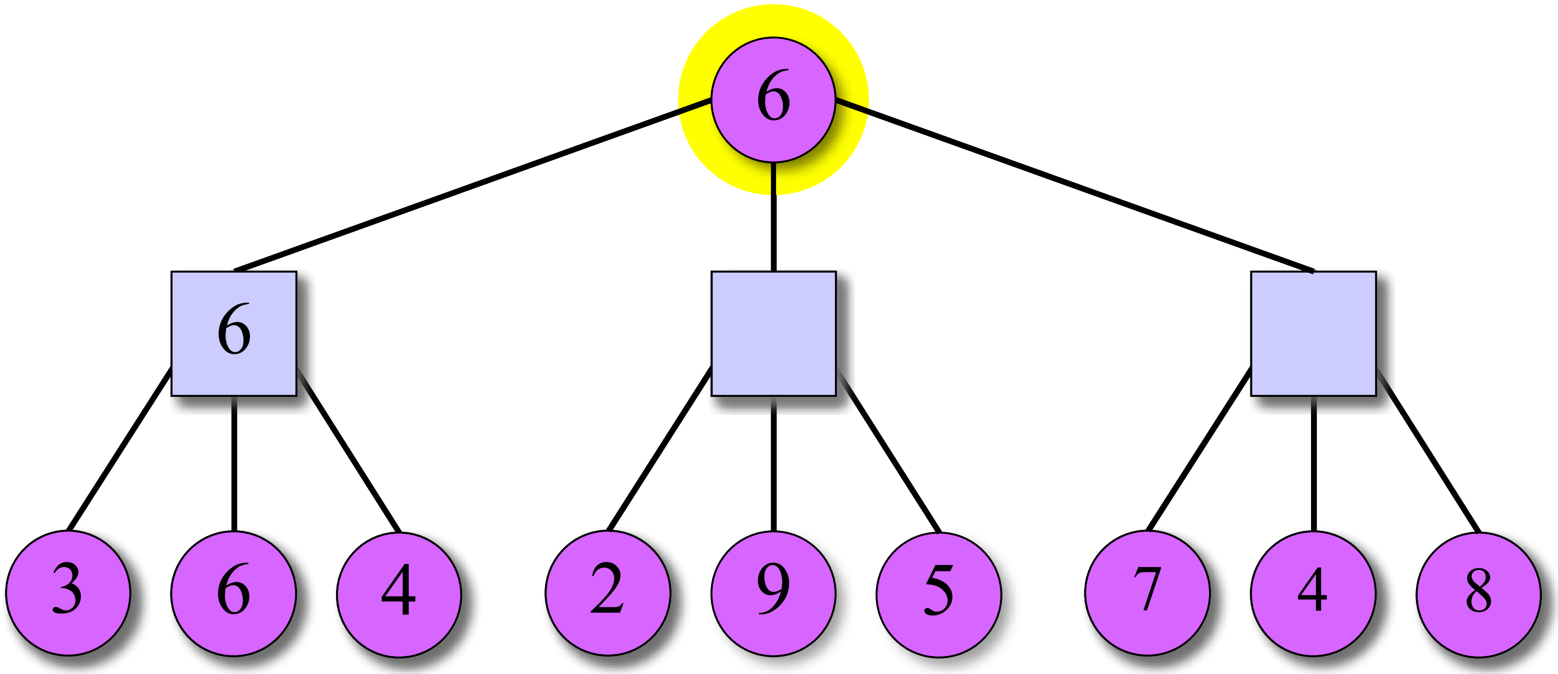
IDEA: If MAX ■ discovers a move so good that MIN ● would never allow that position, MAX's other children need not be searched — ***beta cutoff***.

Alpha-Beta Pruning



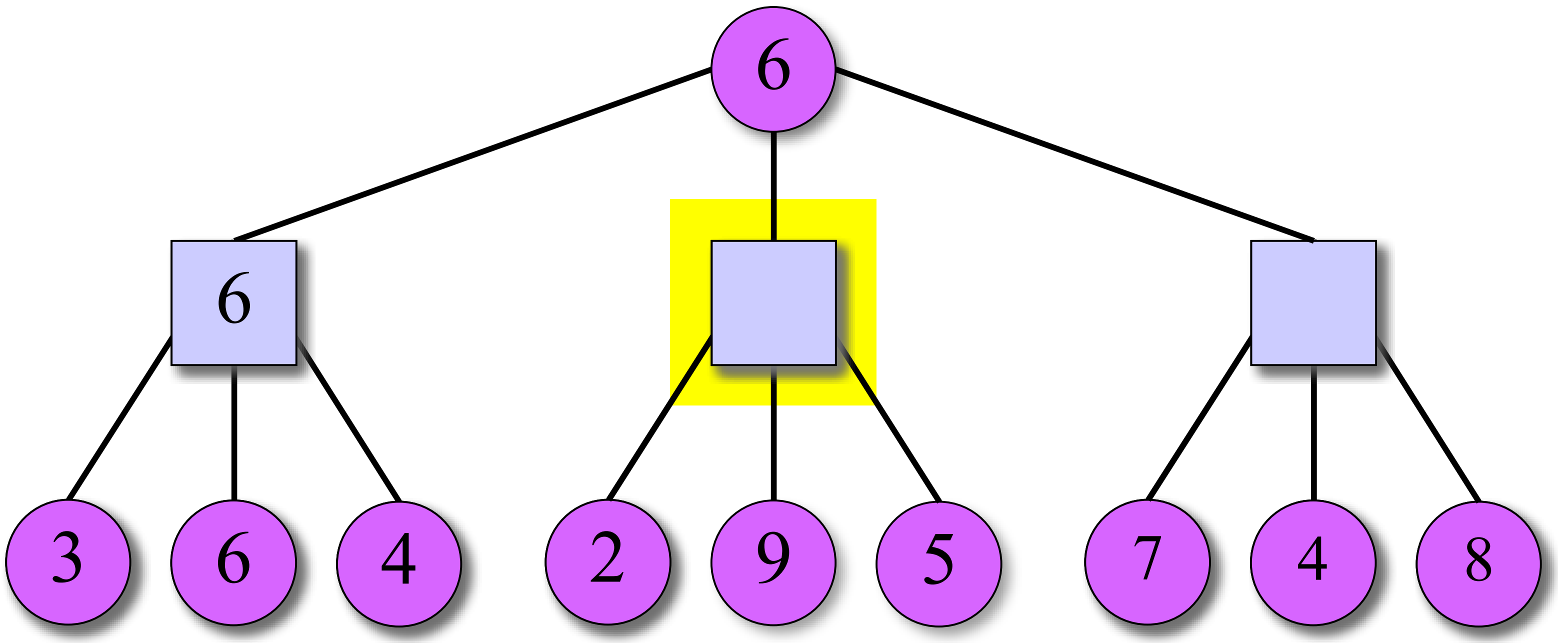
IDEA: If MAX ■ discovers a move so good that MIN ● would never allow that position, MAX's other children need not be searched — ***beta cutoff***.

Alpha-Beta Pruning



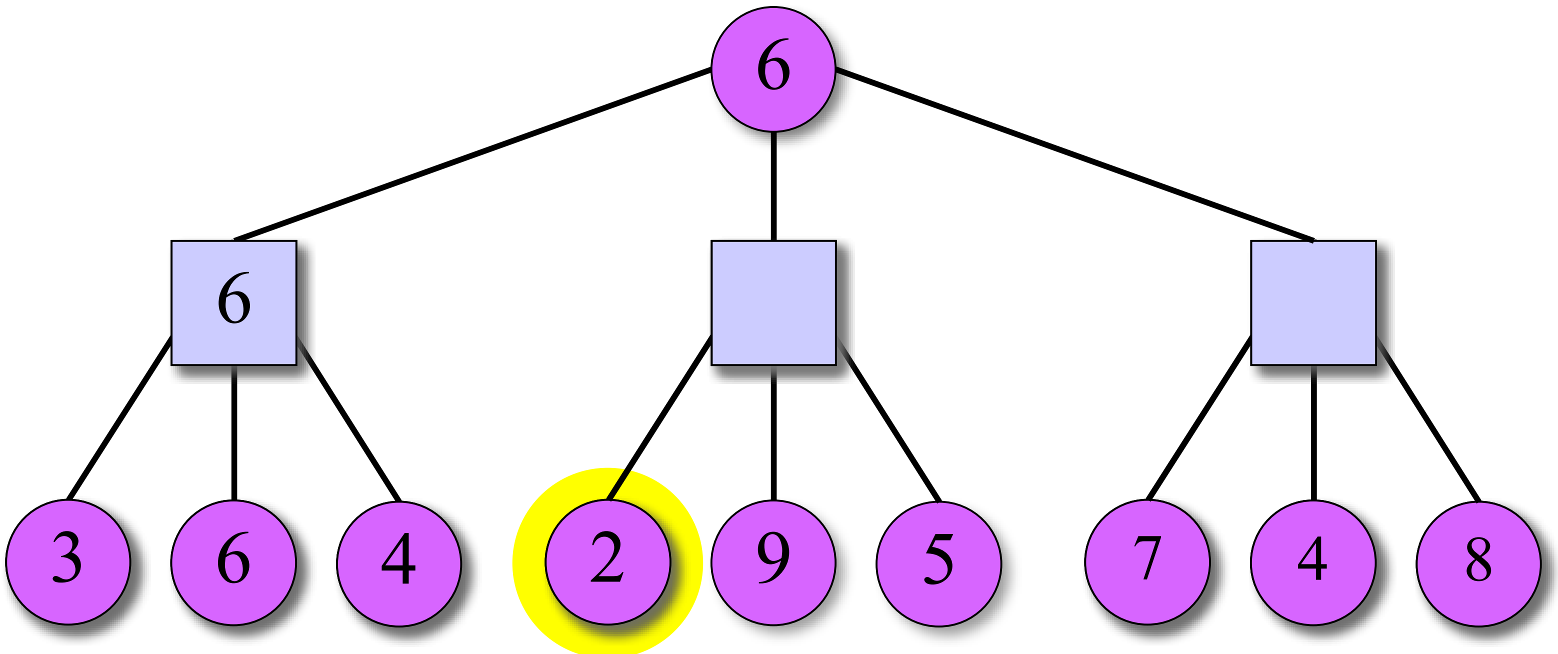
IDEA: If MAX ■ discovers a move so good that MIN ● would never allow that position, MAX's other children need not be searched — ***beta cutoff***.

