

Lecture 4

Announcements

- New lab hours arrangement
- Fixed time each week
- Web portal to sign up for help, during times that staff contact hours have been scheduled
- Stay tuned

How many possible interleavings
(including reorderings) of instructions with 2
threads ?

A. 6

B. An infinite number

C. 20

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

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

D. 15

1 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

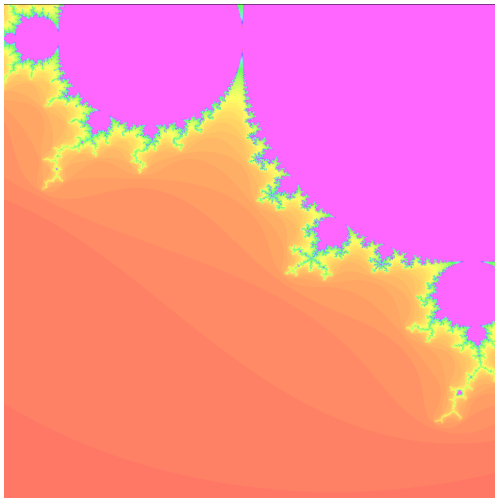
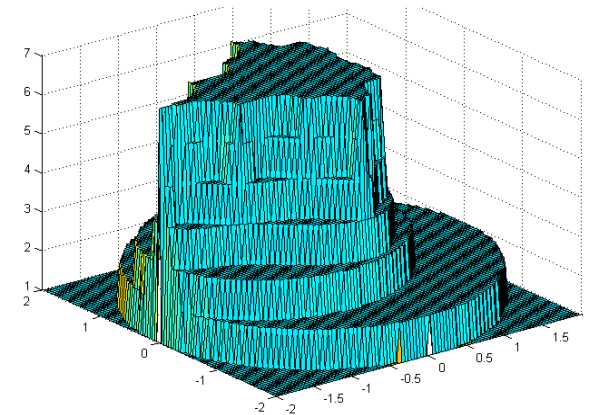
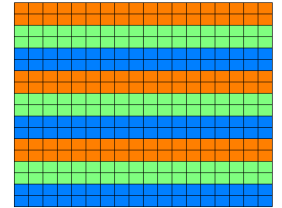
r1(P1) set to 1

P1 writes R1

P2 writes R1

Load imbalance

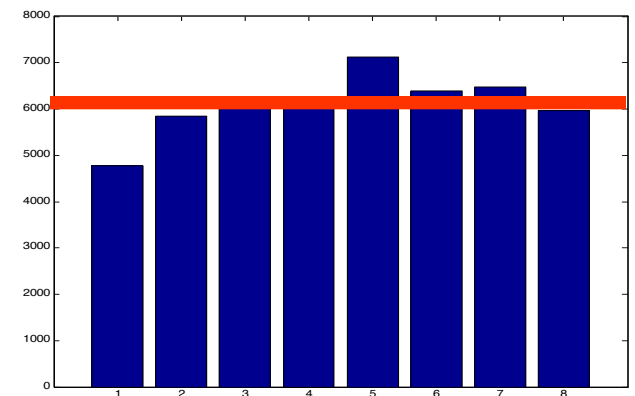
- Some points iterate **longer** than others
- To avoid a load imbalance, we'll use block cyclic partitioning
- The running time T_P is that of the longest running core $\text{MAX}(T(i))$
- Note difference between $T(i)$ & T_P



do

$$z_{k+1} = z_k^2 + c$$

until ($|z_{k+1}| \geq 2$)



