Purity: An Integrated, Fine-Grain, Data-Centric, Communication Profiler for the Chapel Language

Richard B. Johnson
Department of Computer Science
University of Maryland, College Park
College Park, Maryland 20742
rbjohns8@cs.umd.edu

Jeffrey K. Hollingsworth
Department of Computer Science
University of Maryland, College Park
College Park, Maryland 20742
hollings@cs.umd.edu

***Abstract*—We present Purity, a configurable, data-centric, communication profiler for the Chapel language that analyzes memory and communication access patterns in a multi-node PGAS environment. By integrating Purity into the compiler and runtime framework of Chapel we can instrument Chapel programs to capture memory and communication operations and produce both online and fine-grain post execution reporting. Our profiler is equipped with a sampling mechanism for reducing overhead, handles complex data structures, and generates detailed execution profiles that map data motion to the variable, field, loop, and node levels for both distributed and non-distributed instantiations. In a case study, Purity provided valuable insight into task and data locality which allowed us to develop a programmatic solution for reducing nearly 90% of remote operations in SSCA#2.**

***Keywords*—dynamic analysis; static analysis; access patterns; communication operations; remote address; PGAS; Chapel;**

I. Introduction

Measuring the performance of parallel applications is important in order to ensure the best utilization of available system resources in a distributed environment. There are many performance aspects to consider when profiling program execution. The performance tuning on a per node basis is typically centered around CPU cores utilization, GPU processing, memory, and disk access. However, over an entire cluster the focus tends to shift to workload balance, task and data locality, and network latency and utilization.

High-level parallel languages such as Chapel need tools to better measure and understand task and data locality in a multi-node environment. What is missing in the HPC community is a fine-grain tool that can contextually map data motion back to the variables used within various sections of code to identify where bottlenecks exist. In this paper we present a data-centric, communication profiler called Purity which addresses these concerns by providing detailed profiling of program executions for the Chapel language.

II. Design

Our primary goal for the design was to develop a comprehensive tool that can analyze and profile memory and communication access patterns over a multi-node PGAS environment. The Chapel community has expressed keen interest in the development of a profiling system that is integrated into the Chapel framework. We have achieved both these goals with Purity.

To effectively profile PGAS access patterns, we determined that all communications need to be monitored during runtime execution and mapped back to their respective source variables. Each communication operation will contain references to a local and remote memory address. These addresses can be resolved if we establish an association between allocated memory ranges and variable definition identifiers. Generating this map will involve both static and dynamic analysis.

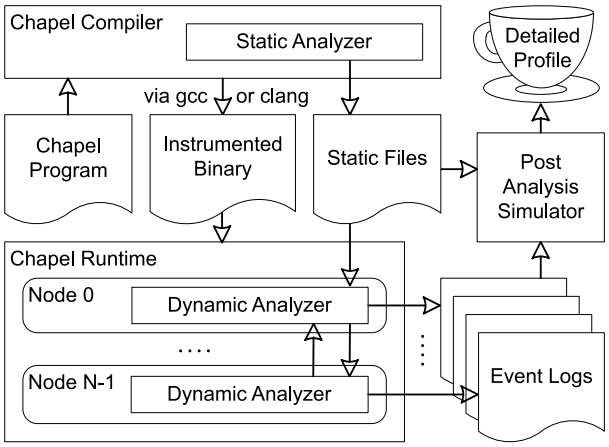

Fig. 1. Purity: Pipeline Overview

The static analyzer's primary job is to identify and instrument the source variables defined in the user modules which are relevant to remote communications. It will also generate static files pertaining to key aspects of the user modules of a Chapel program for use in a later stage of the analysis.

Dynamic analysis is responsible for associating memory ranges with variable definitions for use with mapping the remote addresses of communication operations during profiling. However, the prospect of building and maintaining a map of the entire partitioned global address space (PGAS) during runtime could range anywhere from cost prohibitive to downright intractable, depending on the program's needs and the number of nodes involved. First, it will incur a

large overhead on system resources of the principle node(s) which could slow program execution. Secondly, this scenario will require every node to report all of their heap operations over the network, including potentially even stack variable addresses during procedure invocations.

Thus in order to mitigate impact on system and network resources that profiling may incur, dynamic analysis must be performed independently on each node. However doing so leaves each analyzer with only a partial view of the PGAS. A unified view is required in order to resolve the definition identifier of a remote address for the corresponding communication. Therefore, each dynamic analyzer records the memory and communication operations of its host for post processing.

The post processing reads in the static files generated by the static analyzer in the compiler and the event logs produced by the dynamic analyzers on each node at runtime, performs a series of analyses, and produce a detailed profile for each execution of the instrumented Chapel program. The profile consists of a series of comma delimited files which provide a summary ranking variables by percentage of remote operations, loop analysis, request aggregation and byte throughput report at the node and flat index level, and a coverage report.

III. Implementation

The integration of Purity into Chapel involved the modification of the compiler with the introduction of five new passes for static analysis and a profile module. Incorporated into the runtime framework were the dynamic analyzer, a C implemented thread-safe ADT library, and a signal processing layer. Purity can be configured both through the command line and by environment variables. Purity was built from the Chapel 1.15.0 open-source distribution which includes the Chapel compiler, internal modules, runtime framework, and third-party dependencies.

A. Static Analysis

The static analyzer comprises of a source analysis pass, profile propagation, and four instrumentation passes. It also incorporates a profile module into the Chapel program. The profile module provides functionality for launching, updating, and finalizing dynamic analysis across the network during runtime. Through C interoperability with Chapel, the profile module also allows the compiler to link instrumented call expressions to dynamic analysis functions in the runtime framework.

Source analysis is performed at the earliest point possible in the compiler, right after parsing is complete, and has several responsibilities. First, it analyzes the abstract syntax tree (AST) structures to identify candidate variables and their definitions for instrumentation in later passes. Secondly, the pass stores relevant information pertaining to modules, class and record definitions, procedures, variables, and domains. Third, the body of each function definition is analyzed to identify and catalog loop hierarchies. Finally, its responsible for loading the profile module. The data gathered by source

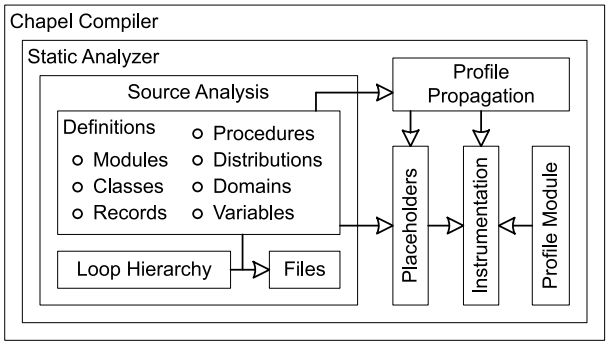

Fig. 2. Purity: Static Analysis

analysis is used to both inform procedure propagation and generate static files at a later stage.

Profile propagation was designed to persist profile information between compiler passes, disambiguate the different definitions of source variables when multiple definitions are generated by the compiler, and when required, spread contextual annotations down the chains of compiler generated temporary variables that originated from interactions with one or more candidate variables. The annotations of temporary variables aid in the instrumentation of candidate variable definitions and 'new' allocations in later passes so that memory addresses can be properly associated with the correct definition identifiers at runtime.

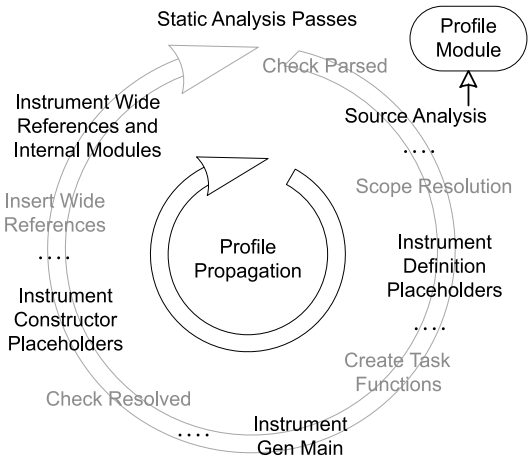

Fig. 3. Compiler Pass Constellation

After deep inspection we discovered that certain interactions with wide references in the compiler's AST representations will result in the generation of communication operations. A wide reference is a reference to a memory address in the PGAS environment that could either be local or remote. The problem is we are not able to determine which variables require instrumentation until after the 'insert wide references' pass which is a late pass in the compiler. Therefore, placeholders are needed for the preliminary instrumentation of candidate

variables.

Late stage instrumentation is part of the reason why the static analyzer requires an additional four passes. The first pass follows 'scope resolution' and adds definition placeholders to the ASTs. The second pass comes after 'create task functions' and is responsible for instrumenting the launching of the dynamic analyzer across all node, consolidation of online reporting, and the finalization of the analysis in compiler generated main or 'chpl_gen_main' which is managed and controlled by the main node. The third pass was inserted after 'check resolved' and instruments compiler generated class constructors with field definition placeholders. The final instrumentation is performed after 'insert wide references' and is responsible for the identification and replacement of placeholders for wide reference variables, as well as the instrumentation of ChapelArray and Atomics internal modules.

The Chapel compiler has two back ends, native C and LLVM. Since all instrumentation is completed before the 'codegen' pass, the profiler will support either Chapel option.

B. Dynamic Analysis

Dynamic analysis comprises of a memory tracker, pulse sampling, synchronization, and an event recorder. Pulse sampling uses signal processing as a means to provide an easy way to scale back the handling and recording of runtime operations. We describe the sampling mechanism later in this section. Dynamic analysis also provides online, high-level aggregate reporting of local, remote, cached, and prefetched operations. Each node will host a dynamic analyzer with a partial view of the PGAS and will report memory and communication operations to an event logger for post processing.

