Alembic: Automatic Locality Extraction via Migration

Brandon Holt Preston Briggs Luis Ceze Mark Oskin
University of Washington
{bholt, preston, luisceze, oskin}@cs.washington.edu

Abstract

Partitioned Global Address Space (PGAS) environments simplify writing parallel code for clusters because they make data movement implicit – dereferencing global pointers automatically moves data around. However, it does not free the programmer from needing to reason about locality – poor placement of data can lead to excessive and even unnecessary communication. For this reason, modern PGAS languages such as X10, Chapel, and UPC allow programmers to express data-layout constraints and explicitly move computation. This places an extra burden on the programmer, and is less effective for applications with limited or data-dependent locality (e.g., graph analytics).

This paper proposes Alembic, a new static analysis that frees programmers from having to manually move computation to exploit locality in PGAS programs. It works by determining regions of code that access the same cluster node, then transforming the code to migrate parts of the execution to increase the proportion of accesses to local data. We implement the analysis and transformation for C++ in LLVM and show that in irregular application kernels, Alembic can achieve 82% of the performance of hand-tuned communication (for comparison, naïve compiler-generated communication achieves only 13%).

Categories and Subject Descriptors D.3.4 [*Processors*]: Compilers, Run-time environments; D.3.2 [*Language Classifications*]: Concurrent, distributed, and parallel languages

General Terms Compilers; Distributed systems; Languages

Keywords PGAS; LLVM; Locality; Thread migration; Continuation-passing style

OOPSLA '14, October 20–24, 2014, Portland, OR, USA.

http://dx.doi.org/10.1145/2660193.2660194

1. Introduction

When targeting distributed systems, such as commodity clusters, application developers must deal with both parallelism and locality. Often these are at odds as placing more data on a single machine node improves locality but may decrease the ability to exploit parallelism across the entire system. This lack of separability forces programmers to reason about these two conflicting drivers of performance in tandem.

Partitioned Global Address Space (PGAS) [8–10] languages simplify the expression of parallel computations on large distributed systems. The programmer writes to a shared memory model, and under the hood the runtime system manages the movement of data. PGAS languages provide the concept of a *global* pointer that can reference memory anywhere in the system. Programmers can manipulate and dereference these just like normal pointers. While elegant, PGAS systems do not remove the fundamental conflict between parallelism and locality; in fact, they can easily lead to less efficient applications [14]. Compared with expressing all data movement manually, PGAS models may hide cases where the way the algorithm is expressed leads to excessive communication.

Thus even in a PGAS system, programmers wishing to exploit the last ounce of performance must manage locality themselves. The typical way to do this is to carefully layout data structures such that blocks of data accessed together are placed together and then when spawning new threads, explicitly place them where most of the data the thread will access is located. This is not an ideal solution for two reasons: (1) it can make otherwise-elegant PGAS implementations excessively complex with explicit *computation movement* – in effect, instead of explicitly moving data around with MPI invocations, the programmer is explicitly moving computation around using a variety of techniques (spawning new threads, continuations, etc); and (2) not all applications are amenable to easy partitions of computation and data – notably, irregular graph algorithms lack spatial locality, so placing the computation at any fixed location in the system guarantees several remote accesses and poor performance.

This paper introduces Alembic, a compilation technique for PGAS systems that automatically extracts locality from programmer-created threads. At a high level, Alembic statically analyzes the code looking for sequences of memory references that go to the same node in the distributed system.

It then transforms a single programmer-directed thread into a series of component threads, each spawned on the machine node that hosts the majority of the data that component thread will access. Synchronization is added to ensure the sequential semantics the programmer has expressed are maintained, and communication is inserted between component threads to pass the necessary context state.

Alembic provides a substantial performance boost for PGAS code. Elegantly written PGAS implementations of common graph algorithms, such as breadth first search, achieve only 13% of the performance of implementations where the programmer explicitly (and awkwardly) manages both parallelism and locality. Alembic can transform these cleanly written algorithms into high performance locality-aware codes, achieving 82% of the performance of the hand-tuned implementation on average.

