LLNL-PROC-800725

Transitioning the Scientific Software Toolchain to Clang/LLVM

M. M. Pozulp, S. A. Dawson, R. C. Bleile, P. S. Brantley, M. S. McKinley, M. J. O'Brien, D. F. Richards

January 8, 2020

European LLVM Developers' Meeting
Paris, France
April 6, 2020 through April 7, 2020

Transitioning the Scientific Software Toolchain to Clang/LLVM

M. Pozulp[1, 2], S. Dawson[1], R. Bleile[1, 3], P. Brantley[1], M. S. McKinley[1], M. O'Brien[1], and D. Richards[1]

[1]Lawrence Livermore National Laboratory, Livermore, CA USA
{pozulp1, dawson6, bleile1, brantley1, mckinley9, obrien20, richards12}@llnl.gov

[2]University of California, Davis, CA USA

[3]University of Oregon, Eugene, OR USA

***Abstract*—For the past 25 years, many of the largest scientific software applications at Lawrence Livermore National Laboratory (LLNL) have used the Intel C/C++ compiler (icc/icpc) to compile the executables provided to users on x86. This spring 2020, the Monte Carlo Transport Project will release our first executable compiled with clang, which builds 25% faster and runs 6.1% faster than icpc. The poster accompanying this paper will describe the challenges of switching toolchains and the resulting advantages of using a clang/LLVM toolchain for large scientific software applications at LLNL.**

I. Background

In 1995 at Lawrence Livermore National Laboratory (LLNL), we observed that the Intel C compiler (icc) generated the best code on x86 and so we used it to compile the executables that we provided to users running on x86. Today our applications look very different than they did in 1995. At least 12 of our 15 largest applications [1] are written in a mix of C, C++98, and C++11, so we use the Intel C++ compiler (icpc) instead of icc. Together these 12 applications consist of about 5 million lines of code. This poster will focus on Mercury, a 3D massively parallel Monte Carlo particle transport application containing approximately 300,000 lines of code [2].

II. Motivation

Our motivation to investigate clang was based off an analysis of the relative advantages of icpc (TABLE I) and clang (TABLE II) for our applications. *We are not dropping support for icpc.* We will continue to build with icpc for testing and we will support users who want to build Mercury with icpc.

TABLE I
Advantages of staying with icpc

- no work required to continue using the same compiler
- compatibility with Intel developer tools
- compatibility with Intel libraries

III. Performance

We provide users with two Mercury executables on x86, a MPI-only executable and a MPI+OpenMP executable. The performance analysis in this paper is limited to the former. For a discussion of the latter, please see Section IV.

TABLE II
Advantages of switching to clang

- generates faster code
- faster compilation
- better C++ standard support
- can target non-x86 platforms like ARM, Power, and GPUs
- better documentation [3] [4]
- compatibility with open source developer tools
- compatibility with open source libraries
- open source
  - no vendor lock-in
  - we can fix bugs ourselves
  - we can understand what the compiler is doing
  - external collaborators can build our code and reproduce our results
  - no denial-of-service due to license server outage
  - no license fee
- big community
  - active mailing lists (llvm-dev, cfe-dev, etc.)
  - annual developer meetings in San Jose and Europe
  - more users often means fewer bugs and more features
- innovative toolchain technologies like ThinLTO [5] and sanitizers [6]
- better support for popular gcc language extensions [7]
- our vendor partners' compilers are increasingly clang-based (e.g. Cray)

We used LLNL's RZGenie cluster [8] which contains 48 dual-socket nodes with Intel Xeon E5-2695 v4 "Broadwell" CPUs [9]. The nodes are connected by an Intel Omni-Path interconnect and they run RHEL-based TOSS3 Linux.

TABLE III shows the times to compile the Mercury executable. TABLE IV shows the compile and link flags. A serial compile is 15% faster with clang, but clang takes almost twice as long to link. The **real build**, a hybrid distributed- and shared-memory parallel build that our developers use to build an executable (i.e. compile and link using 4 nodes), is 25% faster with clang. We are using ThinLTO [5] to do whole-program optimization at link time with lld [10]. Without ThinLTO, clang links in 4.3s, a 35% speedup over icpc.

