Massachusetts Institute of Technology
6.892: Theory of Parallel Systems
Professor Charles E. Leiserson

Lecture 7
February 27, 1997
Scribe: Michael Duff

Lecture 7: Algorithmic Multithreaded Computing

1 Multithreading

People usually employ the term "multithreading" to signify at least three different things:

Architecture – featherweight (threads at the instruction level)

- Superscalar, VLIW, dataflow processors

Operating Systems – heavy/middleweight (threads at the program level)

- Processes, OS threads

Languages – lightweight (threads at the procedural level)

- Dataflow: Id, pH
- Parallel Lisps: Multilisp, Balinda
- Algorithmic: **Cilk**

2 Multithreaded Computation

In multithreaded computation, once a procedure is called, the parent procedure continues to execute until a synchronization is required. Thus, the computation unfolds dynamically as a DAG of threads embedded in a tree of procedures (see Figure 1).

- A *procedure* is a sequence of non-blocking *threads*.
- A procedure can *spawn* other procedures.
- Procedures are organized into a *tree hierarchy*.
- Return values induce additional *dependencies* among the threads. For example, if the argument to a procedure is the return value of a previously-called procedure, then the second procedure cannot execute until the first has completed.

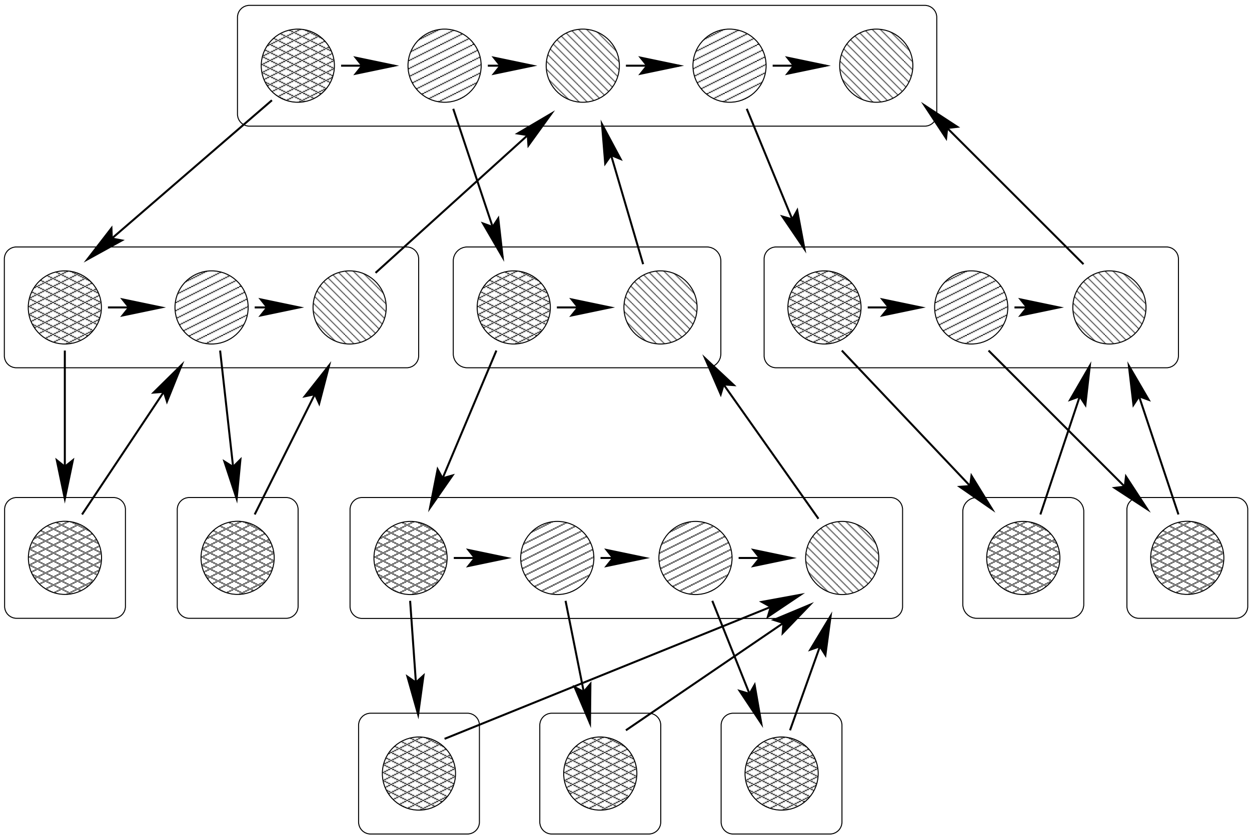

Figure 1: Multithread Computation DAG. In this diagram, procedures are represented as rounded rectangles and threads are shown as round nodes. Downward edges are spawns, horizontal edges represent sequential dependencies within a procedure, and returns are shown as upward arrows.

