“Going Parallel with C++11”

Supercomputing 2013

Joe Hummel, PhD

U. of Illinois, Chicago

jhummel2@uic.edu

Jens Mache, PhD

Lewis & Clark College

jmache@lclark.edu

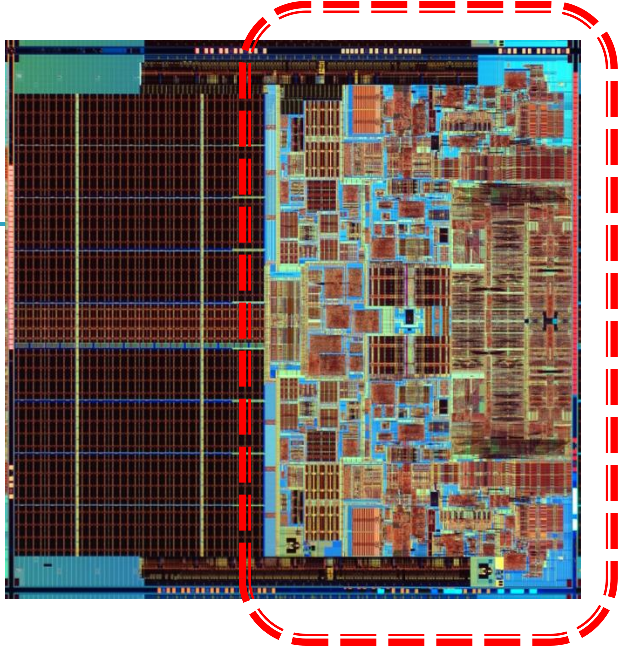
http://www.joehummel.net/downloads.html

Agenda

- New standard of C++ has been ratified
  - “C++0x” ==> “C++11”

- Lots of new features
- We’ll focus on concurrency features

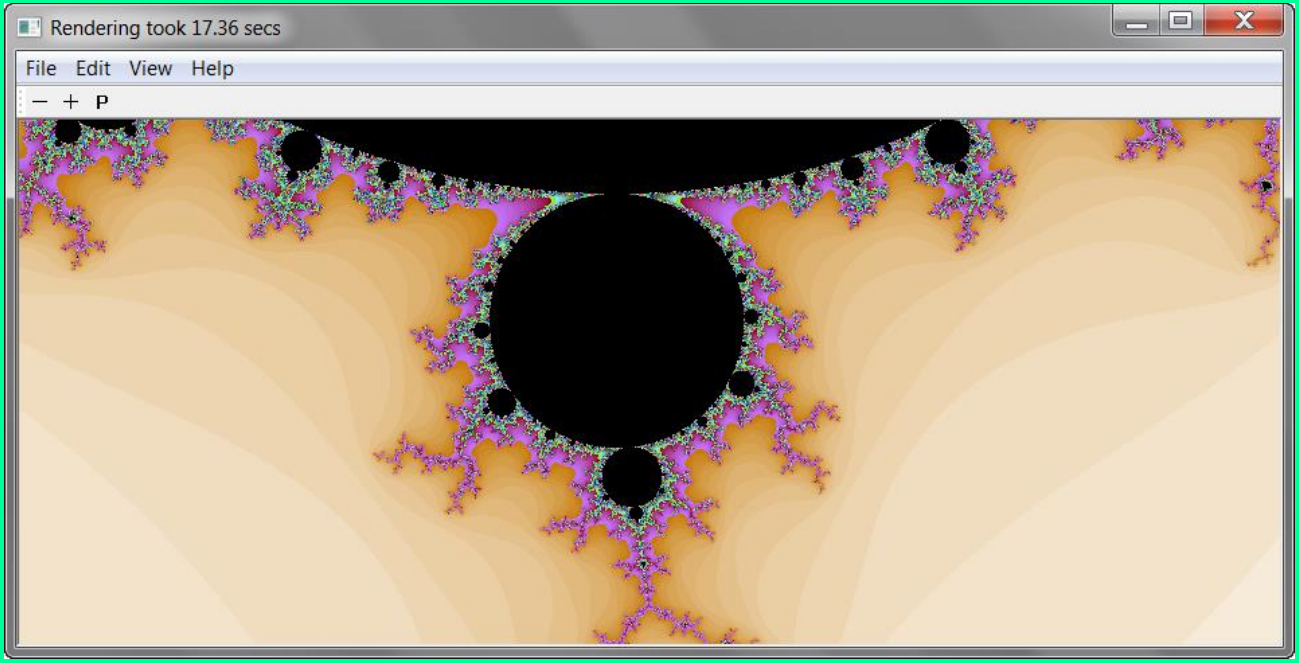
Motivation



- **Async programming:**
  - *Better responsiveness…*
  - *GUIs (desktop, web, mobile)*
  - *Cloud*
  - *Windows 8*
- **Parallel programming:**
  - *Better performance…*
  - *Financials*
  - *Pharma*
  - *Engineering*
  - *Big data*

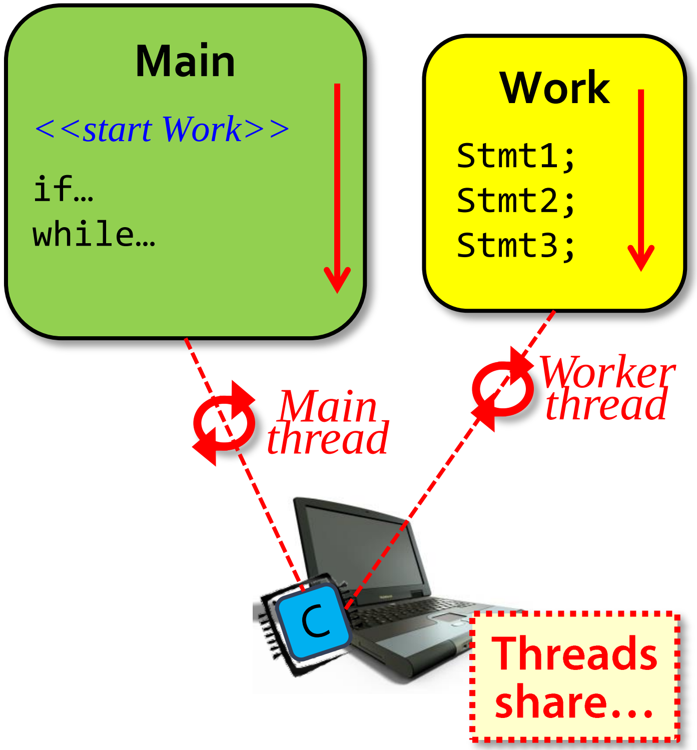
Demo

- Mandelbrot Set…



Execution Model

- Single core:



Main

<<start Work>>

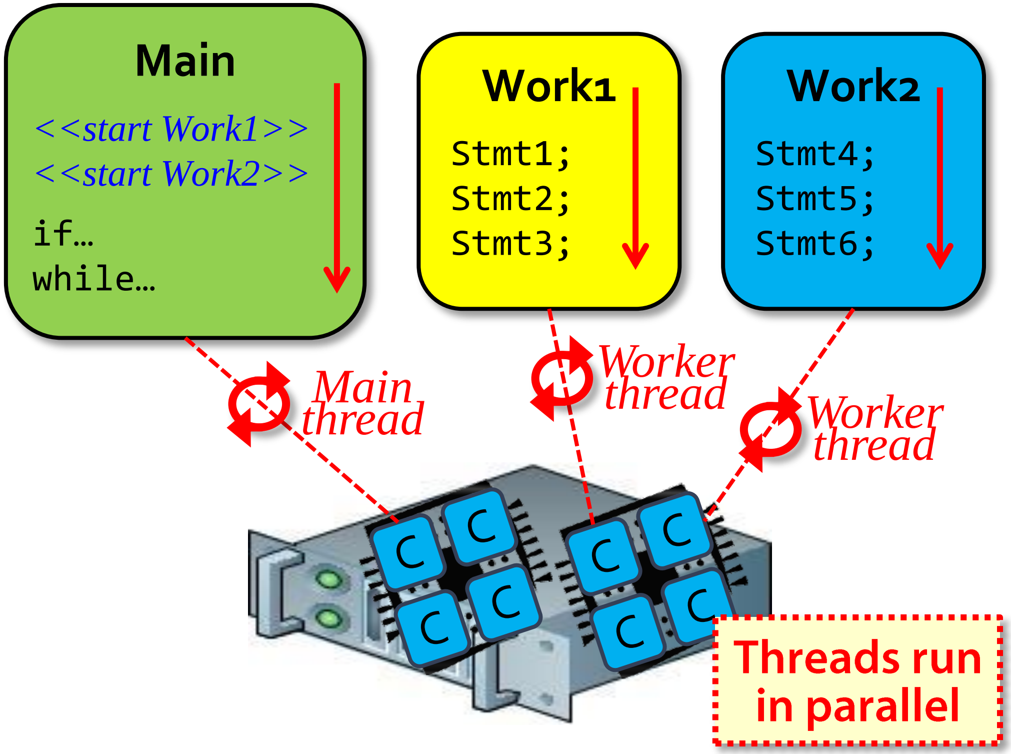
```
if…
while…
```

Work

```
Stmt1;
Stmt2;
Stmt3;
```

Threads share…

- Multicore:



Main

<<start Work1>>
<<start Work2>>

```
if…
while…
```

Work1

```
Stmt1;
Stmt2;
Stmt3;
```

Work2

```
Stmt4;
Stmt5;
Stmt6;
```

Threads run in parallel

Threading models

- Numerous threading models are available:
  - *POSIX (aka Pthreads)*
  - *Win32 (aka Windows)*
  - *Boost*
  - *Java*
  - *.NET*
  - …

C++11

- C++11 threads are the new kid on the block
  - `std::thread` *class now part of standard C++ library*
  - `std::thread` *is an abstraction — maps to local platform threads (POSIX, Windows, etc.)*

“Hello World” with std::thread

```
#include <thread>
#include <iostream>

void func()
{
  std::cout << "**Inside thread "
   << std::this_thread::get_id() << "!" << std::endl;
}

int main()
{
  std::thread  t( func );


  t.join();
  return 0;
}
```