Alpha-Beta Pruning



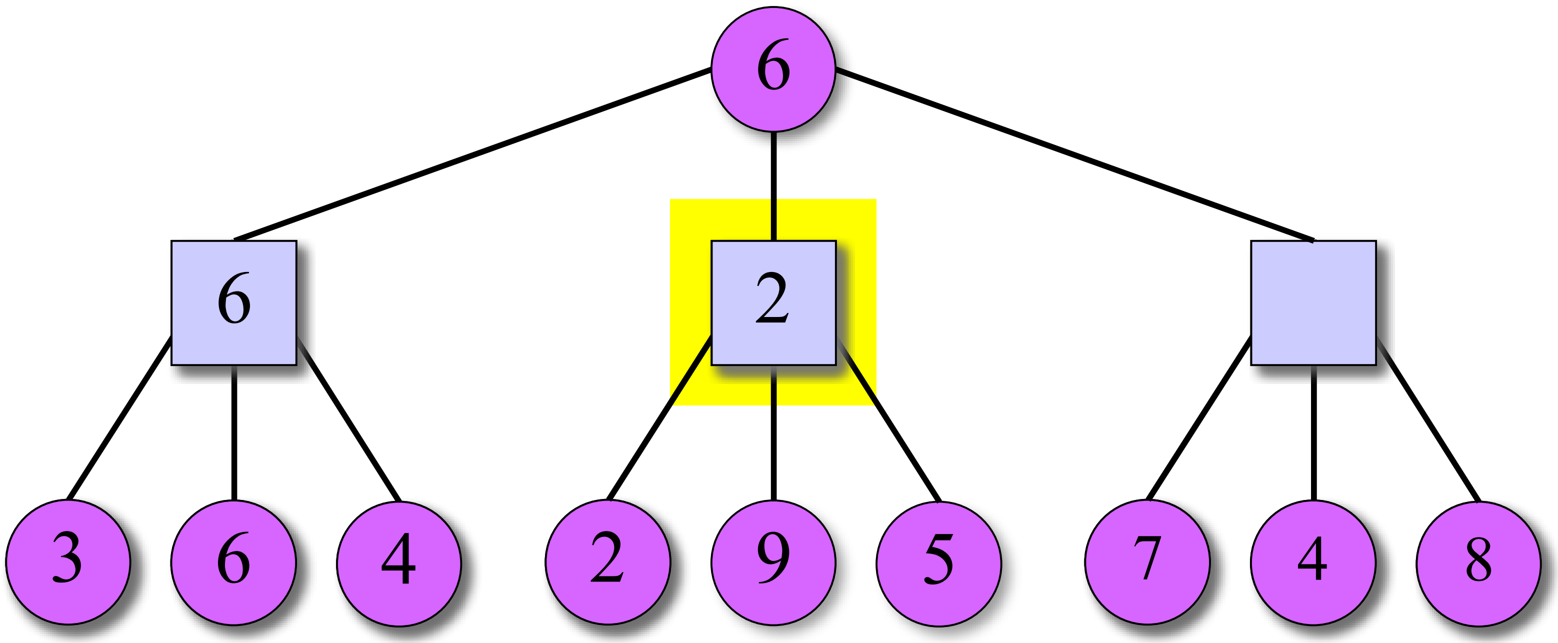
IDEA: If MAX discovers a move so good that MIN would never allow that position, MAX's other children need not be searched — ***beta cutoff***.

Alpha-Beta Pruning



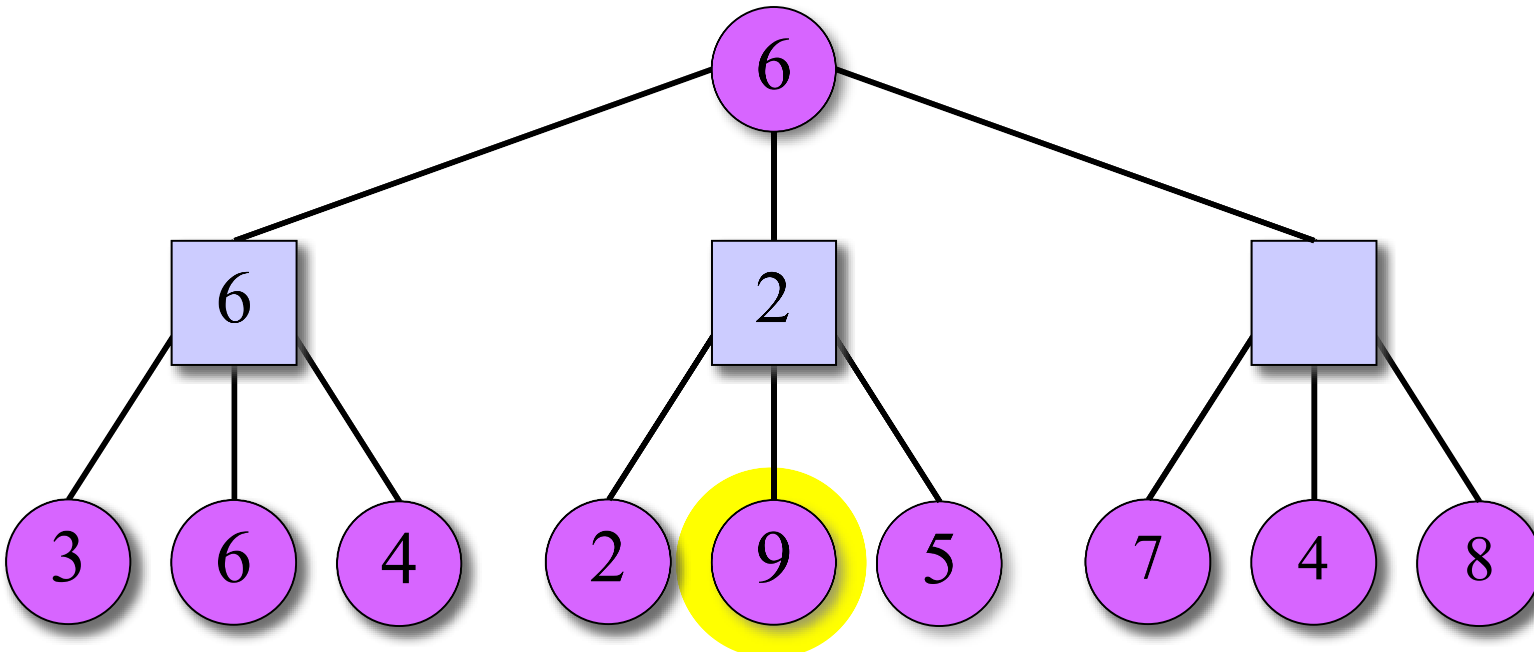
Idea: If MAX ■ discovers a move so good that MIN ● would never allow that position, MAX's other children need not be searched — ***beta cutoff***.

