EasyProf: An easy-to-use parallel application profiling toolkit based on Contech

Ziheng Liao, Xingyu Jin*

Carnegie Mellon University
zihengl@andrew.cmu.edu & xingyuj1@andrew.cmu.edu

December 11, 2017

I. Summary

We implemented EasyProf (formerly named EasyMPI), a light-weight profiler tool for parallel applications built on top of frameworks like pthread, cilk and OpenMP, based on the instrumentation framework Contech[1]. Deliverables include an interactive application ("the profiler") that can take in an parallel application's taskgraph generated by Contech and output detailed performance statistics about the parallel application, an application ("the visualizer") that can take in the same taskgraph and generate graphical visualization of the parallel application, and this project report.

II. Background

Performance is always one of the most important aspects to consider in terms of parallel application design. However, it's often non-trivial to measure a parallel application's performance and figure out the bottlenecks. Therefore, we decide to build a profiling tool that can help programmers learn more about their parallel applications' performance.

We implemented our profiling tool on top of Contech, an instrumentation framework which is capable of collecting traces from parallel applications and generating task graphs from the collected trace. Such task graphs can then be processed by various "backends" to yield useful profiling results. EasyProf can be regarded as a powerful backend for Contech such that given a task graph file generated by running a Contech-instrumented executable, it can provide detailed statistics and useful information such as visualization of the task graph, work imbalance across multiple execution contexts, total time spent on synchronization, etc. to the programmer.

Since the profiler is based on the task graph collected by Contech, the vital part of our project is to effectively analyze the task

*Thanks Prof. Brain Railing for his help in developing this project.

graph and extract as much useful information about the application being profiled as we can, then organize and summarize such information to present detailed, human-readable profiling result to the user. Following paragraphs in this section will give a brief introduction about the structure of Contech's taskgraph and key data structures of EasyProf, while the next section will focus more on our approach to traverse the task graph and extract useful data.

A "task" in the context of task graph can be seen as an aggregation of actions. Such action can be an instruction, a basic block, a function, etc. Under the scope of parallel computing, aggregation of actions into a task happens when no synchronization is involved. Such aggregated tasks are named work tasks in Contech. The other type of task, which represents synchronizations happening during program execution, are further divided into create tasks, join tasks, barrier tasks and sync tasks. While other names of task types are self-explanatory, sync tasks in Contech represents synchronizations other than create, join and barrier, for example lock and unlock. All the tasks and dependencies between these tasks forms a directed acyclic graph called task graph, where nodes of the task graph represents tasks and directed edges represents dependencies between tasks. Figure 1[1] illustrates the aggregation of actions into tasks. Figure 2[1] shows a visualization of Contech task graph. Figure 3[1] shows different types of synchronizations represented in the task graph.

