ALGORITHMIC MULTITHREADED PROGRAMMING USING CILK

Charles E. Leiserson
MIT Laboratory for Computer Science

presented by Arvind

What Is Cilk?

A multithreaded C language with a provably good runtime system.

Platforms

- *Sun UltraSPARC Enterprise*
- *SGI Challenge/Origin*
- *DEC Alphaserver*
- *Intel Quad PC (Linux)*

Applications

- virus shell assembly
- graphics rendering
- numerical linear algebra
- *n*-body simulation
- ⋆Socrates and Cilkchess chess programs

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

Fibonacci

C code

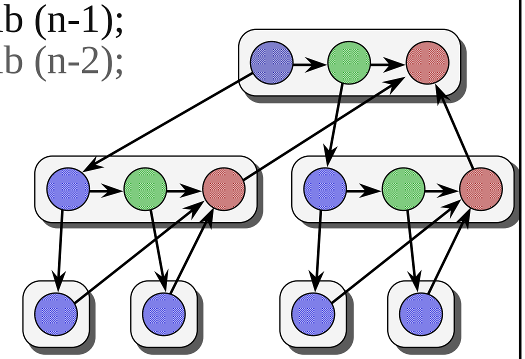
```
int fib (int n)
{     if (n<2) return (n);
      else
      {     int x,y;
            x = fib (n-1);
            y = fib (n-2);
            return (x+y);
      }
}
```

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);
       }
}
```

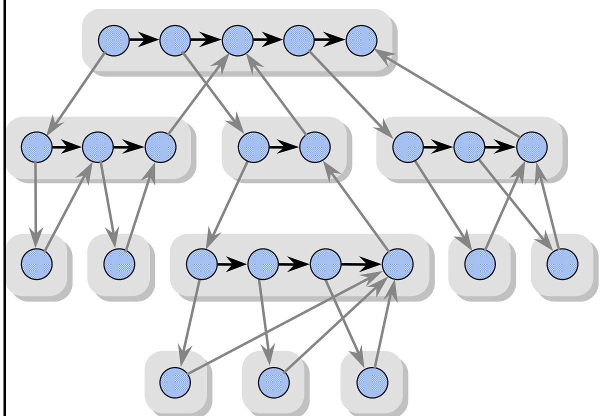

Multithreaded Fibonacci

```
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);
       }
}
```



The computation dag unfolds dynamically.

Multithreaded Computation



A *procedure* is a sequence of atomic ***threads***.

A procedure can ***spawn*** other procedures.

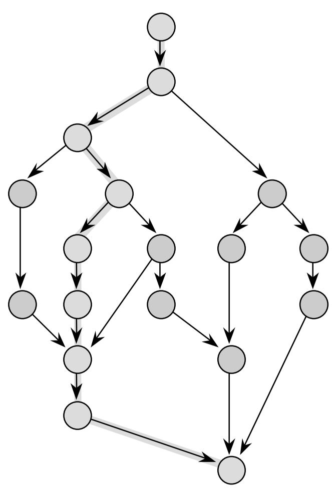
Procedures are organized into a ***tree hierarchy***.

Return values induce additional ***dependences*** among the threads.

The computation unfolds dynamically as a ***dag*** of threads embedded in a tree of procedures.

Algorithmic Complexity Measures

T_P = execution time on P processors



T_1 = *work*

T_∞ = *critical-path length*

T_1/T_∞ = ~~*average parallelism*~~

Lower Bounds

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

Matrix Multiplication

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

$$C = A \times B$$

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

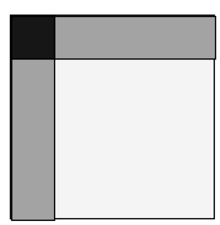
Ordinary Matrix Multiplication

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

IDEA: Spawn n^2 inner products in parallel. Compute each inner product in parallel.

Work: $\Theta(n^3)$
Critical path: $\Theta(\lg n)$
Average parallelism: $\Theta(n^3/\lg n)$

BUT, this algorithm has poor locality and does not exploit the cache hierarchy of modern microprocessors.



Recursive Matrix Multiplication

Divide and conquer on $n \times n$ matrices

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

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

8 multiplications of $(n/2) \times (n/2)$ matrices
1 addition of $n \times n$ matrices

Matrix Multiplication in Cilk

$C = AB$

