CSCI-GA.3033-009
Multicore Processors:
Architecture & Programming

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
Concurrency Platforms

- Libraries
  - Thread-pool Libraries
    - .NET Thread pool class
  - Message Passing Libraries
    - MPI
  - Task-parallel Libraries
    - Intel TBB
- Data-Parallel Programming Languages
  - RapidMind → Intel Ct
  - NESL
- Functional
  - Haskell
  - Erlang
- Parallel Language Extensions
  - OpenMP
  - Cilk/ Cilk++

Where is Pthreads btw??
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 → Cilk++ → 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
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 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);
  }
}
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);
    }
}

Dynamic Multithreading

Example: fib(4)

“Processor oblivious”

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

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

\textbf{Work}: \( T_1 = 17 \)

\textbf{Span}: \( T_\infty = 8 \)

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

Using many more than 2 processors makes little sense.
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){
    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/2;
    }
    sync;
}
```

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

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 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:** $T_1/T_\infty \Theta(1)$ !!!
```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.
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
for (int i=0; i<n; ++i) {
    cilk_spawn foo(i);
}
```

**Not this:**
```cilk
for (int i=0; i<n; ++i) {
    cilk_spawn foo(i);
}
cilk_sync;
```
#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;
}

#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[])
    {
        pthread_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)

- Serial Code
- Cilk++
Question: How Will you Parallelize This?

\[
X[0] = 0; \\
Y[0] = 1; \\
\]

\[
\text{for} \ (k = 1; \ k < 100; \ k++) \\
\quad \{ \\
\qquad X[k] = Y[k-1] + 1; \\
\qquad Y[k] = X[k-1] + 2; \\
\quad \} \\
\]
Haskell

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

Functional languages
- Haskell
- ML
- Scheme
- Erlang

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

- Functional programming is a style of programming in which the basic method of computation is the application of functions to arguments.

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

- Summing up number from 1 to 10
- Imperative programming
- Mainly variable assignment
- 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 → data is separate from the logic → no side effects → order does not matter → 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*pi*r

• Expressions combine values using *functions* and *operators*. 
Functions

The solution of a quadratic equation:

\[ \frac{-b + \sqrt{b^2 - 4ac}}{2a} \]
Definitions and Types

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.
Function Types in Haskell

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.

Function types specify the type of the arguments and the result.

The body specifies how the result is computed.

The arguments are given names, after the function name.

area :: Int -> Int -> Int

area l b = l*b
Function Notation

*Function arguments need **not** be enclosed in brackets!*

Example:

average :: Float -> Float -> Float

average \( x \ y = \frac{x + y}{2} \)

Calls:

average 2 3 \rightarrow 2.5

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](http://www.haskell.org/haskellwiki/Haskell_in_5_steps)
Transactional Memories
What Are We Talking About?

• Incorporating transactions in parallel programming \(\rightarrow\) computations wrapped in transactions

• A alternative way to coordinate concurrent threads

• Characteristics of a transaction: **ACI**
  – Atomicity
  – Consistency
  – Isolation
Databases!!

- Database systems have successfully exploited parallel hardware for decades.
- Databases achieve good performance by executing many queries simultaneously and by running queries on multiple processors when possible.
- The author of an individual query need not worry about this parallelism!
Databases!!

- DB programming model $\rightarrow$ transactions
- Computation executes as if it was the only computation accessing the DB.
- Results indistinguishable from the situation in which the transactions run one after the other $\rightarrow$ serializability
- Transactions allow concurrent operations to access a common DB and still produce predictable, reproducible results.
Why Don’t we Learn from DB?

Transactional Memory

In multicore, the main data storage during execution is typically the memory.
A transaction is a sequence of actions that appears indivisible and instantaneous to an outside observer.

**Failure Atomicity**
All constituent actions in a transaction complete successfully, or none of these actions appear to start executing.

**Consistency**
Application dependent

**Isolation**
Transactions do not interfere with each other while they are running, regardless of whether or not they are executing in parallel.

**Durability**
Once a transaction commits, its result is permanent.
Example

Thread 1

\[
\begin{align*}
\text{begin}_\text{xaction} \\
A &= A - 20 \\
B &= B + 20 \\
A &= A - B \\
C &= C + 20 \\
\text{end}_\text{xaction}
\end{align*}
\]

THREAD 1’S ACCESSES AND UPDATES TO A, B, C ARE ATOMIC

Thread 2

\[
\begin{align*}
\text{begin}_\text{xaction} \\
C &= C - 30 \\
A &= A + 30 \\
\text{end}_\text{xaction}
\end{align*}
\]

THREAD 2 SEES EITHER “ALL” OR “NONE” OF THREAD 1’S UPDATES
Another Example

What values does T2 see?

