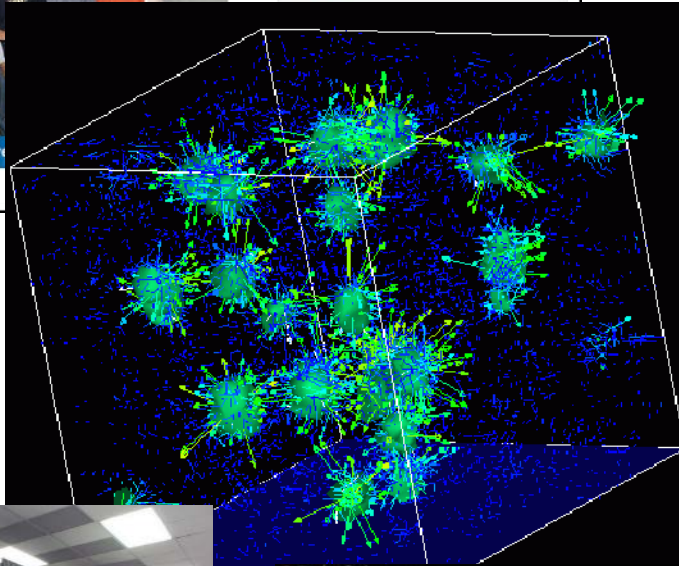




# Survey of Scientific Computing (SciComp 301)

Dave Biersach  
Brookhaven National  
Laboratory  
[dbiersach@bnl.gov](mailto:dbiersach@bnl.gov)



```
1 using System;
2 using System.Collections.Generic;
3 using System.ComponentModel;
4 using System.Data;
5 using System.Drawing;
6 using System.Linq;
7 using System.Text;
8 using System.Windows.Forms;
9
10 namespace SimpleEvents
11 {
12     public partial class Form1 : Form
13     {
14         Person person = new Person();
15
16         public Form1()
17         {
18             InitializeComponent();
19             person.FirstName = "Christian";
20             person.LastName = "Pano";
21         }
22
23         private void button1_Click(object sender, EventArgs e)
24         {
25             person.MainColor = textBox1.Text;
26         }
27     }
28 }
```

**Session 27**  
Parallel Programming  
Using Threads

# Session Goals

- Learn about PC system architecture (symmetric multiprocessing vs **multicore**)
- Appreciate OS task and thread scheduling (multitasking vs. **multithreading**)
- Implement the **Thread Control Block (TCB) pattern** of central dispatch control, in order to **parallelize numerical integration using Simpson's Rule**

# Admiral Grace Hopper



**Grace Brewster Murray Hopper** (née Murray December 9, 1906 – January 1, 1992) was an American computer scientist and United States Navy rear admiral.<sup>[1]</sup> One of the first programmers of the Harvard Mark I computer, she was a pioneer of computer programming who invented one of the first linkers. She popularized the idea of machine-independent programming languages, which led to the development of COBOL, an early high-level programming language still in use today.

Prior to joining the Navy, Hopper earned a Ph.D. in mathematics from Yale University and was a professor of mathematics at Vassar College. Hopper attempted to enlist in the Navy during World War II but was rejected because she was 34 years old. She instead joined the Navy Reserves. Hopper began her computing career in 1944 when she worked on the Harvard Mark I team led by Howard H. Aiken. In 1949, she joined the Eckert–Mauchly Computer Corporation and was part of the team that developed the UNIVAC I computer. At Eckert–Mauchly she began developing the compiler. She believed that a programming language based on English was possible. Her compiler converted English terms into machine code understood by computers. By 1952, Hopper had finished her program linker (originally called a compiler), which was written for the A-0 System.<sup>[2][3][4][5]</sup> During her wartime service, she co-authored three papers based on her work on the Harvard Mark 1.

# Admiral Grace Hopper

In accordance with Navy attrition regulations, Hopper retired from the Naval Reserve with the rank of commander at age 60 at the end of 1966.<sup>[26]</sup> She was recalled to active duty in August 1967 for a six-month period that turned into an indefinite assignment. She again retired in 1971 but was again asked to return to active duty in 1972. She was promoted to captain in 1973 by Admiral Elmo R. Zumwalt, Jr.<sup>[27]</sup>

After Republican Representative Philip Crane saw her on a March 1983 segment of *60 Minutes*, he championed H.J.Res. 341<sup>[27]</sup>, a joint resolution originating in the House of Representatives, which led to her promotion on 15 December 1983 to commodore by special Presidential appointment by President Ronald Reagan.<sup>[27][28][29][30]</sup> She remained on active duty for several years beyond mandatory retirement by special approval of Congress.<sup>[31]</sup> Effective November 8, 1985, the rank of commodore was renamed rear admiral (lower half) and Hopper became one of the Navy's few female admirals.





# My Dinner with Admiral Hopper - 1987

“When farmers wanted to plow bigger fields, they didn’t breed bigger and bigger horses...



In 1987 Admiral Hopper came to West Point (USMA) to receive an award for a lifetime of service. I was selected to have dinner with her before the presentation...

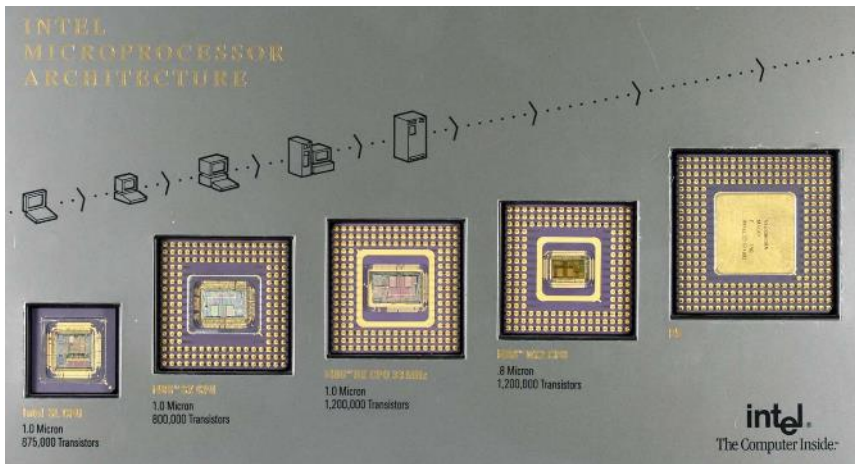
... so I asked for her thoughts about the future of computing...

... they learned how to stitch multiple horses together.”



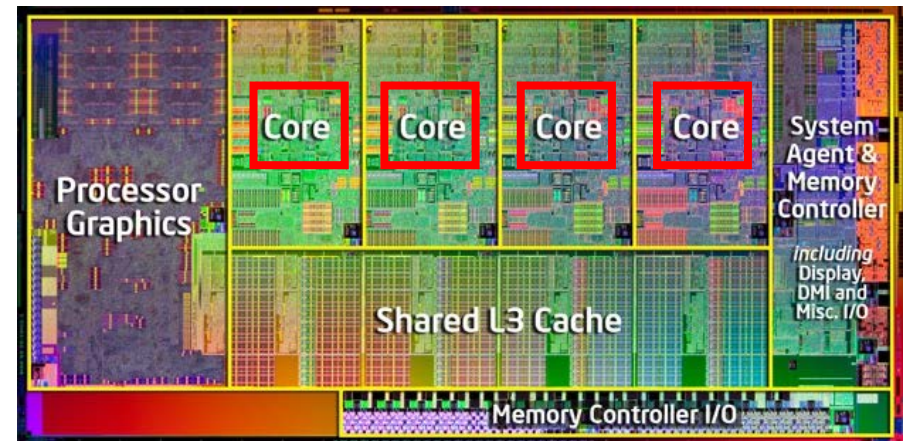
# My Dinner with Admiral Hopper - 1987

“When farmers wanted to plow bigger fields, they didn’t breed bigger and bigger horses...



**Intel CPUs from 1985 - 2008**

... they learned how to stitch multiple horses together.”



**Intel CPUs from 2008 - 2020**

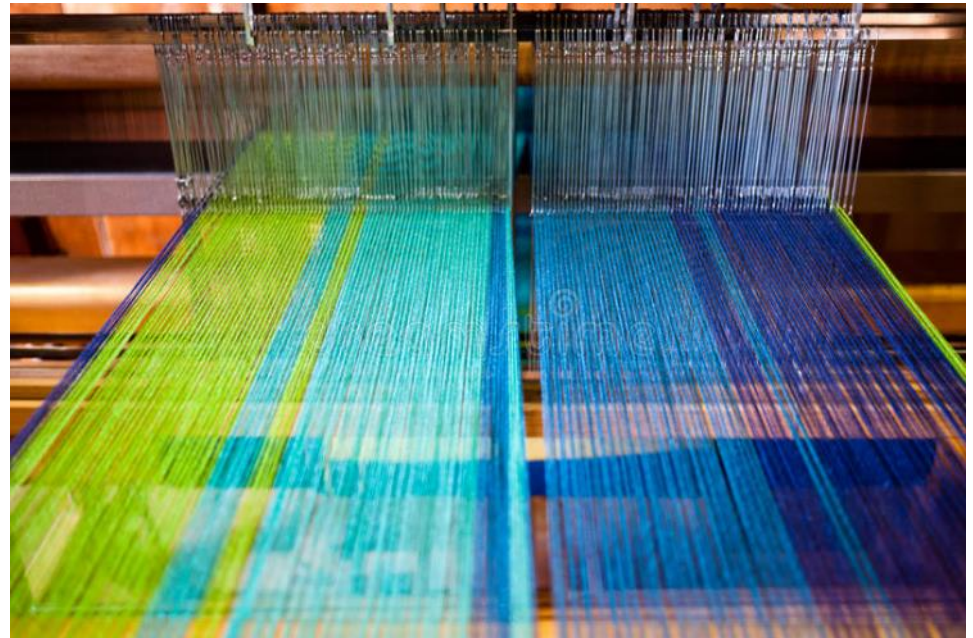
# My Dinner with Admiral Hopper - 1987

“A task is like a blanket.  
A multitasking OS switches  
between blankets...



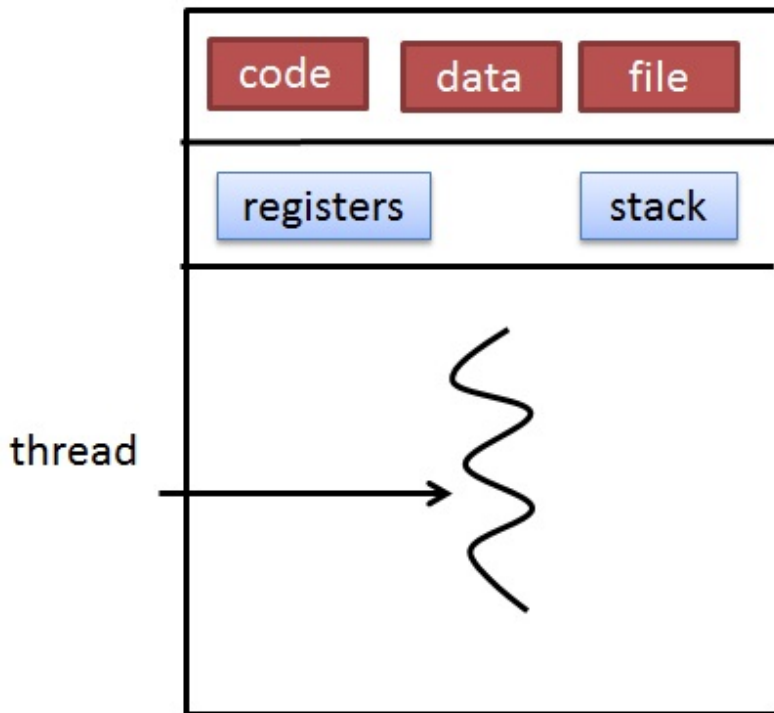
... but a blanket is made from individual  
threads. A multithreaded OS switches  
between the parts of each blanket.

