Object-Oriented Programming for Scientific Computing

C++11 Multithreading

Ole Klein

Interdisciplinary Center for Scientific Computing
Heidelberg University
ole.klein@iwr.uni-heidelberg.de

7. Juli 2015

C++11: Random Numbers

C++11 provides powerful random number generators that are able to work with different distributions, e.g. uniformly distributed random numbers:

```
#include<iostream>
#include<iomanip>
#include<string>
#include<map>
#include<random>
#include<cmath>

int main()
{
  // Implementation-dependent predefined RNG
  std::default_random_engine e1;
  // Produce uniformly distributed integers between 1 and 6
  std::uniform_int_distribution<int> uniformDist(1,6);
  // Draw a number
  int mean = uniformDist(e1);
  std::cout << "Randomly␣generated␣mean:␣" << mean << std::endl;
```

Normally Distributed Random Numbers

```
// Generator for non-deterministic uniformly distributed random
    numbers
std::random_device rd;
// Alternative RNG based on Mersenne Twister algorithm
std::mt19937 e2(rd());
// Create normal distribution around mean
std::normal_distribution<> normalDist(mean,2);
std::map<int,int> hist;

// Create random numbers and count
for (int n=0; n<10000; ++n)
  ++hist[std::round(normalDist(e2))];

// Print histogram to screen
std::cout << "Normal␣distribution␣around␣mean␣" << mean << ":"
    << std::endl;
for (auto p : hist)
  std::cout << std::fixed << std::setprecision(1) << std::setw(2)
    << p.first << "␣" << std::string(p.second/200,'*')
    << std::endl;
}
```

Own RNG for Uniformly Distributed Integers

```
class RandomInt
{
  public:
    RandomInt(int low, int high, unsigned int seed=0) :
      rand_{std::bind(std::uniform_int_distribution<>{low,high},
          std::default_random_engine{seed})}
    {}
    int operator() () const
    {
      return rand_();
    }
  private:
    std::function<int()> rand_;
};
```

Own RNG for Normally Distributed Doubles

This also works for other number types:

```
class RandomDouble
{
  public:
    RandomDouble(double mean, double stddev, unsigned int seed=0) :
      rand_{std::bind(std::normal_distribution<>{mean,stddev},
          std::mt19937{seed})}
    {}
    double operator() () const
    {
      return rand_();
    }
  private:
    std::function<double()> rand_;
};
```

Application

```
int main()
{
  // Generator for non-deterministic uniformly distributed random
      numbers
  std::random_device rd;
  RandomInt randInt{1,6,rd()};
  int mean = randInt();
  std::cout << "Random␣generated␣mean:␣" << mean << std::endl;
  RandomDouble randDouble{(double)mean,2.,rd()};
  std::map <int,int> hist;

  // Create random numbers and count
  for (int n=0; n<10000; ++n)
    ++hist[std::round(randDouble())];

  // Print histogram to screen
  std::cout << "Normal␣distribution␣around␣mean␣" << mean << ":"
      << std::endl;
  for (auto p : hist)
    std::cout << std::fixed << std::setprecision(1) << std::setw(2)
      << p.first << "␣" << std::string(p.second/200,'*')
      << std::endl;
}
```

Application

```
  // Create random numbers and count
  for (int n=0; n<10000; ++n)
    ++hist[std::round(randDouble())];

  // Print histogram to screen
  std::cout << "Normal␣distribution␣around␣mean␣" << mean << ":"
      << std::endl;
  for (auto p : hist)
    std::cout << std::fixed << std::setprecision(1) << std::setw(2)
      << p.first << "␣" << std::string(p.second/200,'*')
      << std::endl;
}
```

Processes and threads in Unix

- The following information is associated with a process in UNIX:
  - IDs (process, user, group)
  - environment variables
  - directory
  - code
  - registers, stack, heap
  - file descriptors, signals
  - message queues, pipes, shared-memory segments
  - shared libraries
- Each process has its own address space.

- Threads (small-scale processes) exist within a process and share the address space of the main process.
- A thread consists of:
  - ID
  - stack pointer
  - registers
  - scheduling properties
  - signals
- Switching between threads is faster than switching between processes.