3 Multithreaded Programs

An example of an algorithm that lends itself to multithreaded execution is a generic divide and conquer algorithm (e.g. merge sort).

```
DivideAndConquer ( A )
{
    if ( A is small ) return ( f(A) );
    A1, A2 = divide ( A );

    S1 = spawn DivideAndConquer ( A1 );
    S2 = spawn DivideAndConquer ( A2 );
    sync;

    S = merge ( S1, S2 );
    return ( S );
}
```

The first line handles the base case of the algorithm and is executed at a leaf of the dag and returns. If the base case does not apply, then the problem is divided into two sub problems (preferably of equal size). Next, a new thread is spawned to begin working recursively on the first subproblem. The difference between serial and multithreaded programs such as a Cilk program is that the parent can continue executing. As a result, another thread is spawned to work on the second subproblem in parallel with the first. When the **sync** is reached, the parent waits for its spawned children to return. Once all of the children at that level returned, the parent can continue execution. Next, for the divide and Conquer algorithm, the results of the two subproblems are merged and the solution is returned.

The diagram in Figure 2 depicts the dag which is generated for the divide and conquer algorithm.

4 Cilk

A multithreaded C language with a provably good runtime system.

The Cilk language was developed primarily at LCS under the guidance of Professor Leiserson. Cilk is still evolving and is currently at version 5.

Some of the applications for Cilk are: protein folder, virus shell assembly, graphics rendering, numerical linear algebra, n-body simulation, *Socrates chess program.

Over his history, Cilk has been supported on the following platforms:

MPP's: CM5, Paragon, SP2

SMP's: SGI, Sun

NOW's: PVM, Cilk-NOW

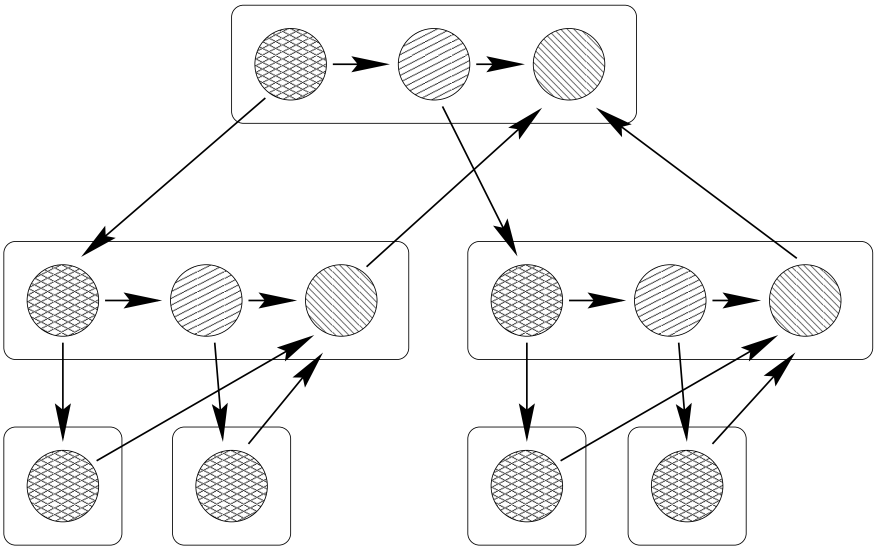

Figure 2: Divide and conquer

Cilk automatically manages low-level aspects of parallel execution, including protocols and scheduling. One of Cilk's design goals was to impose no protocols in the Cilk programming language so the programmer does not need to be concerned with such low-level operations. Cilk therefore follows lexical, not dynamic, specification of correctness.

The Cilk language has the serial semantics of C. By removing the Cilk keywords, an ordinary C program remains which can be run serially. Compare the following two implementations of a procedure to calculate the Fibonacci of an argument n.

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);
    }
}
```

The corresponding C code would then be:

