High-level parallel programming in C++

Dénes Mátételki

www.emerson.com

March 18, 2012

Table of contents

1. Theory
   - High level vs. low level
   - Algorithms
2. Code samples
   - Map
   - Reduce
   - Sort
3. Final thoughts
   - Grainsize
   - Convolution
   - Summary

Comparison

High level

- Auto scaling-up
- Threadpool handling, load balancing.
- Synchronization and mutexes are handled.

Low level

- Manual thread creation.
- Manual joins and mutex handling.
- Better for event and I/O based threading.
- Compiler and external library independend.

Compared softwares (performance, code complexity)

Used

- Standard c++ (serial examples)
- openMP[1]
- Intel Thread Building Blocks (TBB)[2]
- QtConcurrent[3]

Skipped

- std::thread, std::mutex (c++0x)[6]
- POSIX threads[7]
- QThread[8]

Co-existence[5]

Possible, but the separate threadpools can lead to oversubscription.

Comparison

openMP

- Compiler support needed.
- C, C++, fortran.
- Best for bounded loops.
- No need for big code re-write.
- Hard to debug.
- Managed by a non-profit organization.

Intel TBB

- Object oriented.
- Concurrent data types.
- Parallel algorithms.
- Work stealing: dynamic load sharing.
- Relies heavily on templates.
- Heavy code rewrite is needed.

QtConcurrent

- Object oriented
- Limited number of algorithms.
- ...

Used algorithms for testing

List

- Map - Applies a given function to each element of a container.
- Reduction - Combines the results of sub-parts.
- Sort - Puts elements of a list in a certain order.

Note

- The used container is an `std::vector<float>`
- Container size was 60 million with random floats [1, 1000]
- Execution times are the avareges of 3 executions.
- Used hardware was an Intel Xeon 64-bit machine with 6 cores (12 threads), 3,4Mz.
- Compiled with gcc-4.4 and use flags: `-O3 -ffast-math -fwhole-program -fomit-frame-pointer -march=native -m64`

Serial map

c++ code

```cpp
float modify(float value)
{
  return 13.37 * pow(sqrt(value), log(value));
}


void serialMap(std::vector<float>& data)
{
  for (size_t i = 0; i < data.size(); i++)
    modify(data[i]);
}
```

Note

- "chunksize" equals the size of the data.
- This modify function will be used by the parallel examples too.

openMP parallel map

c++ code

```
void openMpMap(std::vector<float>& data,
               const int numberOfThreads,
               const int chunkSize)
{
  size_t i;

#pragma omp parallel for      \
  default(shared) private(i)  \
  schedule(dynamic, chunkSize) \
  num_threads(numberOfThreads)

  for (i = 0; i < data.size(); i++)
    data[i] = modify(data[i]);
}
```

Note

Making it run in parallel is just a single pragma line.

Intel TBB map

c++ code

```cpp
class itbbMap {
public:

  itbbMap(std::vector<float>& data)
    : m_data(data) {}

  void operator()(const tbb::blocked_range<size_t>& r) const {
    for( size_t i = r.begin(); i != r.end(); i++ )
      m_data[i] = modify(m_data[i]);
  }

private:
  std::vector<float>& m_data;
};


tbb::task_scheduler_init init(NUMBER_OF_THREADS);
itbbMap im(data);
tbb::parallel_for(tbb::blocked_range<size_t>(0, data.size(), CHUNK_SIZE), im);
```

Note

Running a functor on chunks in parallel.

QtConcurrent map

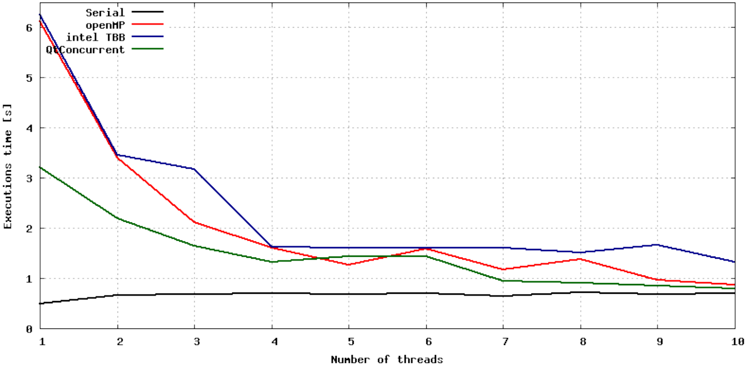
c++ code

```cpp
void QtMap(std::vector<float>& data)
{
  QtConcurrent::blockingMap(data, modify);
}

QThreadPool::globalInstance()->setMaxThreadCount(NUMBER_OF_THREADS);
```

Note

- Chunksize is 1.
- Blocks till the iterator reaches the end.

