San José State University

CMPE 275

Enterprise App Development

Section 47

Spring 2024
Instructor: John Gash

Team:
Steven Chang
Brahma Teja Chilumula
Avi Ajmera
Hasan Mhowwala

Mini Project - 2, HPC Simulation

Due: Monday, April 8th

Overview

This mini-project served as a practical exploration into High Performance Computing (HPC) concepts, providing us with hands-on experience with parallel data extraction techniques and technologies. Our implementation process involved considerable trial and error, with numerous iterations and discarded code along the way.

Scrapped Iterations:

We started our implementation with how the dataset was stored/organized on disk/storage. We experimented with altering the datasets organization, further segmenting the data by regions (determined by lat, lon), stations, and parameters (not all at once). The files that composed the original dataset were already relatively small, and that these approaches lead to excessive file fragmentation.

Our initial design had both master and worker processes in C++. Frontend initiation in Python requirement, we later converted the C++ master node logic into Python, using mpi4py. In one iteration, we had the master process load the entire data set into memory. We used OpenMP to parallelize the loading. The airnow dataset is relatively small (<40mb) so it could reasonably be stored in memory. Additionally, the dataset included a lot of duplicate data in the form of the duplicate station information. In this iteration, each Stations's data was only stored once, in the form of an Object that had pointers to all of the individual pollutant data points associated with it.

The idea behind this was that it would allow us to send all of the data required for specific processing tasks to workers without the need for excessive MPI communication. The issue with this was that it led to the master process handling much more work than the worker nodes as it had to formating and serialize the data before it could be sent, and this process was more complex than the tasks the worker nodes were performing on said data. Additionally, this method was needlessly complex, requiring many different serialization methods for the various formats of data being sent.

Final Iteration:

In our final iteration, we used a scatter/gather methodology for data dispersion. The master process divides up the files in the dataset, assigning files to different worker processes for processing. We distributed the files with MPI_Send and MPI_Recv calls manually rather than using MPI's scatter, gather methods. (*See main.py, line 18 - distribute_file() & nodes.hpp, line 254 - recieveString()*)

The worker processes loads files from disk and performs data analysis/processing. The processing we perform is rather simple/basic, simply extracting aggregate statistics from the chunk of data. We collect this data both per parameter and in aggregate for each file. The various aggregate processing tasks were performed in parallel one each node via OpenMP parallel for loops. (*See nodes.hpp, line 162 - calculateAQIStats() & nodes.hpp, line 199- calculateConcentrationStats()*)

Each worker process also compared their data to the pre-fire baseline. The baseline data files were loaded into a shared memory segment so that every worker process on the same node could access it. Upon initiation, each worker attempts to create the shared memory segment, or open it if it already exists. We utilized a semaphore for synchronization of the shared memory to ensure that the data was already loaded into shared memory before trying to access it. (*See nodes.hpp, line 33 - loadBaseLine()*)

Challenges:

The biggest challenge we faced was designing data types for communication and processing. Because any data being sent needs to be a raw byte stream, everything needs to be serialized/deserialized. We utilized json as a serialization format, using boost property trees to serialize and deserialize data (*See record.hpp*). This approach was not ideal as it made creating complicated data structures for communication very difficult. Any change to a data's structure required editing each of its serialization/deserialization functions, and the lack of compiler errors for trying to access missing fields of a JSON string made development difficult.

We briefly looked into protocol buffers for serialization because they can compile the data structures into C++ objects that the compiler could recognize, but because it would require an external library we weren't sure if we could utilize them.

Running Program:
Prerequisites:

- Clang-17, earlier versions might work, but we used clang-17
- Boost
- OpenMPI
- Python 3.10
    - mpi4py
    - glob - used for parsing through the file paths

1. Extract full-data.tar.gz into project root. Should follow the pattern data/YYYYMMDD/YYYYMMDD-HH.csv
2. Create a build folder called "b" in cpp-src. From inside of it run `cmake ..` and `make`
3. Run ./run_mpi.sh from the root of the project
    a. You can edit the file to adjust how many worker processes are spawned.

Team Contributions
Steven Chang:
Brahma Teja Chilumula:
Avi Ajmera:
Hasan Mhowwala: