Revision control, build tools and performance optimization

Sergei Iskakov

2019 Many Electron Collaboration Summer School Coding Workshop

June 8 2019
Revision control and build tools
Outline

• Revision control systems
  – Centralized repository systems (SVN, CVS, …)
  – Distributed revision control systems (Git)

• Build tools
  – Build tools and project structure
  – Build and dependency management

• Testing and continuous integration
  – Unit test frameworks
  – Continuous integration
Revision control

• Why do we need it?
  – Allows to keep track changes
    • Avoid to create filename mess (e.g. filename.1, filename.2, filename.blablabla ...)
    • Easy to compare different versions
  – Makes it easy to work in a team
    • Conflict resolving
    • Avoid sending files by email
  – Build automation and continuous integration
Which VCS to choose

- Distributed (git)
  - Local repository copy
  - More complicated (local + remote)
  - Provided infrastructure (github, bitbucket)
  - Make easy to perform code review

- Centralized (svn, etc.)
  - Remote repository
  - Simple and easy
  - Needs additional infrastructure
Git. First steps

- Create repository on github
- Init local repository
  - $> git init .
- Add all files to the stage
  - $> git add .
- Commit to the local repository
  - $> git commit -m “blah-blah-blah”
- Setup remote repository
  - $> git remote add origin <remote repository URL>
- Publish local repository remotely
  - $> git push -u origin master
Git. First steps

• Get remote repository
  – $> \text{git clone <Repository URL>}$

✓ Clones remote repository locally
✓ Properly configures remote origin
Git. First step

- File changing
  - check project status
    - $> git status
      - modified: src/blabla.cpp
      - modified: src/blabla.h
  - add changes to stage
    - $> git add <files>
  - commit changes
    - $> git commit -m "bla"
  - publish remotely
    - $> git push
Git. First steps

- **Undo local changes**
  - `$> git status`
  - `$> git checkout -- <Filename>`
  - `$> git status`

On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
Git. First steps
Git. First steps

- Get changes from remote
  - $> \text{git pull}$
Git. Conflicts

• During the work your file was changed remotely. pull/push will failed

```
error: Your local changes to the following files would be overwritten by merge:
  src/blabla.h
Please, To https://github.com/iskakoff/blahblahblah.git
  ! [rejected] master -> master (non-fast-forward)
error: failed to push some refs to 'https://github.com/iskakoff/blahblahblah.git'
```

• Conflict need to be resolved
  – `$> git commit -a -m "blah" && git pull`

```
Auto-merging src/blabla.h
CONFLICT (content): Merge conflict in src/blabla.h
Automatic merge failed; fix conflicts and then commit the result.
```
Git. Conflicts

- "<<<<<<<" start of remote changes
- ======== start of local changes
- >>>>>>>>> end of conflicted part
- Resolve conflict, add, commit and push
Git. Branches

- Create a branch from master for every new feature.
- Implement and test it on a branch.
- Merge a branch into master
Git. Branches

• Create branch
  – `$> git checkout -b <branch name>`

• Setup remote branch
  – `$> git remote add <remote branch name>`

• Publish your branch remotely
  – `$> git push`
Git. Advanced steps

• Does "Master->Branch->Master" pattern is good?
Git. Advanced steps

• Does "Master->Branch->Master" pattern is good?
  – No!
Git. Advanced steps

• Does "Master->Branch->Master" pattern is good?
  – No!
  – Many developer can create a mess in the code
Git. Advanced steps

• Does "Master->Branch->Master" pattern is good?
  – No!
  – Many developer can create a mess in the code
• Pull-request should be used
Git. Advanced steps

• Does "Master->Branch->Master" pattern is good?
  – No!
  – Many developer can create a mess in the code

• Pull-request should be used
  – Changes can be approved by other developer
Git. Advanced steps

• Does "Master->Branch->Master" pattern is good?
  – No!
  – Many developer can create a mess in the code

• Pull-request should be used
  – Changes can be approved by other developer
  – Easy way to force code-review
Git. Advanced steps

• Continuous integration (CI)
  – All machines configuration is different (different library versions, different compilers)
  – What works for you does not suppose work on any different computer
  – Use CI services
    • travis
    • jenkins
    • codeship
Build tools. Examples