Alpha-Beta Pruning



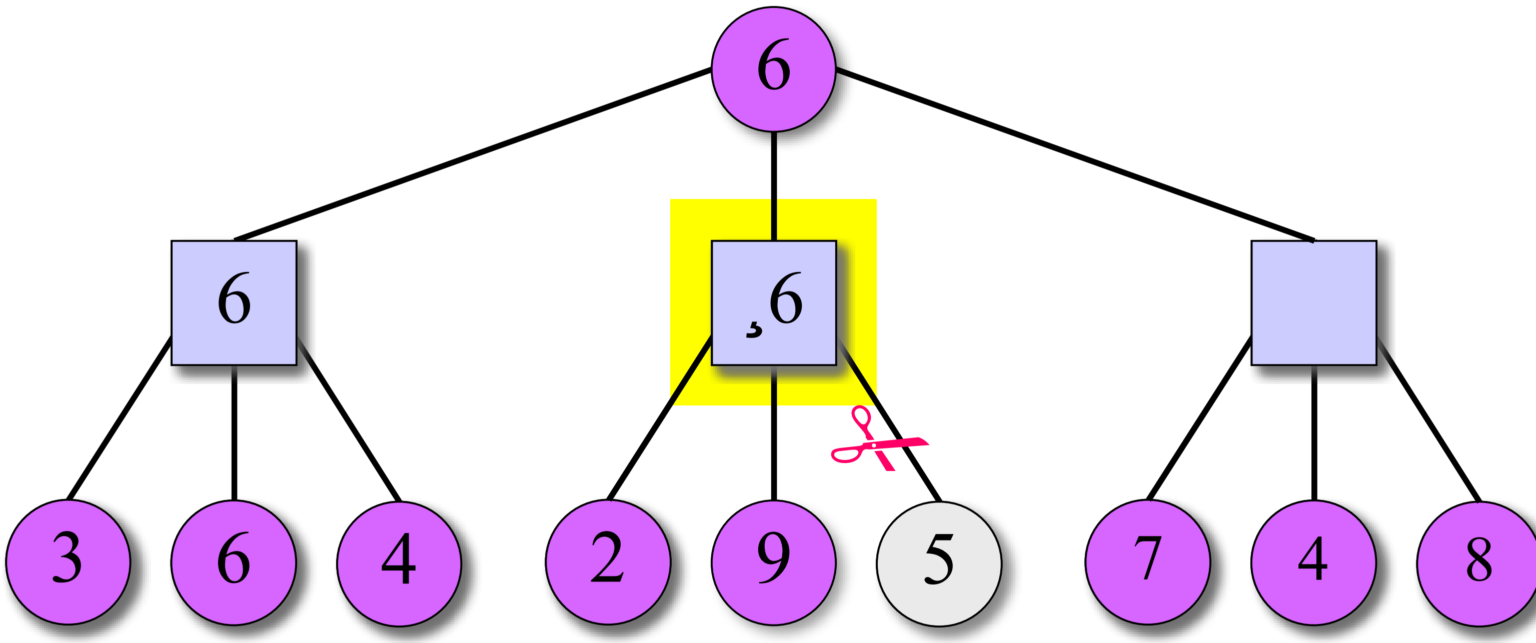
IDEA: If MAX ■ discovers a move so good that MIN ● would never allow that position, MAX's other children need not be searched — ***beta cutoff***.

Alpha-Beta Pruning



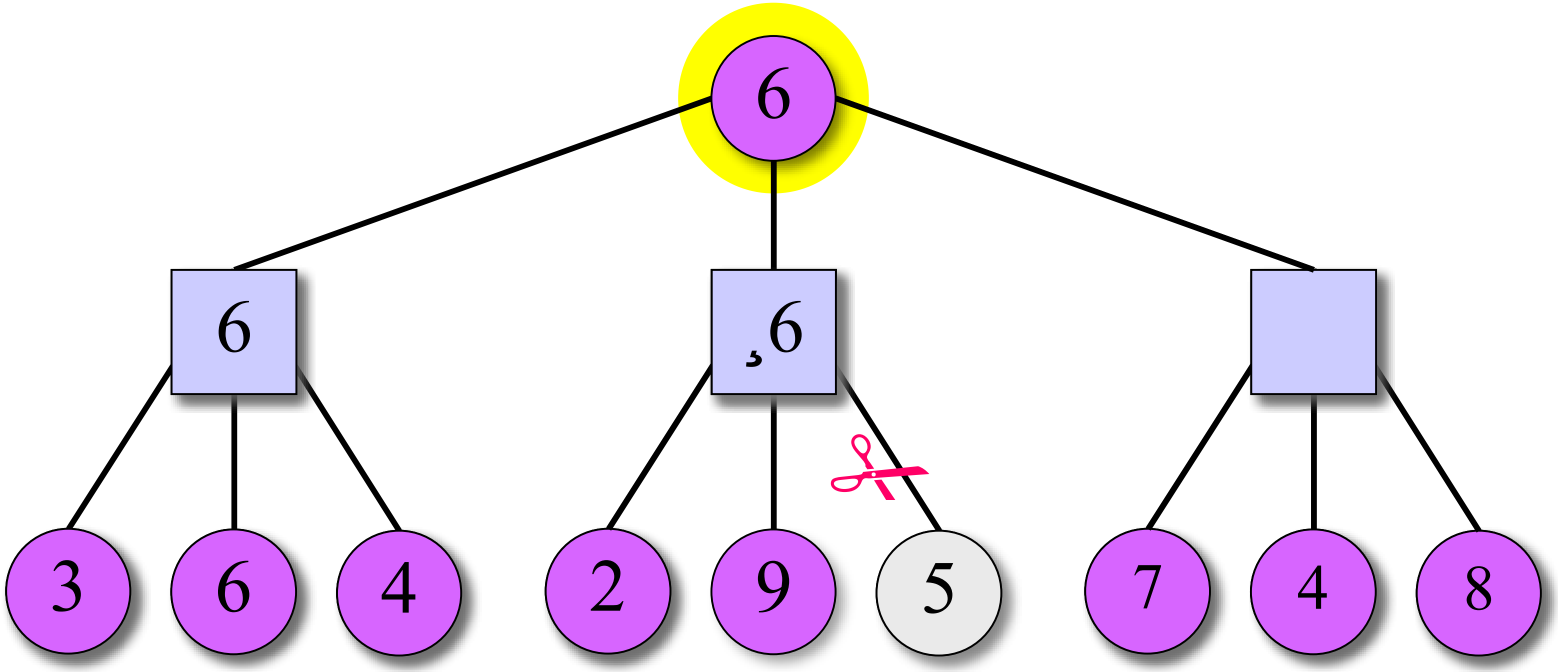
IDEA: If MAX ■ discovers a move so good that MIN ● would never allow that position, MAX's other children need not be searched — ***beta cutoff***.

Alpha-Beta Pruning



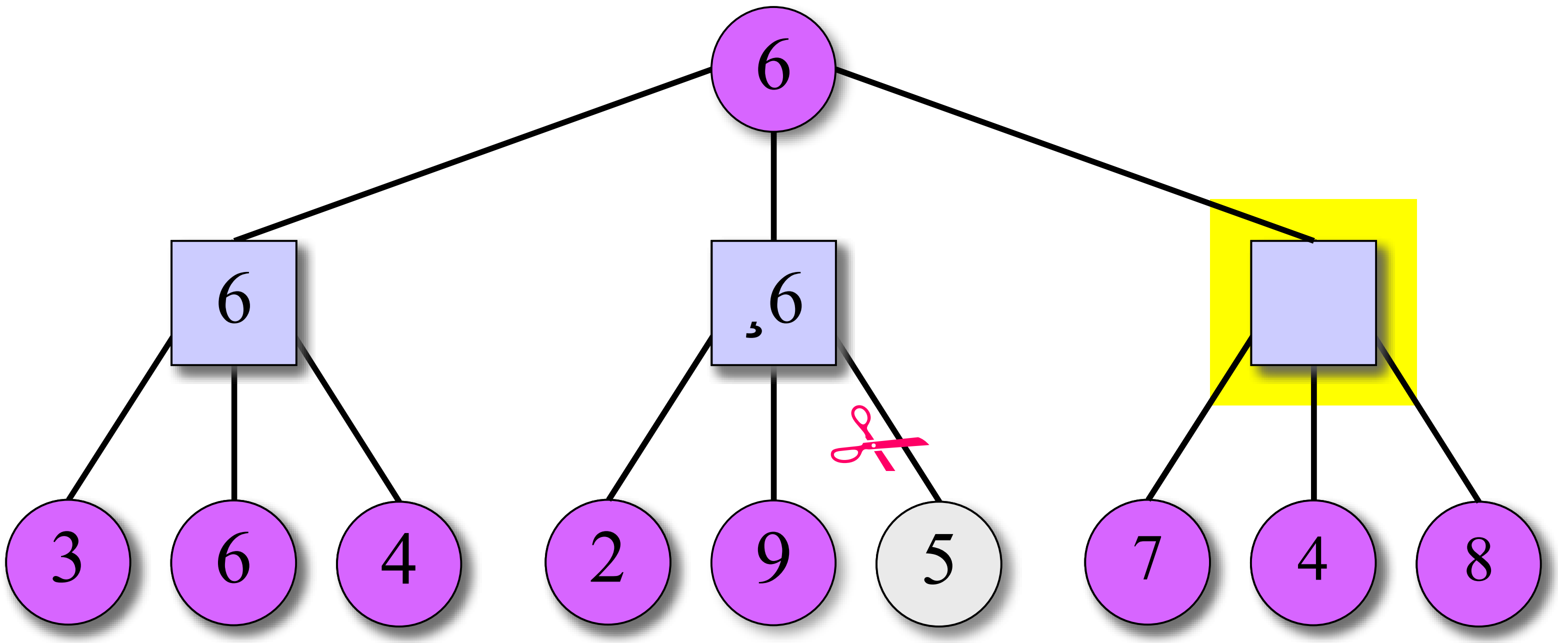
IDEA: If MAX ■ discovers a move so good that MIN ● would never allow that position, MAX's other children need not be searched — ***beta cutoff***.

