

APPENDIX A

ARTIFACT DESCRIPTION APPENDIX: IMPROVING THE I/O PERFORMANCE AND MEMORY USAGE OF THE XOLOTL CLUSTER DYNAMICS APPLICATION

A. Abstract

This appendix describes how to obtain, configure, build, and run the Xolotl code used to generate the performance and scalability plots shown in our poster.

B. Description

1) Check-list (artifact meta information):

- **Program:** Xolotl
- **Compilation:** Intel 17 or GCC 7 C++ compiler
- **Data set:** Benchmark inputs provided in Xolotl repository
- **Run-time environment:** Linux-based software stack
- **Hardware:** Cray XC30 used, but Linux cluster would suffice
- **Experiment workflow:** Run in parallel on 1, 2, or 4 nodes
- **Publicly available?:** Yes

2) *How software can be obtained:* The Xolotl source code is publicly and freely available from its GitHub repository (<https://github.com/ORNLFusion/xolotl>).

The I/O performance/scalability experiments compared commit cfa35c1c from the h5-pcr branch against commit d75a62c9 of the master branch for the improvements to the startStop1D monitor function. The tridyn monitor improvements then compared that h5-pcr version against commit e2209067 of the same branch, and then the single-file, single-open comparison compared that version against commit be4f87fd of the h5-pcr branch.

The memory usage reduction experiments compared the commit 0b330184 from branch master-pcr (labeled “opt” in the poster plot) against commit 598048c8 of the master branch (labeled “master” in the poster plot).

3) *Hardware dependencies:* To evaluate our I/O-related modifications, we collected timing data from Xolotl runs on the Eos Cray XC30 system of the Oak Ridge Leadership Computing Facility (OLCF). Because this system’s compute nodes use commodity CPUs, do not contain accelerators, and run a Linux-based software stack, we believe that the positive impact of our changes would also be demonstrated (perhaps to a lesser degree due to changes in the interconnection network and parallel filesystem) if run on a traditional Linux cluster.

To identify the location of Xolotl’s peak memory usage, we used a Linux server with a large amount of memory per core (512 GiB for 48 cores total, or >10 GiB per core). A system with a smaller amount of memory per core could have been used, but it would impose a greater limit on the size of the reaction network (the V independent variable from our peak memory usage plot) that one could use with the original version of the code.

4) *Software dependencies:* On the OLCF Eos system, we used the system’s default Intel 17 compilers to build Xolotl and its library dependencies. On the large-memory Linux system, we used the GCC 7.3 compilers.

Xolotl depends on the PETSc, HDF5, and Boost libraries. Source code and in some cases precompiled binaries for these libraries are freely and publicly available from their web sites (<https://www.mcs.anl.gov/petsc/>, <https://www.hdfgroup.org>, and <https://www.boost.org>, respectively).

Xolotl also depends on an MPI implementation. On the OLCF Eos system, we used the MPI implementation provided by the default “PrgEnv-intel/5.2.82” module. On the large-memory Linux system we built our own OpenMPI 3.1.0 library with source code obtained freely from the Open MPI web site (<https://www.open-mpi.org/>).

As described in our poster, our Xolotl memory usage reduction modifications depend on an alternative application programming interface (API) function to the PETSc library. Because the pre-built PETSc library on OLCF Eos does not contain this API function (named `DMDASetBlockFillSparse`), we built our own PETSc library using sources obtained from the PETSc Git repository. The code we built was obtained by checking out commit e5cfcc5c (repository tag “v3.9.3”) and added our alternative API function implementation to the `include/petscdmda.h` and `src/dm/impls/da/fdda.c` files. Versions of the “master” branch starting with commit 09e28618 also contain the modified API function and could be used instead of a patched version of the v3.9.3 sources.

On OLCF Eos we used the system’s pre-built HDF5 libraries from the system’s “cray-hdf5-parallel/1.10.0.1” module. On the large-memory Linux system, we built our own HDF5 library, version 1.10.2, with support for parallel file access.

We used our own build of the unmodified Boost version 1.67.0 sources on both systems.

5) *Datasets:* The input files used for our experiments are available in the Xolotl Git repository (see Section A-B2), h5-pcr branch as of commit dc49b262, in the `benchmarks/XolEdEos-benchmark` directory.

For the peak memory usage measurements, we used a slightly modified version of the `params_PSI2.txt` benchmark input available in the master branch of the Xolotl repository in the `benchmarks` directory. The modifications were to:

- change the `perfHandler` setting from “dummy” to “std” so that the program would collect and print performance data using its built-in performance data collection infrastructure
- add a line `rng=print 1536` so that program runs would always use the same random number seed, and print that seed for verification
- vary the size of the reaction network the program would build by changing the maximum number of cluster vacancies V (fourth value in the `netParam` parameter)

C. Installation

The procedure for configuring, building, and installing Xolotl is documented at <https://github.com/ORNLFusion/xolotl/wiki>.

The configuration step uses CMake scripts to identify the location and capabilities of Xolotl’s software dependencies.

After configuring, we use `make xolotl` to build the Xolotl executable without building Xolotl’s entire test suite.

Because the OLCF Eos system does not allow running executables from a user’s home directory, installation involves copying the Xolotl executable and a small number of shared libraries that our dynamically-linked executable requires to the system’s scratch filesystem that *is* accessible by software running on the system’s compute nodes. The shared objects we copy are: our build of PETSc (see Section A-B4), some Boost libraries (chrono, timer, filesystem, and system), and the two GCC compiler runtime shared libraries that the executable requires (libgcc_s and libstdc++).

the peak memory usage would shift from Xolotl initialization to the first time step when the PETSc library does on-demand initialization of its internal data structures.

D. Experiment workflow

For our I/O-related performance measurements, we ran Xolotl on the OLCF Eos system from a batch job. Our batch script loads the modules needed to run, adds the location of shared libraries we copied to the scratch filesystem (see Section A-C) to the LD_LIBRARY_PATH environment variable, and runs the program using

```
aprun -n $num_processes -N 32 -S 16 -j2 \
  -cc cpu $PWD/xolotl params_0.700000.txt
```

where the `$num_processes` variable was computed from the number of nodes requested by the batch job. These flags to the program launcher indicate to enable HyperThreading, create 32 processes per node, and use process affinity to tie processes to specific cpu cores (actually hardware thread contexts given use of HyperThreading).

For the memory usage experiments, we used an interactive shell to run a single Xolotl process under control of the Valgrind Massif tool. Because the version of Xolotl used creates the same reaction network in each process, examining memory usage of a single process’ program initialization provided the same insight as would running a multi-process instance.

E. Evaluation and expected result

When a Xolotl run does a graceful shutdown, it prints the performance data it collected using its built-in performance data collection infrastructure. In particular, it prints the minimum, maximum, average, and standard deviation of the values it collected across all processes in the run. For our poster comparisons, we only considered the maximum timings reported because those timings would determine the overall program’s run time. The data points shown in the poster represent the average of those maximum values across at least three runs with the same software version/input configuration.

If one were to reproduce our experiments, we would expect that they would observe a non-negligible performance improvement from our parallel write optimizations, though the amount of improvement would be determined by the characteristics of the interconnection network used in their parallel system and the amount of contention experienced on that network while the program was running.

For the peak memory usage measurements, we would expect that people reproducing our experiments would see the peak memory usage of the modified version of the code be dramatically less than the original version, and that the location of