CS 240A: Shared Memory & Multicore Programming with Cilk++

- Multicore and NUMA architectures
- Multithreaded Programming
- Cilk++ as a concurrency platform
- Work and Span

Thanks to Charles E. Leiserson for some of these slides
Multicore Architecture

Chip Multiprocessor (CMP)
cc-NUMA Architectures

AMD 8-way Opteron Server (neumann@cs.ucsb.edu)

A processor (CMP) with 2/4 cores

Memory bank local to a processor

Point-to-point interconnect
cc-NUMA Architectures

- No Front Side Bus
- Integrated memory controller
- On–die interconnect among CMPs
- Main memory is physically distributed among CMPs (i.e. each piece of memory has an affinity to a CMP)
- NUMA: Non–uniform memory access.
  - For multi–socket servers only
  - Your desktop is safe (well, for now at least)
  - Triton nodes are not NUMA either
Desktop Multicores Today

This is your AMD Barcelona or Intel Core i7!

On-die interconnect

Private cache: Cache coherence is required
Multithreaded Programming

- POSIX Threads (Pthreads) is a set of threading interfaces developed by the IEEE
- “Assembly language” of shared memory programming
- Programmer has to manually:
  - Create and terminate threads
  - Wait for threads to complete
  - Manage interaction between threads using mutexes, condition variables, etc.
Concurrency Platforms

• Programming directly on PThreads is painful and error-prone.
• With PThreads, you either sacrifice memory usage or load-balance among processors.
• A concurrency platform provides linguistic support and handles load balancing.
• Examples:
  • Threading Building Blocks (TBB)
  • OpenMP
  • Cilk++
Cilk vs PThreads

How will the following code execute in PThreads? In Cilk?

```c
for (i=1; i<1000000000; i++) {
    spawn-or-fork foo(i);
}
sync-or-join;
```

What if `foo` contains code that waits (e.g., spins) on a variable being set by another instance of `foo`?

They have different **liveness** properties:

- Cilk threads are spawned lazily, “may” parallelism
- PThreads are spawned eagerly, “must” parallelism
Cilk vs OpenMP

- Cilk++ guarantees space bounds
  - On P processors, Cilk++ uses no more than P times the stack space of a serial execution.
- Cilk++ has a solution for global variables (called “reducers” / “hyperobjects”)
- Cilk++ has nested parallelism that works and provides guaranteed speed-up.
  - Indeed, cilk scheduler is provably optimal.
- Cilk++ has a race detector (cilkscreen) for debugging and software release.

*Keep in mind that platform comparisons are (always will be) subject to debate*
Complexity Measures

\[ T_P = \text{execution time on } P \text{ processors} \]

\[ T_1 = \text{work} \quad T_\infty = \text{span}^* \]

WORK LAW

- \[ T_P \geq T_1 / P \]

SPAN LAW

- \[ T_P \geq T_\infty \]

*Also called \textit{critical-path length} or \textit{computational depth}.*
Series Composition

**Work:** \( T_1(A \cup B) = T_1(A) + T_1(B) \)

**Span:** \( T_\infty(A \cup B) = T_\infty(A) + T_\infty(B) \)
Parallel Composition

Work: $T_1(A \cup B) = T_1(A) + T_1(B)$

Span: $T_\infty(A \cup B) = \max\{T_\infty(A), T_\infty(B)\}$
**Def.** \( T_1 / T_P = \text{speedup} \) on \( P \) processors.

If \( T_1 / T_P = \Theta(P) \), we have *linear speedup*,

\[ = P, \] we have *perfect linear speedup*,

\[ > P, \] we have *superlinear speedup*,

which is not possible in this performance model, because of the *Work Law* \( T_P \geq T_1 / P \).
Scheduling

- Cilk++ allows the programmer to express potential parallelism in an application.
- The Cilk++ scheduler maps strands onto processors dynamically at runtime.
- Since on-line schedulers are complicated, we’ll explore the ideas with an off-line scheduler.
Greedy Scheduling

**IDEA:** Do as much as possible on every step.

**Definition:** A strand is *ready* if all its *predecessors* have executed.
Greedy Scheduling

**IDEA:** Do as much as possible on every step.

**Definition:** A strand is *ready* if all its **predecessors** have executed.

**Complete step**
- $\geq P$ strands ready.
- Run any $P$. 

$P = 3$
Greedy Scheduling

**IDEA:** Do as much as possible on every step.

*Definition:* A strand is *ready* if all its *predecessors* have executed.

**Complete step**
- $\geq P$ strands ready.
- Run any $P$.

**Incomplete step**
- $< P$ strands ready.
- Run all of them.
Theorem: Any greedy scheduler achieves

\[ T_P \leq T_1/P + T_\infty. \]

Proof.