Today's lecture

- Measuring Performance
- Using locks correctly
- Testing and Debugging

Measuring performance

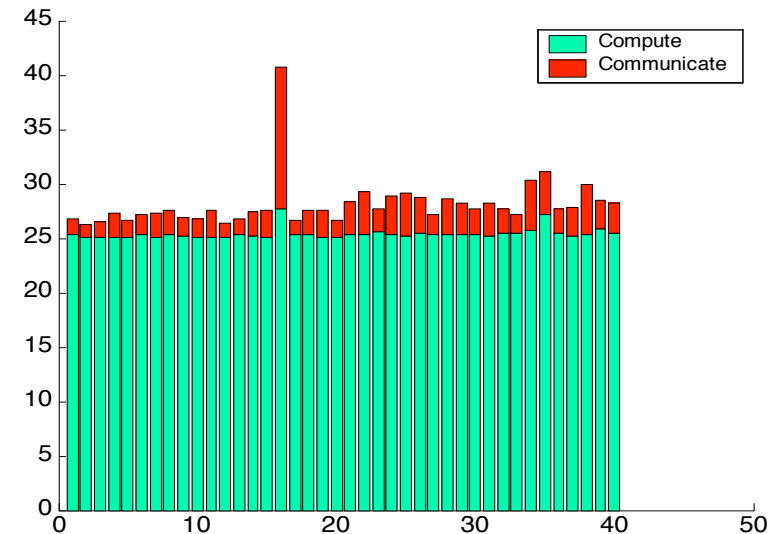
- Performance is the goal in parallel computation
- We'll measure elapsed, or “wall clock” time
`gettimeofday( )`
- Performance measurement tools: *gprof* or *valgrind* [to be discussed]
- Report where you ran, software versions, CPU
 - ▶ `uname -a`
Linux ccom-bang-login.local 2.6.32-431.29.2.el6.x86_64 #1 SMP Tue Sep 9 21:36:05 UTC 2014 x86_64 x86_64 x86_64 GNU/Linux
 - ▶ `gcc --version`
gcc version 4.8.4 (GCC)
 - ▶ `/proc/cpuinfo`
`/sys/devices/system/cpu/cpu0, cpu1, .... P-1`

Challenges to measuring performance

- Measurements are imprecise
 - “Heisenberg uncertainty principle:” measurement technique may affect performance
 - Overheads, quantization, clock resolution
- May be impractical to time a complete run
 - Measure a representative sample, ignore startup
- Transient behavior can affect reproducibility
 - Repeat your measurements
 - “Warm up” the code before collecting measurements
 - Average time, maximum time, minimum time?

Measurement collection

- ▶ Repeat results 3 to 5 times until at least 2 measurements agree to within 5% or 10%
- Report outliers only if significant
- A scatter plot or error bar can be useful



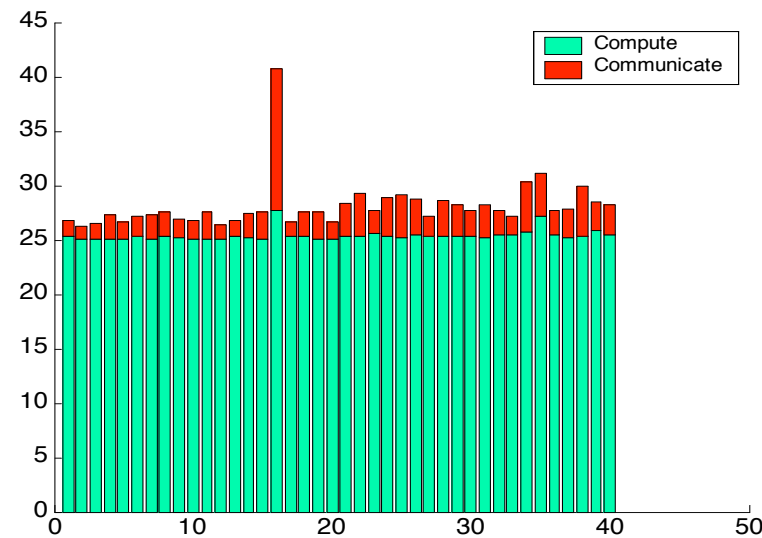
CLICKERS OUT

What time should we report?

