Lecture 7: Other Concurrency Models

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com
We have a problem that we want to solve in parallel, what shall we do?
Two Aspects of Parallel Programming

- Correctness: avoiding race conditions and deadlocks
- Performance: efficient use of resources
Where is Pthreads btw??
Family Tree

Languages
- Cilk
  - space efficient scheduler
  - cache-oblivious algorithms
- Threaded-C
  - continuation tasks
  - task stealing

Pragmas
- OpenMP*
  - fork/join tasks
- OpenMP taskqueue
  - while & recursion
- ECMA .NET*
  - parallel iteration classes

Libraries
- STL
  - generic programming
- JSR-166
  - (FJTask) containers
- STAPL
  - recursive ranges
- Intel® TBB

Chare Kernel
- small tasks

*Other names and brands may be claimed as the property of others

Source: Arch D. Robison slides about TBB
TIOBE Programming Community Index

Source: www.tiobe.com
CILK

http://supertech.csail.mit.edu/cilk/
http://www.cilkplus.org/
CILK

- Extends the C language with just a handful of keywords (*Cilk++ is faithful extension of C++*)
- Cilk $\rightarrow$ Cilk++ $\rightarrow$ Intel Cilk Plus
- Shared-memory multiprocessor
- CILK is processor oblivious
  - The runtime, not the programmer, decides how to schedule threads among processors.
- Cilk provides *no* new data types.
- Example applications:
  - n-body simulation
  - graphics rendering
  - Heuristic search
  - Dense and sparse matrix computation
Implications

Code like the following executes properly without any risk of blowing out memory:

```plaintext
for (i=1; i<1000000000; i++) {
    spawn foo(i);
}

sync;
```

Recursion ... Recursion ... Recursion
Fibonacci

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

**Cilk code**

```c

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);
    }
}
```

Note: in Cilk Plus:

spawn → cilk_spawn
sync → cilk_sync
Basic Cilk Keywords

```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);
  }
}
```

Identifies a function as a **Cilk procedure**, capable of being spawned in parallel.

The named **child** Cilk procedure can execute in parallel with the **parent** caller.

Control cannot pass this point until all spawned children have returned.
Dynamic Multithreading

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);
  }
}

Example: fib(4)

"Processor oblivious"

The computation dag unfolds dynamically.
Example: \( \text{fib}(4) \)

Assume for simplicity that each Cilk thread in \( \text{fib()} \) takes unit time to execute.

**Work:** \( T_1 = 17 \)

**Span:** \( T_\infty = 8 \)

**Parallelism:** \( \frac{T_1}{T_\infty} = 2.125 \)

Using many more than 2 processors makes little sense.
Parallelizing Vector Addition

C

```c
void vadd (real *A, real *B, int n){
    int i; for (i=0; i<n; i++) A[i] += B[i];
}
```
Parallelizing Vector Addition

```c
void vadd (real *A, real *B, int n){
    int i; for (i=0; i<n; i++) A[i]+=B[i];
}
```

```c
void vadd (real *A, real *B, int n){
    if (n<=BASE) {
        int i; for (i=0; i<n; i++) A[i]+=B[i];
    } else {
        vadd (A, B, n/2);
        vadd (A+n/2, B+n/2, n-n/2);
    }
}
```

Parallelization strategy:
1. Convert loops to recursion.
Parallelizing Vector Addition

**C**

```c
void vadd (real *A, real *B, int n) {
    int i; for (i=0; i<n; i++) A[i] += B[i];
}
```

**CILK**

```c
void vadd (real *A, real *B, int n) {
    if (n <= BASE) {
        int i; for (i=0; i<n; i++) A[i] += B[i];
    } else {
        vadd (A, B, n/2;
        vadd (A+n/2, B+n/2, n-n/2;
    }
}
```

**Parallelization strategy:**
1. Convert loops to recursion.
2. Insert Cilk keywords.
Vector Addition Analysis

To add two vectors of length $n$, where $\text{BASE} = \Theta(1)$:

**Work:** $T_1 = ? \quad \Theta(n)$

**Span:** $T_\infty = ? \quad \Theta(\lg n)$

**Parallelism:** $T_1/T_\infty = ? \quad \Theta(n/\lg n)$
Another Parallelization

C

```c
void vadd1 (real *A, real *B, int n){
    int i; for (i=0; i<n; i++) A[i]+=B[i];
}
void vadd (real *A, real *B, int n){
    int j; for (j=0; j<n; j+=BASE) {
        vadd1(A+j, B+j, min(BASE, n-j));
    }
}
```

Cilk

```c
Cilk void vadd1 (real *A, real *B, int n){
    int i; for (i=0; i<n; i++) A[i]+=B[i];
}
Cilk void vadd (real *A, real *B, int n){
    int j; for (j=0; j<n; j+=BASE) {
        spawn vadd1(A+j, B+j, min(BASE, n-j));
    }
sync;
}
```
To add two vectors of length $n$, where $\text{BASE} = \Theta(1)$:

**Work:** $T_1 = ? \Theta(n)$

**Span:** $T_\infty = ? \Theta(n)$

**Parallelism:** $T_1/T_\infty \Theta(1)$ !!!
Definition. A **determinacy race** occurs when two logically parallel instructions access the same location of memory and one of the instructions performs a write.
Race Condition Revisited

- Famous race bugs:
  - The Therac-25 radiation therapy machine, which killed three people and injured several others.
  - The North American Blackout of 2003, which left over 50 million people without power.
- **Cilkscreen race detection tool**

<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>

Instruction A and instruction B both access a location \( x \), and \( A \parallel B \).
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);
    }
}

Cilk++ source

Cilk++ Compiler

Conventional Compiler

Linker

Binary

Cilkscreen Race Detector

Parallel Regression Tests

Reliable Multi-Threaded Code
Cilk’s Work-Stealing Scheduler

Each processor maintains a work deque of ready threads, and it manipulates the bottom of the deque like a stack.
Cilk’s Work-Stealing Scheduler

Each processor maintains a work deque of ready threads, and it manipulates the bottom of the deque like a stack.
Cilk's Work-Stealing Scheduler

Each processor maintains a *work deque* of ready threads, and it manipulates the bottom of the deque like a stack.

Return!
Cilk’s Work-Stealing Scheduler
Each processor maintains a **work deque** of ready threads, and it manipulates the bottom of the deque like a stack.
Cilk’s Work-Stealing Scheduler

Each processor maintains a work deque of ready threads, and it manipulates the bottom of the deque like a stack.

When a processor runs out of work, it steals a thread from the top of a random victim’s deque.
Cilk’s Work-Stealing Scheduler

Each processor maintains a **work deque** of ready threads, and it manipulates the bottom of the deque like a stack.

When a processor runs out of work, it **steals** a thread from the top of a **random** victim’s deque.
Cilk’s Work-Stealing Scheduler

Each processor maintains a work deque of ready threads, and it manipulates the bottom of the deque like a stack.

When a processor runs out of work, it steals a thread from the top of a random victim’s deque.
Cilk’s Work-Stealing Scheduler

Each processor maintains a work deque of ready threads, and it manipulates the bottom of the deque like a stack.

When a processor runs out of work, it steals a thread from the top of a random victim’s deque.
for (i=1; i<1000000000; i++) {
    spawn foo(i);
}
sync;

How will this code perform in Cilk vs Pthreads?
Cilk++ vs OpenMP

- Cilk++ uses no more than $P$ times the stack space of a serial execution. \([p = \#\text{processors}]\)
- Cilk++ has nested parallelism that works and provides guaranteed speed-up.
- Cilk++ has a race detector for debugging and software release.
- There is a \texttt{cilk\_for} but for programmer convenience only. The compiler converts it to spawns/syncs under the covers.
- Cilk way of thinking depends on recursion (divide on conquer).
Tips on Parallelism With Cilk

1. 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
#include <cilk.h>

void foo(int i) {
    // Do some work here...
}

cilk_for (int i=0; i<n; ++i) {
    foo(i);
}
cilk_sync;
```

Not this:
```cilk
cilk_for (int i=0; i<n; ++i) {
    cilk_spawn foo(i);
}
cilk_sync;
```
```c
#include <stdio.h>
#include <stdlib.h>

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

int main(int argc, char *argv[]) {
  int n = atoi(argv[1]);
  int result = fib(n);
  printf("Fibonacci of %d is %d.\n", n, result);
  return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>