Alpha-Beta Pruning



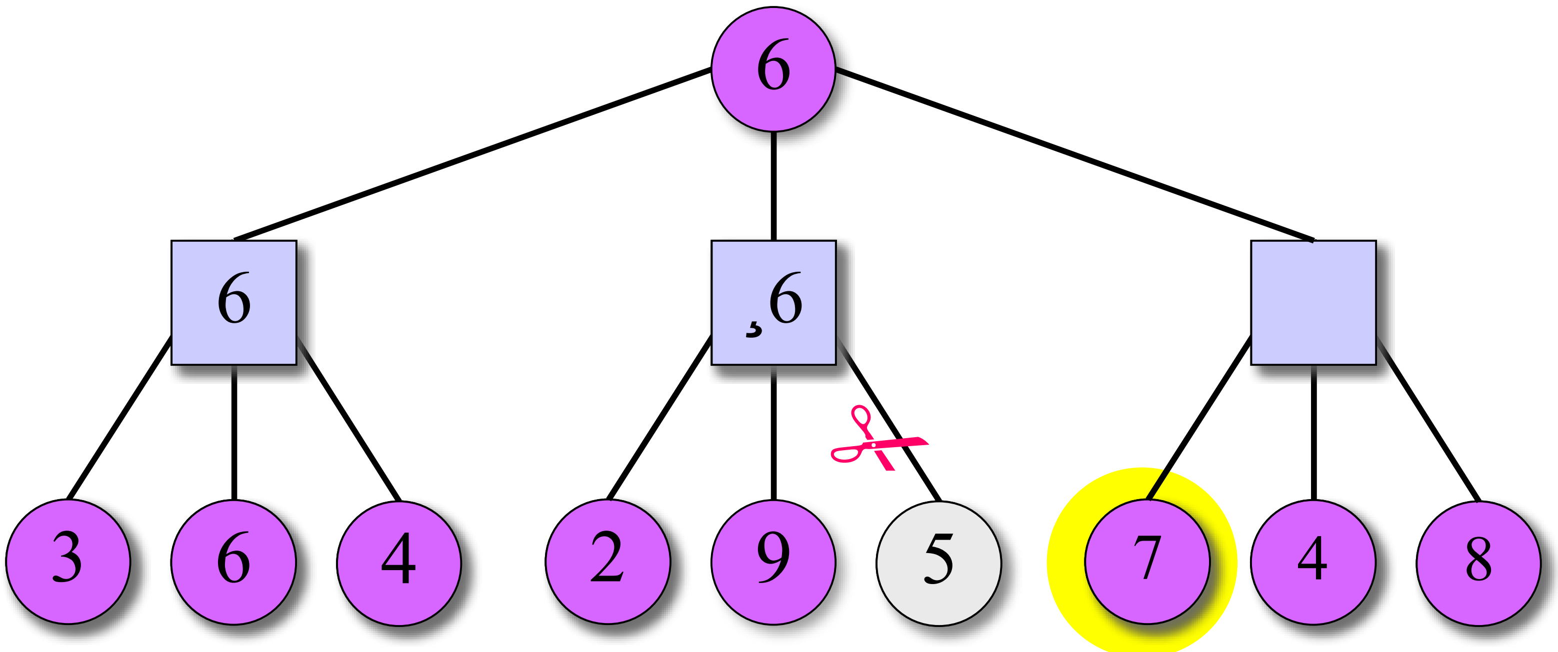
IDEA: If MAX discovers a move so good that MIN would never allow that position, MAX's other children need not be searched — ***beta cutoff***.

Alpha-Beta Pruning



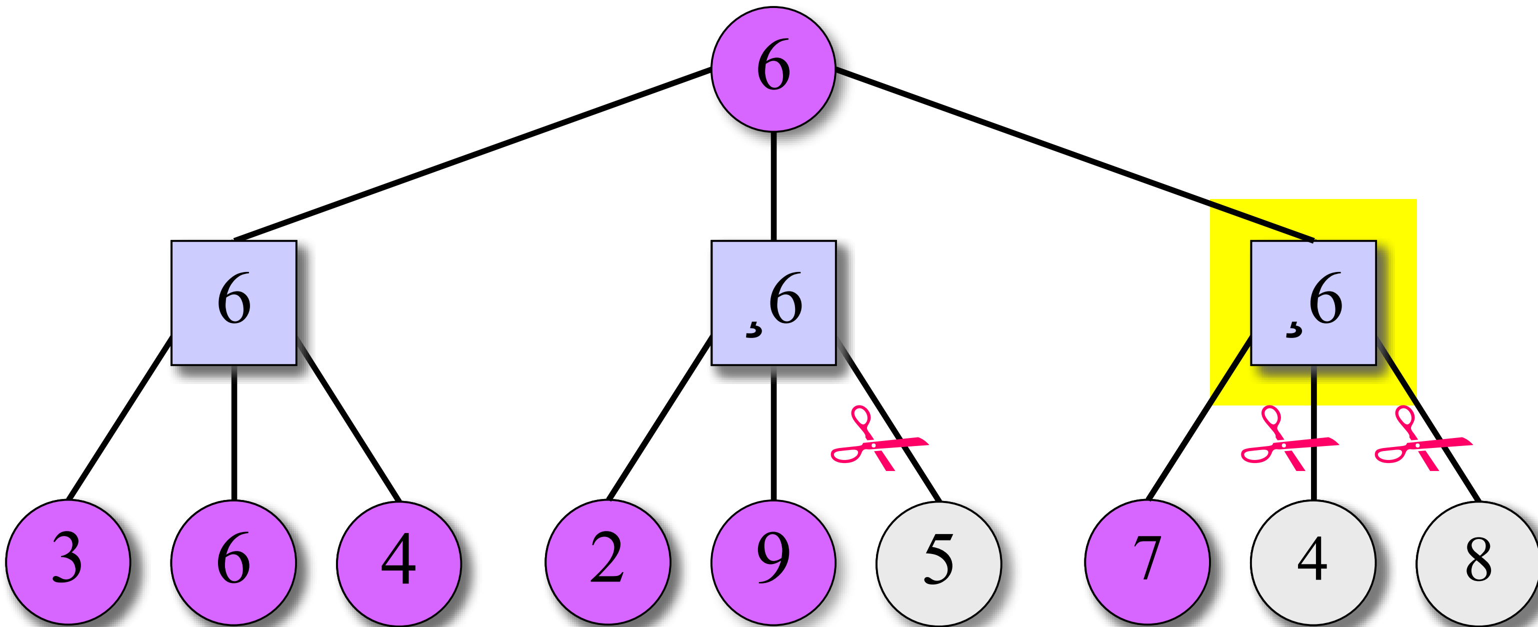
IDEA: If MAX discovers a move so good that MIN would never allow that position, MAX's other children need not be searched — ***beta cutoff***.

