COMP 422, Lecture 6: Advanced Cilk Features

(Sections 2.6 - 2.9 & 5.1 - 5.4 of Cilk Reference Manual)

Vivek Sarkar

Department of Computer Science
Rice University

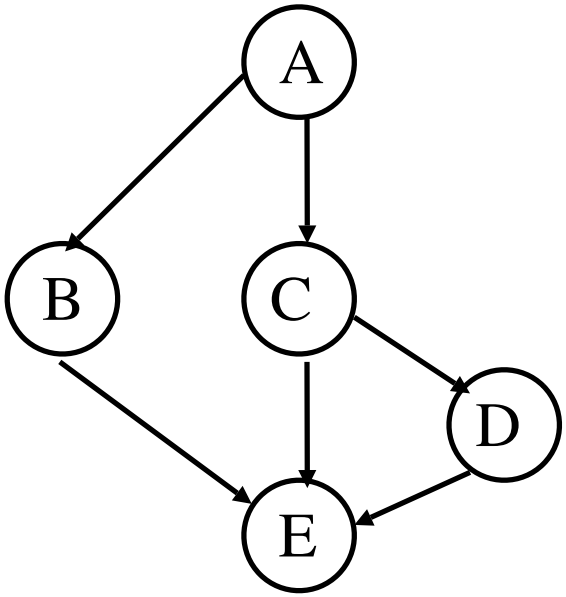
vsarkar@rice.edu

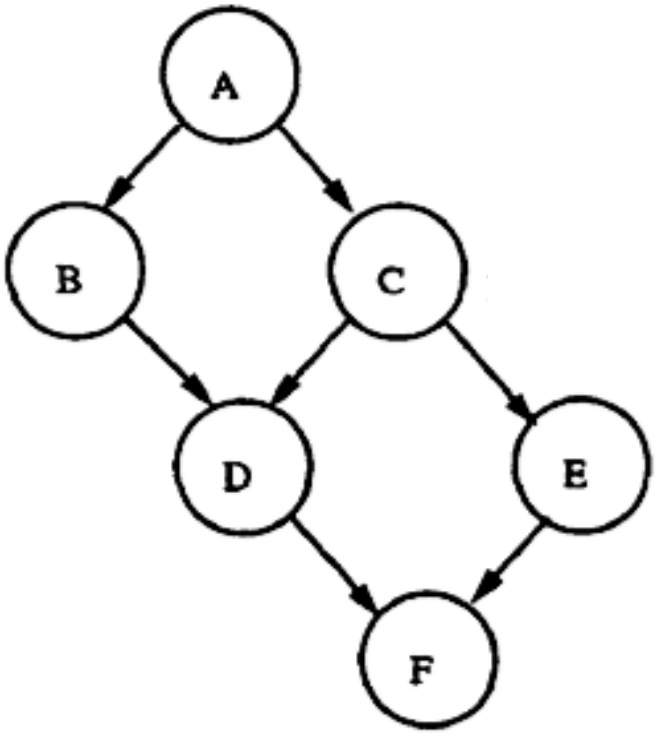
Recap of Lecture 5

- **Thread Basics**
  - —pthread_create
    - Each thread has a private stack
  - —pthread_join
- **Introduction to Cilk**
  - —cilk --- keyword to distinguish Cilk procedures from C procedures
  - —spawn --- execute child Cilk procedure call in parallel with parent procedure
    - Cactus stack enables child procedure to access parent's stack variables
  - —sync --- suspend till all child procedures have completed
    - Applies to all child procedures (no lexical scoping)
    - "As an aid to programmers, Cilk inserts an implicit sync before every return, if it is not present already. As a consequence, a procedure never terminates while it has outstanding children."

