Parallel & Concurrent Programming:
Cilk

Emery Berger

CMPSCI 691W
Spring 2006

Outline

- So far:
  - Programming with threads, etc.
    - POSIX, Java
  - **Implicitly** parallel programming language: Flux
- This time:
  - Cilk: **explicitly** parallel programming language

Parallel Programming

- Decomposition
  - into parallel threads
- Mapping
  - of threads to processors
- Communication
  - to move data across threads
- Synchronization
  - among threads

Goals of Parallel Models

- Simplify software development
- Architecture independence
  - Lifespan of parallel architectures…
- Understandable
- Provide guaranteed performance
- Ease of use
  - Conceal decomposition, mapping, communication, & synchronization!

Taxonomy of Languages

- Fully abstract (Haskell, Unity)
- Explicit **parallelism** (Multilisp, Fortran+, NESL)
- + explicit **decomposition** (CODE, **Flux**)
- + explicit **mapping** (Linda)
- + explicit **communication** (static dataflow)
- + **synchronization** (everything explicit)
  (MPI, `fork()`, **Java**, **POSIX threads**, Ada, occam)
  - *message passing, shared memory, rendezvous*

- structure, communication: *dynamic* | *limited*

Cilk

- Explicit everything *except mapping*
- Extension of C for parallel programming
  - Shared memory only
- Benefits:
  - Provably-efficient work stealing scheduler
    - "Performance guarantees"
  - Clean programming model
- Implemented as source-to-source compiler generating C

Fibonacci Example

```
int main
(int argc, char *argv[])
{
  int n, result;
  n = atoi(argv[1]);
  result = fib(n);

  printf("Result:%d\n",
         result);
  return 0;
}
```

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

    return (x+y);
  }
}
```

Fibonacci in Cilk

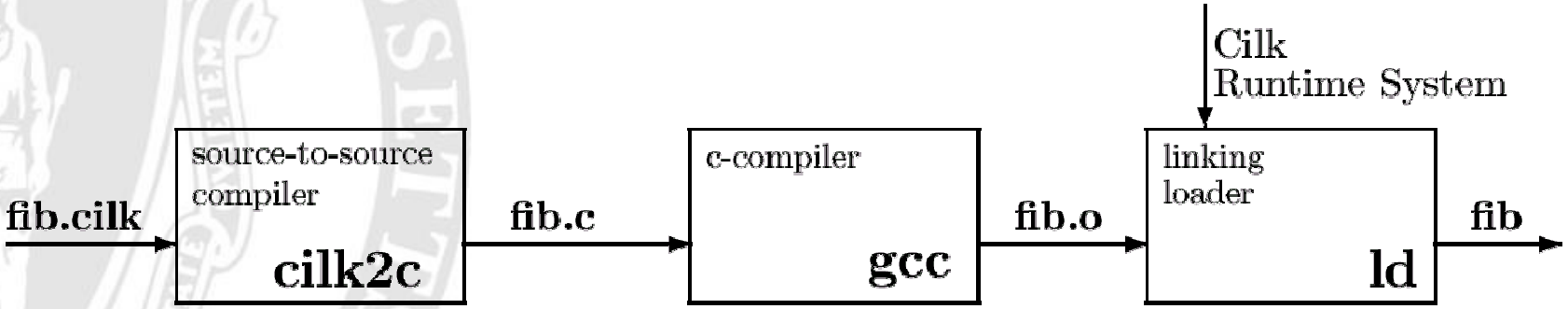
```
cilk int main
(int argc, char *argv[])
{
  int n, result;
  n = atoi(argv[1]);
  result = spawn fib(n);
  sync;
  printf("Result:%d\n",
         result);
  return 0;
}
```

