Multithreaded Programming in *Cilk*

LECTURE 3

Charles E. Leiserson

Supercomputing Technologies Research Group
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology

Minicourse Outline

- LECTURE 1
  Basic Cilk programming: Cilk keywords, performance measures, scheduling.
- LECTURE 2
  Analysis of Cilk algorithms: matrix multiplication, sorting, tableau construction.
- LABORATORY
  Programming matrix multiplication in Cilk — Dr. Bradley C. Kuszmaul
- LECTURE 3
  Advanced Cilk programming: inlets, abort, speculation, data synchronization, & more.

LECTURE 3

- **Inlets**
- **Abort**
- **Speculative Computing**
- **Data Synchronization**
- **Under the Covers**
- **JCilk**
- **Conclusion**

Operating on Returned Values

Programmers may sometimes wish to incorporate a value returned from a spawned child into the parent frame by means other than a simple variable assignment.

Example:

```
x += spawn foo(a,b,c);
```

Cilk achieves this functionality using an internal function, called an ***inlet***, which is executed as a secondary thread on the parent frame when the child returns.

Semantics of Inlets

```
int max, ix = -1;
inlet void update ( int val, int index ) {
  if (idx == -1 || val > max) {
    ix = index; max = val;
  }
}
⋮
for (i=0; i<1000000; i++) {
  update ( spawn foo(i), i );
}
sync; /* ix now indexes the largest foo(i) */
```

- The **inlet** keyword defines a **void** internal function to be an inlet.
- In the current implementation of Cilk, the inlet definition may not contain a **spawn**, and only the first argument of the inlet may be spawned at the call site.

Semantics of Inlets

```
int max, ix = -1;
inlet void update ( int val, int index ) {
  if (idx == -1 || val > max) {
    ix = index; max = val;
  }
}
⋮
for (i=0; i<1000000; i++) {
  update ( spawn foo(i), i );
}
sync; /* ix now indexes the largest foo(i) */
```

1. The non-**spawn** args to **update()** are evaluated.
2. The Cilk procedure **foo(i)** is spawned.
3. Control passes to the next statement.
4. When **foo(i)** returns, **update()** is invoked.

Semantics of Inlets

```
int max, ix = -1;
inlet void update ( int val, int index ) {
  if (idx == -1 || val > max) {
    ix = index; max = val;
  }
}
:
for (i=0; i<1000000; i++) {
  update ( spawn foo(i), i );
}
sync; /* ix now indexes the largest foo(i) */
```

Cilk provides ***implicit atomicity*** among the threads belonging to the same frame, and thus no locking is necessary to avoid data races.

Implicit Inlets

```
cilk int wfib(int n) {
  if (n == 0) {
    return 0;
  } else {
    int i, x = 1;
    for (i=0; i<=n-2; i++) {
      x += spawn wfib(i);
    }
    sync;
    return x;
  }
}
```

For assignment operators, the Cilk compiler automatically generates an ***implicit inlet*** to perform the update.

LECTURE 3

- **Inlets**
- **Abort**
- **Speculative Computing**
- **Data Synchronization**
- **Under the Covers**
- **JCilk**
- **Conclusion**

Computing a Product

$$p = \prod_{i=0}^{n} A_i$$

```
int product(int *A, int n) {
  int i, p=1;
  for (i=0; i<n; i++) {
    p *= A[i];
  }
  return p;
}
```

Optimization: Quit early if the partial product ever becomes 0.

Computing a Product

$$p = \prod_{i=0}^{n} A_i$$

```
int product(int *A, int n) {
  int i, p=1;
  for (i=0; i<n; i++) {
    p *= A[i];
    if (p == 0) break;
  }
  return p;
}
```

Optimization: Quit early if the partial product ever becomes 0.

Computing a Product in Parallel

$$p = \prod_{i=0}^{n} A_i$$

```
cilk int prod(int *A, int n) {
  int p = 1;
  if (n == 1) {
    return A[0];
  } else {
    p *= spawn product(A, n/2);
    p *= spawn product(A+n/2, n-n/2);
    sync;
    return p;
  }
}
```

How do we quit early if we discover a zero?

