Data Centric Performance Measurement Techniques for Chapel Programs

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*—Chapel is an emerging PGAS (Partitioned Global Address Space) language whose design goal is to make parallel programming more productive and generally accessible. To date, the implementation effort has focused primarily on correctness over performance. We present a performance measurement technique for Chapel and the idea is also applicable to other PGAS models. The unique feature of our tool is that it associates the performance statistics not to the code regions (functions), but to the variables (including the heap allocated, static, and local variables) in the source code. Unlike code-centric methods, this data-centric analysis capability exposes new optimization opportunities that are useful in resolving data locality problems. This paper introduces our idea and implementations of the approach with three benchmarks. We also include a case study optimizing benchmarks based on the information from our tool. The optimized versions improved the performance by a factor of 1.4x for LULESH, 2.3x for MiniMD, and 2.1x for CLOMP with simple modifications to the source code.**

Keywords- Data-Centric Profiling, Chapel Performance Analysis, PGAS Model, Benchmark Optimization

I. INTRODUCTION

To exploit locality and scalability in High Performance Computing (HPC), while alleviating the burden on programmers, researchers continue to look for new programming models other than the traditional C +MPI/OpenMP model. PGAS [1] is an approach to that goal and it has been actively studied in both academia and industry. PGAS provides users with a flat address space atop a physically distributed computer memory system.

Compared to earlier PGAS languages: UPC [2], Co-Array Fortran [3], and Titanium [4], Chapel [5] is a relatively new PGAS language. It supports both task parallelism and data parallelism with locality-aware language constructs. Chapel supports code reuse and rapid prototyping via object-oriented design, type inference, and features for generic programming. Another important feature of Chapel is the language's elegance and conciseness. For example, a Chapel version of the LULESH benchmark has just 1288 lines of code (plus 266 lines of comments and 487 blank lines), while the corresponding C+OpenMP+MPI version is nearly 4 times bigger [6].

Performance is critical in HPC programming, but it is usually difficult to identify the inefficiencies and performance bottlenecks of high-level applications at the source code level. Therefore, a number of profilers have been developed to help programmers optimize their parallel programs.

The classic code-centric view of the performance data is helpful in pinpointing the hot spots at the granularity from the instruction-level to the procedure-level in the program. However, the traditional code-centric view of performance data lacks the capability to find performance problems associated with the different variables accessed by specific lines in the code. Data-centric approaches that relate performance to data structures rather than code regions are especially important for HPC/PGAS applications since memory allocation and data movement are often the key bottlenecks. Therefore a profiling tool that can identify the inefficiencies by memory regions is highly desirable.

II. RELATED WORK

A. HPC Profiling Tools

There are a variety of tools that can analyze High Performance Computing (HPC) applications, based on different profiling methods such as sampling and direct instrumentation. For instance, the Tuning Analysis Utilities (TAU) specializes in profiling with some tracing functionalities [7]. Vampir provides interactive visualization and exploration of parallel event traces [8]. HPCToolkit [9] is an integrated suite of tools for performance measurement and analysis on computers ranging from multicore desktop systems to supercomputers. It relies on periodic sampling to capture the dynamic run-time behavior of parallel applications.

A few profilers partially support PGAS programs, but fewer have data-centric features. Tallent and Kerbyson [10] have proposed their method to profile PNNL's Global Arrays based on HPCToolkit [9], which provides a code-centric, data-centric and time-centric view. However, it can only be applied to Global Arrays and lacks a hierarchical mapping of the latency attribution to complex data structures. Parallel Performance Wizard (PPW) is another tool that supports both UPC and SHMEM. One important feature of this framework is the use of generic operation types instead of model-specific constructs whenever possible, thus it has the potential to support multiple PGAS languages [11]. Seisei Itahashi et al. are working on a profiler for X10. It's a modified version of x10 compiler built with Polyglot, a source-to-source translator that inserts probe code into a target x10 program code [12]. Sebastian et al. are working on extending some existing tools like Vampir

1530-2075/17 $31.00 © 2017 IEEE
DOI 10.1109/IPDPS.2017.37

to support the OpenSHMEM standard for parallel programming [13].

B. Chapel Performance Analysis

Chapel permits a programmer to control and reason about locality, using a locale type. Locales are abstract units of target architecture; for most conventional parallel architectures, a locale describes a compute node, such as a multicore or SMP processor. At the low level, a Chapel programmer can explicitly control the system resources on which a task is run, or a variable is allocated. At a higher level, Chapel programmers can specify how domains and arrays are distributed amongst locales, resulting in distributed-memory data-parallelism [5]. In this work, we focus on the single locale. The idea works for multiple locales but implementation details currently stop us from applying the tool to multiple locales, such as tracking data through GASNet between nodes.

To date, there aren't any performance analysis tools that support Chapel's unique language features. The TAU suite [7] from ParaTools demonstrated its support for Chapel with a simple program. HPCToolkit can profile Chapel programs, but it does not associate the work offloaded to worker threads to the full calling context that it came from. Pprof from Google's gperftools partially supports profiling Chapel. It is code-centric and more useful in profiling the runtime library from a Chapel developer's perspective.

To be able to associate the performance statistics to source code variables, not just functions, we also need a data-centric way to map and present the profiling data. HPCToolkit's data-centric component [14], derived from the original HPCToolKit, has been used to profile several HPC benchmarks, either for single-locale or multi-locale environments. Nonetheless, it only tracks the memory allocation and deallocation of static variables and heap-allocated variables that have a size of over 4K bytes. Local variables are completely omitted. Additionally, after the Chapel compiler's translation, the global variables in Chapel source code aren't properly treated. Therefore, most variables in Chapel benchmarks are counted as "unknown data" (in experiments, CLOMP has 96.88% performance statics falling in the "unknown data" category and LULESH reports 95.1% in "unknown data"), which does not provide useful information to programmers.

III. BLAME DEFINITION

Our data-centric approach builds on a performance mapping technique called "variable blame", proposed by Nick Rutar [15]. A variable's blame is a percentage that indicates the share of certain performance metrics, such as time, cache misses and I/O operations due to individual variables.

