

Introduction to Multicore Programming

Marc Moreno Maza

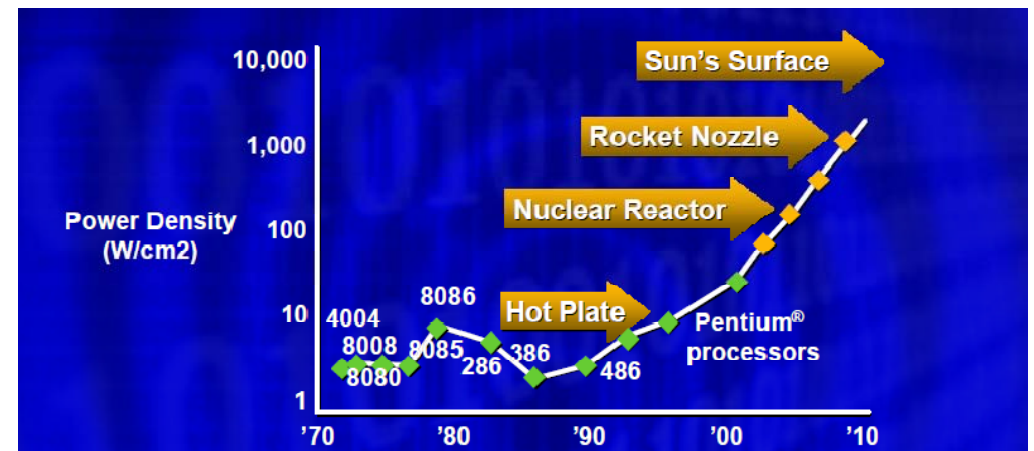
University of Western Ontario, London, Ontario (Canada)

CS 3101

- 1 Multi-core Architecture
 - Multi-core processor
 - CPU Coherence
- 2 Concurrency Platforms
 - An overview of Cilk++
 - Race Conditions and Cilkscreen
 - MMM in Cilk++

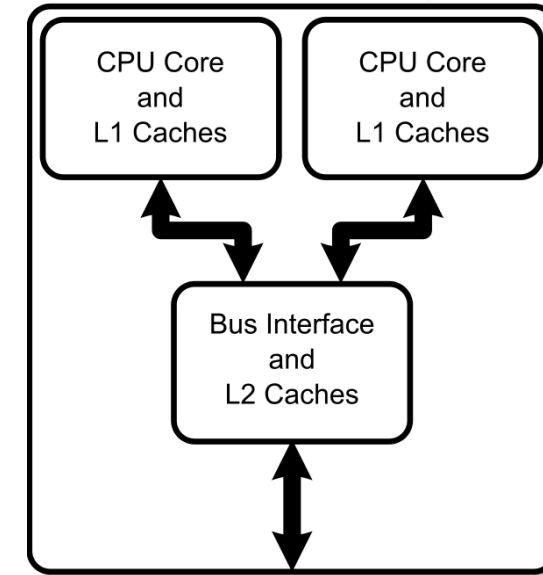
Plan

- 1 Multi-core Architecture
 - Multi-core processor
 - CPU Coherence
- 2 Concurrency Platforms
 - An overview of Cilk++
 - Race Conditions and Cilkscreen
 - MMM in Cilk++

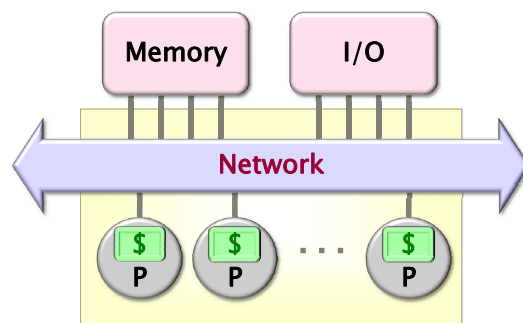




Dual CPU Core Chip



Multi-core processor



Chip Multiprocessor (CMP)

- 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**.

Cache Coherence (1/6)

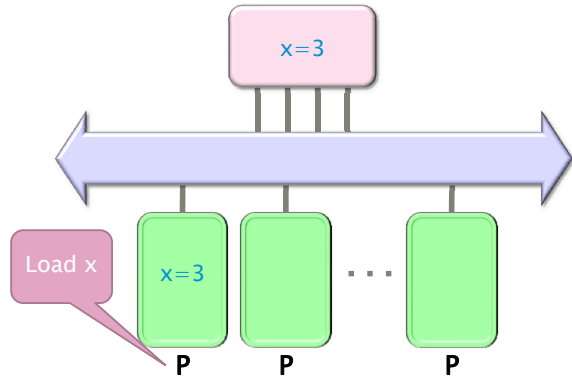


Figure: Processor P_1 reads $x=3$ first from the backing store (higher-level memory)

Cache Coherence (2/6)