Cilk's Abort Feature

```
cilk int product(int *A, int n) {
  int p = 1;
  inlet void mult(int x) {
    p *= x;
    return;
  }

  if (n == 1) {
    return A[0];
  } else {
    mult( spawn product(A, n/2) );
    mult( spawn product(A+n/2, n-n/2) );
    sync;
    return p;
  }
}
```

1. Recode the implicit inlet to make it explicit.

Cilk's Abort Feature

```
cilk int product(int *A, int n) {
  int p = 1;
  inlet void mult(int x) {
    p *= x;


    return;
  }

  if (n == 1) {
    return A[0];
  } else {
    mult( spawn product(A, n/2) );
    mult( spawn product(A+n/2, n-n/2) );
    sync;
    return p;
  }
}
```

2. Check for 0 within the inlet.

Cilk's Abort Feature

```
cilk int product(int *A, int n) {
  int p = 1;
  inlet void mult(int x) {
    p *= x;
    if (p == 0) {
      abort; /* Aborts existing children, */
    }        /* but not future ones.      */
    return;
  }

  if (n == 1) {
    return A[0];
  } else {
    mult( spawn product(A, n/2) );
    mult( spawn product(A+n/2, n-n/2) );
    sync;
    return p;
  }
}
```

2. Check for 0 within the inlet.

Cilk's Abort Feature

```
cilk int product(int *A, int n) {
  int p = 1;
  inlet void mult(int x) {
    p *= x;
    if (p == 0) {
      abort; /* Aborts existing children, */
    }        /* but not future ones.      */
    return;
  }

  if (n == 1) {
    return A[0];
  } else {
    mult( spawn product(A, n/2) );



    mult( spawn product(A+n/2, n-n/2) );
    sync;
    return p;
  }
}
```


Cilk's Abort Feature

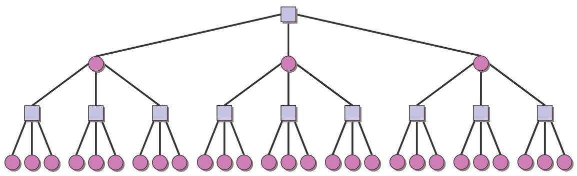
```
cilk int product(int *A, int n) {
  int p = 1;
  inlet void mult(int x) {
    p *= x;
    if (p == 0) {
      abort; /* Aborts existing children, */
    }        /* but not future ones.      */
    return;
  }

  if (n == 1) {
    return A[0];
  } else {
    mult( spawn product(A, n/2) );
    if (p == 0) { /* Don't spawn if we've */
      return 0;   /* already aborted!     */
    }
    mult( spawn product(A+n/2, n-n/2) );
    sync;
    return p;
  }
}
```

Implicit atomicity eases reasoning about races.

LECTURE 3

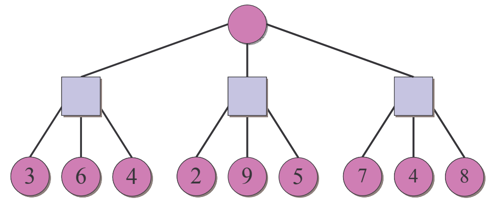
- Inlets
- Abort
- **Speculative Computing**
- **Data Synchronization**
- **Under the Covers**
- **JCilk**
- **Conclusion**

Min-Max Search



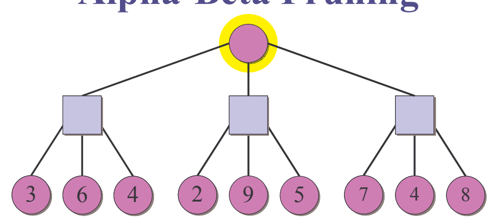
- Two players: MAX ■ and MIN ●.
- The game tree represents all moves from the current position within a given search depth.
- At leaves, apply a static evaluation function.
- MAX chooses the maximum score among its children.
- MIN chooses the minimum score among its children.

Alpha-Beta Pruning



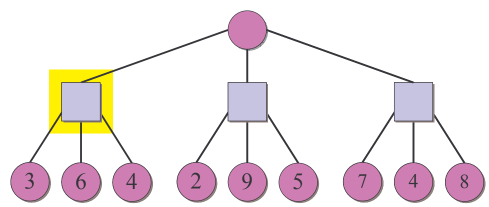
IDEA: If MAX ■ discovers a move so good that MIN ● would never allow that position, MAX's other children need not be searched — ***beta cutoff***.

