ChplBlamer: A Data-centric and Code-centric Combined Profiler for Multi-locale 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
Parallel programming is hard, and it is even harder to analyze parallel programs and identify specific performance bottlenecks. Chapel is an emerging Partitioned-Global-Address-Space (PGAS) language that provides productive parallel programming. Most established profilers either completely lack the capacity to profile Chapel programs or generate information that cannot provide insightful guidance in a user-level context. To address this issue, we developed ChplBlamer to pinpoint performance losses due to data distribution and remote data accesses. We use a data-centric and code-centric combined approach to help Chapel users quickly identify performance bottlenecks in the source. To demonstrate the utility of ChplBlamer, we studied three multi-locale Chapel benchmarks. For each benchmark, ChplBlamer found the causes of the performance losses. With the optimization guidance provided by ChplBlamer, we significantly improved the performance by up to 4x with little code modification.

CCS CONCEPTS
• Computer systems organization → Distributed architectures • Software and its engineering → Software notations and tools

KEYWORDS
Data-centric profiling, Multi-locale Chapel, Benchmark optimization, PGAS, Performance evaluation

ACM Reference format:

1. INTRODUCTION
The traditional message passing based parallel programming for a distributed system is extraordinarily difficult and only expert programmers spending considerable time can hope to achieve good speedups. Shared memory multi-threaded programming represented by OpenMP is easier to write but is limited by the hardware to a single node. PGAS is an alternative programming paradigm that marries the good performance scalability of message passing with the good programmability of a shared memory model.

PGAS improves productivity in High Performance Computing (HPC) by introducing an additional layer of abstraction, described as a “global address space” for a distributed system. Because of this additional layer of abstraction, users do not have to explicitly handle the communication between nodes. Higher-level language abstraction frees users from non-trivial data partitioning, and discreetly arranging send and receive calls between nodes. For example, consider a common distributed vector addition \( C = A + B \), while MPI+OpenMP needs to carefully partition the vectors for each node based on different distribution schemes and insert pragmas to make use of multiple cores on each node. Chapel [18] users can simply finish the job using a forall loop and any domain distribution from the Chapel standard library.

The implicit communication and synchronization are usually managed by the compiler or the runtime library of a PGAS language. However, while higher level-abstractions increase productivity for HPC programmers, they also increase the difficulty of diagnosing performance bottlenecks that are now hidden from users. How to associate the performance metrics with original program elements becomes a critical issue in profiling PGAS programs.

Among different PGAS languages, we choose Chapel as the exemplar for our profiler because:

- Chapel is a promising language for productively solving large HPC problems with hybrid parallelism; however, currently, it still has much room for performance improvement, especially in distributed memory systems with extensive communication between locales\(^1\).
- Currently, there are no established profilers that have strong support for Chapel and provide insightful profiling result to Chapel end users.

\(^{1}\) “Locale” is a Chapel abstraction that represents a compute node, such as a multicore or SMP processor in most typical parallel architectures.
A traditional code-centric profiler is helpful in pinpointing hot spots in programs at the granularity of procedures or statements. However, it lacks the capability of associating performance losses with different variables accessed by specific lines of the code. In many cases, it is the data and its movement that cause the greatest performance loss rather than the computation. Again, let’s look at the vector addition example, code-centric profiling can tell you no more than the fact that the line “$C = A*B$” is consuming the majority of CPU time but data-centric profiling tells you which vector(s) is (are) causing the problem. Therefore, you can gain potential speedups by optimizing the access pattern or data layout of the problematic variables. The data-centric approach that relates performance to data structures rather than code regions is especially important for PGAS (e.g., Chapel) applications since they are usually memory-bound and communication-bound. However, these kinds of problems are usually hidden deep inside the runtime libraries and standard modules; so even with data-centric attribution, users can struggle to get insightful guidance for the user-level optimization. Therefore, a profiling tool that is able to identify these performance bottlenecks and associate the performance metrics to the user-level context is highly desirable.

In prior work [4], we proposed a data-centric performance measurement approach for single-locale Chapel programs. In this paper, we extend the work by providing a more functional data-centric and code-centric combined Chapel profiler, named “ChplBlamer”. ChplBlamer improves the prior work in several aspects:

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

Since our blame analysis relies on the LLVM Intermediate Representation (IR) of the target programs, our approach to mapping performance data to higher level abstractions is also applicable to other programming models including Domain-specific languages. To evaluate ChplBlamer, we studied three multi-locale Chapel benchmarks. We demonstrated how to investigate performance bottlenecks through ChplBlamer’s capability of data-centric and code-centric combined profiling. With our tool, we found new optimization opportunities for well-tuned benchmarks and gained significant speedups.

2. RELATED WORK

In this section, we discuss the state-of-art in HPC profiling, Chapel performance analysis, and data-centric attribution, and distinguishes ChplBlamer from the existing tools.

2.1 HPC profiling tools

Much prior work has analyzed High Performance Computing (HPC) applications, based on different profiling methods such as simulation, sampling, and direct instrumentation. TAU [5] and HPCToolkit [6] are two popular performance analysis tools that can be used for different languages on multi-core, single-node servers, and multi-node clusters. Scalasca [7] and Vampir [8] provide event tracing for large-scale parallel applications. Vampir also has an interactive visualization module that can be extended by other tools.


2.2 Chapel performance analysis

There are few profilers that truly support PGAS languages. Tallent and Kerbyson [20] developed a tool exclusively for Global Arrays, based on HPC Toolkit. It lacks a hierarchical mapping of the latency attribution to complex data structures. PPW [21] uses generic operation types instead of model-specific constructs in order to support multiple PGAS models in a single tool; however, Chapel is not one of the supported languages.

