The Implementation of the Cilk-5 Multithreaded Language

M. Frigo, C. E. Leiserson, K. H. Randall

PWL: Zurich

Kornilios Kourtis

27 April, 2017

Previously on PWL: Zurich

Today: Cilk-5

- A parallel programming language
- how to write parallel progrems
- how to efficiently execute them
- targets single (shared-memory) machines

- original paper written in 1998

Why this paper?

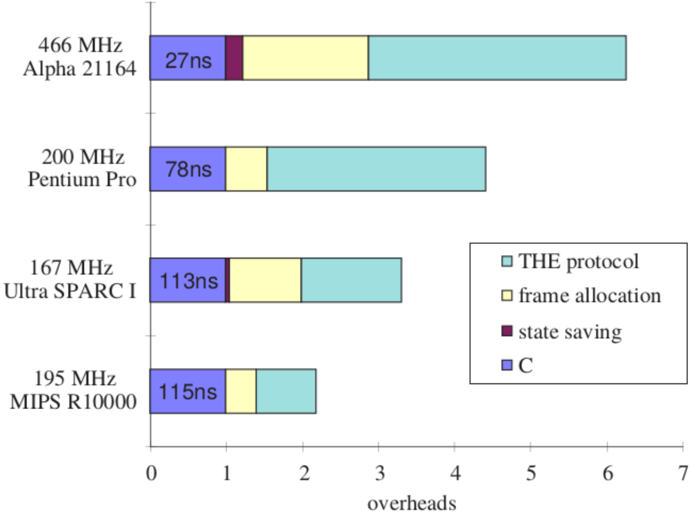
- theory + practice
- topics: parallel programming, synchronization, languages

BACK TO 1998...

back to 1998...

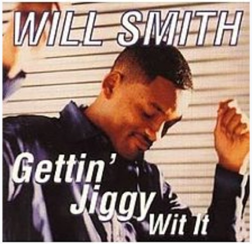
a linguistically simple "inlet" mechanism for nondeterministic control. Cilk-5 is designed to run efficiently on contemporary symmetric multiprocessors (SMP's), which feature hardware support for shared memory. We have coded many applications in Cilk, including the ⋆Socrates and Cilkchess

back to 1998...


466 MHz
Alpha 21164
27ns
200 MHz
Pentium Pro
78ns
167 MHz
Ultra SPARC I
113ns
195 MHz
MIPS R10000
115ns
THE protocol
frame allocation
state saving
C
0
1
2
3
4
5
6
7
overheads

back to 1998...

"Gettin' Jiggy wit It"



Single by Will Smith

from the album *Big Willie Style*

B-side	"Big Willie Style"
Released	January 27, 1998
Format	CD single, Cassette
Recorded	1997

What followed

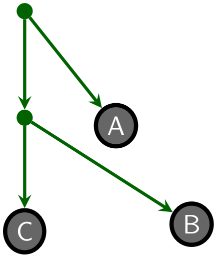
- 2006: Intel releases its first dual-core CPU
- 2006: Cilk Arts commercializes Cilk
- 2008: Cilk++ ships
- 2008: Most influential PLDI paper award for 2008
- 2009: acquired by Intel
- 2010: CilkPlus ships with Intel compiler
- 2014: CilkPlus becomes part of gcc (`-fcilk`)

- Programming model also picked up by other languages: See Fork/Join in Java http://www.oracle.com/technetwork/articles/java/fork-join-422606.html, and OpenMP tasks.

Programming model

- Task parallelism
- Extends C
- two basic keywords: `spawn` and `sync`.