In summary, this paper makes the following contributions:

- An analysis to prove co-location of global memory accesses.
- An optimization system that identifies good candidates for code movement.
- An implementation based on LLVM of these techniques.
- An evaluation of the implementation in a PGAS environment on commodity cluster hardware.

2. Background

2.1 PGAS Systems

PGAS languages make the assumption that every piece of global memory is owned by a particular entity which mediates all accesses to that piece of memory. This entity often corresponds to a physical locality domain, such as a node in a cluster. From this domain, any memory it owns can be accessed directly using simple loads and stores. Memory accesses to a different node are mediated by the host node where that memory is located. The distinction between local and remote memory is hidden from the programmer, typically via a *global pointer* abstraction. Having a single node own each piece of memory makes it much simpler to maintain a single consistent view of shared data for programs. The PGAS runtime system ensures that program-level memory ordering is preserved through the various communication mechanisms. The PGAS model has been applied to a variety of system architectures, not just distributed-memory clusters, and as such often use different terminology. In this paper we adopt Chapel's *locale* [9] to refer to a particular set of computational and memory resources.

2.2 Grappa

Grappa is a PGAS-style programming model and runtime system designed for irregular applications. The primary factors that make an application *irregular* are unpredictable data-dependent access patterns and poor spatial and temporal locality. Examples of such applications include graph analytics on social networks, fraud detection, or meta-genomic analysis. To tackle these kinds of applications, the Grappa runtime, implemented as a C++11 library, uses massive parallelism to tolerate the latency of automatically aggregating communication. In Grappa, the programmer is expected to provide the runtime with many (potentially millions) of fine-grained tasks. The runtime then schedules these tasks on the available computational resources, overlapping the remote memory accesses from one thread with the productive execution of other threads. In Grappa, the unit of work being carried out is referred to as a *task.* The particular execution container (stack, context state, etc) that carries out the execution of a task is a *worker thread*, or just *worker*. We will use the term *thread* to refer to the more abstract notion of a sequential thread of execution.

In Grappa, a task is mostly executed by a single worker, but the runtime has also embraced a delegation-based execution model, similar in many ways to the CmPS model (described below), where arbitrary computations on remote data are shipped to where the data is in order to be executed. Delegate operations block the caller until they return their result in order to preserve a sequential thread of execution. In the existing system, delegate operations are specified explicitly and are the only way to access data on other locales. Composing delegate operations and choosing which code should be executed where becomes the dominating concern when writing and optimizing Grappa code, which this work attempts to mitigate.

2.3 Communication-Passing Style

As the name is intended to invoke, Communication-Passing Style (CmPS) [23] is an analog to continuation-passing style for distributed systems. The core idea is that rather than fetching data remotely, communication is done by sending a *continuation,* which contains everything necessary to resume execution, to the locale where the data resides. Transforming execution in this way preserves the same sequential execution expressed in the source program, but now, if there is more than one access to data on the same locale, no additional communication is necessary.

In this execution model, communication is still implicit; however, forcing migration on every access has downsides if a large amount of state must be carried over to continue execution. Therefore, the CmPS work also formalized a notion they call *computation migration,* where most of the state is *frozen* and left behind, and the reduced continuation is sent, does its computation, and immediately returns to rejoin the rest of the state it left behind. CmPS programs explicitly mark when a frozen migration should be done.

CmPS uses a notion of *address spaces* associated with objects to reason about when migration is necessary, which includes ways to recognize when accesses to different objects refer to the same address space. We refer to this as *locality partitioning*.

The CmPS work established formal operational semantics for a functional language with distributed memory, forming the basis by which we reason that our own continuation-passing transformations are sound. Our Alembic transformation essentially applies the CmPS technique to an imperative, object-oriented context – our variant of C++ with PGAS extensions. Additionally, we design analyses to statically choose when to migrate to minimize communication.