```
int fib (int n)
{   if (n<2) return (n);
```

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

5 Evolution of Cilk

Below is the evolution of Cilk. The primary additions to the language for each new version are noted:

Cilk-1 [*September 1994*] Explicit continuation passing

Cilk-2 [*May 1995*] Call/return semantics

Cilk-3 [*October 1995*] Dag-consistent distributed shared memory

Cilk-4 [*June 1996*] New RTS, inlets, aborts, rudimentary I/O

Cilk-5 [*March 1997*] Faster, more portable runtime system. *Nondeterminator* race detection tool.

You can read more about Cilk on the web at following URL:

`http://theory.lcs.mit.edu/~cilk`

6 Cilk's Work-Stealing Scheduler

IDEA: When a procedure spawns, its processor posts the work locally by growing the stack just as in regular C execution. When an idle processor runs out of work, it steals work from another processor chosen uniformly at random. This method turns out to be nearly optimal.

Another option for distributing work in a multiprocessor is for a processor that has extra work to attempt to find another processor that is not busy. This approach, in general, is disadvantageous because a processor is communicating with others the most when it is the busiest. With Cilk's work-stealing scheduler, on the other hand, no interprocessor communication is necessary if the processors are busy.

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 (see Figure 3).

Cilk's scheduler allows Cilk programs to run efficiently regardless of the number of processors available. When the program is running on a single processor, for example, the thread queue acts like a traditional stack.

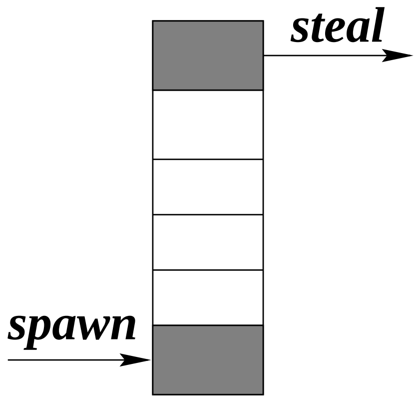

Figure 3: Cilk Scheduler

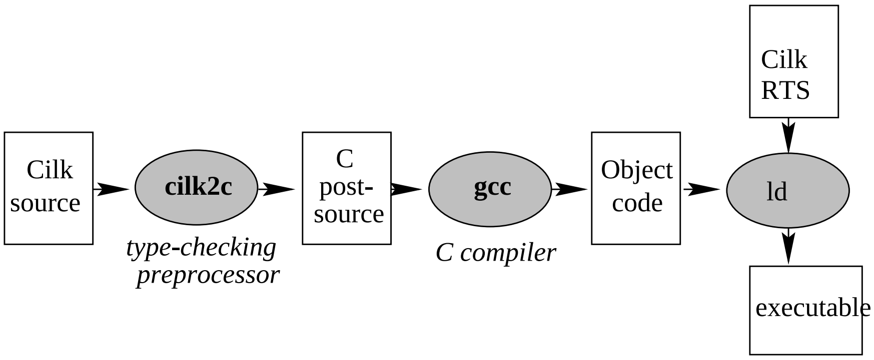

Figure 4: Compiling Cilk-4 Code

7 Compiling Cilk-4 Code

Cilk-4 code is compiled in a 2-step process. First, a source-to-source translator (cilk2c) transforms the Cilk code into common C code. Once in C, a traditional C compiler (e.g. gcc) is used to compile the program into object code.

cilk2c generates two clones of each procedure:

- Fast clone - serial, common-case code
- Slow clone - performs parallel bookkeeping

The fast clone is always spawned. It is a more compact version of the procedure and does not include the overhead needed to manage parallel book-keeping. Note that in the fast clone, a *sync* is basically a NOP. Cilk developers have achieved a very low overhead for the Cilk spawns - only approximately 3 times the overhead of a C procedure call.

SLOW
FAST
FAST
FAST
FAST

Figure 5: Cilk stack

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.

8 Cilk Chess Programs

Cilk has been applied to many problems, but the most well-known applications are the chess-playing programs implemented in Cilk. These programs have performed extremely well in international contests since Cilk-1 was released in 1994.

- *Socrates placed 3rd in the 1994 International Computer Chess Championship running on NCSA's 512-node CM5. The program was ported to Cilk-1 from a skeletal, serial C program (Dailey, Kaufman) in just over a month.
- *Socrates 2.0 took 2nd place in the 1995 World Computer Chess Championship in Hong Kong running on Sandia National Labs' 1824-node Intel Paragon. The port of Cilk-1 from the CM5 to the Paragon took under a week.
- The Cilk group's newest chess program, Cilkchess, which is programmed in Cilk-4 running on a 12-processor Sun Ultrasparc SMP, took first prize in the 1996 Dutch Open Computer Chess Tournament with 10 points out of 11.

9 Algorithmic Complexity Measures

Recall:

$$T_P = \textbf{execution time on P processors}$$
$$T_1 = \textbf{work}$$
$$T_\infty = \textbf{critical-path length}$$
$$T_1/T_\infty = \textbf{average parallelism}$$

Lower Bounds:

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

10 Greedy Scheduling

Brent's Theorem

For any dag with work T_1 and critical-path length T_∞, a greedy scheduler achieves

$$T_P \leq T_1/P + T_\infty \ .$$

