Lecture 26:
Parallelism in the Real World, Design of 15-210, and Concurrency

Lecturer: Umut A. Acar
Why Parallelism?

- Desire to solve larger more important problems.
  - One machine/processor insufficient
  - Multiple machines needed.
  - Possible because problems often decompose.

- Examples:
  - Scientific computing: massive computations
  - Internet: massive data, many queries
  - Big data
Cray-1 (1976): the world’s most expensive love seat
Data Center: Hundred's of thousands of computers
Since 2005: Multicore computers

AMD Opteron (sixteen-core) Model 6274
by AMD

List Price: $693.00
Price: **$599.99**
You Save: $93.01 (13%)

Only 1 left in stock (more on the way).
Ships from and sold by Amazon.com. Gift-wrap available.

Want it delivered Monday, November 5? Order it in the next 14 hours and 37 minutes
Delivery may be impacted by Hurricane Sandy. Proceed to checkout to see estimated

43 new from $599.99
Moore’s Law

Microprocessor Transistor Counts 1971-2011 & Moore’s Law

curve shows transistor count doubling every two years
Moore’s Law and Performance

Intel CPU Trends
(sources: Intel, Wikipedia, K. Olukotun)
Parallelism is here... And Growing!

Number of Cores

2006  2007  2008  2009  2010  ...

Core 2 Duo (2)
Core 2 Quad (4)
Dunnington (6)
Nehalem: 8+
Lorabbee: 12-32
Future: 100+

Parallelism for the Masses
"Opportunities and Challenges"

Andrew Chien, 2008
64 core blade servers ($6K) (shared memory)

**AMD Opteron (sixteen-core) Model 6274**

by **AMD**

⭐⭐⭐⭐⭐ (1 customer review)

List Price: **$933.00**

Price: **$599.99** Prime

You Save: $33.01 (13%)

**Only 1 left in stock (more on the way).**

Ships from and sold by Amazon.com. Gift-wrap available.

Want it delivered Monday, November 5? Order it in the next 14 hours and 37 minutes. Delivery may be impacted by Hurricane Sandy. Proceed to checkout to see estimated

43 new from $599.99
1024 “cuda” cores

EVGA GeForce GTX 590 Classified 3DVI/Mini-Display Port SLI Ready Li 03G-P3-1596-AR
by EVGA

⭐⭐⭐⭐⭐ (16 customer reviews) | ⬇️ Like (29)

Price: $924.56

In Stock.
Ships from and sold by J-Electronics.

Only 1 left in stock--order soon.

5 new from $749.99   2 used from $695.00
Circa November 2012
Samsung Galaxy S IV is now Official: Octa-Core CPU, 5” Full HD Display & 13MP Camera

Samsung has just announced the Samsung Galaxy S4 at their Mobile Unpacked Event 2013 Episode 1 in New York, USA. The Galaxy S4 features a stunning 4.99” Full HD (1920×1080) SuperAMOED display. With a 441 ppi pixel density, your eyes won’t be able to distinguish the pixels, which ensures excellent visual comfort. Even though the Galaxy S4 has a large display and a massive battery of 2,600 MAh, it’s only 7.9mm thick. Samsung’s latest flagship device is PACKED with powerful components, consisting of Samsung’s latest Exynos 5 Octa-Core (5410) CPU based on ARM’s big.LITTLE technology with Quad Cortex-
Intel Has a 48-Core Chip for Smartphones and Tablets

By Wolfgang Gruener OCTOBER 31, 2012 9:20 AM - Source: Computerworld

Intel has developed a prototype of a 48-core processor for smartphones. Before you ask: No, you can't buy a 48-core smartphone next year.
Parallel Hardware

• Many forms of parallelism
  – Supercomputers: large scale, shared memory
  – Clusters and data centers: large-scale, distributed memory
  – Multicores: tightly coupled, smaller scale

• Parallelism is important in the real world.
Key Challenge: Software (How to Write Parallel Code?)