- \# complete steps \( \leq T_1/P \), since each complete step performs \( P \) work.
- \# incomplete steps \( \leq T_\infty \), since each incomplete step reduces the span of the unexecuted dag by 1. \( \blacksquare \)
Optimality of Greedy

**Corollary.** Any greedy scheduler achieves within a factor of 2 of optimal.

**Proof.** Let $T_P^*$ be the execution time produced by the optimal scheduler. Since $T_P^* \geq \max\{T_1/P, T_\infty\}$ by the Work and Span Laws, we have

$$T_P \leq T_1/P + T_\infty \leq 2 \cdot \max\{T_1/P, T_\infty\} \leq 2T_P^* .$$
**Corollary.** Any greedy scheduler achieves near-perfect linear speedup whenever $P \ll \frac{T_1}{T_\infty}$.

**Proof.** Since $P \ll \frac{T_1}{T_\infty}$ is equivalent to $T_\infty \ll \frac{T_1}{P}$, the Greedy Scheduling Theorem gives us

$$T_P \leq \frac{T_1}{P} + T_\infty \approx \frac{T_1}{P}.$$ 

Thus, the speedup is $\frac{T_1}{T_P} \approx P$. ■

**Definition.** The quantity $\frac{T_1}{PT_\infty}$ is called the *parallel slackness*. 
Each worker (processor) maintains a **work deque** of ready strands, and it manipulates the bottom of the deque like a stack.
Cilk++ Runtime System

Each worker (processor) maintains a *work deque* of ready strands, and it manipulates the bottom of the deque like a stack.
Cilk++ Runtime System

Each worker (processor) maintains a work deque of ready strands, and it manipulates the bottom of the deque like a stack.
Each worker (processor) maintains a work deque of ready strands, and it manipulates the bottom of the deque like a stack.
Each worker (processor) maintains a work deque of ready strands, and it manipulates the bottom of the deque like a stack.
Each worker (processor) maintains a work deque of ready strands, and it manipulates the bottom of the deque like a stack.

When a worker runs out of work, it steals from the top of a random victim’s deque.
Each worker (processor) maintains a **work deque** of ready strands, and it manipulates the bottom of the deque like a stack.

When a worker runs out of work, it *steals* from the top of a *random* victim’s deque.
Each worker (processor) maintains a *work deque* of ready strands, and it manipulates the bottom of the deque like a stack.

When a worker runs out of work, it *steals* from the top of a *random* victim’s deque.
Cilk++ Runtime System

Each worker (processor) maintains a work deque of ready strands, and it manipulates the bottom of the deque like a stack.

When a worker runs out of work, it steals from the top of a random victim’s deque.
Cilk++ Runtime System

Each worker (processor) maintains a work deque of ready strands, and it manipulates the bottom of the deque like a stack.

When a worker runs out of work, it steals from the top of a random victim’s deque.
Cilk++ Runtime System

Each worker (processor) maintains a work deque of ready strands, and it manipulates the bottom of the deque like a stack.

Theorem: With sufficient parallelism, workers steal infrequently ⇒ linear speed–up.
Great, how do we program it?

- Cilk++ is a faithful extension of C++
- Often use divide–and–conquer
- Three (really two) hints to the compiler:
  - `cilk_spawn`: this function can run in parallel with the caller
  - `cilk_sync`: all spawned children must return before execution can continue
  - `cilk_for`: all iterations of this loop can run in parallel
  - Compiler translates `cilk_for` into `cilk_spawn` & `cilk_sync` under the covers
Nested Parallelism

Example: Quicksort

template<typename T>
void qsort(T begin, T end) {
  if (begin != end) {
    T middle = partition(
      begin,
      end,
      bind2nd( less<typename iterator_traits<T>::value_type>(),
        *begin )
    );
    cilk_spawn qsort(begin, middle);
    qsort(max(begin + 1, middle), end);
    cilk_sync;
  }
}

The named child function may execute in parallel with the parent caller.

Control cannot pass this point until all spawned children have returned.
Cilk++ Loops

Example: Matrix transpose

cilk_for (int i=1; i<n; ++i) {
    cilk_for (int j=0; j<i; ++j) {
        B[i][j] = A[j][i];
    }
}

- A cilk_for loop’s iterations execute in parallel.
- The index must be declared in the loop initializer.
- The end condition is evaluated exactly once at the beginning of the loop.
- Loop increments should be a const value
Serial Correctness

The *serialization* is the code with the Cilk++ keywords replaced by null or C++ keywords.