TABLE III
Compile Time Comparison

	icpc 19.0.4	**clang 9.0.0**	**icpc/clang**
serial compile	17m37s	15m16s	1.15
link time	5.8s	11.3s	0.51
real build	1m21s	1m5s	1.25

TABLE IV
COMPILE AND LINK FLAGS

	icpc 19.0.4	**clang 9.0.0**
CXXFLAGS	-g -O2 -std=c++11 -no-ftz -fp-model precise -fp-model source -nolib-inline -prec-div -prec-sqrt -unroll-aggressive -funroll-loops -diag-disable cpu-dispatch -finline-functions -gxx-name=/path/to/gcc-7.1.0 -wd10397 -wd2650 -qoverride-limits -fno-builtin-malloc -fno-builtin-calloc -fno-builtin-realloc -fno-builtin-free -ip -no-ipo	-g -O2 -std=c++11 -ffp-contract=off -flto=thin -fstandalone-debug
LDFLAGS	-	-flto=thin -fuse-ld=lld

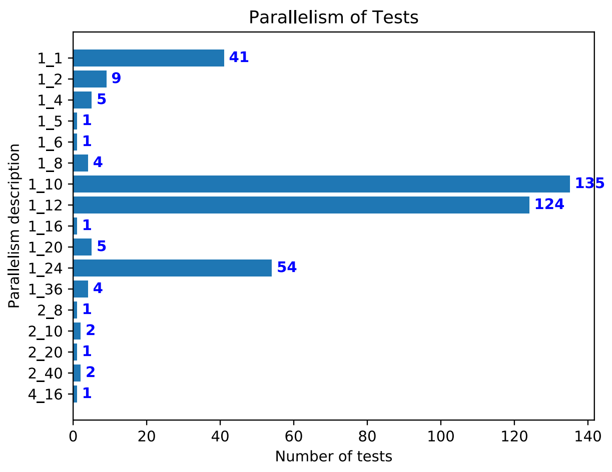

Fig. 1. This plot shows the parallelism distribution of the 391 tests. 41 of the tests were serial, meaning they ran on 1 node using 1 task (1_1) while the remaining 350 tests were parallel. The vertical axis label **X_Y** means a total of Y tasks ran across X nodes.

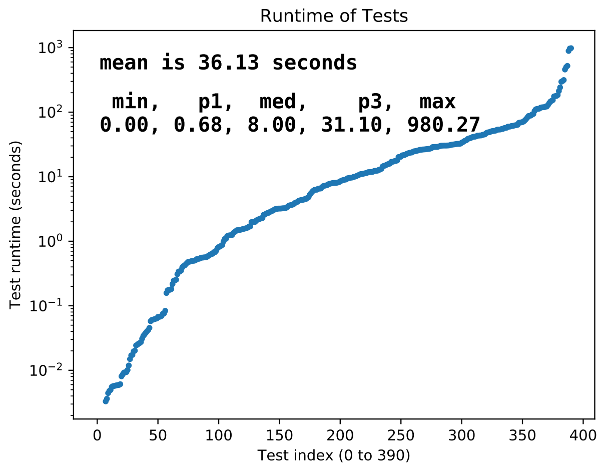

Fig. 2. This plot shows the runtime distribution of the 391 tests. The shortest tests ran for 3 milliseconds and the longest test ran for 16 minutes 20 seconds. **p1** is the 25th percentile and **p3** is the 75th percentile. The total was 14125 seconds or 3 hours, 55 minutes, and 25 seconds.