3. Language

We start by introducing some concrete syntax and semantics to define the context for the rest of the techniques in this work and establish some common terminology. The aim of our particular implementation of the PGAS model is to stay within the confines of plain C++ as much as possible, both for ease of adoption as well as ease of implementation, so our extensions are confined to *attributes* that express where operations can execute. The syntax and semantics should not be particularly surprising to anyone familiar with PGAS languages, and the techniques we apply should be generalizable to other PGAS environments.

In order to interoperate with the existing Grappa runtime, which is a plain C++11 library, each of the constructs below maps to a C++ class and can be coerced between its "library" and "language" forms. This means that any part of the application can be written without relying on special compiler support, and just the region where the new syntax is used will be manipulated by the passes described in this paper.

3.1 Global Pointers

A fundamental primitive of PGAS-style languages and runtimes is the *global* pointer, which encodes the locale where it is valid in addition to the address in the locale's memory. These pointers, like normal pointers, may refer to data on task stacks, static data, or heap allocations. These pointers, all encoding an address on one particular locale, are expressed using a new `global` modifier on pointer types: `int global* x`.[1] We encode the `global` attribute as a custom *address space*, part of the Embedded C extensions [22], which gets propagated into the compiler's intermediate code. We refer to pointers without any modifier as *local* pointers, signifying that they do not encode a particular locale, but are only guaranteed to be valid where they were generated.

Because global pointers are only valid on one particular locale, a dereference of one implies the chance of communication, since the actual load or store must be executed on the locale indicated by the global pointer. The PGAS language is responsible for ensuring this, typically by turning each global load or store into a `put` or `get` operation supplied by the runtime.

[1] The syntax of pointer modifiers in C/C++ is undeniably confusing. Just as `int const*` indicates that the pointer cannot modify the `int` it points to, so `int global*` indicates that the object it points to may be remote. As with `const`, `global int*` would also be correct, but we prefer the first version.

Global pointers are *deeply* global – pointers computed as offsets from a global pointer, via member accesses or array indexing, are also global. The locale of the resulting pointer, however, is not necessarily the same as the original pointer; it depends on the operation and the type of the object pointed to. These rules will be discussed in more detail in Section 4.1.

Method calls through global pointers are allowed. Because the receiver is now global, any references to the objects' fields must also be associated with the same locale. And local pointers used or returned by the method are only valid where that object resides, so they must also be made global. The details of this transformation will be covered in Section 4.4.

Global pointers can be explicitly constructed from a local pointer and a locale, or may come from allocating out of some global heap which is distributed in some fashion over the locales in the system. In both cases, the pointer must carry the information about how the object it refers to is distributed so that operations on the pointer, such as indexing off of it, can be resolved correctly. PGAS languages often provide a variety of choices for how to distribute arrays, such as Chapel's domain distributions. In Grappa, we have a simple block-cyclic heap with a fixed block size. Objects allocated from the heap must be aligned to the block size so they are not split between multiple locales. Elements of arrays allocated from the heap are distributed round-robin among locales.

3.2 Symmetric Pointers

Globally distributed data structures are an important part of PGAS environments. For instance, it can be useful to have a hash table that tasks on all locales can operate on and see a consistent view. These distributed objects can be implemented in various ways. In Grappa, we implement them using a handle, or *proxy,* to the global object on every locale. Methods called on these proxies from any locale observe the state of one globally distributed object. Internally, the implementation of these methods coordinates among all the other proxy objects and any additional global state to provide this illusion, allowing optimizations, such as buffered updates, to be hidden from the user behind this level of abstraction. These uses are discussed in more detail in a previous publication [18].

In order for our language to handle these objects correctly, we introduce a notion of *symmetric* objects, referred to by symmetric pointers, which have a copy on every locale. Distinct from global pointers, which are valid on one locale only, a symmetric pointer has a valid address on *all* locales. In order to refer to one of these globally-distributed objects, all one needs is a symmetric pointer to its proxies. Methods called through these symmetric pointers go to whichever copy is on the current locale, which then takes care of maintaining the illusion of one distributed object. One additional constraint is that methods called using symmetric pointers must be executed entirely on one copy of the object – if the method is inlined, for example, we must ensure all the references to the symmetric pointer resolve to the same locale. This ensures

that any state maintained internally in each proxy is kept in a consistent state.