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

int main(int argc, char *argv[]) {
  int n = atoi(argv[1]);
  int result = fib(n);
  printf("Fibonacci of %d is %d.\n", n, result);
  return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>

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

int main(int argc, char *argv[]) {
  int n = atoi(argv[1]);
  int result = fib(n);
  printf("Fibonacci of %d is %d.\n", n, result);
  return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>

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

int main(int argc, char *argv[]) {
  int n = atoi(argv[1]);
  int result = fib(n);
  printf("Fibonacci of %d is %d.\n", n, result);
  return 0;
}
```
Multicore Performance Improvement Realized on a Collision Detection Algorithm

Speed-up of Quicksort Algorithm
(Each core: x86 1.7 GHz 2GB RAM)
Question: How Will you Parallelize This?

\[
\begin{align*}
X[0] &= 0; \\
Y[0] &= 1; \\
\text{for } (k = 1; k < 100; k++) & \quad X[k] = Y[k-1] + 1; \\
& \quad Y[k] = X[k-1] + 2;
\end{align*}
\]
Haskell

http://www.haskell.org
Example of Programming Languages

Functional languages
- Haskell
- ML
- Scheme
- Erlang

Imperative languages
- C
- C++
- Fortran
- Assembler
- Visual C++ (2)
- Java
- Visual Basic
- Visual Basic
What is Functional Programming?

• Functional programming is **style of programming** in which the basic method of computation is **the application of functions to arguments**

```plaintext
total = 0;
for (i = 1; i ≤ 10; ++i)
    total = total+i;
```

• Summing up number from 1 to 10
• Imperative programming
• Mainly variable assignment

```plaintext
sum [1..10]
```

• Summing up number from 1 to 10
• Functional programming
What does this have to do with multicore programming?

• One of the hardest things in parallel programming is locking

• Functional programming does not use mutable data $\rightarrow$ data is separate from the logic $\rightarrow$ no side effects $\rightarrow$ order does not matter $\rightarrow$ parallelism is easier to find

• We will look at one excellent example of functional programming: HASKELL
Haskell Is:

• Memory managed
• The #1 on Language Shootout for threading
• Pure functional programming
• Simplest Haskell program:
  
  ```haskell
  main = return ()
  ```
Values and Expressions

• A value is a piece of data.
  
  2, 4, 3.14159, "John",

• An expression computes a value.
  
  2 + 2, 2*π*r

• Expressions combine values using functions and operators.
The solution of a quadratic equation:

\[
\frac{-b + \sqrt{b^2 - 4ac}}{2a}
\]
A definition gives a name to a value.

area :: Int
area = 41*37

Names start with a small letter, and are made up of letters and digits.

Types specify what kind of value this is.

An expression says how the value is computed.
In Haskell, \( f :: A \rightarrow B \) means for every \( x \in A \),

\[
f(x) = \begin{cases} 
\text{some element } y = f(x) \in B \\
\text{run forever}
\end{cases}
\]

In words, “if \( f(x) \) terminates, then \( f(x) \in B \).”
A function definition specifies how the result is computed from the arguments.

```
area :: Int -> Int -> Int
area l b = l*b
```

Function types specify the types of the arguments and the result. The body specifies how the result is computed. The arguments are given names, after the function name.
Function Notation

*Function arguments need not be enclosed in brackets!*

Example:  
\[
\text{average} :: \text{Float} \rightarrow \text{Float} \rightarrow \text{Float}
\]

\[
\text{average } x \ y = \frac{x + y}{2}
\]

Calls:  
\[
\text{average 2 3} \rightarrow 2.5
\]

\[
\text{average } (2+2) (3*3) \rightarrow 6.5
\]

Brackets are for grouping only!
Functional Programming

• A functional program consists mostly of function definitions.

• Simple functions are used to define more complex ones, which are used to define still more complex ones, and so on.

• Finally, we define a function to compute the output of the entire program from its inputs.
Compiling Your Haskell Program

• to get the interactive system:
  $ ghci

• To generate an executable:
  $ ghc -o hello hello.hs

• Check:
  http://www.haskell.org/haskellwiki/Haskell_in_5_steps
Conclusion

• Keep three aspects in mind when writing parallel programs for multicore processors:
  – Correctness
  – Scalability
  – Development time