Alpha-Beta Pruning



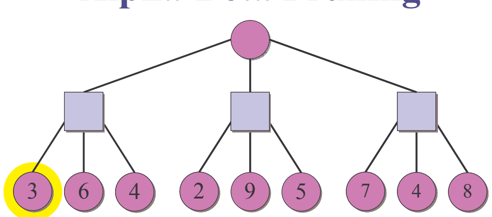
IDEA: If MAX ■ discovers a move so good that MIN ● would never allow that position, MAX's other children need not be searched — ***beta cutoff***.

Alpha-Beta Pruning



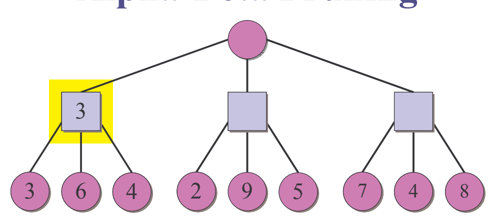
IDEA: If MAX ■ discovers a move so good that MIN ● would never allow that position, MAX's other children need not be searched — ***beta cutoff***.

Alpha-Beta Pruning



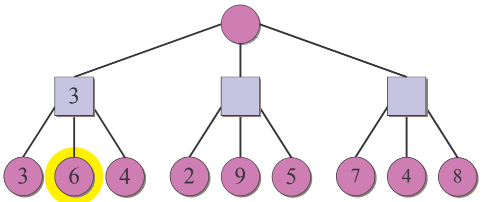
IDEA: If MAX ■ discovers a move so good that MIN ● would never allow that position, MAX's other children need not be searched — ***beta cutoff***.

Alpha-Beta Pruning



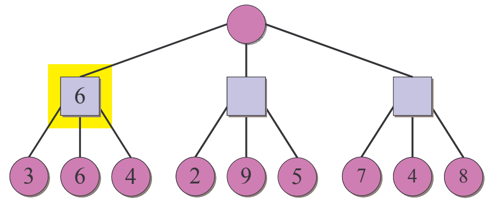
IDEA: If MAX ■ discovers a move so good that MIN ● would never allow that position, MAX's other children need not be searched — ***beta cutoff***.

Alpha-Beta Pruning



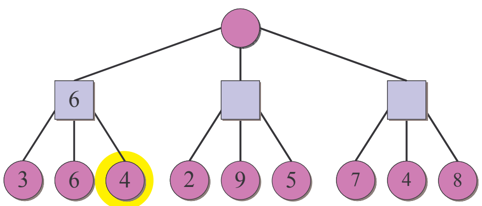
IDEA: If MAX discovers a move so good that MIN would never allow that position, MAX's other children need not be searched — ***beta cutoff***.

Alpha-Beta Pruning



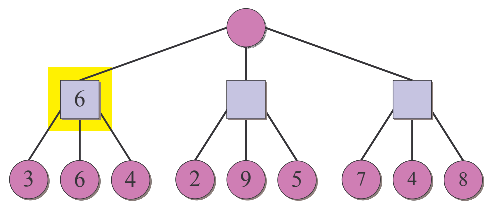
IDEA: If MAX discovers a move so good that MIN would never allow that position, MAX's other children need not be searched — ***beta cutoff***.

Alpha-Beta Pruning



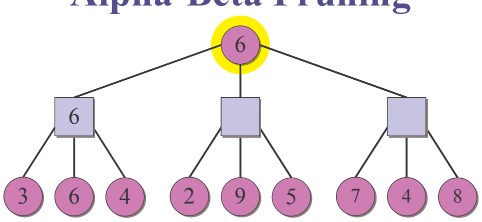
IDEA: If MAX discovers a move so good that MIN would never allow that position, MAX's other children need not be searched — ***beta cutoff***.

Alpha-Beta Pruning



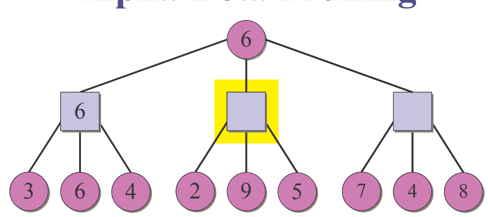
IDEA: If MAX discovers a move so good that MIN would never allow that position, MAX's other children need not be searched — ***beta cutoff***.