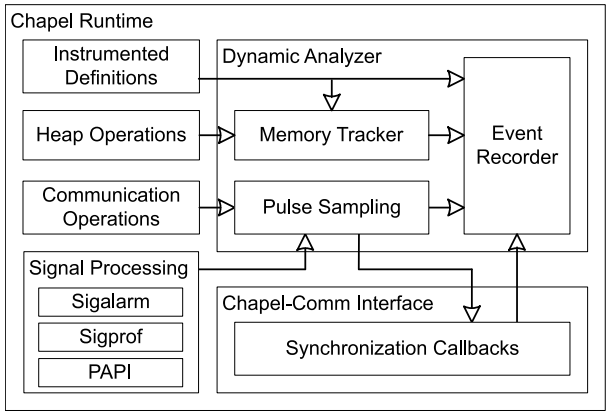

Fig. 4. Purity: Dynamic Analysis

Since the runtime framework was implemented in the C language, an abstract datatype (ADT) library was constructed to manage various aspects of the dynamic analyzer. Included in this library is a hashmap ADT which is thread-safe, supports dynamic growth, can be extended for multimap capabilities, and was designed so that all operations perform in constant time. Primarily it is used for task mapping and keeping track of the depth of parallelism to assist the memory tracker in the dynamic mapping of memory addresses for optionally either or both heap instantiations and stack variables with source variable definition identifiers. These structures will also assist in the resolution of variables effected by domain resizing during runtime regardless of whether the allocations are distributed or locally stored.

Pulse sampling can be enabled to reduce the analysis overhead and file size of generated event logs. While sampling is inactive, memory operations will still be processed and recorded but communications will not. The Chapel-communication interface layer is also informed so that unsampled communications will not generate synchronizations. Pulse sampling is controlled by a signal processing layer which supports options for 'sigalarm', 'sigprof', and PAPI. Given a sequence of intervals and callback functions, sampling can be enabled or disabled for controlled durations. Intervals for 'sigalarm' and 'sigprof' are in seconds, while PAPI intervals are measured in total instructions or 'PAPI_TOT_INS'.

Synchronization allows post analysis to model the memory state of each node for resolving communications. To achieve this, a monotonically increasing synchronization identifier is recorded with each operation and exchanged through piggybacking remote communications between nodes for synchronizing the event logs. Existing AM requests and AM handler functions inside the Chapel-GASNet interface were expanded to support both one-way and two-way sharing of synchronization identifiers. Direct calls to GASNet operations are followed by a short AM request instead to avoid having to modify all conduits in the third-party GASNet library. When 'chpl_comm_barrier' is called, all other nodes share synchronization information with the main node.

C. Post Analysis

The current version of the post analysis was written in Ruby and contains a parser for handling static and event logs, a simulator for synchronizing and maintaining a unified view of memory allocation states for each node in the cluster in order to resolve communications, and performs several analyses. First, it produces a summary of local, cached, prefetched, and remote get and put operations for each source variable tracked, which are ranked by percentage of total remote operations performed. Secondly, the loop hierarchy maps generated by the static analyzer are used in loop analysis to provide a breakdown of each variable ranked by the number of remote operations performed at each relevant loop. Third, request aggregation and byte throughput analysis drills down further by generating node and flat PGAS index matrices for each variable.

Finally, coverage analysis captures all unmapped heap allocations and unresolved communication operations, aggregates them by module and line number, and ranks them from most to least unmapped remote operations. Generally, 95% or better is considered acceptable coverage for the execution of any standard benchmark with a reasonable input size, with an

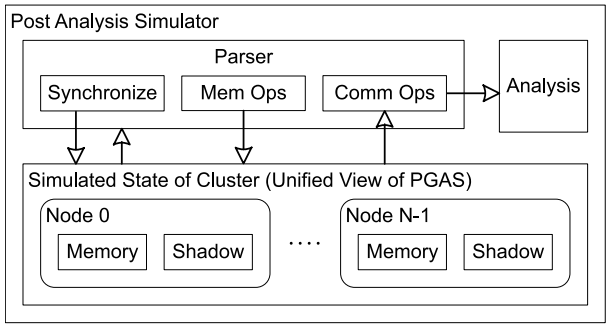

Fig. 5. Purity: Post Analysis

aim of 99% or better. To account for any slightly out of sync matching of communications and referenced allocations between the event logs that may arise due to network latency or nondeterministic thread behavior, freed heap allocations are removed from the host node's memory structure and then stored in a separate shadow memory structure for a few parsing cycles. This second channel mapping usually improves coverage by less than 1%.

Though rather complex, the true strength of Purity resides in its ability to account for everything. At each point in the program's execution and over the entire cluster Purity can answer:

- How was each communication handled?
- Which module and loop did the operation occur?
- Which nodes were involved?
- How many bytes were transmitted?
- Which variable instantiations were involved?
- Which definitions were they instantiated from?
- Which fields, subfields, and/or indices were affected?

In fact, the only thing the analysis can't answer is "What was the value stored at index X_i over the PGAS for a particular variable instantiation X at time T in the program's execution?", as this would require an enormous amount of time and space to record.