Alpha-Beta Pruning



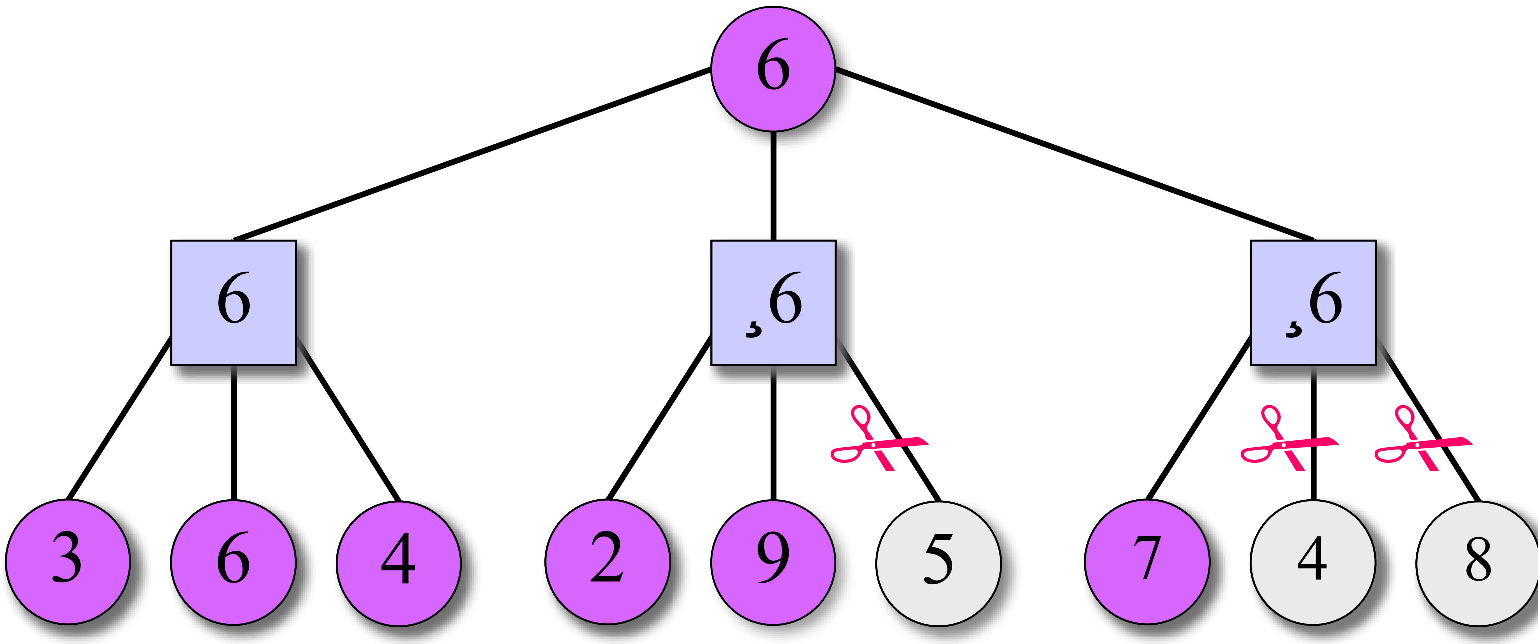
IDEA: If MAX discovers a move so good that MIN would never allow that position, MAX's other children need not be searched — ***beta cutoff***.

Alpha-Beta Pruning



IDEA: If MAX discovers a move so good that MIN would never allow that position, MAX's other children need not be searched — ***beta cutoff***.

Alpha-Beta Pruning

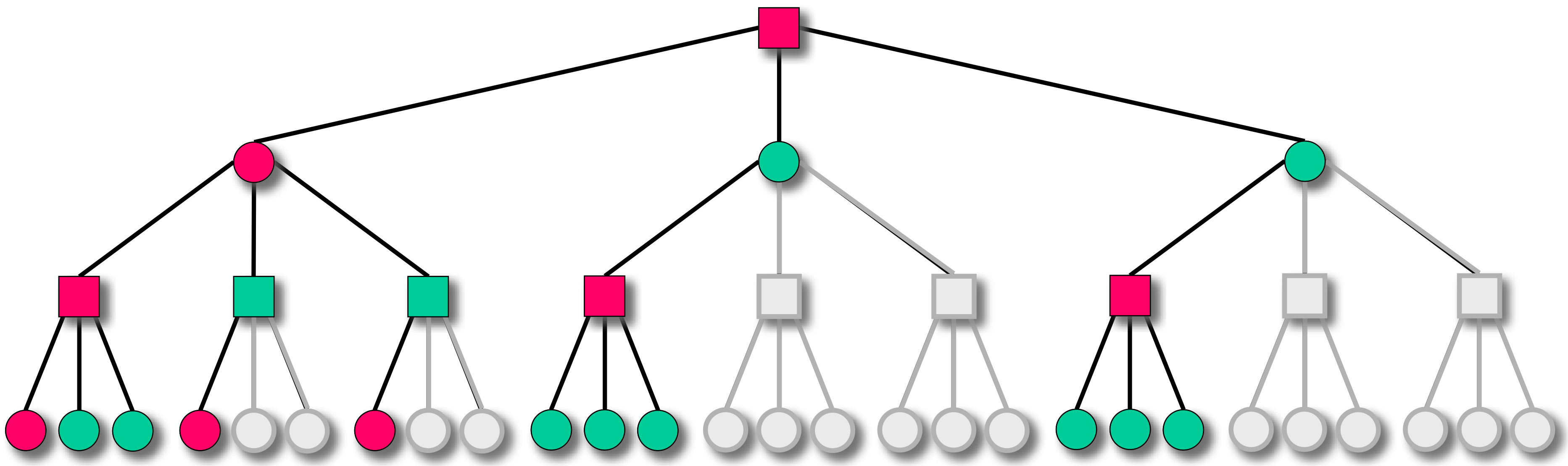


IDEA: If MAX ■ discovers a move so good that MIN ● would never allow that position, MAX's other children need not be searched — ***beta cutoff***.

Unfortunately, this heuristic is inherently serial.

Parallel Min-Max Search

OBSERVATION: In a best-ordered tree, the degree of every internal node is either 1 or ***maximal***.



IDEA: [Feldman-Mysliwietz-Monien 91] If the first child fails to generate a cutoff, ***speculate*** that the remaining children can be searched in parallel without wasting any work: "***young brothers wait***."

Parallel Alpha-Beta (I)

```
cilk int search( position *prev, int move, int depth )
{
  position   cur;                /* current position        */
  int        bestscore = -INF;   /* best score so far       */
  int        num_moves;          /* number of children      */
  int        mv;                 /* index of child          */
  int        sc;                 /* child's score           */
  int        cutoff = FALSE;     /* have we seen a cutoff? */
```

- View from MAX's standpoint; MIN's viewpoint can be obtained by negating scores — ***negamax***.
- The current position is generated by the child, not by the parent.
- The **`alpha`** and **`beta`** limits and the move list are fields of the **`position`** structure.

Parallel Alpha-Beta (II)

```
inlet void get_score(int child_sc)
{
  child_sc = -child_sc;   /* negamax */

  if (child_sc > bestscore)
  {
    bestscore = child_sc;

    if (child_sc > cur.alpha)
    {
      cur.alpha = child_sc;

      if (child_sc >= cur.beta)
      {
        cutoff = TRUE;  /* no need to search further */
        abort;          /* terminate other children  */
      }
    }
  }
}
```

The **abort** keyword causes all spawned children of the frame to terminate abruptly.

Parallel Alpha-Beta (III)

```
/* create current position and set up for search */

make_move(prev, move, &cur);

sc = eval(&cur);   /* static evaluation */

if ( abs(sc)>=MATE || depth<=0 )   /* leaf node */
{
  return (sc);
}

cur.alpha = -prev->beta;   /* negamax */
cur.beta = -prev->alpha;

/* generate moves, hopefully in best-first order*/

num_moves = gen_moves(&cur);
```

Parallel Alpha-Beta (IV)

```
  /* search the moves */

  for (mv=0; !cutoff && mv<num_moves; mv++)
  {
    get_score( spawn search(&cur, mv, depth-1) );
    if ((mv==0) sync; /* young brothers wait */
  }

  sync;  /* this sync is outside the loop so that the
            searches after the first execute in parallel */

  return (bestscore);
}
```