To our knowledge, there are only four prior tools that profile Chapel code. HPC Toolkit [6] 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 [12] attributes performance data to function level, but it does not distinguish generated functions from user functions and does not contain calling context.

Chplvis [16] is built for Chapel programmers. It visualizes the inter-locale communication and task computation of Chapel programs that help the user to discover the pitfalls of certain uses of parallelism in their code. However, it needs source modifications and it shows only the phenomena but not the causes of performance issues. Our previous work [4] is the only one that profiles Chapel code and presents the result in a completely user-level context. However, it also has several limitations. First, it does not support multi-locale Chapel programs, which largely limits the application of the tool since it misses the meat of PGAS. Second, it only supports synchronous parallel constructs in Chapel and cannot handle asynchronous tasking features such as a "begin" block. Finally, the way it instruments the Chapel runtime library and collects runtime information is not scalable, which makes it unsuitable for reasonably long-running executions. ChplBlamer addresses all these issues with substantially lower overhead.

Besides automated performance measurement approaches, there exist several manual studies of Chapel’s performance. Johnson and Hollingsworth [13] conducted several case studies of Chapel’s performance for a single-node environment. They chose OpenMP as the reference and hand-tuned the generated code. Kayraklıoğlu and El-Ghazawi [14] examined several language optimizations provided by Chapel on a set of multi-locale benchmarks and analyzed their impact on programmer productivity quantitatively. Haque and Richards [15] implemented CoMD in Chapel and demonstrated their competitive optimizations.
2.3 Data-centric attribution
Performance tools [9, 10, 11] employ data-centric attribution to diagnose memory-related bottlenecks in different aspects. HPCToolkit-data-centric [17] has gained success in profiling several HPC benchmarks. However, it only tracks static variables and heap variables that have a size of over 4k bytes and completely ignores stack variables. Additionally, the Chapel global variables are not correctly handled, resulting in over 95% of performance data being categorized into the “unknown data”, which is not useful for Chapel programmers. Instead, whatever complicated third-party library or runtime module the language uses, our data-centric profiler is able to let the performance issue rooted there bubble up to the user code.

3. BACKGROUND
ChplBlamer is based on Blame [19], which works on single-threaded C or FORTRAN programs. This section briefly describes the necessary background and the infrastructure that ChplBlamer is built on.

3.1 Blame definition
The blame of a variable is a percentage indicating the weight of that variable in terms of some performance metric, such as CPU clock cycles, cache misses, bytes transferred over a network. We use event-based sampling to collect the runtime information and static analysis to get the full data-flow information between variables. If a sample is triggered for an instruction that will contribute to the value of a given variable, then that particular variable is blamed for the sample. In short, for each variable \( v \), there are three concepts in the blame calculation:

- **BlameSet**: a set of statements (represented as line numbers) that affect the value of \( v \). It is the union of all backward slices for each instruction that writes to the memory region of variable \( v \) (including all \( v \)’s aliases, fields, and aliases of fields if \( v \) is a structure).
- **Blame Samples**: a set of samples attributed to \( v \).
- **Blame**: the proportion of the size of \( v \)’s Blame Samples over the total number of samples collected.

Consider the simple example in Code 1 that illustrates how the blame is calculated for each variable:

```plaintext
1 a = 8;          //s1
2 b = a * a;      //s2,s3
3 for (i = 0; i < 10; i++) //s4
4 b = b + i;
5 c = a + b;      //s5
```

**Code 1. Example code for blame calculation.**

First, using static analysis, we calculate the BlameSet of each variable. For example, variable \( b \) has a BlameSet of \{1, 2, 3, 4\} because \( b \) has two write statements on line 2 and 4, and it depends on the value of \( a \) and the loop iterator \( i \). For \( c \), even though there is only one write statement on line 5, it needs to include the BlameSet of \( a \) and \( b \) since its value depends on theirs. Next, suppose we collect 5 samples during the runtime, labeled from s1 to s5. We keep a sample in the Blame Samples of variable \( a \) if its corresponding line number is within the BlameSet of variable \( a \). Finally, we calculate the blame value. For example, there are 4 out of 5 samples that variable \( b \) is blamed for, so the blame of \( b \) is 80%, as shown in Table I.

<table>
<thead>
<tr>
<th>Variable</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>BlameSet</td>
<td>1</td>
<td>2,3,4</td>
<td>1,2,3,4,5</td>
<td>3</td>
</tr>
<tr>
<td>Blame Samples</td>
<td>s1</td>
<td>s1,s2,s3,s4</td>
<td>s1,s2,s3,s4,s5</td>
<td>s4</td>
</tr>
<tr>
<td>Blame</td>
<td>20%</td>
<td>80%</td>
<td>100%</td>
<td>20%</td>
</tr>
</tbody>
</table>

3.2 Tool framework
ChplBlamer leverages static analysis and dynamic analysis to find performance bottlenecks and implement data-centric attribution.

