Introduction to Multicore Programming

Marc Moreno Maza

University of Western Ontario, London, Ontario (Canada)

CS 4435 - CS 9624
Plan

1. Multi-core Architecture
   - Multi-core processor
   - CPU Cache
   - CPU Coherence

2. Concurrency Platforms
   - PThreads
   - TBB
   - Open MP
   - Cilk ++
   - Race Conditions and Cilkscreen
   - MMM in Cilk++
Plan

1. Multi-core Architecture
   - Multi-core processor
   - CPU Cache
   - CPU Coherence

2. Concurrency Platforms
   - PThreads
   - TBB
   - Open MP
   - Cilk ++
   - Race Conditions and Cilkscreen
   - MMM in Cilk++
Chip Multiprocessor (CMP)

Memory

I/O

Network

P

P

P

(Moreno Maza)
Multi-core processor

- A multi-core processor is an integrated circuit to which two or more individual processors (called cores in this sense) have been attached.

- In a many-core processor the number of cores is large enough that traditional multi-processor techniques are no longer efficient.

- Cores on a multi-core device can be coupled tightly or loosely:
  - may share or may not share a cache,
  - implement inter-core communications methods or message passing.

- Cores on a multi-core implement the same architecture features as single-core systems such as instruction pipeline parallelism (ILP), vector-processing, SIMD or multi-threading.

- Many applications do not realize yet large speedup factors: parallelizing algorithms and software is a major on-going research area.
A **CPU cache** is an auxiliary memory which is smaller, faster memory than the main memory and which stores copies of the main memory locations that are expectedly frequently used.

Most modern desktop and server CPUs have at least three independent caches: the **data cache**, the **instruction cache** and the **translation look-aside buffer**.
Each location in each memory (main or cache) has
- a datum (cache line) which ranges between 8 and 512 bytes in size, while a datum requested by a CPU instruction ranges between 1 and 16.
- a unique index (called address in the case of the main memory)

In the cache, each location has also a tag (storing the address of the corresponding cached datum).
When the CPU needs to read or write a location, it checks the cache:

- if it finds it there, we have a cache hit
- if not, we have a cache miss and (in most cases) the processor needs to create a new entry in the cache.

Making room for a new entry requires a replacement policy: the Least Recently Used (LRU) discards the least recently used items first; this requires to use age bits.
CPU Cache (4/7)

- **Read latency** (time to read a datum from the main memory) requires to keep the CPU busy with something else:
  - **out-of-order execution**: attempt to execute independent instructions arising after the instruction that is waiting due to the cache miss
  - **hyper-threading (HT)**: allows an alternate thread to use the CPU
Modifying data in the cache requires a **write policy** for updating the main memory
- **write-through cache**: writes are immediately mirrored to main memory
- **write-back cache**: the main memory is mirrored when that data is evicted from the cache

The cache copy may become out-of-date or stale, if other processors modify the original entry in the main memory.
The replacement policy decides where in the cache a copy of a particular entry of main memory will go:

- **fully associative**: any entry in the cache can hold it
- **direct mapped**: only one possible entry in the cache can hold it
- **N-way set associative**: \( N \) possible entries can hold it
Cache Performance for SPEC CPU2000 by J.F. Cantin and M.D. Hill.

The SPEC CPU2000 suite is a collection of 26 compute-intensive, non-trivial programs used to evaluate the performance of a computer’s CPU, memory system, and compilers (http://www.spec.org/osg/cpu2000 ).
Cache Coherence (1/6)

Figure: Processor $P_1$ reads $x=3$ first from the backing store (higher-level memory)
Cache Coherence (2/6)

Figure: Next, Processor $P_2$ loads $x=3$ from the same memory
**Cache Coherence (3/6)**

**Figure:** Processor $P_4$ loads $x=3$ from the same memory
Cache Coherence (4/6)

Figure: Processor $P_2$ issues a write $x=5$
Cache Coherence (5/6)

**Figure:** Processor $P_2$ writes $x=5$ in his local cache.
Cache Coherence (6/6)

Figure: Processor $P_1$ issues a read $x$, which is now invalid in its cache
MSI Protocol

In this cache coherence protocol each block contained inside a cache can have one of three possible states:

- **M**: the cache line has been *modified* and the corresponding data is inconsistent with the backing store; the cache has the responsibility to write the block to the backing store when it is evicted.

