Efficient and Robust C++ Scientific Programming and Parallel Computing in Linux

Jianfei Chen

November, 2015
1. Introduction
2. Hints
3. C++11 Features
4. Linear Algebra Libraries
5. Parallel programming with OpenMP
6. Basic usage for Linux
Why C++

- C++ is the fastest: loop is $>10000\times$ faster than Matlab and $>10\times$ faster than Python;
- For ML most time is wasted on waiting program to run: *program* efficiency is more important than *programmer* efficiency.
- C++ is statically typed, avoid a lot of errors in compile time.

Why this presentation

- Most people who think they understand C++ do not understand C++;
- Considerable training is needed to write efficient, readable, and robust code.

Some other good languages for ML

- Matlab: good for prototyping, fast linear algebra, very portable, slow loop.
- Python: moderate fast linear algebra, moderately portable, rich libraries, slow loop.
- Scala: Statically typed, concise grammar, fast loop, Spark, JVM-based.
- Julia: emerging language for Stat and ML.
Some numbers

Jeff Dean’s *the numbers every engineer should know.*

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time</th>
<th>OP/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 cache reference</td>
<td>0.5 ns</td>
<td>2G</td>
</tr>
<tr>
<td>Branch mispredict</td>
<td>5 ns</td>
<td>200M</td>
</tr>
<tr>
<td>L2 cache reference</td>
<td>7 ns</td>
<td>140M</td>
</tr>
<tr>
<td>Mutex lock/unlock</td>
<td>100 ns</td>
<td>10M</td>
</tr>
<tr>
<td>Main memory reference</td>
<td>100 ns</td>
<td>10M</td>
</tr>
<tr>
<td>Compress 1K bytes with Zippy</td>
<td>10,000 ns</td>
<td>100K</td>
</tr>
<tr>
<td>Send 2K bytes over 1 Gbps network</td>
<td>20,000 ns</td>
<td>50K</td>
</tr>
<tr>
<td>Read 1 MB sequentially from memory</td>
<td>250,000 ns</td>
<td>4K</td>
</tr>
<tr>
<td>Round trip within same datacenter</td>
<td>500,000 ns</td>
<td>2K</td>
</tr>
<tr>
<td>Disk seek</td>
<td>10,000,000 ns</td>
<td>100</td>
</tr>
<tr>
<td>Read 1 MB sequentially from network</td>
<td>10,000,000 ns</td>
<td>100</td>
</tr>
<tr>
<td>Read 1 MB sequentially from disk</td>
<td>30,000,000 ns</td>
<td>33</td>
</tr>
<tr>
<td>Send packet CA-&gt;Netherlands-&gt;CA</td>
<td>150,000,000 ns</td>
<td>7</td>
</tr>
</tbody>
</table>
### Some numbers

The numbers every machine learning researcher should know.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time</th>
<th>FLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer arithmetic (A+M)</td>
<td>0.5 ns</td>
<td>2G</td>
</tr>
<tr>
<td>Integer module (A+M)</td>
<td>10 ns</td>
<td>100M</td>
</tr>
<tr>
<td>Floating point multiplication</td>
<td>2 ns</td>
<td>500M</td>
</tr>
<tr>
<td>Floating point division</td>
<td>2 ns</td>
<td>500M</td>
</tr>
<tr>
<td>Random number generation (LC)</td>
<td>9 ns</td>
<td>110M</td>
</tr>
<tr>
<td>Random number generation (MT)</td>
<td>8 ns</td>
<td>120M</td>
</tr>
<tr>
<td>Random number generation (Gaussian)</td>
<td>70 ns</td>
<td>15M</td>
</tr>
<tr>
<td>Matrix multiplication 500 × 500 (for)</td>
<td>520 ms</td>
<td>500M</td>
</tr>
<tr>
<td>Matrix multiplication 500 × 500 (MKL)</td>
<td>25 ms</td>
<td>9.5G</td>
</tr>
<tr>
<td>Matrix multiplication 500 × 500 (MKL-parallel)</td>
<td>7 ms</td>
<td>33G</td>
</tr>
<tr>
<td>Matrix multiplication 10K × 10K (for)</td>
<td>&gt;5 min</td>
<td>-</td>
</tr>
<tr>
<td>Matrix multiplication 10K × 10K (MKL)</td>
<td>188s</td>
<td>10G</td>
</tr>
<tr>
<td>Matrix multiplication 10K × 10K (MKL-parallel)</td>
<td>32s</td>
<td>58G</td>
</tr>
<tr>
<td>500 dim dot product (for)</td>
<td>2µs</td>
<td>430M</td>
</tr>
<tr>
<td>500 dim dot product (MKL)</td>
<td>250ns</td>
<td>3.8G</td>
</tr>
</tbody>
</table>
Do NOT blend C with C++