Blame is an inclusive data-centric method that utilizes the control flow and full data flow information to map performance data all back to variables in the source code. During runtime, event-driven sampling is used. If a sample is triggered for an instruction that is part of the data flow of a given variable, then that particular variable will be blamed for the sample.

Formally, "blame" is presented in terms of values for each variable for one run of a program. Let S be the set of all samples gathered during the run of the program. For a given sample s within S, let W be the set of all statements containing a write to the memory region allocated to the variable v, the aliases of v, and all fields of v. For a structure, this includes all sub-fields within the hierarchy of v. The blame set for v is the union of all the statements in the backward slices for each of the statements in set W,

$$BlameSet(v, W) = \bigcup_{w \in W} BackwardsSlice(w)$$

Variable v is blamed for sample s in the cases where s is a member of the BlameSet(v, W) which we represent with this function,

$$isBlamed(v, s)\{if\big(s \in BlameSet(v)\big)\ then\ 1\ else\ 0\}$$

The blame percentage for a variable for the entire program is the number of samples that are blamed to a particular variable divided by the total number of samples. This is represented by the following formula,

$$BlamePercentage(v, S) = \frac{\sum_{s \in S} isBlamed(v,s)}{|S|}$$

After computing the *BlamePercentage* of variable v, we can say that v is responsible for that fraction of whatever performance metric we chose to generate the samples. For example, if we chose clock cycles as the performance metric and the *BlamePercentage* was x, we would say that v was responsible for the x fraction of all clock cycles over the course of the program.

Calculating "blame" for a given variable is a multi-step process employing both static and dynamic information. Static information is computed once before execution of the program based on data flow relationships. Dynamic information is retrieved for each sample. The ultimate blame value for each variable will be determined by combining these two types of information after program termination.

Consider the small example in Fig. 1 to illustrate how blame is calculated. This piece of code contains 5 lines. During the execution, there are 4 samples; each one is marked as "Sample #". The blame lines (set of statements, represented by line numbers, that will contribute to the value of the corresponding variable) for each variable is shown in Table I.

We statically analyzed the source code at the instruction level to obtain the table. Line 16 is an assignment to the variable a, so 16 is included in a's Blame Lines set; this assignment also has a contribution to the value of c, so that c has line 16 as well. Line 17 is the same case as line 16, so variable b and c include line 17. Line 18 is a condition statement, which has an implicit effect on the value of a, following the policy of our tool, variable a should also take it as its blame line. Line 19 is a direct write to a, so a has line 19, even this statement is not necessarily executed during runtime. Line 20 is a write to variable c, and it's included in

c's. Although this example is analyzed with source lines, the actual blame analysis is based on machine instruction so complex examples where there are multiple statements per line are correctly accounted for.

```
16      a=2;
17      b=3;              //Sample 1
18      if a<b            //Sample 2
19        a=b+1;          //Sample 3
20      c=a+b;            //Sample 4
```

Figure 1. Example Code

TABLE I. VARIABLE-LINES MAP FOR EXAMPLE CODE

Variable Name	a	b	c
Blame Lines	16,18,19	17	16,17,18,19,20

With Table I and runtime data (four samples), we can attribute each sample to variable(s) by checking the existence of the line number on which the sample is triggered in each variable's Blame Lines set. For example, sample 1 falls on line 17; since the blame lines sets of variable *b* and *c* both contain line 17, they are both blamed for this sample.

According to the "blame" definition, out of the total of four samples, we have two samples assigned to variable *a*, one for *b*, and four for *c*, therefore, the ultimate performance data for variable *a*, *b* and *c* are 50%, 25%, and 100%, respectively. For any given sample, multiple variables may share in the blame. For example, sample 1 is attributed to both variable *b* and *c* as we explained before. Therefore, in a given function, the total percentage assigned to all variables can possibly be more than 100%.

The complete process of calculating blame is explained in the following section.

IV. IMPLEMENTING BLAME FOR PGAS LANGUAGES

The framework of our data centric profiling process is presented in Fig. 2. It consists of four steps, combining the static (pre-run) information and dynamic (runtime) information of a binary to map performance data to variables in the source code. Our tool tries to push as much work as possible to the static analysis and postmortem process parts, in order to minimize the perturbation to the runtime performance. Step 1 can be run on a single locale. Step 2 is the program execution under a monitor process. Step 3 is a post processing step, which can be embarrassingly parallel for multi-locale cases. Step 4 is responsible for profiling data aggregation, processing, and presentation to the user via a GUI.

A. Static Analysis

First, we need to get full data flow relations and build data flow dependency trees for each function. The root of each tree is a local variable or an "exit variable." We define an exit variable as having scope outside of the function. This includes incoming parameters that are pointers, global variables used by the function, and return values.

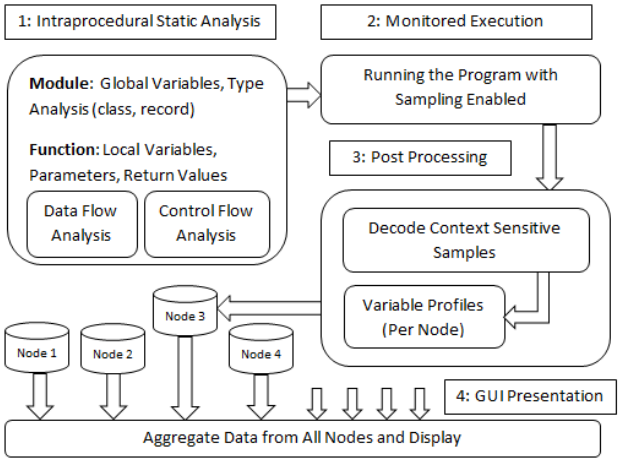

Figure 2. Process of Calculating the Performance Data for Variables

We use the LLVM frontend for Chapel to get the LLVM Intermediate Representation (IR). We also need the debug information to map instructions to source code and associate memory addresses with program variables. Because Chapel did not support debugging information for user code in the LLVM frontend, we implemented the debugging information generation functionality by modifying the frontend in the Chapel compiler. This step will end up generating comprehensive variable-line mapping information for the entire program.