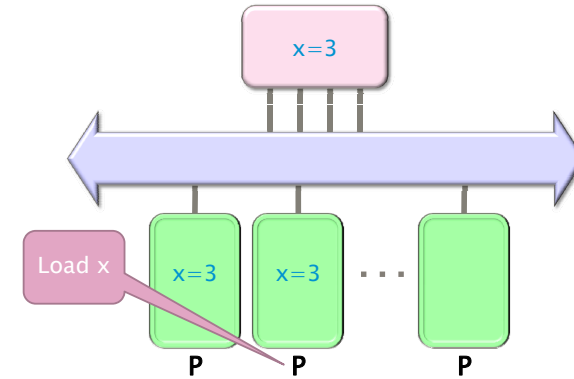


Figure: Next, Processor P_2 loads $x=3$ from the same memory

Cache Coherence (3/6)

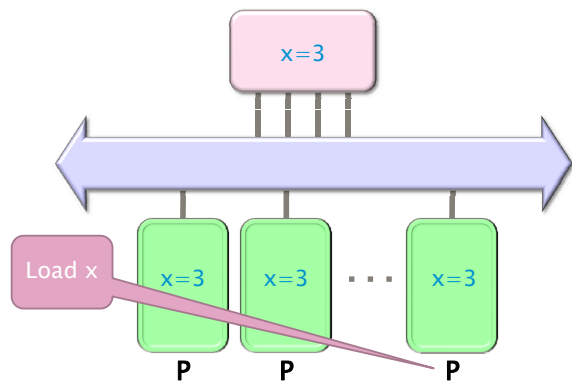


Figure: Processor P_4 loads $x=3$ from the same memory

Cache Coherence (4/6)

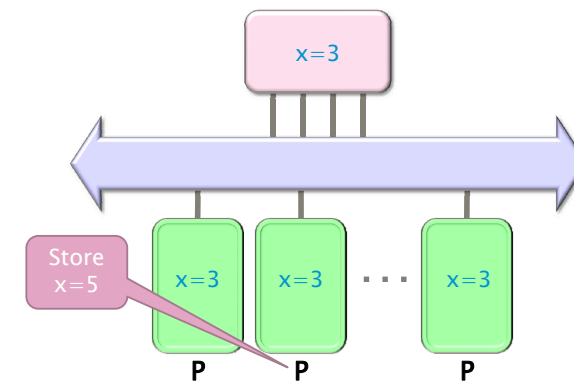
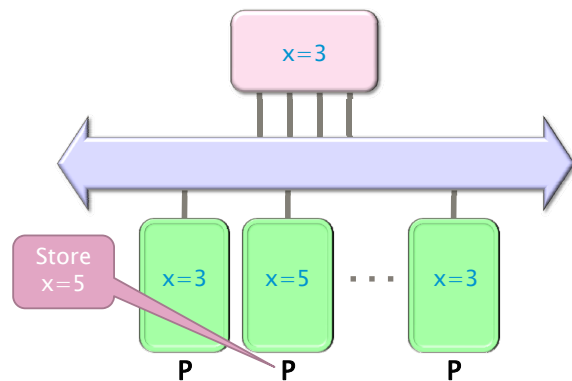
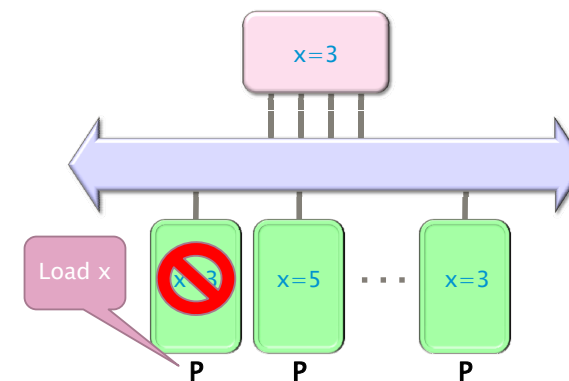


Figure: Processor P_2 issues a write $x=5$

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
<http://suif.stanford.edu/papers/anderson95/paper.html>

Multi-core processor (cntd)

Plan

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**

1 Multi-core Architecture

- Multi-core processor
- CPU Coherence

2 Concurrency Platforms

- An overview of Cilk++
- Race Conditions and Cilkscreen
- MMM in Cilk++

From Cilk to Cilk++

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 and became Cilk Plus, see <http://www.cilk.com/>
- Cilk++ can be freely downloaded at <http://software.intel.com/en-us/articles/download-intel-ci>
- Cilk is still developed at MIT <http://supertech.csail.mit.edu/cilk/>