```
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);
  }
}
```

Other Extensions

- **Inlets**
  - Atomic execution
  - Implicit in calls like `x += spawn fib(n-1)`
- **Abort**
  - Terminates work no longer needed (e.g. for parallel search)
- **Locking**
  - Access to shared data (sigh)

Compiling Cilk



- Inserts calls to runtime system:
  - Executes threads
  - Distributes work (**work-stealing scheduling**)

Work-First Principle

- **Work** = amount of time needed to execute the computation serially
- **Critical path length** = execution time on infinite number of processors
- **Work-First Principle**:
  - Minimize scheduling overhead by possibly increasing critical path

Work-First Principle

- T_P = time on P processors:
  - $T_P = T_1/P + O(T_\infty)$
  - $T_P \leq T_1/P + c_\infty T_\infty$
- *Average parallelism (max speedup)*
  - $P_{AVERAGE} = T_1/T_\infty$
- *Parallel slackness*
  - $P_{AVERAGE}/P$

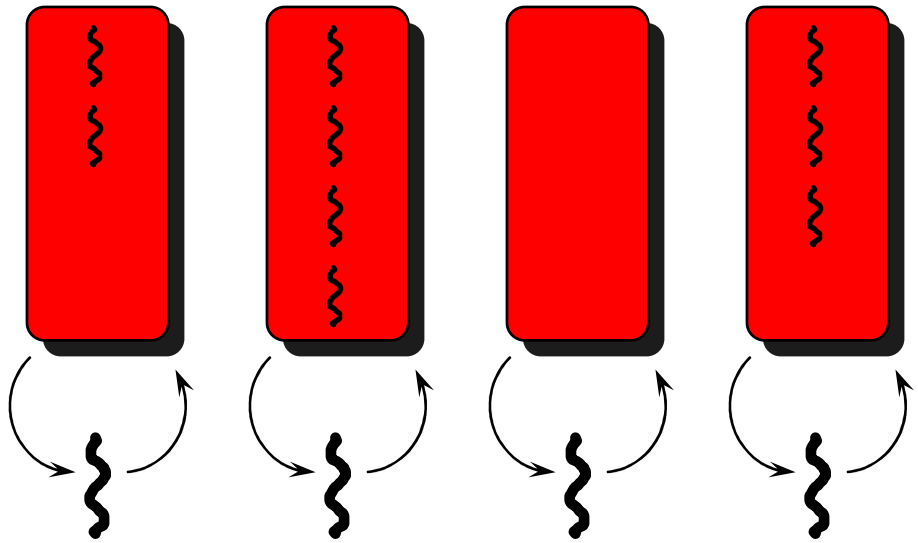
Work-First Principle, II

- Assumption of parallel slackness:
  - $P_{AVERAGE}/P \gg c_\infty$
- Combining these with inequality:
  - $T_P \approx T_1/P$
- Work overhead:
  - $c_1 = T_1/T_S$
- $T_P \approx c_1 T_S / P$

- *Conclusion:* Minimize work overhead

Work-Stealing

- *Ready deque* of threads
- Workers treat deque as stack, pushing and popping calls onto **bottom**



- Out of work: **steal** from **top** of another workers' deque
  - parents stolen before children
  - **asymptotically optimal – greedy schedule**
- Implemented using two versions of each procedure: *fast clone* for common case & *slow clone* for steals

Fast Clone

- Run when procedure spawned
  - Little support for parallelism
- Whenever call is made:
  - Save complete state
  - Push onto bottom of deque
- When call returns:
  - Check to see if procedure was stolen
  - If stolen, return immediately
  - If not stolen, continue execution
- Children never stolen $\Rightarrow$ sync = no-op

Fast Clone Example

```
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);
  }
}
```

Fast Clone Example

```
1  int fib (int n)
2  {
3    fib_frame *f;              frame pointer
4    f = alloc(sizeof(*f));     allocate frame
5    f->sig = fib_sig;          initialize frame
6    if (n<2) {
7      free(f, sizeof(*f));     free frame
8      return n;
9    }
10   else { … }
```

Fast Clone Example