In order to get full data-flow analysis information, we track all temporary variables that were automatically generated during compilation along with source code variables in static and dynamic analysis. However, we flag these internal elements and don't display them in the GUI.

Blame is propagated by both explicit and implicit transfer. Explicit transfer occurs when there is an explicit data dependency between variables. For example, in the code block in Fig. 1, variable c ultimately contains the product of all the work that went into producing its value. Because c explicitly depends on *a* and *b*, so both *a* and *b*'s blame lines are transferred to *c*'s. Implicit transfer is a little more complicated. It happens when there is no direct value assignment between two variables but there exists a variable used in a control dependency. For example, a loop index is incremented for every iteration but is never explicitly involved in any calculation in the loop. However, all variables that are within the loop body will inherit blame from the index variable. The same situation happens to the standard conditional statements and select-when statements.

The above two kinds of transfers rely on explicit and implicit relationships, respectively. The relationships are analyzed statically and the information is stored per function. For explicit relationships, we build a graph based on the data flow between variables. For implicit relationships, we use the control flow graph and generated dominator tree to infer implicit relationships for each basic block. All variables within control dependent basic blocks have a relationship to the implicit variables responsible for the control flow that resulted in the blocks present in that control flow path [15].

Transfer Function: The data flow relationships are all recorded at the function level and calculated through intraprocedural analysis. For interprocedural analysis, we need a mechanism to communicate the blame between functions. The exit variables (parameters that are pointers, return values, global variables) are used to build a transfer function for each procedure. All explicit and implicit blame for each function is represented in terms of these exit variables. For callee functions, we check each exit variable and determine whether that variable is responsible for that sample point (if a sample happened on that function call) at runtime. For caller functions, we have multiple parameters that could be responsible for the function call. At runtime, we use the transfer function to match the blamed exit variables(s) from the callee to the blamed parameter(s) in the caller. Once we have that information, we can establish a blame relationship between the blamed parameter(s) and the parameter(s) that are not blamed in the caller.

B. Execution with Sampling

The execution step involves running the program under a monitoring process and generating raw sample data.

Our sampling uses hardware support via PMU (performance monitoring unit) that exists in most recent processors. A PMU can be configured to trigger an interrupt when a marked event count reaches some threshold.

We use the PAPI [16] library as the interface to utilize PMUs. When a PMU triggers a sample event, the profiler receives a signal and reads PMU registers to extract the precise instruction pointer of the sampled instruction. The marked PAPI event we used as the performance metric is PAPI_TOT_CYC.

Skid is an important factor that most sampling based profilers need to take into account. Some previous work [17] has been done to avoid this problem by sampling instructions instead of events. We plan to add a skid compensation feature in the future.

For each sample that is triggered during the program execution, we need to perform a stack walk to get the call path. We create a separate monitoring process to ensure thread safety. Whenever the sampling process receives an overflow signal indicating it is time to take a sample, the monitoring process will record a stack walk for the associated thread. To implement our monitoring process, we use Dyninst [18] to run our application. When running under Dyninst, all signals are first sent to the Dyninst monitoring process.

To support multi-threading, we instrument the Chapel tasking layer to enable the generation of performance counter overflow signals for each thread. For parallel blocks in Chapel (i.e., forall and coforall), the master thread spawns worker threads to execute. To get the full call paths for the samples for each worker thread, we keep a unique tag for each spawn operation and record the stack trace before the spawn operation begins. This allows us to combine the pre-spawn stack trace with the post-spawn stack trace and produce a full call path for worker threads.

C. Post-mortem Processing

The last step is run after the program has terminated, and takes the raw context sensitive samples and the stored intraprocedural analysis and combines them to determine the final blame.

First, we convert the raw context sensitive samples, which are basically a bunch of addresses, to filenames and line numbers using the DyninstAPI [18]. We do the same conversion for the worker pre-spawn stack traces and post-spawn stack traces, keeping the unique spawn tag for each sample along the way.

Second, we continue to consolidate the stack traces from the first step. We glue the pre-spawn stack trace and post-spawn stack trace based on the unique spawn tag for each sample, to constitute a full call path for worker threads. We trim out redundant stack information that existed in the call path. When encountering samples of which the post-spawn stack trace has no stack frames from the user code, we trace back to its pre-spawn stack trace and locate the place in the user code that caused this sample. Finally, we have a complete, clean call path of the application w/o libraries for each sample. All the context information, including module name, file name, line number and stack order number are stored in an abstraction named “instance” for each sample.

Later, we combine the stored intraprocedural analysis results with the runtime data (“instances”) to determine the ultimate performance data for variables in the source code for each node (or “locale” specifically for Chapel). After resolving the addresses to functions and line numbers, we can utilize the predetermined exit variables to apply transfer functions at each level of the call trace. This means we can bubble the blame up as far as we need to assign it to the appropriate variables in different functions. For variables that are used as parameters, the blame from callee function will be transferred if they are blamed using transfer functions. For those that are not used as parameters, the blame can be assigned without transfer functions since there is no need to propagate their blame share to variables in the upper-level functions. Now we have the performance data for each variable at a per node level. For single locale programs, that will be enough to present, but for multi-locale, we need to aggregate the results across the nodes.

D. Data Presentation

Our tool includes a GUI with three different windows to view the data: a flat data-centric view, a traditional code-centric view, and a hybrid view. The flat data centric view is given to the user by default. It provides a flat view of all the variables defined in the program, ranked in descending order by the percentage of blame they are assigned. We show the performance data for each variable along with its type and context of definition. The second window is a traditional code- centric view that attributes samples to different functions instead of variables. Because we have all the context sensitive samples, we can obtain this view with almost no overhead. The third way of viewing is a hybrid approach between code-centric and data-centric using “blame points”. Blame points are points in the program that are deemed to have interesting variables; the most common

one is the main function, since the variables in there cannot be bubbled up any further in the call stack. Fig. 3 is a screenshot of the GUI with flat data-centric and code-centric main display for one run of MiniMD. The right side is the unique data-centric result that our tool provides using blame analysis while the left side is the inclusive view of the regular sampling based code-centric result.