Map execution times



Note

Serial remained the fastest (memory bound?) - No need to paralellize.

Serial reduce

c++ code

```
float serialReduce(std::vector<float>& data)
{
  float min(FLT_MAX);
  for (size_t i = 0; i < data.size(); i++)
    if (data[i] < min)
      min = data[i];

  return min;
}
```

Note

- Minimum value search.
- Not actually a reduce.
- Following examples will try to achive this too.

openMP reduce

c++ code

```cpp
int openMpReduce(std::vector<float>& data,
                 const int numberOfThreads,
                 const int chunkSize)
{
  size_t i;
  std::vector<float> separate_results(numberOfThreads, FLT_MAX);

#pragma omp parallel \
  default(shared) private(i) \
  num_threads(numberOfThreads)
  {
    int threadId = omp_get_thread_num();

#pragma omp for schedule(dynamic, chunkSize)

  for (i = 0; i < data.size(); i++)
    if (separate_results[threadId] < data[i])
      separate_results[threadId] = data[i];
  }

  float min(FLT_MAX);
  for (i  = 0; i < numberOfThreads; i++)
    if (separate_results[i] < min)
      min = separate_results[i];

  return min;
}
```

Intel TBB reduce

c++ code

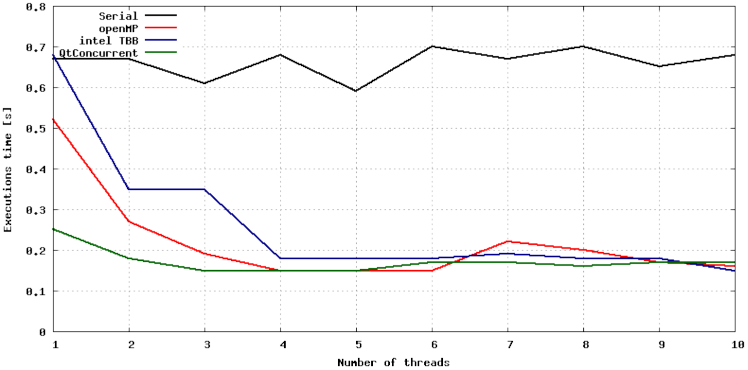
```
class itbbReduce {
    const std::vector<float>& m_data;
public:
    float m_min;

    itbbReduce(std::vector<float>& data) : m_data(data) , m_min(FLT_MAX) {}
    itbbReduce(itbbReduce& other, tbb::split) : m_data(other.m_data), m_min(FLT_MAX) {}

    void operator()(const tbb::blocked_range<size_t>& r) {
      float min = m_min;
      for(size_t i = r.begin(); i != r.end(); i++)
        if ( m_data[i] < min )
          min = m_data[i];

      m_min = min;
    }

    void join(const itbbReduce& other) {
      if ( other.m_min < m_min )
        m_min = other.m_min;
    }
};

itbbReduce mif(data);
tbb::parallel_reduce(tbb::blocked_range<size_t>(0, data.size(), CHUNK_SIZE), mif);
float min = mif.m_min;
```

QtConcurrent reduce

c++ code

```
void findMinimum(const std::vector<float>::const_iterator begin,
                 const std::vector<float>::const_iterator end,
                 float *result)
{
  result = std::min_element(begin, end);
}


float QtReduce(std::vector<float>& data,
               const int numberOfThreads,
               const int chunkSize)
{
  std::vector<float> separate_results(numberOfThreads, FLT_MAX);
  QFutureSynchronizer<void> synchronizer;

  for(int i = 0; i < numberOfThreads; i++)
    synchronizer.addFuture(QtConcurrent::run(findLocalMinimum,
                                             data.begin()+i*chunkSize,
                                             data.begin()+(i+1)*chunkSize,
                                             separate_results.data()+i));

  synchronizer.waitForFinished();

  float min(FLT_MAX);
  findMinimum(separate_results.begin(), separate_results.end(), min);
  return min;
}
```

Reduce execution times



Note

No need for more than 4 threads.

