Lecture 3
Announcements

• Lab hours have been posted
Using Bang

- Do not use Bang’s *front end* for heavy computation
- Use batch, or interactive nodes, via *qlogin*
- Use the front end for editing & compiling only
Today’s lecture

• Synchronization
• The Mandelbrot set computation
• Measuring Performance
Recapping from last time: inside a data race

- Assume $x$ is initially 0
  
  ```
  x = x + 1;
  ```

- Generated assembly code
  ```
  r1 ← (x)
  r1 ← r1 + #1
  r1 → (x)
  ```

- Possible interleaving with two threads
  
  P1
  ```
  r1 ← x
  r1 ← r1 + #1
  x ← r1
  ```

  P2
  ```
  r1 ← x
  r1 ← r1 + #1
  x ← r1
  ```

  r1(P1) gets 0
  r2(P2) also gets 0
  r1(P1) set to 1
  r1(P1) set to 1
  P1 writes R1
  P2 writes R1
CLICKERS OUT
How many possible interleavings (including reorderings) of instructions with 2 threads?

A. 6
B. An infinite number
C. 20
D. 15

For n threads and m instructions there are \((nm)! / ((m!)^n)\) possible orderings

http://math.stackexchange.com/questions/77721/number-of-instruction-interleaving

1 Possible interleaving with two threads

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>r1 ← x</td>
<td>r1 ← x</td>
</tr>
<tr>
<td>r1</td>
<td>r1 ← r1+ #1</td>
<td>r1 ← r1+ #1</td>
</tr>
<tr>
<td>x</td>
<td>x ← r1</td>
<td>x ← r1</td>
</tr>
</tbody>
</table>

r1(P1) gets 0  
r2(P2) also gets 0  
r1(P1) set to 1  
r1(P1) set to 1  
P1 writes R1  
P2 writes R1
Avoiding the data race in summation

• Perform the global summation in `main()`
• After a thread joins, add its contribution to the global sum, one thread at a time
• We need to wrap `std::ref()` around reference arguments, `int64_t &`, compiler needs a hint*

```cpp
void
sum(int TID, int N, int NT
    int64_t& localSum ){
    ....
    for (int i=i0;  i<i1;  i++)
        localSum += x[i];
}
```

```cpp
int64_t global_sum;
...
int64_t *locSums = new int64_t[NT];
for(int t=0; t<NT; t++)
    thrds[t] = thread(sum,t,N,NT,ref(locSums[t]));
for(int t=0; t<NT; t++){
    thrds[t].join();
    global_sum += locSums[t];
}
```

* Williams, pp. 23-4
Creating references in thread callbacks

• The thread constructor copies each argument into local private storage, once the thread has launched.

• Consider this thread launch and join, where $V = 77$ before the launch:

```
thrds[t] = thread(Fn,t,V)
    ....
thrds[t].join();
```

• Here is the thread function:

```
void Fn(int TID, int& Result){
    ....
    Result = 100;
}
```

• What is the value of $V$ after we join the thread?
What is value of V after the join?

V=77;
thrds[t] = thread(Fn, t, V)
...
thrds[t].join();

Thread function

void Fn(int TID, int& Result){
...
    Result = 100;
}

A. Not defined
B. 100
C. 77
Creating references in thread callbacks

• When we use `ref()` we are telling the compiler to generate a reference to \( V \). A copy of this reference is passed to \( \text{Fn} \)
\[
\text{thrds}[t] = \text{thread}((\text{Fn}, t, \text{ref}(V))
\]

• By copying a reference to \( V \), rather than \( V \) itself, we are able to update \( V \). Otherwise, we’d update the copy of \( V \)

• Using `ref()` is helpful in other ways: it avoids the costly copying overhead when \( V \) is a large struct

• Arrays need not be passed via `ref()`
Strategies for avoiding data races

• Restructure the program
  ‣ Migrate shared updates into main

• Program synchronization
  ‣ Critical sections
  ‣ Barriers
  ‣ Atomics
Critical Sections

• Our brute for solution of forcing all global update to occur within a single thread is awkward and can by costly
• In practice, we synchronize inside the thread function
• We need a way to permit only 1 thread at a time to write to the shared memory location(s)
• The code performing the operation is called a critical section
• We use mutual exclusion to implement a critical section
• A critical section is non-parallelizing computation. What are sensible guidelines for using it?

```
Begin Critical Section
x++;
End Critical Section
```
What sensible guidelines should we use to keep the cost of critical sections low?