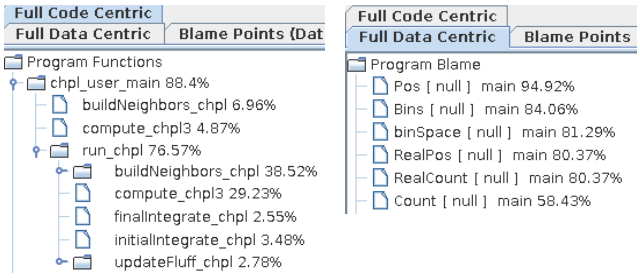

Figure 3. GUI Screenshot

The ongoing code development of this project can be found at https://github.com/hzhang86/BForChapel.

V. EXPERIMENT RESULTS

We now show capabilities that our blame mapping technique offers that differ from traditional tools.

We have chosen three Chapel benchmarks to demonstrate the utility of our tool. Two of them, LULESH and MiniMD are from the Chapel source package, the third benchmark CLOMP, was ported by our group member Johnson [19] from the C version of the Livermore OpenMP benchmark on the Coral Collaboration Benchmark Codes website [20]. All experiments were done on a single locale. We used a 12-core SMP system, each is a 2.53GHz Xeon CPU. The Chapel version we used in experiments is 1.11. The threshold we utilized with PAPI to trigger samples is 608,888,809, which is a large prime.

All programs were compiled with "--llvm --no-checks -g" (meaning the llvm frontend with no redundant boundary checks). Specifically, we did not use "--fast" (equivalent to "-O3" in GNU compilers) since our intraprocedural analysis heavily depends on the generated LLVM bitcode of the Chapel program. Using "--fast" option in compilation would result in an LLVM intermediate representation with too many functions removed or renamed, variables optimized out and instructions reordered. These optimizations would make it nearly impossible to map the performance data from the IR nodes (temporary variables and registers) back to the source level variables with real names[1]. However, to validate that the information supplied by our tool w/o "--fast" provides useful guidance, we reran all of the original and optimized benchmarks with the "--fast" option and show that we get similar gains when using this option as without it.

As for the overhead of our tool, taking LULESH as an example: the typical cost per stack walk is 0.051ms while the interval is about 241ms (or a total overhead of 0.02%); the sizes of the datasets generated during runtime are 6MB to 20MB depending on the problem size; post-processing analysis takes an average of 16ms to process one sample.

[1] Ultimately, the long-term answer is to have the Chapel compiler provide additional information to record transformations made by the --fast option. Such information would also be useful for debugging tools. However, such an effort is out of the scope of this paper.

A. MiniMD

MiniMD, short for "Mini Molecular Dynamics", is a proxy application from Sandia's Mantevo group. It represents key idioms of their real applications. Molecular Dynamics codes compute physical properties like energy, pressure, and temperature for a simulated space containing moving atoms. MiniMD was previously implemented in C++ using MPI and OpenMP, requiring about 5,000 lines of code, while the Chapel version only takes about 2,000 lines.

We picked this application for two reasons: first, it is an important strategic benchmark; second, it has several variables with multi-level data structures that are responsible for most computation so data-centric information is particularly useful. The problem size we tested for the benchmark is (16, 16, 16) unit cells (16,384 atoms). All other input parameters are pre-defined in the source by default.

We ran the test multiple times and report mean values to eliminate run to run variance in the data. Variables with the largest blame values are listed in Table II.

TABLE II. VARIABLES AND THEIR BLAME FOR THE RUN OF MINIMD

Name	Type	Blame	Context
Pos	[DistSpace][perBinSpace] v3	96.3%	main
Bins	[Space][perBinSpace] atom	84.2%	main
RealCount	[binSpace] int(32)	80.8%	main
RealPos	[binSpace][perBinSpace] v3	80.8%	main
Count	[DistSpace] int(32)	54.9%	main
binSpace	domain	49.4%	main

First, we describe the roles of the variables that have a large blame. We then explain how we used this information to optimize the program.

The record named "atom" is the most important data structure in the benchmark. It is an abstraction of atoms in the Molecular Dynamics simulation and contains two basic attributes: velocity (v) and force (f), both are (x, y, z) 3-D real values. It also includes the storage for the neighbor list, which stores the bin and index of a neighboring atom. Therefore, in the global space, which is initialized before the main function runs, we have the two most important variables: *Pos*, which is an array of positions, and *Bins*, which is an array of atoms.

Pos(96.3%): Pos serves as one of the root variables for the entire program. It stores all the position data of the atoms in the space. "v3" is a created type, using a 3*real tuple. *DistSpace* is a domain that defines the bounds of the arrays (*Pos, Bins*) and distributes them across locales in the multi-locale environment, while here it is simply the expanded domain of *binSpace*. *perBinSpace* is a one-dimensional domain. *Pos* takes a lot of blame because the positions are accessed and updated frequently in the

program for the computation of the atom forces as well as neighboring atoms' attributes.

Bins(84.2%): The variable *Bins* is a collection of atoms based on spatial position. The benchmark is to simulate the space by calculating the attributes of each atom, thus this variable is read and written frequently and continuously throughout the entire program. The domains of this variable are basically the same as *Pos*, except the first domain 'Space' is exactly equal to *binSpace* instead of *binSpace*.expand() in the single locale environment.

RealCount/RealPos(80.8%): These two variables are array aliases of *Count* and *Pos*, respectively. In Chapel, array slices alias the data in arrays rather than copying it. They are accessed and updated frequently in the time critical code.

Count(54.9%): The variable *Count* is an array that keeps the count of bins in the space. For domain remapping reason, which we will address later, this variable is "written" (not at the source code level, but at the llvm instruction level) during the main calculations, so is its array alias *RealCount.*

binSpace(49.4%): As we introduced earlier, *binSpace* is a domain whose range is determined by the problem size we set, which tells us the number of bins we need in each direction in the simulation space.