Serial correctness can be debugged and verified by running the multithreaded code on a single processor.
How to seamlessly switch between serial c++ and parallel cilk++ programs?

```cpp
#ifdef CILKPAR
    #include <cilk.h>
#else
    #define cilk_for for
    #define cilk_main main
    #define cilk_spawn
    #define cilk_sync
#endif
```

- cilk++ -DCILKPAR -O2 -o parallel.exe main.cpp
- g++ -O2 -o serial.exe main.cpp

Add to the beginning of your program

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

Parallel correctness can be debugged and verified with the Cilkscreen race detector, which guarantees to find inconsistencies with the serial code.
**Race Bugs**

**Definition.** A *determinacy race* occurs when two logically parallel instructions access the same memory location and at least one of the instructions performs a write.

**Example**

```c
int x = 0;
cilk_for(int i=0, i<2, ++i) {
    x++;
}
assert(x == 2);
```

*Dependency Graph*
Race Bugs

Definition. A **determinacy race** occurs when two logically parallel instructions access the same memory location and at least one of the instructions performs a write.

```
r1 = x;
r1 ++;
x = r1;
```

```
r2 = x;
r2 ++;
x = r2;
```

```
x = 0;
assert(x == 2);
```

```
x = 0;
```

```
r1 = x;
r1 ++;
x = r1;
```

```
r2 = x;
r2 ++;
x = r2;
```

```
assert(x == 2);
```
Types of Races

Suppose that instruction $A$ and instruction $B$ both access a location $x$, and suppose that $A \parallel B$ ($A$ is parallel to $B$).

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Race Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>read</td>
<td>none</td>
</tr>
<tr>
<td>read</td>
<td>write</td>
<td>read race</td>
</tr>
<tr>
<td>write</td>
<td>read</td>
<td>read race</td>
</tr>
<tr>
<td>write</td>
<td>write</td>
<td>write race</td>
</tr>
</tbody>
</table>

Two sections of code are *independent* if they have no determinacy races between them.
Avoiding Races

- All the iterations of a `cilk_for` should be independent.
- Between a `cilk_spawn` and the corresponding `cilk_sync`, the code of the spawned child should be independent of the code of the parent, including code executed by additional spawned or called children.

Ex.

```
cilk_spawn qsort(begin, middle);
qsort(max(begin + 1, middle), end);
cilk_sync;
```

Note: The arguments to a spawned function are evaluated in the parent before the spawn occurs.
Cilk++ Reducers

- Hyperobjects: reducers, holders, splitters
- Primarily designed as a solution to global variables, but has broader application

```cpp
int result = 0;
cilk_for (size_t i = 0; i < N; ++i) {
    result += MyFunc(i);
}
```

This uses one of the predefined reducers, but you can also write your own reducer easily.
Hyperobjects under the covers

- A reducer `hyperobject<T>` includes an associative_binary operator `⊗` and an identity element.

- Cilk++ runtime system gives each thread a **private view** of the global variable

- When threads synchronize, their private views are combined with `⊗`
• Cilkscreen runs off the binary executable:
  ▪ Compile your program with \texttt{--fcilkscreen}
  ▪ Go to the directory with your executable and say \texttt{cilkscreen your_program [options]}
  ▪ Cilkscreen prints info about any races it detects

• Cilkscreen \textbf{guarantees} to report a race if there exists a parallel execution that could produce results different from the serial execution.

• It runs about $20$ times slower than single-threaded real-time.
Because the **Span Law** dictates that $T_p \geq T_\infty$, the maximum possible speedup given $T_1$ and $T_\infty$ is

$$T_1/T_\infty = \text{parallelism}$$

= the average amount of work per step along the span.
Three Tips on Parallelism

1. **Minimize span** to maximize parallelism. Try to generate 10 times more parallelism than processors for near-perfect linear speedup.

2. If you have plenty of parallelism, try to trade some of it off for *reduced work overheads*.

3. Use *divide-and-conquer recursion* or *parallel loops* rather than spawning one small thing off after another.

**Do this:**

```cilk
for (int i=0; i<n; ++i) {
    cilk_spawn foo(i);
}
cilk_sync;
```

**Not this:**

```cilk
for (int i=0; i<n; ++i) {
    cilk_spawn foo(i);
}  
cilk_sync;
```
Three Tips on Overheads

1. Make sure that work/#spawns is not too small.
   • Coarsen by using function calls and inlining near the leaves of recursion rather than spawning.

2. Parallelize outer loops if you can, not inner loops (otherwise, you’ll have high burdened parallelism, which includes runtime and scheduling overhead). If you must parallelize an inner loop, coarsen it, but not too much.
   • 500 iterations should be plenty coarse for even the most meager loop. Fewer iterations should suffice for “fatter” loops.

3. Use reducers only in sufficiently fat loops.