First, the static analysis component is implemented as an LLVM pass for Chapel programs’ LLVM bitcode. Graphs are used to represent intra-procedural blame relationships between variables. Each variable (including user-defined variables and compiler-generated variables) is represented as a vertex and each data dependency relationship between two variables is represented as a directed edge from the depending variable to the dependent variable. E.g., for instruction “store int a, int* b” we create one vertex for variable \( a \) and \( b \), respectively, and an edge from \( b \) to \( a \) because the value of \( b \) (here, \( b \) represents the memory space pointed by \( b \)) depends on the value of \( a \). For inter-procedural analysis, we use a mechanism called “transfer function” to propagate blame from the callee to the caller. This is done by establishing dataflow relationships between real parameters and function calls and keeping a set of “exit variables” for each function (“exit variables” are variables that have a scope out of the function, including global variables, incoming parameters that are pointers, and return values). In this way, the blame can be bubbled up along the call chain. The previous graphs are supplemented with inter-procedural edges. These inter-procedural edges are generated after we collected the runtime information.

Second, sampled execution utilizes the PAPI library as the interface to leverage the hardware counters. When the configured performance counter triggers a sample event, the profiler receives a signal and does a stackwalk from that point of the execution using libunwind.

Third, after the previous two steps, we have both static and runtime information of a Chapel program. We combine both information and calculate the blamed variables for each sample collected during the execution.

Lastly, sample profiles are aggregated and blame data for each user-defined variable (compiler-generated variables and that from libraries were flagged in static analysis and removed at this stage) is presented via a GUI.

4. CHALLENGES AND SOLUTIONS
Conducting data-centric profiling on multi-locale Chapel is far more challenging than the single-locale. As we explained in Section 1, Chapel, as a PGAS language, includes a runtime...
middleware to handle inter-node communication and data distribution. A single-line distributed vector addition statement “\( \mathbf{C} = \mathbf{A} + \mathbf{B} \)” in the Chapel source will be compiled to thousands of instructions in the IR. ChplBlamer collects the runtime data at the instruction level but aggregates this PGAS runtime middleware data and attributes the performance data back to the source level for Chapel programmers. To achieve that, we need to address several major challenges.

4.1 Challenges
First, for a variable that is distributed among multiple locales and requires remote access, there are hundreds of aliases and temporary variables representing a block of the data of the variable in the computation. How to identify those data blocks and finally aggregate their individual blame share to the original variable becomes a problem. Moreover, Chapel creates a unique private identifier (PID) for each distributed variable (e.g., \( \text{myVec} \) in Code 2) for future references. Therefore, when it’s accessed and passed through functions, the original logic to handle variables with a type of array or structure in [4] would fail since now the handles of those distributed variables are simply integers and will not be regarded as “exit variables”.

Second, at the IR level, 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. For example, communication calls such as \( \text{chpl_gen_comm_get} \) and \( \text{chpl_gen_comm_put} \), implicitly generate data dependency between the remote data and the local copy. We need to recover this hidden dataflow information to propagate blame properly. Moreover, explicit operations on distributed variables or within a parallel region at the source level will be wrapped into generated functions and implicitly invoked within Chapel runtime functions using function pointers at the IR level. The “transfer function” mechanism in [4] is not able to handle this case, thus it will fail the inter-procedural blame propagation.

Lastly, a multi-locale Chapel program does not launch the same execution from the main function on all locales simultaneously; rather only the master locale launches the execution from the very beginning and all other locales launch their jobs as needed during the entire course of execution (essentially a fork-join model). Therefore, when we walk the stack frame of a thread on a worker locale, it is very likely that the top stack frame (suppose the stack grows downwards) is not the “main” function but somewhere that particular task starts from. Missing the complete calling context precludes propagating blame along the call path appropriately. Moreover, Chapel’s asynchronous tasking feature aggravates this problem, since now a task can be created on Locale 1 at beginning of the execution and later remotely executed by Locale 2, while Locale 0 continues right after it launches that task.

Code 2 is a simple example showing the situation in multi-locale Chapel that the prior tool [4] cannot handle. The variable \( \text{myVec} \) is a distributed array defined on a cyclic distributed domain \( \text{myD} \); forall loop tries to leverage all threads on all locales to initialize \( \text{myVec} \); the begin block creates a new task on the current locale and the parent thread continues without waiting for the block to finish; the on clause launches a remote task on Locale 1 in an asynchronous style as well.

```
use CyclicDist;
const myD = {1..N} dmapped Cyclic(startIdx=1);
var myVec: [myD] real;
forall a in myVec
a = ..;
begin {
localCompute(myVec);
}
on Locales[1] do
remoteCompute(myVec);
```


4.2 Solutions
Since distributed variables are referenced via unique PIDs, we established a mapping between the vertex that represents the PID and the one that represents the original source variable when we build graphs in the static analysis. Every time a distributed variable is accessed, we can locate the corresponding PID node following the dataflow path in the IR. And with the link between the PID and the original variable, we are able to identify exactly which variable is being accessed. However, the biggest question is how do we figure out which nodes represent PIDs and link them to the corresponding variables (we use “object” to refer to the original source variable and all its compiler-generated aliases and temporary copies) since PIDs are no different than constant integers once created at the IR level. We solved that by tracking two critical functions from the Chapel runtime library: \( \text{chpl_getPrivatizedCopy} \) and \( \text{chpl_getPrivatizedClass} \). We found that these functions resolve PID from the corresponding object. Therefore, we are able to identify PIDs and their associated objects. Once we determine a PID and the object, we find all aliases\(^2\) of the PID as well as the aliases of the object backward and forward in this function. Now wherever an object alias is accessed, we can trace it back to the original source variable. Fig. 1 illustrates the idea of this process. At any access point to Object(i), we can follow the red path (the

\(^2\)“Alias” is not technically correct for PIDs since they are integers so “aliases” are just variable copies, but we use it for its literal meaning and the way we find them is similar to alias analysis.

Figure 1. The process of locating the original variable
top arrowed curve) to track down the original distributed variable that this Object(i) was derived from.