C++11 Threads

- POSIX threads (or pthreads) were introduced in 1995 and have been the standard model for threads on UNIX computers for a long time. They are created and managed via a C interface.
- C++11 defines its own threading concept, which basically is a wrapper for POSIX threads. In many cases this makes it easier to write multi-threading programs. The following concepts are supported:

  Threads Thread classes and functions for the rejoining of threads are defined in the header `threads`. This header also contains functions that provide a thread ID and the functionality to detach threads that afterwards run as a separate program (fork).

  Mutual Exclusion The header `mutex` provides classes for mutexes and locks (also recursive variants and ones with timeouts), and additionally functions that check if a lock is available.

C++11 Threads

Condition Variables The header `condition_variable` defines a class that allows the coordination of multiple threads with condition variables.

Futures The header `futures` defines classes and functions whose operations need not be executed immediately, but can be performed asynchronously with the main part of the program. If possible, this is done in parallel, and if not, sequentially at the location where the result is needed.

Atomic Operations The classes from the header `atomic` allow certain operations with integers, Boolean variables and pointers to be performed as an atomic operation.

C++11 Thread Production

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

void Hello(size_t number)
{
    std::cout << "Hello␣from␣thread␣" << number << std::endl;
}

int main()
{
    const int numThreads = 5;
    std::array<std::thread,numThreads> threads;

    // Start threads
    for (size_t i = 0; i < threads.size(); ++i)
        threads[i] = std::thread(Hello, i);

    std::cout << "Hello␣from␣main\n";

    // Rejoin threads, implicit barrier
    for (size_t i = 0; i < threads.size(); ++i)
        threads[i].join();

    return 0;
}
```

Output

```
Hello from thread Hello from thread Hello from thread Hello from thread
    Hello from main
0Hello from thread 213
4
```

Calculation of the Vector Norm

```
#include<array>
#include<vector>
#include<iostream>
#include<thread>

void Norm(const std::vector<double>& x, double& norm, const size_t i, const
    size_t p)
{
  size_t numElem = x.size()/p;
  size_t first = numElem * i + std::min(i,x.size()%p);
  numElem += (i<(x.size()%p)?1:0);
  double localNorm = 0.0;
  for (size_t j=0; j<numElem; ++j)
    localNorm += x[first + j] * x[first + j];
  norm += localNorm;
  // dangerous...
}
```

Calculation of the Vector Norm

```
int main ()
{
  const size_t numThreads = 5;
  const size_t numValues = 1000000;
  std::array<std::thread,numThreads> threads;
  std::vector<double> x(numValues,2.0);
  double norm = 0.0;
  // Create threads
  for (size_t i=0; i<threads.size(); ++i)
    threads[i] = std::thread(Norm,std::cref(x),std::ref(norm),i,numThreads);
  // Rejoin threads with main thread, barrier
  for (size_t i=0; i<threads.size(); ++i)
    threads[i].join();
  std::cout << "Norm is: " << norm << std::endl;
  return 0;
}
```

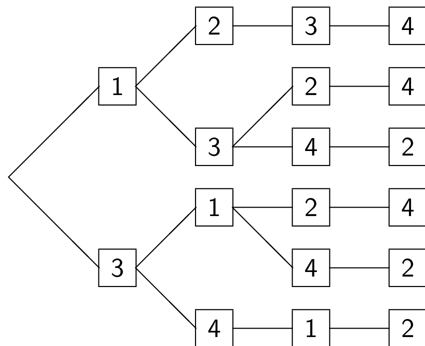
Critical Sections

- A statement of the type $s=s+t$ is not atomic, which means it will not be carried out in one step:

	process Π_1 :		process Π_2 :
1	load s in R1	3	load s in R1
	load t in R2		load t in R2
	add R1, R2, store in R3		add R1, R2, store in R3
2	store R3 in s	4	store R3 in s

- The order of execution of these instructions relative to each other is not fixed.
- This results in an exponentially growing sequence of possible execution orders.

Possible Execution Orders



Result of the calculation

$$s = t_{\Pi_1} + t_{\Pi_2}$$

$$s = t_{\Pi_2}$$

$$s = t_{\Pi_1}$$

$$s = t_{\Pi_2}$$

$$s = t_{\Pi_1}$$

$$s = t_{\Pi_1} + t_{\Pi_2}$$

Only some of the sequences lead to the right result!

Mutual Exclusion

- An additional synchronization is necessary to prevent execution orders that do not provide the correct result.
- Critical sections must be executed under mutual exclusion.
- Mutual exclusion requires:
  - At most one process enters the critical section at a time.
  - There are no deadlocks (i.e. two processes that are waiting on each other).
  - No process waits at a free critical section.
  - When a process enters a critical section, it will leave it successfully in the end.

Mutual Exclusion/Locks