Serial sort

c++ code

```cpp
void serialSort(std::vector<float>& data)
{
  std::sort(data.begin(), data.end());
}
```

Note: quicksort

- Pick a pivot point.
- Partition: Swap elements compared to pivot point.
- Recursively calls itself with the 2 new partitions.

openMP, Intel TBB sort

openMP c++ code

```cpp
#include <parallel/algorithm>

void openMpSort(std::vector<float>& data)
{
  __gnu_parallel::sort(data.begin(), data.end());
}
```

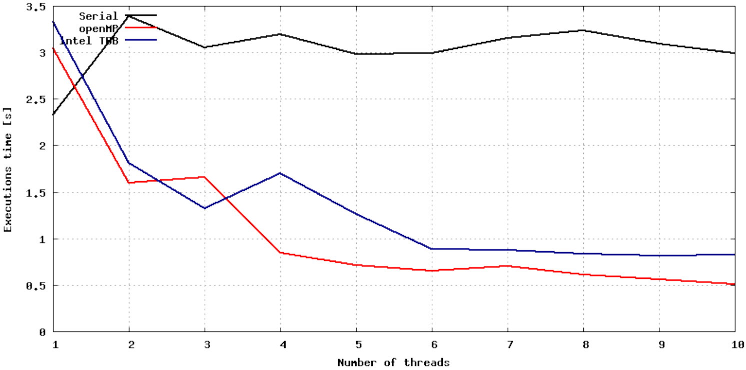
Note

Some algorithms are already rewritten to work in parallel with openMP.

Intel TBB c++ code

```cpp
void itbbSort(std::vector<float>& data)
{
  tbb::parallel_sort(data.begin(), data.end());
}
```

Sort execution times



Note

No need for more than 6 threads.

Custom QtConcurrent sort

c++ code

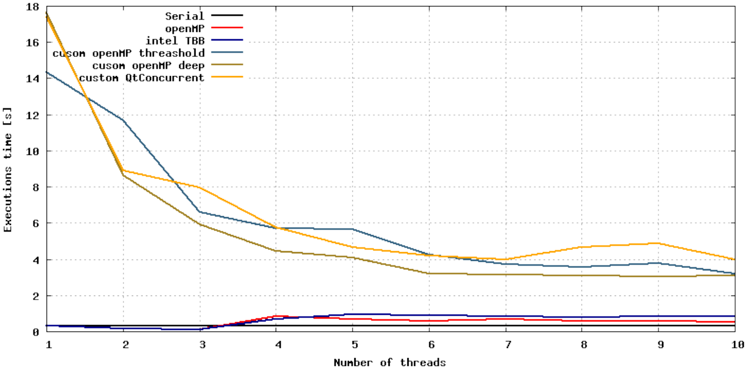
```cpp
template <class SortType>
long QsPartition(SortType outputArray[], long left, long right) { ... }

template <class SortType>
void QsSequential(SortType array[], const long left, const long right) { ... }

template <class SortType>
void QuickSortTask (SortType array[], const long left, const long right, const int deep)
{
  if (left < right) {
    if (deep) {
      const long part = QsPartition(array, left, right);
      QtConcurrent::run(QuickSortTask<SortType>, array, part + 1, right, deep - 1);
      QtConcurrent::run(QuickSortTask<SortType>, array, left, part - 1, deep - 1);
    } else {
      const long part = QsPartition(array, left, right);
      QsSequential(array,part + 1,right);
      QsSequential(array,left,part - 1);
    }
  }
}

void QtSort(std::vector<float>& data)
{
  QtConcurrent::run(QuickSortTask<float>, data.data(), 0, data.size() - 1, 6);
  QThreadPool::globalInstance()->waitForDone();
}
```

Custom openMP sort

c++ code

```
void sample_qsort(float* begin, float* end) { ... }

void sample_qsort_serial(float* begin, float* end) { ... }

void sample_qsort_adaptive(float* begin, float* end, const long nthreshold)
{
  if (begin != end) {
    // parition ...
    if (end - begin + 1 <= nthreshold) {
      sample_qsort_serial(begin, middle);
      sample_qsort_serial(++middle, ++end);
    } else {
#pragma omp task
      sample_qsort_adaptive(begin, middle, nthreshold);
#pragma omp task
      sample_qsort_adaptive(++middle, ++end, nthreshold);
    }
  }
}

void sample_qsort_adaptive(float* begin, float* end)
{
  long nthreshold = ceil(sqrt(end - begin + 1)) / 2;
#pragma omp parallel
#pragma omp single nowait
  sample_qsort_adaptive(begin, end, nthreshold);
}
```

Sort times of custom algorithms



Note

Container size is 6M - miserable...