D. Configurations

For the compiler, profiling can be configured either at the command line or through environment variables. When choosing which integrated profiler option to instrument a Chapel program with, the –profile option has four types: cctimer, ccsignal, dcprof, dcsample. The 'dc' stands for data-centric communication profiler and 'cc' stands for code-centric performance profiler, 'signal' and 'sample' rely upon the signal processing layer for aggregating results. The cctimer and ccsignal options are based on our previous work [1]. The dcprof option will produce full event logs while with dcsample, the output file size can be scaled back. Throughout the course of the paper we will be focusing on the dcprof and dcsample options only.

chpl –profile=dcsample
CHPL_PROFILE=dcsample

The profile input option provides a filename to the static analyzer while compiling a Chapel program. When used in conjunction with dcprof or dcsample, the static analyzer can be directed with a finite definition list of variables to track. If no file is provided, by default all variables in the user modules will be scanned and targeted for instrumentation.

chpl –profile-input=<filename>
CHPL_PROFILE_INPUT=<filename>

At runtime, the profile settings for the dynamic analyzer can be preconfigured and reconfigured for each execution through environment variables. The signal processing layer provides three options: sigalarm, sigprof, PAPI which can be set using the profile signal and interval environment variables. The interval variable can be assigned one or more values as illustrated below.

CHPL_PROFILE_SIGNAL=sigprof
CHPL_PROFILE_INTERVAL=0.01,0.02

The following environment variables will direct Purity as what to log. In general, undefined variables or a variable assigned with a value of zero means no recording of that type will occur. Heap: 1: log only memory operations pertaining to memory addresses that can be associated with a valid variable definition identifier, 2: log all memory operations regardless of their associations. Comms: 1: log remote operations only 2: log both local and remote communications. Sync: 1: one-way, 2: two-way synchronization. Verbose: 0: event logs are populated by brief lines which are recognized by the post analysis simulator, 1: provides full information to go along with debugging or for special purposes. Below is the default configuration.

CHPL_PROFILE_RECORD_HEAP=1
CHPL_PROFILE_RECORD_STACK=1
CHPL_PROFILE_RECORD_COMMS=1
CHPL_PROFILE_RECORD_SYNC=2
CHPL_PROFILE_RECORD_VERBOSE=0
CHPL_PROFILE_PATHOUT=<program directory>

The profiler can also be configured to produce debug information using the follow environment variable.

CHPL_PROFILE_DEBUG=1

E. Challenges

The actual arguments of generated communication calls always involve compiler generated temporary variables and most of the time there is no easy programmatic way to know what these variables are actually referring to. We overcome this by tracking heap and potentially stack memory addresses to resolve the source variables in reference.

The symbol list and AST can best be thought of as an ever changing sea of identifiers and addresses where nothing can be known or relied upon from one pass to the next or sometimes even between multiple sections of the same pass. We use symbol and procedural propagation in an attempt to limit or mitigate these effects.

Both the local and remote addresses of all generated communication calls for atomics refer to compiler generated temporary

stack variables on both sides of the communication. The addresses of local temporary variables need to be associated with the respective atomic's definition identifier while the operation is being performed. This requires special instrumentation of atomic operations inside the internal Atomic module.

The compiler sometimes completely removes any reference to stack variables originally defined in the user modules and then replaces them with compiler generated temporary variables. Disabling copy propagation may fix the problem but it can also cause type incompatibility errors to occur in later passes.

The Chapel runtime environment is totally unaware of remote operations on the receiving side because the read and writes to remote addresses are handled deep within a third-party communication substrate and each substrate handles it differently. As a result, remote communications are only monitored on the sender's side.

It's not apparent always which variables will involve remote operations until after the insert wide reference pass in the compiler, especially when CHPL_GASNET_SEGMENT is set to 'everything'. This pretty much requires the insertion of placeholders early on to be followed up by late stage instrumentation of all relevant source variable.

IV. Evaluation

Two benchmarks, FFT from the High Performance Computing Challenge (HPCC) and SSCA#2 were employed on cluster Deepthought2 to assess the efficacy of Purity on a real system. Deepthought2 is a high-performance computing cluster hosted by the University of Maryland, College Park. It comprises of 444 Dell C8220, 40 C8220X, and 4 R820 Poweredge nodes which utilize a FDR infiniband interconnect [2]. Since Deepthought2 uses infiniband, Chapel communications were configured for GASNet with an IBV substrate, memory segment set to everything, and using MPI as the spawner.

Initially we evaluated the impact on program performance profiling and found to be around a 70% increase. Using the default configuration of the dynamic analyzer, we discovered an average of 69.22% is added to the execution's wall time when testing FFT on Deepthought2, using 4 nodes with 20 threads per node and an input of n=16. As a general rule, the overhead of the dynamic analyzer should grow linearly with the program time. Depending on the Chapel framework build configuration, nature of the Chapel program's interaction with the PGAS environment, and what the profiler has been set to log, the percentage may vary.