Symmetric pointers can be obtained by using a special allocation from the global heap that ensures that all the copies are at the same offset. By obtaining an allocation in this way, the programmer is asserting that their object has symmetric semantics. Variables in the C++ global scope, because of the SPMD nature of the runtime, have the same static offset on every locale, so they may also be treated as distributed objects if they are explicitly annotated as `symmetric`.

3.3 "Anywhere" function annotation

As with unannotated pointers, by default, functions must be assumed to be local, so cannot be moved in a migration. The `anywhere` annotation applied to a function implies that it can safely be run from any locale. This is useful for functions that will take care of inter-locale communication themselves, similar to how symmetric objects work. Furthermore, this annotation is applied to functions whose semantics allow flexibility in where they execute, such as print statements and assertions, or runtime calls such as `spawn`.

3.4 Tasking and synchronization

In this work, we use the tasking and synchronization provided by Grappa unchanged. We introduce some constructs here so that code examples throughout will make sense. As in many parallel frameworks, we express parallelism in the form of *tasks*. A task represents a small amount of sequential work to be run asynchronously some time after it is *spawned.* These short-lived, lightweight parallel threads of execution go by many names, such as *fibers, green threads,* or simply *asyncs.*

Tasks are expressed by passing a C++11 lambda to `spawn`; their initial state is made up of captured variables. In general, tasks may run asynchronously any time after they are spawned and must be explicitly synchronized to ensure they finish. This can be done via ad-hoc synchronization or more structured constructs. For instance, tasks spawned by parallel loops, described below, are typically synchronized using a phaser, which we describe next for reference.

3.4.1 Phased synchronization

A *phaser* [35] is a flexible, reusable global barrier where the number of registered events may be unknown at the start. This is particularly useful for phased rounds of computation where a large amount of parallel work will be recursively spawned, for instance while traversing a graph. Tasks may `enroll` with the phaser before starting and call `complete` when finished, while other tasks can block until the phase is done by calling `wait` on it. Phasers are implemented as symmetric objects in our system, so the same phaser is accessible from all locales.

3.4.2 Parallel loops

Parallel loops (we borrow the name `forall` used in UPC and Chapel [8, 9]) conceptually spawn a separate asynchronous task per iteration. Our parallel loops use a phaser to synchronize all spawned tasks and any additional asynchronous operations that should be completed before the loop is terminated. The non-blocking version, `forall<async>`, can be nested inside other loops and typically uses the phaser of the outermost loop to ensure all iterations complete.

```
struct Counter { long count, winner; };

symmetric Phaser phaser;

void hops(Counter global* A,
          long global* B, size_t N) {
  forall<&phaser>(0, N, [=](long i) {
    Counter global* a = A + B[i];
    long prev = fetch_and_add(&a->count, 1);
    if (prev == 0) a->winner = i;
  });
}

void hops(GlobalAddress<Counter> A,
          GlobalAddress<long> B, size_t N) {
  forall<&phaser>(0, N, [=](long i){
    Locale origin = here();
    phaser.enroll(1);
    delegate<async>(B+i,[=](long& b){
      delegate<async>(A+b,[=](Counter& a){
        long prev = fetch_and_add(&a.count, 1);
        if (prev == 0) a.winner = i;
        phaser.complete(origin, 1);
      });
    });
  });
}
```

Listing 1: *Managing nested delegates and synchronizing them is significantly more tedious and error-prone.* This listing shows code for the HOPS benchmark, a variant on GUPS that tracks which index from B incremented an element of A first. The top version uses the extended syntax and relies on compiler-generated communication; the bottom does explicit movement and synchronization. The first highlighted region (green, dashed border), indicates the first migration, to `B[i]`, the immediately-following region (purple, dotted border) indicates the second hop.

