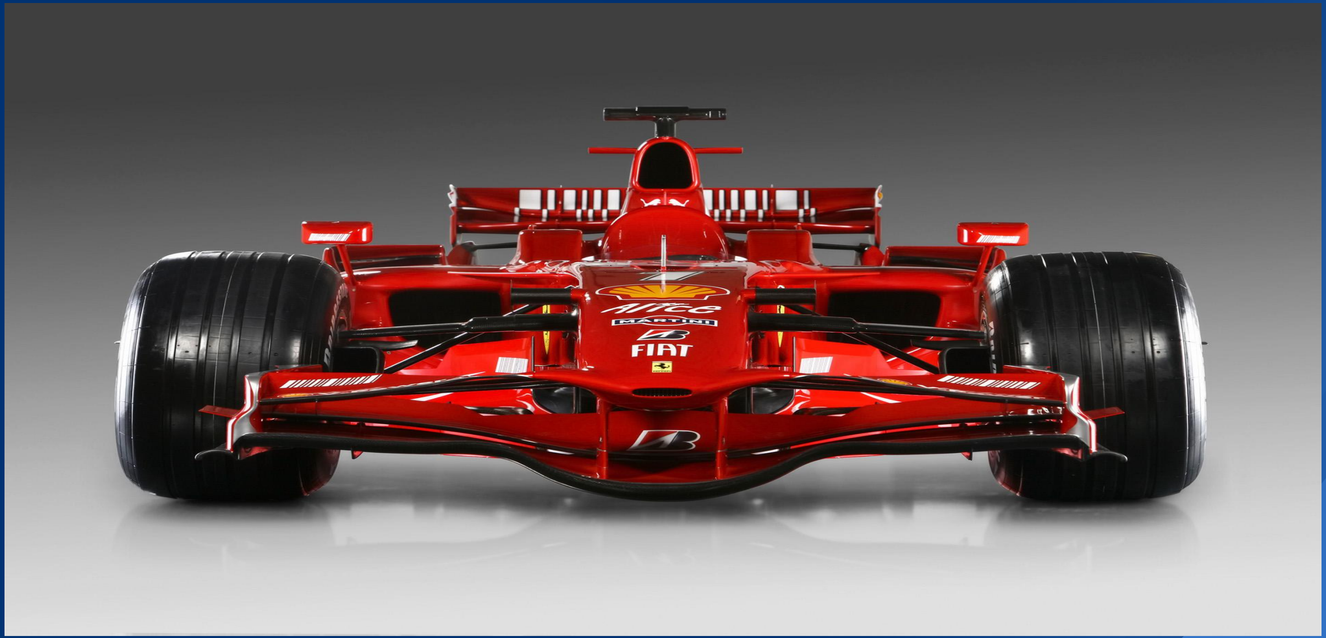
Standard Multithreading Support in C++

By: Chris DeVisser

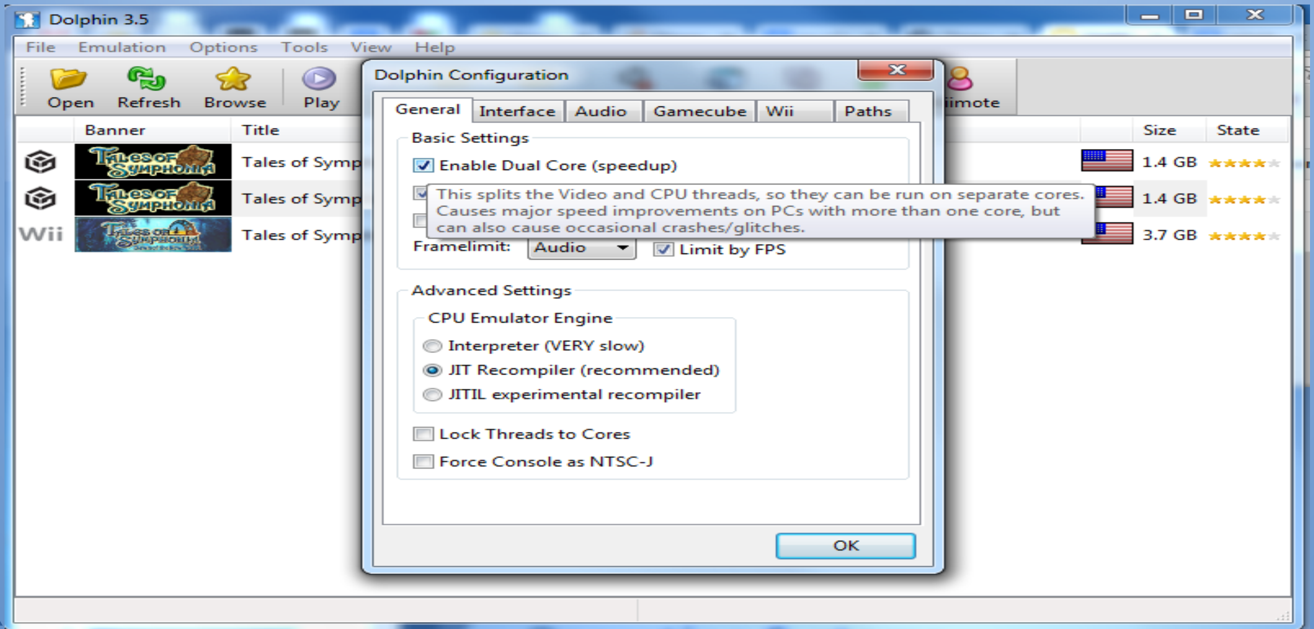


The Importance of Multithreading

- Modern processors have many cores, but often, only one is used in an application