> git clone https://github.com/iskakoff/SSS2019.git
Build tools

Before we start the project

• Programming language based choice
  – Language-specific build tool
  – IDE support

• Codebase structure
  – Intra-project dependencies
  – Build tool dependent project structure
Build tools

- **C/C++/Fortran**
  - Make
  - autoconf
  - CMake
    - Classic Unix build tool. Platform dependent
    - Generates Makefile

- **Java**
  - Ant
  - Maven
    - Replaced Make in Java.
    - Full lifecycle support

- **Python**
  - distutils
  - PyBuilder
  - setuptools
Why should I care?
Why should I care?

- Take care of dependencies
- Compiling source code into binary
- Packaging that binary
- Running tests
- Generating documentation
Codebase structure
Codebase structure
Codebase structure
Codebase structure
Codebase structure

- Try to think in advance
- Use best practices
Codebase structure

- Keep the related files together
- Do not spread class implementation into different files
- Extract most related code into library
- Remove unnecessary dependencies
- Separate tests from main code
- Do not add third-party code into your own codebase
- Refactor code structure
Simple build

- Simple compilation example
  - `{CXX} src/blabla.cpp src/blabla2.cpp -l./src -o blabla.exe`
Simple build

- Simple compilation example
  - `${CXX} src/blabla.cpp src/blabla2.cpp -l./src -o blabla.exe`
  - Add library dependence:
    `${CXX} src/blabla.cpp src/blabla2.cpp -l./src -o blabla.exe -L${somelibdir} -ltest`
Simple build

• Simple compilation example
  – `${CXX} src/blabla.cpp src/blabla2.cpp -l./src -o blabla.exe`
  – Add library dependence:
    `${CXX} src/blabla.cpp src/blabla2.cpp -l./src -o blabla.exe -L${somelibdir} -ltest`
  – Add optimization and additional flags…
    mpicc -O3 -g -DNDEBUG -mkl=sequential -qopenmp -march=core-avx2 -g main.cpp.o -o GF2 -L/usr/bin/icpc-15.0-base/compiler/lib -lmpi_usempi08 -lmpi_usempi_ignore_tkr -lmpi_mpifh -lmpi libhdf5_fortran.a libhdf5.a ibhdf5hl_fortran.a libhdf5_hl.a -lz -ldl -lm -limf
Simple build

• Simple compilation example
  – `{{CXX}} src/blabla.cpp src/blabla2.cpp -l./
     src -o blabla.exe`
  – Add library dependence:
    `{{CXX}} src/blabla.cpp src/blabla2.cpp -l./src
     -o blabla.exe -L{{somelibdir}} -ltest`
  – Add optimization and additional flags...
    `mpicc     -O3 -g -DNDEBUG -mkl=sequential -qopenmp -
     march=core-avx2 -g main.cpp.o -o GF2  -L/usr/bin/
    icpc-15.0-base/compiler/lib  -lmpi_usempif08 -
    lmpi_usempi_ignore_tkr -lmpi_mpifh –lmpi libhdf5_fortran.a
    libhdf5.a ibhdf5hl_fortran.a libhdf5 HL.a -lz -ldl -lm -limf`

• Use build tools
Make

• Make it easy to build
• Platform dependent
• Need to know where libraries are located

```
OBJ=src/bla.bla src/bla.bla2.o
INCLUDE="./src"

bla.bla: $(OBJ)
    $(CXX) $(CXX_FLAGS) $(OBJ) -o bla.bla
	src/%.o : src/%.cpp
    $(CXX) $(CXX_FLAGS) -I$(INCLUDE) -c -o @<
```

• It would be good to have Makefiles generated on every platform
Autoconf

- Autoconf is a system for generating a script which will automatically determine the dependency
  - Needs to create project configuration template
  - And Makefile template
Autoconf
Autoconf

- Twice more work to do!
- Not really platform independent
CMake

• Platform-independent
• Can be easy to write and read
• Automatically generates useful make targets (make clean)
• Wide repository to track dependencies

```
cmake_minimum_required(VERSION 2.8)
project(blabla_project CXX)
include_directories(src)
add_executable(blabla.exe src/blabla.cpp src/blabla2.cpp)
```
CMake. Simple build