Proof. At any point in time, one of the following situations occurs: either P processors are kept busy by the scheduler, or less than P processors are busy. There can be at most T_1/P steps of the first kind. Moreover, we know that every step of the second kind reduces the depth of the dag by at least one, since all the nodes with in-degree zero are executed during a step of the second kind. Therefore, there are at most T_∞ steps of the second kind. The total number of steps is therefore $T_P \leq T_1/P + T_\infty$. □

Corollary: Any greedy scheduler achieves linear speedup when $P = O(T_1/T_\infty)$.
Proof. We have $T_P \leq T_1/P + T_\infty = O(T_1/P)$. □

11 Benchmarks

The benchmarks in Table 1 were compiled by running Cilk-1 code on a CM5. Note the low efficiency (T_{serial}/T_1) for the *fib* benchmark. This was a result of the large overhead of spawns in Cilk-1 coupled with the trivial work performed as the base case of this algorithm.

Observe that $T_1/P + T_\infty$ is a good approximation of T_P. The only exception is the T_P entry for knary on 256 processors. In this case, however, observe that $T_1/P + 2T_\infty$ yields an estimate which is bigger than the measured value 5.687, indicating that the coefficient of the critical-path is between 1 and 2.

12 Cilk Performance

Using the work-stealing approach, when there is not enough work to do, the processors are making progress along the critical path. Instrumentation in the Cilk scheduler provides an accurate measure of work and critical path. Cilk's scheduler achieves:

- $T_P = T_1/P + O(T_\infty)$ expected time (provably)
- $T_P \approx T_1/P + T_\infty$ time (empirically)

	fib (33)	queens (15)	pfold (3,3,4)	ray (500, 500)	knary (10, 5, 2)
T_{serial}	8.487	252.1	615.15	729.2	288.6
T_1	73.16	254.6	647.8	732.5	314.6
T_{serial}/T_1	0.116	0.9902	0.9496	0.9955	0.9174
T_∞	0.000326	0.0345	0.04354	0.0415	4.458
T_1/T_∞	224417	7380	14879	17650	70.56
thread len	4.276μs	1208μs	68.08μs	1726μs	53.69μs
32 processors					
T_P	2.298	8.012	20.26	21.68	15.13
$T_1/P + T_\infty$	2.287	7.991	20.29	22.93	14.28
space	70	95	47	39	41
requests	185.8	48.0	88.6	218.1	92639
steals	56.63	18.47	26.06	79.25	18031
256 processors					
T_P	0.2892	1.045	2.590	2.765	8.590
$T_1/P + T_\infty$	0.2861	1.029	2.574	2.903	5.687
space	66	76	47	32	48
requests	73.66	80.40	97.79	82.75	151803
steals	24.10	21.20	23.05	18.34	6378

Table 1: Table of Cilk-1 benchmarks

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

Linear speedup is desirable because adding additional processors decreases the running time of the program proportionally.

Note that the average cost of a spawn in Cilk-4 is less than 3 times the cost of a C function call. However, it is extremely important to avoid overheads (i.e. a constant factor in the T_1/P term). In Cilk, overheads were amortized against the critical path, not the work, in order to keep the constant factor associated with the T_1/P factor as close to 1 as possible.

Note also that the T_∞ term here is really $O(T_\infty)$, and a constant factor here does not affect the performance of the Cilk scheduler very much.

13 *Socrates Speedup Anomaly

$$\boldsymbol{T_P = T_1/P + T_\infty}$$

Original Program	***Proposed Program***
$\boldsymbol{T_{32} = 65}$ seconds.	$\boldsymbol{T_{32} = 40}$ seconds.
$\boldsymbol{T_1 = 2048}$ seconds	$\boldsymbol{T_1 = 1024}$ seconds
$\boldsymbol{T_\infty = 1}$ second.	$\boldsymbol{T_\infty = 8}$ seconds.
$\boldsymbol{T_{32} = 2048/32 + 1}$	$\boldsymbol{T_{32} = 1024/32 + 8}$
$\boldsymbol{= 65}$ seconds.	$\boldsymbol{= 40}$ seconds.
$\boldsymbol{T_{512} = 2048/512 + 1}$	$\boldsymbol{T_{512} = 1024/512 + 8}$
$\boldsymbol{= 5}$ seconds.	$\boldsymbol{= 10}$ seconds.

A clever optimization was proposed for the *Socrates chess-playing program, which decreased the total work substantially at the cost of increasing the critical path. Brent's theorem was applied to predict the running time on a 512-processor system compared to a 32-processor machine. In this case, the large critical path dominated over the multiprocessor speedup and would have caused the proposed program to run more slowly than the original on the 512-processor system. This example emphasizes the importance of using Brent's theorem to predict the performance of an algorithm on a P-processor machine.