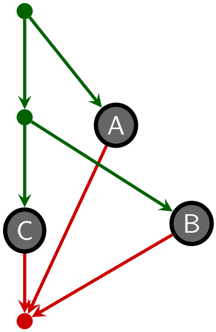
```
/* Cilk example */
x = spawn A();
y = spawn B();
C();
```



Programming model

- Task parallelism
- Extends C
- two basic keywords: `spawn` and `sync`.

```
/* Cilk example */
x = spawn A();
y = spawn B();
C();
sync;
/* x,y are available */
```



Example: run-length encoding (RLE)

(nope, not fib)

IN: `[a,a,a,a,b,b,b,c,c,c,c,c]`

OUT: `[(a,4),(b,3),(c,5)]`

Sequential RLE
(accumulator)

```
def rle(xs):
    ret,curr,freq = ([],xs[0],1)
    for item in xs[1:]:
        if item == curr:
            freq += 1
        else:
            ret.append((curr,freq))
            curr,freq = (item,1)
        ret.append((curr,freq))
    return ret
```

Recursive RLE

```
def rle_rec(xs):
    if len(xs) <= 1:
        return [(xs[0], 1)]

    mid = len(xs) // 2
    rle1 = rle_rec(xs[:mid])
    rle2 = rle_rec(xs[mid:])

    return rle_merge(rle1, rle2)

def rle_merge(rle1,rle2):
    if rle1[-1][0] == rle2[0][0]:
        r1, rle1 = rle1[-1], rle1[:-1]
        r2, rle2 = rle2[0], rle2[1:]
        rle1.append((r1[0],r1[1] + r2[1]))
    return rle1 + rle2
```

Recursive parallel RLE

```
def rle_rec(xs):
    if len(xs) <= 1:
        return [(xs[0], 1)]

    mid = len(xs) // 2
    rle1 = spawn rle_rec(xs[:mid])
    rle2 = spawn rle_rec(xs[mid:])
    sync
    return rle_merge(rle1, rle2)

def rle_merge(rle1,rle2):
    if rle1[-1][0] == rle2[0][0]:
        r1, rle1 = rle1[-1], rle1[:-1]
        r2, rle2 = rle2[0], rle2[1:]
        rle1.append((r1[0],r1[1] + r2[1]))
    return rle1 + rle2
```

Recursive parallel RLE

```
def rle_rec(xs):
    if len(xs) <= 1:
        return [(xs[0],

    mid = len(xs) // 2
    rle1 = spawn rle_rec(xs[:mid])
    rle2 = spawn rle_rec(xs[mid:])
    sync
    return rle_merge(rle1, rle2)

def rle_merge(rle1,rle2):
    if rle1[-1][0] == rle2[0][0]:
        r1, rle1 = rle1[-1], rle1[:-1]
        r2, rle2 = rle2[0], rle2[1:]
        rle1.append((r1[0],r1[1] + r2[1]))
    return rle1 + rle2
```

NOTE: Removing the `spawn` and `sync` keywords results in a valid sequential program. In Cilk, this is called the **C elision**.

product using spawn/sync

```
cilk int product(int *A, int n) {
  if (n == 1)
    return A[0];
  x1 = spawn product(A,n/2);
  x2 = spawn product(A+n/2,n-n/2);
  sync;
  return (x1*x2);
}
```

also: inlet and abort

```
cilk int product(int *A, int n) {
    int p=1;
    inlet void mult(int x) {
        p *= x;
        if (p == 0)
            abort;
        return;
    }
    if (n == 1)
         return A[0];
    mult( spawn product(A, n/2) );
    mult( spawn product(A+n/2, n-n/2) );
    sync;
    return p;
}
```

Executing Cilk programs

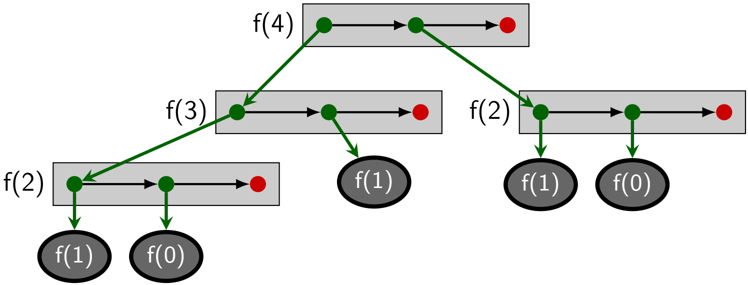
A dynamic graph (tasks, spawns, syncs)

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

Executing Cilk programs

A dynamic graph (tasks, spawns, syncs)

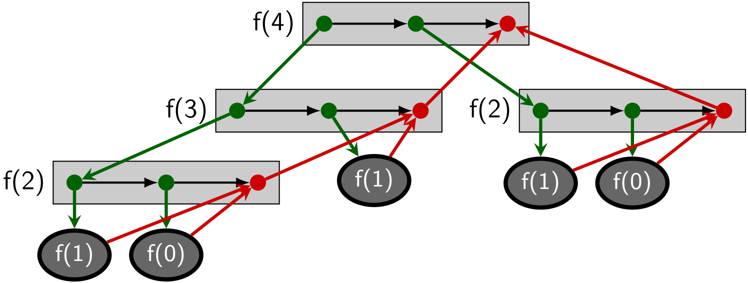
```
cilk int fib(int n) {
    if (n < 2) return (n);
    x = spawn fib(n - 1);
    y = spawn fib(n - 2);
    sync;
    return x + y;
}
```



Executing Cilk programs

A dynamic graph (tasks, spawns, syncs)

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



How do we efficiently execute Cilk programs?

The work-first principle: Minimize the scheduling overhead borne by the work of a computation. Specifically, move overheads out of the work and onto the critical path.

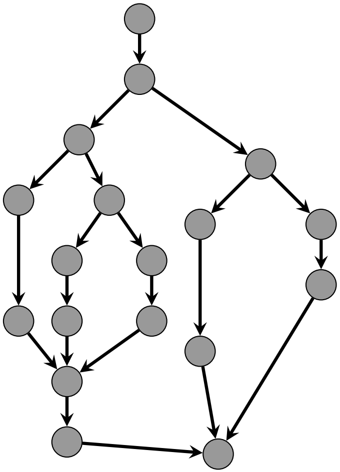
(i.e., make common case fast, push overheads to the rare case)

How do we efficiently execute Cilk programs?

The work-first principle: Minimize the scheduling overhead borne by the work of a computation. Specifically, move overheads out of the work and onto the critical path.

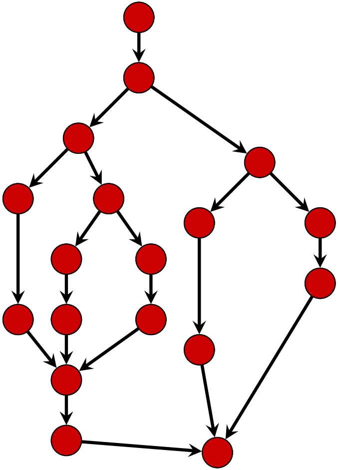
(i.e., make common case fast, push overheads to the rare case)

Performance model



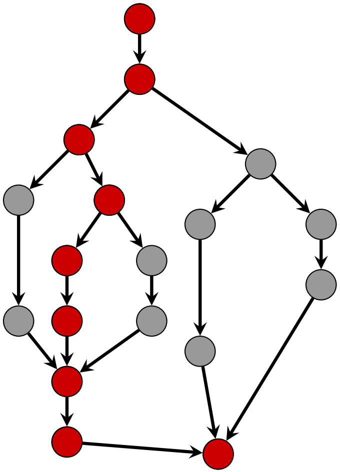
- T_p: Execution time on P CPUs

Performance model



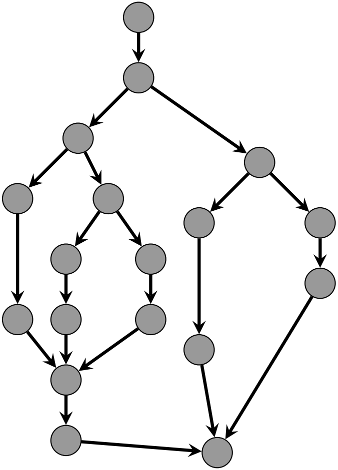
- T_p: Execution time on P CPUs
- T_1: **work**
  (Execution time for all nodes)