Two quicksort approach to

Treshold

```
void qsort(float* begin,
           float* end,
           const long nthreshold)
{
  if (begin != end) {
    // parition ...
    if (end-begin+1 <= nthreshold) {
      // serial sort ...
    } else {
      // parallel sort ...
    }
  }
}

long deep =
  ceil(sqrt(end - begin + 1)) / 2;
```

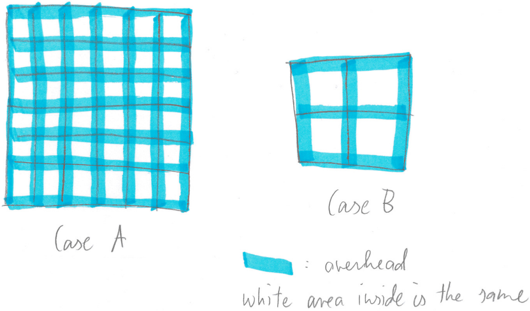
Depth

```
void qsort(float* begin,
           float* end,
           const int deep)
{
  if (begin != end) {
    // parition ...
    if (deep) {
      // serial sort ...
    } else {
      // parallel sort with deep-1
    }
  }
}

long deep = 15;
```

Note

Depth seems simpler yet faster.

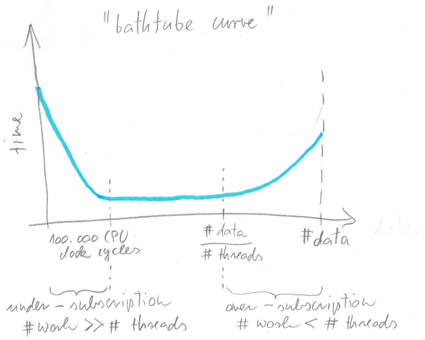
Chunk size



Note

- Unit is loop interaction per chunk. Default value is 1.
- Too small chunks can introduce more overhead than useful work.

Grain size



Note

- Unit is CPU cycles.
- Should be at least 100.000.

Task stealing - Intel TBB

Task stealing

- Each thread has a queue of tasks.
- If a thread has no more tasks then it "steals" from another.
- Think about tasks, not about threads when programming.

Threadpool

A threadpool with a commond concurrent queue of tasks is a common practice in networking servers.

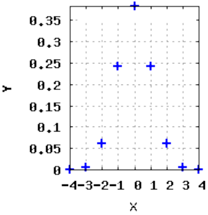
Work stealing

Another implementation is Cilk[4] - where each processor has a stack of frames.

1D gaussian filter

c++ code

```cpp
void serialConvolution(std::vector<float>& output,
                       const std::vector<float>& input,
                       const std::vector<float>& kernel)
{
  // skipping the edges: separate loops, paddings
  // output.size == input.size()-kernel.size()-1;

  for (size_t i = 0; i < output.size(); i++) {
    float sum = 0;
    for (size_t j = 0; j <= kernel.size(); j++)
      sum += input[i+j] * kernel[j];

    output[i] = sum;
  }
}
```



Note

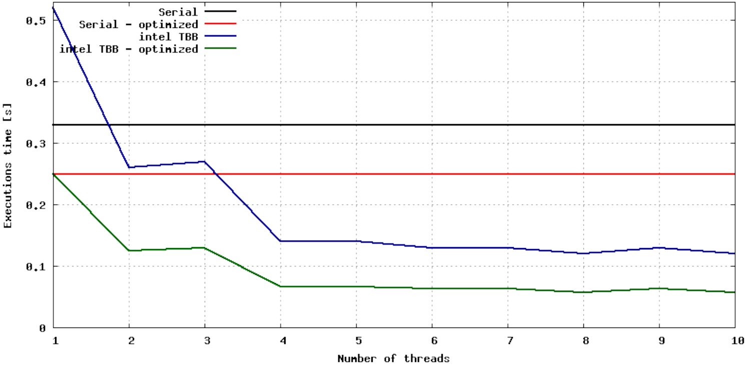
```cpp
float kernel[7] = { 0.06, 0.061, 0.242, 0.383, 0.242, 0.061, 0.06 }
```