"*Cilk*ifying" Alpha-Beta

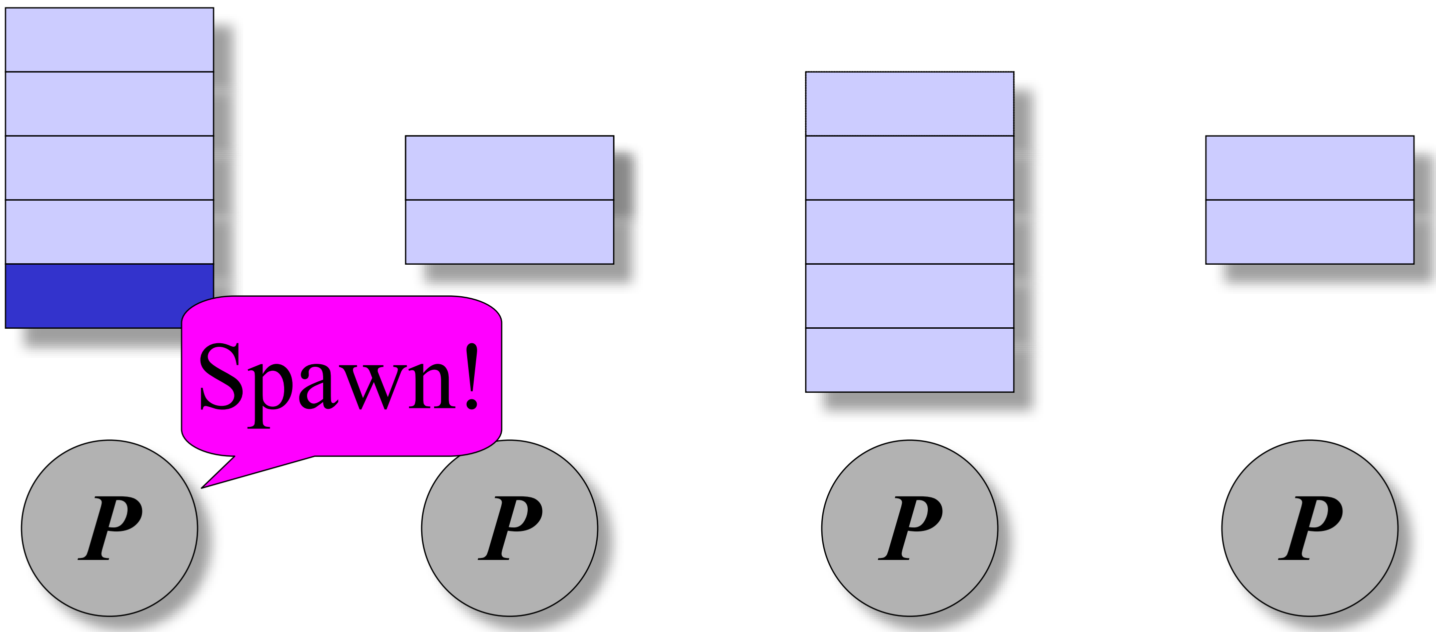
- Only 6 ***Cilk*** keywords need be embedded in the C program to parallelize it.
  - In fact, the program can be parallelized using only 5 keywords at the expense of minimal obfuscation.
- Typically, the major programming challenge for "*Cilk*ifying" an application is to express the program recursively, as opposed to actually parallelizing it.

OUTLINE

- Overview
- Cilk Performance
- A Chess Lesson
- Alpha-Beta Search
- *Cilk*'s Scheduler
- **Conclusion**