3.5 Example: HOPS

To motivate this work, we use a simple benchmark based on the HPCC random-access benchmark GUPS [19]. In GUPS, an array of random numbers, `B`, is used to index into another array, `A`, and atomically modify the element there. There are more elements in `B` than `A`, so most elements will be visited multiple times. The modified benchmark, which we call *HOPS,* additionally tracks which element from `B` first reaches a given element in `A`. This operation is meant to be representative of work done when visiting objects in irregular applications, and should look familiar to those who know the parent-claiming step of the Graph500 BFS benchmark [16]. In addition, we disregard the distribution of the `B` array when initially placing tasks in order to better demonstrate an opportunity to *hop* directly from one locale to another

when migrating. Two implementations of HOPS are shown above in Listing 1, one using the extended C++ syntax, the other explicit communication, with the two migrated regions highlighted in each.

4. Alembic analysis

Since memory regions are owned by locales, we can think of accesses to that memory as points in the execution that are *anchored* to a particular locale (i.e., a `load` from a global pointer must occur on the locale it points to). These *anchor points* are constraints on the execution of the task, with the start of the task anchored wherever the runtime invokes it. Rather than thinking of remote accesses as necessary communication points, we can instead think of them merely as constraining execution of that part of the task to a particular locale. Many instructions are not anchored, meaning that we could choose to execute them at either location.

Tasks can be thought of as being divided into regions based on locality. At each transition between regions, a continuation is constructed and sent to where the next region is to be executed. These migrations may either be *blocking,* in which case control returns to the home locale immediately after, or *chained,* hopping from one locale directly to the next. Though executed on different locales, these regions still represent a single sequential task.

Considering task execution in this way enables many useful optimizations. Regions that include more than one anchor point can save on round-trips. Values produced and consumed on the same locale need not be communicated. Finally, when a continuation constitutes the remainder of the task, the migration can be *asynchronous,* immediately freeing up the worker executing it. We do not consider opportunities to further parallelize tasks, counting on the programmer to express the concurrency they desire with explicit task spawns.

The goal of our analysis is to choose how to divide tasks into locality regions and transform them into a series of continuation-passing migrations that minimize communication cost. Our analysis operates at the level of standard compiler optimizations, specifically, on LLVM's intermediate representation (IR) [27]. First, *locality partitioning* divides anchor points into sets proven to be on the same locale. Next, *region selection* enumerates and evaluates possible regions. Finally, a transform pass extracts the regions, computes continuations, and inserts runtime calls to do the migration. The following sections describe the steps in more detail.

4.1 Locality Partitioning Algorithm

Anchors are instructions that access memory, restricting them to execute where that memory is. For most anchor points, the region of memory is demarcated by a pointer and size. While the precise locale of the pointer will almost never be known statically, it is often possible to prove that two anchor points' locales are the same. The goal of *locality partitioning* is to find as many of these co-located anchors as possible.

Expression	Locality	Operation
Local pointer:		
`p`	`here()`	identity
`&p[4]`	`locale(p)`	array index
`&p->f`	`locale(p)`	field offset
`p->adj()`	`locale(p)`	local pointer
`p->adj()+9`	`locale(p)`	local pointer index
`new T[4]`	`here()`	allocation
Global pointer:		
`g`	`locale(g)`	identity
`&g[4]`	`unknown`	array index
`&g->f`	`locale(g)`	field offset
`g->adj()`	`locale(g)`	local pointer
`g->adj()+9`	`locale(g)`	local pointer index
`make_global(p,3)`	`3`	constructor
`global_alloc<T>(4)`	`unknown`	allocation

Table 1: Locality of various pointer operations. In these examples, assume `T` is aligned to the block size, and the method `adj()` returns a local pointer.

We take an approach similar to *value partitioning* [1, 5] to divide anchor points into different locality sets. Value partitioning is a variant of *value numbering* which tries to divide value-producing instructions into *congruence classes* (or *sets*) for the purpose of eliminating redundant computations. Congruence is a recursive property, so in order for two values to be congruent, their respective operands must be in the same congruence sets. Value partitioning can be approached either from an *optimistic* perspective, where values are considered congruent until proven incongruent, or *pessimistic,* where values begin in their own sets and are merged when proven congruent. Both approaches are conservative.