Running an instrumented application will

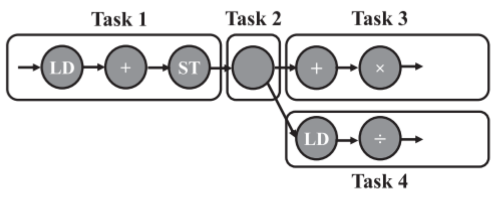


Figure 1: *Aggregation of action into tasks*

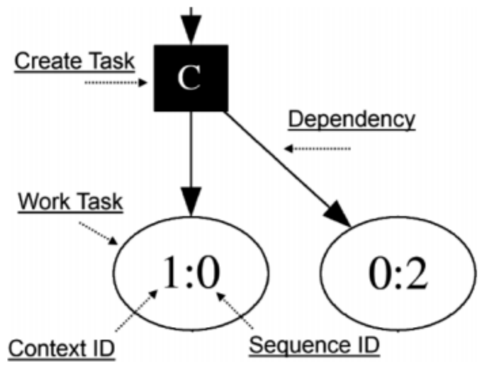


Figure 2: *Contech task graph representation*

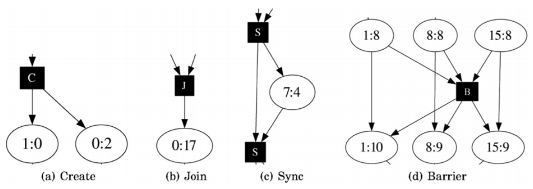


Figure 3: *Contech task types*

generate such a task graph, which contains massive information about each task. For example, starting and ending timestamp of the task, memory operation and footprints, etc. EasyProf traverses a given task graph, collects such information and stores it inside its data structure. Since there are too many data structures storing various informations, only typical types of used data structures are introduced here. Some overall statistics like the total number of memory operations are stored in single (c++) uint or uint64 variables (listing 1, line 2). Statistics about a specific execution context (e.g. threads in the context of pthread) are stored in maps in c++ (listing 1, line 3), which are k-v stores whose keys are the context ids and values are the aggregated information. Statistics about specific context groups (e.g. groups of threads spawned closely to handle similar workload) are stored in vectors of maps in c++ (listing 1, line 4) whose each element is a map representing a particular context group.

Listing 1: *Example data structures*

```
1  // Example data structures.
2  uint64 totalMemOps = 0;
3  map<ContextId, uint64> maxTimePerContext;
4  vector<map<ContextId, uint64> > totalTimePerGroup;
```

The profiler's input is the task graph mentioned previously, and the profiler will start to analyze the task graph. After the task graph is successfully evaluated, the profiler will provide an interactive shell for the user. User can query for their desired information by entering specific commands. A list of available commands, the same of which will be shown by the profiler itself when you enter the help command, is presented below. As the profiler is the final product, the output of each command will be explained in the "Results" section.

stat	[basic statistics about the program]
mem	[information about memory usage]
main	[information about main execution context]
group	[information about execution context group]
g{groupId}	[information about specific context group]
c{contextId}	[information about specific context]
quit	[quit EasyProf]

Table 1: *available commands of EasyProf*

The visualizer's input (please add here)

III. Approach

EasyProf targets any parallel computing frameworks supported by Contech. Since our work is based on Contech's task graph, any parallel application that can be instrumented by Contech to generate a valid task graph can therefore be profiled and analyzed by EasyProf. By the time of this report, current version of Contech supports pthread, cilk, OpenMP and OpenMPI applications. Our work was previously named EasyMPI and in the beginning we wished to target only MPI applications, but the power of Contech enabled us to profile a much larger domain of parallel applications.

The following content of this section will be separated into two parts. The first part, written by Ziheng Liao, will introduce the approach of the profiler, while the second part, written by Xingyu Jin, will introduce the approach of the visualizer.

i. Profiler

It's worthy to explain a few names that will be used in the following paragraphs. To be specific, backends are programs to analyze task graphs generated by Contech, and an execution context is an abstraction that contains a set of tasks which can not be executed in parallel with each other, for example an execution context may only contain all tasks generated by a single thread, in a pthread application.

As previously mentioned, task graphs generated by Contech contains massive information. Most existing backends (programs to analyze task graphs generated by Contech) try to first traverse the task graph using a predefined order given by the task graph class in Contech. While this approach is good at aggregating overall statistics, it barely consider individual execution contexts as well as dependencies between contexts. However, providing only raw statistics about the whole application can hardly help the programmer to form a clear picture about the application's performance and identify the real bottleneck of the application. In particular, as a programmer myself, I would like to know more information about the work imbalance between execution contexts, communication to computation ratio of each execution context, total time taken by the application to run sequential code, etc. So in EasyProf's profiler, we managed to aggregate execution contexts inside execution context groups, where inside each group the execution contexts are likely to run similar tasks in parallel. For example, imagine generating 16 threads to concurrently calculate a matrix product. Execution contexts representing these threads should belong to the same group. We then traverse the whole task graph group by group, inside each group we traverse context by context, and inside each context we traverse by following the dependency of tasks. Information is collected in "whole application", "per group" and "per context" granularities while we traverse the graph, and analyzed in such granularities after graph traversal completes.

Execution context groups are generated and expanded when contexts are created. Since it's likely that contexts created close to each other belongs to the same execution group, the profiler will automatically group these contexts together. A parameter (which can be manually tuned) specifies the max distance by number of tasks between context creations for two contexts to belong to the same group. Figure 4 shows a visualized task graph, and contexts 0-8 would be assigned to the same group when max distance is set to 3. (the green box in the figure means creation).