Similarly the growth and size of the event log files can vary depending the program, problem size, duration of execution, and the profilers configuration for burst sampling and what is to be recorded. Table I illustrates the average line size and operations recorded over all of the event logs generated by FFT using 4 nodes and an input size of n=16 while using burst sampling with intervals 0.01, 0.02 seconds. For this run the combined size of the static files are at 8,244 bytes, event logs sum to 10,975,920 bytes, and post analysis results are at 732,866 bytes. FFT is a fairly well optimized benchmark so 10 MB of event logs should represent a lower bound.

TABLE I
FFT Benchmark Event Log Record Sizes and Totals

Event Type	Average Record Size	Total Log Records
Heap Operation	29.54 Bytes	132,486
Stack Operation	26.16 Bytes	66,886
Communication	51.25 Bytes	28,349
Synchronization	16.73 Bytes	127,424

Deepthought2, 4 nodes, 20 threads per node, n=16
Using the default profile record configuration
and burst sampling with intervals=0.01,0.02

A. HPCC-FFT

The FFT benchmark was chosen from HPCC because it utilizes both block and cycle distributions to perform one-dimensional Discrete Fourier Transform (DFT) computations [3]. The implementation strategy uses radix-4 butterflies and involves dividing DFT into two phases which rely on different distributed memory layouts for eliminating remote operations when executing computations over 4^n nodes. The algorithm is initialized by an input vector called `z`. The first phase uses `Zblk` which is a permutated block distributed vector computed from `z` and the second phase uses a cyclic distribution `Zcyc`. After each phase the values are copied from one storage format to the other.

Runs were made using an input of n=16 or problem size of 65,536 over 4 nodes with 20 threads each. In both cases, the figures were obtained from the median result of a series of runs based on the total number of remote operations.

TABLE II
FFT Online Overview

Access	Operation	Normal	Cached
local	get	12,344,241	9,157,766
remote	get	222,641	127,444
cached	get	0	89,467
remote	prefetch	0	1,674
local	put	1,692,848	1,442,956
remote	put	107,279	2,376

Deepthought2, 4 nodes, 20 threads per node, n=16

Initially we compared enabling and disabling the Chapel PGAS cache to test the effectiveness of the automated remote cache option. This option can be enabled in the Chapel compiler by using –cache-remote. The cache provides aggregation, write behind, and read ahead features at runtime which can reduce remote communications. Table II provides an overview of local, remote, cached, and prefetch operations that were produced during the executions. The percentage of operations that were remote without caching were 2.3% and 1.22% when caching was enabled with 40.49% of potential remote operations recovered from cache. Also we found that runs with

caching enabled produced roughly 60% less remote operations than without caching.

TABLE III
FFT LOOP ANALYSIS - TOP RANKING VARIABLES

Loop		Remote		Local		% of
Module	**Range**	**Get**	**Put**	**Get**	**Put**	**Remote**
Zcyc, fft.chpl, 101						
fft.chpl	327-328	1877	0	432	0	17.19%
fft.chpl	117-118	1753	0	344	0	16.06%
fft.chpl	112-113	0	1152	0	215	10.55%
fft.chpl	324-325	0	1056	0	217	9.67%
Zblk, fft.chpl, 92						
ChapelArray	3562-3563	3709	0	740	0	33.97%
fft.chpl	327-328	0	0	0	1692	0.00%
fft.chpl	117-118	0	0	0	1526	0.00%
fft.chpl	112-113	0	0	374	0	0.00%
fft.chpl	324-325	0	0	348	0	0.00%
Twiddles, fft.chpl, 77						
ChapelArray	3562-3563	385	0	59	0	3.53%
fft.chpl	281-285	0	268	0	77	2.45%
fft.chpl	143-173	101	0	7820	0	0.93%

Deepthought2, 4 nodes, 20 threads per node, n=16

We then performed loop analysis for FFT. Table III provides a partial view of the results for the three top ranking variables. After comparing against the relevant source files, we determined that these loops can be categorized into two cases that account for 90.97% of all remote operations in this run.

```
112, 324:   forall (b, c) in zip(Zblk, Zcyc) do
113, 325:     c = b;
 ...
117, 327:   forall (b, c) in zip(Zblk, Zcyc) do
118, 328:     b = c;
```

Fig. 6. fft.chpl, Parallel mapping between block and cyclic distributions.

Case 1: 53.47% of total remote operations occurred on fft.chpl, lines 112-113 and 324-325 which uses parallel loops to map vector values from the block distributed domain of `Zblk` into the cyclic distributed domain of `Zcyc` and loops fft.chpl, lines 117-118 and 327-328 which map the reverse as shown in Figure 6. Furthermore, the fact that all operations on `Zblk` are local and 82.86% of accesses to `Zcyc` are remote, according to Table III, suggests that the parallel zip iteration feature in Chapel uses the domain of the first zip entry to determine task locality since all four loops use `Zblk` as the first entry.