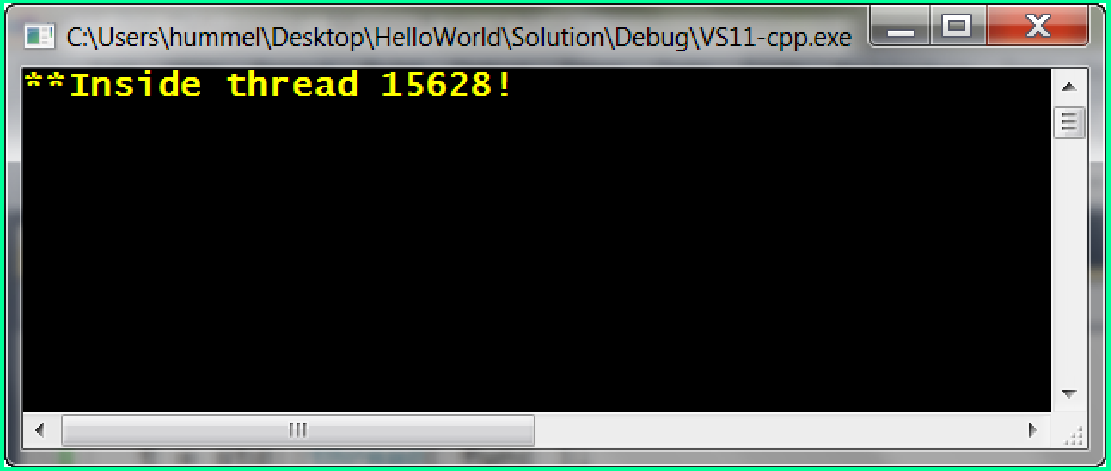
A simple function for thread to do…

Create & schedule thread to execute ***func****…*

Wait for thread to finish…

Demo

- Hello world…



C:\Users\hummel\Desktop\HelloWorld\Solution\Debug\VS11-cpp.exe

```
**Inside thread 15628!
```

Avoiding errors / program termination…

```
#include <thread>
#include <iostream>

void func()
{
  std::cout << "**Hello world...\n";
}

int main()
{
  std::thread  t( func );


  t.join();
  return 0;
}
```

*(1) Thread function must do **exception handling**; unhandled exceptions ==> error termination…*

```
void func()
{
  try
  {
     // computation:
  }
  catch(...)
  {
     // do something:
  }
}
```

*(2) Must **join** with thread *before* handle goes out of scope, otherwise error termination…*

***NOTE**: avoid use of detach( ) in C++11, difficult to use safely with general resource cleanup. But it's an option.*

Speaking of avoiding errors…

- std::thread written to *prevent* copying of threads

```
int main()
{
  std::thread  t( func );


  std::thread  t2(t);
  std::thread  t3 = t;
```

compilation errors…

But you can move, and reference…

```
std::thread  t2( std::move(t) );

// NOTE: t is no longer valid!
assert( t.joinable() == false );

std::thread& t3 = t2;
  .
  .
  .
t2.join();  // or t3.join();
```

std::move tells compiler to invoke move constructor: thread(thread&& other)

std::thread

- Constructors:

```
class thread
{
   thread();    // creates new thread object that does *not* represent a thread (i.e. not joinable)

   thread( std::Function&& f, Args&&... args );    // creates new thread to execute f

   thread( thread&& other);    // *move* constructor

   ~~thread( thread& other);~~    // *copy* constructor --- not available
```

```
template<class std::Function, class... Args>
                  explicit thread(std::Function&& f, Args&&... args);
```

Programming style

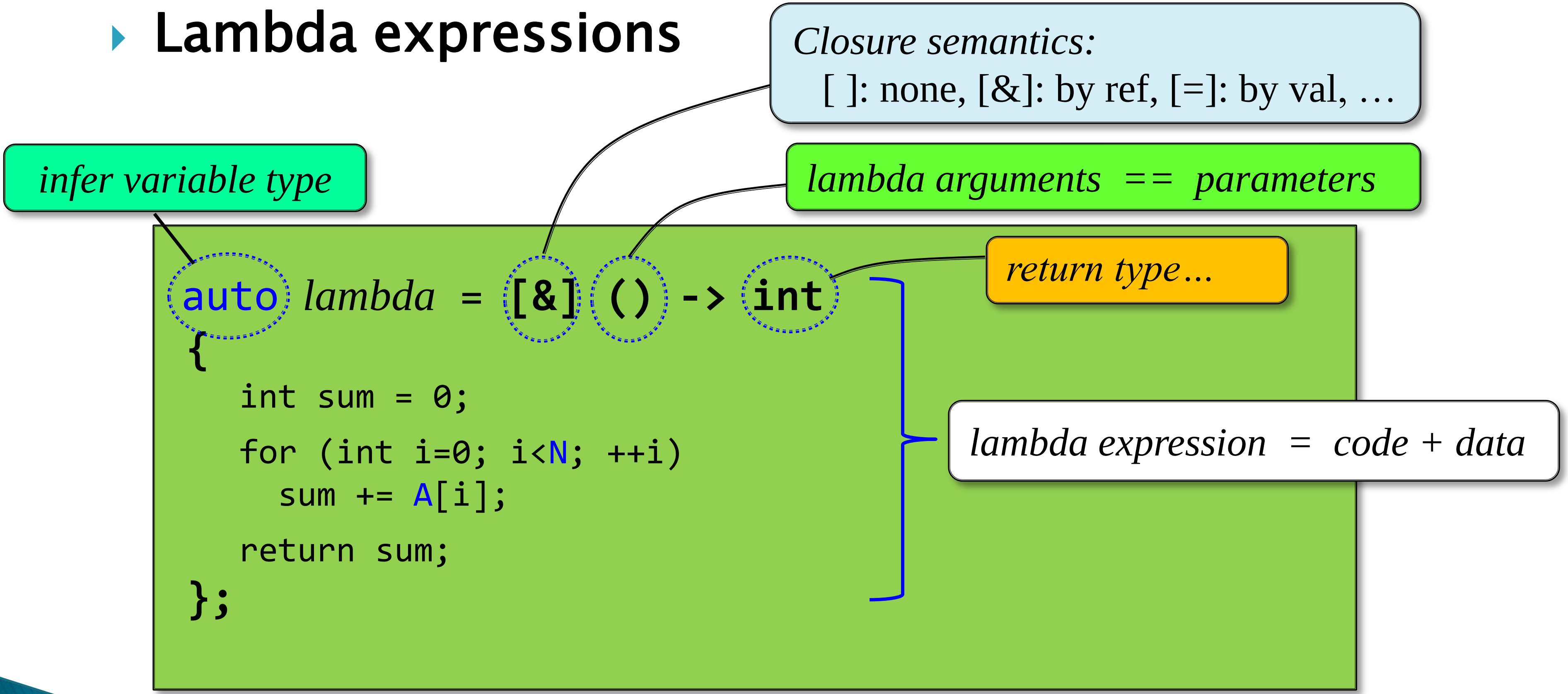
- Old school:
  - thread functions (what we just saw)
- Middle school:
  - function objects

```
class FuncObject
{
  public:
    void operator() (void)
    { cout << this_thread::get_id() << endl; }
};

int main()
{
  FuncObject   f;
  std::thread  t( f );
```

- New school:
  - C++11 now offers lambda expressions
    - *aka anonymous functions*

New C++11 language features

- Type inference
- Lambda expressions



```
auto lambda = [&] () -> int
{
   int sum = 0;
   for (int i=0; i<N; ++i)
     sum += A[i];
   return sum;
};
```

Closure semantics:
[ ]: none, [&]: by ref, [=]: by val, …

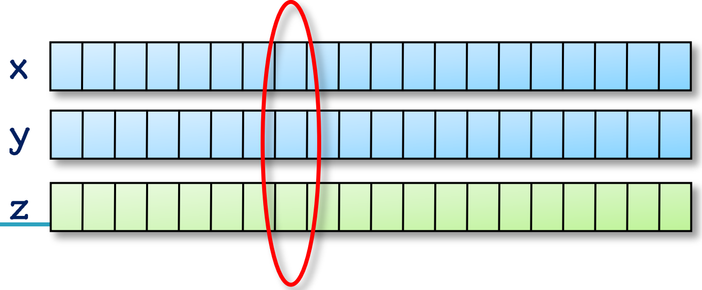
infer variable type

lambda arguments == parameters

return type…

lambda expression = code + data

Example: saxpy



- Saxpy == *Scalar Alpha X Plus Y*
  - *Scalar multiplication and vector addition*

```
for (int i=0; i<n; i++)
   z[i] = a * x[i] + y[i];
```

Parallel

```
auto code = [&](int start, int end) -> void
{
  for (int i = start; i < end; i++)
    z[i] = a * x[i] + y[i];
};


thread t1(code, 0   /*start*/, N/2 /*end*/);
thread t2(code, N/2 /*start*/, N   /*end*/);
```