```c
int x = 0; int y = 0;

T1

atomic {
  x = 42;
  y = 42;
}

T2

atomic {
  int tmp1 = x;
  int tmp2 = y;
}
```
Yet Another Example

- Transactions **appear** to execute in commit order
  - Flow (RAW) dependency cause transaction violation and restart

```
Transaction A

```
```
ld 0xdddd
...
st 0xbeef

```

```
Transaction B

```
```
ld 0xdddd
...
ld 0xbbbb

```

```
Transaction C

```
```
ld 0xbeef

```

Violation!

Re-execute with new data
Who Uses TM?

• Programmer

• Compiler designer to implement some high-level language features

• **Important**: Beside using TM for parallel programming, it can also be used in:
  – error recovery
  – real-time programming
  – multitasking
Important: Different implementations of TM may have different names for functions.
void PushLeft(DQueue *q, int val) {
    QNode *qn = malloc(sizeof(QNode));
    qn->val = val;
    QNode *leftSentinel = q->left;
    QNode *oldLeftNode = leftSentinel->right;
    qn->left = leftSentinel;
    qn->right = oldLeftNode;
    leftSentinel->right = qn;
    oldLeftNode->left = qn;
}

void PushLeft(DQueue *q, int val) {
    QNode *qn = malloc(sizeof(QNode));
    qn->val = val;
    do {
        StartTx();
        QNode *leftSentinel = ReadTx(&q->left);
        QNode *oldLeftNode = ReadTx((leftSentinel->right));
        WriteTx(&(qn->left), leftSentinel);
        WriteTx(&(qn->right), oldLeftNode);
        WriteTx((leftSentinel->right), qn);
        WriteTx((oldLeftNode->left), qn);
    } while (!CommitTx());
}
Concurrency Control: Conflict-Detection-Resolution

- **Conflict** occurs when two transactions perform conflicting operations on the same piece (2 writes, or a read and a write).
- The conflict is **detected** when the underlying TM system determines that the conflict has occurred.
- The conflict is **resolved** when the underlying system or code in a transaction takes some action to avoid the conflict—e.g., by delaying or aborting one of the conflicting transactions.
Some TM implementations use Pessimistic control, others use Optimistic control.
<table>
<thead>
<tr>
<th>Read</th>
<th>Optimistic</th>
<th>Pessimistic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Optimistic</td>
<td>TCC, TL2, SigTM</td>
<td>Intel C++ STM, Intel Java STM, HASTM, Microsoft OSTM</td>
</tr>
<tr>
<td>Pessimistic</td>
<td>LogTM</td>
<td>Intel C++ STM</td>
</tr>
</tbody>
</table>

Example of Available TMs
Version Management

• What to do about writes before a transaction commits?
  
• Eager version management
  – The transaction directly modifies the data in memory
  – Keeps an undo-log holding overwritten data
  – Requires pessimistic concurrency control

• Lazy version management
  – Updates are delayed until a transaction commits
  – Transaction maintains its tentative writes in a transaction-private redo-log
Conflict Detection

• With pessimistic approach it is easy \(\rightarrow\) locks!

• With optimistic approach, there are several issues:
  – Granularity of conflict (cache line, objects, ...)
  – The time at which detection occurs:
    • When transaction declares its intend to access the data \(\rightarrow\) eager conflict detection
    • On validation: can occur several times during transaction lifetime
    • On commit \(\rightarrow\) lazy conflict detection
  – Which kind of access is treated as conflicts?
    • Among concurrent transactions
    • Between active and committed transactions
What Can Go Wrong Here?

Assume X is initially 0

```c
// Thread 1
do {
    StartTx();
    WriteTx(&x, 1);
} while (!CommitTx());