- Significantly speeds up long operations if done correctly
- Increases application responsiveness
- We're moving into a multithreaded world

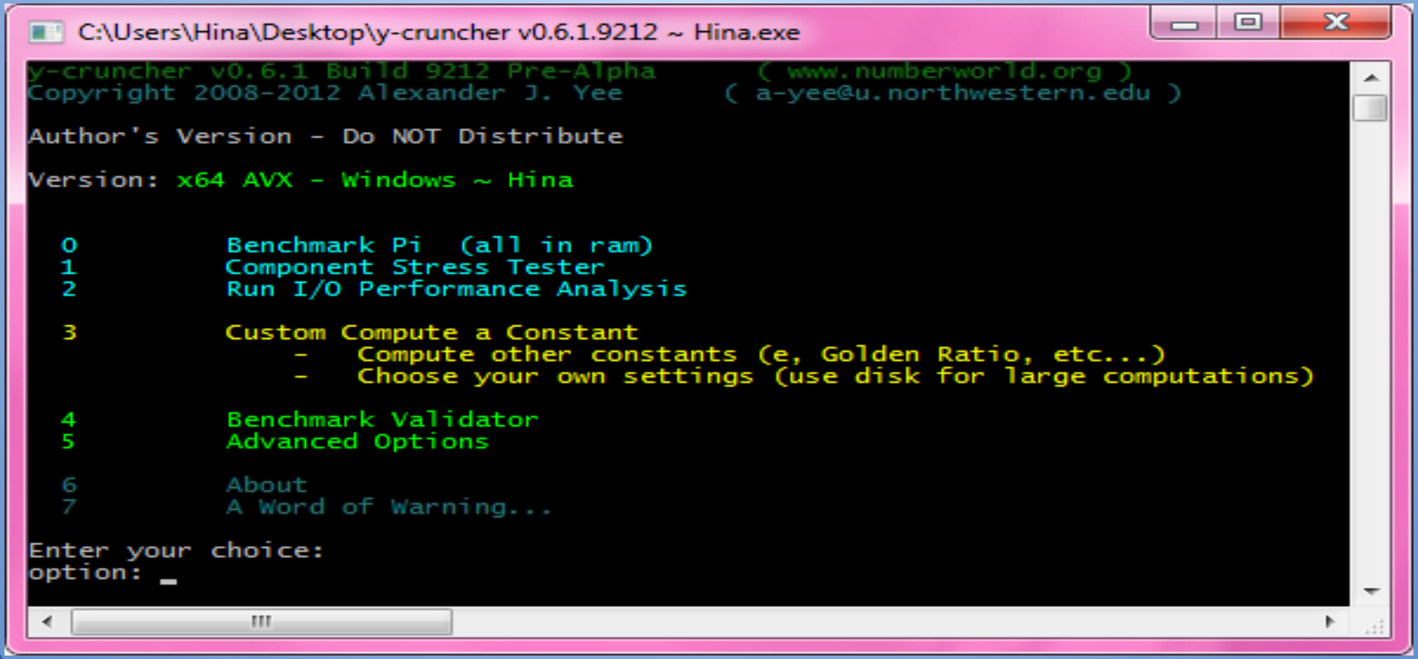
What Uses Multithreading?