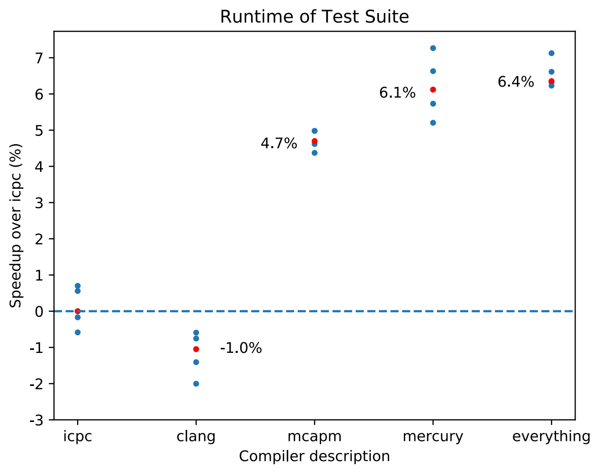

Fig. 3. This plot shows the percent speedup of 5 test suite runs for 5 executables compared to the fastest run with icpc (through which the horizontal line is drawn). The median of the 5 runs is labeled and highlighted in red. **icpc** is the executable compiled with icpc 19.0.4. **clang** is the executable compiled with clang 9.0.0. **mcapm** is the clang-compiled executable with ThinLTO used for Mercury and MCAPM [11]. **mercury** is the clang-compiled executable with ThinLTO used for Mercury. **everything** is the clang-compiled executable with ThinLTO used for everything.

For the runtime comparison we ran a suite of 391 tests. The runtime of a test is the time spent tracking particles (for a brief description of the Monte Carlo particle transport algorithm in Mercury see Appendix A). Fig. 1 shows the parallelism distribution of the 391 tests.

Fig. 2 shows the runtime distribution of the 391 tests. The 391 tests varied from a minimum of 3 milliseconds to a maximum of 16 minutes.

We ran the suite five times using the same executable in order to quantify the run-to-run variability. The sum of the runtimes for the 391 tests for the fastest run of the 5 runs with the icpc-compiled executable was 14125 seconds or 3 hours, 55 minutes, and 25 seconds. This is the runtime to which all other runs of the test suite are compared.

Fig. 3 compares the runtime of our test suite for icpc and clang, as well as clang with and without ThinLTO. We found that without ThinLTO, clang was 1% slower than icpc, but with ThinLTO clang was 6.1% faster than icpc. We did not see a significant difference when we used ThinLTO for Mercury versus ThinLTO for everything (i.e. Mercury and all libraries). However, we did see a slowdown when we used it for Mercury and MCAPM [11], which we are still trying to understand. For a list of libraries used in Mercury see TABLE V.

IV. Discussion

The 25% build time speedup and 6.1% runtime speedup are great, and we may be able to improve on 6.1% if we can understand and tweak the ThinLTO settings. For example, `llvm/lib/Transforms/IPO/FunctionImport.cpp` has a value `import-instr-limit` with default value 100 and description “Only import functions with less than N instructions.” If we increase this parameter will we get more

TABLE V
LIBRARIES LINKED INTO MERCURY

LLNL Libraries:

Name	Version
timers	2.1.27
ft_hash	2.1.27
overlink	4.7.1
mcapm	2.7.12
mpism	1.1.19
tdf	2.3.58
silo	4.10.3
sha	1.0
memusage	2.1.27
physicsutils	0.0.190904
gidiplus	3.17.115
nuclear	r184
rng	3.0
yogrt	-

Third-Party Libraries:

Name	Version
mvapich2	2.3.1
python	2.7.14
tcmalloc	2.6.1
hdf5	1.8.10p1
zlib	1.2.9
szip	2.1

cross-translation-unit inlining and an even better runtime? We also still need to try PGO.

We are running all our tests with Address Sanitizer, but with Leak Sanitizer turned off. We want to use Thread Sanitizer and Archer [12] to fix problems with our MPI+OpenMP executable which appear when compiling with clang at `-O2` but not icpc.

Finally, we want to use clang on Sierra, a hybrid POWER9 CPU and NVIDIA Volta V100 GPU machine sited at LLNL [13] and currently the number two supercomputer on the TOP500 [14]. We already use clang to target POWER9, but we use nvcc to compile our CUDA code for the V100. We would prefer to use clang for all the reasons described in [15].

V. ACKNOWLEDGEMENTS

The title for this document was inspired by a presentation at the 2019 LLVM Developers' Meeting, "Transitioning the Networking Software Toolchain to Clang/LLVM" [16].

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

APPENDIX A
MONTE CARLO PARTICLE TRANSPORT

Monte Carlo particle transport has a single high-level **while** loop which performs most of the useful work (see Algorithm 1). Calculations spend 90% of the runtime in this loop. Calculations spend 50% of the runtime in the "cycleTracking" phase of the loop which contains about 100,000 lines of reachable code (see Algorithm 2). For this analysis, we compare time in "cycleTracking" instead of total runtime. That means we ignore input parsing, allocation, I/O, and most other syscalls.