Since we won’t know if an integer is actually a PID until we meet one of those functions, then how does the top-level function know if the integer is actually used as a PID deeply down in the call path? The answer is that we won’t know in the intra-procedural static analysis step. We treat all integer parameters as potential PIDs and store their aliases before the execution. Later, in the postmortem analysis, we can get all the PIDs back frame by frame along the call path if any integer parameters are determined as PIDs in a certain frame.

In regard to the second challenge, we observed that non-user functions containing important dataflow information fall into two categories: module functions and runtime functions.

For functions from Chapel standard modules, since their definitions are also in the IR, we implemented a simplified blame analysis to figure out the blamed parameters that are responsible for any call to that function. For a few functions from Chapel runtime library, we manually figured out the blamed parameter indices since their function bodies are not included in the IR. In this way, we keep propagating blame to the callers via blamed parameters for calls to those functions.

To solve the problem of wrapper functions and function pointers, we conduct additional analysis on the program IR. First, we record the table of function pointers (with symbol names) that point to all generated wrapper functions in this program. Second, we extract parameters of three critical functions from the Chapel runtime library: chpl_executeOn, chpl_taskListAddBegin, and chpl_taskListAddCoStmt. The most important parameter of these functions is a constant integer that equals to the index of the corresponding function pointer in the table we previously recorded. In this way, we retrieve the exact wrapper function that will be called. Finally, we also need to match the parameters of those runtime functions to the real parameters that will be fed to the wrapper function. This is also tricky since the parameters are decomposed and reconstructed and it’s not a 1-1 correspondence. With all these efforts, we are able to mimic the explicit operations at the source level with the program IR and recover the dataflow information.

Lastly, in order to get the complete user-level calling context for each sample, we instrumented both the tasking and communication layers of the Chapel runtime using callback functions. In the tasking layer, we insert a callback in the function add_to_task_pool, so that every time a new task is added to the local task pool, we unwind the stack of the current thread and keep the unique function ID (referred as “fID”) for that task. The stacktrace shows the call path before a local task is executed. In the communication layer, we insert callbacks in function chpl_comm_execute_on, chpl_comm_execute_on_nb, and chpl_comm_execute_on_fast, so that every time a remote task is created and sent to another locale, we unwind the stack and keep the unique fID, as well as the locale IDs of the sender and receiver (referred as “sID” and “rID”) for the task. The stacktrace shows the call path before a remote task is launched.

During the instrumentation and sampling of the program execution, we also track the frame name as we unwind the stack. Once we find the top frame is one of the fork wrapper functions of a remote task defined in the Chapel runtime (e.g. “fork_wrapper”), we read the fID, sID, and rID from the function parameters, or simply fID if the top frame indicates a local task.

Finally, during the process of stacktrace concatenation, we use the above keys (fID, sID, rID) to find the call path before a certain task. The parent task is found iteratively until we see the user main function in the stack. In the meantime, we also remove frames that are not resolved to user functions so that the ultimate sample stacktraces are in a full user-level context. Fig. 2 shows the flowchart of this concatenation process; we use hash maps to retrieve stacktraces efficiently.

![Figure 2. Stacktrace concatenation flowchart](image)

The way we reconstruct the calling context for samples brings us two benefits:

First, it essentially solves the asynchronous and remote tasking problem. Since every piece of sequential work is a task in Chapel, with the keys (fID, sID, rID), we know what it does and where it was launched. If Locale 1 launched a task on Locale 2 at Time 1 and Locale 2 later executed the task at Time 2 where a sample is triggered, we can reproduce the calling context without interfering with the program execution.

Second, it significantly reduces the runtime overhead. While unwinding the stack in the callbacks, we also keep those keys in a set that is shared by all threads in the same locale. Therefore, the next time a task with the same key comes in, we need not unwind the stack again since the same information has been recorded. The prior Chapel tool [4] unwinds the stack every time a task list is executed (A task list is created for each “forall” or “coforall” parallel loop). That approach could not handle asynchronous parallelism and would incur unacceptable overhead in certain circumstances. Table II shows that this approach can reduce the average overhead from 3.5x to 14% for three single-locale Chapel benchmarks. The overhead is measured using the formula\(\left(\frac{\text{profiled execution}}{\text{clean execution}} - 1\right) \times 100\%\).

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>MiniMD</th>
<th>CLOMP</th>
<th>LULESH</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prior overhead</td>
<td>4.2x</td>
<td>1.4x</td>
<td>4.9x</td>
</tr>
<tr>
<td>Current overhead</td>
<td>5%</td>
<td>9%</td>
<td>27%</td>
</tr>
</tbody>
</table>

Table II. Tool overhead comparison on single-locale
5. ADDITIONAL FEATURES
We provide two new capabilities for the tool: exclusive blame and inter-node load imbalance check.

5.1 Exclusive blame
So far, the blame calculation is an inclusive data-centric profiling approach. This means that the blame value of a certain variable will absorb all the blame of its dependent variables. Therefore, the variables that hold the ultimate results will stand out in terms of weight. For example, variable c in Code 1 has a blame value of 100% since the whole block of code is to compute the value of c. After examining the code, we find that the most contribution to c is from b since c depends on b and b is responsible for all previous computation except for the last assignment to c. Therefore, optimizing the computation of b (line 2 to 4) may potentially gain better speedup than optimizing the single assignment to c (line 5).