• At a high-level, it is a two step process:
  – Design a work-efficient, low-span parallel algorithm
  – Implement it on the target hardware

• In reality: each system required different code because programming systems are immature
  – Huge effort to generate efficient parallel code.
    • Example: Quicksort in MPI is 1700 lines of code, and about the same in CUDA
Teaching Challenge: How to Teach Parallelism

- Education must prepare for the “real world” and research.
- Essential to teach parallel algorithm design.
- But many different models of parallelism.
- Inadequate parallel systems make practical work gruesome. Practically impractical in the context of an undergraduate course.
Enable parallel thinking by raising abstraction level

I. **Parallel thinking:** Applicable to many machine models and programming languages

II. **Reason** about correctness and efficiency of algorithms and data structures.
Parallel Thinking

• Recognizing true dependences: unteach sequential programming.

• Parallel algorithm-design techniques
  – Operations on aggregates: map/reduce/scan
  – Divide & conquer, contraction
  – Viewing computation as DAG

• Cost model based on work and span
Quicksort from Aho-Hopcroft-Ullman (1974)

procedure QUICKSORT(S):
  if S contains at most one element then return S
  else
    begin
      choose an element a randomly from S;
      let $S_1$, $S_2$ and $S_3$ be the sequences of elements in $S$ less than, equal to, and greater than a, respectively;
      return (QUICKSORT($S_1$) followed by $S_2$ followed by QUICKSORT($S_3$))
    end
public void quickSort(int[] a, int left, int right) {
    int i = left-1;  int j = right;
    if (right <= left) return;
    while (true) {
        while (a[++i] < a[right]);
        while (a[right] < a[--j])
            if (j==left) break;
        if (i>= j) break;
        swap(a,i,j); }
    swap(a, i, right);
    quickSort(a, left, i - 1);
    quickSort(a, i+1, right); }

Quicksort from Sedgewick (2003)
Programming Parallel Algorithms

- NESL
- Java
- F#
- C#
- Cilk
- OpenMP
- MPI
- Parallel ML
- Threading Building Blocks (TBB)
- Scala
- Parallel ML
- MPI
- OpenMP
- Cilk
- C#
- F#
- Java
- NESL
OpenMP Overview

C$OMP FLUSH
C$OMP THREADPRIVATE(/ABC/)
C$OMP parallel do shared(a, b, c)
call OMP_INIT_LOCK (jlok)
C$OMP ATOMIC
C$OMP SINGLE PRIVATE(X)
setenv OMP_SCHEDULE “dynamic”
C$OMP PARALLEL DO ORDERED PRIVATE (A, B, C)
C$OMP PARALLEL REDUCTION (+: A, B)
C$OMP PARALLEL COPYIN(/blk/)
C$OMP DO lastprivate(XX)
Nthreads = OMP_GET_NUM_PROCS()
omp_set_lock(lck)

#pragma omp critical
CALL OMP_SET_NUM_THREADS(10)
call omp_test_lock(jlok)
C$OMP MASTER
OpenMP Execution Model

- Fork-join parallelism
- Traditionally flat parallelism
- Recently nested parallelism. But not very good.
Example: Numerical Integration

Mathematically, we know that:

\[ \int_{0}^{1} \frac{4.0}{1+x^2} \, dx = \pi \]

We can approximate the integral as a sum of rectangles:

\[ \sum_{i=0}^{N} F(x_i) \Delta x \approx \pi \]

where each rectangle has width \( \Delta x \) and height \( F(x_i) \) at the middle of interval \( i \).
The code for Approximating PI

```java
static long num_steps = 100000;
double step;
void main ()
{
    int i; double x, pi, sum = 0.0;

    step = 1.0/(double) num_steps;
x = 0.5 * step;
for (i=0;i<= num_steps; i++){
x+=step;
    sum += 4.0/(1.0+x*x);
}
pi = step * sum;
}
```
The code for Approximating PI

```c
#include <omp.h>
static long num_steps = 100000; double step;
void main ()
{
    int i; double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;
#pragma omp parallel for private(i, x) reduction(+:sum)
    for (i=0;i<= num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
}
```