Lecture 5 Review Question

- **Which of the following graphs can be realized as a computation dag using Cilk's async and spawn constructs only?**





Acknowledgments for today's lecture

- **Cilk lecture by Charles Leiserson and Bradley Kuszmaul (Lecture 3, Advanced Cilk programming)**
  - **http://supertech.csail.mit.edu/cilk/lecture-3.pdf**
  - **Slides modified with permission from authors**

Outline

- **inlet**
- **abort**
- **Cilk_alloca**
- **SYNCHED**
  - **Matrix Multiply example**
- **Cilk_lockvar**

Operating on Returned Values

Programmers may sometimes wish to operate on a return value without waiting on a sync

Example:

```
for (i=0; i<1000000; i++) {
  update(spawn foo(i), i );
}
sync;
/* All spawns and updates are now
completed */
```

Cilk achieves this functionality using an internal function, called an ***inlet***, which can be viewed as an "event handler" thread (task) executed by the parent 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 implicitly guarantees ***atomicity*** among the threads/tasks (including inlets) belonging to the same procedure instance, 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.

Outline

- **inlet**
- **abort**
- **Cilk_alloca**
- **SYNCHED**
  - **Matrix Multiply example**
- **Cilk_lockvar**

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 {
    /* Note use of implicit inlets */
    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) );
    if (p == 0) { /* Add check for future */
      return 0;   /* children            */
    }
    mult( spawn product(A+n/2, n-n/2) );
    sync;
    return p;
  }
}
```

Potential anomalies with abort

- Abort can be useful for speculative parallelism such as Parallel Min-Max Search but …
  - —No guarantee on when child is terminated
    - It may not be instantly
    - It's possible that child completes normally anyway
    - Think of abort as a "best effort" to terminate child procedures, with no guarantees
  - —When control resumes at sync, return values of aborted children will not be set
    - Programmer needs to handle this case
  - —Abort may or may not terminate future children depending on whether or not they've been spawned
    - Only the children after a sync are guaranteed to not be terminated by an abort before the sync
- See Section 2.7.2 for details

Outline

- **inlet**
- **abort**
- **Cilk_alloca**
- **SYNCHED**
  - **Matrix Multiply example**
- **Cilk_lockvar**

Cilk_alloca

- Extension of alloca from sequential C
- ptr = Cilk_alloca(size);
  - Allocated in stack frame of procedure calling Cilk_alloca
  - Freed when procedure returns
  - Well suited for data sharing in divide-and-conquer parallelism due to Cilk's cactus stack
- Should only be called from cilk procedures
  - "In the current release, Cilk's version of Cilk alloca() does not work properly when it is called from within a C function. Similarly, the C function alloca() does not work properly when called within a Cilk procedure."

Outline

- **inlet**
- **abort**
- **Cilk_alloca**
- **SYNCHED**
  - **Matrix Multiply example**
- **Cilk_lockvar**

SYNCHED variable

Cilk language feature: A programmer can check whether a Cilk procedure is "synched" (without actually performing a `sync`) by testing the pseudovariable `SYNCHED`:

- `SYNCHED` = 0, some spawned children might not have returned.
- `SYNCHED` = 1, all spawned children have definitely returned.

Square-Matrix Multiplication

$$\begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} * \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{pmatrix}$$

C A B

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

Assume for simplicity that $n = 2^k$.

Recursive Matrix Multiplication

Divide and conquer —

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} * \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$= \begin{bmatrix} A_{11}B_{11} & A_{11}B_{12} \\ A_{21}B_{11} & A_{21}B_{12} \end{bmatrix} + \begin{bmatrix} A_{12}B_{21} & A_{12}B_{22} \\ A_{22}B_{21} & A_{22}B_{22} \end{bmatrix}$$

8 multiplications of $(n/2) * (n/2)$ matrices.
1 addition of $n * n$ matrices.

Matrix Multiply in Pseudo-Cilk

```
cilk void Mult(*C, *A, *B, n) {
  float *T = Cilk_alloca(n*n*sizeof(float));
  <base case & partition matrices>
  spawn Mult(C11,A11,B11,n/2);
  spawn Mult(C12,A11,B12,n/2);
  spawn Mult(C22,A21,B12,n/2);
  spawn Mult(C21,A21,B11,n/2);
  spawn Mult(T11,A12,B21,n/2);
  spawn Mult(T12,A12,B22,n/2);
  spawn Mult(T22,A22,B22,n/2);
  spawn Mult(T21,A22,B21,n/2);
  sync;
  spawn Add(C,T,n);
  sync;
  return;
}
```

$C = A^* B$

Matrix Multiply in Pseudo-Cilk