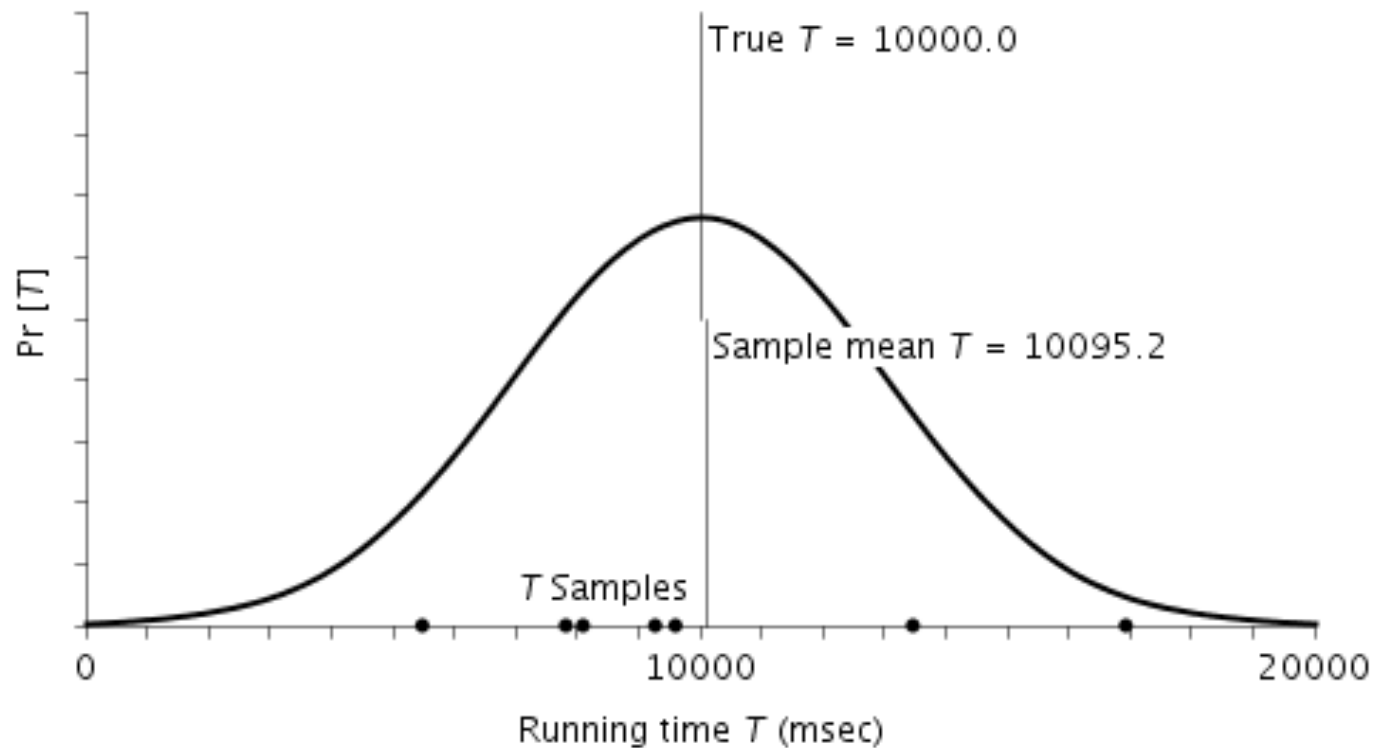
A. Average

B. Minimum

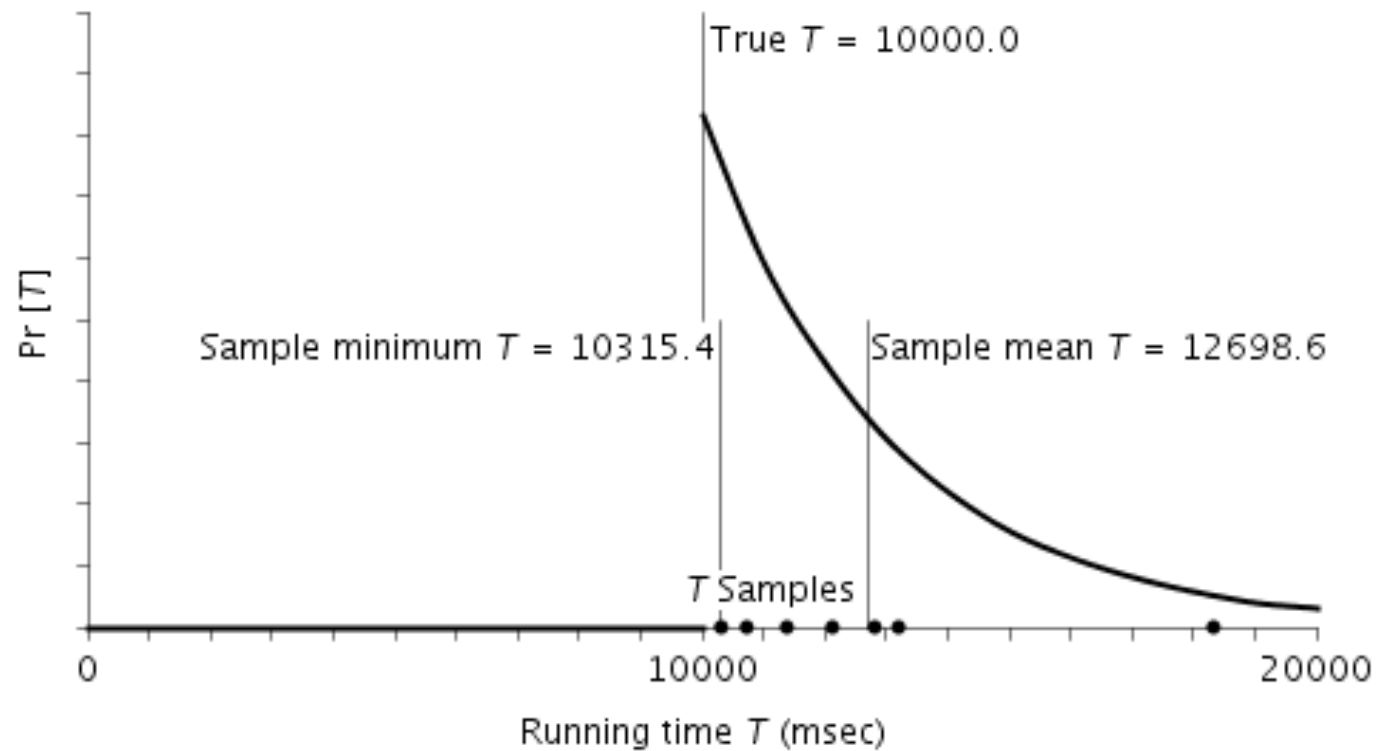
C. Maximum



Why do we take the minimum time?



Measurement errors are not distributed symmetrically



Summary of today's lecture

- If we want to modify arguments passed to a thread function, we need to use `ref( )`
- We use mutexes to implement critical sections. Supported by the C++ thread library
- Lab 1: implement the Mandelbrot set computation, including load balancing
- Report the minimum time when taking repeated timings, measuring wall clock time

Today's lecture

- Measuring Performance
- Using locks correctly
- Testing and Debugging

Making sure we unlock the lock

- When we use mutexes, we need to be sure to unlock the lock when the critical section finishes
 - Failing to do so in a timely manner can hurt performance
 - Failing to do so at all can lead to *deadlock*
- A common problem is to have no way of unlocking the lock when a function exits abnormally, i.e. an exception
- C++ provides the `lock_guard` class template to help avoid this difficulty and we should use this instead of `mutex.lock` and `unlock`
- Implements the *RAII* principle *Williams 38-40*
- *Resource Allocation is Initialization*

Why should we unlock the lock in a timely manner?