<table>
<thead>
<tr>
<th>C</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>int a[N];</td>
<td>vector a(N);</td>
</tr>
<tr>
<td>char[]</td>
<td>string</td>
</tr>
<tr>
<td>FILE*</td>
<td>fstream</td>
</tr>
<tr>
<td>memset(a, 0, N*sizeof(int));</td>
<td>fill(a.begin(), a.end(), 0);</td>
</tr>
<tr>
<td>qsort</td>
<td>sort</td>
</tr>
<tr>
<td>int *pa = &amp;a;</td>
<td>int &amp;ra = a;</td>
</tr>
<tr>
<td>int *a = new int[N];</td>
<td>vector&lt;int&gt; a(N);</td>
</tr>
<tr>
<td>delete[] a;</td>
<td>-</td>
</tr>
</tbody>
</table>

**Table:** C expression vs C++ expression
Pass by value

```cpp
void sum(vector<int> a, vector<int> b, vector<int> &c) {
    blah blah
}
sum(A, B, C);
```

What happens: allocate memory for a, allocate memory for b, copy A to a, copy B to b, compute the sum, deallocate a and b

Pass by reference avoids additional copy and malloc

```cpp
void sum(const vector<int> &a, const vector<int> &b, vector<int> &c) {
    blah blah
}
sum(A, B, C);
```
Pass by Reference and std::move

Return value: allocate c, compute \( c = a + b \), copy c to C, deallocate c.

```cpp
vector<int> sum(blah &a, blah &b) {
    vector<int> c; c = a + b; return c;
} C = sum(A, B);
```

Return reference: error

```cpp
vector<int>& sum(blah &a, blah &b) {
    vector<int> c; c = a + b; return c;
} C = sum(A, B);
```

Return move avoids additional copy and malloc

```cpp
vector<int> sum(blah &a, blah &b) {
    vector<int> c; c = a + b; return move(c);
} C = sum(A, B); // C.data is pointed to c.data
```
Always use -O2

Plain, without optimization, *never use*

```
g++ main.cpp -o main
```

With optimization and debug symbol

```
g++ -O2 -g main.cpp -o main
```

Disable optimization, for debugging

```
g++ -O0 -g main.cpp -o main
```
protected:

CMFCMenuBar m_wndMenuBar;
CMFCToolBar m_wndToolBar;
CMFCToolBarImages m_UserImages;
CFileView m_wndFileView;
CClassView m_wndClassView;
COutputWnd m_wndOutput;
CPropertiesWnd m_wndProperties;

protected:

afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
afx_msg void OnWindowManager();
afx_msg void OnViewCustomize();
afx_msg LRESULT OnToolBarCreateNew(WPARAM wp, LPARAM lp);
afx_msg void OnApplicationLook(UINT id);
afx_msg void OnUpdateApplicationLook(CCmdUI* pCmdUI);
afx_msg void OnSettingChange(UINT uFlags, LPCTSTR lpszSection);
DECLARE_MESSAGE_MAP()
protected:
CMFCMenuBar menuBar_;  
CMFCToolBar toolBar_;  
CMFCStatusBar statusBar_;  
CMFCToolBarImages userImages_;  
CFileView fileView_;  
CClassView classView_;  
COutputWnd output_;  
CPropertiesWnd properties_;  

protected:
afx_msg int OnCreate(LPCREATESTRUCT createStruct);  
afx_msg void OnWindowManager();  
afx_msg void OnViewCustomize();  
afx_msg HRESULT OnToolBarCreateNew(WPARAM wp, LPARAM lp);  
afx_msg void OnApplicationLook(UINT id);  
afx_msg void OnUpdateApplicationLook(CCmdUI* cmdUI);  
afx_msg void OnSettingChange(UINT flags, LPCTSTR section);  
DECLARE_MESSAGE_MAP()
Robert Cecil Martin: nowadays HN and other forms of type encoding are simply impediments. They make it harder to change the name or type of a variable, function, member or class. They make it harder to read the code. And they create the possibility that the encoding system will mislead the reader.

Linus Torvalds: Encoding the type of a function into the name (so-called Hungarian notation) is brain damaged—the compiler knows the types anyway and can check those, and it only confuses the programmer.

Steve McConnell: Although the Hungarian naming convention is no longer in widespread use, the basic idea of standardizing on terse, precise abbreviations continues to have value. Standardized prefixes allow you to check types accurately when you’re using abstract data types that your compiler can’t necessarily check.