Case 2: Another 33.97% of all remote operations have been reported to take place in a loop of one of Chapels internal modules ChapelArray.chpl, lines 3562-3563 in reference to `Zblk`. Further inspection allowed us to trace this back to fft.chpl, line 295 where bit reverse operations over the domain of `Zblk` remap it's values into `Perm`. ChapelArray.chpl, lines 3562-3563 loop in reference to `Twiddles` accounts for 3.53% of remote operations and also stems from the bit reverse operations in fft.chpl when applied through the `bitReverseShuffle()` procedure. The benchmark uses this function to permutate the values of distributed vectors prior to DFT computations.

```
108:  bitReverseShuffle (Zblk); // from main()
 ...
261:  bitReverseShuffle (Twiddles); // from initVectors ()
 ...
293:  proc bitReverseShuffle (Vect: [?Dom]) {
294:    const numBits = log2(Vect.numElements),
295:    Perm: [Dom] Vect.eltType = [i in Dom]
            Vect( bitReverse (i , revBits =numBits));
296:    Vect = Perm;
297:  }
 ...
322:  bitReverseShuffle (Zblk); // from verifyResults ()
```

Fig. 7. fft.chpl, bitReverseShuffle() and its invocations

After further analysis of the AST and intermediate representation (IR), we determined in both cases that the remote accesses are intentional and not incidental, e.g. not stemming from an unintended consequence of some implicit feature of the language or nonoptimal compiler generation. We also concluded that there is no programmatic approach that will reduce remote operations in these loops any further. In other words, the developer(s) clearly understood how to apply the language features of Chapel in the most efficient way to achieve good task locality. Therefore, no major improvement can be made for the FFT benchmark. However, this analysis still demonstrates the effectiveness of Purity to isolate variable usage in sections of code associated with the highest dependency on remote memory in the PGAS environment, which leads to insight into what is actually occurring during runtime execution of a Chapel program.

B. SSCA#2

The next benchmark we evaluated was SSCA#2 which stands for Scalable Synthetic Compact Applications graph analysis, version 2 [4]. The benchmark generates and performs a series of approximate betweenness centrality (BC) computations over a weighted, directed multigraph given a set of starting vertices.

TABLE IV
SSCA#2 ONLINE OVERVIEW

Access	**Operation**	**4N 20T**	**8N 10T**
local	get	164,906,162	78,732,860
remote	get	11,639,989	3,667,122
local	put	65,107,050	12,997,944
remote	put	599,594	761,280

Deepthought2, N nodes, T threads per node,
scale=8, low scale=5, high scale=8

Tests were performed on Deepthought2 under two different cluster configurations, 4 nodes with 20 threads each and 8 nodes with 10 threads each. Once again, median results were used. The overviews in Table IV produced at runtime found a percentage of remote to total operations to be 5.05% and 4.61% respectively for the two configurations.

Through loop analysis we discovered that 42.14% of all remote operations occurred in the loop at SSCA2_kernels.chpl on line 582 (see Table V and Figure 9). In order to understand what this means we first need to look at how starting vertices and their corresponding tasks are delegated in the BC algorithm.

```
279:  forall s in starting_vertices do on
          vertex_domain.dist.idxToLocale(s) {
...
286:      const tid = TPVM.gettid();
287:      const tpv = TPVM.getTPV(tid);
```

Fig. 8. SSCA2_kernels.chpl, from Approximate Betweenness Centrality

The outer most parallel loop of BC delegates tasks over starting vertices to the respective node on which that vertex is stored as illustrated in Figure 8 in order to maximize task and data locality in a parallel environment. Each task requires a set of variables for processing its assigned starting vertex and these variables are managed by a `taskPrivateData` class. In order to reuse the instantiations of `taskPrivateData` for future tasks, task private variable array or `TPV`, a block distributed vector of `taskPrivateData`, is allocated and managed by `TPVM`. `TPVM` is an instantiation of `TPVManager` class which during the invocation of `gettid()` is responsible for yielding thread control back to the runtime framework until the requested element in `TPV` is no longer in use by another task and can be obtained and safely used by the requesting task.

TABLE V
SSCA#2 LOOP ANALYSIS - TOP RANKING VARIABLES

Loop or Line		Remote		% of
Module	**Range**	**Get**	**Put**	**Remote**
TPVM.TPV, SSCA2_kernels.chpl, 577				
SSCA2_kernels.chpl	582-582	4544	0	42.14%
SSCA2_kernels.chpl	586	7	0	0.06%
SSCA2_kernels.chpl	589	5	0	0.05%
SSCA2_kernels.chpl	581	5	0	0.05%
taskPrivateData::barrier.tasksFinished, SSCA2_kernels.chpl, 600				
SSCA2_kernels.chpl	345-423	795	0	7.37%
SSCA2_kernels.chpl	456-463	192	0	1.78%
SSCA2_kernels.chpl	331-463	44	0	0.41%
taskPrivateData::BCaux, SSCA2_kernels.chpl, 569				
SSCA2_kernels.chpl	550	252	0	2.34%
Atomics.chpl	1260	0	146	1.35%
Atomics.chpl	1396	0	97	0.90%
SSCA2_kernels.chpl	331-463	87	0	0.81%
Atomics.chpl	1240	0	56	0.52%

Deepthought2, 4 nodes, 20 threads per node, scale=5, low scale=3, high scale=4

The while condition of the loop in `gettid()` (see Figure 9 again) first accesses the `this` reference of the `TPVManager` class, followed by the class field `TPV` and then performs a get index operation over a distributed array to obtain a `used` field in `taskPrivateData` class and invoke a test and set atomic operation.