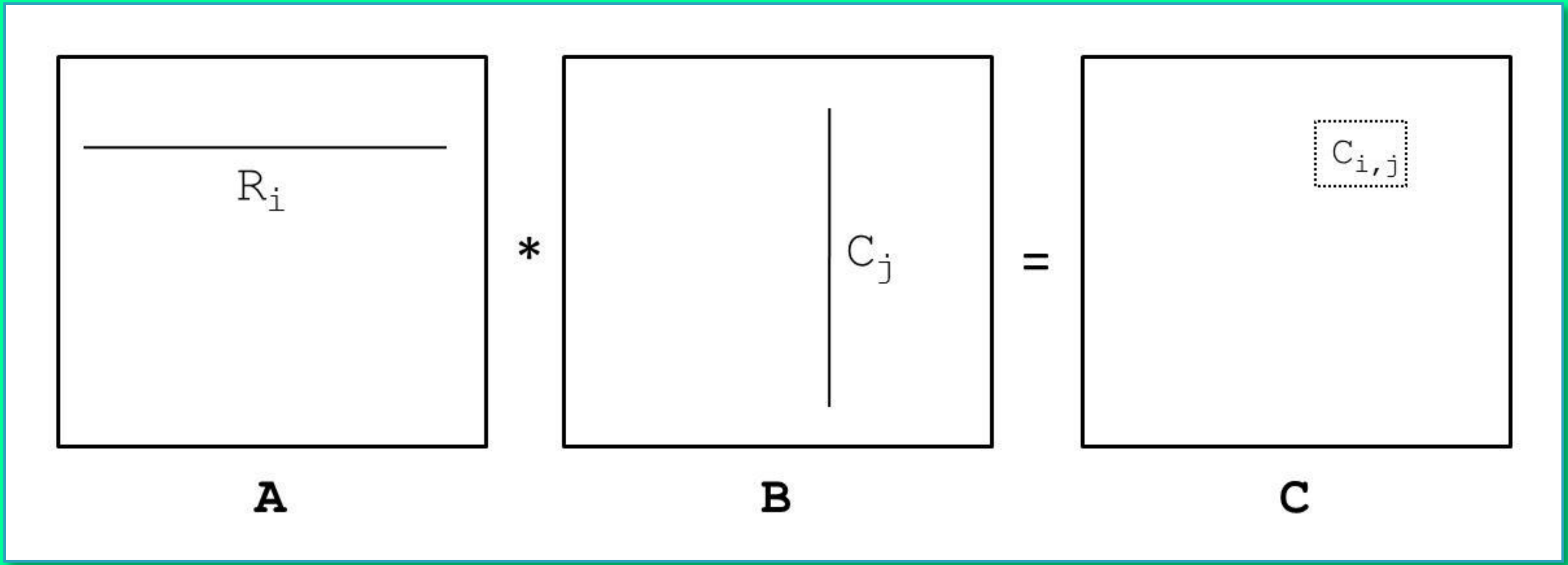
Trade-offs

- **Lambdas**:
  - Easier and more readable -- code remains inline
  - Potentially more dangerous ([&] captures everything by ref)

- **Functions**:
  - More efficient -- lambdas involve class, function objects
  - Potentially safer -- requires explicit variable scoping
  - More cumbersome and less readable

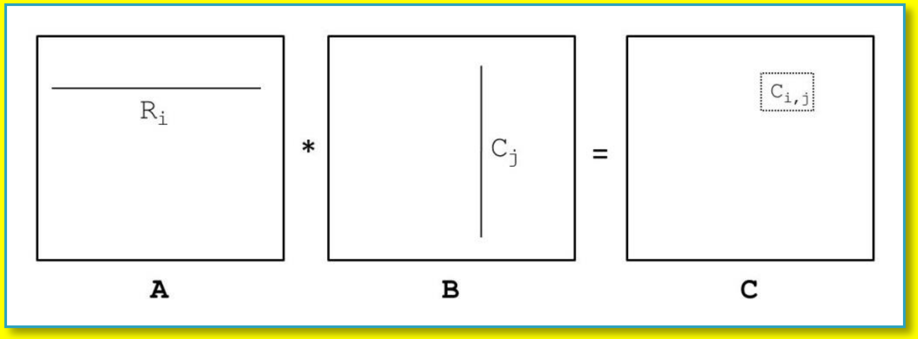
Demo: a complete example

- Matrix multiply…



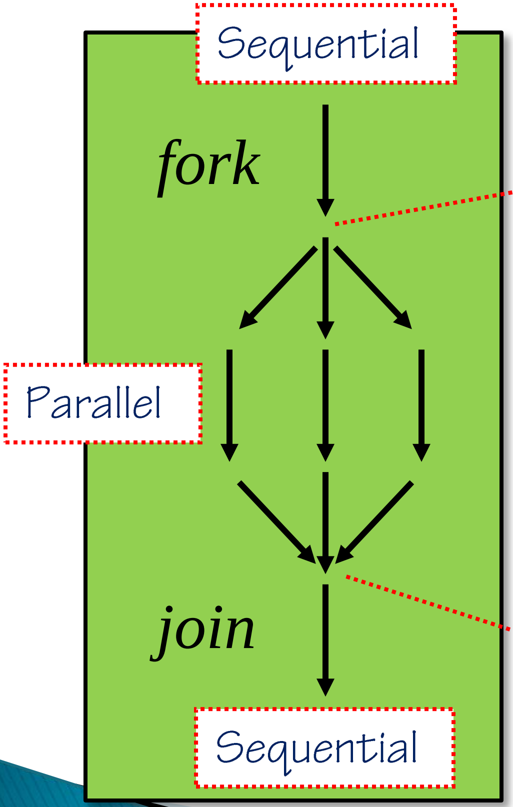
Sequential version...

```
//
// Naïve, triply-nested sequential solution:
//
for (int i = 0; i < N; i++)
{
   for (int j = 0; j < N; j++)
   {
      C[i][j] = 0.0;

      for (int k = 0; k < N; k++)
         C[i][j] += (A[i][k] * B[k][j]);
   }
}
```



Structured ("fork-join") parallelism

- A common pattern when creating multiple threads



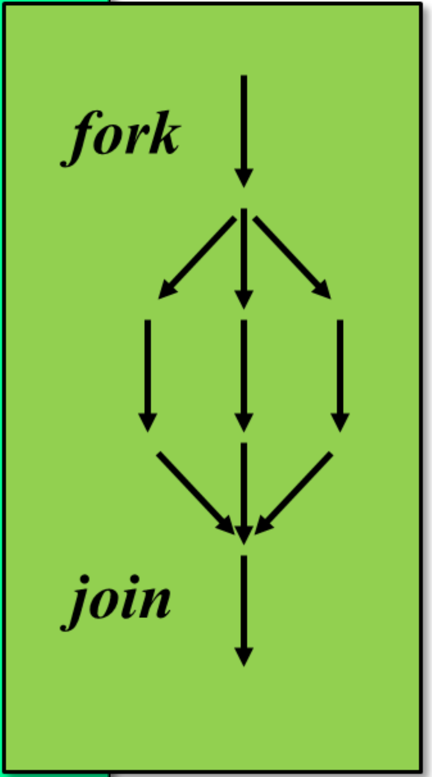
```
#include <vector>

std::vector<std::thread>  threads;

int cores = std::thread::hardware_concurrency();

for (int i=0; i<cores; ++i)   // 1 per core:
{
  auto code = []() { DoSomeWork(); };
  threads.push_back( thread(code) );
}
```

```
for (std::thread& t : threads)  // new range-based for:
  t.join();
```

Parallel solution

```
// 1 thread per core:
numthreads = thread::hardware_concurrency();
```

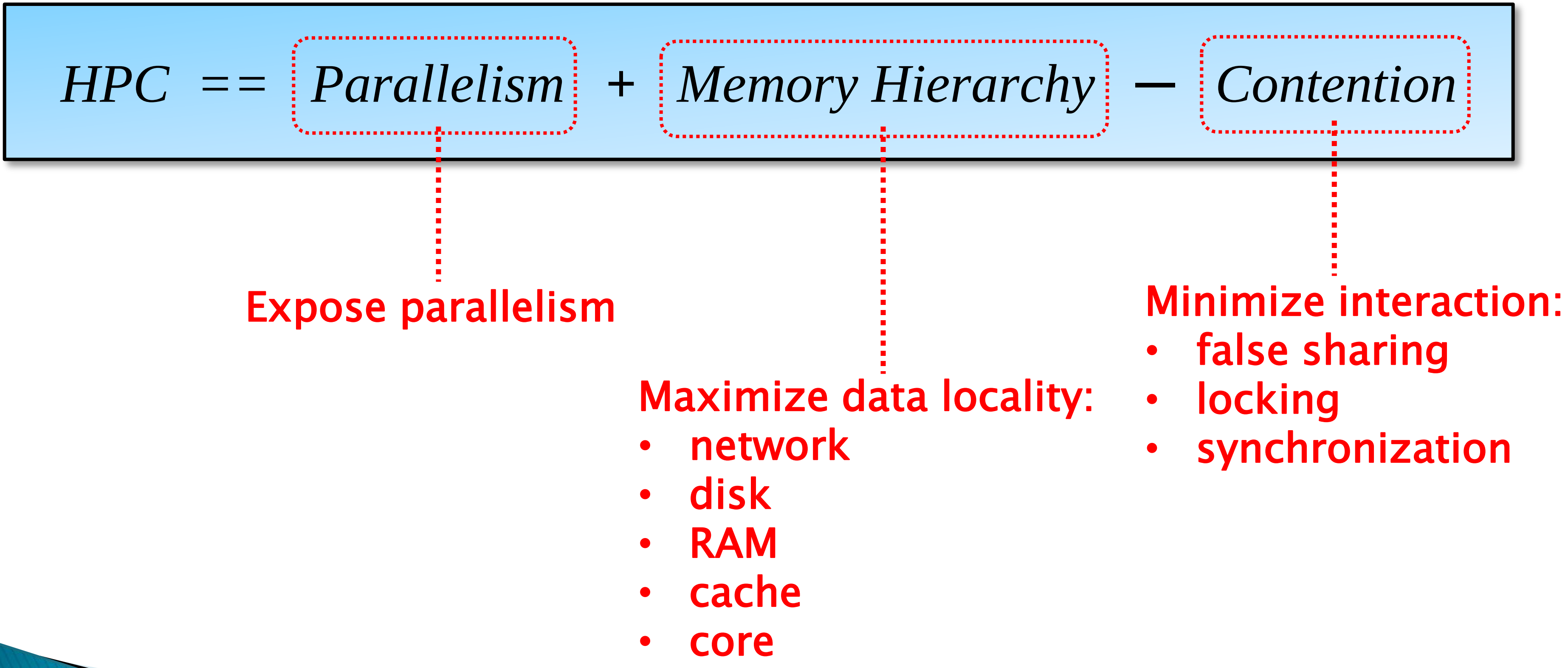
```
int rows  = N / numthreads;
int extra = N % numthreads;
int start = 0;     // each thread does [start..end)
int end   = rows;

auto code = [N, &C, &A, &B](int start, int end) -> void
{
  for (int i = start; i < end; i++)
    for (int j = 0; j < N; j++)
    {
      C[i][j] = 0.0;
      for (int k = 0; k < N; k++)
        C[i][j] += (A[i][k] * B[k][j]);
    }
};

vector<thread>  workers;

for (int t = 1; t <= numthreads; t++)
{
  if (t == numthreads) // last thread does extra rows:
    end += extra;

  workers.push_back( thread(code, start, end) );

  start = end;
  end   = start + rows;
}
```



```
for (thread& t : workers)
   t.join();
```