***Discussion and Optimization**:* After a brief review of the benchmark source, we found three functions that handle most of the computation workload inside the real simulation function: *run*. They are *buildNeighbors, updateFluff,* and *ForceLJ.Compute*. Combined with our profiling results, it was discovered that the hot spots of these three functions are inside the nested for loop, where *Bins* and *Pos* are calculated after several domain remapping operations. The function *buildNeighbors* is used to put atoms into correct bins and rebuild neighbor lists; *updateFluff* is to update ghost information of *Pos* and *Bins*, and *ForceLJ.compute* is to compute forces between atoms.

The original code uses succinct zippered iteration expressions to do domain remapping in nested loops. Specifically, zippered iteration refers to a way in which Chapel for-loops can be driven by multiple iterands in a coordinated manner. However, based on our experience with Chapel, that could produce significant overhead, especially in a large nested loop. Johnson's work [19] has done some optimizations on substituting for those zippered iterations in MiniMD. We applied their modifications to the source and obtained a significant improvement in the performance. The full details about the modifications can be found in Appendix A of [19].

The optimization opportunity found by Johnson, et al. was based on a manual performance analysis of the generated code, a complicated and painful process that required examining over 50 C files mixed with user code and Chapel library code. It is very difficult to identify bottlenecks by hand and even harder to map them back to Chapel source code and then optimize them. Using our tool, program tuners can quickly identify the problematic variables in the source code. In the case of MiniMD, by searching for the two most blamed variables, *Pos* and *Bins*, we were able to quickly locate those forall loops that contain zippered iteration and domain remapping. Based on our previous experience of the poor performance when using zippered iteration and domain remapping, we could apply those transformations to increase the performance.

TABLE III. RESULTS W/ OR W/O "--FAST" FLAGS

	Original(s)	Optimized(s)	Speedup
w/o --fast	20.87	9.23	2.26
w/ --fast	6.41	2.50	2.56

Table III shows the performance improvement after our optimization. We gained a speedup of 2.26 on a small-sized problem (size=16). To show that our optimization works no matter which compilation flags we use, we applied the same optimization to the benchmark recompiled with "--fast" option, and ran multiple tests. The result of "--fast" version shows that with compiler optimization enabled, our optimization still produces similar speedups.

B. CLOMP

CLOMP stands for C version of the Livermore OpenMP benchmark [20] and is used to simulate a typical scientific application to measure the overhead of different usage of OpenMP primitives. We selected variables with blame larger than 10% in Table IV.

CLOMP is a simple benchmark. After the initialization, the application starts the simulation by calling the *update_part* function over and over again through the top loop function *do_parallel_version*, inside which, there is only one function: *parallel_cycle*. The function *parallel_cycle* calls four subprograms: *parallel_module1, parallel_module2, parallel_module3 and parallel_module4*. The difference between these subprograms is simply the number of the parallel forall loops in each function. Besides this dominating calculation, there are multiple re-initializations and *calc_deposit* function calls, which only consume a small portion of the total run time. The roles of blamed variables are introduced below.

TABLE IV. PROFILING RESULT FOR THE RUN OF CLOMP

Name	Type	Blame	Context
partArray	[partDomain] Part	99.5%	main
->partArray[i]	Part	99.5%	main
->partArray[i].zoneArray[j]	Zone	99.0%	main
->partArray[i].zoneArray[j].value	real	99.0%	main
->partArray[i].residue	real	12.3%	main
remaining_deposit	real	11.8%	update_part

"->" symbol is used to represent field to its parent struct relation, with the parent struct variable listed above it in the table

partArray(99.5%): Variable *partArray* is the top-level data structure in the benchmark that holds everything of importance. It is created as a global variable, so that the

final blame value from all the points wherever a piece of it gets written, that portion of blame will be aggregated to one single variable in the last step of our tool. The *partArray* is defined on a *partDomain*, which is based on the configurable constant *CLOMP_numParts* so that we can control the size of the domain in the execution command line. Besides other attributes in the *Part* data structure, we have an array of zones, which is created with the self-defined *Zone* type and runtime configurable array size: *CLOMP_zonesPerPart*.

partArray[i].zoneArray[j](99.0%): This variable is the element of the *zoneArray* in the global variable *partArray*. By following the hierarchical symbol "->", we are able to find which field of the complex data structure is actually responsible for the most computation. Here, we can see the *value* in *Zone* takes most credits so we have a basic idea that the whole program is trying to compute the field *value* for each *zone* in *partArray*.

partArray[i].residue(12.3%): The residue is a field of the *Part* class, it is calculated in the *update_part* function. It's another important field that needs to be updated frequently for each part besides all the zones.

remaining_deposit(11.8%): This is simply a local variable defined in function *update_part.* Since the function *update_part* is called frequently, so this variable is accessed a lot as well. It is used as a temporary variable for computing the value of *partArray[i].residue.*

***Discussion and Optimization**:* Since the number of parts and the number of zones per part are determined on the command line, we can use a large 2D array to hold those values, like Johnson and Hollingsworth did [19]. Accessing elements in one big array is much faster than through nested structures. The performance improved by up to a factor of 2.13x. The details can be found in Table V. We also list the result with compiler optimization enabled, in which case we even get better speedups with the same manual optimization.

TABLE V. RESULTS W/ OR W/O "--FAST" FLAGS

Flag	Problem Size	Original (s)	Optimized (s)	Speedup
w/o --fast	1024/64,000	4.02	2.18	1.84
	65536/10	4.79	4.40	1.09
	12/640,000	3.87	1.82	2.13
	65536/6400	7.88	7.14	1.10
w/ --fast	1024/64,000	3.72	1.44	2.59
	65536/10	5.13	2.14	2.40
	12/640,000	3.75	1.41	2.65
	65536/6400	7.98	4.07	1.96

numThreads=12, allocThreads=12, flopScale=1, timescale=100

C. LULESH

LULESH was first implemented by Lawrence Livermore National Lab (LLNL) and has since become a widely studied proxy application in DOE co-design efforts for exascale. LULESH approximates the hydrodynamics equations discretely by partitioning the spatial problem domain into a collection of volumetric elements defined by a mesh. It has a collection of implementation versions based on most modern HPC programming models and languages, including Chapel. The problem size we chose is 15 elements per edge for the time limit considering our current sampling threshold.