TABLE VI
SSCA#2 REMOTE REQUESTS

TPVM.TPV, SSCA2_kernels.chpl, 577				
	Sender			
Receiver	n0	n1	n2	n3
n0	0	1282	1601	1678

Deepthought2, 4 nodes, 20 threads per node, scale=5, low scale=3, high scale=4

Node-level request aggregation for `TPVM.TPV` in Table VI shows that all remote operations access the memory of the main node from other nodes, which indicates that `TPVManager` and the underlining Chapel framework structures that manage the `TPV` array were instantiated and only exist on the main node.

To confirm this, we used –no-inline to disable the inline pass in the compiler which revealed that most of the remote get operations for `TPVM.TPV` occurred inside the internal module ChapelArray.chpl on line 1978. This line involves acquiring an array instance field, which references a subclass of `BaseArr`, in order to access its type and also obtains the privatized identifier field to be used as arguments for looking up the address of a privatized class. Remote get operations are performed every time other nodes attempt to access these fields.

```
576:  class TPVManager {
579:    proc gettid() {
580:      const tid = this.currTPV.fetchAdd(1) % numTPVs;
581:      on this.TPV[tid] do
582:        while this.TPV[tid].used.testAndSet() do
              chpl_task_yield();
583:      return tid;
584:    }
```

Fig. 9. Original: SSCA2_kernel.chpl

```
576:  class TPVManager {
579:    proc gettid() {
580:      const tid = this.currTPV.fetchAdd(1) % numTPVs;
581:      on this.TPV[tid] do {
582:        const t = this.TPV[tid];
583:        while t.used.testAndSet() do chpl_task_yield();
584:      }
585:      return tid;
586:    }
```

Fig. 10. Optimized: SSCA2_kernel.chpl

With these insights in mind, a simple programmatic solution was devised in order to eliminate redundant remote communications. Figure 9 shows the original `gettid()` function in the SSCA#2 kernel and Figure 10 illustrates the optimization that was discovered. Since the instantiation of `TPVManager` and the underlining structures that manage `TPV` reside in memory on the main node, resolving `this` for `TPVManager`, its member variable `TPV`, and the corresponding element at

index `tid` over the PGAS before entering the while loop will significantly reduce redundant remote operations generated by other nodes. Test and set operations can still be performed on the network atomic `used` inside the loop, which will allow `gettid()` to appropriate the next `tid` to a requesting task.

TABLE VII
OPTIMIZED SSCA#2 ONLINE OVERVIEW

Access	Operation	4N 20T	8N 10T
local	get	58,274,483	65,646,873
remote	get	814,509	1,148,972
local	put	242,889,900	32,550,756
remote	put	599,449	760,957

Deepthought2, N nodes, T threads per node,
scale=8, low scale=5, high scale=8

The optimized output in Table VII shows the percentage of remote to total operations for the optimized version to be 0.47% and 1.91% respectively with a reduction of 88.45% and 56.87% of remote operations from the original version of SSCA#2.

V. RECOMMENDATIONS

The finding in SSCA#2 provides a classic example where a programmatic solution can significantly reduce remote operations. In this case, accessing `TPV` required a remote get operation to resolve `_pid` for `record _array` in ChapelArray.chpl which is an underlining structure that manages the distributed array for `TPV`. Then a second remote get operation was needed to acquire the `TPV` element at index `tid`. By resolving the `TPV` element before entering the loop, redundant remote operations can be avoided. Further testing on Deepthought2 indicated that enabling automated remote caching does not alleviate the problem. Therefore, either it must be handled explicitly by the developer or an automated solution in the compiler is needed to identify, target, and mitigate these cases.

VI. RELATED WORK

Hui Zhang [5] has developed a data-centric performance profiler based on his preliminary work [6]. He employs an algorithm originally proposed by Nick Rutar [7] which assigns a percentage of "blame" to variables based on their involvement in computational impact on performance. His profiler handles heap, stack, and local variables and can also manage complex data structures. Using PAPI it provides a data sampling mechanism for reducing overhead. However, this version of his profiler can only profile Chapel programs in single-node environments. In contrast, Purity is a multi-node, data-centric communication profiler which focuses on resolving local and remote memory addresses of communication operations to determine which variables were involved. It is not concerned with CPU usage or program wall time. In fact Purity has no concept of time at all. For instance, it does not rely on the CPU times produced by a cluster of nodes since there is no guarantee that the clock times will be synchronized. Instead Purity relies upon a system that exchanges monotonically increasing synchronization identifiers between nodes to establish an ordering of events. It employs a simulator which models a unified view of the memory allocation environment of the entire PGAS over the course of events while performing fine-grain post analysis.

Parallel performance wizard (PPW) [8] is a profiler that provides support for different parallel programming models which incorporate PGAS, such as UPC and SHMEM. Highlights of PPW which peaked our interest were the Event Type Mapper and Visualization Manager. For Purity, we considered adding an analysis to characterize the communication access patterns of each variable within the execution of various loops and sections of code. We also considered developing a GUI application for post analysis that can produce visualizations similar to the data transfers and array distribution visualizations in PPW.