Locality partitioning differs primarily in the definition of *congruence.* Rather than finding when operations compute the same value, we are concerned with finding when pointer values are guaranteed to be on the same locale. Table 1 shows a number of operations on pointers and the information available about their relative locality. For example, field offsets in block-size aligned objects are guaranteed to be on the same locale as the global pointer.

The locality rules supported by our C++ PGAS language use only local reasoning. One could imagine extending this in languages with more rich global locality information to prove co-locality in more situations. For example in Chapel [9], *domain distribution* information could be used to prove that elements with the same index in arrays with the same distribution have the same locale. To support such features, additional locality rules would simply need to be added.

The list of locality rules need not be exhaustive – any operation not covered will be conservatively placed in a new locality set. Some instructions may have no information available about the region of memory they access, such as an

opaque function call. These must remain on the home locale to ensure that they are executed in the context they expect; the programming model ensures that they handle any necessary communication themselves.

The current implementation takes a pessimistic partitioning approach, initially placing all anchor points in distinct locality sets and merging sets when it proves they are on the same locale. This does limit our analysis in the same way as for value partitioning: we must rely on visiting anchor points in a topological order, and therefore cannot use loop-carried information to prove co-locality. A future implementation could use the optimistic value partitioning approach if this was shown to be too limiting.

4.2 Region Selection

Once the anchor points have been classified, the next task is to choose where to execute the remaining unconstrained instructions. The goal is to come up with a sequence of migrations, constrained by anchor points, that will result in the minimum amount of communication. Recall from Section 2.3 that the continuation must include everything needed to resume execution; the communication cost is the size of this continuation. In some situations it is preferable to leave some state on the original task's stack, migrate a smaller continuation, and return to pick up the rest (in CmPS this was a *freeze* operation).

This analysis divides the instructions in each *task* into *regions* by locality. All the anchors in a region are proven to be to the same locale. Non-anchor instructions, including symmetric pointer accesses and `anywhere` function calls, are placed in one region or another to minimize communication. Though it would likely lead to some improvement in communication, to simplify the problem, this pass does not consider duplicating instructions in more than one region, nor splitting the thread to expose additional parallelism. This means that regions do not overlap; the task is still a single thread of execution whose control and data jumps around the system. A more ambitious transformation which does allow for these is left for future work.

Before diving into the details of the algorithm, let us revisit the HOPS code in Listing 1, which will be used throughout this text to explain the mechanics of Alembic. A parallel loop from 0–N creates tasks for each iteration. Each task gets the random value stored at `B[i]`, a global access, and uses that to index into `A`, likely referring to another locale, on which it performs an atomic increment. The LLVM IR corresponding to this task, on which our analyses operate, is shown in Figure 1a. Anchors are annotated with their locality sets and two distinct migrated regions are shown highlighted. The un-highlighted instructions in these regions must be hoisted to make the regions contiguous.

Anchor points are annotated with their locality set in blue. The locality regions we would like to infer are highlighted: the first migrated region, after the horizontal rule, to be executed at `B[i]`, and purple for the region at the element in `A`. However, instructions highlighted in red, which are anchored where the task started, currently prevent these regions from being contiguous.