Optimized convolution

c++ code

```cpp
void operator()(const tbb::blocked_range<size_t>& r) const
{
  // skipping the edges, shall be done in separate task
  const float* p = &m_input[0] + r.begin();
  float* d = &m_output[0] + r.begin();

  const size_t n = m_kernel.size();
  float k[n]; // pre-read kernel
  float c[n]; // pre-read values
  k[0] = m_kernel[0];
  for (size_t i = 1; i < n; ++i) {
    c[i] = m_input[i-1];
    k[i] = m_kernel[i];
  }

  // chunk size % kernel.size() != 0 should be handled...
  for (size_t i = 0; i < r.size(); i += n) {
    d[i+0] = (c[0] = p[i+0])*k[0]+c[1]*k[2]+c[2]*k[2]+c[3]*k[3]+c[4]*k[4]+c[5]*k[5]+c[6]*k[6];
    d[i+1] = (c[6] = p[i+1])*k[0]+c[0]*k[2]+c[1]*k[2]+c[2]*k[3]+c[3]*k[4]+c[4]*k[5]+c[5]*k[6];
    d[i+2] = (c[5] = p[i+2])*k[0]+c[6]*k[2]+c[0]*k[2]+c[1]*k[3]+c[2]*k[4]+c[3]*k[5]+c[4]*k[6];
    d[i+3] = (c[4] = p[i+3])*k[0]+c[5]*k[2]+c[6]*k[2]+c[0]*k[3]+c[1]*k[4]+c[2]*k[5]+c[3]*k[6];
    d[i+4] = (c[3] = p[i+4])*k[0]+c[4]*k[2]+c[5]*k[2]+c[6]*k[3]+c[0]*k[4]+c[1]*k[5]+c[2]*k[6];
    d[i+5] = (c[2] = p[i+5])*k[0]+c[3]*k[2]+c[4]*k[2]+c[5]*k[3]+c[6]*k[4]+c[0]*k[5]+c[1]*k[6];
    d[i+6] = (c[1] = p[i+6])*k[0]+c[2]*k[2]+c[3]*k[2]+c[4]*k[3]+c[5]*k[4]+c[6]*k[5]+c[0]*k[6];
  }
}
```

Convolution running times



Note

Memory-read optimalization can result the same performance improvements as parallelization.

Things to keep in mind

Checklist

- Pass primitive types by value.
- Pass objects by address.
- Have function-local copies of member variables.
- Avoid to read values multiple times.
- Choose correct chunk size.
- Instead of shared memory, consider reduction.
- Plan datastructures to avoid memory-boundings.*

Things to keep in mind

Checklist

- Pass primitive types by value.
- Pass objects by address.
- Have function-local copies of member variables.
- Avoid to read values multiple times.
- Choose correct chunk size.
- Instead of shared memory, consider reduction.
- Plan datastructures to avoid memory-boundings.*

*data-oriented design[9]

If only someone could tell us more about it...

Links

- openMP.http://openmp.org
- Intel Thread Building Blocks.http://threadingbuildingblocks.org/
- QtConcurrent.http://doc.qt.nokia.com/4.8-snapshot/qtconcurrent.html
- Cilk.http://software.intel.com/en-us/articles/intel-cilk-plus
- Comparison of Intel TBB, openMP and native threads.http://software.intel.com/en-us/articles/intel-threading-building-blocks-openmp-or-native-threads/
- std::thread in C++http://en.cppreference.com/w/cpp/thread
- POSIX threads tutorial.http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html
- Qt threads.http://qt-project.org/doc/qt-4.8/threads.html
- Data oriented design.http://gamesfromwithin.com/data-oriented-design
- LaTeX beamer class for creating presentations.https://bitbucket.org/rivanvx/beamer/wiki/Home
- Gnuplot - An open source plotting software.http://www.gnuplot.info/