Future programming will all be about  
how to *properly* use **threads**.”

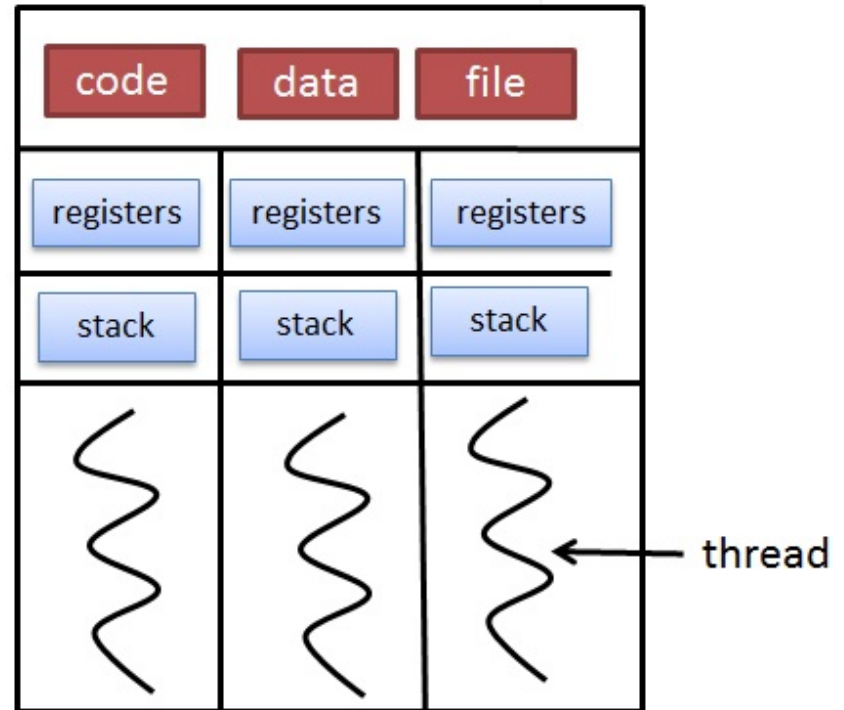




# A “Thread” of Execution



Single-threaded Process



Multithreaded Process



# System Architecture

- Due to Moore's law and exponentially rising power demands, system designers moved away from computers having multiple physical CPUs (**SMP** - symmetric multi-processing)
  - The new model became “multi-core” with computers having one main CPU that is divided internally into multiple execution cores
  - Each core can run independently from the others
  - System memory (global variables) are shared between all cores – but each core gets its own local (stack based) variables
- Multicore designs are lower cost and demand less power since core-to-core communication (wiring) is all “on-chip” (inside the CPU) versus SMP where wiring is on the circuit board connecting the physically separated CPUs

# CPU $\Leftrightarrow$ OS Coordination

- A single-threaded application can only run on **one** core at a time – the other cores (and even other CPUs in an SMP machine) will be *underutilized* (a waste of money)
- Modern programmers try to extract maximum performance from the locally available hardware
  - Applications must be designed to run sections of the code in **parallel** on multiple threads (and hence multiple cores) at the same time
  - For time sensitive apps, a good software design will keep each CPU Core **utilization near 100%** for the duration of the run
  - However there is no sense in launching too many threads – the OS then spends too much time **context switching** (*thrashing*) between all the threads trying to ensure no thread gets starved for CPU time

# Managing Threads

- Each application has as single **primary thread** of execution, created by the OS when the program is launched
  - All apps start off as **single-threaded**
  - You as the developer must explicitly *ask* the OS (via system calls) to **create** additional threads to run parts of your code in parallel (to the maximum degree logically possible)
- You control the lifetime of the child threads you create
  - The **primary thread** is controlled by the **OS** and is terminated *only* when your application ends
  - Any child thread you create is killed by the OS when the main thread dies, so you must ensure all of your child threads **cleanly exit** before **main() returns**

# Using the C++ Thread Library

- Threads cannot “see” or modify the local variables in another thread – threads can only modify **global** (heap) memory
  - Threads start running the instant you create them!
  - Each thread has **its own stack** – it gets its **own copy** of any local variables, including passed in function parameters
- You don’t want to spin a thread in a “runnable” (hot) state
  - A thread should invoke **yield()** to surrender its time quantum back to the OS scheduler if there is no meaningful work that can be done
  - Threads are often waiting for an external resource to continue
- C++ has **native** support for creating & managing threads



# Open Lab 1

## Simple Threading

```
#include "stdafx.h"

using namespace std;

void DisplayThreadId(){
    cout << " {threadid = "
    << this_thread::get_id()
    << "}" << endl;
}

void func(string msg ) {
    cout << "enter func()";
    DisplayThreadId();
    cout << "\t" << msg << endl;
    cout << "exit func()";
    DisplayThreadId();
}

int main(){
    cout << "enter main()";
    DisplayThreadId();
    cout << "\tStarting new thread..." << endl;
    thread t1(func, "Threading is cool!");
    // Wait here for thread 1 to exit
    t1.join();
    cout << "exit main()";
    DisplayThreadId();
    return 0;
}
```

- `this_thread::get_id()`
- `thread()` constructor
- A thread **function**
- Passing thread parameters
- Wait for exit via `t1.join()`

# Run Lab 1

## Simple Threading

```
#include "stdafx.h"

using namespace std;

void DisplayThreadId(){
    cout << " {threadid = "
        << this_thread::get_id()
        << "}" << endl;
}

void func(string msg ) {
    cout << "enter func()";
    DisplayThreadId();
    cout << "\t" << msg << endl;
    cout << "exit func()";
    DisplayThreadId();
}

int main(){
    cout << "enter main()";
    DisplayThreadId();
    cout << "\tStarting new thread..." << endl;
    thread t1(func, "Threading is cool!");
    // Wait here for thread 1 to exit
    t1.join();
    cout << "exit main()";
    DisplayThreadId();
    return 0;
}
```

- `this_thread::get_id()`
- `thread()` constructor
- A thread **function**
- Passing thread parameters
- Wait for exit via `t1.join()`

```
simple-threading
File Edit View Terminal Tabs Help
enter main() {threadid = 140228593728704
    Starting new thread...
enter func() {threadid = 140228354119424
    Threading is cool!
exit func() {threadid = 140228354119424
exit main() {threadid = 140228593728704

Process returned 0 (0x0)   execution time : 0.043 s
Press ENTER to continue.
```

# Open Lab 2

## Mutex

```
#include "stdafx.h"

using namespace std;

void DisplayThreadId()
{
    cout << " {threadid = "
        << this_thread::get_id()
        << "}" << endl;
}

void func(string msg, string symbol, int count)
{
    cout << msg << " enter func()";
    DisplayThreadId();
    for (int i{}; i < count; ++i)
    {
        cout << symbol;
        this_thread::sleep_for(1ns);
    }
    cout << endl << msg << " exit func()";
    DisplayThreadId();
}

int main()
{
    cout << "enter main()";
    DisplayThreadId();
    thread t1(func, "Thread 1", "[", 10);
    thread t2(func, "Thread 2", "@@", 20);
    t1.join();
    t2.join();
    cout << "exit main()";
    DisplayThreadId();
    return 0;
}
```

- Shared thread function
- Two simultaneous threads
- `this_thread::sleep()`
- C++ STL Numeric *literals*

```
#include "stdafx.h"

using namespace std;

void DisplayThreadId()
{
    cout << " {threadid = "
         << this_thread::get_id()
         << "}" << endl;
}

void func(string msg, string symbol, int count)
{
    cout << msg << " enter func()";
    DisplayThreadId();
    for (int i{}; i < count; ++i)
    {
        cout << symbol;
        this_thread::sleep_for(1ns);
    }
    cout << endl << msg << " exit func()";
    DisplayThreadId();
}

int main()
{
    cout << "enter main()";
    DisplayThreadId();
    thread t1(func, "Thread 1", "[", 10);
    thread t2(func, "Thread 2", "@@", 20);
    t1.join();
    t2.join();
    cout << "exit main()";
    DisplayThreadId();
    return 0;
}
```

- ```

mutex
File Edit View Terminal Tabs Help

enter main() {threadid = 140526266379456}
Thread 2 enter func() {threadid = 140526018377472}
@@Thread 1 enter func() {threadid = 140526026770176}
[[@@[[@@[[@@[[@@[[@@[[@@[[@@[[@@
Thread 1 exit func() {threadid = 140526026770176}
@@@@@@@@@@@@@@@@@@@@
Thread 2 exit func() {threadid = 140526018377472}
exit main() {threadid = 140526266379456}

Process returned 0 (0x0)    execution time : 0.038 s
Press ENTER to continue.

```



Mutex = Mutually Exclusive (only one owner)

## Edit Lab 2

# Mutex

```
main.cpp X
1  #include "stdafx.h"
2
3  using namespace std;
4
5  mutex g_console_mutex;
6
7  void DisplayThreadId()
8  {
9      cout << " {threadid = "
10         << this_thread::get_id()
11         << "}" << endl;
12  }
13
14  void func(string msg, string symbol, int count)
15  {
16      g_console_mutex.lock();
17      cout << msg << " enter func()";
18      DisplayThreadId();
19      for (int i{}; i < count; ++i)
20      {
21          cout << symbol;
22          this_thread::sleep_for(1ns);
23      }
24      cout << endl << msg << " exit func()";
25      DisplayThreadId();
26      g_console_mutex.unlock();
27  }
28
```

- Declare global mutex variable
- This thread **locks** the mutex
- Perform assigned work
- Thread **unlocks** the mutex

## Mutex = Mutually Exclusive (only one owner)

# Run Lab 2

## Mutex

```
main.cpp X
1  #include "stdafx.h"
2
3  using namespace std;
4
5  mutex g_console_mutex;
6
7  void DisplayThreadId()
8  {
9      cout << " {threadid = "
10         << this_thread::get_id()
11         << "}" << endl;
12  }
13
14  void func(string msg, string symbol, int count)
15  {
16      g_console_mutex.lock();
17      cout << msg << " enter func()";
18      DisplayThreadId();
19      for (int i{}; i < count; ++i)
20      {
21          cout << symbol;
22          this_thread::sleep_for(1ns);
23      }
24      cout << endl << msg << " exit func()";
25      DisplayThreadId();
26      g_console_mutex.unlock();
27  }
28
```

- Declare global mutex variable
- This thread **locks** the mutex
- Perform assigned work
- Thread **unlocks** the mutex
- Threads are now *synchronized*

```

mutex
File Edit View Terminal Tabs Help

enter main() {threadid = 140664053075136}
Thread 2 enter func() {threadid = 140663805073152}
aa
Thread 2 exit func() {threadid = 140663805073152}
Thread 1 enter func() {threadid = 140663813465856}
[] [] [] [] [] [] [] []
Thread 1 exit func() {threadid = 140663813465856}
exit main() {threadid = 140664053075136}

Process returned 0 (0x0)   execution time : 0.049 s
Press ENTER to continue.

