# EasyMPI: an easy-to-use OpenMPI profiler

Ziheng Liao, Xingyu Jin

November 1, 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 Background

As people come to realize the tremendous benefits that parallel computing can bring along to the modern computing industry, more and more programmers begin to learn how to write parallel programs. However, it's often hard for even an experienced programmer to write a program that efficiently exploits the potential parallelism and achieve desired performance, without knowing what is actually happening during the execution of the parallel program. For instance, in a framework implementing the message passing model, the bottleneck of the parallel program could be its sequentially executed part, the overhead of passing messages between processes, workload imbalance, etc. It will be extremely helpful if there is a profiler that collects and analyses different aspects of a program execution and provides useful feedbacks to the programmer.

There exists many powerful profilers for various popular parallel computing frameworks that are free-to-download. However, most of them are hard to configure and have relatively steep learning curves. Our aim in this project is to design a light-weight profiler for OpenMPI (a popular message passing parallel computing framework) that can deliver straightforward yet vital statistics about a parallel program, and make it easy for programmers to make correct decisions to optimize their parallelization approaches.

## 2 The Challenge

Making a profiler is never easy work. There are several challenges we have thought of at this stage of the project, as listed below.

- It's hard to build the lower level of this profiler, which is the instrumentation of a OpenMPI program. Luckily, Prof. Railing is willing to let us build upon Contech[1], a

compiler-based instrumentation framework that generates trace-based taskgraphs for analysis.

- It's hard to make a wise choice of what data we need from an execution of an instrumented program, and how to utilize/analyse these data to generate a insightful result that the user can make use of.
- We plan to include a feature that before a programmer actually writes a parallel program, our profiler can also collect information from the serial version of the code and provide some extra information that may help the programmer design the parallel code. This introduces the problem of scaling, which is how can we (partially) predict the performance of a parallel program (for example, predicting the message passing overhead on a specific cluster, given message size and number of processes) on different environments.
- The overhead of profiling is also important, as it shouldn't be too significant to impact the behaviour of the original parallel program.

# 3 Resources

We should need the latedays cluster for developing and testing our profiler. We should need Contech, the parallel instrumentation framework that we have described before, and build our profiler on top of it.

# 4 Goals and Deliverables

## 4.1 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.

Also, an important thing to a profiler is its own overhead.(i.e. the 'performance impact' of the profiler). We can't make any assumptions or promises on that yet, since we have no idea what will Contech's instrumentation overhead be if we want to achieve all the goals stated above. But desirably we would like the overhead of this profiler to be less than 20-30% of the original parallel program.

## 4.2 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).

### 4.3 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 Platform Choice

We choose to base our profiler on the OpenMPI framework. Among the three popular parallel computing frameworks that we are familiar with (CUDA, OpenMP, OpenMPI), CUDA and OpenMP already have relatively easy-to-use and powerful profilers made by large corporations like Nvidia and Intel. OpenMPI has less viable profiling tools so that we decided our project might fill the gap.

Also, since we are trying to perform predictive analysis over the sequential version of the parallel code, OpenMPI might be easier for us to achieve that goal by simulating the message passing step(s) between processes and measure the overhead. For other frameworks (like OpenMP) that built upon a shared memory model, locking/unlocking overhead and overhead incurred by shared memory access (e.g. false sharing, cache miss, etc) can be rather hard to simulate.

In addition, Contech, which would be the basis of our project, is also compatible with OpenMPI. However, if there are unresolvable compatibility issues between Contech and OpenMPI, we might still choose to switch to OpenMP for the project.

## 6 Schedule

- Week 0 (Wed. 11/01 - Sun. 11/05) Propose the project, meet with Prof. Railing to discuss about questions and issues, study Contech, Prepare the dev environment.
- Week 1 (Mon. 11/06 - Sun. 11/12) Finish customization of Contech (if needed) and the data collection part of the profiler. Begin to implement the backend and make some progress.
- Week 2 (Mon. 11/13 - Sun. 11/19) Finish implementing and testing 80% of backend (analysis) functions. Start checkpoint report.

- Week 3 (Mon. 11/20 - Sun. 11/26) Finish checkpoint report, Finish the rest of backend implementation and testing, begin frontend (the web interface) implementation and predictor implementation.
- Week 4 (Mon. 11/27 - Sun. 12/03) Finish frontend implementation and predictor implementation.
- Week 5 (Mon. 12/04 - Sun. 12/10) Tweak the product, finish project report, finish project poster.

Week 1 - 4 are critical for the project development and workloads may be distributed to these weeks differently from proposed.

# 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