- **Generate Makefile**
  - cmake `<codebase root>`

- **Run make**
  - make
  - make all
  - make test
  - make world
CMake. Dependencies

• Add some library dependency

```cpp
cmake_minimum_required(VERSION 2.8)
project(blabla_project CXX)
find_package(MPI)
include_directories(${MPI_CXX_INCLUDE_PATH})
include_directories(src)
add_executable(blabla.exe src/blabla.cpp src/blabla2.cpp)
target_link_libraries(blabla.exe PUBLIC ${MPI_CXX_LIBRARIES})
```

• Resolve library and include path

  • c++  -Wl,-search_paths_first -Wl,-headerpad_max_install_names  CMakeFiles/blabla.exe.dir/src/blabla.cpp.o  CMakeFiles/blabla.exe.dir/src/blabla2.cpp.o  -o blabla.exe /usr/local/Cellar/open-mpi/2.0.1/lib/libmpi.dylib
CMake. Testing

- And tests

```cmake
enable_testing()
add_subdirectory(test)
add_test(blabla_test test/blabla_test)
add_test(blabla2_test test/blabla2_test)
```

```cmake
add_executable(blabla_test blabla_test.cpp)
add_executable(blabla2_test blabla2_test.cpp)
```

- Creates target "test"
- Make it easy to automate testing
CMake. Custom libraries

• Not all libraries provides proper cmake find script
• Need to create your own.
  – find_path for includes
  – find_library for libs
  – Try to follow naming conventions
CMake. Summary

• Find all dependencies
• Build all tests
• Compile code in very different environments
• Everything is platform and filesystem independent
Continuous integration

• Build and test your code for different architecture
• Can be easily integrated with version control system
• Each push or pull-request will be tested
• Helps to make code more stable
Github+CI. Travis CI

• Simple integration
  – Sign in to Travis CI with your GitHub account
  – Accept the GitHub access permissions
  – Write and push travis configuration file into your repository

• Simply can test different OS/compilers/library versions
Github+Travis

- Specify language
- OS-version
- Compiler version
- Pre-install libraries
- Build and test
Github+Travis

- Publish commit
- Check for CI report
- Fix tests failure if necessary
- Publish commit
IDEs

- Integrate build, version control, debugger
- Language specific (KDevelop, VS, Clion)
  - Utilize language features
  - Better refactoring support
  - Can have additional plugins
- Configurable (vi, eclipse)
  - More flexibility
  - Sometimes less functionality
IDEs. VCS integration

- Visualize revision history
- Do not need to remember all commands
- Easy to resolve conflict
IDEs. Debugging
IDEs. Debugging

• Easy to setup breakpoints
IDEs. Debugging

• Easy to setup breakpoints
• Easy to go over stacktrace
IDEs. Debugging

- Easy to setup breakpoints
- Easy to go over stacktrace
- Easy to check variable values
Optimization and benchmarking
Intro

• Examples
  https://github.com/iskakoff/SSS2019.git
• Do we need to optimize?
• When do we need to optimize?
• Tools for profiling
Old school optimization tips

• Don't do it!
• If you have to, don’t do it until the end of the project
• Don’t do any more optimizing than you must

We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.

D. Knuth
Don't optimize*

• Processors are getting faster
• Don't waste time on optimization
  • It's much better to make progress in the project even it runs slow
• Optimization makes code worse
  • More unreadable
  • Increase codebase
  • Can introduce unexpected bugs

*From the authors of "Don't split the function"
Don't optimize until the end

- Unnecessary optimization
  - Until the end of the project hotspots are unknown
  - Make everything work
  - Make everything right
- Profile before optimize
  - Find bottlenecks
  - Make it fast
What is about now?