```
cilk void Mult(*C, *A, *B, n) {
  float *T = Cilk_alloca(n*n*sizeof(float));
  < base case & partition matrices >
  spawn Mult(C11,A11,B11,n/2);
  spawn Mult(C12,A11,B12,n/2);
  spawn Mult(C22,A21,B12,n/2);
  spawn Mult(C21,A21,B11,n/2);
  spawn Mult(T11,A12,B21,n/2);
  spawn Mult(T12,A12,B22,n/2);
  spawn Mult(T22,A22,B22,n/2);
  spawn Mult(T21,A22,B21,n/2);
  sync;
  spawn Add(C,T,n);
  sync;
  return;
}
```

```
cilk void Add(*C, *T, n) {
  <base case & partition matrices>
  spawn Add(C11,T11,n/2);
  spawn Add(C12,T12,n/2);
  spawn Add(C21,T21,n/2);
  spawn Add(C22,T22,n/2);
  sync;
  return;
}
```

$C = A * B$

$C = C + T$

Work of Matrix Addition

```
cilk void Add(*C, *T, n) {
   <base case & partition matrices>
   spawn Add(C11,T11,n/2);
   spawn Add(C12,T12,n/2);
   spawn Add(C21,T21,n/2);
   spawn Add(C22,T22,n/2);
   sync;
   return;
}
```

Work: $A_1(n) = 4A_1(n/2) + \Theta(1)$

$= \Theta(n^2)$ — **CASE 1**

Span of Matrix Addition

```
cilk void Add(*C, *T, n) {
   <base case & partition matrices>
   spawn Add(C11,T11,n/2);
   spawn Add(C12,T12,n/2);
   spawn Add(C21,T21,n/2);
   spawn Add(C22,T22,n/2);
   sync;
   return;
}
```

maximum

Span: $A_\infty(n) = A_\infty(n/2) + \Theta(1)$

$= \Theta(\lg n)$ — **CASE 2**

Work of Matrix Multiplication

```
cilk void Mult(*C, *A, *B, n) {
  float *T = Cilk_alloca(n*n*sizeof(float));
  <base case & partition matrices>
  spawn Mult(C11,A11,B11,n/2);
  spawn Mult(C12,A11,B12,n/2);
  :
  spawn Mult(T21,A22,B21,n/2);
  sync;
  spawn Add(C,T,n);
  sync;
  return;
}
```

8

Work: $M_1(n) = 8\,M_1(n/2) + A_1(n) + \Theta(1)$

$= 8\,M_1(n/2) + \Theta(n^2)$

$= \Theta(n^3)$ — **CASE 1**

Span of Matrix Multiplication

```
cilk void Mult(*C, *A, *B, n) {
  float *T = Cilk_alloca(n*n*sizeof(float));
  <base case & partition matrices>
  spawn Mult(C11,A11,B11,n/2);
  spawn Mult(C12,A11,B12,n/2);
  :
  spawn Mult(T21,A22,B21,n/2);
  sync;
  spawn Add(C,T,n);
  sync;
  return;
}
```

8

$$\textbf{\textit{Span:}}\ M_\infty(n) = M_\infty(n/2) + A_\infty(n) + \Theta(1)$$
$$= M_\infty(n/2) + \Theta(\lg n)$$
$$= \Theta(\lg^2 n) \quad \text{— CASE 2}$$

Parallelism of Matrix Multiply

Work: $M_1(n) = \Theta(n^3)$

Span: $M_\infty(n) = \Theta(\lg^2 n)$

Parallelism: $$\frac{M_1(n)}{M_\infty(n)} = \Theta(n^3/\lg^2 n)$$

For $1000 * 1000$ matrices,
parallelism $\sim (10^3)^3/10^2 = 10^7$.

Stack Temporaries

