Milestone Report

Yifei Chen (yifeiche)
Peiyuan Liao (peiyuanl)

Detailed Project Schedule

(11-30) - (12-2)

- Ensure that all five implementations (top-down, bottom-up, middle-out, OLS, WLS) are correct (Yifei)
    - Manually run small test cases: e.g. TourismSmall (a-la Hyndman), or hand crafted
- Benchmark performance in both speed, memory and metric (Yifei + Peiyuan)
    - Yifei: implement MSE, MAE and SMAPE
    - Peiyuan: ensure fair measurements against Nixtla, optionally capture more metrics (e.g. MPI communication)
- Implement further DP communication reduction technique for OLS & WLS (Peiyuan)

(12-3) - (12-6)

- Clean up data preparation notebook & write about LHTS hierarchical data format (Peiyuan)
    - To be incorporated into the final report
- Implement non-matrix communication reduction / speed-up technique for bottom-up, top-down and middle-out (Peiyuan)
- Implement sparse matrix optimization for all methods (Peiyuan)
- Set-up and benchmark on GHC single-node (MPI + Eigen + pybind + OpenMP) (Yifei)
- Begin writing performance report (Yifei)
- Cleaning up Wikipedia data for forecasting (Peiyuan)

(12-7) - (12-10)

- Implement non-matrix communication reduction / speed-up technique for OLS & WLS (Peiyuan)
- Set-up and benchmark on GHC multi-node (MPI + Eigen + pybind + OpenMP) (Yifei)
- Cleaning up SUMO / Google Books for forecasting (Peiyuan)
- Continue writing performance report (Yifei)
- Start writing final report (Yifei + Peiyuan)

(12-11) - (12-14)

- Explore more communication reduction & speedup techniques across the board (Peiyuan)
- Keep writing final report (Yifei + Peiyuan)
- Start making poster (Yifei)
- Prepare Prophet + LHTS (our project) demo (Peiyuan)

    - Determine whether to use Google Books, TourismSmall (a-la Hyndman) or Wikipedia
    - Determine whether to build a front-end or just present the pip package

(12-15) - (12-17)

- Complete final report (Yifei + Peiyuan)
- Clean up code artifact (Peiyuan)
- Wrap up posters (Yifei + Peiyuan)
- Finish the project website (Yifei)
- Wrap up Prophet + LHTS demo (Peiyuan)

Summary of Current Progress

We identified a gap in existing research in hierarchical time-series forecasting, namely the inefficiency in matrix-based reconciliation methods in non-uniform data access latency and tight memory limit scenarios. To this end, we first examined existing datasets in literature, then constructed three new benchmarks based on novel datasets that are orders of magnitude larger than the existing counterparts. We were able to stress test a well-known reconciliation implementation in Python (Nixtla).

We then implemented a preliminary version of LHTS (large-scale hierarchical time-series reconciliation), a Python-pip package that lets data scientists easily access a parallel system that reconciles large hierarchical time-series datasets in single CPU-node, multi-process and multi CPU-node settings. The core artifact is implemented in C++ with MPI, OpenMP and LAPACK/BLAS (through the Eigen library), connected to Python via pybind11 and built by setuptools and cmake. This way, users can easily call highly-optimized routines leveraging SIMD / threaded-parallelism / message-passing with numpy arrays they've produced in the same Python process, which tends to be an existing forecast produced by other frameworks like PyTorch, sklearn, Prophet or Nixtla. We have so far implemented naive single-process and MPI solutions for five of the most popular reconciliation algorithms (bottom-up, top-down, middle-out, OLS, WLS), and have also implemented a simple reduction technique leveraging data parallelism similar to the ones used in deep learning. Preliminary benchmarks indicate that our single process C++ implementation is 25x faster than Nixtla in top-down and middle-out methods, and our data-parallel communication reduction technique is ~25% faster than its naive MPI counterpart.

Existing Goals & Deliverables Post-mortem

5.1 Plan to Achieve

75% A modular framework that is easy to install and useful to a data scientist in performing large-scale forecast reconciliation.

This is on track to completion. We have a pip package that invokes a cmake routine to compile the C++ source code and package it into a python module. Any data scientist can easily integrate our solution into their daily workflow by invoking their python script using mpirun, and import our C++ modules through pybind11.

75% Naive single-process, matrix-based solution for all 5 methods. Latency is recorded as gathering individual forecasts from all processes.

This is complete.