• It's ok to optimize
• It's ok to learn best practices in efficient programming
• Why should you write awful inefficient code?
• Large inefficient code will make you suffer
Early optimization. Dos

• Architectural optimizations
  • Application structure
  • Effective connection between components
• Proper choice of data structures
• Choose proper algorithm
• Reduce Memory Allocation and Copying
• Try to stay at high level
Choice of data structures

• Different data structures has different scaling
• Use the data structure that fits your task
• C++ has more than 15 data containers
  • Sequence containers (vector, list, dqueue, …)
  • Associative containers (map, set, …)
  • Unordered associative containers (unordered map, …)
  • Container adaptors (queue, stack, …)
Choice of data structures. list vs vector

Proper choice of algorithms
Proper choice of algorithms

- Pay attention to Big-O
  - Keep in mind the size of the problem
  - Best-case or worst-case performance
- Don't forget about C

<table>
<thead>
<tr>
<th></th>
<th>(\log_2 n)</th>
<th>(n)</th>
<th>(n \log_2 n)</th>
<th>(n^2)</th>
<th>(2^n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>&lt; 1 (\mu)s</td>
<td>&lt; 1 (\mu)s</td>
<td>&lt; 1 (\mu)s</td>
<td>&lt; 1 (\mu)s</td>
<td>1 (\mu)s</td>
</tr>
<tr>
<td>20</td>
<td>&lt; 1 (\mu)s</td>
<td>&lt; 1 (\mu)s</td>
<td>&lt; 1 (\mu)s</td>
<td>&lt; 1 (\mu)s</td>
<td>1 (\mu)</td>
</tr>
<tr>
<td>30</td>
<td>&lt; 1 (\mu)s</td>
<td>&lt; 1 (\mu)s</td>
<td>&lt; 1 (\mu)s</td>
<td>&lt; 1 (\mu)s</td>
<td>1 s</td>
</tr>
<tr>
<td>40</td>
<td>&lt; 1 (\mu)s</td>
<td>&lt; 1 (\mu)s</td>
<td>&lt; 1 (\mu)s</td>
<td>1.6 (\mu)s</td>
<td>18 min</td>
</tr>
<tr>
<td>50</td>
<td>&lt; 1 (\mu)s</td>
<td>&lt; 1 (\mu)s</td>
<td>&lt; 1 (\mu)s</td>
<td>2.5 (\mu)s</td>
<td>10^{13} years</td>
</tr>
<tr>
<td>100</td>
<td>&lt; 1 (\mu)s</td>
<td>&lt; 1 (\mu)s</td>
<td>&lt; 1 (\mu)s</td>
<td>10 (\mu)s</td>
<td>(\infty)</td>
</tr>
<tr>
<td>1,000</td>
<td>&lt; 1 (\mu)s</td>
<td>1 (\mu)s</td>
<td>10 (\mu)s</td>
<td>1 ms</td>
<td>(\infty)</td>
</tr>
<tr>
<td>10,000</td>
<td>&lt; 1 (\mu)s</td>
<td>10 (\mu)s</td>
<td>130 (\mu)s</td>
<td>100 ms</td>
<td>(\infty)</td>
</tr>
<tr>
<td>100,000</td>
<td>&lt; 1 (\mu)s</td>
<td>100 (\mu)s</td>
<td>2 ms</td>
<td>10 s</td>
<td>(\infty)</td>
</tr>
<tr>
<td>1,000,000</td>
<td>&lt; 1 (\mu)s</td>
<td>1 ms &gt;</td>
<td>20 ms</td>
<td>17 min</td>
<td>(\infty)</td>
</tr>
</tbody>
</table>
Proper choice of algorithms

• Pay attention to Big-O
  • Keep in mind the size of the problem
  • Best-case or worst-case performance
  • Don't forget about "C"
• Analyze whether the current algorithm is appropriate for the problem
Proper choice of algorithms

- Pay attention to Big-O
  - Keep in mind the size of the problem
  - Best-case or worst-case performance
  - Don't forget about "C"
- Analyze whether the current algorithm is appropriate for the problem
- Try to use existent libraries
Early optimization. Don'ts

- Optimization without profiling
  - Wasting extra time on non critical parts
  - Making code more complicated
  - The 90/10 Rule
- Go deep in low level programming
- Don't do simple optimizations (i++ -> ++i)
  - Compiler is smart enough
Low level programming

• Use assembler code inserts
  • Not portable
  • Takes a lot effort
  • Barely better than compiler generated code

• Using architecture specific code
  • MMX/AVX/AVX2/AVX512 etc.
  • See AVX2 examples
Architecture specific code
Architecture specific code