Performance model



- T_p: Execution time on P CPUs
- T_1: **work**
  (Execution time for all nodes)
- T_∞: **span / critical path**
  (Execution time for ∞ CPUs)

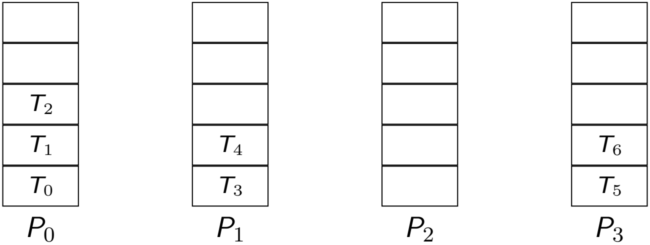
Performance model



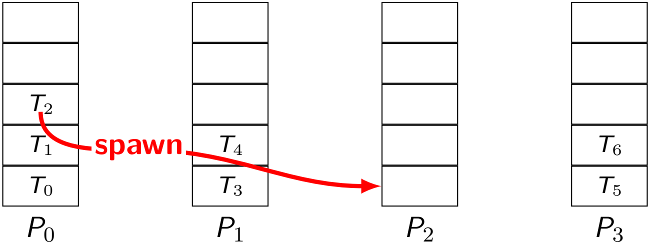
- T_p: Execution time on P CPUs
- T_1: **work**
  (Execution time for all nodes)
- T_∞: **span** / **critical path**
  (Execution time for ∞ CPUs)

- work-first: move overheads to T_∞

Scheduiling tasks

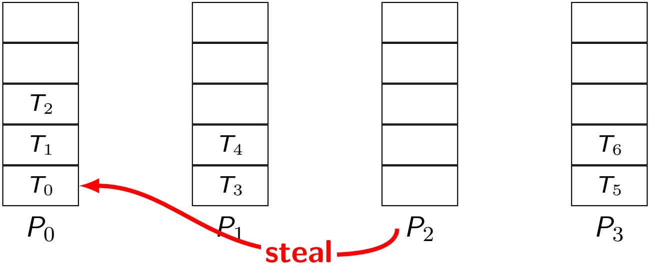


Scheduiling tasks



work sharing: when new tasks are created, scheduler tries to send them to inactive CPUs

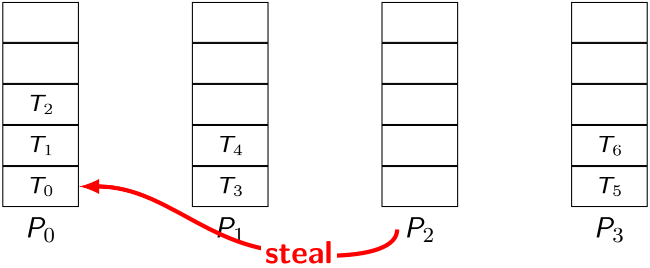
Scheduiling tasks



work stealing: Idle processors try to steal tasks

- child executes first
- tasks are stolen from the bottom
- intuitively similar to DFS (good space properties)
- where do we steal from?

Scheduiling tasks



work stealing: Idle processors try to steal tasks

- child executes first
- tasks are stolen from the bottom
- intuitively similar to DFS (good space properties)
- where do we steal from? **randomly choose**

Cilk's work-stealing scheduler

- $T_p = T_1/P + \mathcal{O}(T_\infty)$
  - (T_1/P and T_∞ are lower bounds)

- requires $S_p \leq PS_1$ stack space

see: Scheduling Multithreaded Computations by Work Stealing, by Blumofe and Leiserson.

Parallel slackness

Cilk's basic assumption

- program parallelism T_1/T_∞
- machine parallelism P

- $P \ll T_1/T_\infty$
- parallel slack: $(T_1/T_\infty)/P$

$$\left.\begin{array}{r} T_p = T_1/P + \mathcal{O}(T_\infty) \\ T_1/P \gg T_\infty \end{array}\right\} \Rightarrow T_p \approx T_1/P \qquad \text{(linear speedup)}$$

- Intiuitively:
  - + program does not depend on number of CPUs
  - + allows better load balancing

Parallel slackness $\Rightarrow$ work-first principle

$$
\begin{aligned}
T_p &\approx T_1/P \\
c_1 = T_1/T_s \Rightarrow T_p &\approx c_1 T_s/P
\end{aligned}
$$

T_s: time of C elision

principle: minimize c_1 even at the expanse of c_∞

(yet, this has limits – authors' discussion on Cilk-4)

Compilation strategy

Two versions of each Cilk function (task)

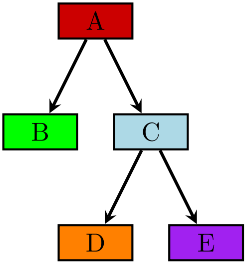
- fast version
  - invariant: never stolen
  - as close to C elision as possible
- slow version
  - when a task is stolen, slow version is executed on thief processor
  - needs to restore state
- Original implementation: cilk2c translation

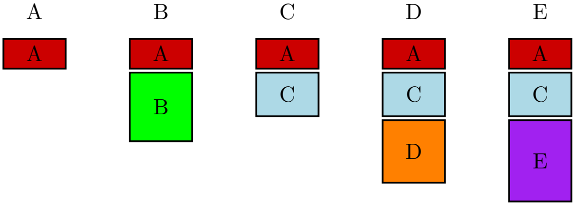
C elision

Only use the execution stack

```
f(n) {
    // spawn f(n-1)
    x = f(n-1);
    // spawn f(n-2)
    y = f(n-2);
    // sync
    return x + y;
}
```

Cactus stack





Cilk fast version

use the execution stack, record state

```
f(n) {
    frame = alloc_frame();
    // spawn f(n-1)
    frame->entry = 1;
    frame->n = n;
    push(frame);
    x = f(n-1);
    frame = pop();
    if (!frame)
      return STOLEN;
    // spawn f(n-2)
    ...
    // sync
    free_frame(frame)
    return x + y;
}
```

Cilk slow version

restore state, sync

```
f_slow(frame) {
    switch (frame->entry) {
        case 1: goto a;
        ...
    }
    // spawn f(n-1)
    ... (same as fast version) ...
    if (0) { a: n = frame->n; }
    // spawn f(n-2)
    ... (same as fast version) ...
    // sync
    wait_for_children();
    free_frame(frame)
    return frame->x + frame->y;
}
```

worker/thief synchronization

- basic scheduler data structure: double-ended queue (*deque*)
- worker: operates on **T**ail: push / pop
- thief: operates on **H**ead: steal
- traditional approach: lock for accessing deque
  - adds overhead to worker!

TH(E) protocol

Worker/Victim

```
push() {
  T++;
}

pop() {
  T--;
  if (H > T) {
    T++;
    lock(L);
    T--;
    if (H > T) {
      T++;
      unlock(L);
      return FAILURE;
    }
    unlock(L);
  }
  return SUCCESS;
}
```

Thief

```
steal() {
  lock(L);
  H++;
  if (H > T) {
    H--;
    unlock(L);
    return FAILURE;
  }
  unlock(L);
  return SUCCESS;
}
```

Note #1: improving performance

```
def rle_rec(xs):
    if len(xs) <= cutoff:
        return rle(xs)
    mid = len(xs) // 2
    rle1 = spawn rle_rec(xs[mid:])
    rle2 = spawn rle_rec(xs[:mid])
    sync
    return rle_merge(rle1, rle2)
```

Note #2: data structures

- efficient partition and concatation
- lists: poor partition
- arrays: poor concatation

- Ropes: an Alternative to Strings
  Boehm et al. (1995)
- Skip lists: a probabilistic alternative to balanced trees
  Pugh (1990)
- implementation: `https://github.com/kkourt/xarray`

Note #3: data vs task parallelism

```
reduce(rle_merge, map(lambda x: [(x,1)], input))
```

(`rle_merge` is an associative operation)

Thank you!

Qs? Pizza?