Implementation of a Multi-locale Chapel Profiler

Hui Zhang
Department of Computer Science
University of Maryland
College Park, MD, USA
hzhang86@cs.umd.edu

Jeffrey K. Hollingsworth
Department of Computer Science
University of Maryland
College Park, MD, USA
hollings@cs.umd.edu

ABSTRACT

We developed a data-centric and code-centric combined tool, ChplBlamer, to pinpoint performance loss in multi-locale Chapel programs. To evaluate ChplBlamer, we studied three multi-locale Chapel benchmarks. For each one, ChplBlamer found the causes of performance losses. With the optimization guidance provided by ChplBlamer, we significantly improved the performance by up to 4x with little code modification.

KEYWORDS

Data-centric profiling, Multi-locale Chapel

1. INTRODUCTION

The data-centric approach that relates performance to data structures rather than code regions is especially important for PGAS applications since they are usually memory-bound and communication-bound. In our previous work [4], we proposed an idea of data-centric performance measurement approach for single-locale Chapel programs. This work extends the prior work by providing a more functional Chapel profiler, which improves the previous work in several aspects:

1. It supports more generic Chapel code, including multi-locale Chapel, abstractions that support both asynchronous and remote tasks.
2. It provides additional capabilities: inter-node load imbalance checks, and exclusive blame to help users investigate performance issues.
3. The instrumentation to the Chapel runtime library is optimized and the runtime overhead is significantly reduced from the previous 3.5x to the current 14%.

2. RELATED WORK

To our knowledge, there are only four prior tools that profile Chapel code. HPCToolkit [1] has intrinsic support for profiling all multithreaded programs, but it does not associate the work offloaded to worker threads to Chapel's user-level calling context. Pprof [2] attributes performance data to the function level, but it does not distinguish generated functions from user functions and does not contain calling context. Chplvis [3] visualizes the inter-locale communication and task computation of Chapel programs. However, it needs source modifications and it does not map performance data back to higher-level source abstractions. The most recent work [4] is the only one that profiles Chapel code and presents the result in a completely user-level context. However, it does not support multi-locale Chapel and asynchronous tasking, which largely limited its application; it also incurs high runtime overhead in certain cases.

3. CHALLENGES AND NEW FEATURES

There are several major problems that we solved to advance the previous work.

3.1 Multi-locale Chapel

Conducting data-centric profiling on multi-locale Chapel is far more challenging than the single-locale. ChplBlamer collects performance data at the instruction level then map them back to higher level data-centric abstractions.

First, for a variable that is distributed among multiple locales and requires remote access, there are hundreds of aliases and temporary variables representing the data of the variable in the computation. How to identify those data blocks and finally aggregate their individual blame to the original variable is a problem. We addressed that by associating each distributed variable with its privatized ID.

Second, multi-locale Chapel programs call functions from the runtime library and standard modules to retrieve the locality information for remote data access, which involves implicit dataflow information. Also, function pointers are used for executing parallel tasks, which precludes the inter-procedural blame propagation.

Lastly, for multi-locale Chapel, it's harder to get the complete calling context for each sample, which is necessary to transfer blame along the call path appropriately. Chapel's asynchronous tasking feature aggravates this problem. We instrumented both the tasking and communication layers of the Chapel runtime using callback functions, where we record keys (function ID, locale IDs) for each local or remote task. Using keys to reconstruct the calling context will: 1st, resolve the asynchronous and remote tasking problem since with the keys we know what each task does and where it was launched; 2nd, significantly reduce the runtime overhead since we can avoid unwinding the stack for tasks that came from the same calling context. This approach reduces the average overhead from 3.5x to 14% for three single-locale Chapel benchmarks in the previous work [4].

3.2 Exclusive blame

The original blame calculation is an inclusive data-centric profiling approach. Therefore, the variables that hold the ultimate results will stand out in terms of the importance. To supplement that, we provide another way of evaluating variables in terms of the potential of optimization: exclusive blame. Exclusive blame only attributes a line to a variable if there is a direct write to the variable at that line. Therefore, computation-intensive variables will have a larger percentage. E.g., if a sample is mapped to the line "*b = a + 1*" and we have another line "*c = b*", while the inclusive blame also blames *c* for this sample, the exclusive blame only blames *b*.