- **S**: this block is unmodified and is *shared*, that is, exists in at least one cache. The cache can evict the data without writing it to the backing store.

- **I**: this block is *invalid*, and must be fetched from memory or another cache if the block is to be stored in this cache.

These coherency states are maintained through communication between the caches and the backing store.

The caches have different responsibilities when blocks are read or written, or when they learn of other caches issuing reads or writes for a block.
True Sharing and False Sharing

- **True sharing:**
  - True sharing cache misses occur whenever two processors access the same data word.
  - True sharing requires the processors involved to explicitly synchronize with each other to ensure program correctness.
  - A computation is said to have **temporal locality** if it re-uses much of the data it has been accessing.
  - Programs with high temporal locality tend to have less true sharing.

- **False sharing:**
  - False sharing results when different processors use different data that happen to be co-located on the same cache line.
  - A computation is said to have **spatial locality** if it uses multiple words in a cache line before the line is displaced from the cache.
  - Enhancing spatial locality often minimizes false sharing.

- See *Data and Computation Transformations for Multiprocessors* by J.M. Anderson, S.P. Amarasinghe and M.S. Lam
Advantages:
- Cache coherency circuitry operate at higher rate than off-chip.
- Reduced power consumption for a dual core vs two coupled single-core processors (better quality communication signals, cache can be shared)

Challenges:
- Adjustments to existing software (including OS) are required to maximize performance
- Production yields down (an Intel quad-core is in fact a double dual-core)
- Two processing cores sharing the same bus and memory bandwidth may limit performances
- High levels of false or true sharing and synchronization can easily overwhelm the advantage of parallelism
Plan

1. Multi-core Architecture
   - Multi-core processor
   - CPU Cache
   - CPU Coherence

2. Concurrency Platforms
   - PThreads
   - TBB
   - Open MP
   - Cilk ++
   - Race Conditions and Cilkscreen
   - MMM in Cilk++
Concurrency Platforms

- Programming directly on processor cores is painful and error-prone.
- Concurrency platforms
  - abstract processor cores, handles synchronization, communication protocols
  - (optionally) perform load balancing
- Examples of concurrency platforms:
  - Pthreads
  - Threading Building Blocks (TBB)
  - OpenMP
  - Cilk++
- We use an implementation of the Fibonacci sequence $F_{n+2} = F_{n+1} + F_n$ to compare these four concurrency platforms.
Fibonacci Execution

Key idea for parallelization
The calculations of $\text{fib}(n-1)$ and $\text{fib}(n-2)$ can be executed simultaneously without mutual interference.
PThreads

- Pthreads is a POSIX standard for threads, communicating though shared memory.

- Pthreads defines a set of C programming language types, functions and constants.

- It is implemented with a `pthread.h` header and a thread library.

- Programmers can use Pthreads to create, manipulate and manage threads.

- In particular, programmers can synchronize between threads using mutexes, condition variables and semaphores.

- This is a Do-it-yourself concurrency platform: programmers have to map threads onto the computer resources (static scheduling).
Key PThread Function

```c
ing pthread_create(
    pthread_t *thread,
    // returned identifier for the new thread
    const pthread_attr_t *attr,
    // object to set thread attributes (NULL for default)
    void *(func)(void *),
    // routine executed after creation
    void *arg
    // a single argument passed to func
) // returns error status

int pthread_join ( 
    pthread_t thread, 
    // identifier of thread to wait for
    void **status 
    // terminating thread's status (NULL to ignore) 
) // returns error status 

*WinAPI threads provide similar functionality. 
```
**Overhead:** The cost of creating a thread is more than 10,000 cycles. This enforces coarse-grained concurrency. (Thread pools can help.)

