Introduction to Parallel Computing

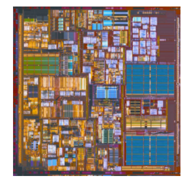
Tali Moreshet
Department of Engineering
Swarthmore College

SWARTHMORE

Computer Architecture
Summer School
08/20/08 - 08/21/08

Uniprocessor

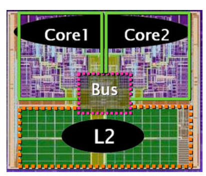
- Single processor on a chip
- Runs a single program at a time
- Moore's Law: The number of transistors on a chip doubles every ~2 years
  - Transistor size shrinks
  - Clock speeds increase
  - Can fit more logic on a chip
- Program performance increases with new processor generations



Intel Pentium4

Multiprocessor

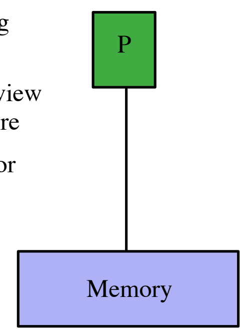
- But…
  complexity and power consumption also increase
- ⇒ More processors on a chip
  - Multi-core
  - Chip Multiprocessor (CMP)
- Clock speeds level off
- To increase program performance need to rewrite it!
  - Parallel programming



Intel Core Duo

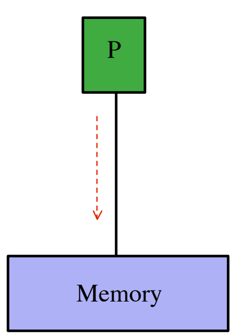
Processor and Memory

- Why is parallel programming more challenging?
- First, let's take a simplified view of microprocessor architecture
- Starting with the uniprocessor



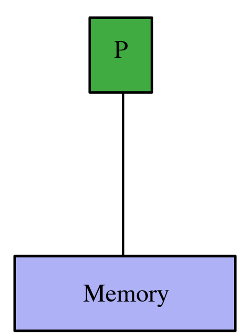
Processor and Memory

- Memory is located off-chip, far from the processor
- To read from memory:
  - Send address on bus
  - Wait for memory
  - Receive data from memory
- To write to memory:
  - Send address and data on bus
  - Possibly wait for an acknowledgement from memory



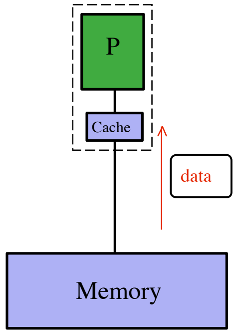
Processor and Memory

- Memory is large and slow
- How can we get data to the processor faster?



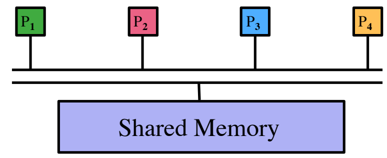
Cache

- A cache is memory that is:
  - Smaller
  - Faster
  - Closer to processor
  - Often on-chip
- To read from memory:
  - Send address on bus
  - Cache is searched first
  - Cache hit
    → shorter latency
  - Cache miss
    → send address to memory
    receive data from memory
    store in cache for later use



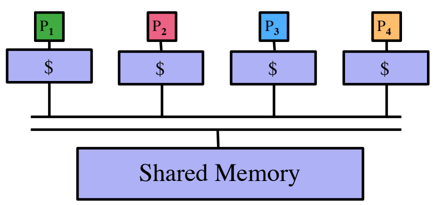
Multiprocessor Memory Architecture

- Processors connected to shared memory via a shared bus
- All processors see all memory activity
- Memory is large and slow
- How can we get data to the processor faster?



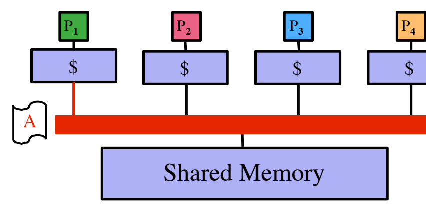
Multiprocessor Memory Architecture

- Each processor sees only its own local cache, and shared memory
- Local cache accesses are faster
- What are the new issues here?