- A. The mutex is serial work so other processors may have to wait unnecessarily
- B. To prevent deadlock

RAII and Lock_guard

- The lock_guard constructor acquires (locks) the provided lock constructor argument
- When a lock_guard destructor is called, it releases (unlocks) the lock

```
int val;  
std::mutex valMutex;  
...  
{  
    std::lock_guard<std::mutex> lg(valMutex); // lock and automatically unlock  
    if (val >= 0)  
        f(val);  
    else  
        f(-val); // pass negated negative val  
} // ensure lock gets released here
```

Linked list example

```
std::list<int> some_list;  
std::mutex some_mutex;  
void add_to_list(int new_value) {  
    std::lock_guard<std::mutex> guard(some_mutex);  
    some_list.push_back(new_value);  
}  
bool list_contains(int value_to_find) {  
    std::lock_guard<std::mutex> guard(some_mutex);  
    return std::find(some_list.begin(),some_list.end(),value_to_find)  
        != some_list.end();  
}
```



Object oriented design

```
std::mutex some_mutex;
```

```
Class MyList {
```

```
private:
```

```
    std::list<int> some_list;
```

```
public:
```

```
void add_to_list(int new_value) {
```

```
    std::lock_guard<std::mutex> guard(some_mutex);
```

```
    some_list.push_back(new_value);
```

```
}
```

```
bool list_contains(int value_to_find) {
```

```
    std::lock_guard<std::mutex> guard(some_mutex);
```

```
    return std::find(some_list.begin(),some_list.end(),value_to_find)
```

```
        != some_list.end();
```

```
}
```

```
}
```



Should we put the mutex inside the class definition?

A. Yes

B. No

```
Class MyList {
```

```
private:
```

```
    std::list<int> some_list;
```

```
    std::mutex some_mutex;
```

```
public:
```

```
void add_to_list(int new_value) {
```

```
    std::lock_guard<std::mutex>
```

```
    guard(some_mutex);
```

```
    some_list.push_back(new_value);
```

```
}
```

```
.... };
```

Code Structure Issues

- Beware of returning or passing references to members that must be updated with synchronization
- Think about what can eventually be done with those references, and use care in making a call to an external function within a critical section

```
class s_data {  
...  
public:  
    void do_something();  
};
```

```
class data_wrapper {  
private:  
    s_data data;  
    std::mutex m;  
public:  
    template<typename Function>  
    void process_data(Function func) {  
        std::lock_guard<std::mutex> l(m);  
        func(data);  
    }  
};
```

Accidentally passing a reference to unprotected data

- Beware of returning or passing references to members that must be updated with synchronization
- Think about what can eventually be done with those references, and use care in making a call to an external function within a critical section

```
class some_data { ...  
public:  
    void do_something();  
    some_data* unprotected;  
    void a_fn(s_data& protected_data) {  
        unprotected=&protected_data;  
    }  
    data_wrapper x;  
    void foo() {  
        x.process_data(a_function);  
        unprotected->do_something();  
    }  
}
```

```
class data_wrapper {  
private:  
    some_data data;  
    std::mutex m;  
public:  
    template<typename Function>  
    void process_data(Function func) {  
        std::lock_guard<std::mutex> l(m);  
        func(data);  
    }  
};
```

Accidentally passing a reference to unprotected data

```
void malicious_fn(some_data& protected_data) {  
    unprotected=&protected_data;  
}  
data_wrapper x;  
some_data* unprotected;
```

```
void foo() {  
    x.process_data(malicious_fn);  
    unprotected->do_something();  
}
```

```
class some_data {  
    int a;  
    std::string b;  
public:  
    void do_something();  
};
```

```
class data_wrapper {  
private:  
    some_data data;  
    std::mutex m;  
public:  
    template<typename Function>  
    void process_data(Function func) {  
        std::lock_guard<std::mutex> l(m);  
        func(data);  
    }  
};
```

What can go wrong with the call to `malicious_fn()`?