Algorithm 1 Monte Carlo particle transport

```
Read in nuclear data
while cycles remain do
    cycleInit
    cycleTracking
    cycleFinalize
end while
```

Algorithm 2 Particle tracking ("cycleTracking")

```
Compute distance to cell boundary, $d_{cell\_boundary}$
Sample distance to collision, $d_{collision}$
Compute distance to census, $d_{census}$
Compute min ($d_{cell\_boundary}$, $d_{collision}$, $d_{census}$)
case cell_boundary: move particle across cell boundary
case collision: move particle to collision site, sample rxn
case census: move particle to census site, save particle
```

REFERENCES

[1] Llnl computer codes. https://wci.llnl.gov/simulation/computer-codes. Accessed: 2019-12-29.
[2] Mercury. https://wci.llnl.gov/simulation/computer-codes/mercury. Accessed: 2019-12-29.
[3] Clang: a c language family frontend for llvm. https://clang.llvm.org. Accessed: 2019-12-29.
[4] The llvm compiler infrastructure. https://llvm.org. Accessed: 2019-12-29.
[5] Thinlto: Scalable and incremental lto. http://blog.llvm.org/2016/06/thinlto-scalable-and-incremental-lto.html. Accessed: 2019-12-29.
[6] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference (USENIX ATC 12)*, pages 309–318, Boston, MA, 2012. USENIX.
[7] Clang language extensions. https://clang.llvm.org/docs/LanguageExtensions.html. Accessed: 2019-12-29.
[8] Rzgenie. https://hpc.llnl.gov/hardware/platforms/RZGenie. Accessed: 2019-12-29.
[9] Intel® xeon® processor e5-2695 v4. https://ark.intel.com/content/www/us/en/ark/products/91316/intel-xeon-processor-e5-2695-v4-45m-cache-2-10-ghz.html. Accessed: 2019-12-29.
[10] Lld - the llvm linker. http://lld.llvm.org/. Accessed: 2019-12-29.
[11] Jim A. Rathkopf. Mcapm: All particle method generator and collision package. Technical Report UCRL-ID-112310, Lawrence Livermore National Laboratory, Livermore, California, 1992.
[12] S. Atzeni, G. Gopalakrishnan, Z. Rakamaric, D. H. Ahn, I. Laguna, M. Schulz, G. L. Lee, J. Protze, and M. S. Müller. Archer: Effectively spotting data races in large openmp applications. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 53–62, May 2016.
[13] Sudharshan S. Vazhkudai, Bronis R. de Supinski, Arthur S. Bland, Al Geist, James Sexton, Jim Kahle, Christopher J. Zimmer, Scott Atchley, Sarp Oral, Don E. Maxwell, Veronica G. Vergara Larrea, Adam Bertsch, Robin Goldstone, Wayne Joubert, Chris Chambreau, David Appelhans, Robert Blackmore, Ben Casses, George Chochia, Gene Davison, Matthew A. Ezell, Tom Gooding, Elsa Gonsiorowski, Leopold Grinberg, Bill Hanson, Bill Hartner, Ian Karlin, Matthew L. Leininger, Dustin Leverman, Chris Marroquin, Adam Moody, Martin Ohmacht, Ramesh Pankajakshan, Fernando Pizzano, James H. Rogers, Bryan Rosenburg, Drew Schmidt, Mallikarjun Shankar, Feiyi Wang, Py Watson, Bob Walkup, Lance D. Weems, and Junqi Yin. The design, deployment, and evaluation of the coral pre-exascale systems. *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, pages 52:1–52:12, 2018.
[14] Top500 november 2019. https://www.top500.org/lists/2019/11/. Accessed: 2019-12-29.
[15] Jingyue Wu, Artem Belevich, Eli Bendersky, Mark Heffernan, Chris Leary, Jacques Pienaar, Bjarke Roune, Rob Springer, Xuetian Weng, and Robert Hundt. Gpucc: An open-source gpgpu compiler. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO '16, page 105–116, New York, NY, USA, 2016. Association for Computing Machinery.
[16] Transitioning the networking software toolchain to clang/llvm. https://www.youtube.com/watch?v=0rICCQb_mRg&feature=youtu.be. Accessed: 2019-12-29.