```
cilk   Mult(*C,*A,*B,n)
  {  float T[n][n];
     «base case and partition matrices»
     spawn  Mult(C11,A11,B11,n/2);
     spawn  Mult(C12,A11,B12,n/2);
     spawn  Mult(C21,A21,B11,n/2);
     spawn  Mult(C22,A21,B12,n/2);
     spawn  Mult(T11,A12,B21,n/2);
     spawn  Mult(T12,A12,B22,n/2);
     spawn  Mult(T21,A22,B21,n/2);
     spawn  Mult(T22,A22,B22,n/2);
     sync  ;
     spawn  Add(C,T,n);
     sync  ;
  }
```

$C = A+B$

```
cilk   Add(*C,*A, n)
  {  «base case and partition matrices»
     spawn  Add(C11,A11,n/2);
     spawn  Add(C12,A12,n/2);
     spawn  Add(C21,A21,n/2);
     spawn  Add(C22,A22,n/2);
     sync  ;
  }
```

Analysis of Matrix Multiplication

Matrix addition:

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

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

Matrix multiplication:

$$M_1(n) = 4M_1(n/2) + \Theta(n^2) = \Theta(n^3)$$

$$M_\infty(n) = M_\infty(n/2) + \Theta(\lg n) = \Theta(\lg^2 n)$$

Average parallelism:

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

For 1000×1000 matrices, average parallelism $\approx 10^7$.

Stack Temporaries

```
cilk  Mult(*C,*A,*B,n)
  { float T[n][n];
    «base case and partition matrices»
    spawn  Mult(C11,A11,B11,n/2);
    spawn  Mult(C12,A11,B12,n/2);
    spawn  Mult(C21,A21,B11,n/2);
    spawn  Mult(C22,A21,B12,n/2);
    spawn  Mult(T11,A12,B21,n/2);
    spawn  Mult(T12,A12,B22,n/2);
    spawn  Mult(T21,A22,B21,n/2);
    spawn  Mult(T22,A22,B22,n/2);
    sync  ;
    spawn  Add(C,T,n);
    sync  ;
  }
```

In modern hierarchical-memory microprocessors, memory accesses are so expensive that higher performance often results from minimizing storage.

No-Temp Matrix Multiplication