```
#include<iostream>
#include<vector>
#include<thread>
#include<mutex>

std::mutex m;

void e(int& sh)
{
  m.lock();
  sh += 1; // modify shared data
  m.unlock(); // very important
}
```

- C++11 contains the construct `mutex` for this.
- A `mutex` can lock a section, so that it can only be accessed by one process.
- All other processes have to wait in front of the critical section. It should therefore be as small as possible.
- The same mutex must be available in all of the threads.

Lock Guard

```
void f(int& sh)
{
  std::lock_guard<std::mutex> locked{m};
  sh += 1; // modify shared data
}
```

- There are helper classes for releasing a `lock` again (even if an exception is thrown).
- The simplest of these is a `lock_guard`.
- At initialization it gets a `mutex` and locks it, and it releases the lock when the destructor of `lock_guard` is called.
- A `lock_guard` should therefore be defined either at the end of a function or in a separate block to call the destructor as early as possible.

Unique Lock

A `unique_lock` has more functionality than a `lock_guard`. It has functions that lock or release a `mutex` and can test whether a critical section can be entered, so that otherwise something else can be done. A `unique_lock` can also take over an already locked `mutex` or defer locking until it is needed.

```
void g (int& sh)
{
  std::unique_lock<std::mutex> locked{m,std::defer_lock}; // administrate
      mutex, but don't block it
  bool successful = false;
  while (!successful)
  {
    if (locked.try_lock()) // lock mutex if able
    {
      sh += 1; // modify shared data
      successful = true;
      locked.unlock();
      // actually not necessary in this example
    }
    else
    {
      // do something different...
    }
  }
```

Main Program

```
int main ()
{
  const size_t numThreads = std::thread::hardware_concurrency();
  std::vector<std::thread> threads{numThreads};
  int result = 0;
  // start threads
  for (size_t i=0; i<threads.size(); ++i)
    threads[i] = std::thread{e,std::ref(result)};
  // Rejoin threads, implicit barrier
  for (size_t i=0; i<threads.size(); ++i)
    threads[i].join();
  std::cout << "Your␣hardware␣supports␣" << result << "␣threads" <<
      std::endl;

  return 0;
}
```

Calculation of the Vector Norm with a Mutex

```
#include<array>
#include<vector>
#include<iostream>
#include<thread>
#include<mutex>
#include<cmath>

static std::mutex nLock;

void Norm(const std::vector<double>& x, double& norm, const size_t i, const
     size_t p)
{
    size_t numElem = x.size()/p;
    size_t first = numElem * i + std::min(i,x.size()%p);
    numElem += (i<(x.size()%p)?1:0);
    double localNorm = 0.0;
    for (size_t j=0;j<numElem;++j)
        localNorm+=x[first+j]*x[first+j];

    std::lock_guard<std::mutex> block_threads_until_finish_this_job(nLock);
    norm += localNorm;
}
```

Calculation of the Vector Norm with a Mutex

```
int main()
{
    const size_t numThreads = 5;
    const size_t numValues = 10000000;
    std::array<std::thread,numThreads> threads;
    std::vector<double> x(numValues,2.0);

    double norm = 0.0;
    // Start threads
    for (size_t i = 0; i < threads.size(); ++i)
        threads[i] =
            std::thread(Norm,std::cref(x),std::ref(norm),i,numThreads);