LULESH Chapel source was designed to mirror the overall structure of the C++ LULESH but use Chapel constructs wherever they can to help make the code more concise and compact. It was implemented as a single locale, multi-threading program. The function *LagrangeleapFrog* in *main* is responsible for 96% running time and most work is done underneath the forall parallel blocks inside its subroutines, therefore traditional code-centric profiler would only give limited insight while our profiler provides more insights about what to check and where to look for optimizations.

To compare our tool to prior approaches, we used pprof from gperftools, an existing code-centric profiler that works for Chapel, to profile the benchmark. The profile output of the top ten functions is shown in Fig. 4. The columns are:

1. Number of profiling samples in this function
2. Percentage of profiling samples in this function
3. Cumulative percentage of samples
4. Number of samples in this function and its callees
5. Percentage of samples in this function and its callees
6. Function name

```
Using local file ./lulesh.
Using local file prof.log.
Total: 17947 samples
   14180  79.0%  79.0%    14180  79.0% __sched_yield
     834   4.6%  83.7%      943   5.3% coforall_fn_chpl22
     694   3.9%  87.5%      694   3.9% __pthread_setcancelstate
     216   1.2%  88.7%      216   1.2% atomic_fetch_add_explicit__real64
     163   0.9%  89.6%      164   0.9% coforall_fn_chpl38
     160   0.9%  90.5%      272   1.5% CalcElemNodeNormals_chpl
     143   0.8%  91.3%      291   1.6% coforall_fn_chpl31
     123   0.7%  92.0%      586   3.3% coforall_fn_chpl19
     104   0.6%  92.6%      104   0.6% chpl_thread_yield
      95   0.5%  93.1%       95   0.5% _init
```

Figure 4. Profile Output of LULESH

As we can see, the output is a bit confusing. First, it mixes functions generated from the Chapel runtime libraries and the user code. The function that takes the largest portion of time on the list is *__sched_yield*, a system call that's referenced by Chapel threading layer. It's used to force the running thread to relinquish the processor and move the thread to the end of the queue; time spent in this function is often due to load imbalance or lack of parallelism elsewhere in the program. The only function that can be recognized by users on the list is *CalcElemNodeNormals,* which only consumes 0.9% of the total time and reveals limited optimization opportunities. Second, the information isn't fine-grained enough to identify the specific performance bottlenecks in the code. In comparison, the blame profiling result in Table VI is richer with variable-specific information, thus provides better guidance to user code optimizations.

The function *main* primarily serves as the highest level structure: initializing the test, starting the main loop that contains the core work, timing, and printing results. Almost

all the samples fell within the function *LagrangeLeapFrog*. The "Context" column in Table VI only lists the subroutines where the corresponding variables are defined. Note the sum of the blame for all variables is over 100%. As we briefly explained in section III, multiple variables could be blamed for a single sample. In this case, all the samples that were counted for *hourmodx's* will be assigned to *hx, shx,* and *hgfx* as well, thus these variables all get their own number of samples incremented. Therefore, the overall blame is larger than 100% for almost all programs as long as there is data dependency between variables. The roles of variables that are blamed most are introduced below.

TABLE VI. VARIABLES AND THEIR BLAME FOR THE RUN OF LULESH

Name	Type	Blame	Context
hgfz	8*real	30.8%	CalcFBHourglassForceForElems
hgfx	8*real	29.5%	CalcFBHourglassForceForElems
hgfy	8*real	29.2%	CalcFBHourglassForceForElems
shz	real	27.9%	CalcElemFBHourglassForce
hz	4*real	27.6%	CalcElemFBHourglassForce
shx	real	26.9%	CalcElemFBHourglassForce
shy	real	26.6%	CalcElemFBHourglassForce
hx	4*real	26.6%	CalcElemFBHourglassForce
hy	4*real	26.6%	CalcElemFBHourglassForce
hourgam	8*(4*real)	25.0%	CalcFBHourglassForceForElems
determ	[Elems] real	15.7%	CalcVolumeForceForElems
b_x	8*real	9.7%	IntegrateStressForElems
b_z	8*real	9.7%	IntegrateStressForElems
b_y	8*real	8.7%	IntegrateStressForElems
dvdx(y/z)	[Elems] 8*real	8.3%	CalcHourglassControlForElems
hourmodx	real	5.8%	CalcFBHourglassForceForElems
hourmody	real	5.1%	CalcFBHourglassForceForElems
hourmodz	real	4.8%	CaclFBHourglassForceForElems

hgfx(29.5%): LULESH is a symmetric 3-D simulation, thus we'll just use the x-axis variable to represent corresponding variables in all 3 dimensions later in the paper. Here, *hgfx* is an 8*real tuple, defined in *CalcFBHourglassForceForElems*. Together with *shx, hx, hourgam* and *hourmodx,* they compute the hourglass control force for each element.

determ(15.7%): The variable *determ* is a higher level data abstraction defined in *CalcVolumeForceForElems*. It's a local array with a domain being dynamically allocated on the heap every time the function is called. The same situation happens to the variable *dvdx*, which is defined in *CalcHourglassControlForElems*. We will explore the potential optimization opportunities of these variables later in the discussion.

b_x(9.7%): The variable *b_x* is also a 8*real (floating point double)tuple declared in *IntegrateStressForElems* and passed into *CalcElemNodeNormals* as a reference. The value of *b_x* is assigned through a nested function inside *CalcElemNodeNormals*. The call stack will lead us to the specific place where possible chances may exist for optimizations.

hourmodx(5.8%): The variable *hourmodx* is a local variable defined inside a nested for loop in function *CaclFBHourglassForceForElems.* It is used to calculate the value of *hgfx*. There is only one place in the code that writes to this variable, but it is updated frequently due to the loop and it acts as an important role in transferring blame.

```
for param i in 1..4 { //P 1
  var hourmodx, hourmody, hourmodz: real;
  // reduction
  for param j in 1..8 { //P 2
    hourmodx += x8n[eli][j] * gammaCoef[i][j];
    hourmody += y8n[eli][j] * gammaCoef[i][j];
    hourmodz += z8n[eli][j] * gammaCoef[i][j];
  }
  for param j in 1..8 {  //P 3
    hourgam[j][i] = gammaCoef[i][j] - volinv *
      (dvdx[eli][j] * hourmodx +
       dvdy[eli][j] * hourmody +
       dvdz[eli][j] * hourmodz);
  }
}
```