```
cilk void Mult(*C, *A, *B, n) {
  float *T = Cilk_alloca(n*n*sizeof(float));
  h base case & partition matrices i
  spawn Mult(C11,A11,B11,n/2);
  spawn Mult(C12,A11,B12,n/2);
  :
  spawn Mult(T21,A22,B21,n/2);
  sync;
  spawn Add(C,T,n);
  sync;
  return;
}
```

In hierarchical-memory machines (especially chip multiprocessors), memory accesses are so expensive that minimizing storage often yields higher performance.

IDEA: Trade off parallelism for less storage.

No-Temp Matrix Multiplication

```
cilk void MultA(*C, *A, *B, n) {
  // C = C + A * B
  <base case & partition matrices>
  spawn MultA(C11,A11,B11,n/2);
  spawn MultA(C12,A11,B12,n/2);
  spawn MultA(C22,A21,B12,n/2);
  spawn MultA(C21,A21,B11,n/2);
  sync;
  spawn MultA(C21,A22,B21,n/2);
  spawn MultA(C22,A22,B22,n/2);
  spawn MultA(C12,A12,B22,n/2);
  spawn MultA(C11,A12,B21,n/2);
  sync;
  return;
}
```

Saves space, but at what expense?

Work of No-Temp Multiply

```
cilk void MultA(*C, *A, *B, n) {
  // C = C + A * B
  <base case & partition matrices>
  spawn MultA(C11,A11,B11,n/2);
  spawn MultA(C12,A11,B12,n/2);
  spawn MultA(C22,A21,B12,n/2);
  spawn MultA(C21,A21,B11,n/2);
  sync;
  spawn MultA(C21,A22,B21,n/2);
  spawn MultA(C22,A22,B22,n/2);
  spawn MultA(C12,A12,B22,n/2);
  spawn MultA(C11,A12,B21,n/2);
  sync;
  return;
}
```

Work: $M_1(n) = 8\,M_1(n/2) + \Theta(1)$

$= \Theta(n^3)$ — **CASE 1**

Span of No-Temp Multiply

```
cilk void MultA(*C, *A, *B, n) {
  // C = C + A * B
  h base case & partition matrices i
  spawn MultA(C11,A11,B11,n/2);      } maximum
  spawn MultA(C12,A11,B12,n/2);
  spawn MultA(C22,A21,B12,n/2);
  spawn MultA(C21,A21,B11,n/2);
  sync;
  spawn MultA(C21,A22,B21,n/2);      } maximum
  spawn MultA(C22,A22,B22,n/2);
  spawn MultA(C12,A12,B22,n/2);
  spawn MultA(C11,A12,B21,n/2);
  sync;
  return;
}
```

Span: $M_\infty(n) = 2\,M_\infty(n/2) + \Theta(1)$

$= \Theta(n)$ — CASE 1

Parallelism of No-Temp Multiply

Work: $M_1(n) = \Theta(n^3)$

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

Parallelism: $\dfrac{M_1(n)}{M_\infty(n)} = \Theta(n^2)$

For 1000 * 1000 matrices,
parallelism ~ $(10^3)^3/10^3 = 10^6$.

Faster in practice!

Best of Both Worlds

```
cilk void Mult1(*C, *A, *B, n) {// multiply & store
  h base case & partition matrices i
  spawn Mult1(C11,A11,B11,n/2); // multiply & store
  spawn Mult1(C12,A11,B12,n/2);
  spawn Mult1(C22,A21,B12,n/2);
  spawn Mult1(C21,A21,B11,n/2);
  if (SYNCHED) {
    spawn MultA1(C11,A12,B21,n/2); // multiply & add
    spawn MultA1(C12,A12,B22,n/2);
    spawn MultA1(C22,A22,B22,n/2);
    spawn MultA1(C21,A22,B21,n/2);
  } else {
    float *T = Cilk_alloca(n*n*sizeof(float));
    spawn Mult1(T11,A12,B21,n/2); // multiply & store
    spawn Mult1(T12,A12,B22,n/2);
    spawn Mult1(T22,A22,B22,n/2);
    spawn Mult1(T21,A22,B21,n/2);
    sync;
    spawn Add(C,T,n); // C = C + T
  }
  sync;
  return;
}
```

Outline

- **inlet**
- **abort**
- **Cilk_alloca**
- **SYNCHED**
  - **Matrix Multiply example**
- **Cilk_lockvar**

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.

Getting started with Cilk on Ada

1. cp /projects/comp422/pkgs/cilk-5.4.6.tar.gz .
   - Copy of http://supertech.csail.mit.edu/cilk/cilk-5.4.6.tar.gz
2. tar -xzf cilk-5.4.6.tar.gz
3. aclocal
4. automake
5. autoconf
6. ./configure
7. make
8. cd ./examples
9. make
10. Try an example e.g., fib -nproc 4 -stats 1 42

Contact the TA if you run into any problems

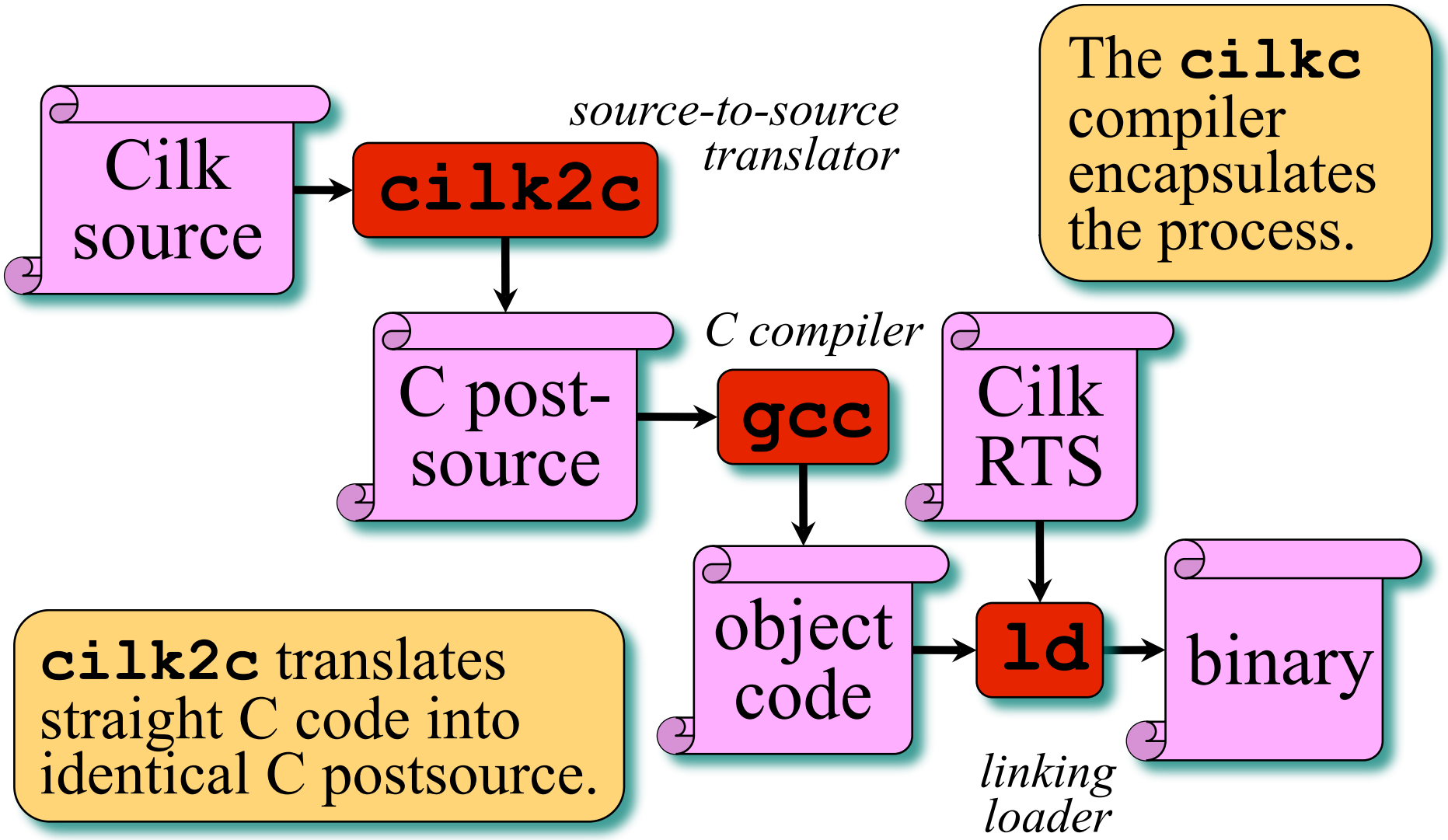
Summary of Today's Lecture

- **Advanced Cilk Topics:**
  - **—inlet, abort, Cilk_alloca, SYNCHED, Cilk_lockvar**

Reading List for Next Lecture (Jan 22nd)

- **Sections 7.10 (OpenMP)**

Compiling Cilk

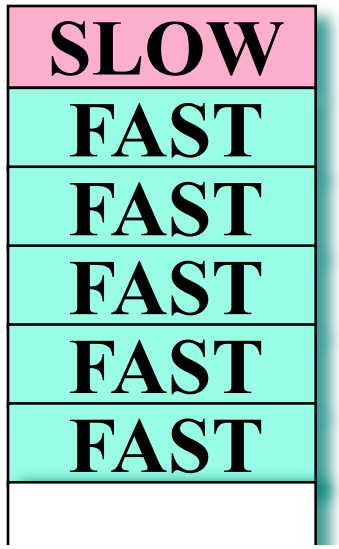


Cilk's Compiler Strategy

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

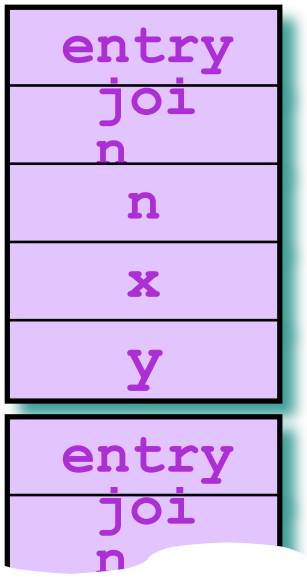
```
frame->entry = 1;
frame->n = n;
push(frame);