- Games
- Number-crunching applications
- Operating Systems
- Most responsive applications



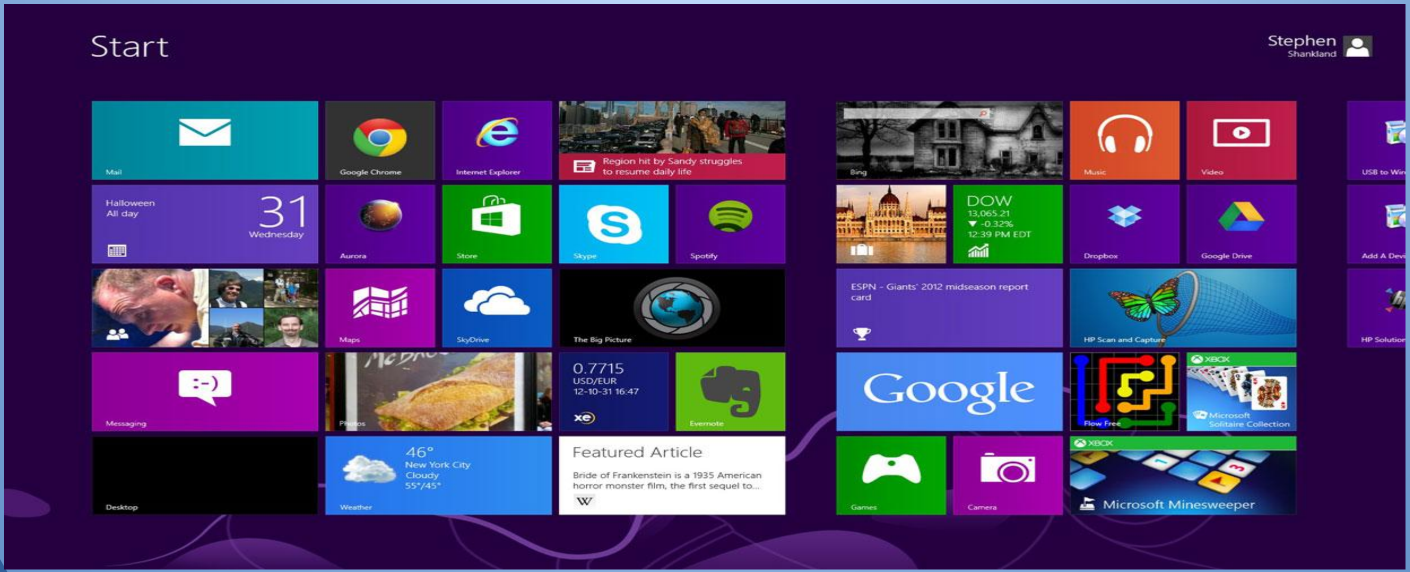
y-cruncher

- Holds the record for most digits of pi calculated, with 10 trillion
- Uses multithreading extensively



Windows 8

- "Fast and Fluid" => new policy
- All functions that could take longer than 50ms were made asynchronous
- Apps stay responsive



C++ Concurrency

- C++11 brought standardized concurrency
- Launching another thread is easy:

C++ Concurrency

- C++11 brought standardized concurrency
- Launching another thread is easy:

```
#include <thread>
int main() {
    std::thread t([]{/*stuff*/});
    t.join();
}
```

Launching Asynchronous Tasks

- std::future holds a task's status

Launching Asynchronous Tasks

- std::future holds a task's status

```
auto fut = std::async([]{/*stuff*/});
//concurrent
fut.wait();
```

Launching Asynchronous Tasks

- std::future holds a task's status

```
auto fut = std::async([]{/*stuff*/});
//concurrent
fut.wait();
```

- Be sure to assign the result of async, or the function may not launch a new thread

Preventing Half-Written Data

- Two threads use the same variable

Preventing Half-Written Data

- Two threads use the same variable

```
std::atomic<int> i;
```

Preventing Half-Written Data

- Two threads use the same variable

```
std::atomic<int> i;

//thread 1
++i;
```

Preventing Half-Written Data

- Two threads use the same variable

```
std::atomic<int> i;

//thread 1
++i;

//thread 2
std::cout << i;
```