Alpha-Beta Pruning



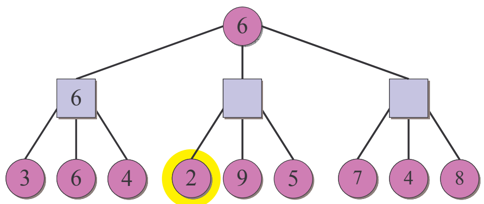
IDEA: If MAX discovers a move so good that MIN would never allow that position, MAX's other children need not be searched — ***beta cutoff***.

Alpha-Beta Pruning



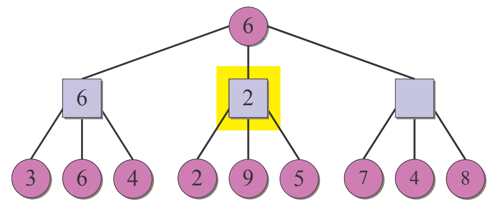
IDEA: If MAX discovers a move so good that MIN would never allow that position, MAX's other children need not be searched — ***beta cutoff***.

Alpha-Beta Pruning



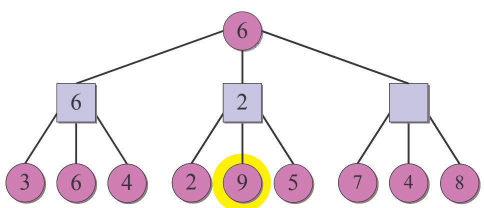
IDEA: If MAX discovers a move so good that MIN would never allow that position, MAX's other children need not be searched — ***beta cutoff***.

Alpha-Beta Pruning



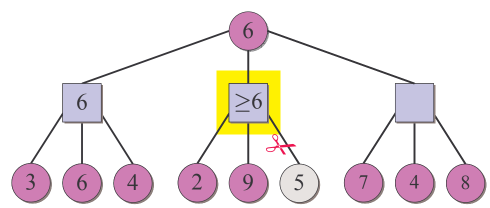
IDEA: If MAX discovers a move so good that MIN would never allow that position, MAX's other children need not be searched — ***beta cutoff***.

Alpha-Beta Pruning



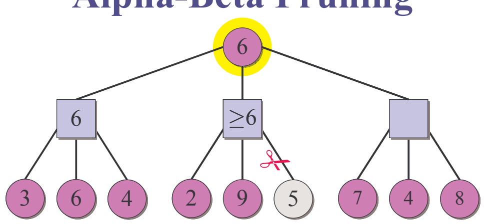
IDEA: If MAX discovers a move so good that MIN would never allow that position, MAX's other children need not be searched — ***beta cutoff***.

Alpha-Beta Pruning



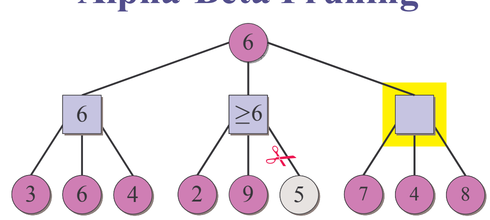
IDEA: If MAX discovers a move so good that MIN would never allow that position, MAX's other children need not be searched — ***beta cutoff***.

Alpha-Beta Pruning



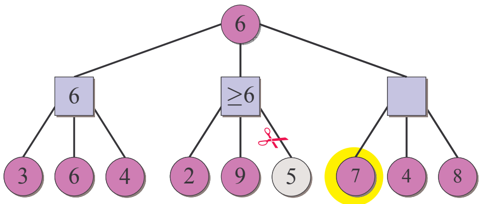
IDEA: If MAX discovers a move so good that MIN would never allow that position, MAX's other children need not be searched — ***beta cutoff***.

Alpha-Beta Pruning



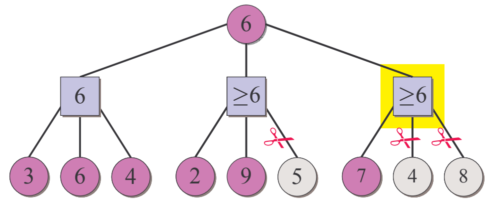
IDEA: If MAX discovers a move so good that MIN would never allow that position, MAX's other children need not be searched — ***beta cutoff***.