To supplement the original inclusive blame, we provide another way of evaluating the weight of 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, more blame will be aggregated to computation-intensive variables, where usually more optimization opportunities may exist. Table III shows the process and result of exclusive blame calculation for Code 1. Now the most blamed variable is b, so we can quickly locate b’s first write statement with the multiplication as the potential performance bottleneck and try to optimize it.

Table III. Exclusive blame calculation for Code 1.

<table>
<thead>
<tr>
<th>Variable</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>BlameSet</td>
<td>1</td>
<td>2, 4</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>Blame Samples</td>
<td>s1</td>
<td>s2, s3</td>
<td>s5</td>
<td>s4</td>
</tr>
<tr>
<td>Total</td>
<td>20%</td>
<td>40%</td>
<td>20%</td>
<td>20%</td>
</tr>
</tbody>
</table>

5.2 Inter-node load imbalance check
There are three ways to view the data: a flat data-centric view, a traditional code-centric view, and a hybrid view using the concept of “blame point” where you can basically stop blame propagation at a certain point in the call path and reflect the performance statistics at that scope.

Besides the above three different views, we also include a view of workload information. Clicking on a particular variable in the data-centric view will pop up a window showing the total CPU seconds for that variable on each locale. Note that the total CPU time is the aggregation of all cores involved in that locale. You can also drill down from each node to display the specific samples that contribute to the time, which can be used with data profiles to verify the result. The different time on each locale shows the load imbalance situation in terms of this 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. Fig. 3 shows an example for variable Ab of HPL on 32 locales.

6. CASE STUDIES
We evaluated ChplBlamer on a local InfiniBand-based cluster Deepthought2. Deepthought2 consists of 484 nodes with dual socket (20 cores per node) Ivy Bridge 2.8 GHz processors. We used from 2 to 32 nodes in each case in this evaluation. We use CPU clock cycles as the sampling event and the sampling period is 1,073,807,359, which is a large prime. Taking HPL as an example, the time overhead of ChplBlamer ranges from 13% to 25% with 2 to 32 nodes, respectively; the space overhead is 90KB for problem size 500 with 32 nodes.

In this section, we studied three well-known multi-local Chapel benchmarks. All programs were built with Chapel 1.15 and the --fast (equivalent to “-O3” in GNU compilers) optimization. The description of each benchmark is as follows:

- HPL [1], the High Performance Linpack benchmark solves a uniformly random system of linear equations and reports time and floating-point execution rate using a standard formula for operation count.
- ISx [2] is the scalable Integer Sorting application. The Chapel version is fully Single-Program-Multiple-Data (SPMD), creating a task per locale and a task per physical core on each locale.
- LULESH [3] approximates the hydrodynamics equations discretely by partitioning the spatial problem domain into a collection of volumetric elements defined by a mesh. It has many implementations for most HPC programming models and languages, including Chapel.

We tried different Chapel configurations to get the fastest run time and use that as our performance baseline. There are two environment variables we tuned for performance: CHPL_TASKS (“fifo” or “qthreads” implementation as the Chapel tasking layer) and CHPL_RT_NUM_THREADS_PER_LOCALE (up to how many threads can be created per node). Based on our experimental results, we’ve concluded that fifo is better for HPL and LULESH while qthreads is better for ISx. Those values are what we used to measure the performance reported in the rest of this section. As for CHPL_RT_NUM_THREADS_PER_LOCALE, it only affects the fifo version and the details will be discussed case by case. We focused on the strong scaling study for HPL.

---

1 Ideally, we would not need to set this environment variable since we want fifo to spawn as many threads as it needs and qthreads internally creates fixed number of user-level threads. However, the performance difference makes it a worthwhile effort.
and LULESH (fixed problem size), and the weak scaling study for ISx (fixed problem size per task).

To clarify, while profiling, we only used “fifo” as the Chapel tasking layer and compiled all programs with --llv --no-checks (using the llvm frontend with no boundary checks). We did not use --fast in profiling since our intraprocedural analysis heavily depends on the generated LLVM bitcode of the Chapel program and --fast option loses too much debug information that we need to associate the IR-level objects (temporary variables and registers) with the source-level variables. We also discussed the optimization guided by our tool for each benchmark in detail. Since all execution time was measured for binaries built with --fast, we demonstrated that the optimization found by profiling non-optimized versions still helps in tuning the optimized versions.

6.1 HPL

![Figure 4. Data-centric blame for HPL on 2 locales. The red rectangles enclose interesting variables with their name, type, full calling context and blame percentage.](image)

Fig. 4 shows the blame for each variable as well as the context information in the source, including the name, type, and full call path to the point where the associated variable is declared. Note that there are some functions ending with numbers (e.g., “on_fi44” and “coforall_fi14”) in the calling context; they are auto-generated by the compiler to handle the parallel constructs (e.g., “forall”) in the program. Users can simply ignore those functions or treat them as code blocks of the nearest caller function when analyzing the result. First, we describe the purpose of the most blamed variables and functions (the corresponding blame is shown in the parentheses). Then we explain how we interpret the blame results in both code-centric and data-centric ways to discover the performance bottlenecks and scalability issues, and how we use this information to optimize the program.

\textit{Ab}(55.7\%): A 2D array allocated on distributed memory with the distributed domain \textit{MatVectSpace}. It holds the value of a matrix and a vector and is responsible for the calculation of most linear equations in the program.

\textit{MatVectSpace}(18.6\%): A 2D domain that represents the \textit{n x n} matrix adjacent to the column vector \textit{b}. It uses the BlockCyclic distribution to distribute \textit{Ab} to all nodes, leveraging the spatial locality in the blocked-computation. The default block size is 8, which can be tuned for performance in executions.