Tip: Use RAII

- Never forget to clean up again
- Exception-safe

Tip: Use RAII

- Never forget to clean up again
- Exception-safe

```
int i = 0;
std::mutex m;
```

Tip: Use RAII

- Never forget to clean up again
- Exception-safe

```
int i = 0;
std::mutex m;

//threads
m.lock();
++i;
//oops, forgot to unlock
```

Tip: Use RAII

- Never forget to clean up again
- Exception-safe

```
int i = 0;
std::mutex m;

//threads
std::lock_guard<std::mutex> lock(m);
++i;
//m unlocked when lock goes out of scope
```

Managing Shared Resource Use

- Two threads use the same resource

Managing Shared Resource Use

- Two threads use the same resource

```
std::ofstream fout("out.txt");
std::mutex file;
```

Managing Shared Resource Use

- Two threads use the same resource

```
std::ofstream fout("out.txt");
std::mutex file;

//thread 1
std::lock_guard<std::mutex> lock(file);
fout << "thread 1\n";
```

Managing Shared Resource Use

- Two threads use the same resource

```
std::ofstream fout("out.txt");
std::mutex file;

//thread 1
std::lock_guard<std::mutex> lock(file);
fout << "thread 1\n";

//thread 2
std::lock_guard<std::mutex> lock(file);
fout << "thread 2\n";
```

Tip: Avoid Deadlocks

- Occur when two threads wait on each other

}

Tip: Avoid Deadlocks

- Occur when two threads wait on each other

//example adapted from http://pages.cs.wisc.edu/~remzi/Classes/537/Fall2005/Lectures/lecture8.pdf

```
int X = 0, Y = 0;
std::mutex x, y;
```

Tip: Avoid Deadlocks

- Occur when two threads wait on each other

//example adapted from http://pages.cs.wisc.edu/~remzi/Classes/537/Fall2005/Lectures/lecture8.pdf

```
int X = 0, Y = 0;
std::mutex x, y;

//thread 1                                   //thread 2
std::lock_guard<std::mutex> lock(x);           std::lock_guard<std::mutex> lock(y);
++X;                                         --Y;
```

Tip: Avoid Deadlocks

- Occur when two threads wait on each other

```
//example adapted from http://pages.cs.wisc.edu/~remzi/Classes/537/Fall2005/Lectures/lecture8.pdf

int X = 0, Y = 0;
std::mutex x, y;

//thread 1
std::lock_guard<std::mutex> lock(x);
++X;
{
   std::lock_guard<std::mutex> lock(y);
   ++Y;
}

//thread 2
  std::lock_guard<std::mutex> lock(y);
--Y;
{
     std::lock_guard<std::mutex> lock(x);
   --X;
}
```

Tip: Avoid Deadlocks

- Occur when two threads wait on each other

```
//example adapted from http://pages.cs.wisc.edu/~remzi/Classes/537/Fall2005/Lectures/lecture8.pdf

int X = 0, Y = 0;
std::mutex x, y;
```

```
//thread 1
std::lock_guard<std::mutex> lock(x);
++X;
{
   std::lock_guard<std::mutex> lock(y);
   ++Y;
}
```

```
//thread 2
  std::lock_guard<std::mutex> lock(y);
--Y;
{
     std::lock_guard<std::mutex> lock(x);
   --X;
}
```

Tip: Avoid Deadlocks

```
//thread 1
std::lock_guard<std::mutex> lock(x);
++X;
{
    std::lock_guard<std::mutex> lock(y);
    ++Y;
}
```

```
//thread 2
std::lock_guard<std::mutex> lock(y);
--Y;
{
    std::lock_guard<std::mutex> lock(x);
    --X;
}
```

Tip: Avoid Deadlocks

```
//thread 1
std::lock_guard<std::mutex> lock(x);
++X;
{
    std::lock_guard<std::mutex> lock(y);
    ++Y;
}
```

```
//thread 2
  std::lock_guard<std::mutex> lock(y);
--Y;
{
      std::lock_guard<std::mutex> lock(x);
    --X;
}
```

- Imagine this scenario:

Tip: Avoid Deadlocks

```
//thread 1
std::lock_guard<std::mutex> lock(x);
++X;
{
    std::lock_guard<std::mutex> lock(y);
    ++Y;
}
```

```
//thread 2
std::lock_guard<std::mutex> lock(y);
--Y;
{
    std::lock_guard<std::mutex> lock(x);
    --X;
}
```