Alpha-Beta Pruning



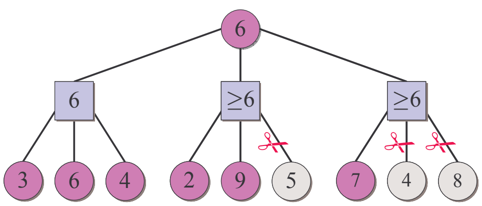
IDEA: If MAX ■ discovers a move so good that MIN ● would never allow that position, MAX's other children need not be searched — ***beta cutoff***.

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

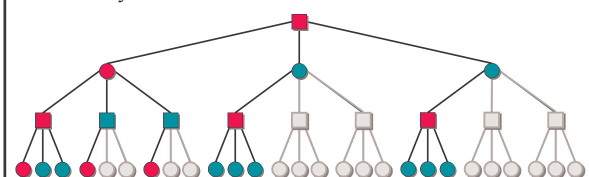


IDEA: If MAX ■ discovers a move so good that MIN ● would never allow that position, MAX's other children need not be searched — ***beta cutoff***.

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 perspective; MIN's viewpoint can be obtained by negating scores — ***negamax***.
- The node generates it[...] m its parent's position **pre**[...]
- The **alpha** and **beta** [...] are fields of the **position** data structure.

#Cilk keywords used so far

1

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) { /* Beta cutoff */
        cutoff = TRUE; /* No need to search more    */
        abort;         /* Terminate other children */
      }
    }
  }
}
```

3

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);
```

3

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;
  return (bestscore);
}
```

- 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.

6

LECTURE 3

- Inlets
- Abort
- Speculative Computing
- **Data Synchronization**
- **Under the Covers**
- **JCilk**
- **Conclusion**

Mutual Exclusion

Cilk's solution to mutual exclusion is no better than anybody else's.

Cilk provides a library of spin locks declared with `Cilk_lockvar`.

- To avoid deadlock with the Cilk scheduler, a lock should only be held within a Cilk thread.
- *I.e.*, `spawn` and `sync` should not be executed while a lock is held.

Fortunately, Cilk's control parallelism often mitigates the need for extensive locking.

Cilk's Memory Model

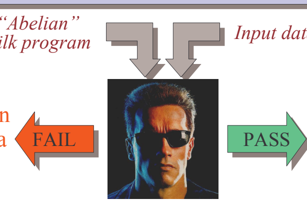
Programmers may also synchronize through memory using lock-free protocols, although Cilk is agnostic on consistency model.

- If a program contains no data races, Cilk effectively supports sequential consistency.
- If a program contains data races, Cilk's behavior depends on the consistency model of the underlying hardware.

To aid portability, the `Cilk_fence()` function implements a memory barrier on machines with weak memory models.

Debugging Data Races

Cilk's ***Nondeterminator*** debugging tool provably guarantees to detect and localize data-race bugs.

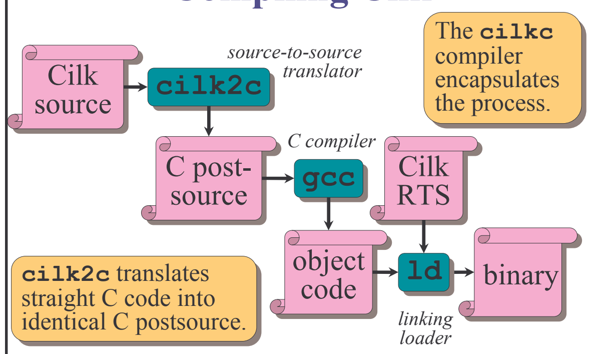


A ***data race*** occurs whenever two logically parallel threads, holding no locks in common, access the same location and one of the threads modifies the location.

LECTURE 3

- Inlets
- Abort
- Speculative Computing
- Data Synchronization
- **Under the Covers**
- **JCilk**
- **Conclusion**

Compiling Cilk

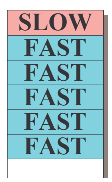


Cilk's Compiler Strategy

The **cilk2c** translator generates two "clones" of each Cilk procedure:

- *fast clone*—serial, common-case code.
- *slow clone*—code with parallel bookkeeping.

- The *fast clone* is always spawned, saving live variables on Cilk's work deque (shadow stack).
- The *slow clone* is resumed if a thread is stolen, restoring variables from the shadow stack.
- A check is made whenever a procedure returns to see if the resuming parent has been stolen.



Compiling spawn — Fast Clone

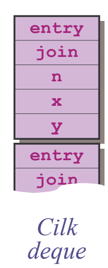
Cilk source

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

cilk2c

C post-source

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

x = fib(n-1);         } run child

if (pop()==FAILURE) {
  frame->x = x;
  frame->join--;
  ⟨clean up &
   return to scheduler⟩
}                     } resume parent remotely
```