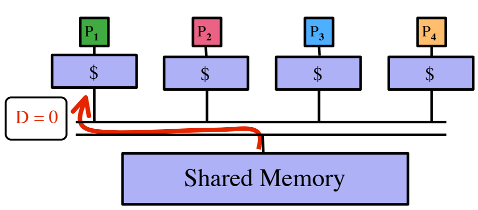
Memory Consistency

- P_1 broadcasts a read request for address A



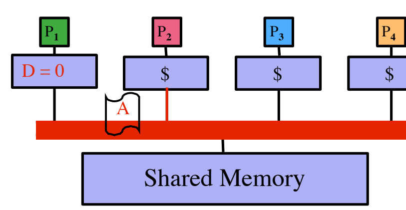
Memory Consistency

- Memory responds



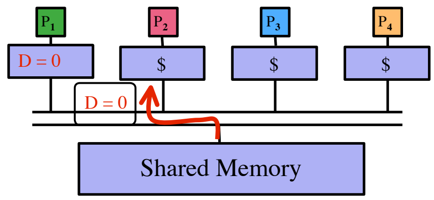
Memory Consistency

- P_2 broadcasts a read request for address A



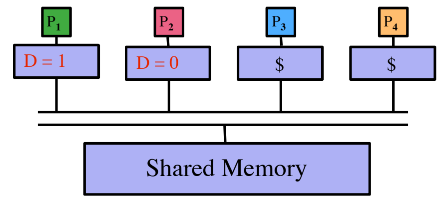
Memory Consistency

- Memory responds



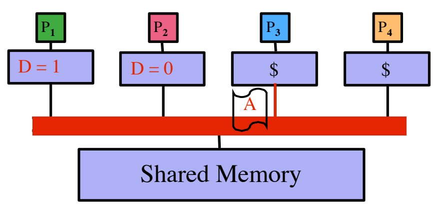
Memory Consistency

- Two different values for A exist in the system



Memory Consistency

- P_3 broadcasts a read request for address A
- Which data should it read?
- Cache coherence



Parallel Programming

- Serial execution of a single thread
  Multiple threads running concurrently

Challenges

- Splitting application to utilize cores
  - Ideally: number of threads == number of cores
- Balancing the work among cores
- Coordination among various code parts
  - All accessing a single shared memory
  - Unpredictable delays such as cache misses

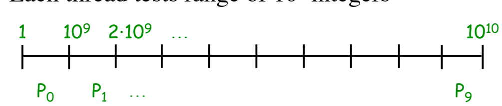
Prime Number Example

- Task:
  Print primes from 1 to 10^{10}
- Hardware:
  Ten-processor CMP
  One thread per processor
- Goal:
  Close to maximum possible speedup
  Ten fold speedup over uniprocessor (?)

Example adopted from
"Art of Multiprocessor Programming"
Herlihy-Shavit

Load Balancing

- Split the work evenly to 10 threads
- Each thread tests range of 10^9 integers

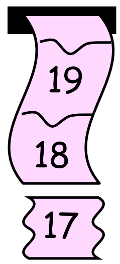


But

- Higher ranges have fewer primes
- Larger numbers are harder to test
- Workloads are uneven, hard to predict
- Need dynamic load balancing

Shared Counter

- Each thread takes a number
- Tests if prime
- Takes next available number
- Until no more numbers left



Procedure for thread *i*

```
int counter = new Counter(1);

void primePrint {
  long j = 0;
  while (j < 10^10) {
    j = counter.getAndIncrement();
    if (isPrime(j))
      print(j);
  }
}
```

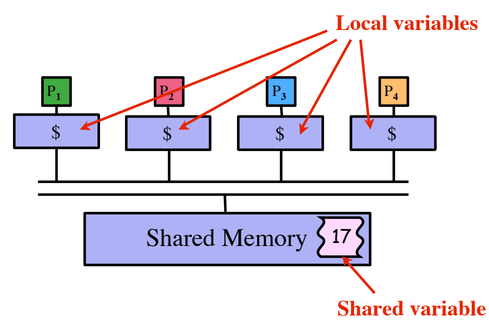

Procedure for thread *i*

```
int counter = new Counter(1);

void primePrint {
  long j = 0;
  while (j < 10^10) {
    j = counter.getAndIncrement();
    if (isPrime(j))
      print(j);
  }
}
```

Shared counter object

Where Are Variables Stored?