- Take a lot of effort
Architecture specific code

• Take a lot of effort
• Can give decent speed up
Architecture specific code

• Take a lot of effort
• Can give decent speed up
• ....
Architecture specific code

- Take a lot of effort
- Can give decent speed up
- ....
- PROFIT! Good compiler will outperform by orders of magnitude
Profiling
Profiling

• Using timers
  • C++ chrono
  • MPI_Wtime
  • etc.
Profiling

- Using timers
  - C++ chrono
  - MPI_Wtime
  - etc.
- Using special tools
  - gprof
  - instruments
  - VTune
  - Valgrind
Memory

• Memory Is Slow
  • Usually programs are memory bound

• Memory Is Not Accessed in Bytes
  • Read by big portions
  • Compensates slow access

• Memory Is Hierarchical
  • Main memory + levels of cache
  • Memory locality
  • Different CPUs have separate memory
Memory access

- Sequential access
  - Good for prediction
  - Minimize cache misses
- Random access
  - Increase chance of cache misses
  - Still fast if within memory page
- See example memory_access_pattern
Memory access

```
[siskakov@pauli-master build]$ /data/perf/usr/bin/perf stat -e l1-dcache-loads,l1-dcache-load-misses ./memory_access_patterns/Example07 1
0 - 1.00562 s Linear
1 - 1.00264 s Linear
2 - 1.01355 s Linear
3 - 1.11986 s Linear
4 - 1.11992 s Linear

Performance counter stats for './memory_access_patterns/Example07 1':

5 370 680 957  L1-dcache-loads:u
175 809     L1-dcache-load-misses:u  # 0,00% of all L1-dcache hits

5,265163503 seconds time elapsed

[siskakov@pauli-master build]$ /data/perf/usr/bin/perf stat -e l1-dcache-loads,l1-dcache-load-misses ./memory_access_patterns/Example07 2
0 - 1.3776 s Random page
1 - 1.37096 s Random page
2 - 1.38872 s Random page
3 - 1.37121 s Random page
4 - 1.31227 s Random page

Performance counter stats for './memory_access_patterns/Example07 2':

5 371 534 913  L1-dcache-loads:u
7 327 171     L1-dcache-load-misses:u  # 0,14% of all L1-dcache hits

6,824185803 seconds time elapsed

[siskakov@pauli-master build]$ /data/perf/usr/bin/perf stat -e l1-dcache-loads,l1-dcache-load-misses ./memory_access_patterns/Example07 3
0 - 4.93823 s Random heap
1 - 4.93211 s Random heap
2 - 4.93179 s Random heap
3 - 4.93158 s Random heap
4 - 4.93159 s Random heap

Performance counter stats for './memory_access_patterns/Example07 3':

10 720 717 046  L1-dcache-loads:u
1 406 948 659     L1-dcache-load-misses:u  # 13,12% of all L1-dcache hits

24,669064750 seconds time elapsed
```
Memory allocation
Memory allocation

• Contiguous allocation
  • Increase performance for dense operation
  • Can be used directly in HPC libraries
Memory allocation

- Contiguous allocation
  - Increase performance for dense operation
  - Can be used directly in HPC libraries
- Noncontiguous allocation
  - Introduce storage overhead
  - Usually needs more cache updates
Memory allocation

- Contiguous allocation
  - Increase performance for dense operation
  - Can be used directly in HPC libraries
- Noncontiguous allocation
  - Introduce storage overhead
  - Usually needs more cache updates
- See example memory
Memory allocation

- Contiguous allocation
  - Increase performance for dense operation
  - Can be used directly in HPC libraries
- Noncontiguous allocation
  - Introduce storage overhead
  - Usually needs more cache updates
- See example memory
- Try to use linear algebra
Algorithm optimization

• Detect hotspots
  • Use profiler
• Check inter-dependency
  • Different part can be independent
  • Rearrange data processing
• Use better algorithm
• Simple change can lead to huge speedup
  • See loop example
Algorithm optimization

- Profiling
Algorithm optimization

• Simple processing rearrangement
Algorithm optimization

• Profiling

• >1.5x performance improvement
Algorithm optimization

- Better algorithm
- $O(n^2) \rightarrow O(N \log N)$
- 2.5x speedup
Thank you