Private clause creates data local to a thread

Reduction used to manage dependencies
Cilk/Cilk++: extending C/C++ with fork-join parallelism for shared-memory systems

Slides by Charles E. Leiserson
Supercomputing Technologies Research Group
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cilk Is Simple

• Cilk extends the C language with just a **handful** of keywords.
• Every Cilk program has a **serial semantics**.
• Not only is Cilk fast, it provides **performance guarantees** based on performance abstractions.
• Cilk is **processor-oblivious**.
• Cilk’s **provably good** runtime system automatically manages low-level aspects of parallel execution, including protocols, load balancing, and scheduling.
• Cilk supports **speculative** parallelism.
An Example: 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);
  }
}
```

Cilk is a **faithful** extension of C. A Cilk program’s **serial elision** is always a legal implementation of Cilk semantics. Cilk provides **no** new data types.
**Basic Cilk Keywords**

```c
#include <cilk.h>

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.
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: 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.

**Side benefit:**
D&C is generally good for caches!
How do the problems do on a modern multicore

![Bar Chart]

- Sort
- Duplicate Removal
- Min Spanning Tree
- Max Independent Set
- Spanning Forest
- Breadth First Search
- Delaunay Triangulation
- Triangle Ray Inter.
- Nearest Neighbors
- Sparse MxV
- Nbody
- Suffix Array
Demo Time
Concurrency versus Parallelism

- **Concurrency:** non-determinacy due to interleaving multiple processes. Property of the application.
- **Parallelism:** using multiple processors/cores running at the same time. Property of the machine.

<table>
<thead>
<tr>
<th>Parallelism</th>
<th>Concurrency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial</td>
<td>Deterministic</td>
</tr>
<tr>
<td></td>
<td>Traditional programming</td>
</tr>
<tr>
<td></td>
<td>Traditional OS</td>
</tr>
<tr>
<td>Parallel</td>
<td>Deterministic parallelism</td>
</tr>
<tr>
<td></td>
<td>Concurrent parallelism</td>
</tr>
</tbody>
</table>
Concurrency is Hard
Stack Example

• Serial
• Using locks
• Using compare and swap
• Using transactions

This is just a simple stack.
Concurrency : Stack Example 1

```c
struct link {int v; link* next;}

struct stack {
    link* headPtr;

    void push(link* a) {
        a->next = headPtr;
        headPtr = a;
    }

    link* pop() {
        link* h = headPtr;
        if (headPtr != NULL)
            headPtr = headPtr->next;
        return h;
    }
}
```
Concurrency: Stack Example 1

```c
struct link { int v; link* next; }

struct stack {
    link* headPtr;
    void push(link* a) {
        a->next = headPtr;
        headPtr = a;
    }
    link* pop() {
        link* h = headPtr;
        if (headPtr != NULL)
            headPtr = headPtr->next;
        return h;
    }
}
```
Concurrency: Stack Example 1

```c
struct link {int v; link* next;}

struct stack {
    link* headPtr;

    void push(link* a) {
        a->next = headPtr;
        headPtr = a;
    }

    link* pop() {
        link* h = headPtr;
        if (headPtr != NULL)
            headPtr = headPtr->next;
        return h;
    }
}
```
Concurrency : Stack Example 1

```c
struct link {int v; link* next;}
struct stack {
    link* headPtr;
    void push(link* a) {
        a->next = headPtr;
        headPtr = a;
    }
    link* pop() {
        link* h = headPtr;
        if (headPtr != NULL)
            headPtr = headPtr->next;
        return h;
    }
}
```
Stack Example

• Serial
• Using locks
• Using compare and swap
• Using transactions

This is just a simple stack.
Concurrency : Stack Example 2

```c
struct link {int v; link* next;}

struct stack {
  link* headPtr;

  void push(link* a) {
    withlock {
      a->next = headPtr;
      headPtr = a;
    }
  }

  link* pop() {
    withlock {
      link* h = headPtr;
      if (headPtr != NULL)
        headPtr = headPtr->next;
      return h;
    }
  }
}
```
Concurrency : Stack Example 2