Bjarne Stroustrup: No I don’t recommend ‘Hungarian’. I regard ‘Hungarian’ (embedding an abbreviated version of a type in a variable name) as a technique that can be useful in untyped languages, but is completely unsuitable for a language that supports generic programming and object-oriented programming — both of which emphasize selection of operations based on the type and arguments (known to the language or to the run-time support). In this case, ‘building the type of an object into names’ simply complicates and minimizes abstraction.
Outline

1 Introduction
2 Hints
3 C++11 Features
4 Linear Algebra Libraries
5 Parallel programming with OpenMP
6 Basic usage for Linux
Random Number Generators: mt19937, minstd_rand;
Distributions: uniform_int, uniform_real, bernoulli, binomial,
negative_binomial, geometric, poisson, exponential, gamma,
weibull, normal, lognormal, chi_squared, cauchy, student_t,
 discrete, etc.

mt19937 generator;
normal_distribution<double> gaussian(0, 1);
cout << gaussian(generator) << endl;
Lambda expression

**Function object**

```cpp
struct GreaterThanThreshold {
    GreaterThanThreshold(int threshold) :
        threshold(threshold) {}
    bool operator () (int x) { return x > threshold; }
    int threshold;
};
GreaterThanThreshold greaterThan3(3);
greaterThan3(4);    // 1
greaterThan3(2);    // 0
```
Lambda expression is a syntactic sugar for functional object. (closure)

```cpp
template <class T>
void trace(T func, string tag) {
    cout << "Entering" << tag << endl;
    func();
    cout << "Leaving" << tag << endl;
}

int a = 0;
trace([&]() { // Capture local variable
    a += 5;
}, "Work");
cout << a << endl;
```

Output: Entering Work, Leaving Work, 5
C++97

```cpp
vector<int> a;
for (vector<int>::iterator it = a.begin(); it != a.end(); it++) {
    cout << *it << endl;
}
```

C++11

```cpp
vector<int> a;
for (auto it = a.begin(); it != a.end(); it++) {
    cout << *it << endl;
}
```
**Range based for loop**

**C++97**

```cpp
vector<int> a;
for (unsigned n=0; n<a.size(); n++) {
    a[n]++;
}
```

**C++11**

```cpp
vector<int> a;
for (auto &elem: a) {
    elem++;
}
```
Range based for loop

C++97

```cpp
vector<vector<vector<int>>> a;
for (unsigned i=0; i<a.size(); i++)
    for (unsigned j=0; j<a[i].size(); j++)
        for (unsigned k=0; k<a[i][j].size(); k++)
            a[i][j][k] = blah blah
```

C++11

```cpp
vector<vector<vector<int>>> a;
for (auto &submatrix: a)
    for (auto &subvector: submatrix)
        for (auto &entry: subvector)
            entry = blah blah
```
How to write robust code

Two common mistakes in C/C++

- Memory leakage;
- Incorrect loop variable names.

Solutions:

- Do not use new and delete, always use containers (vector, etc.);
- Use range based for loop and auto.
Outline

1. Introduction
2. Hints
3. C++11 Features
4. Linear Algebra Libraries
5. Parallel programming with OpenMP
6. Basic usage for Linux
As we have seen before, using linear algebra libraries are faster, and easier to write than for loops. There are two libraries I know

- **Eigen**: widely adopted, easy to install, difficult to learn, rich functionality;
- **Armadillo**: difficult to install, easy to learn, minimal functionality, can link with different implementations (e.g. NVBLAS).
Example

MVNRnd(mu, Sigma). (~/mfs/jianfei/git/c++-programming)

```cpp
#include <iostream>
#include <armadillo>
using namespace std;
using namespace arma;
const int N = 1000;
vec mvnrnd(vec &mu, mat &Sigma) {
    int N = mu.n_rows;
    mat R = chol(Sigma);
    vec z = randn(N, 1);
    return mu + R * z;
}
int main() {
    vec mu = randn(N, 1);
    mat Sigma = randn(N, N);
    Sigma = Sigma.t() * Sigma + 0.05 * eye(N, N);
    vec x = mvnrnd(mu, Sigma);
    cout << x << endl;
}
```
Outline

1. Introduction
2. Hints
3. C++11 Features
4. Linear Algebra Libraries
5. Parallel programming with OpenMP
6. Basic usage for Linux
OpenMP Programming model

