Multicore (and Shared Memory) Programming with Cilk++

- Multicore and NUMA architectures
- Multithreaded Programming
- Cilk++ as a concurrency platform
- Divide and conquer paradigm for Cilk++

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)

Memory bank local to a CMP

CMP with 4 cores
cc-NUMA Architectures

- No Front Side Bus
- Integrated memory controller
- On-die interconnect among CMPs
- Main memory is physically distributed among CMPs -- each piece of memory has an affinity to one CMP
- NUMA: Non-uniform memory access
  - For multi-socket servers only
  - Your laptop is safe (well, for now at least)
  - Triton nodes are also NUMA!
Desktop Multicores Today

This is your AMD Shanghai or Intel Core i7 (Nehalem)!

On-chip interconnect

Private cache: Cache coherence is required
Multithreaded Programming

- A thread of execution is a fork of a computer program into two or more concurrently running tasks.
- POSIX Threads (Pthreads) is a set of threading interfaces developed by the IEEE
- Assembly of shared memory programming
- Programmer has to manually:
  - Create and terminating threads
  - Wait for threads to complete
  - Manage the interaction between threads using mutexes, condition variables, etc.
Concurrent 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?

This difference is a liveness property:
- 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 serial semantics.
- Cilk++ has a solution for global variables (a construct called "hyperobjects")
- Cilk++ has nested parallelism that works and provides guaranteed speed-up.
- Cilk++ has a race detector for debugging and software release.
Great, how do we program in it?

- Cilk++ is a faithful extension of C++
- Programmers implement algorithms mostly in the *divide–and–conquer* paradigm. Two hints to the compiler:
  - `cilk_spawn`: *the following function can run in parallel with the caller.*
  - `cilk_sync`: *all spawned children must return before program execution can continue*
- Third keyword for programmer convenience only (compiler converts it to spawns & syncs under the covers)
  - `cilk_for`
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;
    }
}
Cilk++ Loops

Example: Matrix transpose

```cpp
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.
Serialization

How to seamlessly switch between serial c++ and parallel cilk++ programs?

```c
#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!
Parallel Correctness can be debugged and verified with the Cilkscreen race detector, which guarantees to find inconsistencies with the serial code quickly.

```c
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);
    }
}
```
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++;  // A
    x++;  // B
} 
assert(x == 2);  // C
```

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.

```
int x = 0;

r1 = x;
r1++;  // 2
x = r1;  // 7

r2 = x;
r2++;  // 5
x = r2;  // 6

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

Suppose that instruction A and instruction B both access a location x, and suppose that A∥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;
```
Cilkscreen

- Cilkscreen runs off the binary executable:
  - Compile your program with the `–fcilkscreen` option to include debugging information.
  - Go to the directory with your executable and execute `cilkscreen your_program [options]`
  - Cilkscreen prints information about any races it detects.

- For a given input, Cilkscreen mathematically guarantees to localize a race if there exists a parallel execution that could produce results different from the serial execution.

- It runs about 20 times slower than real-time.
Complexity Measures

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

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

\textbf{WORK LAW}

\[ T_P \geq T_1 / P \]

\textbf{SPAN LAW}

\[ T_P \geq T_\infty \]

*Also called \textit{critical-path length} or \textit{computational depth}.*
**Def.**  \( \frac{T_1}{T_P} = \text{speedup} \) on \( P \) processors.

If \( \frac{T_1}{T_P} = \Theta(P) \), 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* \( T_P \geq T_1/P \).
Because the Span Law dictates that $T_p \geq T_\infty$, the maximum possible speedup given $T_1$ and $T_\infty$ is

$$\frac{T_1}{T_\infty} = \textit{parallelism}$$

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

1. **Minimize the 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:**

```c
for (int i=0; i<n; ++i) {
    foo(i);
}
```

**Not this:**

```c
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. 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.
Sorting

- Sorting is possibly the most frequently executed operation in computing!
- **Quicksort** is the fastest sorting algorithm in practice with an average running time of $O(N \log N)$, *(but $O(N^2)$ worst case performance)*
- **Mergesort** has worst case performance of $O(N \log N)$ for sorting N elements
- Both based on the recursive **divide-and-conquer** paradigm
Basic Quicksort sorting an array S works as follows:

- If the number of elements in S is 0 or 1, then return.
- Pick any element \( v \) in S. Call this pivot.
- Partition the set \( S-\{v\} \) into two disjoint groups:
  - \( S_1 = \{x \in S-\{v\} \mid x \leq v\} \)
  - \( S_2 = \{x \in S-\{v\} \mid x \geq v\} \)
- Return quicksort\((S_1)\) followed by \( v \) followed by quicksort\((S_2)\)
QUICKSORT

Select Pivot
QUICKSORT

Partition around Pivot

13 45 14 56
32 21 31 78
34

13 31 21
14 32

34

45 56
78
QUICKSORT

Quicksort recursively

13 14 21 31 32

34

45 56 78

13 14 21 31 32

34

45 56 78

13 14 21 31 32

34

45 56 78