Figure 5. Code Snapshot of LULESH Hot Spot

Discussion and Optimization: From Table VI we discovered variables that hold the most blame in the program. After checking for the code, we found that *hgfx(y/z), shx(y/z), hx(y/z), hourgam* and *hourmodx(y/z)* have direct data dependency between them: *hgfx(y/z)* depends on the value of *shx(y/z); shx(y/z)* depends on *hourgam* and *hx(y/z); hx(y/z)* depends on *hourgam*, which ultimately depends on the value of *hourmodx(y/z)*. By further checking the code-centric data, it was discovered that over 21% of total time came from the loop block in Fig. 5. Therefore, optimizing this *for* loop is a good way to improve the overall performance.

TABLE VII. RESULTS FOR LOOP UNROLLING METHODS

Unrolling tag	Run time (s)	Speedup
Original	12.47	1.00
0 params	12.04	1.04
P 1	11.65	1.07
P 2	12.95	0.96
P 3	11.78	1.06
P1+P2	12.59	0.99
P1+P3	11.89	1.05
P2+P3	12.60	0.99
P1+U2	12.10	1.03
P1+U3	12.33	1.01
P1+U2+U3	12.75	0.98

'U x' means we manually do the unrolling for that for loop in place x

The keyword "param" before the loop iterator in Chapel causes the compiler to optimize the code by unrolling the loop. However, sometimes it would be counterproductive since it enlarges the code size. Therefore, we did multiple

controlled tests by preserving or eliminating these keywords in each location (denoted as "P #"). We further combined it with manual unrolling to see if that would be beneficial as well. The experiment results are displayed in Table VII.

Among all the options, we can see that simply keeping "param" for the outermost loop (P 1) gives us the best performance for this block of code. By shortening the execution time of this loop block, we expect to decrease the blame of those variables used in the loop, e.g., *hourmodx* and *hourgam*. This is proven in Table VIII.

The second optimization we made to the benchmark is from observing the variables *determ* and *dvdx*. At first, they seemed hard to optimize since the calculations of their values are deep down in the subroutines after their declarations. Without changing the algorithm, we can't simplify the computation. Fortunately, inspired by the optimization in Johnson's paper [19], we did Variable Globalization (VG). This optimization moves the declarations of several safe local variables to the global space so that they won't be dynamically allocated every time when the function is called. In this way, we saved about 19% execution time.

Another optimization we found through analyzing the profiling result is to work on variable *b_x(y/z)*. The values of *b_x*, *b_y,* and *b_z*, representing the "normal" from each face in the program, are computed in function *CalcElemNodeNormals* ("CENN" for short). Inside CENN, partial results are calculated through the nested function *ElemFaceNormal* and stored in temporary variables. Finally, the partial results from multiple *ElemFaceNormal* calls are added up through an addition operation on tuples. Since all temporary variables are tuple type, it involves tuple constructions and destructions, which are not cheap when they are nested deeply inside a big loop. We optimized this part by directly assigning intermediate results to the passed-in variables, thus avoiding redundant tuple constructions. This optimization denoted as "CENN" is able to reduce execution time by 7%.

Table VIII shows a profiling result comparison between each optimization we applied to the program. Instead of the default descending order, we group the variables that are affected by the same optimization. It gives us a clearer view of how a particular optimization would affect the profiling result of the relevant variables. The first optimization "P 1" reduces the computation time of that for loop, which directly affects variables *hourgam, hourmodx(y/z)*, indirectly affects variables like *hgfx(y/z),* etc. Therefore, we can see the decrease of the ratio of the above variables by comparing the 3rd and 2nd columns in Table VIII. The second optimization "VG" relates to *determ* and *dvdx* since it would reduce the number of times that these variables are declared and initialized. The total reduction in time brought by this optimization was achieved by hoisting many similar variables. The last optimization "CENN" focuses on simplifying the calculation of *b_x(y/z)*. By comparing the 5th and 2nd column in Table VIII of these three variables, there is an obvious drop in their weight, which also meets our expectation.

Table IX summarized the timing results of all versions of LULESH benchmark. The speedup column is the exclusive effect that the corresponding option achieves. Best case is the combination of all three optimizations. Overall, we achieved a factor of 1.4x speedup by modifying only 20~30 lines of source code.

We also list the "w/ --fast" column in Table IX. The overall speedup is bigger than that of "w/o --fast". The first and third manual optimization that we made obtain smaller speedups than before, that's probably because the "--fast" flag has already done some similar work for the original code, so our manual modifications gain less speedup.

TABLE VIII. PPROFILING RESULTS COMPARISON BETWEEN DIFFERENT OPTIMIZATIONS

variable name	Blame (%)			
	Original	***P1***	***VG***	***CENN***
hgfx	29.5%	20.5%	31.3%	26.4 %
hgfy	29.2%	18.8%	31.3%	27.4%
hgfz	30.8%	19.8%	28.0%	27.1%
shx	26.9%	18.1%	27.7%	23.08%
shy	26.6%	17.0%	28.0%	24.8%
shz	27.9%	17.4%	27.0%	24.4%
hx	26.6%	17.0%	27.7%	23.1%
hy	26.6%	16.3%	27.0%	23.4%
hz	27.6%	17.0%	27.0%	23.8%
hourgam	25.0%	13.2%	25.7%	22.1%
hourmodx	5.8%	2.8%	7.3%	6.4%
hourmody	5.1%	3.8%	6.1%	6.7%
hourmodz	4.8%	2.4%	8.3%	6.0%
dvdx(y/z)	8.3%	7.3%	8.2%	7.0%
determ	15.7%	20.8%	14.8%	16.1%
b_x	9.7%	10.4%	9.0%	6.0%
b_y	8.7%	10.1%	9.0%	6.0%
b_z	9.7%	10.8%	9.3%	6.0%

TABLE IX. RESULTS W/ OR W/O "--FAST" FLAGS

	w/o --fast		**w/ --fast**	
	Run Time(s)	***Speedup***	***Run Time(s)***	***Speedup***
Best Case	9.02	1.38	3.20	1.47
VG	9.98	1.25	3.39	1.39
P 1	11.65	1.07	4.54	1.04
CENN	11.57	1.08	4.59	1.02
Original	12.47	1.00	4.70	1.00

VI. CONCLUSION AND FUTURE WORK

Compared to traditional code-centric profilers, a data-centric profiler exposes the performance bottlenecks from a different view, thus giving the application programmers more insights to address performance issues. New parallel programming models provide newer abstractions for programmers. However, performance tools need to keep

pace with these changes to present useful performance information in an intuitive way.