**Scalability:** Fibonacci code gets about 1.5 speedup for 2 cores for computing $\text{fib}(40)$.
- Indeed the thread creation overhead is so large that only one thread is used, see below.
- Consequently, one needs to rewrite the code for more than 2 cores.

**Simplicity:** Programmers must engage in error-prone protocols in order to schedule and load-balance.
```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;
    }
}

typedef struct {
    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
        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;
}
A C++ library that run on top of native threads

Programmers specify tasks rather than threads:
- Tasks are objects. Each task object has an input parameter and an output parameter.
- One needs to define (at least) the methods: one for creating a task and one for executing it.
- Tasks are launched by a spawn or a spawn_and_wait_for_all statement.

Tasks are automatically load-balanced across the threads using the work stealing principle.

TBB Developed by Intel and focus on performance
TBB provides many C++ templates to express common patterns simply, such as:

- `parallel_for` for loop parallelism,
- `parallel_reduce` for data aggregation
- pipeline and filter for software pipelining

TBB provides concurrent container classes which allow multiple threads to safely access and update items in the container concurrently.

TBB also provides a variety of mutual-exclusion library functions, including locks.
class FibTask: public task {
public:
    const long n;
    long* const sum;
    FibTask( long n_, long* sum_ ) :
        n(n_), sum(sum_) {}  
task* execute() {
        if( n < 2 ) {
            *sum = n;
        } else {
            long x, y;
            FibTask& a = *new( allocate_child() )
                FibTask(n-1,&x);
            FibTask& b = *new( allocate_child() )
                FibTask(n-2,&y);
            set_ref_count(3);
            spawn( b );
            spawn_and_wait_for_all( a );
            *sum = x+y;
        }
        return NULL;
    }
};

(Moreno Maza)
Open MP

- Several compilers available, both open-source and Visual Studio.
- Runs on top of native threads
- **Linguistic extensions to C/C++ or Fortran in the form of compiler pragmas** (compiler directives):
  - `# pragma omp task shared(x)` implies that the next statement is an independent task;
  - moreover sharing of memory is managed explicitly
  - other pragmas express directives for scheduling, loop parallelism and data aggregation.
- Supports loop parallelism and, more recently in Version 3.0, task parallelism with dynamic scheduling.
- OpenMP provides a variety of **synchronization constructs** (barriers, mutual-exclusion locks, etc.)
int fib(int n)
{
    if (n < 2) return n;
    int x, y;
    #pragma omp task shared(x)
    x = fib(n - 1);
    #pragma omp task shared(y)
    y = fib(n - 2);
    #pragma omp taskwait
    return x+y;
}
Concurrent Platforms

From Cilk to Cilk++

- Cilk has been developed since 1994 at the MIT Laboratory for Computer Science by Prof. Charles E. Leiserson and his group, in particular by Matteo Frigo.
- Besides being used for research and teaching, Cilk was the system used to code the three world-class chess programs: Tech, Socrates, and Cilkchess.
- Over the years, the implementations of Cilk have run on computers ranging from networks of Linux laptops to an 1824-nodes Intel Paragon.
- From 2007 to 2009 Cilk has lead to Cilk++, developed by Cilk Arts, an MIT spin-off, which was acquired by Intel in July 2009. Today, it can be freely downloaded. The place where to start is http://www.cilk.com/
- Cilk is still developed at MIT http://supertech.csail.mit.edu/cilk/
Cilk ++

- Cilk++ (resp. Cilk) is a small set of linguistic extensions to C++ (resp. C) supporting fork-join parallelism.
- Both Cilk and Cilk++ feature a provably efficient work-stealing scheduler.
- Cilk++ provides a hyperobject library for parallelizing code with global variables and performing reduction for data aggregation.
- Cilk++ includes the Cilkscreen race detector and the Cilkview performance analyzer.
Nested Parallelism in Cilk ++

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

- The named child function `cilk_spawn fib(n-1)` may execute in parallel with its parent.
- Cilk++ keywords `cilk_spawn` and `cilk_sync` grant permissions for parallel execution. They do not command parallel execution.
Loop Parallelism in Cilk ++

\[
\begin{pmatrix}
  a_{11} & a_{12} & \cdots & a_{1n} \\
  a_{21} & a_{22} & \cdots & a_{2n} \\
  \vdots & \vdots & \ddots & \vdots \\
  a_{n1} & a_{n2} & \cdots & a_{nn}
\end{pmatrix}
\]

\[
\begin{pmatrix}
  a_{11} & a_{21} & \cdots & a_{n1} \\
  a_{12} & a_{22} & \cdots & a_{n2} \\
  \vdots & \vdots & \ddots & \vdots \\
  a_{1n} & a_{2n} & \cdots & a_{nn}
\end{pmatrix}
\]

\[
A \rightarrow A^T
\]

\[
\begin{aligned}
// indices run from 0, not 1 \\
cilk_for (int i=1; i<n; ++i) {
  for (int j=0; j<i; ++j) {
    double temp = A[i][j];
    A[i][j] = A[j][i];
    A[j][i] = temp;
  }
}
\end{aligned}
\]

The iterations of a cilk_for loop may execute in parallel.
Cilk (resp. Cilk++) is a multithreaded language for parallel programming that generalizes the semantics of C (resp. C++) by introducing linguistic constructs for parallel control.

Cilk (resp. Cilk++) is a faithful extension of C (resp. C++):

- The C (resp. C++) elision of a Cilk (resp. Cilk++) is a correct implementation of the semantics of the program.
- Moreover, on one processor, a parallel Cilk (resp. Cilk++) program scales down to run nearly as fast as its C (resp. C++) elision.

To obtain the serialization of a Cilk++ program