```
11     int x, y;
12     f->entry = 1;              save PC
13     f->n = n;                  save live vars
14     *T = f;                    store frame pointer
15     push();                    push frame
16     x = fib (n-1);             do C call
17     if (pop(x) == FAILURE)     pop frame
18       return 0;                procedure stolen
19     < … >                      second spawn
20     ;                          sync is free!
21     free(f, sizeof(*f));       free frame
22     return (x+y);
23 } }
```

Slow Clone

- Used when procedure stolen
  - Similar to fast clone,
    but supports concurrent execution
- Restores program counter &
  procedure state using copy stored on
  deque
- Calling **sync** makes call to runtime
  system to check on children's status

The T.H.E. Protocol

- Deques held in shared memory
  - Workers operate at bottom, thieves at top
- Must prevent race conditions where thief and victim try to access same procedure frame
- Locking deques would be expensive for workers
  - Violates work-first principle
- T.H.E Protocol removes overhead of common case (no conflict)

The T.H.E. Protocol

- Assumes only reads and writes atomic
- Head of the deque is H, tail is T, and (T ≥ H)
  - Only thief can change H
  - Only worker can change T
- To steal, thieves must get the lock L.
  - At most two processors operating on deque
- Three cases of interaction:
  - Two or more items on deque – each gets one
  - No items on deque – both worker and thief fail
  - One item on deque – either worker or thief gets frame, but not both

One item on deque case

- Both thief and worker assume they can get a procedure frame and change H or T
- Both thief and worker try to get frame:
  - One or both will discover H > T, depending on instruction order.