Compiling sync — Fast Clone

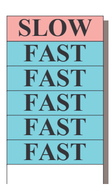
Cilk source

```
sync;
```

cilk2c

C post-source

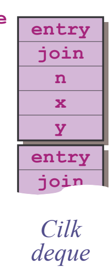
```
;
```



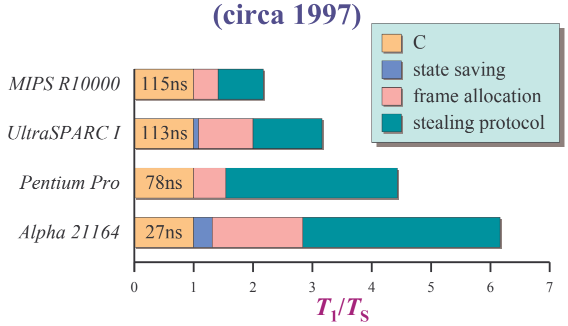
No synchronization overhead in the fast clone!

Compiling the Slow Clone

```
void fib_slow(fib_frame *frame) {
  int n,x,y;
  switch (frame->entry) {      } restore program counter
    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--;
    ⟨clean up &
     return to scheduler⟩
  }                            } same as fast clone

  if (0) {
    L1:;
    n = frame->n;
  }                            } restore local variables if resuming
       ⋮                       } continue
}
```



Breakdown of Work Overhead
(circa 1997)

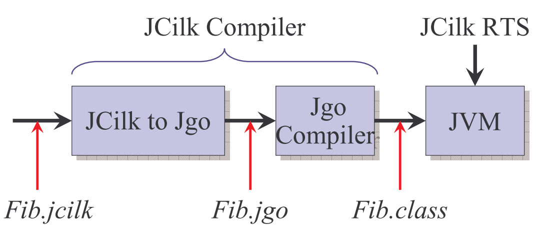


***Benchmark*: `fib` on one processor.**

LECTURE 3

- Inlets
- Abort
- Speculative Computing
- Data Synchronization
- Under the Covers
- **JCilk**
- **Conclusion**

The JCilk System



- Jgo = Java + `goto`.
- The Jgo compiler was built by modifying `gcj` to accept `goto` statements so that a continuation mechanism for JCilk could be implemented.

JCilk Keywords

`cilk`
`spawn`
`sync`
`SYNCHED`
} Same as Cilk, except that `cilk` can also modify `try`.

~~`inlet`~~
~~`abort`~~
} Eliminated!

JCilk leverages Java's exception mechanism to render two Cilk keywords unnecessary.

Exception Handling in Java

"During the process of throwing an exception, the Java virtual machine ***abruptly completes***, one by one, any expressions, statements, method and constructor invocations, initializers, and field initialization expressions that have begun but not completed execution in the current thread. This process continues until a handler is found that indicates that it handles that particular exception by naming the class of the exception or a superclass of the class of the exception."

— J. Gosling, B Joy, G. Steele, and G. Bracha, *Java Language Specification*, 2000, pp. 219–220.

Exception Handling in JCilk

```
private cilk void foo() throws IOException {
   spawn A();
   cilk try {
      spawn B();
      cilk try {
         spawn C();
      } catch(ArithmeticEx'n e) {
         doSomething();
      }
   } catch(RuntimeException e) {
      doSomethingElse();
   }
   spawn D();
   doYetSomethingElse();
   sync;
}
```

Exception Handling in JCilk

```
private cilk void foo() throws IOException {
    spawn A();
    cilk try {
        spawn B();
        cilk try {
            spawn C();
        } catch(ArithmeticEx’n e) {
            doSomething();
        }
    } catch(RuntimeException e) {
        doSomethingElse();
    }
    spawn D();
    doYetSomethingElse();
    sync;
}
```

Exception!