    // Rejoin threads
    for (size_t i = 0; i < threads.size(); ++i)
        threads[i].join();

    std::cout << "Norm is: " << sqrt(norm) << std::endl;
    return 0;
}
```

Parallelization of the Sum

- The calculation of the global sum of the norm is not parallel when using a `mutex` for the calculation of $s = s + t$.
- It can be parallelized as follows ($P = 8$):

$$s = \underbrace{\underbrace{\underbrace{s_0 + s_1}_{s_{01}} + \underbrace{s_2 + s_3}_{s_{23}}}_{s_{0123}} + \underbrace{\underbrace{s_4 + s_5}_{s_{45}} + \underbrace{s_6 + s_7}_{s_{67}}}_{s_{4567}}}_{s}$$

- This reduces the complexity from $O(P)$ to $O(\log_2 P)$.

Tree Combine

If we use a binary representation of the process ID, the communication structure is a binary tree:

000
$\sum s_i$

000 | 100
$s_0 + s_1 + s_2 + s_3$ | $s_4 + s_5 + s_6 + s_7$

000 | 010 | 100 | 110
$s_0 + s_1$ | $s_2 + s_3$ | $s_4 + s_5$ | $s_6 + s_7$

000 | 001 | 010 | 011 | 100 | 101 | 110 | 111
s_0 | s_1 | s_2 | s_3 | s_4 | s_5 | s_6 | s_7

Calculation of the Vector Norm with Tree Combine

```
#include <array>
#include <vector>
#include <iostream>
#include <thread>
#include <cmath>

void Norm(const std::vector<double>& x, std::vector<double>& norm,
    std::vector<bool>& flag,const size_t i, const size_t d)
{
    size_t p = pow(2,d);
    size_t numElem = x.size()/p;
    size_t first = numElem * i + std::min(i,x.size()%p);
    numElem += (i<(x.size()%p)?1:0);
    for (size_t j=0;j<numElem;++j)
        norm[i]+=x[first+j]*x[first+j];
```

Calculation of the Vector Norm with Tree Combine

```
    // Tree combine
    for (size_t j=0;j<d;++j)
    {
        size_t m = pow(2,j);
        if (i&m)
        {
            flag[i]=true;
            break;
        }
        while (!flag[i|m]);
        norm[i] += norm[i|m];
    }
}
```

Calculation of the Vector Norm with Tree Combine

```
int main()
{
    const size_t logThreads = 1;
    const size_t numThreads = pow(2,logThreads);
    const size_t numValues = 10000000;
    std::array<std::thread,numThreads> threads;
    std::vector<double> x(numValues,2.0);
    std::vector<bool> flag(numThreads,false);

    std::vector<double> norm(numThreads,0.0);
    // Start threads
    for (size_t i=0; i<threads.size(); ++i)
        threads[i] = std::thread(Norm,std::cref(x),std::ref(norm),
                                 std::ref(flag),i,logThreads);

    // Rejoin threads
    for (size_t i=0; i<threads.size(); ++i)
        threads[i].join();

    std::cout << "Norm is: " << sqrt(norm[0]) << std::endl;
    return 0;
}
```

Condition Synchronization

- Tree combine is a form of condition synchronization.
- A process is waiting until a condition (Boolean expression) is true. The condition is made true by another process.
- Here several processes wait until their flags become true.
- The flags are also called condition variables. Their proper initialization is important.
- We implement the synchronization with *busy wait*. That's probably not a good idea for multi-threading.
- If condition variables are used repeatedly (e.g. when several sums must be calculated in succession), the following rules should be followed:
    - A process which is waiting on a condition variable also resets it again.
    - A condition variable may only be set to true again if it is sure that it has been reset earlier.

Thread Creation with async

The function async allows the easy creation of a thread that performs its calculation without interaction with the main program or other threads. Its result can later be queried using the function get.

```
#include<iostream>
#include<vector>
#include<numeric>
#include<thread>
#include<future>

template<class T> struct Accum  {  // simple accumulator function
    object
  T* b;
  T* e;
  T  val;
  Accum(T* bb, T* ee, T vv) : b{bb}, e{ee}, val{vv} {}
  T operator() () { return std::accumulate(b,e,val); }
};
```

Thread Creation with async

The function `async` allows the easy creation of a thread that performs its calculation without interaction with the main program or other threads. Its result can later be queried using the function `get`.