```
#define cilk_for for
#define cilk_spawn
#define cilk_sync
```
Serial Semantics (2/2)

```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++ source**

**Serialization**

(Moreno Maza)
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);
    }
}
Scheduling (2/3)

- Cilk/Cilk++ randomized work-stealing scheduler load-balances the computation at run-time. Each processor maintains a ready deque:
  - A ready deque is a double ended queue, where each entry is a procedure instance that is ready to execute.
  - Adding a procedure instance to the bottom of the deque represents a procedure call being spawned.
  - A procedure instance being deleted from the bottom of the deque represents the processor beginning/resuming execution on that procedure.
  - Deletion from the top of the deque corresponds to that procedure instance being stolen.

- A mathematical proof guarantees near-perfect linear speed-up on applications with sufficient parallelism, as long as the architecture has sufficient memory bandwidth.

- A spawn/return in Cilk is over 100 times faster than a Pthread create/exit and less than 3 times slower than an ordinary C function call on a modern Intel processor.
Scheduling (2/3)

Each processor possesses a deque

push, pop, delete

queue end

stack end
Concurrency Platforms

Cilk ++

Introduction to Multicore Programming
Concurrency Platforms

Cilk ++

Moreno Maza

Introduction to Multicore Programming

CS 433 - CS 9624

53 / 60
The Cilk++ Platform

```cpp
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 (1/3)

Example

```c
int x = 0;
cilk_for(int i=0, i<2, ++i) {
    x++;
}
assert(x == 2);
```

Dependency Graph

- 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.
- The arguments to a spawned function are evaluated in the parent before the spawn occurs.
Race Bugs (2/3)