We augmented the Blame [15] tool to enable its support to PGAS languages, using Chapel as an exemplar. In this paper, we introduced a state of art profiling tool that has been utilized to analyze the performance issues of HPC/PGAS programs. Compared to original Blame, we also support profiling multi-threaded programs.

We also demonstrated the functionality and usability of our tool on three well-known benchmarks. With the guidance supplied by the blame, we significantly improved the performance by factors of 1.4x for LULESH, 2.3x for MiniMD, and up to 2.1x for CLOMP, with minimal changes to the source code. We also concluded that domain remapping and zippered iterations are expensive to use.

There are several ways we want to extend the current work. First, we want to enhance the tool's support for more Chapel language features, like reduction operations and iterators. Second, we want to explore the multi-locale realm of Chapel; specifically, we plan to track the data mapping to different locales and blame communication cost back to key data structures. We need to keep track of which global variable that each piece of data in certain locale came from, and how they are transferred. Third, we will continue to explore more optimizations for benchmarks with the guidance of the profiling data and deeper understanding of the source code.

ACKNOWLEDGMENT

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research (ASCR), Scientific Discovery through Advanced Computing (SciDAC) program under Award Numbers ER26054, the ASCR X-Stack project under award Number ER26143, and partially by the Department of Defense through a contract with the University of Maryland. We would also like to thank our colleague Richard Johnson for sharing the pearls of wisdom in optimizing benchmarks.

REFERENCES

[1] Almasi G. PGAS (Partitioned Global Address Space) Languages. In Encyclopedia of Parallel Computing 2011 (pp. 1539-1545). Springer US.

[2] Carlson WW, et al. Introduction to UPC and language specification. Technical Report CCS-TR-99-157, IDA Center for Computing Sciences; 1999 May 13.

[3] Numrich RW, Reid J. Co-Array Fortran for parallel programming. In ACM Sigplan Fortran Forum 1998 Aug 1 (Vol. 17, No. 2, pp. 1-31). ACM.

[4] Krishnamurthy, et al. Titanium: a high performance Java dialect. In PPSC 1999.

[5] Chamberlain BL, Callahan D, Zima HP. Parallel programmability and the chapel language. International Journal of High Performance Computing Applications. 2007 Aug 1;21(3):291-312.

[6] Lydia Duncan, "Chapel: A Productive Parallel Programming Language", Women Techmakers Community Talks, January 19, 2016. [Online] Available: http://chapel.cray.com/presentations/Duncan-WomenTechmakers.pdf

[7] Shende SS, Malony AD. The TAU parallel performance system. International Journal of High Performance Computing Applications. 2006 May 1;20(2):287-311.

[8] Müller MS, et al. Developing Scalable Applications with Vampir, VampirServer, and VampirTrace. In PARCO 2007 Sep (Vol. 15, pp. 637-644).

[9] Adhianto L, et al. HPCToolkit: Tools for performance analysis of optimized parallel programs. Concurrency and Computation: Practice and Experience. 2010 Apr 25;22(6):685-701.

[10] Tallent NR, Kerbyson D. Data-centric performance analysis of PGAS applications. InProc. of the Second Intl. Workshop on High-performance Infrastructure for Scalable Tools (WHIST), San Servolo Island, Venice, Italy 2012.

[11] Su HH, Billingsley M, George AD. Parallel performance wizard: A performance analysis tool for partitioned global-address-space programming. In Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on 2008 Apr 14 (pp. 1-8). IEEE.

[12] Itahashi S, Sato Y, Chiba S. Toward a profiling tool for visualizing implicit behavior in X10. In 2014 X10 Workshop (X10'14) co-located with PLDI'14, Edinburgh, UK, 2014 June 12

[13] Oeste S, Knüpfer A, Ilsche T. Towards Parallel Performance Analysis Tools for the OpenSHMEM Standard. In Workshop on OpenSHMEM and Related Technologies 2014 Mar 4 (pp. 90-104). Springer International Publishing.

[14] Liu X, Mellor-Crummey J. A data-centric profiler for parallel programs. In 2013 SC-International Conference for High Performance Computing, Networking, Storage and Analysis (SC) 2013 Nov 17 (pp. 1-12). IEEE.

[15] Rutar NJ. Foo's To Blame: Techniques For Mapping Performance Data To Program Variables. Ph.D. dissertation, University of Maryland, 2011

[16] Mucci PJ, Browne S, Deane C, Ho G. PAPI: A portable interface to hardware performance counters. In Proceedings of the department of defense HPCMP users group conference 1999 Jun 7 (pp. 7-10).

[17] Dean J, Hicks JE, Waldspurger CA, Weihl WE, Chrysos G. ProfileMe: Hardware support for instruction-level profiling on out-of-order processors. InProceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture 1997 Dec 1 (pp. 292-302). IEEE Computer Society.

[18] Buck B, Hollingsworth JK. An API for runtime code patching. The International Journal of High Performance Computing Applications. 2000 Nov;14(4):317-29.

[19] Johnson RB, Hollingsworth JK. Optimizing Chapel for single-node environments. In2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) 2016 May 23 (pp. 1558-1567). IEEE.

[20] "CORAL Collaboration Benchmark Codes," Oak Ridge, Argonne, Livermore. [Online]. Available: https://asc.llnl.gov/CORAL-benchmarks/