```

## Open Lab 3

# Race Condition

```
#include "stdafx.h"

using namespace std;

std::atomic<int> g_counter{};
std::atomic<bool> g_won{false};

void func(string name)
{
    while(g_counter > 0)
    {
        g_counter--;
        std::this_thread::sleep_for(1us);
    }
    if (!g_won)
    {
        g_won = true;
        cout << name << " wins!" << endl;
    }
}

int main()
{
    for (int run{}; run < 10; run++)
    {
        cout << "Race " << run << ": ";
        g_counter = 20000;
        g_won = false;

        thread t1(func, "Thread 1");
        thread t2(func, "Thread 2");

        t1.join();
        t2.join();
    }
    return 0;
}
```

- Two simultaneous threads
- Decrement a global counter
- First to reach zero wins
- Wait for both threads to end

# Run Lab 3

## Race Condition

```
#include "stdafx.h"

using namespace std;

std::atomic<int> g_counter{};
std::atomic<bool> g_won{false};

void func(string name)
{
    while(g_counter > 0)
    {
        g_counter--;
        std::this_thread::sleep_for(1us);
    }
    if (!g_won)
    {
        g_won = true;
        cout << name << " wins!" << endl;
    }
}

int main()
{
    for (int run{}; run < 10; run++)
    {
        cout << "Race " << run << ": ";
        g_counter = 20000;
        g_won = false;

        thread t1(func, "Thread 1");
        thread t2(func, "Thread 2");

        t1.join();
        t2.join();
    }
    return 0;
}
```

- Two simultaneous threads
- Decrement a global counter
- First to reach zero wins
- Result is non-deterministic
- This is a *Race Condition*

```
race-condition
File Edit View Terminal Tabs Help
Race 0: Thread 2 wins!
Race 1: Thread 1 wins!
Race 2: Thread 1 wins!
Race 3: Thread 1 wins!
Race 4: Thread 1 wins!
Race 5: Thread 2 wins!
Race 6: Thread 1 wins!
Race 7: Thread 1 wins!
Race 8: Thread 1 wins!
Race 9: Thread 1 wins!

Process returned 0 (0x0)   execution time : 8.866 s
Press ENTER to continue.
```



## Open Lab 4 : Non-Atomic Operations

1. Implement a C++ multithreaded console program that performs **10** runs of the same experiment
2. Each run of the experiment should first reset a **global counter** to **zero**, and then launch **20** threads
3. Each thread should single increment (++) the global counter **50,000** times
4. Once all **20** threads complete, the program should display the global counter value for that particular run
5. With **20** threads incrementing the global counter **50,000** times, the final value for each run *should be* **1,000,000**
6. Once all **10** runs complete, the program should end

# Open Lab 4 : Non-Atomic Operations

```
14 void func()  
15 {  
16     //g_console_mutex.lock();  
17     for (int i{}; i < 50000; i++) {  
18         g_counter++;  
19     }  
20     //g_console_mutex.unlock();  
21 }
```

Increment  
the global  
counter  
50,000 times

```
23 int main()  
24 {  
25     cout.imbue(std::locale(""));  
26  
27     for (int i{}; i < 10; ++i) {  
28         g_counter = 0;  
29         vector<thread> threadPool;  
30         for (size_t t{}; t < 20; ++t)  
31             threadPool.push_back(  
32                 thread(func));  
33         for (auto& t : threadPool)  
34             t.join();  
35         cout << "Run " << i << ": "  
36             << "Counter = " << (int)g_counter  
37             << endl;  
38     }  
39     return 0;  
40 }  
41 }
```