75% Naive MPI-based solution for all 5 methods (top-down, bottom-up, middle-out, OLS and WLS). Expect to achieve at 4x less memory footprint and 5x less communication overhead over naive single-process solution. 100% MPI + OpenMP + LAPACK solution for all 5 methods (top-down, bottom-up, middle-out, OLS and WLS). Expect to achieve a further 20% to 80% speedup.

We have implemented this before the milestone deadline. The tricky thing in the claim is that since we are using Eigen to leverage a lot of the SIMD & threaded parallelism, we have not observed as much memory footprint reduction or speed-up. In fact, on our 12-core, hyper threaded machine, the MPI solution is slower than the naive single-process solution as different MPI processes fight for threads during their optimized matrix Eigen routines. Our new estimates on memory footprint & speed-up are much more conservative.

It is trivial to implement a solution that leverages no parallelism at all, so we will still use Nixtla (which uses numba & numpy behind the scenes who also have highly optimized BLAS/LAPACK routines) and our naive-single process method as baselines.

The silver lining in this experience is that we discovered Eigen (which had good compatibility with Python, decent performance and sparse matrix routines), which saves us a decent amount of time having to manually implement SIMD & OpenMP optimizations.

100% Communication reduction techniques for top-down, bottom-up and middle-out approaches. Expect to achieve at least 5x speedup over the matrix-based solution.

We are en route to deliver this. We have implemented a simple data-parallel solution but the speedup is less than expected. We realized that we need to implement some non-matrix based solutions for more speedup.

75% A simple benchmark system for single-node and multi-node CPU reconciliation performance evaluation: memory, communication, latency and error metrics (SMAPE + MAPE).

We realized that it’s much easier to benchmark most of these performance metrics in Python via pybind11, in addition to making it fair against Nixtla.

75% Data loader for time-series forecasts and sample program for producing forecasts for the benchmark datasets. The data loader simluates loading serialized forecasts into each process.

We realized that writing the data loaders are non-trivial and we may be better off just implementing pybind11, which is what we did.

75% Performance study mentioned above.

We are en route to deliver this.

5.1.1 Demo (Plan)

100% A Jupyter notebook producing Prophet forecasts for Wikipedia dataset and serializing them into per-process files.

We are en route to deliver this.

100% An example program leveraging our system to simulate loading per-process forecasts and perform reconciliation, and calculates performance.

We are en route to deliver this; we realized that this could be a Python script run through mpirun and mpi4py. We could then export results to the Jupyter notebook for analysis.

5.2 Hope to Achieve

125% Local OLS and WLS reconciliation method, with discussion on impact in error metric (e.g. SMAPE and MAPE of new approach vs old, as well as increase in performance).

We realized that OLS and WLS just need good work distribution of the G matrix computation, which is very similar to the techniques employed in bottom-up, top-down and middle-out. Therefore, it makes less sense to implement a “local” OLS / WLS reconciliation.

125% Global communication reduction techniques for OLS and WLS reconciliation methods. Expect to achieve at least 2x speedup over the matrix-based solution.

See above. Our speedup estimates are more conservative now.

125% Connecting the system to Python via pybind11 and mpi4py, so that the per-process forecasts could be readily reconciled as soon as it is produced from another library in memory.

We realized that this is actually easier to do than the dataloader, so we pivoted to finish this first.

5.2.1 Demo (Hope)

125% A Python program launched through mpi4py that trains the Prophet model on individual time series in Wikipedia.

We are en route to deliver this.

125% Within the same program, calls the Python binding to our system to reconcile forecasts

We are en route to deliver this.

125% Compute and report error metrics

We are en route to deliver this.

New List of Goals & Deliverables

- A novel large-scale hierarchical time-series forecasting benchmark, consisting of two datasets: M5 retail forecasting (sparse) and Wikipedia time-series (dense & flat)
    - Hierarchy comes in a condensed matrix format
    - Sample trace for reconciliation is generated by Prophet
    - 125%: include SUMO + Google Books 1-gram time-series (sparse & tall)
- A parallel system (LHTS) performing large-scale hierarchical time-series forecast reconciliation.
    - Available as a pip package to install
    - Performs both single-process and multi-process forecast reconciliation
    - Supports all five popular methods (top-down, bottom-up, middle-out, OLS, WLS)
    - 125%: explore more reconciliation methods
    - Multi-process methods are optimized for performance and communication reduction, and works for multi-node settings
- A detailed performance analysis comparing our parallel system with popular implementation (Nixtla)
    - Compare latency, performance, communication
    - Compare under different MPI & OpenMP settings, on single-node and multi-node hardware
    - 125%: reason performance with respect to NUMA on GHC
- A demonstration of usage of LHTS in daily data science workflow
    - Launch a multi-process forecasting job in Python with Prophet
    - Readily reconcile forecast
    - Compute performance
    - 125%: launch jobs through Ray