Locks are bad for many reasons

- Easy to create deadlocks
- If a process stalls in a lock it holds everyone else up
Stack Example

• Serial
• Using locks
• Using compare and swap
• Using transactions

This is just a simple stack.
Concurrency : CAS

The **Compare and Swap** instruction is a built in instruction on many machines that atomically Compares a value in a register to one in memory and swaps in a new value if equal.

```c
bool CAS(int *ptr, int oldv, in newv) {
    if (*ptr = oldv) {
        *ptr = newv;
        return true;
    } else return false;
}
```

On x86 instruction set implemented with

```c
lock cmpxchg ptr, newv
```
oldv in EAX, puts flag in ZF
Concurrency : Stack Example 3

```c
struct stack {
    link* headPtr;

    void push(link* a) {
        do {
            link* h = headPtr;
            a->next = h;
            while (!CAS(&headPtr, h, a));
        }

        link* pop() {
            do {
                link* h = headPtr;
                if (h == NULL) return NULL;
                link* nxt = h->next;
                while (!CAS(&headPtr, h, nxt))
                return h;
            }
        }
    }
}
```
Concurrency : Stack Example 3

```c
struct stack {
    link* headPtr;

    void push(link* a) {
        do {
            link* h = headPtr;
            a->next = h;
            while (!CAS(&headPtr, h, a));
        } while (false);
    }

    link* pop() {
        do {
            link* h = headPtr;
            if (h == NULL) return NULL;
            link* nxt = h->next;
            while (!CAS(&headPtr, h, nxt))
                h = headPtr;
            return h;
        } while (false);
    }
}
```
Concurrency : Stack Example 3

```c
struct stack {
    link* headPtr;

    void push(link* a) {
        do {
            link* h = headPtr;
            a->next = h;
            while (!CAS(&headPtr, h, a));
        } while (h != NULL);
    }

    link* pop() {
        do {
            link* h = headPtr;
            if (h == NULL) return NULL;
            link* nxt = h->next;
            while (!CAS(&headPtr, h, nxt))
                return h;
        } while (h != NULL);
    }

};
```
Concurrency: Stack Example 3

```c
struct stack {
    link* headPtr;

    void push(link* a) {
        do {
            link* h = headPtr;
            a->next = h;
        } while (!CAS(&headPtr, h, a));
    }

    link* pop() {
        do {
            link* h = headPtr;
            if (h == NULL) return NULL;
            link* nxt = h->next;
            while (!CAS(&headPtr, h, nxt))
        } while (nxt = h->next);
        return h;
    }
}
```
Concurrent Stack Example 3’

P1 : x = s.pop(); y = s.pop(); s.push(x);
P2 : z = s.pop();

Before:  

After:  
P2: h = headPtr;
P2: nxt = h->next;
P1: everything
P2: CAS(&headPtr, h, nxt)

The ABA problem

Can be fixed with counter and 2CAS, but...
Stack Example

• Serial
• Using locks
• Using compare and swap
• Using transactions

This is just a simple stack.
Concurreny : Stack Example 4

struct link {int v; link* next;}

struct stack {
    link* headPtr;

    void push(link* a) {
        atomic {
            a->next = headPtr;
            headPtr = a;
        }
    }

    link* pop() {
        atomic {
            link* h = headPtr;
            if (headPtr != NULL) {
                headPtr = headPtr->next;
                return h;
            }
        }
    }

    "using transactional memory"
Concurrency : Stack Example 3’

```c
void swapTop(stack s) {
    link* x = s.pop();
    link* y = s.pop();
    push(x);
    push(y);
}
```

Queues are trickier than stacks.

Complications due to **non-determinacy** in interleaving of instructions/operations on shared data
Summary

• Hardware parallelism
  – Plenty and ubiquitous: datacenters, mainframes, GPUs, multicore chips on desktops and mobiles

• Parallel software is a major challenge
  – Good solutions in specific domains.
  – Inadequate programming systems.

• 15210
  – Parallel thinking at a high abstraction level
  – Parallelism, not concurrency. Avoid concurrency if not needed (e.g., for deterministic computations)