```
double comp(std::vector<double>& v)
  // spawn many tasks if v is large enough
{
  if (v.size()<10000)
    return std::accumulate(v.begin(),v.end(),0.0);
  std::future<double> f0
      {std::async(Accum<double>{&v[0],&v[v.size()/4],0.0})};
  std::future<double> f1
      {std::async(Accum<double>{&v[v.size()/4],&v[v.size()/2],0.0})};
  std::future<double> f2
      {std::async(Accum<double>{&v[v.size()/2],&v[v.size()*3/4],0.0})};
  std::future<double> f3
      {std::async(Accum<double>{&v[v.size()*3/4],&v[v.size()],0.0})};

  // lots of other code could come here...

  return f0.get()+f1.get()+f2.get()+f3.get();
}

int main()
{
  std::vector<double> blub(100000,1.);
```

Discrete Fourier Transform

- Let (z_i, a_i) be pairs of numbers where a_i is the measured value of an unknown function at the location z_i.
- Goal: calculate interpolation

$$A(z) = \frac{1}{N}\left(f_0 + f_1 z + \cdots + f_{N-1} z^{N-1}\right)$$

- Here we use the N-th roots of unity $z_i = e^{\frac{2\pi i}{N}k}$ as nodes. The coefficients of the interpolation polynomial are then

$$f_k = \sum_{j=0}^{N-1} a_j \cdot e^{-2\pi i \frac{jk}{N}} \quad \text{for} \quad k = 0, \ldots, N-1$$

Fast Fourier Transform

- For even values of N, $N = 2n$, the sum

$$f_k = \sum_{j=0}^{2n-1} a_j \cdot e^{-2\pi i \frac{jk}{2n}} \quad \text{for} \quad k = 0, \ldots, 2n-1$$

can be rearranged into

$$f_k = \sum_{j=0}^{n-1} a_{2j} \cdot e^{-2\pi i \frac{2jk}{2n}} + \sum_{j=0}^{n-1} a_{2j+1} \cdot e^{-2\pi i \frac{k(2j+1)}{2n}}$$

Fast Fourier Transform

- Setting $a'_k = a_{2k}$, $f'_k = f_{2k}$, $a''_k = a_{2k+1}$ and $f''_k = f_{2k+1}$, we have

$$f_k = \sum_{j=0}^{n-1} a'_j \cdot e^{-\frac{2\pi i}{n}jk} + e^{-\frac{\pi i}{n}k} \sum_{j=0}^{n-1} a''_j e^{-\frac{2\pi i}{n}jk}$$
$$= \begin{cases} f'_k + e^{-\frac{\pi i}{n}k} f''_k & \text{if } k < n \\ f'_{k-n} + e^{-\frac{\pi i}{n}(k-n)} f''_{k-n} & \text{if } k \geq n \end{cases}$$

- This means we can solve the problem by calculating two Fourier transforms of length $N/2$.
- This can be applied recursively. Since the Fourier transform of a value is the value itself, $N/2$ sums of two values must be calculated on the lowest level.
- Since at each stage N complex multiplications with a unit root and N complex additions are necessary, the complexity can be reduced from $O(N^2)$ to $O(N \cdot \log(N))$.

Recursive Algorithm in Pseudocode

```
procedure R_FFT(X, Y, N, w)
if (n==1) then Y[0] := X[0];
else begin
  R_FFT(<X[0],X[0],...,X[N-2]>,<Q[0],Q[1],...,Q[N-2]>,N/2,w^2);
  R_FFT(<X[1],X[3],...,X[N-1]>,<T[0],T[1],...,T[N-2]>,N/2,w^2);
  for k:= 0 to N-1 do
    Y[k] := Q[k mod (N/2)] + w^k T[k mod (N/2)];
end R_FFT
```

with $\mathrm{w} = e^{-\frac{2\pi i}{N}}$. But: recursion parallelizes poorly.

Iterative Algorithm in Pseudocode