3.3 Inter-node load imbalance check

We also include a view of workload information. The different time on each locale shows the load imbalance situation in terms of a particular variable. For a distributed array, if certain locale consumes significantly more or less time than others, it means significantly greater or fewer array elements are distributed on that locale than others. Thus the user should tune the block size of the distribution based on the array size for that variable.

Table I. Major data-centric and code-centric blame percentages for HPL on different number of locales

#Locales	2	4	8	16	32(200)
Variable Name	**Data-centric Blame**				
Ab	55.7%	45.8%	36.4%	32.7%	23.6%
MatVectSpace	18.6%	27.8%	37.1%	38.7%	51.0%
Function Name	**Code-centric Blame**				
schurComplement	30.7%	22.2%	17.3%	14.0%	8.7%
panelSolve	13.4%	15.7%	15.0%	15.8%	12.0%
polling	3.4%	11.5%	7.5%	11.7%	9.6%
chpl_comm_barrier	7.5%	10.5%	11.4%	11.5%	11.0%
pthread_spin_lock	14.3%	3.8%	7.1%	5.0%	4.5%

4. CASE STUDIES

We studied three multi-locale Chapel benchmarks. All programs were built with Chapel 1.15 and "--fast".

With ChplBlamer's multi-angle profiling, we are able to locate the bottlenecks of performance and scalability. Table I shows the partial profiling result of HPL as an example. We concluded our findings on three benchmarks via ChplBlamer as below:

HPL: The inter-locale overhead becomes dominant over the intra-locale as more locales are involved. Enabling 'local' clause inside function *schurComplement* reduced the execution time by 3.1%. In function, *panelSolve*, we can leverage the *ReplicatedDist* module to create local copies and reduce remote data accesses.

ISx: The Barrier module needs to be further optimized or removed from this application since it becomes a major performance bottleneck as the problem scales. We found a spot where adding 'local' clause can reduce about 15% of the total execution time.

LULESH: The two optimizations found in [4] still help in the multi-locale execution. However, the most significant optimization we found via ChplBlamer is by removing calls to a communication-intensive function *localizeNeighborNodes* and simply doing copies to localize Node's attributes. With some auxiliary data structures, the copying can be done in a completely distributed parallel style. Further, this optimization leads to new opportunities for data localization. Ultimately, we move from having slowdown as more locales were added to having speedups.

Table II summarizes the speedups of three benchmarks with optimizations guided by ChplBlamer.

Table II. Speedups of optimized versions

#Locales	2	4	8	16	32
HPL	1.01x	1.01x	1.01x	1.07x	1.05x
ISx	1.26x	1.12x	1.05x	1.09x	1.11x
LULESH	1.51x	1.85x	2.43x	2.78x	3.98x

5. CONCLUSION

In conclusion, this paper describes ChplBlamer, a profiler to identify, quantify, and analyze the performance bottlenecks in multi-locale Chapel programs. Compared to the state-of-art of Chapel profilers, ChplBlamer fully supports multi-locale, asynchronous tasking, provides additional features, and incurs much lower runtime overhead. Guided by ChplBlamer, we were able to pinpoint performance bottlenecks in three communication-bound Chapel benchmarks and identify the causes in the user-level context. We summarize the optimization as globalization, replication, and localization. With little modification to the code, we gain speedups of 1.05x for HPL, 1.11x for ISx, and 4.0x for LULESH on 32 locales over the currently fastest versions.

6. REFERENCES

[1] Adhianto, Laksono, et al. "HPCToolkit: Tools for performance analysis of optimized parallel programs." Concurrency and Computation: Practice and Experience 22.6 (2010): 685-701.

[2] Vöcking, Heye. "Performance analysis using Great Performance Tools and Linux Trace Toolkit next generation." p. 17. (2012).

[3] Nelson, Philip A., and Greg Titus. "Chplvis: A Communication and Task Visualization Tool for Chapel." Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International. IEEE, 2016.

[4] Zhang, Hui, and Jeffrey K. Hollingsworth. "Data Centric Performance Measurement Techniques for Chapel Programs." Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International. IEEE, 2017.