High–Performance Computing

- Parallelism alone is not enough…



HPC == Parallelism + Memory Hierarchy – Contention

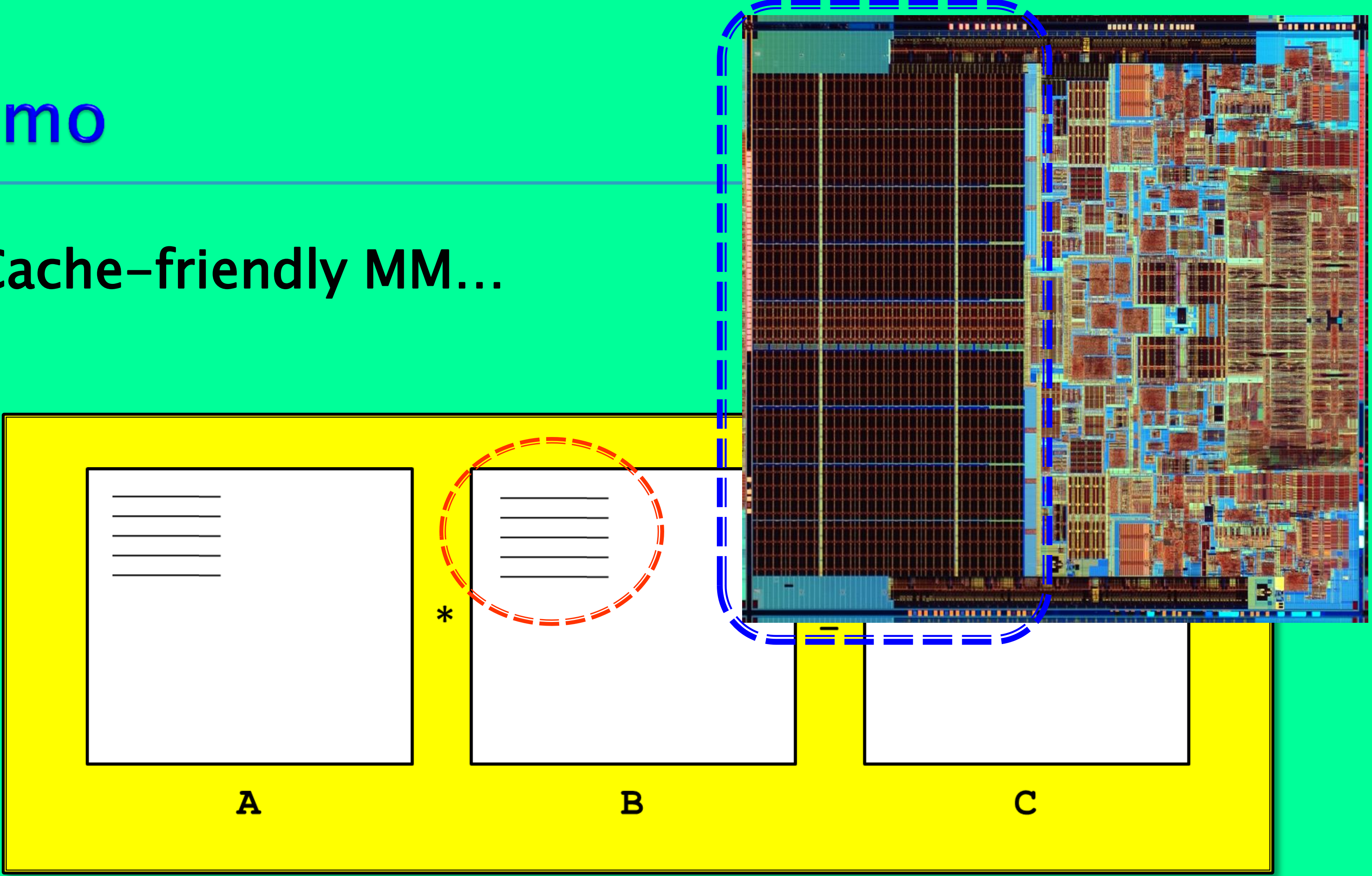
Expose parallelism

Maximize data locality:
- network
- disk
- RAM
- cache
- core

Minimize interaction:
- false sharing
- locking
- synchronization

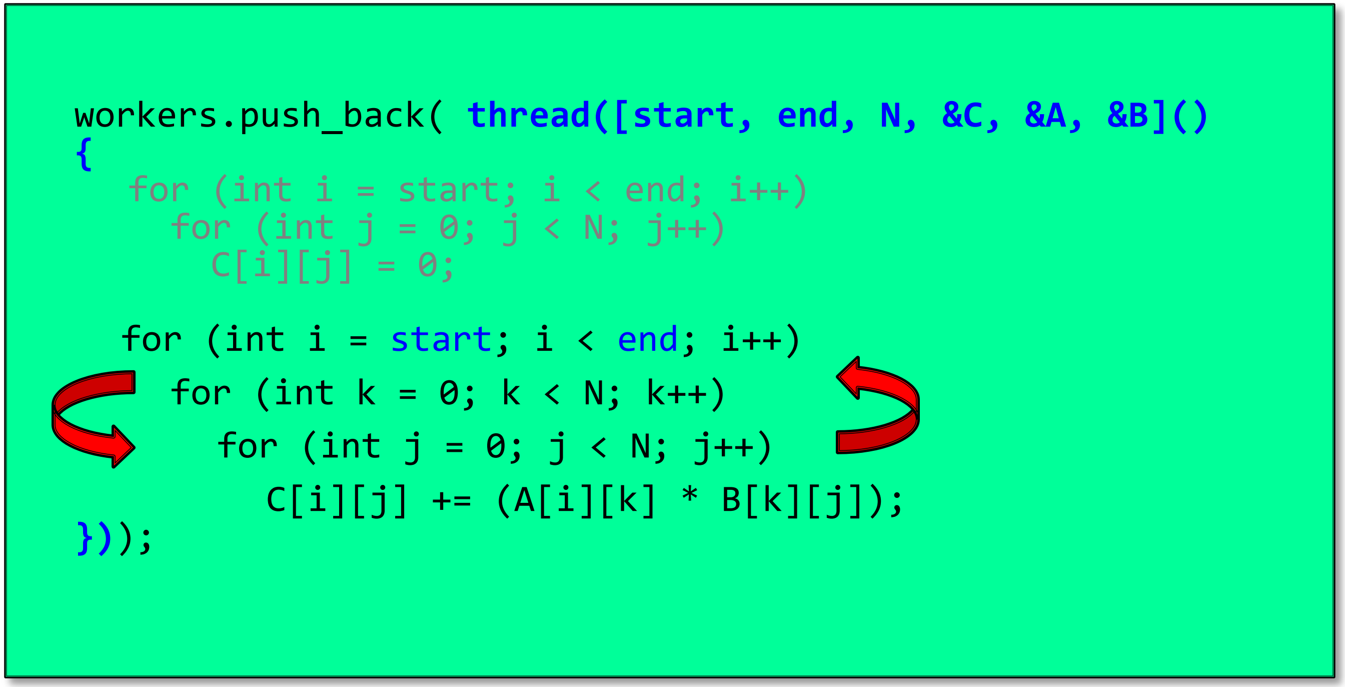
Demo

- Cache-friendly MM…



Loop interchange

- Significantly-better caching, and performance…



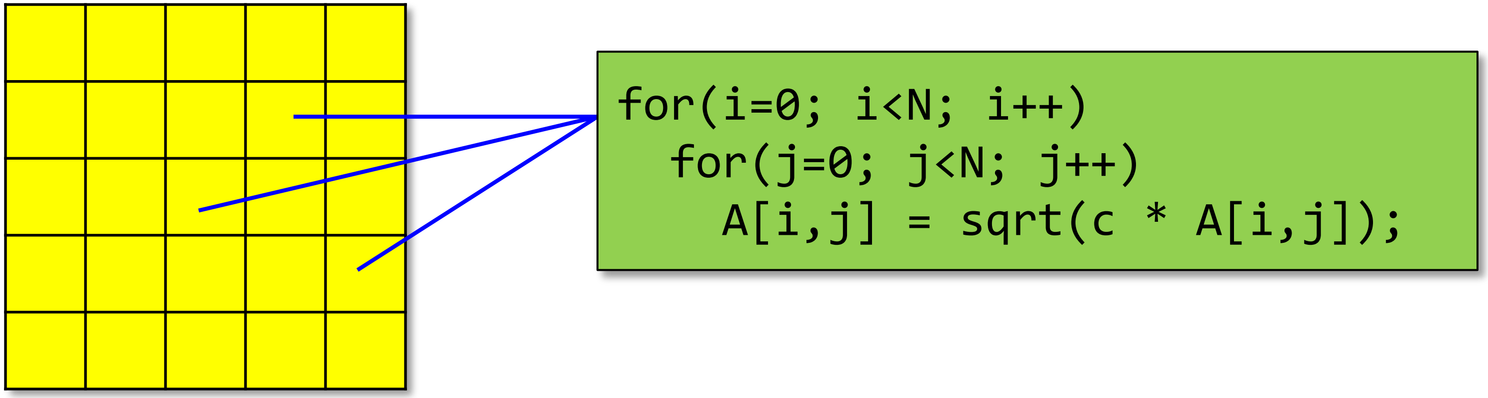
```
workers.push_back( thread([start, end, N, &C, &A, &B]()
{
   for (int i = start; i < end; i++)
     for (int j = 0; j < N; j++)
       C[i][j] = 0;

   for (int i = start; i < end; i++)
     for (int k = 0; k < N; k++)
       for (int j = 0; j < N; j++)
         C[i][j] += (A[i][k] * B[k][j]);
}));
```

