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
```c
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;
}
```
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

- \( T_P = \text{time on } P \text{ processors:} \)
  - \( T_P = T_1/P + O(T_\infty) \)
  - \( T_P \leq T_1/P + c_\infty T_\infty \)

- Average parallelism (max speedup)
  - \( P_{\text{AVERAGE}} = T_1/T_\infty \)

- Parallel slackness
  - \( P_{\text{AVERAGE}}/P \)
Work-First Principle, II

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

```c
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

```c
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
data
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
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

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

- 466 MHz Alpha 21164: 27ns
- 200 MHz Pentium Pro: 78ns
- 167 MHz Ultra SPARC I: 113ns
- 195 MHz MIPS R10000: 115ns

Legend:
- THE protocol
- Frame allocation
- State saving
- C
Scalability

![Graph showing normalized speedup vs. normalized machine size with data points, a model line, and bounds for work and critical path.](image)
Cilk vs. POSIX

- Why use Cilk rather than threads?
- “Nondeterminator” (race detector)
- Are test programs representative?
The End