A context may belong to multiple context groups. This may happen when a context is created by another context and then create its own contexts, or a context create multiple contexts which are logically far away from each other. However, while a context can be the creator of multiple context groups, it can only be created by one context, and we define the context belong to the group where it is not the creator. The creator of a group's statistics will not be collected in 'per group' granularity when we traverse the task graph. In this way we can avoid problems when calculating statistics like workload imbalance between groups, where the group creator's statistics may cause inaccuracy.

Extra information is collected for the "main" context, which is the first execution context of application (context 0). We believe collecting such information is beneficial, since in a parallel program the first execution context is often the "scheduler" that generates other contexts, assign work to other contexts and perform some extra sequential operations before context creation and after context join. Therefore, we collected statistics about sequential part of the main execution context and analyzed them together with other information. User can query about main context's statistics by entering 'main' in the profiler.

Specific steps in collecting all the information we need include migrating information stored in each "Task" object to the data structures mentioned in previous chapters. The "Task" objects are created when reading a task graph, and contains information like task type, start and end time, task dependencies, basic block informations (memory operations, file of source code, line number, etc.). Please refer to the original paper of Contech and the Contech open source project page for more information.

ii. Visualizer

The visualizer aims to help programmer understand parallel programs intuitively with a visible and detailed graph. There are two improvements on the original visualizer of Contech. One is to compress redundant nodes in the graph to make the graph neat. The other is to show more information in each node.

Figure 4 shows the result of the original visualizer of Contech and Figure 5 shows the result of our visualizer. There are many nodes named C or J, because the nodes represent pthread_create or pthread_join in the former graph. Logically, these nodes are not helpful for programmers because programmers usually only care workload that each thread has instead of procedures to create or threads. Thus we can use only one node to represent consecutively create or join a series of threads. This will make the graph much more clean and easy to read. In the latter graph, our visualizer compresses the create node and the join node. The logical flow seems very clear. It depicts that thread 0 generates another 8 threads and finally join all threads. In addition, there are redundant consecutive normal nodes like the first two nodes and the last two nodes. These nodes can be merged to make the graph clean.