  - If thief discovers (H > T):
    - Backs off and restores H
  - If worker discovers (H > T):
    - Restores T, and then tries for the lock
    - Inside lock, procedure can be safely popped if still there

T.H.E. Protocol

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

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

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

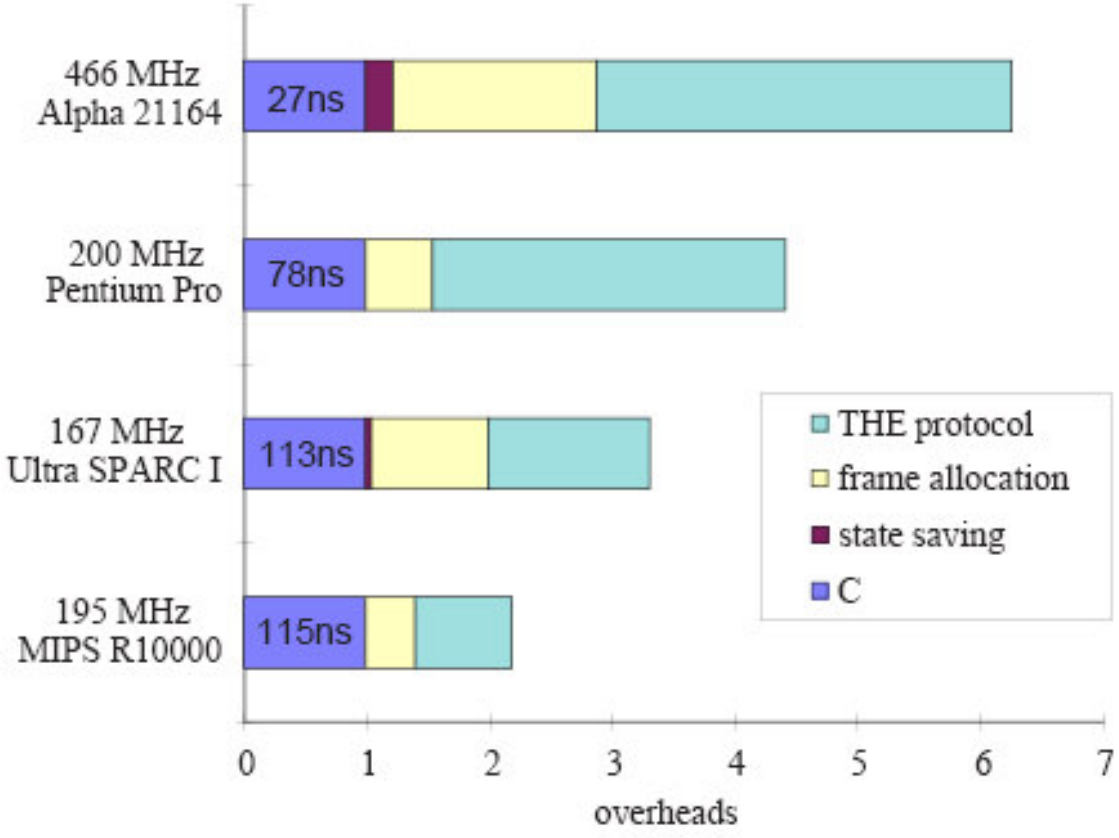
Empirical Results

- 8X Sun SMP:
  average speed up of 6.2 vs. elision (serial C non-threaded versions).
- Assumptions of work-first:
  - Applications tested all showed high amounts of "average parallelism"
  - Work overhead small for most programs

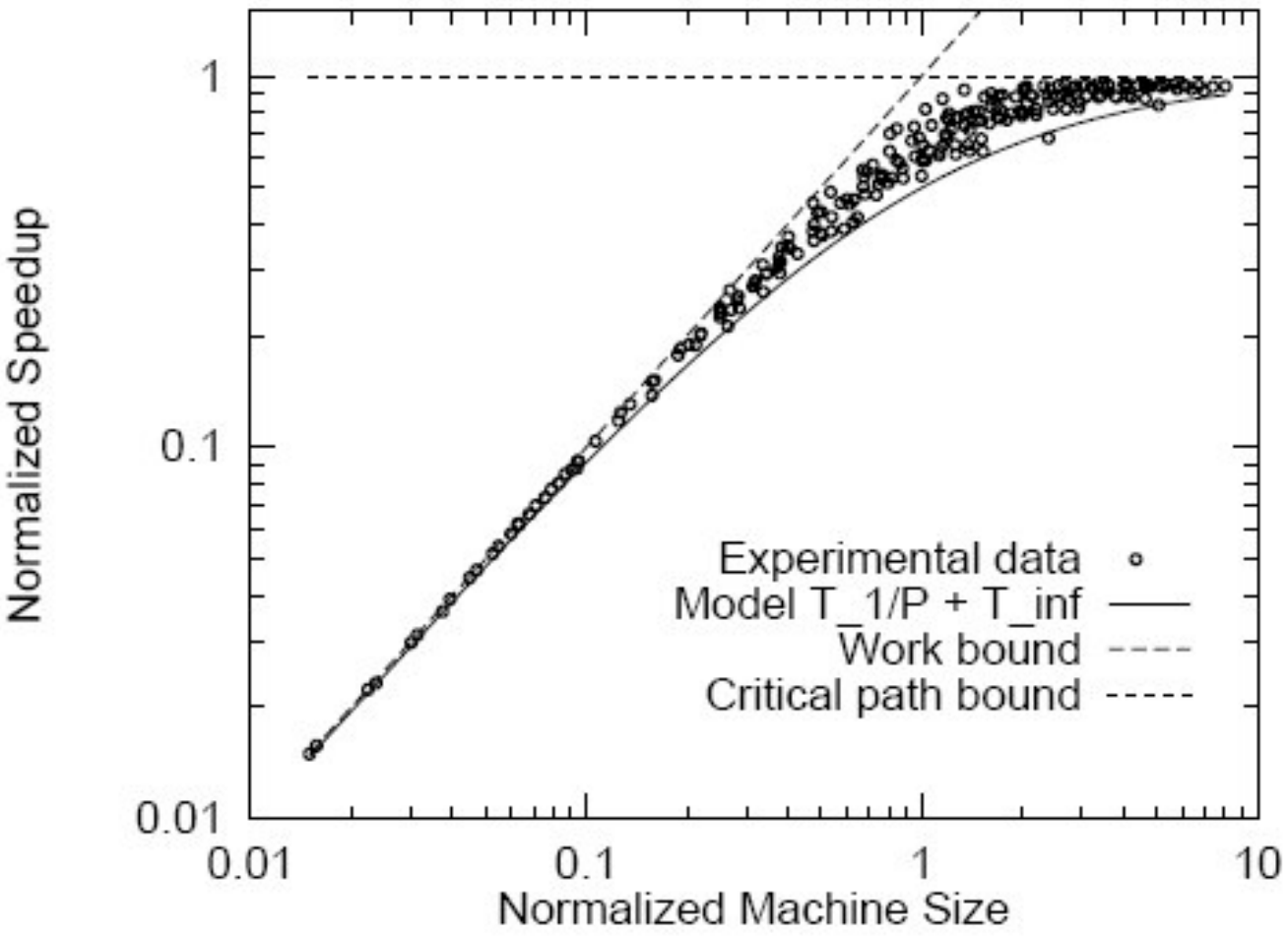
Program Stats

Program	T_1	*Work*	T_∞	$\overline{P}$	T_8	T_1/T_8
`blockedmul`	41.7	40.8	0.00114	35789	5.32	7.8
`bucket`	6.4	6.1	0.0318	192	1.02	6.3
`cholesky`	25.1	22.5	0.0709	317	3.68	6.8
`cilksort`	5.9	5.6	0.00503	1105	0.87	6.7
`fft`	13.0	12.5	0.00561	2228	1.92	6.8
`fib`	25.0	19.2	0.000120	160000	3.19	7.8
`heat`	63.3	63.2	0.191	331	8.32	7.6
`knapsack`†	112.0	104.0	0.000212	490566	14.3	7.8
`knary`	53.0	43.0	2.15	20	20.2	2.6
`lu`	23.1	23.0	0.174	132	3.09	7.5
`magic`	6.1	5.5	0.0780	71	0.848	7.2
`notempmul`	40.4	39.8	0.0142	2803	4.96	8.0
`plu`	196.1	194.1	1.753	112	30.8	6.4
`queens`†	216.0	215.0	0.00156	137821	19.4	11.0
`spacemul`	39.3	38.9	0.000747	52075	4.91	8.0
`strassen`	4.2	4.1	0.154	27	0.767	5.5
`rectmul`	5.0	5.0	0.000578	8599	0.638	7.8
`barnes-hut`	112.0	112.0	0.629	181	14.8	7.6

Overheads


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

Scalability



Cilk vs. POSIX

- Why use Cilk rather than threads?
  - "Nondeterminator" (race detector)
- Are test programs representative?

The End