Types of parallelism

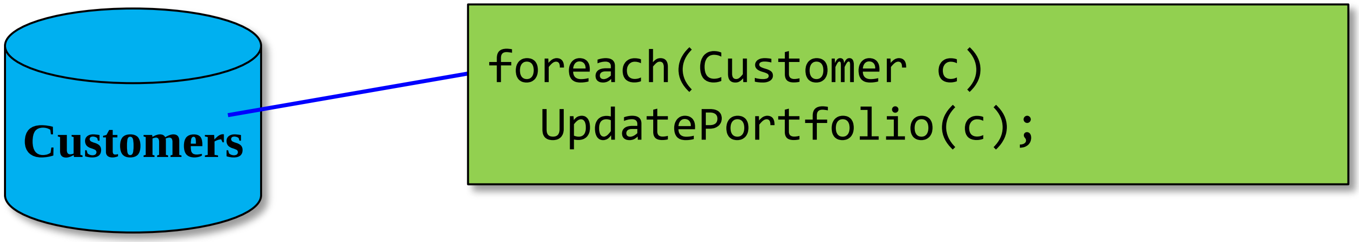
- **Most common types:**
  - Data
  - Task
  - Embarrassingly parallel
  - Dataflow

(1) Data parallelism

- **Def**: ***same*** *operation executed in parallel on* ***different*** *data.*



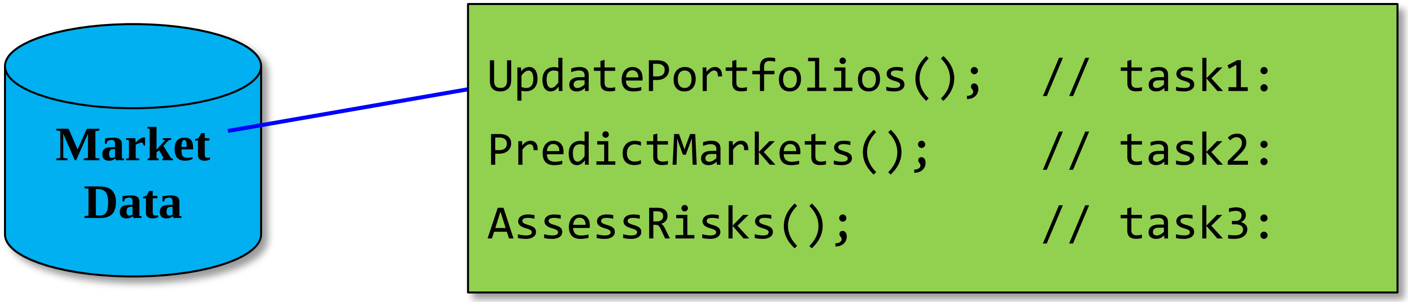
```
for(i=0; i<N; i++)
  for(j=0; j<N; j++)
    A[i,j] = sqrt(c * A[i,j]);
```



```
foreach(Customer c)
  UpdatePortfolio(c);
```

(2) Task parallelism

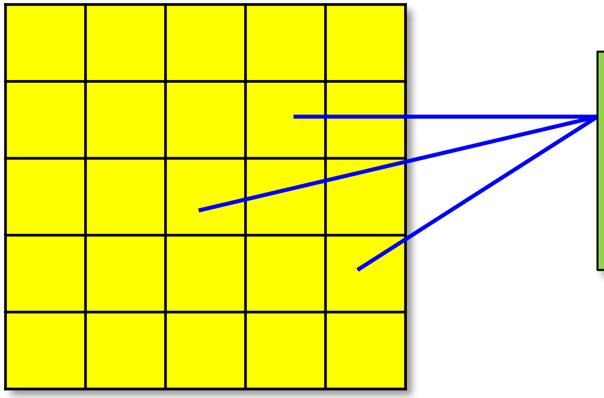
- **Def**: ***__different__*** *operations executed in parallel.*



```
UpdatePortfolios();  // task1:
PredictMarkets();    // task2:
AssessRisks();       // task3:
```

(3) Embarrassingly parallel

- **Def:** *a problem is **embarrassingly parallel** if the computations are **independent** of one another.*

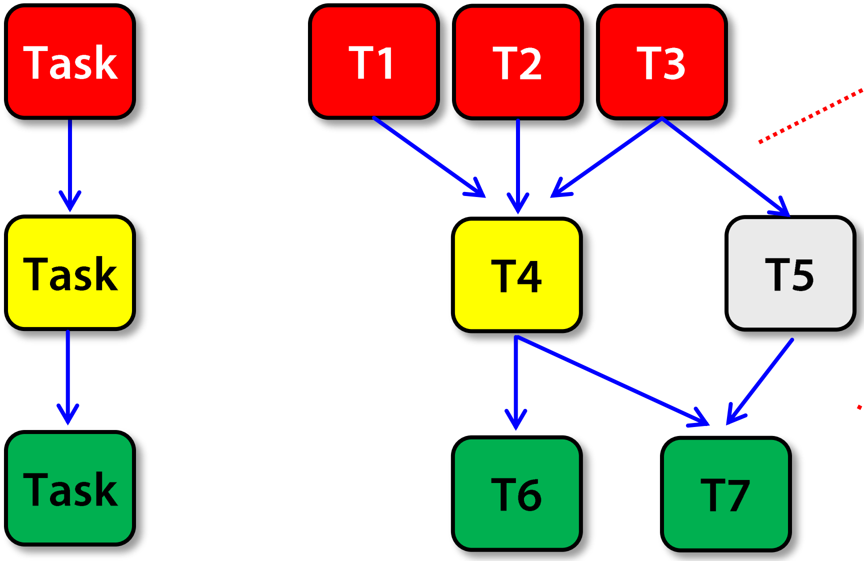


```
for(i=0; i<N; i++)
  for(j=0; j<N; j++)
    A[i,j] = sqrt(c * A[i,j]);
```

*Not embarrassing at all, but in fact yields the best results. "**Delightfully parallel**"*

(4) Dataflow

- **Def:** *when operations* ***depend*** *on one another.*
  - data "flows" from one operation to another…



parallel execution requires communication / coordination

Depending on nature of dataflow, may not parallelize well…

Dataflow example

- Image processing…

```
for(r=1; r<Rows-1; r++)
  for(c=1; c<Cols-1; c++)
    image[r,c] = Avg(image[  r-1, c ],   // N:
                     image[  r+1, c ],   // S:
                     image[  r, c+1 ],   // E:
                     image[  r, c-1 ]);  // W:
```

Status of C++11

Compilers…

- Most compilers fully implement C++11
- gcc 4.8.1 has complete support
  - http://gcc.gnu.org/projects/cxx0x.html
- clang 3.3 has complete support
  - http://clang.llvm.org/cxx_status.html
- Visual C++ 2012 has reasonable support
  - Near complete support for concurrency
  - Most of the major features of C++11 are there as well…
  - Will be nearly complete in VC++ 2013, but not 100%

Compiling with gcc

```
# makefile

# threading library: one of these should work
# tlib=thread
tlib=pthread

# gcc 4.6:
# ver=c++0x
# gcc 4.7 and 4.8:
ver=c++11


build:
         g++  -std=$(ver)  -Wall  main.cpp  -l$(tlib)
```

C++11 Concurrency Features

Concept	Header	Summary
Threads	`<thread>`	Standard, low-level, type-safe; good basis for building HL systems (*futures*, *tasks*, …)
Futures	`<future>`	Via **async** function; hides threading, better harvesting of return value & exception handling
Locking	`<mutex>`	Standard, low-level locking primitives
Condition Vars	`<condition_variable>`	Low-level synchronization primitives
Atomics	`<atomic>`	Predictable, concurrent access without data race
Memory Model		***"Catch Fire" semantics***; if program contains a data race, behavior of memory is **undefined**
Thread Local		Thread-local variables [*problematic => avoid*]

Futures

Futures

- Futures provide a higher level of abstraction
  - *you start an asynchronous / parallel operation*
  - *you are returned a handle to wait for the result*
  - *thread creation, join, and exceptions are handled for you*

std::async + std::future

- Use async to start asynchronous operation
- Use returned future to wait upon result / exception

START

```
#include <future>

std::future<int> f = std::async( []() -> int
  {
    int result = PerformLongRunningOperation();
    return result;
  }
);
.
.
```

lambda return type…

WAIT

```
try
{
  int x = f.get();   // wait if necessary, harvest result:
  cout << x << endl;
}
catch(exception &e)
{
  cout << "**Exception: " << e.what() << endl;
}
```

Async operations

- Run on current thread *or* a new thread
- By default, system decides…
  - *based on current load, available cores, etc.*

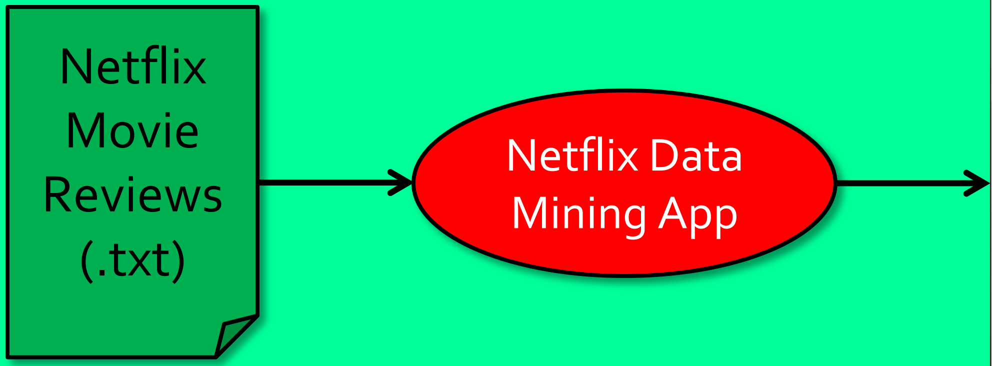
```
// runs on current thread when you "get" value (i.e. lazy execution):
future<T> f1 = std::async( std::launch::deferred, []() -> T {...} );

// runs now on a new, dedicated thread:
future<T> f2 = std::async( std::launch::async, []() -> T {...} );

// let system decide (e.g. maybe you created enough work to keep system busy?):
future<T> f3 = std::async( []() -> T {...} );
```