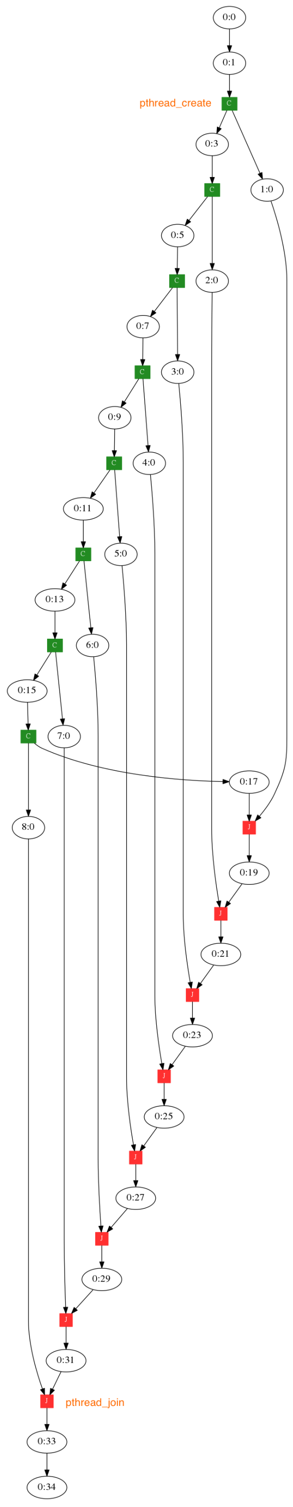


Figure 4: *Graph generated by Original Visualizer in Contech*

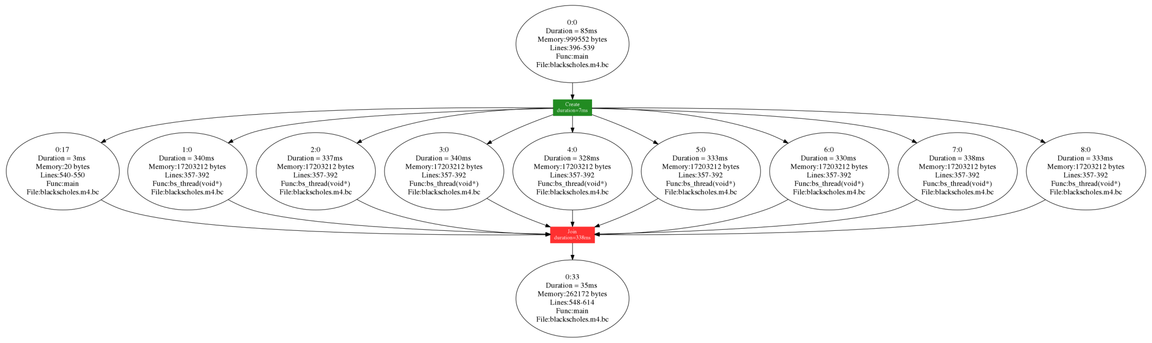


Figure 5: *Graph generated by EasyProf Visualizer*

Besides, The former graph provides very limited information, only context numbers and sequence numbers. However, the latter graph shows more details about each thread, including duration, memory usage, lines of source code, function names, and source files. The additional information will help programmers to understand how parallel programs work in reality. The duration of each thread will show the workload. Programmers can know whether the workload of each thread is balanced by observing and comparing the duration of each thread. They will also know accurate location of codes in source files to identify the code block to optimize. This is very helpful for programmers who begin their study in parallel computing. Therefore, our visualizer generates much more detailed graphs to represent parallel programs and it is quite helpful for programmers to understand parallel computing and find bottlenecks to optimize their codes.

Here is a brief introduction of implementations of our visualizer. EasyProf visualizer is build on the base of Contech. It extracts information from a task graph file and rebuild the graph in memory. We construct a new class to maintain necessary information like duration and memory usage and store adjacent relation between graph nodes. Then we run a breadth first search on the graph. When we visit one node in the graph, if this node represents normal node, then examine its child. If its child is also a normal node, then merge these two nodes. If this node means creation, then examine its grandchild, if its grandchild also means creation, then these two nodes means consecutive creating threads in pthread library and we can merge these nodes. The process for join nodes is the same as create nodes. During merge procedure, we carefully cumulate some useful values in nodes like duration and memory usage and update the list of successors and predecessors of a node. After traversing all nodes, we can get a compressed detailed graph.

IV. Results

i. Profiler

The profiler can analyze a given task graph and output the following information.

General statistics about the program:

- total number of tasks
- total number of create/join/sync/barrier tasks
- total number of basic blocks and unique basic blocks
- average and max time taken by a task
- total number of memory accesses
- total memory footprint in bytes
- total context groups
- maximum number of parallelized execution contexts

Memory usage statistics:

- total memory operations
- total number of read/write/malloc/free operations
- total number of bytes written/read
- potential memory leak
- average and max memory footprint
- average and max memory operations per task.

Statistics about main execution context:

- total time spent by main context
- total time spent by main context before context creation
- total time spent by main context after context join
- sequential execution time percentage of main context
- total time spent by main context for synchronization
- synchronization time percentage of main context

Statistics about context groups:

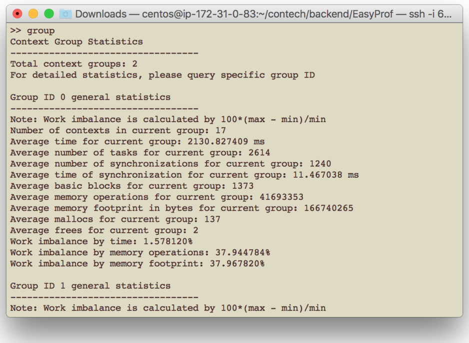
- number of context groups
- max, average and min time spent by contexts in a context group
- max, average and min number of tasks inside contexts in a context group
- max, average and min number of synchronization tasks inside contexts in a context group
- max, average and min time of synchronization tasks inside contexts in a context group
- max, average and min number of memory operations of contexts in a context group
- max, average and min memory footprints of contexts in a context group
- average number of mallocs and frees of contexts in a context group
- work imbalance by time across context

groups
- work imbalance by memory operations across context groups
- work imbalance by memory footprint across context groups
- for the max/min values listed above, also show the corresponding context ID.