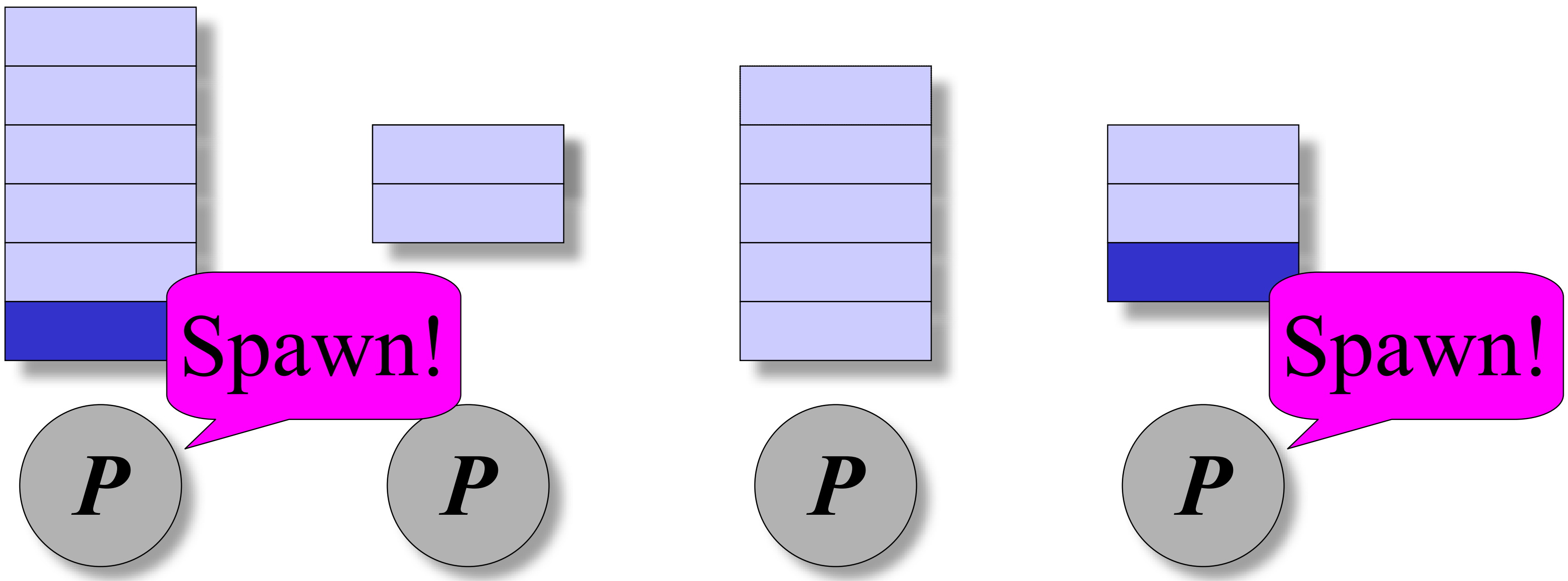
Cilk's Work-Stealing Scheduler

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



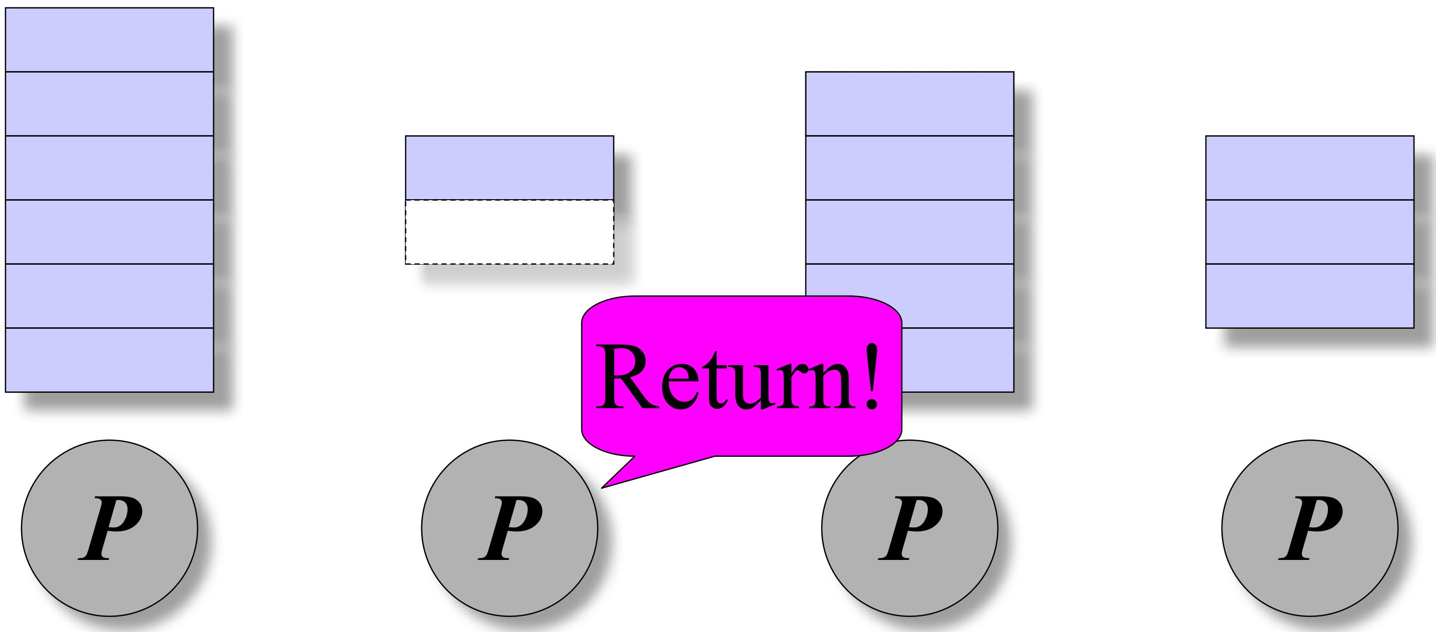
Cilk's Work-Stealing Scheduler

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



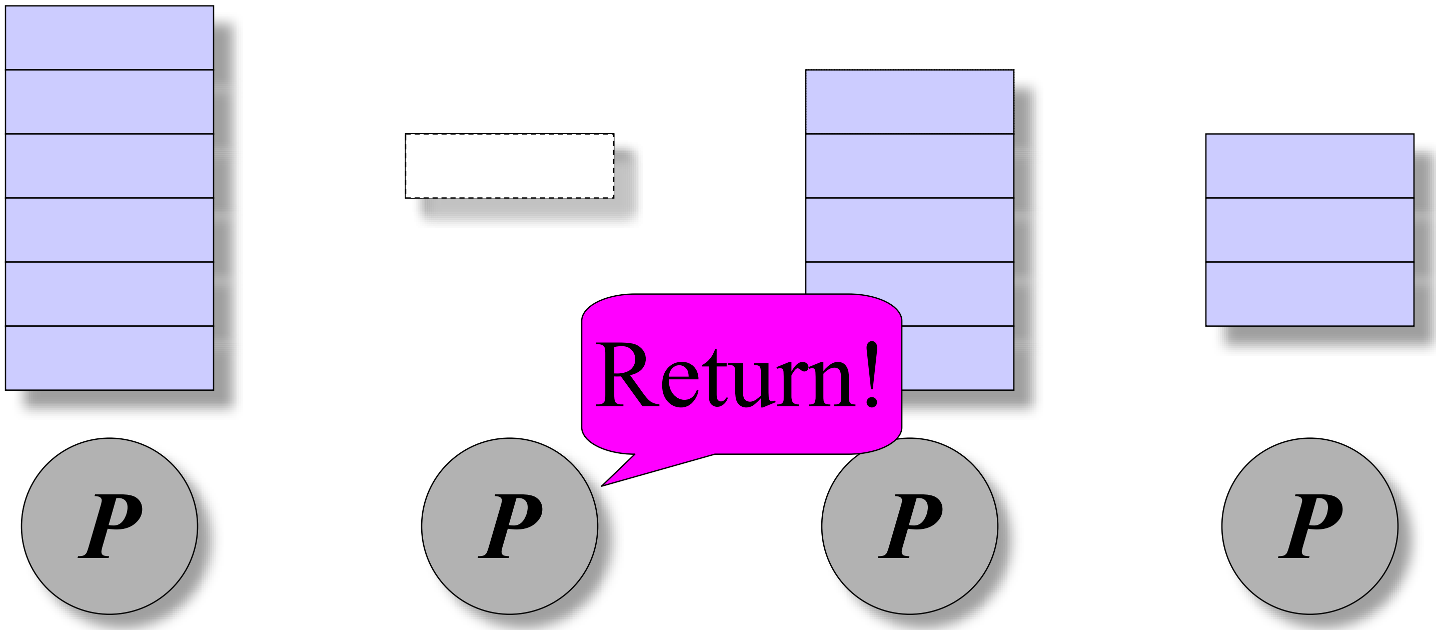
Cilk's Work-Stealing Scheduler

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



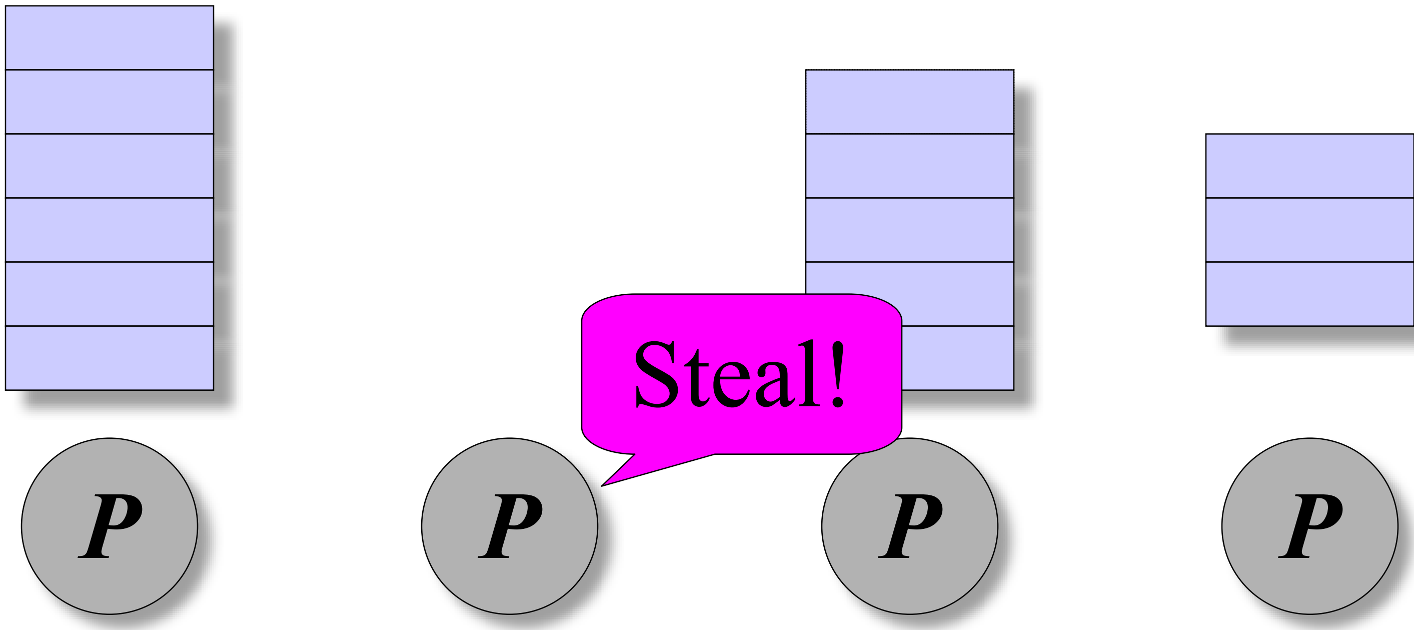
Cilk's Work-Stealing Scheduler

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

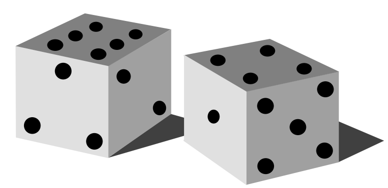


Cilk's Work-Stealing Scheduler

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

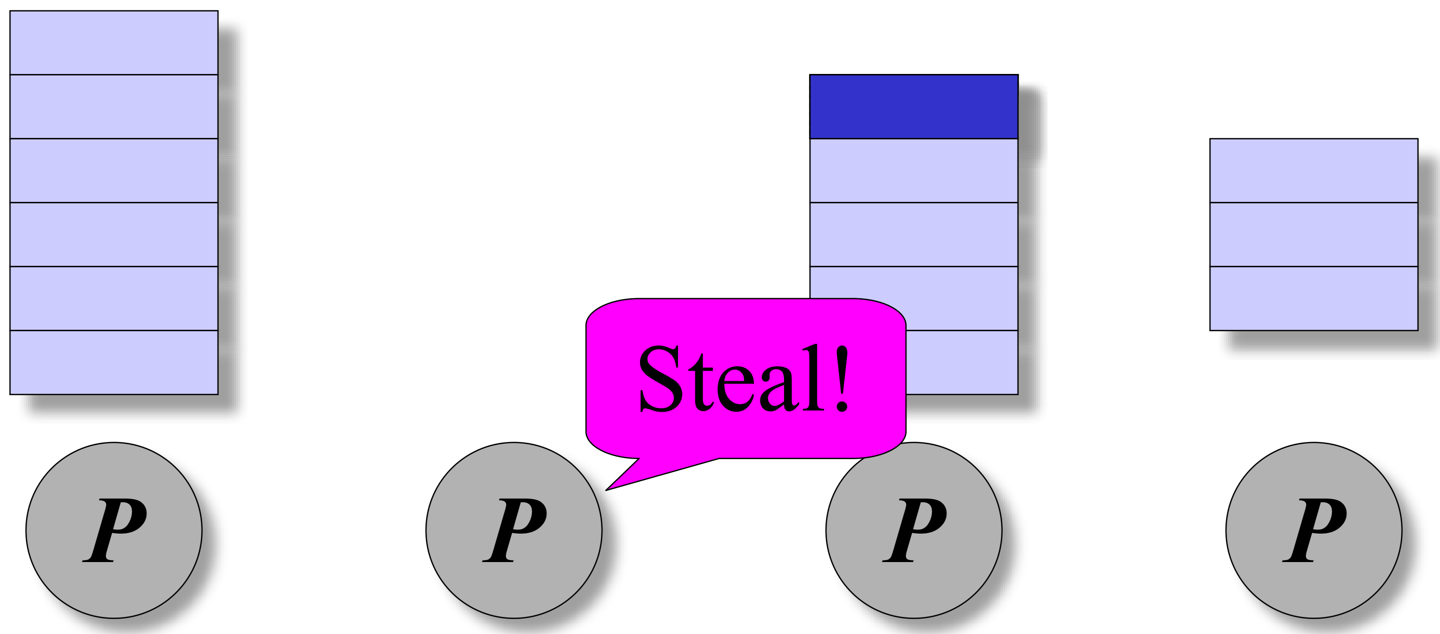


When a processor runs out of work, it ***steals*** a thread from the top of a ***random*** victim's deque.

