EasyMPI: Checkpoint Report

Ziheng Liao, Xingyu Jin

November 20, 2017

Abstract

We are going to implement a easy-to-use yet powerful profiler for OpenMPI which aims to provide useful insights to OpenMPI programmers and help them improve their target OpenMPI programs' performance. Web page of this project is located at: https://alphalzh.github.io/EasyMPI/.

1 Work Completed so far

- Week 0 (Wed. 11/01 - Sun. 11/05) Propose the project, finished background research, meet with Prof. Railing to discuss about questions and issues, study Contech.
- Week 1 (Mon. 11/06 - Sun. 11/12) (A lot of thanks to Prof. Railing, as he configured Contech for MPI on Latedays and started collecting data using our code) Verified that Contech can be used on OpenMPI and started data collection part of the profiler. Studied Contech and tried to configure it on latedays.
- Week 2 (Mon. 11/13 - Sun. 11/19) Collected data from Prof. Railing on Wednesday, begin testing on the collected data and implementing the backend.

2 Challenges

This part lists the challenge we've encountered so far. Solutions or possible workarounds are also listed.

- Since we didn't have root access on latedays cluster, we found it difficult to build and use the instrumentation framework "contech" to collect data ourselves. Luckily, Prof. Railing helped us collected two taskgraphs using contech and our OpenMPI code for assignment 4.
- The taskgraphs collected by contech are huge and might simply not fit into memory for analysis. We created an AWS instance for this task, but the data transfer time is very long. Also we try to avoid opening too many tasks at one time so as to fit everything into memory.

- Instrumentation will likely be costly and cause a lot of overheads. So far, there's about 8-13x slowdown for the taskgraphs collected on OpenMPI programs. However, this might not be a major concern for our profiler. Actual effects of the overhead will be analyzed after we finish most of our implementation and begin testing.

3 Progress

We are about half to one week behind our original schedule, thus the schedule is adjusted accordingly for us to catch up. The major cause for the delay was that we find it difficult to configure contech on latedays and instrument an MPI program by ourselves. We still believe that we can pursue our original objectives, and we plan to utilize the thanksgiving holiday to make major progress on the project goals. We've also adjusted some of our goals to make them more realistic. The nice-to-haves are still available options, if we could follow our schedule strictly. As we are currently analyzing the collected taskgraphs and implementing the backend, we may still change some of these goals based on the results we get, but the majority of goals should stay unchanged.

4 Demo in poster session

We are going to profile our final solution to 15-618 Assignment 4 (and possibly other less efficient solutions we came up with earlier) using the profiler we made for this project, and showcase how could our profiler help us to achieve better speedup.

5 Schedule

- Week 3, first half (Mon. 11/20 - Thurs. 11/23) Finish checkpoint report, finish analyzing the taskgraph and part of backend implementation(Ziheng Liao), start frontend (the web interface) implementation(Xingyu Jin).
- Week 3, second half (Fri. 11/24 - Sun. 11/26) Finish a majority of backend implementation, start unit testing(Ziheng Liao), finish a majority of frontend implementation(Xingyu Jin).
- Week 4 (Mon. 11/27 - Sun. 12/03) Finish backend implementation and unit testing(Ziheng Liao). Finish frontend implementation and start predictor implementation(Xingyu Jin).
- Week 5 (Mon. 12/04 - Sun. 12/10) Finish predictor implementation(Xingyu Jin). Finish integration and functional testing, finish project report, finish project poster(Both).

6 Appendix: Goals and Deliverables

This section is still subject to modification as stated in previous sections.

6.1 Goals to achieve

The profiler should be able to correctly collect data through instrumentation from the execution of any parallel OpenMPI program and make the following analysis:

- Time spent in each basic block (or even finer grained code blocks). Visualization is needed. For this part, we believe that it's necessary to let the user know the most time-consuming part of code in a relatively easy way. If possible, we would like to visualize directly over the source code, or tell the user "which lines of your code take how long to run". For parts of the source code that involves parallel processing, a separated view of the total time that each process takes and the message passing steps take will be generated.

- Topology of basic blocks (a.k.a. task graphs in Contech). This basically shows the execute order and dependencies concerning basic blocks and synchronizations between different processes. This will help the programmer understand the behaviour of their parallel code better. Visualization of such topology can be combined with time spent in each basic block. Please refer to https://dl.acm.org/citation.cfm?doid=2776893 for more information.

- Average size of messages passed by different MPI methods and their overhead (time used for passing these messages). This is a finer-grained breakdown of message passing overhead since different message passing methods (for example, MPI_Bcast and MPI_Send) that appears in the program will be analyzed. Notice that this won't generate analysis with regard to every single process, but instead a single view concerning the whole picture will be generated. Difference between processes will be reflected in the next analysis, which shows workload imbalance.

- Level of workload imbalance. For each process i, two important stats will be provided: the deviation from mean working time across all n processes is calculated by

  $$\Delta T_i = |\frac{1}{n}\Sigma_1^n T_n - T_i|$$

  , and the deviation from mean size of messages sent from processes is calculated by

  $$\Delta S_i = |\frac{1}{n}\Sigma_1^n S_n - S_i|$$

  , where S_i is the total size of messages sent from process i.

- Memory usage. This includes number of cache misses and cache miss rate for each process. If possible(yet highly unlikely so), also show overhead caused by cache misses.

- Feedback of how much improvement one can make by tweaking various aspects of the parallel program. For example, how should the performance be if the workload is perfectly balanced, how should the performance be if the communication overhead is reduced by 50%, etc.

6.2 Goals we hope to achieve

We would like to provide insight to programmers before they turn their sequential code into parallel code. So we could take some predictive measurements. The user is supposed to provide the sequential code, the part of code (by starting and ending line number) that is supposed to be parallelized, the estimated size of data to be passed around as messages, and the number of processes to be used by the parallel program. We hope that we can provide information listed below, which concerns the sequential version of code that has not yet been parallelized.

- For each basic block, how much time does it take to run sequentially?
- If the given code is going to be parallelized, what's the expected speedup? Assumptions will be made to make such predictions, for example, we may assume that the workload is evenly distributed. Predictions will base on pre-programmed simulations that simulates overheads under various situations. For example, we may want to simulate the overhead of broadcasting 4MB of data to 16 processes for 5 times on a given cluster (e.g. latedays).

References

[1] Brian P. Railing, Eric R. Hein, and Thomas M. Conte. 2015. Contech: Efficiently Generating Dynamic Task Graphs for Arbitrary Parallel Programs. ACM Transactions on Architecture and Code Optimization 12, 2 (August 2015), 124. DOI:http://dx.doi.org/10.1145/2776893