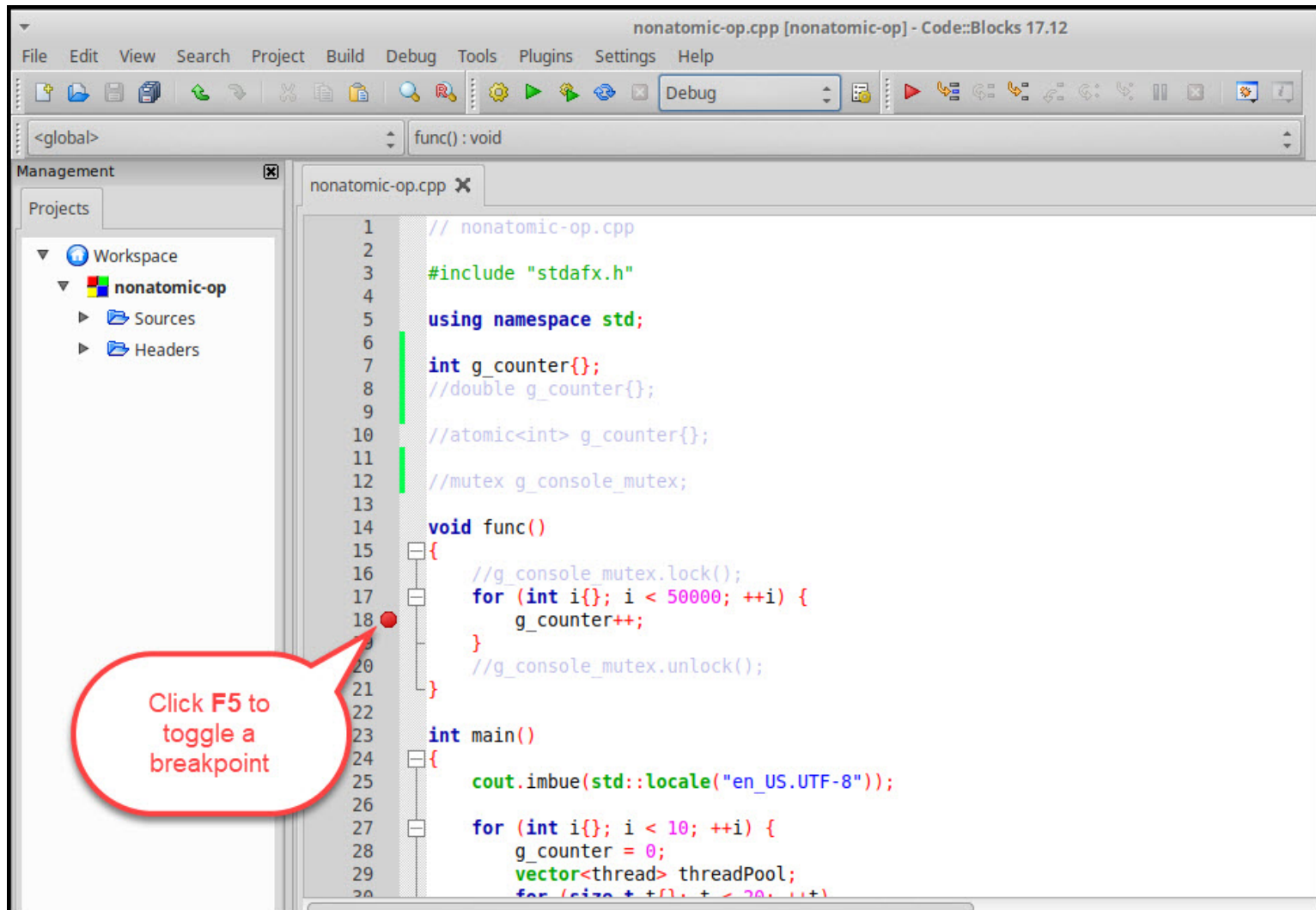
10 runs

20 threads

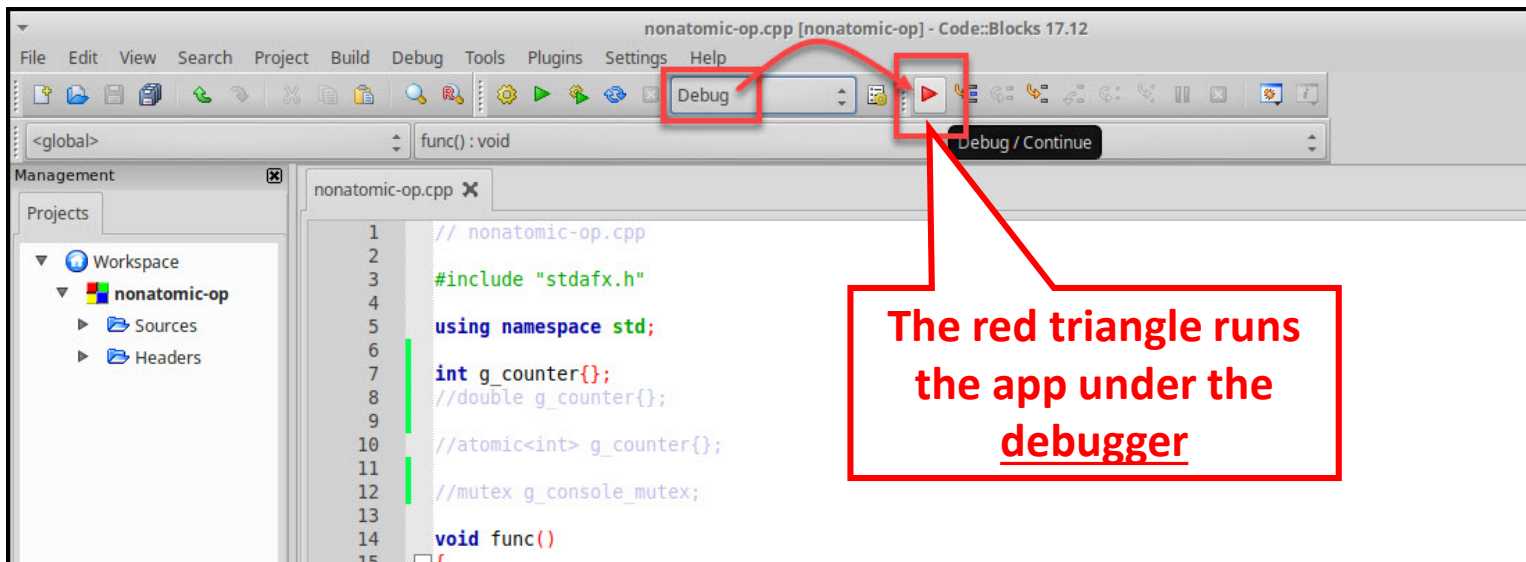
Wait for all  
threads to end

Display the global  
counter value

# Edit Lab 4 : Non-Atomic Operations

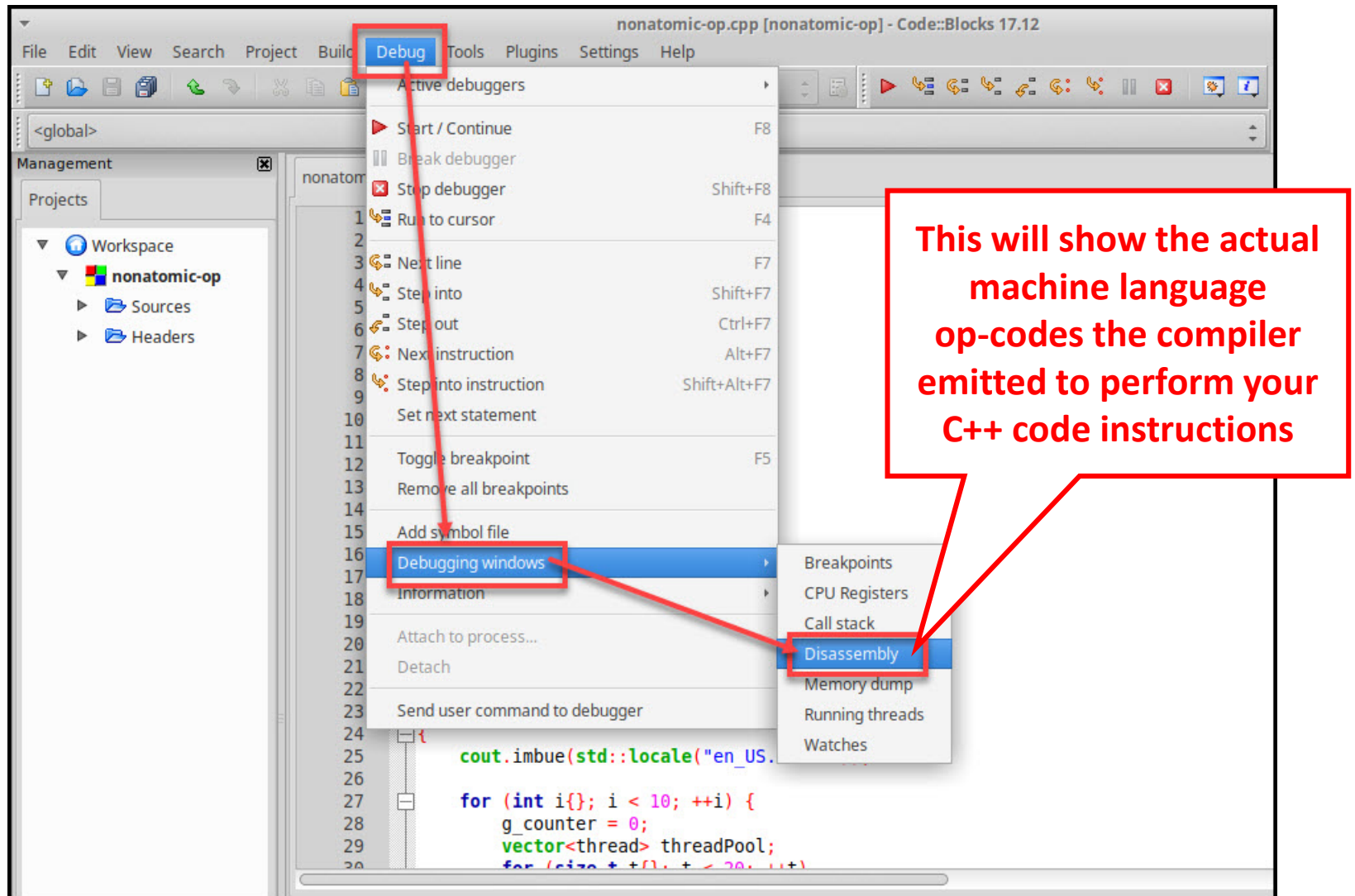


# Run Lab 4 : Non-Atomic Operations





# Run Lab 4 : Non-Atomic Operations



# Run Lab 4 : Non-Atomic Operations

The image shows a C++ IDE with a source code window on the left and a disassembly window on the right. A red callout box points to a C++ code snippet, highlighting a non-atomic increment operation. The disassembly window shows the corresponding assembly instructions for this operation.

**Source Code (C++):**

```
1 //mutex g_console_mutex;
2
3
4
5
6
7
8
9
10 //atomic_int g_counter;
11
12 //mutex g_console_mutex;
13
14 void func()
15 {
16     //g_console_mutex.lock();
17     for (int i{}; i < 50000; ++i) {
18         g_counter++;
19     }
20     //g_console_mutex.unlock();
21 }
22
23 int main()
24 {
25     cout.imbue(std::locale("en_US.UTF-8"));
26
27     for (int i{}; i < 10; ++i) {
28         g_counter = 0;
29         vector<thread> threadPool;
30         for (size_t t{}; t < 20; ++t) {
```

**Disassembly (Assembly):**

```
Function:
Frame start: 0x7fffe984d8c0
0x4011e0    push    rbp
0x4011e1    mov     rbp, rsp
;16 :      //g_console_mutex.lock();
;17 :      for (int i{}; i < 50000; ++i) {
0x4011e4    mov     DWORD PTR [rbp-0x4], 0x0
0x4011eb    cmp     DWORD PTR [rbp-0x4], 0xc350
0x4011f2    jge     0x401217 <func()+55>
0x401209    mov     eax, DWORD PTR [rbp-0x4]
0x40120c    add     eax, 0x1
0x40120f    mov     DWORD PTR [rbp-0x4], eax
0x401212    jmp     0x4011eb <func()+11>
;18 :      g_counter++;
0x4011f8    mov     eax, DWORD PTR ds:0x605218
0x4011ff    add     eax, 0x1
0x401202    mov     DWORD PTR ds:0x605218, eax
;19 :      }
;20 :      //g_console_mutex.unlock();
;21 :      }
0x401217    pop     rbp
0x401218    ret
```

**Callout Text:**

That one ++ statement in C++ required three Intel CPU machine language op-codes to implement

# Run Lab 4 : Non-Atomic Operations

The screenshot shows a C++ IDE with a file named `nonatomic-op.cpp`. The code defines a global counter `g_counter` and a function `func()` that increments it. The `main` function runs `func()` 10 times in parallel using a thread pool. The terminal output shows the counter values for each run, which are not 1,000,000, demonstrating non-atomicity.

```
1 // nonatomic-op.cpp
2
3 #include "stdafx.h"
4
5 using namespace std;
6
7 int g_counter{};
8 //double g_counter{};
9
10 //atomic<int> g_counter{};
11
12 //mutex g_console_mutex;
13
14 void func()
15 {
16     //g_console_mutex.lock();
17     for (int i{}; i < 50000; ++i)
18         g_counter++;
19 }
20 //g_console_mutex.unlock();
21
22
23 int main()
24 {
25     cout.imbue(std::locale("en_US.
26
27     for (int i{}; i < 10; ++i) {
28         g_counter = 0;
29         vector<thread> threadPool;
30         for (size_t t{}; t < 20; ++t) {
```

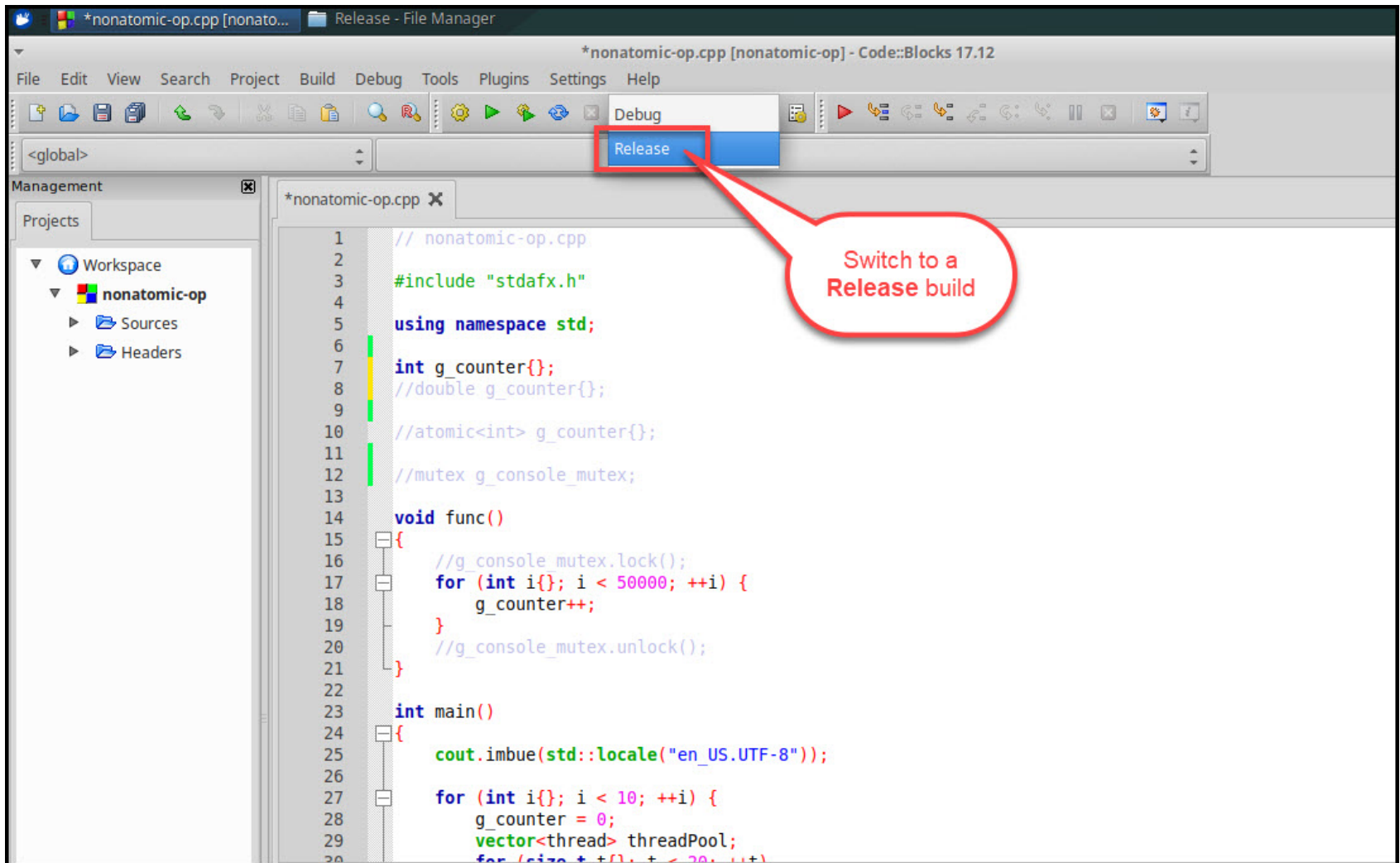
Preemptive multithreading context switches can happen only after a CPU instruction, but still possibly *within* a single C++ statement!

With 20 threads incrementing the global counter 50,000 times, the final value for each run should be 1,000,000

Run 0: Counter = 1,000,000  
Run 1: Counter = 1,000,000  
Run 2: Counter = 869,648  
Run 3: Counter = 608,145  
Run 4: Counter = 838,832  
Run 5: Counter = 732,286  
Run 6: Counter = 700,825  
Run 7: Counter = 704,906  
Run 8: Counter = 696,864  
Run 9: Counter = 804,493

Process returned 0 (0x0) execution time : 0.072 s  
Press ENTER to continue.

# Edit Lab 4 : Non-Atomic Operations



# Run Lab 4 : Non-Atomic Operations

The image shows a C++ IDE with a file named `nonatomic-op.cpp`. The code includes a `for` loop in the `func()` function that increments `g_counter` 50,000 times. A red box highlights the line `g_counter++;` at line 18. A red arrow points from this line to a disassembly window. The disassembly window shows the instruction `add DWORD PTR [rip+0x20118e],0xc350` at address `0x401060`. A red box highlights this instruction. A red arrow points from the disassembly window back to the `for` loop in the code. A red box highlights the `for` loop in the code. A red arrow points from the `for` loop to the disassembly window. A red box highlights the `for` loop in the code. A red arrow points from the `for` loop to the disassembly window.

**A release build optimizes your code for speed, so the compiler figured out a way to implement your entire function in just one CPU instruction!**

**0xc350 in hexadecimal is 50,000 so the compiler wrote the `for()` loop to just directly add 50,000 to `g_counter` in one CPU instruction!**

```
1 // non
2
3 #inclu
4
5 using
6
7 int g
8 //doub
9
10 //atomic<int> g_counter{};
11
12 //mutex g_console_mutex;
13
14 void func()
15 {
16     //g_console_mutex.lock();
17     for (int i{}; i < 50000; ++i) {
18         g_counter++;
19     }
20     //g_console_mutex.unlock();
21 }
22
23 int main()
24 {
25     cout.imbue(std::locale("en_US.UTF-8"));
26
27     for (int i{}; i < 10; ++i) {
28         g_counter = 0;
29         vector<thread> threadPool;
30         for (size_t t{}; t < 20; ++t) {
```

0x7ffe984d960

```
;15 :
;16 : //g_console_mutex.lock();
;17 : for (int i{}; i < 50000; ++i) {
;18 :     g_counter++;
;19 : }
;20 : //g_console_mutex.unlock();
;21 : }
0x40106a    ret
```

Mixed Mode Adjust Save to text file



# Run Lab 4 : Non-Atomic Operations

```
1 // nonatomic-op.cpp
2
3 #include "stdafx.h"
4
5 using namespace std;
6
7 int g_counter{};
8 //double g_counter{};
9
10 //atomic<int> g_counter{};
11
12 //mutex g_console_mutex;
13
14 void func()
15 {
16     //g_console_mutex.lock();
17     for (int i{}; i < 50000; ++i)
18         g_counter++;
19 }
20 //g_console_mutex.unlock();
21
22
23 int main()
24 {
25     cout.imbue(std::locale("en_US.UTF-8"));
26
27     for (int i{}; i < 10; ++i) {
28         g_counter = 0;
29         vector<thread> threadPool;
30         for (size_t t{}; t < 20; ++t) {
```

Run 0: Counter = 1,000,000  
Run 1: Counter = 1,000,000  
Run 2: Counter = 1,000,000  
Run 3: Counter = 1,000,000  
Run 4: Counter = 1,000,000  
Run 5: Counter = 1,000,000  
Run 6: Counter = 1,000,000  
Run 7: Counter = 1,000,000  
Run 8: Counter = 1,000,000  
Run 9: Counter = 1,000,000

Process returned 0 (0x0) execution time : 0.096 s  
Press ENTER to continue.

**Because the Release build needed only one CPU instruction for the entire loop, pre-emptive multithreading can no longer interrupt each thread's function**



# Edit Lab 4 : Non-Atomic Operations

Change **g\_counter** to be of type **double** (floating point) versus type **int**

```
4  using namespace std;
5
6
7  //g_counter{};
8  double g_counter{};
9
10 //atomic<int> g_counter{};
11
12 //mutex g_console_mutex;
13
14 void func()
15 {
16     //g_console_mutex.lock();
17     for (int i{}; i < 50000; ++i) {
18         g_counter++;
19     }
20     //g_console_mutex.unlock();
21 }
22
23
24
25
26
27
28
29
30
```

Disassembly

Function:

Frame start: 0x7ffe984d960

```
0x4010a8    add     eax,0xffffffff6
0x4010ab    jne     0x401080 <func()+32>
;18 :      g_counter++;
0x401065    movsd   xmm0,QWORD PTR [rip+0x202193]
0x40106d    movsd   xmm1,QWORD PTR [rip+0x8d3]
0x401075    nop     WORD PTR cs:[rax+rax*1+0x0]
0x40107f    nop
0x401080    addsd   xmm0,xmm1
0x401084    addsd   xmm0,xmm1
0x401088    addsd   xmm0,xmm1
0x40108c    addsd   xmm0,xmm1
0x401090    addsd   xmm0,xmm1
0x401094    addsd   xmm0,xmm1
0x401098    addsd   xmm0,xmm1
0x40109c    addsd   xmm0,xmm1
0x4010a0    addsd   xmm0,xmm1
0x4010a4    addsd   xmm0,xmm1
0x4010ad    movsd   QWORD PTR [rip+0x20214b],xmm0
;19 :      }
;20 :      //g_console_mutex.unlock();
;21 :  }
0x4010b5    ret
```

It takes many more CPU instructions to implement floating point mathematics, even just a simple **++** operator

# Run Lab 4 : Non-Atomic Operations

The screenshot shows the Visual Studio IDE with a C++ file named `nonatomic-op.cpp`. The code defines a global variable `g_counter` of type `double` and a function `func()` that increments it in a loop. The `Release` build configuration is selected in the toolbar. A terminal window titled `nonatomic-op` displays the output of the program, showing 10 runs with counter values ranging from 500,000 to 900,000. A red callout box points to the `double g_counter{};` line in the code and the terminal output, explaining that the compiler cannot optimize the `++` operator on a floating-point variable to a single CPU instruction, which causes the `Release` build to fail.

```
1 // nonatomic-op.cpp
2
3 #include "stdafx.h"
4
5 using namespace std;
6
7 //int g_counter{};
8 double g_counter{};
9
10 //atomic<int> g_counter{};
11
12 //mutex g_console_mutex;
13
14 void func()
15 {
16     //g_console_mutex.lock();
17     for (int i{}; i < 50000; ++i) {
18         g_counter++;
19     }
20     //g_console_mutex.unlock();
21 }
22
23 int main()
24 {
25     cout.imbue(std::locale("en_US.UTF-8"));
26
27     for (int i{}; i < 10; ++i) {
28         g_counter = 0;
29         vector<thread> threadPool;
30         for (size_t j{}; j < 20; ++j) {
```

nonatomic-op

Run 0: Counter = 500,000  
Run 1: Counter = 500,000  
Run 2: Counter = 650,000  
Run 3: Counter = 450,000  
Run 4: Counter = 550,000  
Run 5: Counter = 500,000  
Run 6: Counter = 650,000  
Run 7: Counter = 650,000  
Run 8: Counter = 900,000  
Run 9: Counter = 800,000

Process returned 0 (0x0) execution time: 0.2 s  
Press ENTER to continue.

Now even the **Release** build fails, because the compiler cannot optimize the **++** operator on a *floating point* variable to just a single CPU machine instruction

# Edit Lab 4 : Non-Atomic Operations

Change to using  
`atomic<int>` for `g_counter`

```
3 //stdafx.h
4
5 namespace std;
6
7 //int g_counter{};
8 //double g_counter{};
9
10 atomic<int> g_counter{};
11
12 //mutex g_console_mutex;
13
14 void c()
15 {
16     console_mutex.lock();
17     for (int i{}; i < 50000; i++)
18         counter ++;
19
20     mutex.unlock();
```

In C++ the `atomic<>` class has built-in support to ensure updates to a variable are fully completed *before* any thread context switch can occur

# Run Lab 4 : Non-Atomic Operations

```
1 // nonatomic-op.cpp
2
3 #include "stdafx.h"
4
5 using namespace std;
6
7 //int g_counter{};
8 //double g_counter{};
9
10 atomic<int> g_counter{};
11
12 //mutex g_console_mutex;
13
14 void func()
15 {
16     //g_console_mutex.lock();
17     for (int i{}; i < 50000; i++) {
18         g_counter ++;
19     }
20     //g_console_mutex.unlock();
21 }
22
23 int main()
24 {
25     cout.imbue(std::locale("en_US.UTF-8"));
26
27     for (int i{}; i < 10; ++i) {
28         g_counter = 0;
29         vector<thread> threadPool;
30         for (size_t t{}; t < 20; ++t) {
```

Run 0: Counter = 1,000,000  
Run 1: Counter = 1,000,000  
Run 2: Counter = 1,000,000  
Run 3: Counter = 1,000,000  
Run 4: Counter = 1,000,000  
Run 5: Counter = 1,000,000  
Run 6: Counter = 1,000,000  
Run 7: Counter = 1,000,000  
Run 8: Counter = 1,000,000  
Run 9: Counter = 1,000,000

Process returned 0 (0x0) execution time : 0.283 s  
Press ENTER to continue.