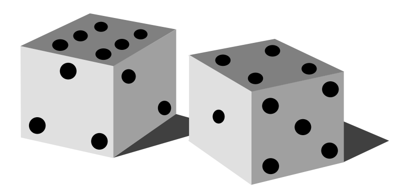


Cilk's Work-Stealing Scheduler

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

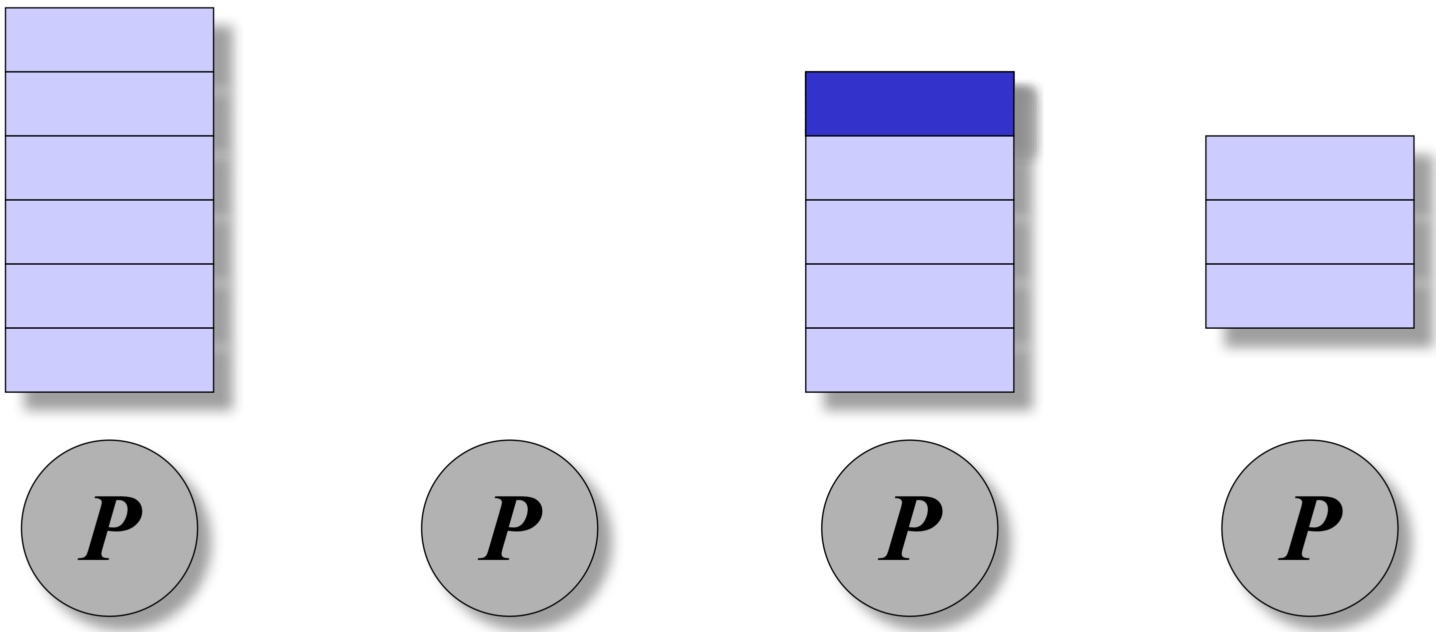


When a processor runs out of work, it ***steals*** a thread from the top of a ***random*** victim's deque.

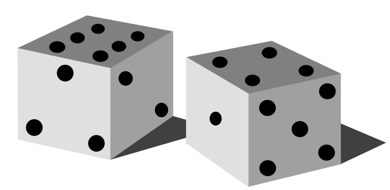


Cilk's Work-Stealing Scheduler

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

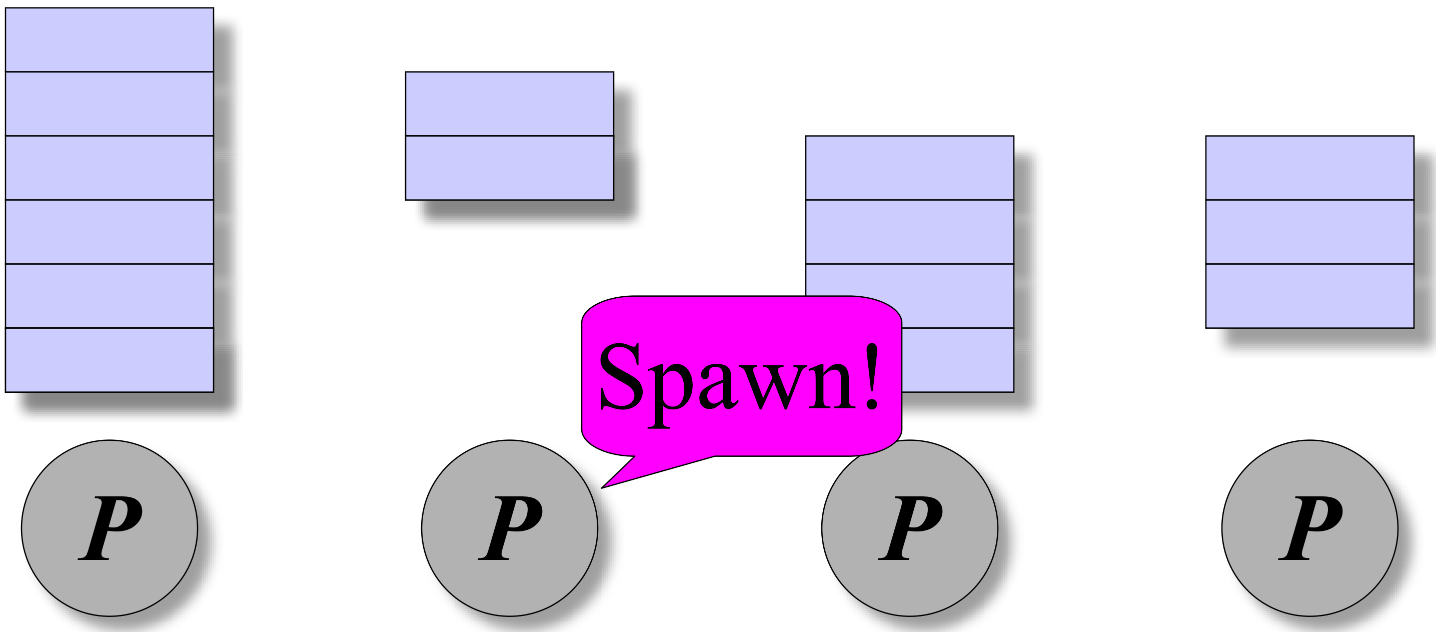


When a processor runs out of work, it ***steals*** a thread from the top of a ***random*** victim's deque.

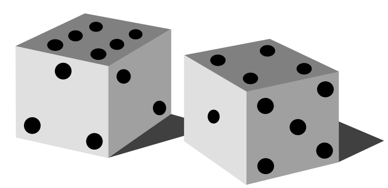


Cilk's Work-Stealing Scheduler

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

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

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 number of steals is $O(PT_1)$. Since there are P processors, the expected time is

$$(T_1 + O(PT_1))/P = T_1/P + O(T_1) \ . \quad \blacksquare$$

OUTLINE

- Overview
- *Cilk* Performance
- A Chess Lesson
- Alpha-Beta Search
- *Cilk*'s Scheduler
- **Conclusion**

Summary

- ***Cilk*** provides a simple parallel programming model that extends C in a faithful manner.
- ***Cilk*** scales down, as well as up.
- ***Cilk*** offers a release strategy for parallel software that supports traditional regression testing.
  - Debug and test the serial elision.
  - Certify absence of races using the Nondeterminator.
- ***Cilk*** encourages divide-and-conquer recursion, which is provably cache-friendly.
- ***Cilk*** is a processor-oblivious language that allows adaptive scheduling in multiprogramming environments.

Cilk Arts, Inc.

- Headquartered in Lexington, Massachusetts.
- Focused on the multicore opportunity.
- Currently prefunded.
- Launch product: ***Cilk++*** (~12 months).
- Business model: license the runtime platform.
- URL: `www.cilk.com`.

Cilk++ Features

- C++.
- Windows and Linux (and other Unixes).
- Remove the limitation that only parallel functions can spawn.
- Allow spawning of arbitrary statement blocks.
- Loop syntax to avoid recoding loops as divide-and-conquer.
- Support for exceptions (à la ***JCilk***).

Cilk Contributors

Kunal Agarwal
Eitan Ben Amos
Bobby Blumofe
Angelina Lee
Ien Cheng
Mingdong Feng
Jeremy Fineman
Matteo Frigo
Michael Halbherr
Chris Joerg
Bradley Kuszmaul
Phil Lisiecki
Rob Miller
Harald Prokop
Keith Randall
Bin Song
Andy Stark
Volker Strumpen
Yuli Zhou

...plus many MIT students and SourceForgers.

World Wide Web

***Cilk** source code, programming examples, documentation, technical papers, tutorials, and up-to-date information can be found at:*

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

Download CILK Today!