Lecture 8: Other Concurrency Platforms

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com
Two Aspects of Parallel Programming

Correctness
- avoiding race conditions and deadlocks

Performance
- efficient use of resources
Concurrency Platforms

Libraries
- Thread-pool Libraries
  - .NET Thread pool class
- Message Passing Libraries
  - MPI
- Task-parallel Libraries
  - Intel TBB

Data-Parallel Programming Languages
- RapidMind
- NESL

Parallel Language Extensions
- OpenMP
- Cilk/ Cilk++

Where is Pthreads btw??
Family Tree

Languages
- Cilk
- OpenMP
- Threading-C
- Threaded-C
- Cilk task stealing

Pragmas
- OpenMP
- OpenMP taskqueue
- OpenMP task stealing

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

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

Source: Arch D. Robison slides about TBB
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
• 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

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

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);
    }
}
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.
Parallelizing Vector Addition

```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;
        sync;
    }
}
```

Parallelization strategy:
1. Convert loops to recursion.
2. Insert Cilk keywords.
void incr (int *counter) {
    *counter++;
}
void main() {
    int x(0);
    cilk spawn incr (&x);
    incr (&x);
    cilk sync;
    assert (x == 2);
}

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 $A \parallel B$. 
```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);
    }
}
```
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.

Spawn!
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.

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.
How will this code perform in Cilk vs Pthreads?

```c
for (i=1; i<1000000000; i++) {
    spawn foo(i);
}
sync;
```
Cilk++ vs OpenMP

- Cilk++ uses no more than $P$ times the stack space of a serial execution.
- Cilk++ has nested parallelism that works and provides guaranteed speed-up.
- Cilk++ has a race detector for debugging and software release.
- There is a `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:**

```c
void foo(int i) {
    cilk_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;
```
```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>
#include <pthread.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 input;
    int output;
    thread_args:

    void *thread_func ( void *ptr )
    {
        int i = ((thread_args *) ptr)->input;
        ((thread_args *) ptr)->output = fib(i);
        return NULL;
    }

    int main(int argc, char *argv[])
    {
        thread_t thread;
        thread_args args;
        int status;
        int result;
        int thread_result;
        if (argc < 2) return 1;
        int n = atoi(argv[1]);
        if (n < 30) result = fib(n);
        else {
            args.input = n-1;
            status = pthread_create(&thread,
                NULL,
                thread_func,
                (void*) &args);

            // main can continue executing while the thread executes.
            result = fib(n-2);
            // Wait for the thread to terminate.
            pthread_join(thread, NULL);
            result += args.output;
        }
        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)
Intel Threading Building Blocks (TBB)

• Open source C++ template library
• Task-based multithreaded applications
• TBB is strictly a library and provides no linguistic support by design.
• The library schedules tasks onto threads and manages load balancing.
• The programmer breaks an application into multiple tasks, which are scheduled using a “work-stealing” scheduler.
```cpp
#include "tbb_blocked_range.h"

class SqChunk {
    float *const local_a;

public:
    void operator()(const blocked_range<size_t>& x) const {
        float *a = local_a;
        for(size_t i=x.begin(); i!=x.end(); ++i)
            a[i] *= a[i];
    }

    SqChunk(float a[]) :
        local_a(a)
    {
    }
};

void Square(float a[], size_t n) {
    parallel_for(blocked_range<size_t>(0,n,1000), SqChunk(a));
}
```
Question: How Will you Parallelize This?

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

• **Classical scenario:**
  (1) Two threads share two locks.
  (2) The two threads take the locks in different order.

• **How do we deal with that?**

```plaintext
thread1 {
  lock(A) ;
  lock(B) ;
  unlock(B) ;
  unlock(A) ;
}

thread2 {
  lock(B) ;
  lock(A) ;
  unlock(A) ;
  unlock(B) ;
}
```

Boom!
Deadlock Detection

• In classical deadlock detection algorithm, it constructs the lock graph of an execution and if the graph has any cycle, raise alarm to notify the deadlock.

• The classical deadlock detection algorithm can not report unless the deadlock occurs.

• To detect potential deadlocks,
  (1) record the locking pattern for each thread during runtime.
  (2) at the program termination analyze the recorded locking patterns to check potential deadlock.
Lock Pattern of a Thread

thread1() {
    lock(L1) ;
    while(cond) {
        lock(L2) ;
        unlock(L2) ;
        lock(L3) ;
        unlock(L3) ;
    }
    unlock(L1) ;
    lock(L3) ;
    lock(L2) ;
    unlock(L2) ;
    unlock(L3) ;
}

execution:

lock(L1)
lock(L2)
unlock(L2)
lock(L3)
unlock(L3)
lock(L2)
unlock(L2)
unlock(L1)
lock(L3)
lock(L2)
unlock(L2)
unlock(L3)
Analyzing Locking Pattern

- **Potential Deadlock Analysis**
  - compares the trees for each pair of threads
  - **nesting(n)**: a set of locks in a path from the root node of n to n.
  - Basic Algorithm
    - for each pair \((t_1, t_2)\) of trees,
      - For all \(n_1\) in \(t_1\) and \(n_2\) in \(t_2\), checks that \(n_1\) in nesting\((n_2)\).
        - if \(n_1\) in nesting\((n_2)\), reports two locks and two threads.
    - In order to avoid issuing warnings when a gate lock prevents a deadlock,
      - checks whether \(\{\text{nesting}(n_1) \setminus \text{nesting}(n_2)\} \setminus \{n_1\text{.lock} , n_2\text{.lock}\}\) is empty or not.
        - if it is empty, there is no gate lock so that reports the potential deadlock.
Limitations of This Method

- Deadlocks will only be found if they involve two threads.
- Works only with binary lock.
- Do not consider start, join synchronization.
Generalized Algorithm

- **Lock graph**
  - constructs a lock tree for each thread during execution.
  - at the end of execution (or at the user command), it constructs a directed graph \( G = (V, E) \) where
    - \( V \) contains all the nodes of all the lock trees
    - \( E \) contains
      - (1) tree edges (from parent to child)
      - (2) inter edges
        - bidirectional edges between nodes that are labeled with the same locks and that are in different lock trees.
For a lock graph $G$, a valid path is a path that does not contain consecutive inter edges and nodes from each lock tree appear as at most one consecutive subsequence in the path.

A valid cycle is a cycle that does not contain consecutive inter edges and nodes from each thread appear as at most one consecutive subsequence in the cycle.
Steps of the Generalized Algorithm

• Potential Deadlock Detection

(1) Constructs the lock graph from an execution.
(2) Traverses all valid paths in the graph to find valid cycle.
(3) To eliminate the false alarms by gate locks, checks for every valid cycle whether there is a gate lock (i.e. whether no two nodes in different lock trees have ancestors labeled with the same lock).
Conclusions

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

• Start from what you know then move on!