Notice the total run time when using the `atomic<>` class to synchronize different threads so they don't all try to update `g_counter` at the same time

# Edit Lab 4 : Non-Atomic Operations

```
nonatomic-op.cpp X
1 // nonatomic-op.cpp
2
3 #include "stdafx.h"
4
5 using namespace std;
6
7 int g_counter{};
8 //double g_counter{};
9
10 //atomic<int> g_counter{};
11
12 mutex g_console_mutex;
13
14 void func()
15 {
16     g_console_mutex.lock();
17     for (int i{}; i < 50000;
18         g_counter++;
19     }
20     g_console_mutex.unlock();
21 }
22
23 int main()
24 {
25     cout.imbue(std::locale("e
26
27     for (int i{}; i < 10; ++i
28         g_counter = 0;
29     vector<thread> thread
30     for (size_t t{}; t <
```

Switch back to a regular **int**  
type for **g\_counter**

Declare a **mutex**

Implement the **mutex lock()**  
and **unlock()** to guard against  
simultaneous edits to  
**g\_counter** by other threads

# Run Lab 4 : Non-Atomic Operations

The image shows a C++ source file named `nonatomic-op.cpp` and its execution output in a terminal window. The code defines a global counter `g_counter` and a mutex `g_console_mutex`. A function `func()` increments the counter 50,000 times while holding the mutex. The `main` function runs `func()` 10 times in parallel using a thread pool. The output shows that the counter value is consistently 1,000,000 across 10 runs, and the execution time is 0.084 seconds.

```
1 // nonatomic-op.cpp
2
3 #include "stdafx.h"
4
5 using namespace std;
6
7 int g_counter{};
8 //double g_counter{};
9
10 //atomic<int> g_counter{};
11
12 mutex g_console_mutex;
13
14 void func()
15 {
16     g_console_mutex.lock();
17     for (int i{}; i < 50000; i++) {
18         g_counter++;
19     }
20     g_console_mutex.unlock();
21 }
22
23 int main()
24 {
25     cout.imbue(std::locale("en_US.UTF-8"));
26
27     for (int i{}; i < 10; ++i) {
28         g_counter = 0;
29         vector<thread> threadPool;
30         for (size_t t{}; t < 20; ++t) {
```

nonatomic-op

Run 0: Counter = 1,000,000  
Run 1: Counter = 1,000,000  
Run 2: Counter = 1,000,000  
Run 3: Counter = 1,000,000  
Run 4: Counter = 1,000,000  
Run 5: Counter = 1,000,000  
Run 6: Counter = 1,000,000  
Run 7: Counter = 1,000,000  
Run 8: Counter = 1,000,000  
Run 9: Counter = 1,000,000

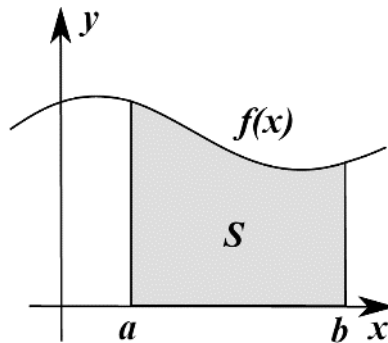
Process returned 0 (0x0) execution time 0.084 s  
Press ENTER to continue.

In C++ on Linux, using a mutex is 5X faster than using `atomic<>` but it does increase code complexity



# Why do we need integrals?

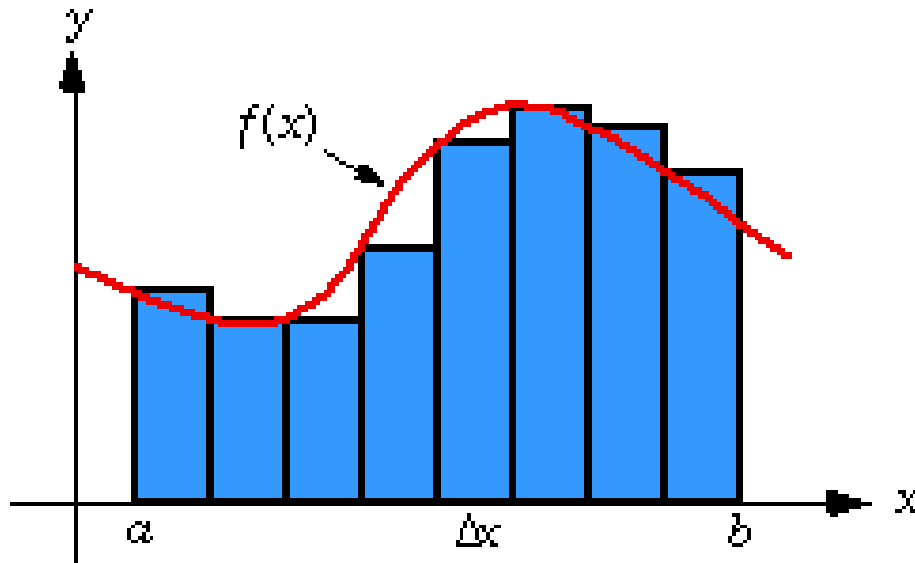
- The **integral** of a function can be defined as the **area under a curve**  $f(x)$  within the region  $[a,b]$



- There are ways to often determine exactly the value of the integral of  $f(x)$  which we would write  $F(x) = \int_a^b f(x)$
- However, sometimes it is not possible to find an analytic expression for  $F(x)$  – so we use **numerical integration**

# Riemann Sums

- One way we can integrate  $f(x)$  is to divide the area under the curve into strips (**intervals**) and sum the area of each strip
- This estimate may not be totally accurate because we might have **gaps** between the true value of  $f(x)$  and the top of a strip

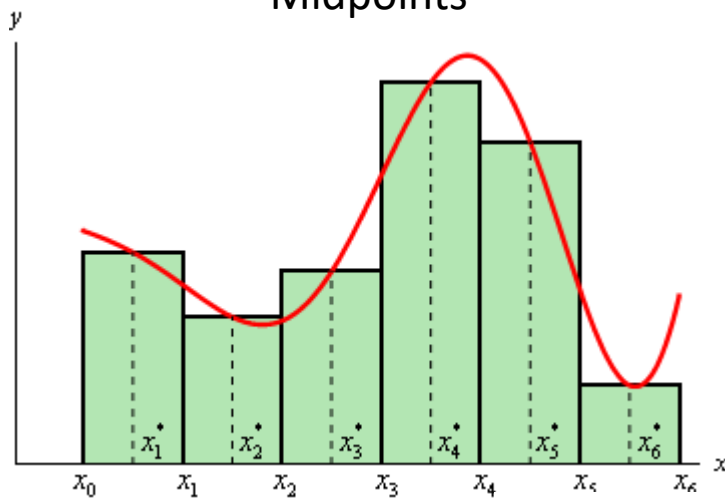


# Riemann Sums

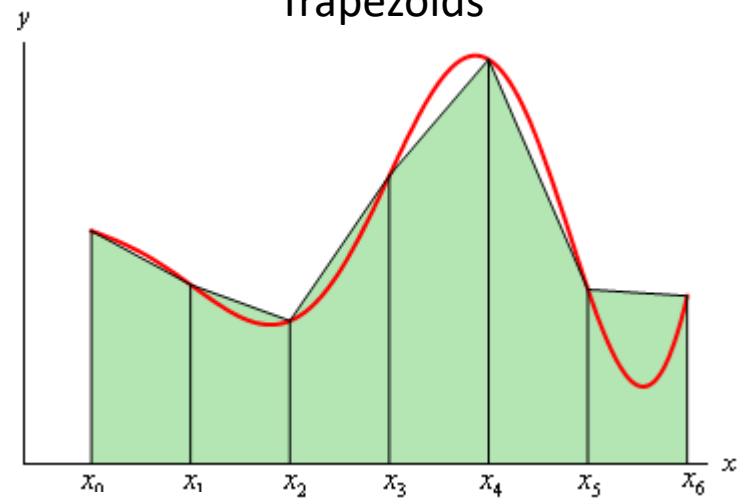
- The width of each strip is  $\Delta x = \frac{(b-a)}{\# \text{ of intervals}}$
- We can minimize the gaps by increasing the number of intervals, which makes the  $\Delta x$  **smaller**
- There are different strategies for determine the shape and height of each strip
  - Left-hand Rule, Right-hand Rule, Midpoint Rule
  - Fit Trapezoids
  - Fit Parabolas (Simpson's Rule)
- Depending upon the particular shape of  $f(x)$ , one method might be more accurate than the others

# Riemann Sums

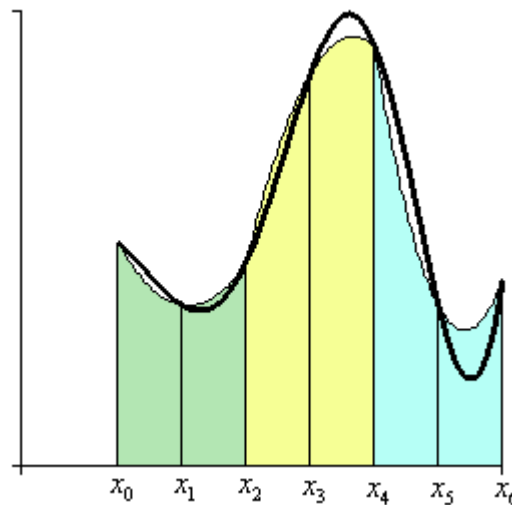
Midpoints



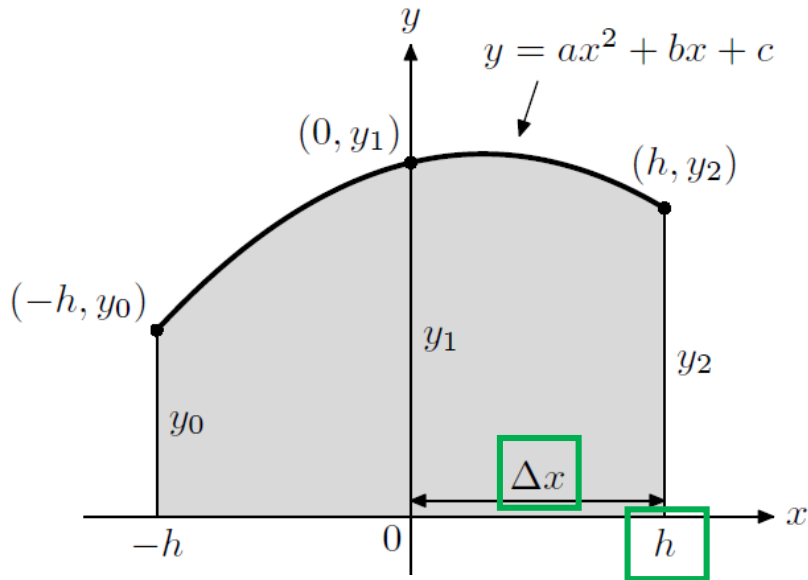
Trapezoids



Parabolas  
(Simpson's Rule)