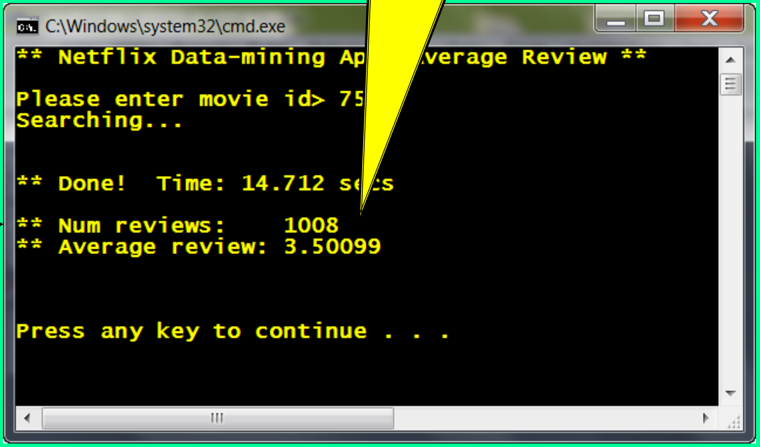
optional argument missing

Demo

- **Netflix data-mining…**

Average rating…





```
** Netflix Data-mining Ap    verage Review **

Please enter movie id> 75
Searching...


** Done!  Time: 14.712 se  s

** Num reviews:     1008
** Average review: 3.50099


Press any key to continue . . .
```

Sequential solution

```
cin >> movieID;

vector<string> ratings = readFile("ratings.txt");

tuple<int,int> results = dataMine(ratings, movieID);

int numRatings = std::get<0>(results);
int sumRatings = std::get<1>(results);
double avgRating = double(numRatings) / double(sumRatings);

cout << numRatings << endl;
cout << avgRating << endl;
```

```
dataMine(vector<string> &ratings, int id)
{
  foreach rating
    if ids match num++, sum += rating;

  return tuple<int,int>(num, sum);
}
```

Parallel solution

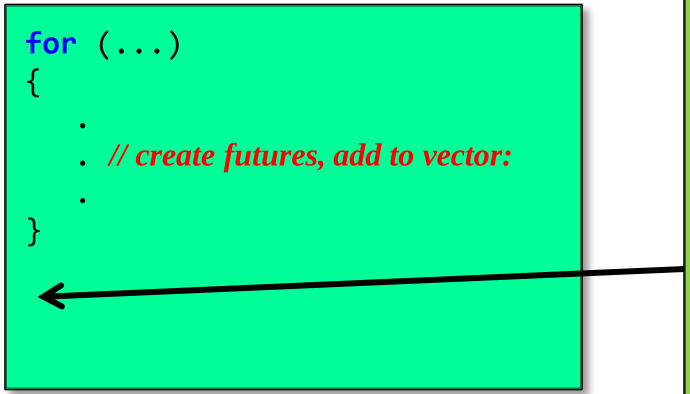
```
dataMine(..., int begin, int end)
{
  foreach rating in begin..end
    if ids match num++, sum += rating;

  return tuple<int,int>(num, sum);
}
```

```
int chunksize = ratings.size() / numthreads;
int leftover  = ratings.size() % numthreads;
int begin     = 0;      // each thread does [start..end)
int end       = chunksize;

vector<future<tuple<int,int>>>  futures;

for (int t = 1; t <= numthreads; t++)
{
   if (t == numthreads)   // last thread does extra rows:
      end += leftover;

   futures.push_back(
     async([&ratings, movieID, begin, end]() -> tuple<int,int>
     {
         return dataMine(ratings, movieID, begin, end);
     })
   );

   begin = end;
   end   = begin + chunksize;
}
```

```
for (future<tuple<int,int>> &f: futures)
{
   tuple<int, int> t = f.get();
   numRatings += std::get<0>(t);
   sumRatings += std::get<1>(t);
}
```

WaitAll

- Futures provide a way to check if result is available
  - *this way we don't "wait" unless there is data to process…*



```
for (...)
{
    .
    . // create futures, add to vector:
    .
}
```

```
// WaitAll:  wait and process futures in order they complete, versus
// the order they appear in vector.  This is O(N), N = vector size.
size_t cur = 0;
size_t running = futures.size();

while (running > 1) {   // poll vector of futures for one that is ready:
   std::future<std::tuple<int,int>> &f = futures[cur];

   auto status = f.wait_for(std::chrono::milliseconds(10));
   if (status == std::future_status::ready) {
      std::tuple<int, int> t = f.get();
      numRatings += std::get<0>(t);
      sumRatings += std::get<1>(t);

      running--;
      futures.erase(futures.begin() + cur);
   }

   cur++; if (cur >= running) cur = 0;
}

std::tuple<int, int> t = futures[0].get();  // last one, just wait:
numRatings += std::get<0>(t);
sumRatings += std::get<1>(t);
```

The Dangers of Concurrency

Beware…

- Beware the many dangers of concurrency:

RACE CONDITIONS LIVELOCK DEADLOCK

OPTIMIZING COMPILERS

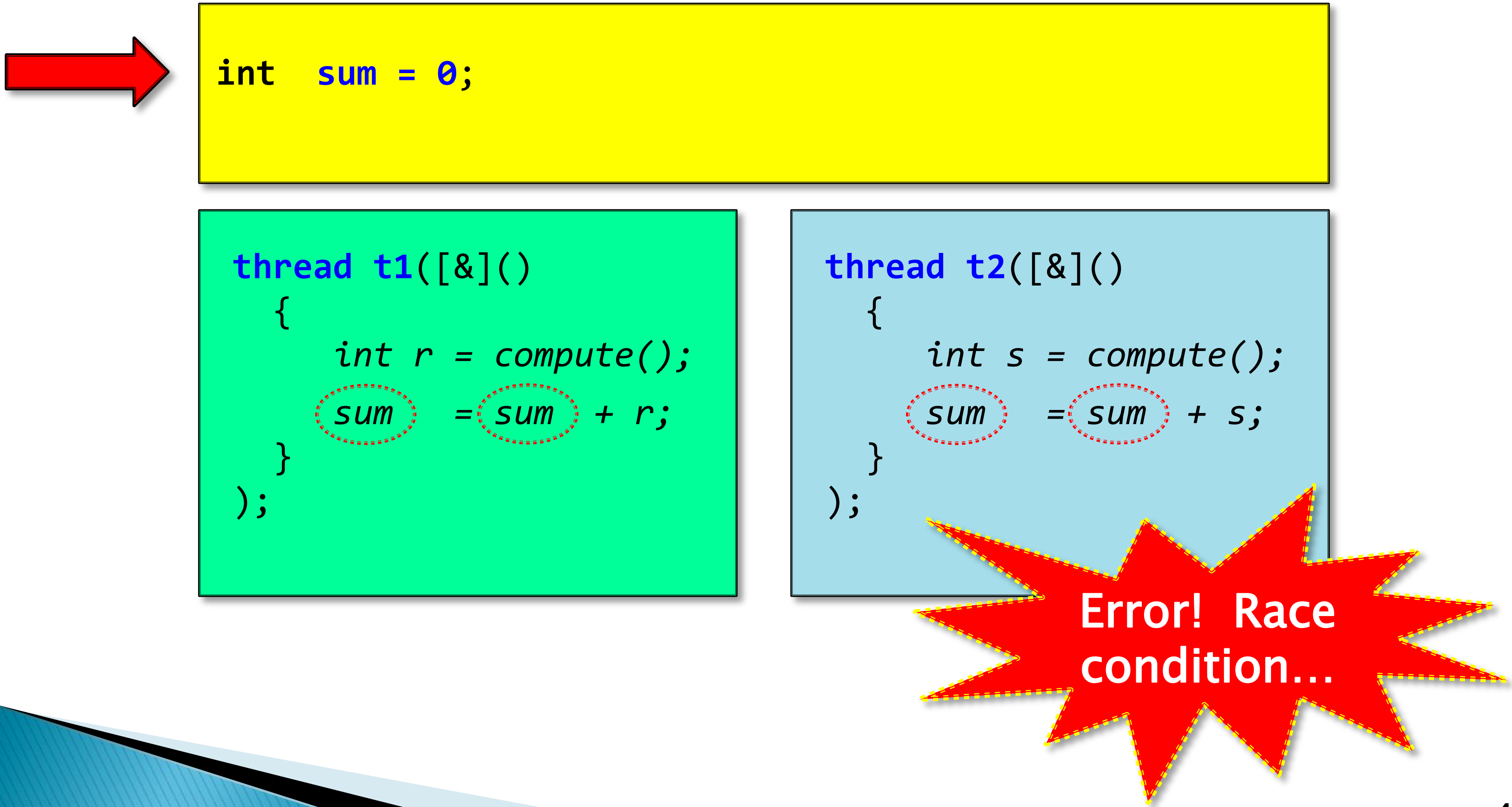
STARVATION OPTIMIZING HARDWARE

- Most common **pitfall** for application developers?

Race conditions…

Example

- Consider 2 threads accessing a shared variable…



```
int  sum = 0;
```

```
thread t1([&]()
  {
      int r = compute();
      sum   = sum  + r;
  }
);
```

```
thread t2([&]()
  {
      int s = compute();
      sum   = sum  + s;
  }
);
```

Error! Race condition…

C++11 Memory Model

- C++ committee thought long and hard on memory model semantics…
  - "*You Don't Know Jack About Shared Variables or Memory Models*", Boehm and Adve, CACM, Feb 2012
- **Conclusion**:
  - *No suitable definition in presence of race conditions*
- **Result in C++11**:
  - *Predictable memory model *only* in data-race-free codes*
  - *Computer may "**catch fire**" in presence of data races*

Data-race-free programs

Def: two memory accesses conflict if they

1. *access the same scalar object or contiguous sequence of bit fields, and*
2. *at least one access is a store.*

Def: two memory accesses participate in a data race if they

1. *conflict, and*
2. *can occur simultaneously.*

- **A program is *data-race-free* (DRF) if no sequentially-consistent execution results in a data race. Avoid anything else.**

How to avoid data races?