Some of the other profiling approaches we looked at were Tallent and Kerbyson [9] which is based on HPCToolkit [10] and supports PNNL's Global Array programming model, Liu and Mellor-Crummey [11] also utilizes the HPCTookit, Itahashi et al. [12] proposed a design of a profiler for the X10 PGAS language, and Oeste et al. [13].

VII. FUTURE WORK

Ideally, we want to release Purity with future versions of Chapel as a production ready tool. However, this comes with its own set of challenges. Our initial approach was to implement and integrate the profiler in a way that least impacted the Chapel source distribution. However, painfully we learned that this is not always possible when devising solutions to overcome some of the problems we encountered. Unfortunately our solutions introduced many changes which required modifications throughout the compiler and to a less extent the runtime framework. A clearer approach for implementation is likely necessary in order for the Chapel community to support Purity in future releases.

Secondly, many approaches were attempted at the compiler level which produced a great deal of dead code and in some cases inoptimal code. The implementation of profile propagation in the static analysis needs to be cleaned up as it currently impacts the performance of the compiler by up to an order of magnitude when enabling Purity. Obviously this needs to be optimized before Purity can be released.

Third, we are planning on developing a GUI in C/C++ for post analysis that would allow a way for the user to scroll to any point in the program simulation, provide a detailed drill down interface, and given a set of parameters can produce intuitive visualizations. The GUI could illustrate either a 3D height or heat map for request aggregation and byte throughput analysis for visualizing node and flat PGAS index matrices of a target variable, either within a specified section of code or over the entire execution.

VIII. Conclusion

We have developed and integrated into the Chapel framework a configurable, data-centric, communication profiler called Purity that analyzes memory and communication access patterns over a multi-node PGAS environment. Our tool provides online reporting, accounts for nested loop hierarchies, handles tracking for complex data structures, and reduces overhead with pulse sampling. Purity produces scalable, fine-grain profile results at the variable, field, loop, node, and flat index level for both distributed and non-distributed instantiations. Using the FFT and SSCA#2 benchmarks, we have demonstrated the efficacy of Purity to identify PGAS bottlenecks and provide the developer with a better understanding of where PGAS dependencies reside and how task and data locality behaves in their Chapel program. During the evaluation, Purity provided us with valuable insight into a simple programmatic solution that reduced remote operations for SSCA#2 by up to 88%.

IX. Acknowledgements

We would like to thank Michael Ferguson, Cray Inc. for his insights in how the Chapel compiler works.

References

[1] R. Johnson and J. K. Hollingsworth, "Optimizing Chapel for Single-Node Environments," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, May 2016, pp. 1558–1567.

[2] "The Deepthought2 HPC cluster," University of Maryland, College Park. [Online]. Available: https://www.glue.umd.edu/hpcc/dt2.html

[3] "HPC Challenge Benchmark," HPCC. [Online]. Available: http://icl.cs.utk.edu/hpcc/

[4] D. A. Bader, J. Feo, J. Gilbert, J. Kepner, D. Koester, E. Loh, K. Madduri, B. Mann, and T. Meuse, "HPCS Scalable Synthetic Compact Applications #2 Graph Analysis," Tech. Rep., September 2007. [Online]. Available: http://www.graphanalysis.org/benchmark/HPCS-SSCA2_Graph-Theory_v2.2.pdf

[5] H. Zhang and J. K. Hollingsworth, "Data Centric Performance Measurement Techniques for Chapel Programs," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2017, pp. 377–386.

[6] ——, "Toward a Data-Centric Profiler for PGAS Applications," in *2015 9th International Conference on Partitioned Global Address Space Programming Models*. IEEE, 2015, pp. 93–95.

[7] N. J. Rutar, "Foo's to blame: Techniques for mapping performance data to program variables." ProQuest Dissertations Publishing, 2011.

[8] H.-H. Su, M. Billingsley, and A. D. George, "Parallel performance wizard: A performance analysis tool for partitioned global-address-space programming," in *2008 IEEE International Symposium on Parallel and Distributed Processing*. IEEE, 2008, pp. 1–8.

[9] N. R. Tallent and D. Kerbyson, "Data-centric Performance Analysis of PGAS Applications," in *Proc. of the Second Intl. Workshop on High-performance Infrastructure for Scalable Tools (WHIST)*, 2012.

[10] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "HPCToolkit: tools for performance analysis of optimized parallel programs," in *Concurrency and Computation: Practice and Experience*, April 2010, pp. 685–701.

[11] X. Liu and J. Mellor-Crummey, "A data-centric profiler for parallel programs," in *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2013, pp. 1–12.

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

[13] S. Oeste, A. Knupfer, and T. Ilsche, "Towards Parallel Performance Analysis Tools for the OpenSHMEM Standard," in *Workshop on OpenSHMEM and Related Technologies*. Springer International Publishing, March 2014, pp. 90–104.