Master Thread

Parallel Task I

A
B
C

Parallel Task II

A
B
C
D

Parallel Task III

A
B

Master Thread

Parallel Task I

A
B
C

Parallel Task II

A
B
C
D

Parallel Task III

A
B

OpenMP

- Standard tool for parallel scientific computing;
- Easier to use than raw threads, and has enough functionality;
- Has an GPU counterpart, OpenACC.
A Map-Reduce framework

- We have some global variable $G$;
- We have some data that can be partitioned $d_1, \ldots, d_n$;
- We perform some operation to data in parallel
  \[ r_i \leftarrow f(d_i, G); \]
- We collect the result via an aggregator
  \[ r \leftarrow A(r_1, \ldots, r_n), \text{sequentially}. \]
A simple example

```cpp
vector<int> a;
int sum = 0;
#pragma omp parallel
{
    // Fork threads
    int local_sum = 0;
    #pragma omp for
    for (size_t i=0; i<a.size(); i++) {
        // Partition Jobs
        local_sum += a[i];
    }

    #pragma omp critical
    {
        sum += local_sum;
    }
}
```
#pragma omp for schedule(type, block)

- schedule(static, block) static scheduling;
- schedule(dynamic, block) dynamic scheduling.

Examples

- For highly balanced cheap jobs. schedule(static, 1000)
- For unbalanced expensive jobs. schedule(dynamic, 1)

Useful functions / environment variables

- omp_get_thread_num: number of current thread;
- omp_get_max_threads: maximum number of threads;
- OMP_NUM_THREADS: number of threads.
Batch gradient descent for logistic regression

```cpp
data;
w;
for (int iter=0; iter<ITER; iter++)
{
    #pragma omp parallel
    {
        // Fork threads
        l_gradient(w.size());
        #pragma omp for
        for (size_t i=0; i<data.size(); i++) {
            l_gradient += grad(data[i], w);
        }
        #pragma omp critical
        {
            w += alpha * l_gradient;
        }
    }
}
```

In source code:

```cpp
#include <omp.h>
```

In Makefile: add

```make
-fopenmp
```
Outline

1. Introduction
2. Hints
3. C++11 Features
4. Linear Algebra Libraries
5. Parallel programming with OpenMP
6. Basic usage for Linux
Passphraseless ssh

- Stored in `/ssh`
- `id_rsa`: private key (key)
- `id_rsa.pub`: public key (lock)
- `authorized_keys`: authorized public keys

Example: passphraseless login to juncluser

```bash
ssh-keygen -t rsa  // On your PC
scp ~/.ssh/id_rsa.pub juncluser3:
cat id_rsa.pub >> ~/.ssh/authorized_keys  // On juncluser
// Or
mkdir ~/.ssh
cp id_rsa.pub ~/.ssh/authorized_keys
```
Passphraseless ssh

Setting passphraseless ssh between all juncluster nodes

```
ssh-keygen -t rsa
cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
scp -r ~/.ssh juncluster1:
scp -r ~/.ssh juncluster2:
...```

Make

Make is a logic-based programming language

Basic syntax

- $A = \text{expression}$
- $(A)$

Horn clause

target: dependencies
  command

C++ compilation:

- Compiler: .cpp file $\rightarrow$ .o file
- Linker: .o file $\rightarrow$ executable
A make template

```
CC=g++  # Define some variables
MKLROOT=/home/jianfei/intel/parallel-2015/mkl
THIRD_PARTY=/home/jianfei/mfs/git/third_party_xun2
LDFLAGS=-L${MKLROOT}/lib/intel64 -L${THIRD_PARTY}/lib -lmkl_intel
CPPFLAGS=-std=c++11 -O2 -I${MKLROOT}/include -I$(THIRD_PARTY)/incl

all: main mvnrnd # The first rule is used by default

%.o: %.cpp      # Pattern
  $(CC) $(CPPFLAGS) -c -o $@ $<

main: main.o
  $(CC) $(CPPFLAGS) -o $@ $< $(LDFLAGS)

mvnrnd: mvnrnd.o
  $(CC) $(CPPFLAGS) -o $@ $< $(LDFLAGS)

.PHONY: clean    # clean is not a real rule
clean:
  -rm *.o main mvnrnd
```
Changed some .cpp s

make

Changed some .h s

make clean  # Make will auto-reexecute changed prerequisites
make -j

What happened:

- execute rule “all” by default;
- Prerequisites “main” and “mvnrnd” does not exist;
- Prerequisite of “main”, “main.o” does not exist;
- Prerequisite of “main.o”, “main.cpp” exist, create “main.o”
- Create “main” by “main: main.o”
- ...
Questions ?