Statistics about specific context:

- total time spent by current context
- total number of tasks inside current context
- total number of synchronization tasks inside current context
- total time of synchronization tasks inside current context
- total basic blocks inside current context
- total memory operations and footprints for current context
- total mallocs and frees for current context
- max time spent by a task in current context
- max memory operations of a task in current context
- max memory footprint of a task in current context
- for the max values listed above, also show the corresponding task ID.

Example screen of gathered statistics from a task graph is shown below. You may need to zoom in for a clearer view. Currently, the profiler takes about 15 seconds to 2 minutes to process a task graph smaller than 500MB. The test is conducted on an AWS EC2 instance with CentOS 7.0, 32GB memory, Intel(R) Xeon(R) CPU E5-2676 v3 2.40GHz CPU. C++ 11 support is needed to compile the project.


Downloads — centos@ip-172-31-0-83:~/contech/backend/EasyProf — ssh -i 6...

```
>> group
Context Group Statistics
----------------------------------
Total context groups: 2
For detailed statistics, please query specific group ID

Group ID 0 general statistics
----------------------------------
Note: Work imbalance is calculated by 100*(max - min)/min
Number of contexts in current group: 17
Average time for current group: 2130.827409 ms
Average number of tasks for current group: 2614
Average number of synchronizations for current group: 1240
Average time of synchronization for current group: 11.467038 ms
Average basic blocks for current group: 1373
Average memory operations for current group: 41693353
Average memory footprint in bytes for current group: 166740265
Average mallocs for current group: 137
Average frees for current group: 2
Work imbalance by time: 1.578120%
Work imbalance by memory operations: 37.944784%
Work imbalance by memory footprint: 37.967820%

Group ID 1 general statistics
----------------------------------
Note: Work imbalance is calculated by 100*(max - min)/min
```

Figure 6: *screen shot of an actual run*

ii. Visualizer

The visualizer provides compressed and detailed graph about a parallel program, as Figure 5 shows. It compresses redundant nodes and make the graph easy to read. It also provides significant information about each node like duration to help programmers to detect bottlenecks in their programs.

Here are information in one node

- Context ID and Sequence ID
- Duration
- Memory usage
- Start line number of corresponding codes
- End line number of corresponding codes

- Function name
- Source file name

Currently, it is proper to display small scale compressed graph by our visualizer. Our visualizer also supports display detailed partial graph without compression on a very large task graph.

V. Future Work

i. Profiler

- Currently, the profiler's processing time of task graph is relatively long. Although dependency exists between different contexts, it's still possible to parallelize the profiler itself to improve its performance.
- Future versions of the profiler may take advantage of synchronization tasks' dependency information and perform related analysis. For example, it may be capable of spotting the "synchronization hotspot" among contexts in the same context group.
- While basic block time recording is possible in Contech, it's currently very inefficient and requires another building pass. If future versions of Contech supports basic block time better, more analysis could be added to EasyProf's profiler, for example memory access latency analysis.

ii. Visualizer

- Currently, the visualizer only supports to compress redundant normal nodes, "create" nodes and "join" nodes. However, there are large number of "sync" nodes in large scale task graphs. Compressing "sync" nodes is much difficult because the relation between two "sync" nodes is complicated and need more research to guarantee a correct compressing solution on "sync" nodes.
- The visualizer can not identify "for" loop in task graphs. Some large scale task graphs contain a "for" loop with many iterations. It will be helpful to detect different iterations in one "for" loop and compress the "for" loop with iteration information. Since parallel "for" loop is very common in parallel programming, it is very interesting to detect and compress "for" loop in task graphs.
- Current visualizer only supports to depict compressed graphs on small task graphs, because it reads all task nodes in memory and run compression procedure. It is better to support partial compression graph when a user only wants to see a compressed graph on a subset of task nodes.

VI. List of Work

- Ziheng Liao: Proposed the project, write the milestone report, implemented the profiler, co-authored the final report. Finished 60% of the project.
- Xingyu Jin: Implemented the visualizer, co-authored the final report. Finished 40% of the project.

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), 1âĂŞ24. DOI:http://dx.doi.org/10.1145/2776893