```
cilk Mult2(*C,*A,*B,n)
  /* C = C + A*B */
  {  «base case and partition matrices»
     spawn Mult2(C11,A11,B11,n/2);
     spawn Mult2(C12,A11,B12,n/2);
     spawn Mult2(C21,A21,B11,n/2);
     spawn Mult2(C22,A21,B12,n/2);
     sync;
     spawn Mult2(C11,A12,B21,n/2);
     spawn Mult2(C12,A12,B22,n/2);
     spawn Mult2(C21,A22,B21,n/2);
     spawn Mult2(C22,A22,B22,n/2);
     sync;
     return;
  }
```

Saves space at the expense of critical path.

Analysis of No-Temp Multiply

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

Critical path:

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

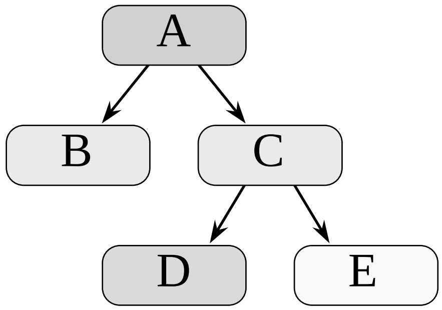
- $f(n) = \Theta(1)$, $n^{\log_b a} = n^{\lg 2} = n \Rightarrow$ Case 1

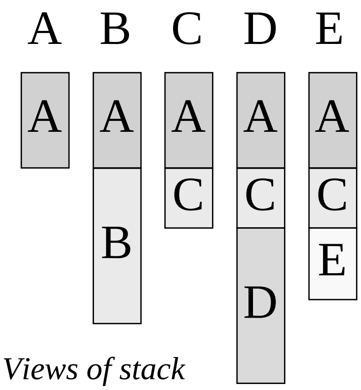
Average parallelism:

$$\frac{M_1(n)}{M_\infty(n)} = \Theta(n^2)$$

For 1000×1000 matrices, average parallelism $\approx 10^6$.

Cactus Stack





Views of 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.

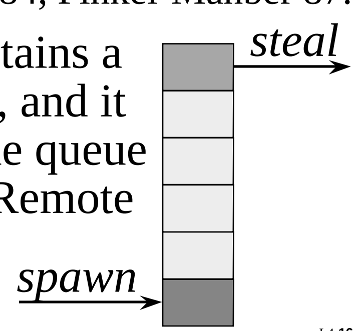
Cilk's *cactus stack* supports several views in parallel.

Cilk's Work-Stealing Scheduler

IDEA: When a procedure spawns, its processor posts the work locally. When an idle processor runs out of work, it steals work from another processor chosen uniformly at random.

Origins: Burton-Sleep 81, Halstead 84, Finkel-Manber 87.

In Cilk, each processor maintains a work queue of ready threads, and it manipulates the bottom of the queue like a procedure call stack. Remote processors steal work from the top of the queue.

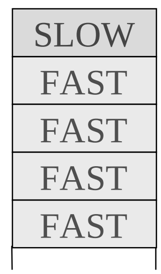


Cilk's Runtime System

The cilk2c compiler generates two clones of each procedure:

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

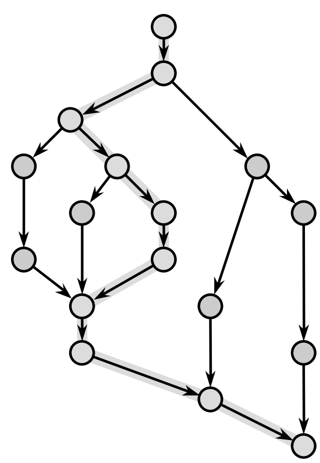
- The fast clone is always spawned.
- The slow clone is resumed if a thread is stolen.
- A check is made whenever a procedure returns to see if the resuming parent thread has been stolen.



Greedy Scheduling

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

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



Corollary. Linear speedup when $P \le T_1/T_\infty$.

Cilk Performance

Instrumentation in Cilk provides accurate measures of work and critical path.

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).

Near-perfect linear speedup if $P \ll T_1/T_\infty$.

The average cost of a spawn in Cilk-5 is about 4 times the cost of a C function call.

Cilk-5 Benchmarks

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	.0015	96898	0.99	18.8	8.0
cilksort	4.1M	5.4	.0048	1125	1.21	0.9	6.0
knapsack	30	682	.0017	392343	1.21	85.0	8.0
lu	2048	155.8	.4161	374	1.02	20.3	7.7
cholesky	1.02M	1427	4.2	340	1.25	208	6.9
heat	2M	62.3	.16	389	1.08	9.4	6.6
fft	1M	4.3	.0024	1792	0.93	0.77	5.6
barnes-hut	65536	124	9.32	13	1.02	24.9	5.0

All benchmarks were run on a Sun Enterprise 5000 SMP with 8 167-megahertz UltraSPARC processors. All times are in seconds.

Performance Bounds

Theorem. For any number P of processors and any computation with work T_1 and critical path T_∞, Cilk's scheduler achieves the following bounds:

Space: $S_P \le S_1 P$.

Time: $T_P = T_1/P + O(T_\infty)$ expected.

For any $\varepsilon>0$, the probability is at least $1-\varepsilon$ that $T_P = T_1/P + O\big(T_\infty + \lg P + \lg(1/\varepsilon)\big)$.

Communication: $O(T_\infty P S_{max})$ expected,

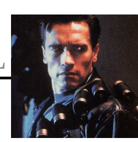
where S_{max} is the largest activation record size.

Debugging Support

Cilk provides a novel debugging tool called the *Nondeterminator* for detecting and localizing determinacy-race bugs.

Cilk program + Input data set

Information localizing a determinacy race. ← FAIL



PASS → *Every* scheduling produces the same result.

A *determinacy race* occurs whenever parallel threads access the same location and one of the threads performs a write.