```
x = 0;
r1 = x;
r2 = x;
```

```
int x = 0;
r1++;
x = r1;
r2++;
x = r2;
```

```
x++;
(   2)
x++;
```

```
assert(x == 2);
```

```
r1++;  
```

```
x = r1;
```

```
r2++;  
```

```
x = r2;
```

```
assert(x == 2);
```
Race Bugs (3/3)

- Watch out for races in packed data structures such as:
  ```
  struct{
    char a;
    char b;
  }
  ```
  Updating `x.a` and `x.b` in parallel can cause races.

- If an ostensibly deterministic Cilk++ program run on a given input could possibly behave any differently than its serialization, Cilkscreen race detector guarantees to report and localize the offending race.

- Employs a regression-test methodology (where the programmer provides test inputs) and dynamic instrumentation of binary code.

- Identifies files-names, lines and variables involved in the race.

- Runs about 20 times slower than real-time.
template<typename T> void multiply_iter_par(int ii, int jj, int kk, T* A, T* B, T* C) {
    cilk_for(int i = 0; i < ii; ++i) {
        for (int k = 0; k < kk; ++k) {
            cilk_for(int j = 0; j < jj; ++j) {
                C[i * jj + j] += A[i * kk + k] + B[k * jj + j];
            }
        }
    }
}

Does not scale up well due to a poor locality and uncontrolled granularity.
```cpp
template<typename T> void multiply_rec_seq_helper(int i0, int i1, int j0, 
    int j1, int k0, int k1, T* A, ptrdiff_t lda, T* B, ptrdiff_t ldb, T* C, 
    ptrdiff_t ldc)
{
    int di = i1 - i0;
    int dj = j1 - j0;
    int dk = k1 - k0;
    if (di >= dj && di >= dk && di >= RECURSION_THRESHOLD) {
        int mi = i0 + di / 2;
        multiply_rec_seq_helper(i0, mi, j0, j1, k0, k1, A, lda, B, ldb, C, ldc);
        multiply_rec_seq_helper(mi, i1, j0, j1, k0, k1, A, lda, B, ldb, C, ldc);
    } else if (dj >= dk && dj >= RECURSION_THRESHOLD) {
        int mj = j0 + dj / 2;
        multiply_rec_seq_helper(i0, i1, j0, mj, k0, k1, A, lda, B, ldb, C, ldc);
        multiply_rec_seq_helper(i0, i1, mj, j1, k0, k1, A, lda, B, ldb, C, ldc);
    } else if (dk >= RECURSION_THRESHOLD) {
        int mk = k0 + dk / 2;
        multiply_rec_seq_helper(i0, i1, j0, j1, k0, mk, A, lda, B, ldb, C, ldc);
        multiply_rec_seq_helper(i0, i1, j0, j1, mk, k1, A, lda, B, ldb, C, ldc);
    } else {
        for (int i = i0; i < i1; ++i)
            for (int k = k0; k < k1; ++k)
                for (int j = j0; j < j1; ++j)
                    C[i * ldc + j] += A[i * lda + k] * B[k * ldb + j];
    }
}
```
template<typename T> inline void multiply_rec_seq(int ii, int jj, int kk, T* A, T* B, T* C) {
    multiply_rec_seq_helper(0, ii, 0, jj, 0, kk, A, kk, B, jj, C, jj);
}

Multiplying a 4000x8000 matrix by a 8000x4000 matrix

- on 32 cores = 8 sockets x 4 cores (Quad Core AMD Opteron 8354) per socket.
- The 32 cores share a L3 32-way set-associative cache of 2 Mbytes.

<table>
<thead>
<tr>
<th>#core</th>
<th>Elision (s)</th>
<th>Parallel (s)</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>420.906</td>
<td>51.365</td>
<td>8.19</td>
</tr>
<tr>
<td>16</td>
<td>432.419</td>
<td>25.845</td>
<td>16.73</td>
</tr>
<tr>
<td>24</td>
<td>413.681</td>
<td>17.361</td>
<td>23.83</td>
</tr>
<tr>
<td>32</td>
<td>389.300</td>
<td>13.051</td>
<td>29.83</td>
</tr>
</tbody>
</table>