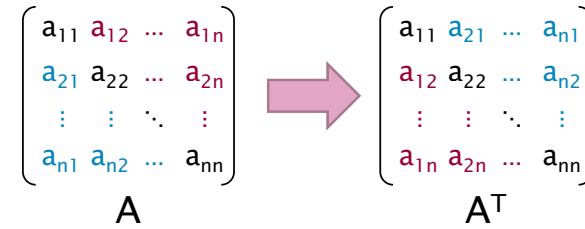
- Cilk++ (resp. Cilk) is a **small set of linguistic extensions to C++** (resp. C) supporting **fork-join parallelism**
- Both Cilk and Cilk++ feature a **provably efficient work-stealing scheduler**.
- Cilk++ provides a **hyperobject library** for parallelizing code with global variables and performing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

Nested Parallelism in Cilk ++

```
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** executes `fib(n-2)`.
- Cilk++ keywords `cilk_spawn` and `cilk_sync` grant **permissions for parallel execution**. They do not command parallel execution.

Loop Parallelism in Cilk ++



```
// 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;
    }
}
```

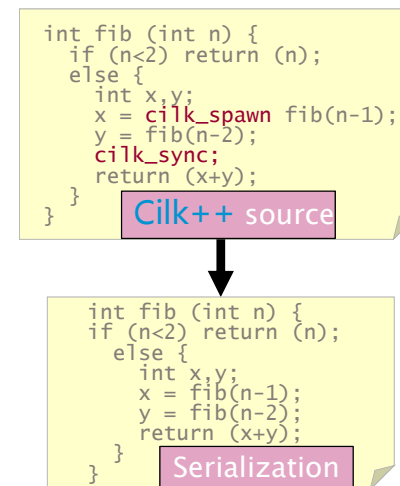
The iterations of a `cilk_for` loop may execute in parallel.

Serial Semantics (1/2)

- 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

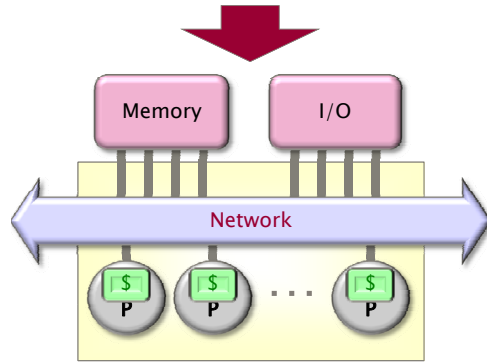
```
#define cilk_for for
#define cilk_spawn
#define cilk_sync
```

Serial Semantics (2/2)



Scheduling

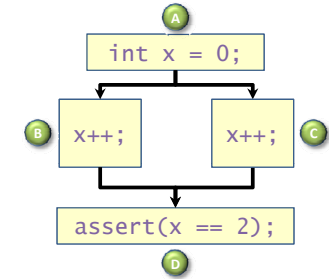
```
int fib (int n) {
  if (n<2) return (n);
  else {
    int x,y;
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
    cilk_sync;
    return (x+y);
  }
}
```



Race Bugs (1/3)

Example

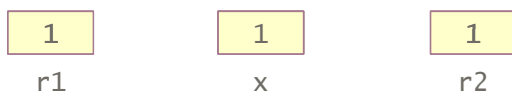
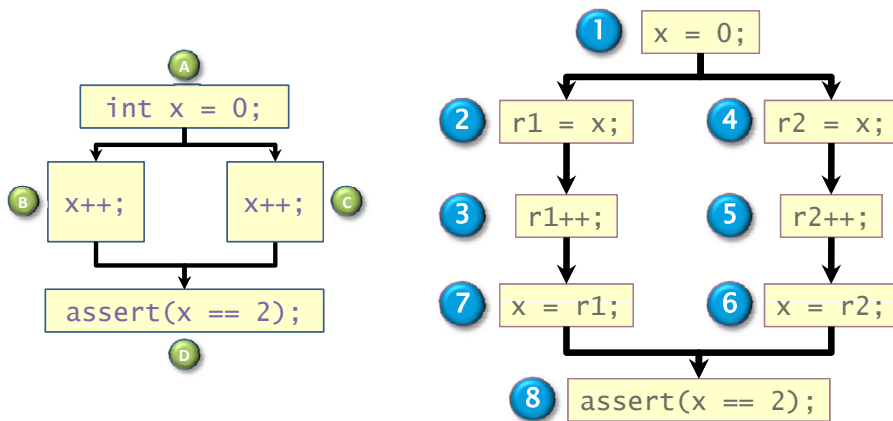
```
int x = 0;
cilk_for(int i=0, i<2, ++i) {
  x++;
}
assert(x == 2);
```



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)



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* 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.

```
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* B, T* C)
{
    multiply_rec_seq_helper(0, ii, 0, jj, 0, kk, A, kk, B, jj, C, j);
}
```

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.

#core	Elision (s)	Parallel (s)	speedup
8	420.906	51.365	8.19
16	432.419	25.845	16.73
24	413.681	17.361	23.83
32	389.300	13.051	29.83