// Thread 2
do {
    StartTx();
    int tmp_1 = ReadTx(&x);
    while (tmp_1 == 0) {
    }
} while (!CommitTx());
```
Can We Make Things Simpler?

- Things look very verbose hence error prone
- Instead of WriteTx and ReadTx can we have something simpler?
- Atomic block of statements
  - getting rid of WriteTx and ReadTx
  - Implemented for many languages
A key advantage of atomic blocks over lock-based critical sections is that the atomic block does not need to name the shared resources that it intends to access or synchronize with.
Example of TM with C/C++

- Can be called transactionally
- The above is Windows version.
- The Linux version:  \texttt{\_\_attribute\_\_tm\_callable} double \texttt{foo()};

\begin{verbatim}
__declspec(tm_callable)
double foo();

__declspec(tm_callable)
double (bar);
\end{verbatim}

\begin{verbatim}
T1

__tm_atomic {
...
    t1 = foo();
...
}

T2

__tm_atomic {
    t2 = bar();
    ...
}
\end{verbatim}
How to Provide the *Illusion* of Transactions?

- **Software Transactional Memory (STM)**
- **Hardware Transactional Memory (HTM)**

Diagram:
- Program
- Transactional Memory
- Hardware
Software Transactional Memory (STM)

Components:

- **transaction descriptor**: is the per-transaction data structure that keeps track of the state of the transaction
- **Undo-log or Redo-log**
- **read-set or write-set**: tracks the memory locations that the transaction has read from or written to
STM

- Compiler instruments code with transaction prolog, epilog, and read/write function.
- Runtime tracks memory accesses, detects conflicts, and commits/aborts execution.
STM vs OpenMP vs Pthreads

Problem to Parallelize:

Algorithm 1 Conjugate Gradients

1: $r_0 = b - Ax_0, \ p_0 = r_0, \ A \text{ spd}$
2: for $i = 0, 1, 2, \ldots$ do
3: \quad $\alpha_i = \frac{r_i^T r_i}{p_i^T A p_i}$
4: \quad $x_{i+1} = x_i + \alpha_i p_i$
5: \quad $r_{i+1} = r_i - \alpha_i A p_i$
6: \quad $\beta_i = \frac{r_{i+1}^T r_{i+1}}{r_i^T r_i}$
7: \quad $p_{i+1} = r_{i+1} + \beta_i p_i$
8: end for
STM vs OpenMP vs Pthreads
Hardware Transactional Memory (HTM)

• Three flavors
  – Full implementation of TM in hardware
  – Allowing hardware transactions to coexist with software transactions
  – Hardware extension to provide speed-up to parts of software TM
HTM

• HTM must perform the following functions
  – identify memory locations for transactional accesses
  – manage the read-sets and write-sets of the transactions
  – detect and resolve data conflicts,
  – manage architectural register state
  – commit or abort transactions
# Requirements for Supporting Transactions

<table>
<thead>
<tr>
<th>Buffering</th>
<th>Transactional cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conflict detection</td>
<td>Cache coherence protocol</td>
</tr>
<tr>
<td>Abort/Recovery</td>
<td>Invalidate transactional cache line</td>
</tr>
<tr>
<td>Commit</td>
<td>Validate transactional cache line</td>
</tr>
</tbody>
</table>
HTM

• Extensions to the instruction set
• Tracking read-sets and buffering write-sets is done using caches and buffers
• Coherence messages trigger conflict detection
• Nearly all conventional HTM proposals perform eager conflict detection
Hardware Support for Performance

1. Record recovery state
2. Buffer updates/track accesses
3. Commit if no external access (discard all updates if conflict)

Architectural Memory state
- A.sum = 80
- B.sum = 220

Core 1
begin_xaction
A.withdraw(20)
B.deposit(20)
end_xaction

Core 2
begin_xaction
Sum = A.sum + B.sum
end_xaction

Source: Konrad Lai (Intel) slides “Transactional Memories”
IBM Blue Gene/Q

The BlueGene/Q processors that powers the 20 petaflops Sequoia supercomputer built by IBM for Lawrence Livermore National Labs is the first commercial processors to include hardware support for transactional memory.
IBM Blue Gene/Q

- multicore 64-bit PowerPC-based system-on-chip
- based on IBM’s 4-way multithreaded PowerPC A2 design
- 1.47 billion transistor chip
- TM will appear in 32MB level 2 cache
- 18 cores
  - 16 for running actual computations
  - 1 for the operating system
  - 1 to improve chip reliability
STM vs HTM

• Software is more flexible than hardware and permits the implementation of a wider variety of more sophisticated algorithms.
• Software is easier to modify and evolve than hardware.
• STMs can integrate more easily with existing systems and language features, such as garbage collection.
• STMs have fewer intrinsic limitations imposed by fixed-size hardware structures, such as caches.
HTM vs STM

- HTM systems can typically execute applications with lower overheads than STM systems.
- Less reliant than STM systems on compiler optimizations to achieve performance.
- HTM systems can have better power and energy profiles.
- Treat all memory accesses within a transaction as implicitly transactional.
- HTM systems can provide strong isolation without requiring changes to non-transactional memory accesses.
- HTM systems are well suited for systems languages such as C/C++ that operate without dynamic compilation, garbage collection, and so on.
Conclusions

• TM appears in recent years as strong candidate for parallel programming

• There are many different implementations and this complicates portability a bit.

• TM is still work-in-progress, details vary between languages and between processors so care is needed when implementing an algorithm in a particular setting.
Conclusions

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