Poster Session Plan

- Our poster will consists of a background on hierarchical time-series forecasting, the theoretical set-up, explaining our experimental framework, and a detailed performance study of our solution under multiple hardware settings and benchmark datasets
- We will also introduce how our benchmark datasets are created, and how open-sourcing may have bigger impacts to the community.
- We will also prepare a demo (to be determined if it includes a UI/UX component), in which a data scientist could launch a multi-process Python script to train a forecasting model on a large dataset over many processes, and readily reconcile them using our LHTS package.

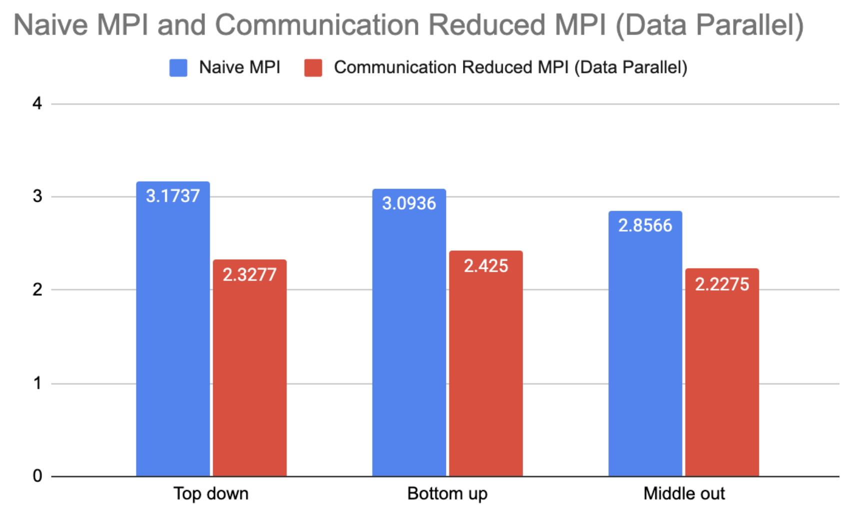
Preliminary Results

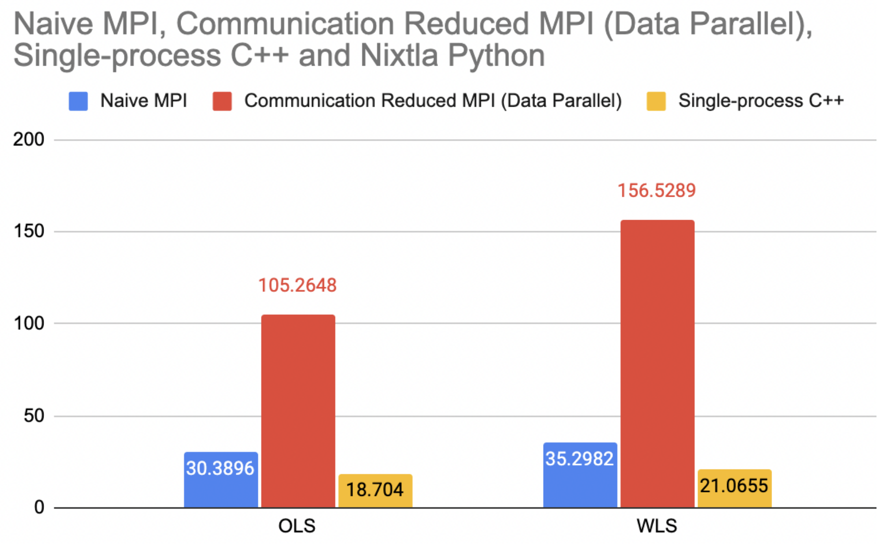
We benchmarked our various solutions against Nixtla on a reduced set of M5 dataset on hobbies (6218 series, 4 levels of hierarchy, 5650 leaves). Below is the raw-output of the CPU platform:

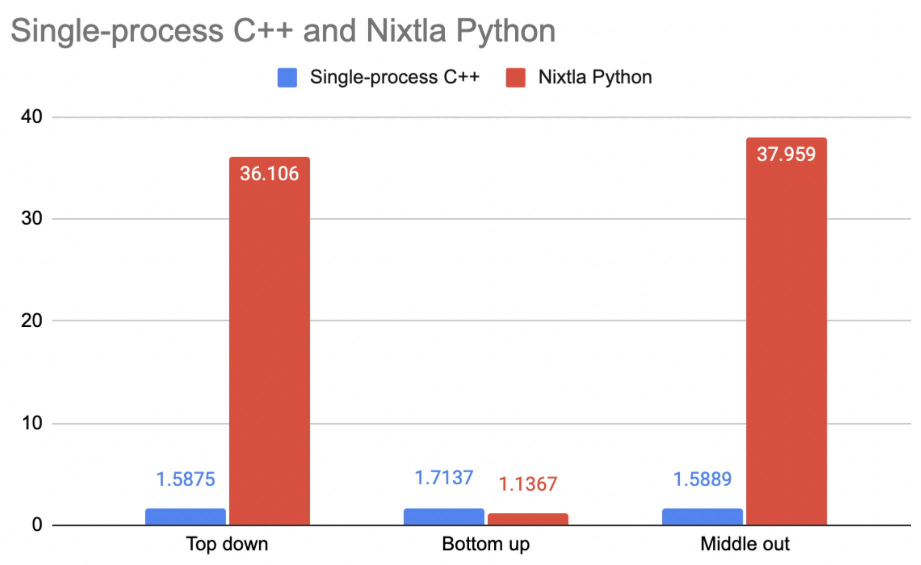
```
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              24
On-line CPU(s) list: 0-23
Thread(s) per core:  2
Core(s) per socket:  12
Socket(s):           1
NUMA node(s):        2
Vendor ID:           AuthenticAMD
CPU family:          23
Model:               1
Model name:          AMD Ryzen Threadripper 1920X 12-Core Processor
Stepping:            1
CPU MHz:             2018.330
CPU max MHz:         3500.0000
CPU min MHz:         2200.0000
BogoMIPS:            6999.17
Virtualization:      AMD-V
L1d cache:           32K
L1i cache:           64K
L2 cache:            512K
L3 cache:            8192K
NUMA node0 CPU(s):   0-5,12-17
NUMA node1 CPU(s):   6-11,18-23
Flags:               fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx
fxsr sse sse2 ht syscall nx mmxext fxsr_opt pdpe1gb rdtscp lm constant_tsc rep_good nopl nonstop_tsc
cpuid extd_apicid amd_dcm aperfmperf pni pclmulqdq monitor ssse3 fma cx16 sse4_1 sse4_2 movbe
popcnt aes xsave avx f16c rdrand lahf_lm cmp_legacy svm extapic cr8_legacy abm sse4a misalignsse
3dnowprefetch osvw skinit wdt tce topoext perfctr_core perfctr_nb bpext perfctr_llc mwaitx cpb hw_pstate
```

sme ssbd sev ibpb vmmcall fsgsbase bmi1 avx2 smep bmi2 rdseed adx smap clflushopt sha_ni xsaveopt xsavec xgetbv1 xsaves clzero irperf xsaveerptr arat npt lbrv svm_lock nrip_save tsc_scale vmcb_clean flushbyasid decodeassists pausefilter pfthreshold avic v_vmsave_vmload vgif overflow_recov succor smca

The system also comes with 6x16GB DDR4 SDRAM. All metrics are measured in seconds.







As shown above, our single process C++ implementation is 25x faster than Nixtla in top-down and middle-out methods, and our data-parallel communication reduction technique is ~25% faster than its naive MPI counterpart. The slowdown in chart #2 in OLS & WLS in DP is due to the recomputation of the G matrix for each process, where contention also happens with repeated computation (since in naive MPI, only rank 0 computes G).

Concerns

- Debugging the nuanced interaction between Eigen, OpenMP, MPI and pybind11 may be hard as our codebase gets more complex, and could slow us down.
- Multi-node testing on GHC may be harder to set-up: very hard to ensure that no one is using >1 node(s).
- Implementing a non-matrix based solution for the reconciliation methods could prove to be hard to make correct and generate no speedup.
- Contention between MPI processes and their OpenMP threads could prove to be a very hard task to performance optimize in our set-up.
- Large-scale hierarchical time-series benchmark datasets have been proven very hard to curate and set-up. We spent 50%+ of our time reading about the Wikipedia time-series, the M5 time-series, and SUMO (which turned out to not natively have a time-series associated with them, so we had to turn to the Google Books N-gram dataset and used the 1-gram). Generating the raw forecast with a naive algorithm (e.g. Prophet) has also proven to be quite time-consuming. We could simply run out of time compiling SUMO & Wikipedia.