![Figure 5. Code-centric blame for HPL on 2 locales. The red rectangles enclose runtime functions that indicate special performance issues and the underscored are user functions to be optimized.](image)

Fig. 5 shows the inclusive code-centric result in the user-level calling context. Several functions are described below:

\textit{LUfactorize}(45.5\%): The function that consumes the most CPU cycles. It computes the blocked LU factorization with pivoting for matrix augmented with a vector of right-hand-side values. The computation is on a block by block basis.

\textit{schurComplement}(30.7\%): Computes the distributed matrix-multiplication. Each locale with a block of data updates itself by multiplying the neighboring left block to the upper block.

\textit{panelSolve}(13.4\%): Does unblocked-LU decomposition within the specified panel and updates the pivot vector for \textit{Ab}. It is difficult to optimize due to its unblocked data access pattern.

\textit{pthread_spin_lock}(14.3\%): A low-level synchronization function used by the Chapel runtime for concurrent operations on the shared memory. The percentage shows the overhead in the Chapel tasking layer.

\textit{chpl_comm_barrier}(7.5\%): A Chapel runtime library function used to implement implicit barriers in Chapel. The time spent on \textit{chpl_comm_barrier} indicates load imbalance in the program.

\textit{polling}(3.4\%): A specific task created on each locale to check for Active Messages (both requests and replies) which are in-bound to that locale. Time spent on \textit{polling} shows the communication overhead in the program.

\textbf{Discussion and Optimization:} Variable \textit{Ab} and \textit{MatVectSpace}, and function \textit{schurComplement} and \textit{panelSolve} are the most telling data about what is going on in the program. Table IV summarizes the profiling data of these program elements after executing on from 2 to 32 locales. CHPL_RT_NUM_THREADS_PER_LOCALE is set to be the number in the parentheses associated with the number of locales entry and unset otherwise, for the best performance on that number of locales. The Same denotation is used in Table V and Table XI. The relative weight change between variable \textit{MatVectSpace} and \textit{Ab} as more locales are used shows that the initialization cost of a distributed domain is very high. The increasing proportion of polling and \textit{chpl_comm_barrier} and the decrease of \textit{pthread_spin_lock} show that the inter-locale
Table IV. Major data-centric and code-centric blame percentages for HPL on different number of locales

<table>
<thead>
<tr>
<th>#Locales</th>
<th>Variable Name</th>
<th>Data-centric Blame</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Ab</td>
<td>55.7% 45.8% 36.4% 32.7% 23.6%</td>
</tr>
<tr>
<td></td>
<td>MatVectSpace</td>
<td>18.6% 27.8% 37.1% 38.7% 51.0%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Code-centric Blame</th>
</tr>
</thead>
<tbody>
<tr>
<td>schurComplement</td>
<td>30.7% 22.2% 17.3% 14.0% 8.7%</td>
</tr>
<tr>
<td>panelSolve</td>
<td>13.4% 15.7% 15.0% 15.8% 12.0%</td>
</tr>
<tr>
<td>polling</td>
<td>3.4% 11.5% 7.5% 11.7% 9.6%</td>
</tr>
<tr>
<td>chpl_comm_barrier</td>
<td>7.5% 10.5% 11.4% 11.5% 11.0%</td>
</tr>
<tr>
<td>pthread_spin_lock</td>
<td>14.3% 3.8% 7.1% 5.0% 4.5%</td>
</tr>
</tbody>
</table>

const DLow: int = panel.dim(2).low;
const DHigh: int = panel.dim(2).high;
const DRep: domain(2)
dmapped ReplicatedDist()={1..1, DLow..DHigh};
var AbRep: [DRep] elemType;
for k in panel.dim(2) {
    AbRep = Ab[k..k, DLow..DHigh];
    forall (i,j) in panel[k+1.., k+1..] {
        local {
            Ab[i,j] -= Ab[i,k] * AbRep[1,j];
        }
    }
}

Code 3. Loop optimization using replication for panelSolve.

We explored several ways to optimize the program. First, in schurComplement, we can enable the 'local' clause right inside the 'forall' loop to assert the local matrix multiplication and remove redundant communication calls. Code segments guaranteed to access only local data may be enclosed within a 'local' statement. The keyword restrains the compiler from generating wide pointers\(^4\) to access some distributed variables. This change reduced the total execution time by 3.1%. The speedups are summarized in Table V.

The optimization to function panelSolve is trickier as most of its computation needs to access remote data. We leveraged the ReplicatedDist module to create a local copy of a row in Ab on each locale to avoid frequent remote accesses within the loop. The modified code is shown in Code 3.

However, this optimization only got a speedup of 1.1x in the test on 4 locales. It is because the overhead caused by updating the local copy (AbRep) in every iteration cannot be compensated by the performance benefit it brings when the block size is not well tuned with different problem sizes and the number of locales, in which case the locality is not fully leveraged.

The HPL benchmark has been studied and highly optimized for years so it is hard to further improve the performance without a major change to the fundamental algorithm. However, we still gained some insights and speedups with the help of ChplBlamer.