A. Keep the critical section short
B. Avoid long running operations
C. Avoid function calls
D. A & B
E. A, B and C
Using mutexes in C++

- The `<mutex>` library provides a mutex class
- A mutex (AKA a "lock") may be CLEAR or SET
  - `Lock()` waits if the lock is set, else sets lock & exits
  - `Unlock()` clears the lock if in the set state

```cpp
void sum(int TID, int N, int NT){
    ...  
    for (int64_t i=i0; i<i1; i++)
        localSum += x[i];
    // Critical section
    mutex_sum.lock();
    global_sum += localSum;
    mutex_sum.unlock();
}
```

Globals:
- `int* x;`
- `mutex mutex_sum;`
- `int64_t global_sum;`
Should Mutexes be ....

A. Local variables
B. Global variables
C. Of either type

A local variable mutex would arise in a thread function that spawned other threads. We would have to pass the mutex via the thread function. In effect, the threads treat the mutex as a global. Not fully global since threads outside of the invoking thread would not see the mutex.

A cleaner solution is to encapsulate locks as class members.
Today’s lecture

• Synchronization

• The Mandelbrot set computation

• Measuring Performance
A quick review of complex numbers

- Define $i = \sqrt{-1}$, $i^2 = -1$
- A complex number $z = x + iy$
  - $x$ is called the real part
  - $y$ is called the imaginary part
- Associate each complex number with a point in the $x$-$y$ plane
- The magnitude of a complex number is the same as a vector length: $|z| = \sqrt{x^2 + y^2}$
- $z^2 = (x + iy)(x + iy)$
  $$= (x^2 - y^2) + 2xyi$$
What is the value of $(3i)(-4i)$?

A. 12  
B. -12  
C. 3-4i
The Mandelbrot set

• Named after B. Mandelbrot
• For which points $c$ in the complex plane does the following iteration remain bounded?
  \[ z_{k+1} = z_k^2 + c, \quad z_0 = 0 \]
  $c$ is a complex number
• Plot the rate at which points in a given region diverge
• Plot $k$ at each position
• The Mandelbrot set is “self similar:” it exhibits recursive structures
Convergence

\[ z_{k+1} = z_k^2 + c, \quad z_0 = 0 \]

- When \( c=0 \) we have
  \[ z_{k+1} = z_k^2 \]
- When \( |z_k| \geq 2 \) the iteration is guaranteed to diverge to \( \infty \)
- Stop the iterations when \( |z_{k+1}| \geq 2 \) or \( k \) reaches some limit
- For any point within a unit disk \( |z| \leq 1 \) we always remain there, so count = \( \infty \)
- Plot \( k \) at each position
Programming Lab #1

- Mandelbrot set computation with C++ threads
- Observe speedups on up to 8 cores
- Load balancing
- Assignment will be automatically graded
  - Tested for correctness
  - Performance measurements
- Serial Provided code available via GitLab
- Start early
Parallelizing the computation

- Split the computational box into regions, assigning each region to a thread
- Different ways of subdividing the work
- “Embarrassingly” parallel, so no communication between threads
Load imbalance

• Some points iterate longer than others
• If we use uniform BLOCK decomposition, some threads finish later than others
• We have a load imbalance

\[ \text{do} \]
\[ z_{k+1} = z_k^2 + c \]
\[ \text{until} \ (|z_{k+1}| \geq 2) \]
for $i = 0$ to $n-1$
  for $j = 0$ to $n-1$
    $z = \text{Complex}(x[i],y[i])$
    while ($|z| < 2$ or $k < \text{MAXITER}$)
      $z = z^2 + c$
    Output$[i,j] = k$
Load balancing efficiency

- If we ignore serial sections and other overheads, we express load imbalance in terms of a **load balancing efficiency** metric.
- Let each processor \( i \) complete its assigned work in time \( T(i) \).
- Thus, the running time on \( P \) cores: \( T_P = \text{MAX} (T(i)) \).

Define

\[
\bar{T} = \sum_i T(i)
\]

We define the **load balancing efficiency**

\[
\eta = \frac{\bar{T}}{PT_P}
\]

- Ideally \( \eta = 1.0 \).
If we are using 2 cores & one core carries 25% of the work, what is $T_2$, assuming $T_1 = 1$?

Note: $T_p$ is the running time on $p$ cores, and is different from $T(i)$, the running time on the $i^{th}$ core

A. 0.25
B. 0.75
C. 1.0

$$\eta = \frac{\bar{T}}{PT_p}$$
Load balancing strategy

• Divide rows into bundles of CHUNK consecutive rows
• Processor k gets chunks spaced chunkSize*NT rows apart
• So core 1 gets strips@ 2*1, 2+1*2*3, 2+2*2*3
• A block cyclic decomposition can balance the workload
• Also called round robin or block cyclic
Changing the input

- Exploring different regions of the bounding box will result in different workload distributions

-b -2.5 -0.75 0 1
-b -2.5 -0.75 -0.25 0.75

i=100 and
i=1000