- Various solutions…
  - □ *redesign to eliminate (e.g. reduction)*
  - □ *use thread-safe entities (e.g. parallel collections)*
  - □ *use synchronization (e.g. locking)*

most preferred

↓

least preferred

Most preferred solution

- Redesign to eliminate shared resource…

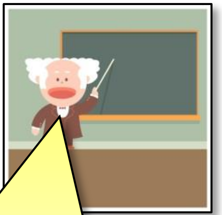
```
int sum = 0;
```

```
auto f1 = async([&]() -> int
  {
      int r = compute();
      return r;
  }
);
```

```
auto f2 = async([&]() -> int
  {
      int s = compute();
      return s;
  }
);
```

```
sum = f1.get() + f2.get();
```

Least preferred solution



Def: a critical section is the smallest region of code involved in a race condition.

- Use std::mutex (aka "lock") to control access to *critical section*…

```
#include <mutex>

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

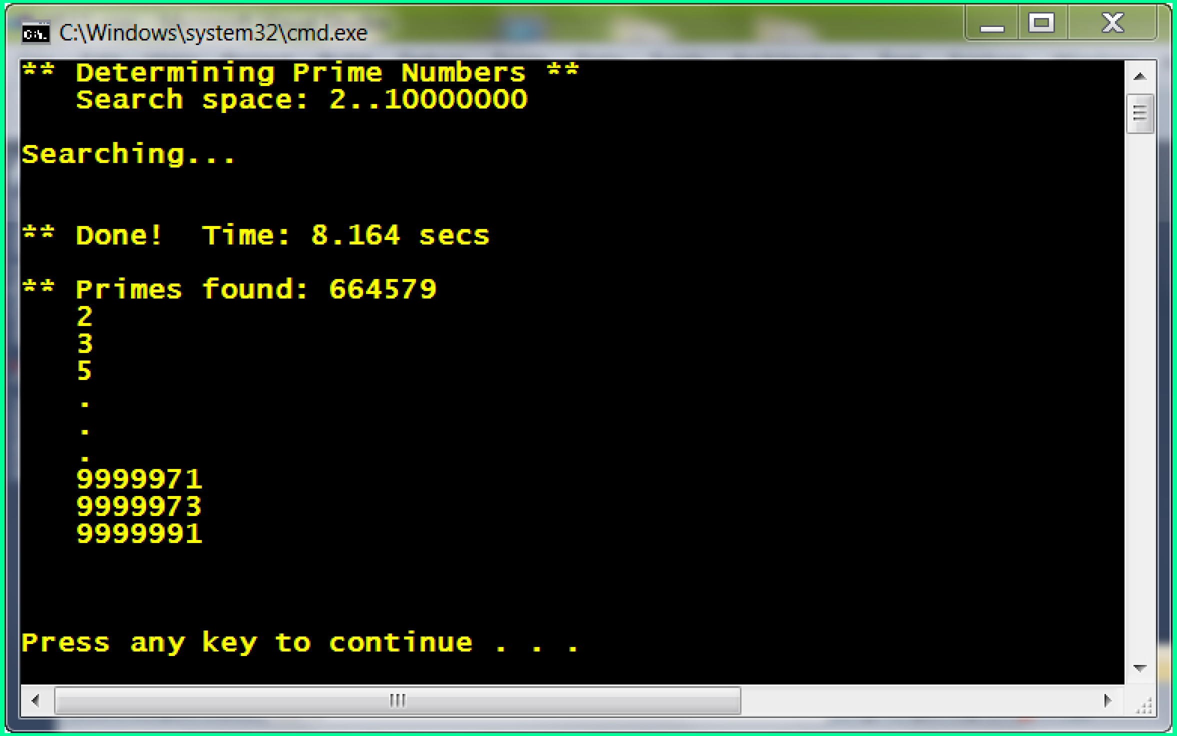
```
thread t1([&]()
  {
     int r = compute();

    m.lock();
     sum = sum + r;
    m.unlock();

  }
);
```

```
thread t2([&]()
  {
     int s = compute();

    m.lock();
     sum = sum + s;
    m.unlock();

  }
);
```

critical section

Demo

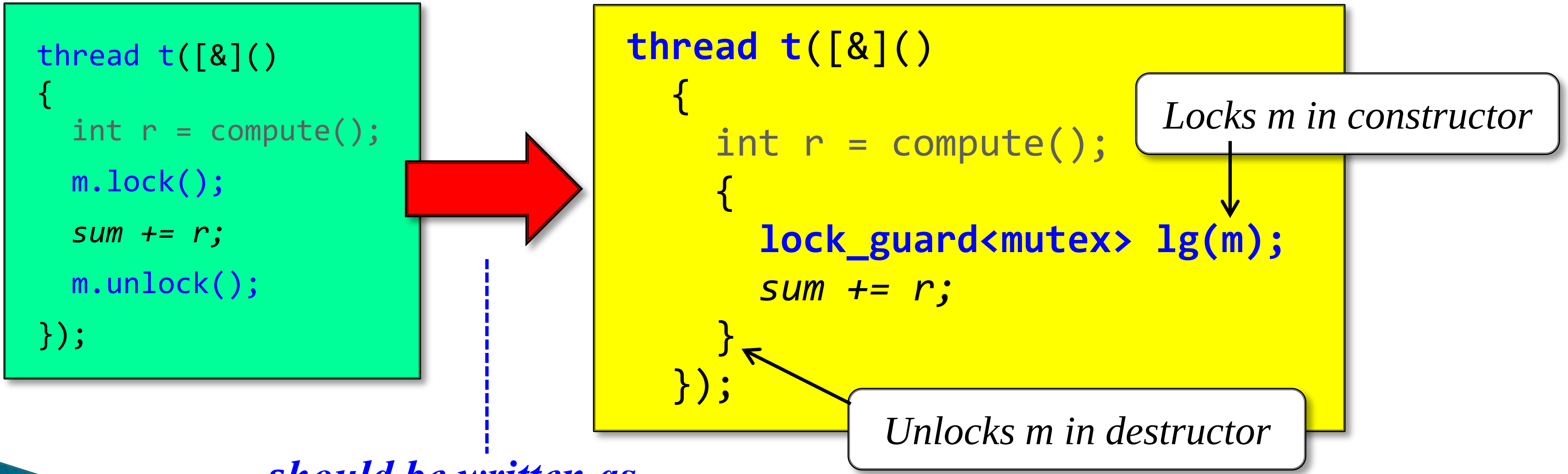
- Prime numbers…



```
** Determining Prime Numbers **
   Search space: 2..10000000

Searching...


** Done!  Time: 8.164 secs

** Primes found: 664579
   2
   3
   5
   .
   .
   .
   9999971
   9999973
   9999991



Press any key to continue . . .
```

RAII

- “***Resource Acquisition Is Initialization***”
  - *Advocated by B. Stroustrup for resource management*
  - *Uses constructor & destructor to properly manage resources (files, threads, locks, …) in presence of exceptions, etc.*



```
thread t([&]()
{
  int r = compute();
  m.lock();
  sum += r;
  m.unlock();
});
```

should be written as…

```
thread t([&]()
  {
    int r = compute();
    {
      lock_guard<mutex> lg(m);
      sum += r;
    }
  });
```

Locks m in constructor

Unlocks m in destructor

Atomics

- Can also use `std::atomic` to prevent data races…
  - *Lighter-weight than locking, but much more limited in applicability*

```
#include <atomic>

std::atomic<int> count(0);
```

```
thread t1([&]()
 {
   count++;
 });
```

```
thread t2([&]()
 {
   count++;
 });
```

```
thread t3([&]()
 {
   count = count + 1;
 });
```

not safe…

Primes…

```
vector<long>  primes;

for (long p = 2; p <= N; p++)
{
  if (isPrime(p))
     primes.push_back(p);
}
```

```
vector<long>    primes;
vector<thread>  workers;

mutex         m;
atomic<long>  candidate = 2;

for (int t = 1; t <= numthreads; t++)
{
  workers.push_back( thread([&]() -> void
    {
      while (true)
      {
         int p = candidate++;
         if (p > N) break;

         if (isPrime(p)) {
            lock_guard<mutex> _(m);
            primes.push_back(p);
         }
      }
    })
  );
}

for (thread& t : workers)
  t.join();