- Imagine this scenario:
    - Thread 1 locks x

Tip: Avoid Deadlocks

```
//thread 1
std::lock_guard<std::mutex> lock(x);
++X;
{
    std::lock_guard<std::mutex> lock(y);
    ++Y;
}
```

```
//thread 2
  std::lock_guard<std::mutex> lock(y);
--Y;
{
    std::lock_guard<std::mutex> lock(x);
   --X;
}
```

- Imagine this scenario:
    - Thread 1 locks x
    - Thread 2 locks y

Tip: Avoid Deadlocks

```
//thread 1
std::lock_guard<std::mutex> lock(x);
++X;
{
   std::lock_guard<std::mutex> lock(y);
   ++Y;
}
```

```
//thread 2
 std::lock_guard<std::mutex> lock(y);
--Y;
{
     std::lock_guard<std::mutex> lock(x);
   --X;
}
```

- Imagine this scenario:
  - Thread 1 locks x
  - Thread 2 locks y
  - Thread 1 tries to lock y and waits

Tip: Avoid Deadlocks

```
//thread 1
std::lock_guard<std::mutex> lock(x);
++X;
{
   std::lock_guard<std::mutex> lock(y);
   ++Y;
}
```

```
//thread 2
  std::lock_guard<std::mutex> lock(y);
--Y;
{
   std::lock_guard<std::mutex> lock(x);
   --X;
}
```

- Imagine this scenario:
  - Thread 1 locks x
  - Thread 2 locks y
  - Thread 1 tries to lock y and waits
  - Thread 2 tries to lock x and waits

Tip: Avoid Deadlocks

```
//thread 1
std::lock_guard<std::mutex> lock(x);
++X;
{
    std::lock_guard<std::mutex> lock(y);
    ++Y;
}
```

```
//thread 2
std::lock_guard<std::mutex> lock(y);
--Y;
{
    std::lock_guard<std::mutex> lock(x);
    --X;
}
```

- Imagine this scenario:
  - Thread 1 locks x
  - Thread 2 locks y
  - Thread 1 tries to lock y and waits
  - Thread 2 tries to lock x and waits
- Deadlock!

Tip: Avoid Deadlocks

- One solution: lock both

```
int X = 0, Y = 0;
std::mutex both;

//thread 1                                  //thread 2
std::lock_guard<std::mutex> lock(both);      std::lock_guard<std::mutex> lock(both);
++X;                                        --X;
++Y;                                        --Y;
```

Tip: Avoid Deadlocks

- Another solution: use atomic variables

```
std::atomic<int> X = 0, Y = 0;

//thread 1                    //thread 2
++X;                          --X;
++Y;                          --Y;
```

Staying Responsive

- GUIs should always be responsive

Staying Responsive

- GUIs should always be responsive

```
void OnButtonClicked() {
   tasks_.emplace_back(std::async([] {
      doLongTask();
   }));
}
```

Splitting up Work

- Jobs can be divided

Splitting up Work

- Jobs can be divided

```
std::vector<std::future<void>> futs;
futs.reserve(numCores);
```

Splitting up Work

- Jobs can be divided

```
std::vector<std::future<void>> futs;
futs.reserve(numCores);
auto perThr = totalToCalc / numCores;
```

Splitting up Work

- Jobs can be divided

```
std::vector<std::future<void>> futs;
futs.reserve(numCores);
auto perThr = totalToCalc / numCores;

for (int i = 0; i < numCores; ++i) {
    futs.emplace_back(
        std::async(calc, perThr*i, (perThr+1) * i)
    );
}
```

Compiler Support

- MSVC - Most
- Clang - Full

Compiler Support

- MSVC - Most
- Clang - Full
- GCC
    - Windows - Atomics

Compiler Support

- MSVC - Most
- Clang - Full
- GCC
  - Windows - Atomics
  - Other - Most

C++ Concurrency Awaits

Go have fun with it!

References

- http://www.numberworld.org/y-cruncher/
- http://www.charlespetzold.com/blog/2011/11/Asynchronous-Processing-in-Windows-8.html
- http://channel9.msdn.com/Shows/Going+Deep/C-and-Beyond-2012-Herb-Sutter-Concurrency-and-Parallelism
- http://en.cppreference.com/w/cpp/thread
- http://pages.cs.wisc.edu/~remzi/Classes/537/Fall2005/Lectures/lecture8.pdf