```
procedure ITERATIVE_FFT(X, Y, N, w)
  r:= log N;
  for i := 0 to N-1 do R[i] := X[i];
  for m := 0 to r-1 do
    for i:= 0 to N-1 do S[i] := R[i];
    for i:= 0 to N-1 do
      /* Let (b_0,b_1,...,b_r-1) the binary representation of i */
        j := (b_0,...,b_m-1,0,b_m+1,...,b_r-1);
        k := (b_0,...,b_m-1,1,b_m+1,...,b_r-1);
        R[i] := S[j] + S[k] x w^(b_m,b_m-1,...,b_0,0,...,0);
    endfor
  endfor
  for i := 0 to N-1 do Y[i] := R[i];
end ITERATIVE_FFT
```

with $w = e^{-\frac{2\pi i}{n}}$

Sequential Implementation of FFT

```
std::vector<std::complex<double>> fft(const
    std::vector<std::complex<double>>& data)
{
    const size_t numLevels=(size_t)std::log2(data.size());
    std::vector<std::complex<double>> result(data.begin(),data.end());
    std::vector<std::complex<double>> resultNew(data.size());
    std::vector<std::complex<double>> root(data.size());
    for (size_t j=0;j<root.size();++j)
        root[j]=unitroot(j,root.size());
    for (size_t i=0;i<numLevels;++i)
    {
        size_t mask=1<<(numLevels-1-i);
        size_t invMask=~mask;
        for (size_t j=0;j<data.size();++j)
        {
            size_t k=j&invMask;
            size_t l=j|mask;
            resultNew[j]=result[k]+result[l]*root[Reversal(j/mask,i+1)*mask];
        }
        if (i!=numLevels-1)
          result.swap(resultNew);
    }
    return resultNew;
}
```

Sorting back Values by Bit Reversal

```
void SortBitreversal(std::vector<std::complex<double>>& result)
{
    std::vector<std::complex<double>> resultNew(result.size());
    const size_t n = std::log2(result.size());
    for (size_t j=0;j<result.size();++j)
      resultNew[Reversal(j,n)]=result[j];
    result.swap(resultNew);
}

void FastSortBitreversal(std::vector<std::complex<double>>& result)
{
  size_t n = result.size();
  const size_t t = std::log2(n);
  size_t l = 1;
  std::vector<size_t> c(n);
  for (size_t q=0;q<t;++q)
  {
    n=n/2;
    for(size_t j=0;j<l;++j)
      c[l+j]=c[j]+n;
    l=2*l;
  }
  std::vector<std::complex<double>> resultNew(result.size());
  for (size_t j=0;j<result.size();++j)
    resultNew[c[j]]=result[j];
  result.swap(resultNew);
}
```

Bit Reversal and Roots of Unity

```
size_t Reversal(size_t k, size_t n)
{
  size_t j=0;
  size_t mask=1;
  if (k&mask)
    ++j;
  for (size_t i=0;i<(n-1);++i)
  {
    mask<<=1;
    j<<=1;
    if (k&mask)
      ++j;
  }
  return j;
}

inline std::complex<double> unitroot(size_t i, size_t N)
{
    double arg = -(i*2*M_PI/N);
    return std::complex<double>(cos(arg),sin(arg));
}
```

Time Measurement

C++11 offers a builtin way to measure time. There are various clocks and data types to store times and timespans.

```
#include<chrono>
#include<iostream>
#include<thread>
using namespace std::chrono;

int main()
{
  steady_clock::time_point start = steady_clock::now();
  std::this_thread::sleep_for(seconds{2});
  auto now = steady_clock::now();
  nanoseconds duration = now-start; // we want the result in ns
  milliseconds durationMs = duration_cast<milliseconds>(duration);
  std::cout << "something took " << duration.count()
            << " ns which is " << durationMs.count() << " ms\n";
  seconds sec = hours{2} + minutes{35} + seconds{9};
  std::cout << "2 h 35m 9s is " << sec.count() << " s\n";
}
```

Main Program for Sequential FFT

```
int main()
{
    const size_t numPoints = pow(2,16);
    std::vector<std::complex<double>> data(numPoints);
    size_t i=1;
    for (auto& x : data)
      x.real(cos(i++));
    auto t0 = std::chrono::steady_clock::now();
    std::vector<std::complex<double>> result1 = fft(data);
    FastSortBitreversal(result1);
    auto t1 = std::chrono::steady_clock::now();
    for (auto& x : result1)
        std::cout << std::abs(x) << " " << x.real() << " " << x.imag()<<
             std::endl;
    std::cout << std::endl << std::endl;

    std::cout << "# Sequential fft took " <<
         std::chrono::duration_cast<std::chrono::milliseconds>(t1-t0).count()
         << " milliseconds." << std::endl;
}
```

Barrier