x = fib(n-1);

if (pop()==FAILURE) {
   frame->x = x;
   frame->join--;
   h clean up &
     return to scheduler i
}
```

suspend parent

run child

resume parent remotely

frame



Cilk deque

Compiling `sync` — Fast Clone

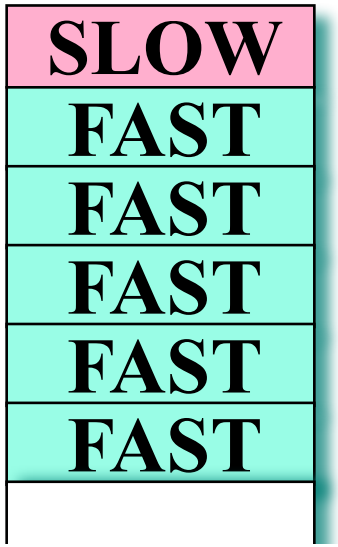
Cilk source

```
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) {
    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--;
    h clean up &
      return to scheduler i
  }

  if (0) {
    L1:;
    n = frame->n;
  }
          :
}
```

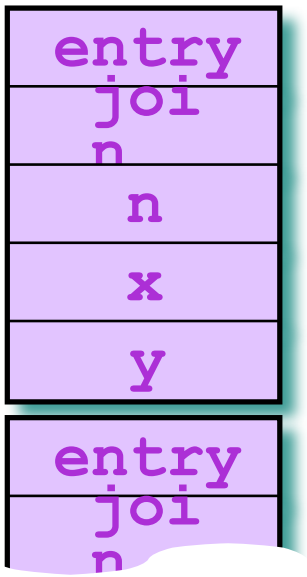
restore program counter

same as fast clone

restore local variables if resuming

continue

frame

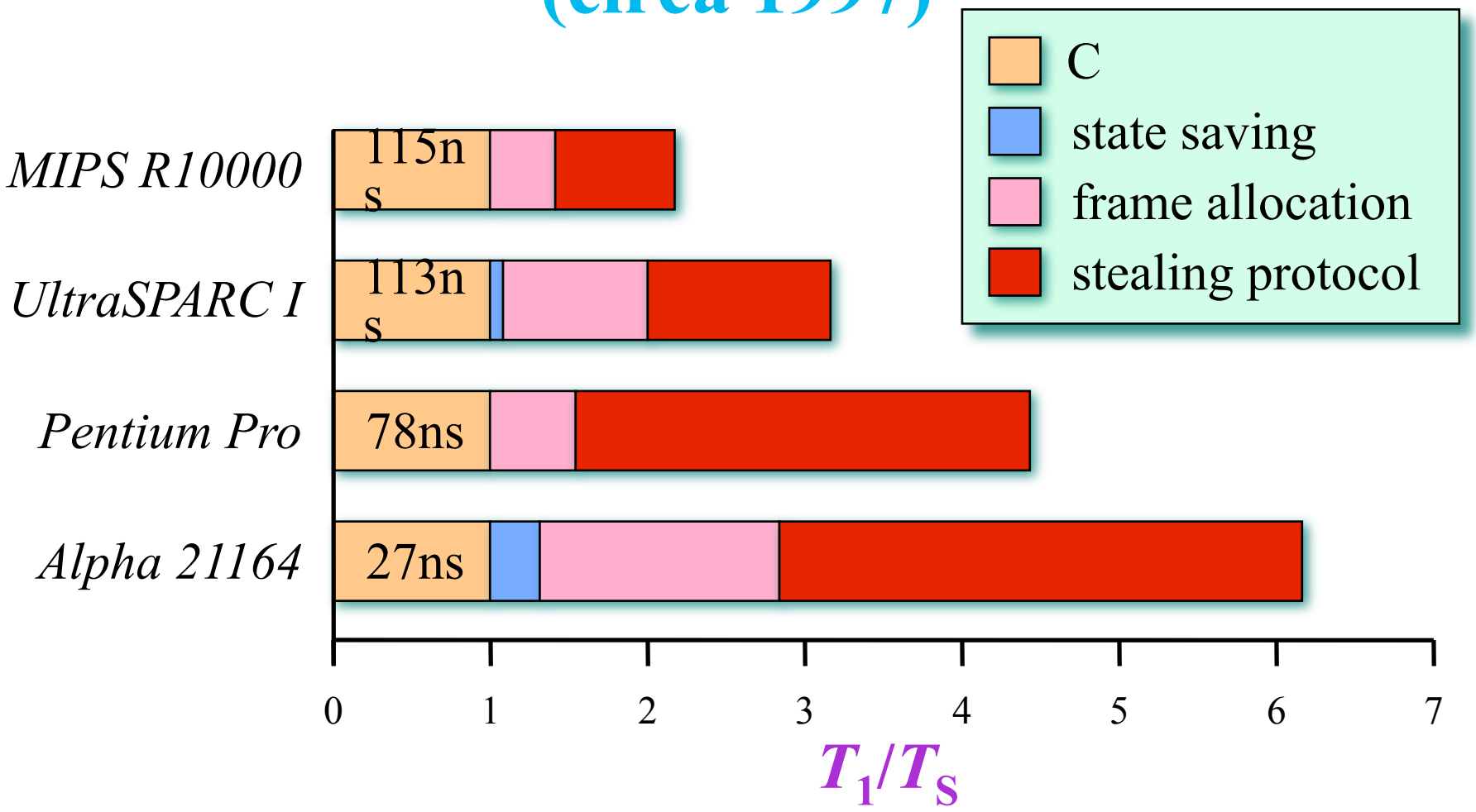


Cilk deque

BACKUP SLIDES START HERE

Breakdown of Work Overhead

(circa 1997)



Benchmark: `fib` on one processor.