sort(primes.begin(), primes.end());
```

Beyond Threads

Tasks

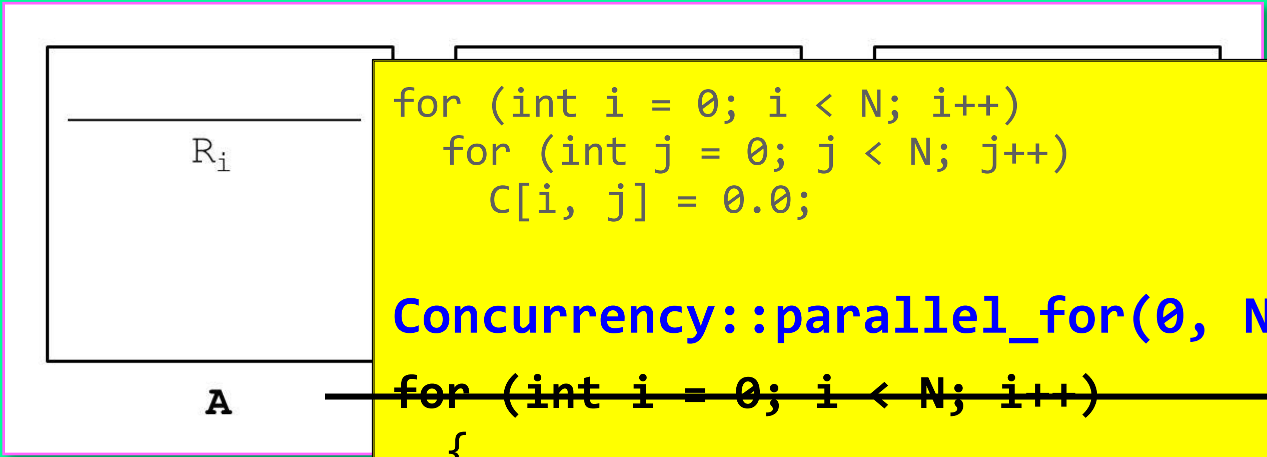
- Tasks are a higher-level abstraction

Task: *a unit of work; an object denoting an ongoing operation or computation.*

  - **Idea**:
    - *developers identify work*
    - *run-time system deals with load-balancing, execution details, etc.*

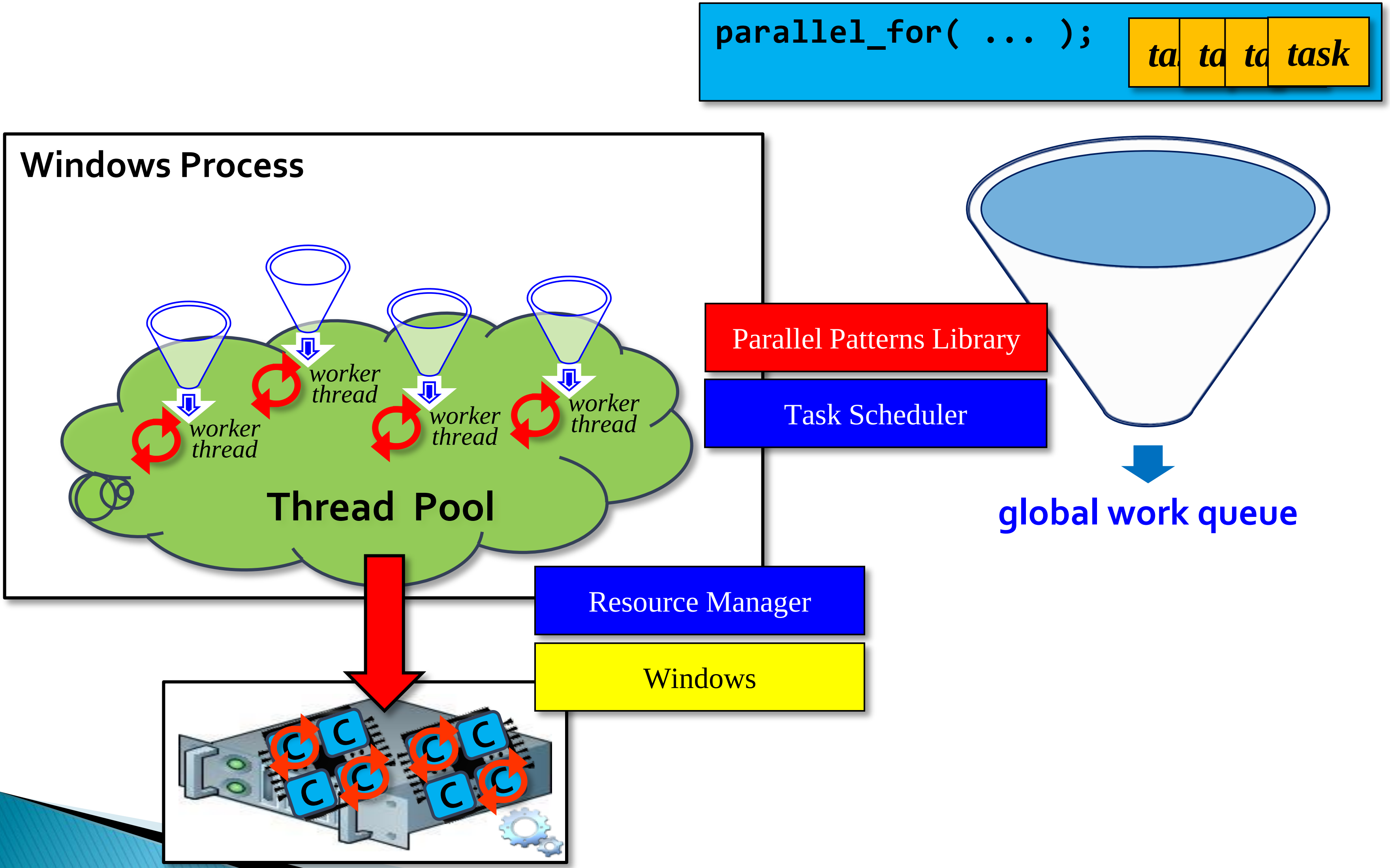
Demo

- Matrix multiply using Microsoft PPL…



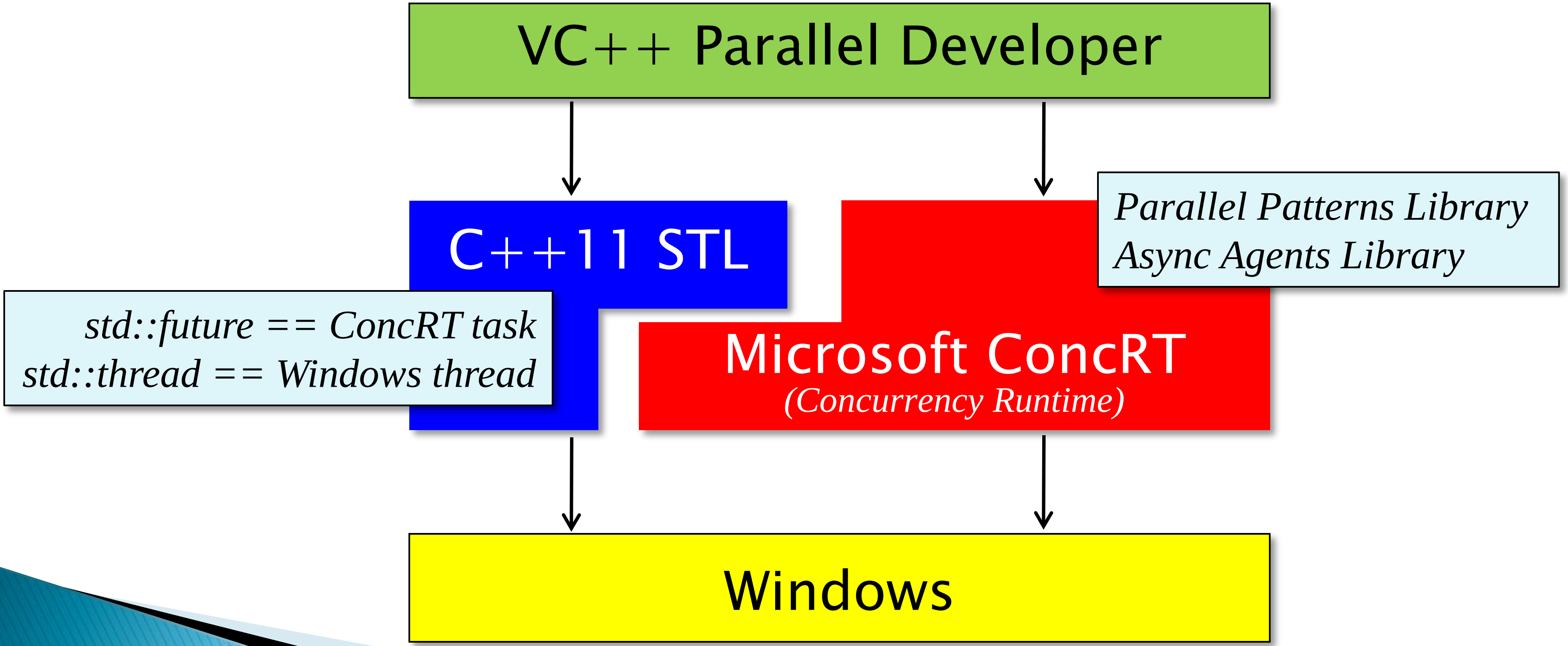
```
for (int i = 0; i < N; i++)
  for (int j = 0; j < N; j++)
    C[i, j] = 0.0;

Concurrency::parallel_for(0, N, [&](int i)
~~for (int i = 0; i < N; i++)~~
  {
      for (int k = 0; k < N; k++)
        for (int j = 0; j < N; j++)
           C[i, j] += (A[i, k] * B[k, j]);
  }
);
```

Execution model



Microsoft ConcRT

- PPL based on Microsoft's ConcRT (*Concurrent Run-Time*)
- C++11 implemented on top of ConcRT



VC++ Parallel Developer

C++11 STL

std::future == ConcRT task
std::thread == Windows thread

Parallel Patterns Library
Async Agents Library

Microsoft ConcRT
(Concurrency Runtime)

Windows

That’s it!

Summary

- **C++11 provides basic concurrency support**
  - *threads*
  - *futures*
  - *locking*
  - *condition variables*
  - *a foundation for platform-neutral parallel libraries*
- **C++11 provides lots of additional features**
  - *lambda expressions, type inference, range-based for, …*
- **Beware of data races**
  - *most common error in parallel programming*
  - *program behavior is undefined*
  - *whenever possible, redesign to eliminate…*

Thank you for attending!

- Joe Hummel, PhD and Jens Mache
  - Emails: jhummel2@uic.edu and jmache@lclark.edu
  - Materials: http://www.joehummel.net/downloads.html

- References:
  - Book: "*C++ Concurrency in Action*", by Anthony Williams
  - Book: "*Multi-Core Programming: Increasing Performance through Software Multi-threading*", by S. Akhter and J. Roberts
  - Talks: Bjarne and friends at "*Going Native 2012*" and "2013"
    - http://channel9.msdn.com/Events/GoingNative/GoingNative-2012
    - http://channel9.msdn.com/Events/GoingNative/2013
  - FAQ: Bjarne Stroustrup's extensive FAQ
    - http://www.stroustrup.com/C++11FAQ.html