Procedure for thread *i*

```
int counter = new Counter(1);

void primePrint {
  long j = 0;
  while (j < 10^10) {
    j = counter.getAndIncrement();
    if (isPrime(j))
      print(j);
  }
}
```

Stop when every value taken

Procedure for thread *i*

```
int counter = new Counter(1);

void primePrint {
  long j = 0;
  while (j < 10^10) {
    j = counter.getAndIncrement();
    if (isPrime(j))
      print(j);
  }
}
```

Increment counter & return new value

Counter Implementation

```
public class Counter {
  private long value;

  public long getAndIncrement() {
    return value++;
  }
}
```


Counter Implementation

```
public class Counter {
  private long value;

  public long getAndIncrement() {
    return value++;
  }
}
```

temp = value;
value = value + 1;
return temp;

Counter:	1	2		3	2
Thread 1:	read 1	write 2	read 2	write 3	
Thread 2:	read 1				write 2

Counter Implementation

```
public class Counter {
  private long value;

  public long getAndIncrement() {
    temp = value;
    value = value + 1;
    return temp;
  }
}
```

Make these steps *atomic* (indivisible)

Hardware solution:
***ReadModifyWrite()* instruction**
Mutual exclusion

Counter Implementation

```
public class Counter {
  private long value;

  public long getAndIncrement() {
    synchronized {
      temp = value;
      value = value + 1;
    }
    return temp;
  }
}
```

Software solution:
Java synchronized block
Mutual exclusion

Mutual Exclusion

- Enable atomic execution of a code section
- Support available in hardware or software

Counter:	1	2		3	2
Thread 1:	read 1	write 2	read 2	write 3	
Thread 2:	read 1				write 2

Counter Implementation

```
public class Counter {
  private long value;

  public long getAndIncrement() {
    temp = value;
    value = value + 1;
    return temp;
  }
}
```

Make these steps *atomic* (indivisible)

General solution:
use locks

Locks

- Locks are means of providing mutual exclusion
- Prevent others from accessing atomic section
- Lock == 1 → lock is taken
  Lock == 0 → lock is free
- To acquire lock:
  Compare-and-Swap
  Atomic: Read lock from shared memory
  Compare to value 0
  Write 1 if compare returned 0
- To release lock:
  Write 0 to lock

Locks (Mutual Exclusion)

```
public interface Lock {
  public void lock();      acquire lock
  public void unlock();    release lock
}
```


Using Locks

```
public class Counter {
 private long value;
 private Lock lock;
 public long getAndIncrement() {
  lock.lock();
  try {
   int temp = value;
   value = value + 1;
  } finally {
    lock.unlock();
  }
  return temp;
 }}
```


Using Locks

```
public class Counter {
 private long value;
 private Lock lock;
 public long getAndIncrement() {
  lock.lock();          acquire lock
  try {
   int temp = value;
   value = value + 1;
  } finally {
    lock.unlock();
  }
  return temp;
 }}
```


Using Locks

```
public class Counter {
 private long value;
 private Lock lock;
 public long getAndIncrement() {
  lock.lock();
  try {
   int temp = value;
   value = value + 1;
  } finally {
    lock.unlock();      release lock
  }                     (no matter what)
  return temp;
 }}
```

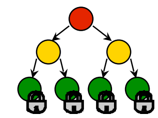

Using Locks

```
public class Counter {
 private long value;
 private Lock lock;
 public long getAndIncrement() {
  lock.lock();
  try {
   int temp = value;       Critical
   value = value + 1;      Section
  } finally {
    lock.unlock();
  }
  return temp;
 }}
```

Parallel Programming for Performance

- Load balancing
- Reduce idle time when threads wait
- Maximize parallel portion of code
- Minimize sequential parts
  - Small critical sections
  - Fine-grained synchronization

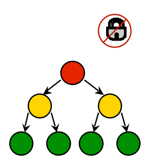
Disadvantages of Locks



- Coarse-grain
  - High contention (on the lock)
  - Low throughput
- Fine-grain
  - Difficult to program and debug
  - Deadlock, interrupt
- Spin-locks - repetitive accesses until free
  - Many memory accesses
  - Useless work
- Alternatives: blocking, queue locks

Transactional Memory

- Lock-free synchronization
- Transaction: Atomic section *lock()* → *unlock()*
- Speculative execution – optimistic
  - No conflicts → commit
  - Conflicts detected → roll back, reissue
- Hardware requirements
  - Additional memory or dedicated cache
  - Changes to cache coherence protocol
- May also be implemented in software



Parallel Programming

- Parallelize as much of the code as possible
- Minimize sequential parts
- Be careful with mutual exclusion
  - Requires waiting
  - Different approaches
  - Each has its advantages

Questions?

- Java code snippets adopted from
  "Art of Multiprocessor Programming", Herlihy-Shavit