```
const size_t numThreads=1;//std::thread::hardware_concurrency();

struct Barrier
{
    std::atomic<size_t> count_;
    std::atomic<size_t> step_;
    Barrier() : count_(0), step_(0)
    {}

    void block(size_t numThreads)
    {
        if (numThreads<2)
            return;
        size_t step = step_.load();
        if (count_.fetch_add(1) == (numThreads-1))
        {
            count_.store(0);
            step_.fetch_add(1);
            return;
        }
        else
        {
            while (step_.load() == step)
              std::this_thread::yield();
            return;
        }
    }
};
```

Parallel FFT: Threads

```
Barrier barrier;

void fftthread(const size_t threadNum, std::vector<std::complex<double>>&
    root, std::vector<std::complex<double>>& result,
    std::vector<std::complex<double>>& resultNew)
{
    const size_t N = std::log2(result.size());
    size_t n = result.size()/numThreads;
    const size_t offset = threadNum*n +
        std::min(threadNum,result.size()%numThreads);
    n += (threadNum<(result.size()%numThreads)?1:0);
    for (size_t i=offset;i<offset+n;++i)
        root[i]=unitroot(i,root.size());
    barrier.block(numThreads);
    for (size_t level=0;level<N;++level)
    {
        size_t mask=1<<(N-level-1);
        size_t invMask=~mask;
```

Parallel FFT: Threads

```
        for (size_t i=0;i<n;++i)
        {
            size_t j=offset+i;
            size_t k=j&invMask;
            size_t l=j|mask;
            size_t e = Reversal(j/mask,level+1)*mask;
            std::cout << "#␣j:␣" << j << "␣k:␣" << k << "␣l:␣" << l << "␣e:␣
                " << e << std::endl;
            resultNew[j]=result[k]+result[l]*root[Reversal(j/mask,level+1)*mask]
        }
        barrier.block(numThreads);
        if (threadNum==0)
            result.swap(resultNew);
        barrier.block(numThreads);
    }
    for (size_t j=offset;j<offset+n;++j)
      resultNew[Reversal(j,N)]=result[j];
}
```

Parallel FFT: Threads

```
std::vector<std::complex<double>> fft(const
    std::vector<std::complex<double>>& data)
{
    std::vector<std::complex<double>> result(data.begin(),data.end());
    std::vector<std::complex<double>> resultNew(data.size());
    std::vector<std::complex<double>> root(data.size());
    std::vector<std::thread> t(numThreads);

    for (size_t p = 0;p<numThreads;++p)
        t[p]=std::thread
            {fftthread,p,std::ref(root),std::ref(result),std::ref(resultNew)};

    for (size_t p = 0;p<t.size();++p)
        t[p].join();

    return resultNew;
}
```

Main Program for Parallel FFT

```
int main()
{
    const size_t numPoints = pow(2,2);
    std::vector<std::complex<double>> data(numPoints);
    size_t i=1;
    for (auto& x : data)
      x.real(cos(i++));
    auto t0 = std::chrono::steady_clock::now();
    std::vector<std::complex<double>> result1 = fft(data);
    auto t1 = std::chrono::steady_clock::now();
    for (auto& x : result1)
        std::cout << std::abs(x) << "␣" << x.real() << "␣" << x.imag()<<
            std::endl;
    std::cout << std::endl << std::endl;

    std::cout << "#␣Parallel␣fft␣with␣" << numThreads << "␣threads␣took␣" <<
        std::chrono::duration_cast<std::chrono::milliseconds>(t1-t0).count()
        << "␣milliseconds." << std::endl;
}
```

Further Reading

C++11 Multi-Threading Tutorial
http://solarianprogrammer.com/2011/012/16/cpp-11-thread-tutorial

Overview over all C++11 thread classes and functions
http://en.cppreference.com/w/cpp/thread

Overview over all C++11 atomic operations
http://en.cppreference.com/w/cpp/atomic

Working draft of the C++11 standard (almost identical to the standard, but available for free)
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3376.pdf