# Simpson's Rule is more accurate!



$$y_0 = ah^2 - bh + c$$

$$y_1 = c$$

$$y_2 = ah^2 + bh + c$$

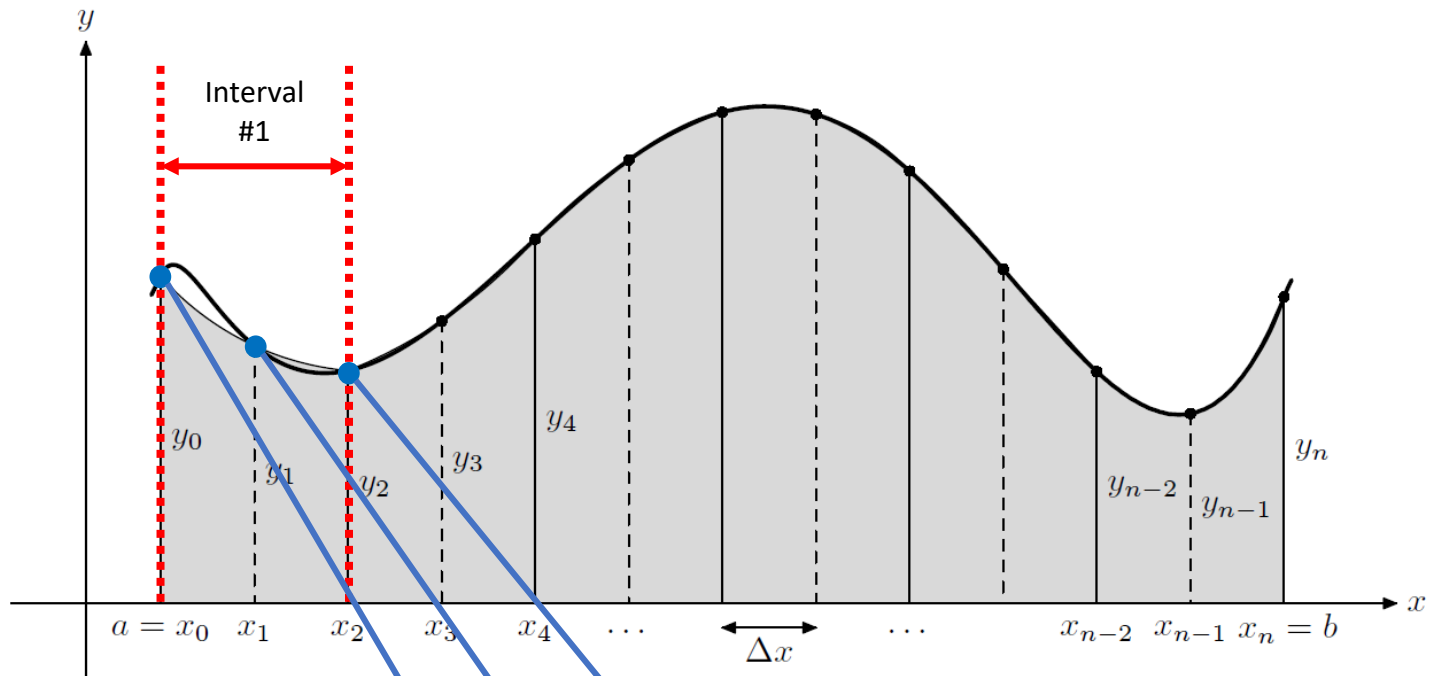
$$y_0 + 4y_1 + y_2 = (ah^2 - bh + c) + 4c + (ah^2 + bh + c) = 2ah^2 + 6c$$

$$\begin{aligned} A &= \int_{-h}^h (ax^2 + bx + c) dx \\ &= \left( \frac{ax^3}{3} + \frac{bx^2}{2} + cx \right) \Big|_{-h}^h \\ &= \frac{2ah^3}{3} + 2ch \\ &= \frac{h}{3} (2ah^2 + 6c) \end{aligned}$$

$$A = \frac{h}{3} (y_0 + 4y_1 + y_2) = \frac{\Delta x}{3} (y_0 + 4y_1 + y_2)$$

# Simpson's Rule is more accurate!

$$y_0 = f(x_0), \quad y_1 = f(x_1), \quad y_2 = f(x_2), \quad \dots, \quad y_n = f(x_n).$$

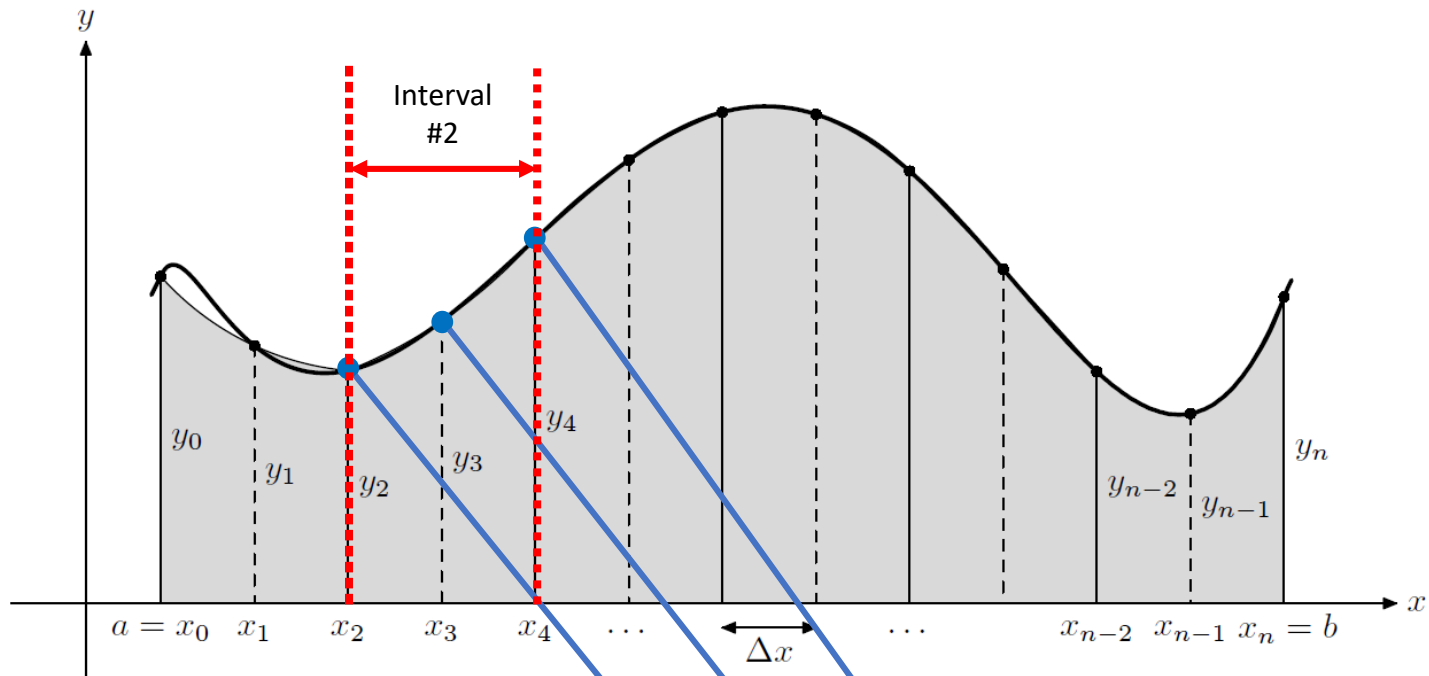


$$\int_a^b f(x) dx \approx \frac{\Delta x}{3} (y_0 + 4y_1 + 1y_2 + \dots)$$



# Simpson's Rule is more accurate!

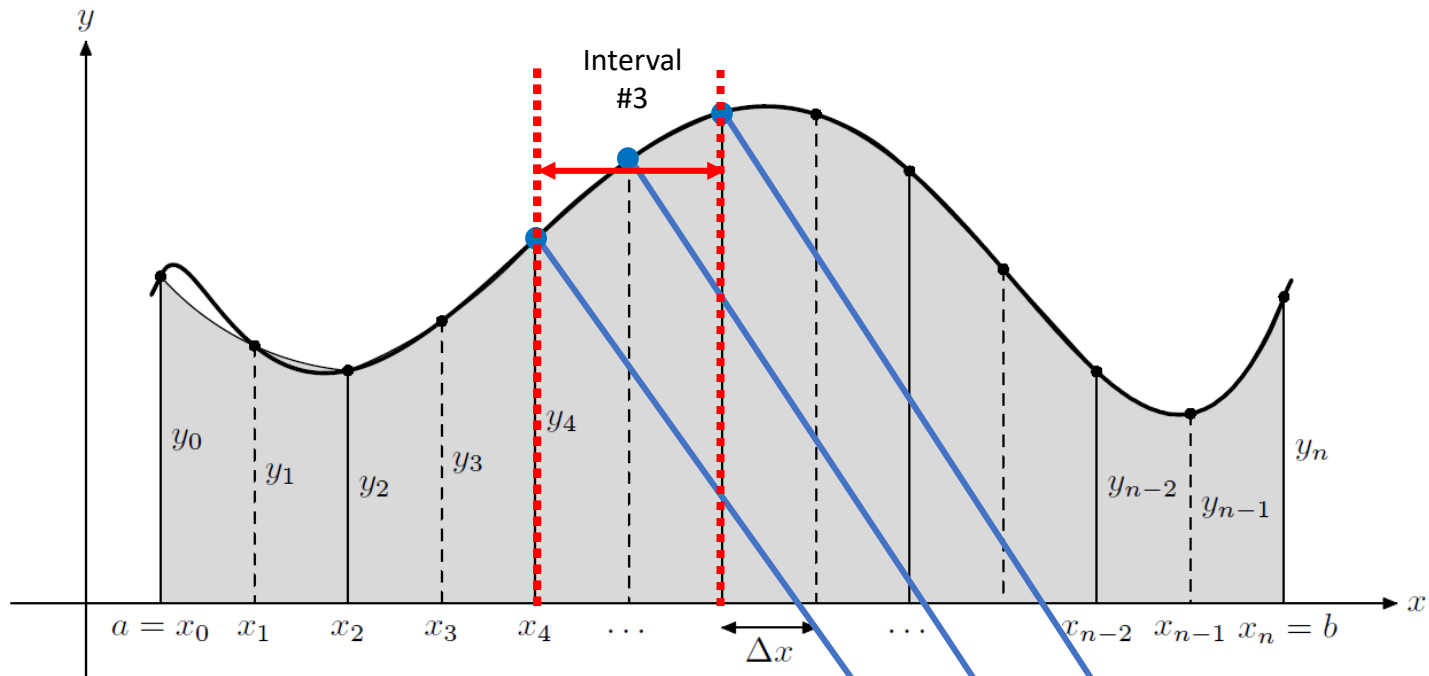
$$y_0 = f(x_0), \quad y_1 = f(x_1), \quad y_2 = f(x_2), \quad \dots, \quad y_n = f(x_n).$$



$$\int_a^b f(x) dx \approx \frac{\Delta x}{3} (y_0 + 4y_1 + \boxed{2y_2} + \boxed{4y_3} + \boxed{1y_4} \dots)$$