Table V. Speedups of the localization optimization on a different number of locales for HPL

<table>
<thead>
<tr>
<th>#Locales</th>
<th>original (s)</th>
<th>localization (s)</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>13.67</td>
<td>18.01</td>
<td>1.01x</td>
</tr>
<tr>
<td></td>
<td>17.70</td>
<td>17.52</td>
<td>1.01x</td>
</tr>
<tr>
<td></td>
<td>19.30</td>
<td>17.98</td>
<td>1.07x</td>
</tr>
<tr>
<td></td>
<td>30.48</td>
<td>29.08</td>
<td>1.05x</td>
</tr>
</tbody>
</table>

6.2 ISx

The Chapel port of ISx is a newly developed benchmark based on the OpenSHMEM implementation [2]. Table VI shows the most blamed objects in both data-centric and code-centric profiling of ISx execution on 2 or 8 locales. The difference between 2-loc and 8-loc tells us which program objects (variable or function) are more affected by the communication and task synchronization cost.

$\text{myBucketedKeys}(41\%)$: It is a local variable in bucketSort, an array of configurable number (default 5,592,400) of keys. Every task allocates each one of this variable and populates the value in function bucketizeLocalKeys.

distribute $\text{myBucketedKeys}(41\%)$: An instance of the Barrier class in Chapel, it is used for task synchronization in the program. Since the current implementation of the Barrier standard module is not expected to perform well at scale, this variable becomes a major performance bottleneck when the number of tasks increases.

sendOffsets$(27\%)$ and $\text{bucketOffsets}(26\%)$: sendOffsets is an array of integers allocated on the master locale and bucketOffsets is the local copy of sendOffsets for each task and is used to compute $\text{myBucketedKeys}$ in function bucketizeLocalKeys.

Function $\text{bucketSort}$ is the core function to implement the sorting algorithm. It consists of 5 steps (each step is implemented by a sub-function): $\text{makeInput}$, $\text{countLocalBucketSizes}$, $\text{bucketizeLocalKeys}$, $\text{exchangeKeys}$, and $\text{countLocalKeys}$.

Discussion and Optimization: With the help of ChplBlamer, we easily identified the most "valuable" variables (such as $\text{barrier}$, $\text{myBucketedKeys}$, and $\text{myKeys}$) and functions (such as bucketizeLocalKeys, countLocalKeys) in terms of the performance optimization.

Optimizing the Barrier module would be the best thing to do since it affects the scalability largely and that is indeed part of the future work of the Chapel team. Here, we optimized the code using localization (localizing certain computation using the 'local' clause), similar to what we did for HPL. By tracking the most blamed variable $\text{myBucketedKeys}$, we found an opportunity for localization inside bucketizeLocalKeys. We enclosed all computation of that function in a 'local' statement. The same modification was done for the scan operation on variable sendOffsets. After the optimization, the blame percentage of those variables and functions also decreased correspondingly, as

\(^4\) Chapel uses "wide pointers" to point to non-local data.
shown in Table VII. Table VIII lists the speedups on a different number of locales with the simple modification.

<table>
<thead>
<tr>
<th>Table VII. Blame change before and after the optimization for related variables and function (bottom row)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>myBucketedKeys</td>
</tr>
<tr>
<td>sendOffsets</td>
</tr>
<tr>
<td>bucketOffsets</td>
</tr>
<tr>
<td>bucketizeLocalKeys</td>
</tr>
</tbody>
</table>

6.3 LULESH

The input data file for all tests is sedov15oct.lmesh, and we carefully tuned CHPL_RT_NUM_THREADS_PER_LOCALE for tests on a different number of locales. The best values are indicated in Table XI. Based on our limited experience, we found that the best value is always 4 when you allocate more than 8 locales. This experience shows the poor intra-node scalability of the program because the thread-level parallelism is not fully utilized. Table IX shows the data-centric blame result of LULESH.

<table>
<thead>
<tr>
<th>Table IX. Data-centric blame for LULESH</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable</td>
</tr>
<tr>
<td>Elem</td>
</tr>
<tr>
<td>x/y/z</td>
</tr>
<tr>
<td>x/y/z</td>
</tr>
<tr>
<td>fx/fy/fz</td>
</tr>
<tr>
<td>dvdx/dvdy/dvdz</td>
</tr>
<tr>
<td>x8n/y8n/z8n</td>
</tr>
<tr>
<td>elemMass</td>
</tr>
<tr>
<td>hgfx/hgfx/hgfx</td>
</tr>
<tr>
<td>shx/shy/shz</td>
</tr>
<tr>
<td>hx/hy/hz</td>
</tr>
<tr>
<td>dxx/dyy/dzz</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table X. Code-centric blame for LULESH</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function (caller-&gt;callee)</td>
</tr>
<tr>
<td>chpl_gen_main-&gt;chpl_user_main</td>
</tr>
<tr>
<td>chpl_user_main-&gt;CalcForceForNode</td>
</tr>
<tr>
<td>CalcForceForNode-&gt;CalcVolumeForceForElems</td>
</tr>
<tr>
<td>CalcVolumeForceForElems-&gt;CalcHourglassControlForElems</td>
</tr>
<tr>
<td>CalcHourglassControlForElems-&gt;CalcFBHourglassForElems</td>
</tr>
<tr>
<td>chpl_user_main-&gt;ApplyMaterialPropertiesForElems</td>
</tr>
<tr>
<td>chpl_user_main-&gt;CalcLagrangeElements</td>
</tr>
<tr>
<td>chpl_user_main-&gt;CalcQForElems</td>
</tr>
<tr>
<td>CalcLagrangeElements-&gt;CalcKinematicsForElems</td>
</tr>
<tr>
<td>CalcQForElems-&gt;CalcMonotonicQGradientsForElems</td>
</tr>
<tr>
<td>CalcVolumeForceForElems-&gt;IntegrateStressForElems</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table VIII. Speedups of the localization optimization on a different number of locales for ISx</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Locales</td>
</tr>
<tr>
<td>original (s)</td>
</tr>
<tr>
<td>localization (s)</td>
</tr>
<tr>
<td>speedup</td>
</tr>
</tbody>
</table>

elemToNode(60.4%): A large distributed array that supports the complement mapping between each element and its surrounding nodes (Node and Element are the two most important units for computation in the program; each Element has 8 neighboring Nodes by default). Therefore, elemToNode is accessed frequently during the entire course of execution by all nodes to retrieve the index information.