An exception causes all subcomputations dynamically enclosed by the catching clause to abort!

Exception Handling in JCilk

```
private cilk void foo() throws IOException {
    spawn A();
    cilk try {
        spawn B();
        cilk try {
            spawn C();
        } catch(ArithmeticEx’n e) {
            doSomething();
        }
    } catch(RuntimeException e) {
        doSomethingElse();
    }
    spawn D();
    doYetSomethingElse();
    sync;
}
```

ArithmeticEx’n

Nothing aborts.

An exception causes all subcomputations dynamically enclosed by the catching clause to abort!

Exception Handling in JCilk

```
private cilk void foo() throws IOException {
    spawn A();
    cilk try {
        spawn B();
        cilk try {
            spawn C();
        } catch(ArithmeticEx’n e) {
            doSomething();
        }
    } catch(RuntimeException e) {
        doSomethingElse();
    }
    spawn D();
    doYetSomethingElse();
    sync;
}
```

RuntimeEx’n

An exception causes all subcomputations dynamically enclosed by the catching clause to abort!

Exception Handling in JCilk

```
private cilk void foo() throws IOException {
    spawn A();
    cilk try {
        spawn B();
        cilk try {
            spawn C();
        } catch(ArithmeticEx’n e) {
            doSomething();
        }
    } catch(RuntimeException e) {
        doSomethingElse();
    }
    spawn D();
    doYetSomethingElse();
    sync;
}
```

IOException

An exception causes all subcomputations dynamically enclosed by the catching clause to abort!

Exception Handling in JCilk

```
private cilk void foo() throws IOException {
    spawn A();
    cilk try {
        spawn B();
        cilk try {
            spawn C();
        } catch(ArithmeticEx’n e) {
            doSomething();
        }
    } catch(RuntimeException e) {
        doSomethingElse();
    }
    spawn D();
    doYetSomethingElse();
    sync;
}
```

RuntimeEx’n

The appropriate **`catch`** *clause is executed only after all spawned methods within the corresponding* **`try`** *block terminate.*

JCilk’s Exception Mechanism

- JCilk’s exception semantics allow programs such as alpha-beta to be coded without Cilk’s **`inlet`** and **`abort`** keywords.
- Unfortunately, Java exceptions are slow, reducing the utility of JCilk’s faithful extension.

LECTURE 3

- Inlets
- Abort
- Speculative Computing
- Data Synchronization
- Under the Covers
- JCilk
- **Conclusion**

Future Work

Adaptive computing

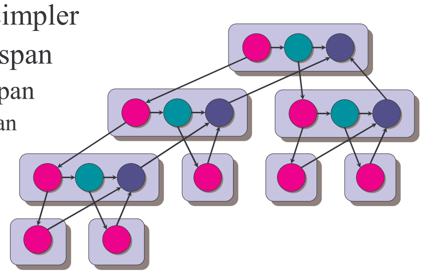
- Get rid of `--nproc` .
- Build a job scheduler that uses ***parallelism feedback*** to balance processor resources among Cilk jobs.

Integrating Cilk with static threads

- Currently, interfacing a Cilk program to other system processes requires arcane knowledge.
- Build linguistic support into Cilk for Cilk processes that communicate.
- Develop a job scheduler that uses ***pipeload*** to allocate resources among Cilk processes.

Key Ideas

- Cilk is simple: `cilk`, `spawn`, `sync`, `SYNCHED`, `inlet`, `abort`
- JCilk is simpler
- Work & span
- Work & span
- Work & span
- Work & span
- Work & span
- Work & span
- Work & span
- Work & span
- Work & span
- Work & span
- Work & span



Open-Cilk Consortium

- We are in the process of forming a consortium to manage, organize, and promote Cilk open-source technology.
- If you are interested in participating, please let us know.

ACM Symposium on Parallelism in Algorithms and Architectures

SPAA 2006

Cambridge, MA, USA
July 30 – August 2, 2006