# Simpson's Rule is more accurate!

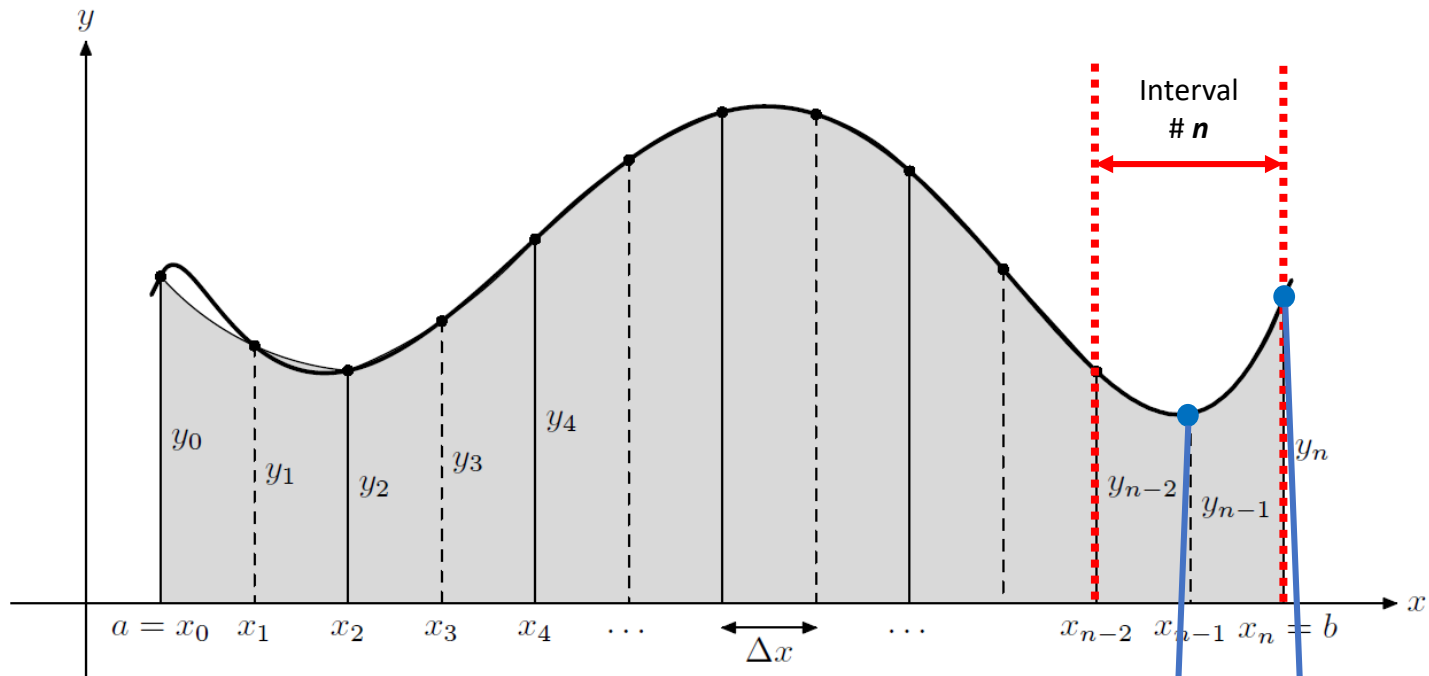
$$y_0 = f(x_0), \quad y_1 = f(x_1), \quad y_2 = f(x_2), \quad \dots, \quad y_n = f(x_n).$$



$$\int_a^b f(x) dx \approx \frac{\Delta x}{3} (y_0 + 4y_1 + 2y_2 + 4y_3 + \boxed{2y_4} + \boxed{4y_5} + \boxed{1y_6} + \dots)$$

# Simpson's Rule is more accurate!

$$y_0 = f(x_0), \quad y_1 = f(x_1), \quad y_2 = f(x_2), \quad \dots, \quad y_n = f(x_n).$$



$$\int_a^b f(x) dx \approx \frac{\Delta x}{3} (y_0 + 4y_1 + 2y_2 + 4y_3 + 2y_4 + \dots + \boxed{4y_{n-1}} + \boxed{y_n})$$

# Simpson's Rule is more accurate!

$$\int_a^b f(x) dx \approx \frac{\Delta x}{3} (y_0 + 4y_1 + 2y_2 + 4y_3 + 2y_4 + \cdots + 4y_{n-1} + y_n)$$

| Point | y0 | y1 | y2 | y3 | y4 | y5 | y6 |
|-------|----|----|----|----|----|----|----|
| Coeff | 1  | 4  | 2  | 4  | 2  | 4  | 1  |

The first and last point have coefficient = 1

Every point with an odd index has coefficient = 4

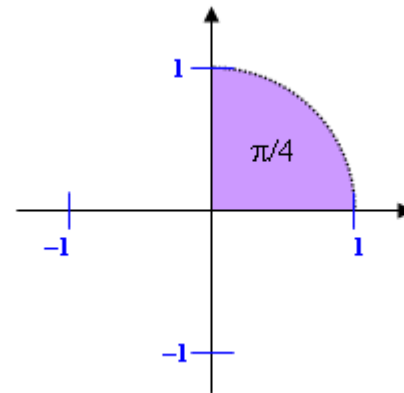
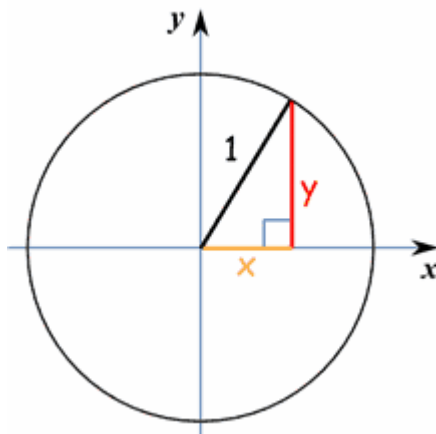
Every point with an even index has coefficient = 2

```
void simpsons(TCB* tcb)
{
    double sum = f(tcb->b) + f(tcb->a);
    double deltaX = (tcb->b - tcb->a) / tcb->intervals;
    tcb->a += deltaX;
    for (int i=1; i < tcb->intervals; i++) {
        int coeff = 2 * (i % 2 + 1);
        double y=f(tcb->a);
        if (!isnan(y)) sum += coeff*y;
        tcb->a += deltaX;
    }
    tcb->integral = (deltaX / 3) * sum;
}
```

## Open Lab 5 – Parallel Simpson's Rule

Numerically calculate to **12** significant digits the value of **4** × **the area of a unit circle** in the **first quadrant** using Simpson's Rule

$$F(x) = 4 \int_0^1 \sqrt{1 - x^2} dx = \pi$$



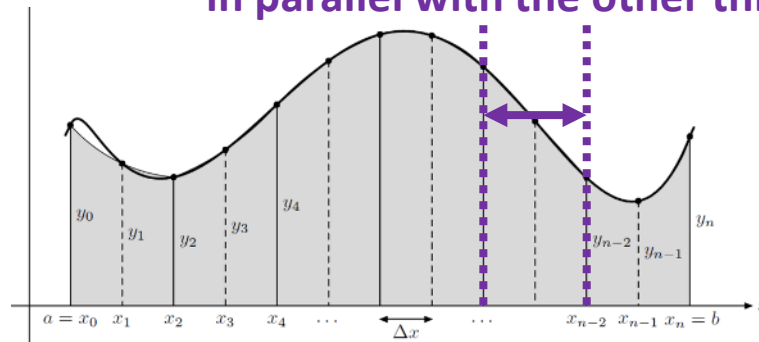
# Open Lab 5 – Parallel Simpson's Rule

```
struct TCB {  
    std::thread* t = nullptr;  
    double a{};  
    double b{};  
    long intervals{};  
    double integral{};  
};
```

- Every thread has its own thread control block (TCB)
- Each **TCB** stores all the information its thread needs to compute **its portion** of the integral

```
inline double f(double x)  
{  
    return sqrt(1 - pow(x, 2));  
}  
  
void simpsons(TCB* tcb)  
{  
    double sum = f(tcb->b) + f(tcb->a);  
    double deltaX = (tcb->b - tcb->a) / tcb->intervals;  
    tcb->a += deltaX;  
    for (int i=1; i < tcb->intervals; i++) {  
        int coeff = 2 * (i % 2 + 1);  
        double y=f(tcb->a);  
        if (!isnan(y)) sum += coeff*y;  
        tcb->a += deltaX;  
    }  
    tcb->integral = (deltaX / 3) * sum;  
}
```

Each thread computes its own slice in parallel with the other threads



```

for (int threads{1}; threads <= max_threads * 4; threads *= 2) {
    auto startTime = chrono::steady_clock::now();

    double interval_width = (g_b - g_a) / threads;

    // Each thread gets its own
    // thread control block (TCB)
    vector<TCB*> threadPool;

    for (int i{}; i < threads; i++) {
        TCB* tcb = new TCB();
        tcb->a = i * interval_width;
        tcb->b = tcb->a + interval_width;
        tcb->intervals = 134217728 / threads; //2^27
        tcb->t = new std::thread(simpsons, tcb);
        // New thread is now running, so add its TCB
        threadPool.push_back(tcb);
    }

    double integral{};
    for (auto& tcb : threadPool)
    {
        // Wait here for this thread to finish
        tcb->t->join();
        // Now we read this thread's result
        integral += tcb->integral;
        delete tcb->t;
    }
    threadPool.clear();

    auto endTime = chrono::steady_clock::now();

    auto totalTime = chrono::duration_cast<chrono::milliseconds>
        (endTime - startTime).count();

    cout << setw(15) << fixed << setprecision(11)
        << integral * 4.0
        << setw(10) << threads
        << setw(10) << fixed << setprecision(3)
        << totalTime << "(ms)" << endl;
}

```

```
int max_threads = thread::hardware_concurrency();
```

$$4 \int_{a=0}^{b=1} \sqrt{1-x^2} dx$$

Given  $n$  = # of threads:

- Divide entire integration region  $[0,1]$  into  $n$  distinct slices  $\therefore$  each thread has its own assigned  $[a, b]$  values
- To achieve accuracy goal, use  $2^{27}$  steps across the **entire** region  $[0,1]$   $\therefore$  each thread only needs to use  $\left(\frac{2^{27}}{n}\right)$  intervals across its slice



# Run Lab 5 – Parallel Simpson's Rule

This machine had 2 cores:

```
parallel-simpsons
File Edit View Terminal Tabs Help
This session can support 2 simultaneous threads

Integral  Threads  Time
3.14159265359  1  5,851(ms)
3.14159265359  2  2,949(ms)
3.14159265359  4  2,962(ms)
3.14159265359  8  2,989(ms)

Process returned 0 (0x0)  execution time : 14.780 s
Press ENTER to continue.
```

This machine had 4 cores:

```
parallel-simpsons
File Edit View Terminal Tabs Help
This session can support 4 simultaneous threads

Integral  Threads  Time
3.14159265359  1  6,048(ms)
3.14159265359  2  3,012(ms)
3.14159265359  4  2,225(ms)
3.14159265359  8  1,561(ms)
3.14159265359 16  1,502(ms)

Process returned 0 (0x0)  execution time : 14.405 s
Press ENTER to continue.
```

- Numerical integration is “embarrassingly parallel” and *scales* well according to number of threads
- However, it is generally not productive to launch more threads than **2x** the number of cores, otherwise useful CPU time is lost due to excessive **context switching**

## Now you know...

- Using **C++ Threads** enables programmers to distribute computation tasks across multiple CPU cores simultaneously, executing many of them in parallel, for a potentially **significant** speed-up over traditional *sequential* algorithms
- However, launching too many threads can actually *decrease* performance – rule of thumb is to dispatch no more threads than **2X** the number of hardware “cores” available
- Critical data operations and input/output may need to be sequenced (protected) by **mutexes** and other synchronization *primitives* (e.g. **atomic<int>**)

# Now you know...

- The **Thread Control Block** (TCB) design pattern is a common way to achieve simple *decoupled* multithreading
  - Each **TCB** can pass multiple controlling parameters to its thread, and can receive output back from its thread in a “lock free” manner
  - The **TCB** pattern fulfills the requirement that every **Scatter** phase (launch threads) has a **Gather** phase (join threads) which collects the work results from each thread before the main program ends
- The advent of many-core GPUs (NVIDIA's latest **Ampere** has **8,192 cores!!**) requires programmers to wield *parallel* algorithms to maximize infrastructure value
  - High performance scientific computing requires expertise with handling **threads** – start to learn them **now!**