Element(37.0%), x/y/z(48.0%), and fx/fy/fz(35.6%): These are attributes of each Node, representing the coordinates, velocities, and forces in each dimension, respectively. They are calculated and updated frequently during the process of simulation.

Discussion and Optimization: We found some optimizations that speed up the program by a factor of 1.4x for LULESH on a single locale [4]. Therefore, we tried those optimizations first to see if they still benefit the performance in a multi-locale environment.

Two of our earlier optimizations from a single locale still help the multi-locale LULESH execution as you can see the speedups of “O1” over “original” in Table XI. The modification to function
CalcElemNodeNormals improves the performance by 6% by minimizing the construction and destruction of temporary tuples. However, the biggest contribution is by safely hoisting several distributed local variables such as dvdx/dvdy/dvdz, xzn/yzn/zdn, dxc/ycyd/yzd to the global space so that they won’t be dynamically allocated whenever the function that declares them is called. We call this optimization “globalization” in the following description. Globalization is very important to multi-local execution since creating and initializing distributed variables is expensive. Within a single locale, frequent data allocation and reclaim also cause thread contention, which is also bad for performance.

However, the scalability of LULESH is still not good enough although the execution time does seem to drop a little bit at 32 locales. We further examined the program by tracking the accesses of the most blamed variables. localizeNeighborNodes is an inline function that is called at multiple places to get the local copies of some attributes, like coordinates (x/y/z) and velocities (xd/ycyd/yzd) of the neighboring Nodes of an Element for the purpose of optimization. However, it performs 32 (when we use the default value 8 for the parameter nodesPerElem) remote data accesses for each Element in a sequential order because the neighboring Nodes may not be on the same locale with the Element. Besides, the function is called inside deeply nested loops, so it causes significant communication overhead.

Fig. 6 illustrates an example of such a case: The attributes of the blue Element and Nodes are stored on Locale 1 while the red ones are on Locale 2; the blue Element on the border of the two locales needs to access the red Nodes on Locale 2, which incurs a remote access.

Figure 6. It illustrates an Element-Node topology that would cause remote data accesses.

To fix the problem, we allocate 6 new array variables: x_map, y_map, z_map, xd_map, yd_map and zd_map to prestore the 8 neighboring Nodes for each Element. They use the same distributed domain Elems so that they can be read or written in a distributed parallel style, just like other Elems based distributed variables. Now except for the first call of localizeNeighborNodes in initMasses, we can remove all other calls to that communication-intensive function localizeNeighborNodes and simply do the copy to localize Node’s attributes.

To update those map variables as the execution continues, we create a function updateNeighborNodeMaps using full parallelism (all available threads) to do so once in each LagrangeNodal call. In this way, we avoided redundant remote accesses to the data that has been accessed before and we refer to this optimization as “replication”.

Replication brings more opportunities for localization, now we can enclose most computation into the “local” statement as long as they are within the loop iteration of same distributed domain Elems. We’ve found several functions that can benefit from this replication and localization combined optimization, such as CalcHourglassControlForElems. The performance improvement is shown in Table XI (“O2” is the combination of all optimizations: globalization, localization, and replication). Overall, we improved the performance of LULESH by a factor of 4x on 32 locales. Significantly, we move from having slowdown as more locales were added to having speedups.

Table XI. Speedups of optimization on a different number of locales for LULESH

<table>
<thead>
<tr>
<th>#Locales</th>
<th>2(12)</th>
<th>4(12)</th>
<th>8(4)</th>
<th>16(4)</th>
<th>32(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>original (s)</td>
<td>17.70</td>
<td>17.99</td>
<td>19.84</td>
<td>22.80</td>
<td>28.26</td>
</tr>
<tr>
<td>O1 (s)</td>
<td>14.89</td>
<td>13.40</td>
<td>14.73</td>
<td>14.51</td>
<td>11.29</td>
</tr>
<tr>
<td>speedup-01</td>
<td>1.19x</td>
<td>1.34x</td>
<td>1.35x</td>
<td>1.57x</td>
<td>2.51x</td>
</tr>
<tr>
<td>O2 (s)</td>
<td>11.73</td>
<td>9.74</td>
<td>8.15</td>
<td>8.20</td>
<td>7.10</td>
</tr>
<tr>
<td>speedup-02</td>
<td>1.51x</td>
<td>1.85x</td>
<td>2.43x</td>
<td>2.78x</td>
<td>3.98x</td>
</tr>
</tbody>
</table>

7. CONCLUSION

We described ChplBlamer, a profiler to identify, quantify, and analyze the performance bottlenecks in multi-local Chapel programs. Compared to previous Chapel data-centric profiling, ChplBlamer fully supports multi-local, asynchronous and remote tasking, provides richer information such as inter-node load imbalance, and incurs much lower runtime overhead, from 3.5x to 14%. Guided by ChplBlamer, we were able to pinpoint performance bottlenecks in three communication-bound multi-local Chapel benchmarks and identify the causes in the user-level context. We used three optimization techniques: globalization, replication, and localization to improve three benchmark codes. 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.

ACKNOWLEDGMENTS

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 Number 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.