- A. It could deadlock on the mutex `m`
- B. It permit accesses w/o mutual exclusion**
- C. Both A and B
- D. I'm not sure

```
some_data* unprotected;  
void malicious_fn(some_data& protected_data) {  
    unprotected=&protected_data;  
}
```

```
data_wrapper x;  
void main() {  
    x.process_data(malicious_fn);  
    unprotected->do_something();  
}
```

```
class data_wrapper {  
private:  
    some_data data;  
    std::mutex m;  
public:  
    template<typename Function>  
    void process_data(Function func) {  
        std::lock_guard<std::mutex> l(m);  
        func(data);  
    }  
};
```

Problems with the traditional stack interface

```
stack<int> s;  
if (!s.empty()){  
    int const value = s.top();  
    s.pop( );  
    do_something(value);  
}
```

```
bool empty() const;  
T const& top() const;  
void push(T const&);  
void pop();
```

Thread A	Thread B
<pre>if(!s.empty()) int const value=s.top(); s.pop(); do_something(value);</pre>	<pre>if(!s.empty()) int const value=s.top(); s.pop(); do_something(value);</pre>

C++ Concurrency in Action, A. Williams, Manning (2012)

What could or could not go wrong?

- A. Both threads could process the same value
- B. Both threads could process neither value
- C. Each thread could process a unique value (i.e. correct behavior)
- D. A & C are correct**
- E. B & C are correct

Thread A	Thread B
<pre>if(!s.empty()) int const value=s.top(); s.pop(); do_something(value);</pre>	<pre>if(!s.empty()) int const value=s.top(); s.pop(); do_something(value);</pre>

What if we had more than 2 threads?

- A. Threads A&B could process the same value
- B. Both threads could process neither value
- C. Each thread could process a unique value (i.e. correct behavior)
- D. A & C are correct
- E. A, B & C are correct

Thread A	Thread B
<pre>if(!s.empty()) int const value=s.top(); s.pop(); do_something(value);</pre>	<pre>if(!s.empty()) int const value=s.top(); s.pop(); do_something(value);</pre>

Building a safe Stack

```
#include <exception>
```

```
#include <memory>
```

```
struct empty_stack: std::exception {  
    const char* what() const throw(); };
```

```
template<typename T>
```

```
class thread_safe_stack {
```

```
public:
```

```
    thread_safe_stack();
```

```
    thread_safe_stack(const thread_safe_stack&);
```

```
    thread_safe_stack& operator=(const thread_safe_stack&) =
```

```
delete;
```

```
    void push(T new_value);
```

```
    std::shared_ptr<T> pop();
```

```
    void pop(T& value);
```

```
    bool empty() const;
```

```
};
```

```
try { s.pop() }  
catch ( .. ) {  
    cout << "Empty Stack!\n" << end;  
}
```

Complete Stack code

```
private:
    std::stack<T> data;
    mutable std::mutex m;
public:
    void push(T new_value) {
        std::lock_guard<std::mutex> lock(m);
        data.push(new_value);
    }
    void pop(T& value) {
        std::lock_guard<std::mutex> lock(m);
        if(data.empty()) throw empty_stack();
        value=data.top();
        data.pop();
    }
    bool empty() const{
        std::lock_guard<std::mutex> lock(m);
        return data.empty();
    }
}
```

Today's lecture

- Measuring Performance
- Using locks correctly
- Testing and Debugging

Bugs

- Not in my code!
- Multithreading introduces two new aspects to debugging
 - Timing
 - Sharing

What are the main causes of errors in threaded program?

A. Race Conditions

B. Deadlock

C. A & B

Race Conditions

- The most common culprit
- Data Races
 - Special type of race condition
- Broken invariants
 - e.g. dangling pointers
- Lifetime issues
- Some race conditions are not exceptional and benign: seat assignments

Best practices

- Isolate serial code from the parallel code and test accordingly
- Design for testability
- Test cases – *think in parallel*
- Brute force testing can give false positives

Have you ever walked a way from a buggy program and done something else, enabling you to find the bug?

- A. Yes
- B. No
- C. Not sure

There is no “right” answer to this question.

Debugging Rules (Agans)

1. Understand the system
2. Make it fail
3. Quit thinking and look
4. Divide and conquer
5. Change one thing at a time
6. Keep an audit trail
7. Check the plug
8. Get a fresh view
9. If you didn't fix it, it ain't fixed

Fin