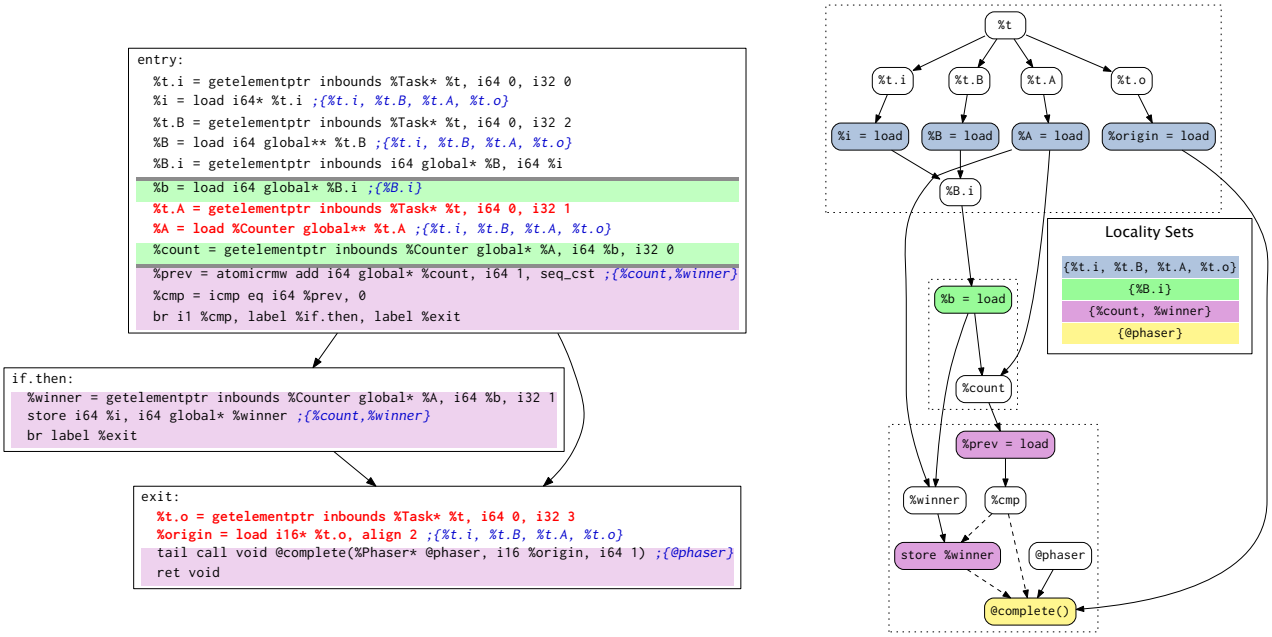
Choosing the optimal migration policy is intractable: it would at the very least require full-program analysis, but would also depend on the layout of data, runtime load balancing, physical interconnect topology, and many other concerns. The hypothesis of this work is that automated decisions at the scope of a task, with the constraints provided by anchor points, are sufficient to compete with communication explicitly provided by the programmer. Even with the above restriction that instructions only appear once, instructions can still be reordered and because we do not know the optimum number of migrations, the problem reduces to finding a minimum k-cut (where k is the number of migrations) on the task's dependence graph, which is known to be NP-complete [15].

Instead, we implement a much simpler greedy algorithm that evaluates a restricted set of candidate regions with a simple cost heuristic. Rather than evaluating all possible reorderings, we determine independent regions for each anchor (migrating back home after each), reorder anchors only with the home region, and attempt to combine adjacent regions pairwise in a greedy fashion. Steps of the algorithm will be explained in the coming sections, but at a high level, it works as follows:

1. For each anchor, expand a region, starting from the anchor, to its maximum allowed extent.
   (a) When encountering other anchors, determine if they: (i) share the same *locality set,* and can be included in the region, (ii) have a *symmetric* locality and can be included, (iii) can be *hoisted* to the home region, or (iv) represent a necessary end to the region.
   (b) At each step, find the inputs and outputs to the region and compute the *cost heuristic* (described below) for the current region, as if any *hoistable* instructions were moved before the region.
   (c) Keep track of the best sub-region.
2. Skip anchors that have already been completely subsumed within another anchor's best region.
3. For each pair of adjacent regions (those whose maximum extents overlap or are adjacent):
   (a) Compute the continuations needed to migrate directly between the two.
   (b) If the cost combined is less than the cost of the two separate migrations, then replace the two separate regions with a new *chained* region containing both.
4. Mark regions whose exits are the end of the task as *async.*

4.2.1 Expanding the region

To find regions of code that can be executed at the location of a given anchor, we start from the anchor instruction and



(a) HOPS Iteration task in LLVM IR.

(b) Dependence graph for HOPS iteration.

Figure 1: Breakdown of the task executing a single iteration of the HOPS loop. In (a) we show the task's instructions, annotated with their locality set (in braces), and divided into regions. At the first horizontal line, the task migrates to `.iB.i` (in green). At the second line, execution migrates again for the atomic increment until the end of the task (purple). Bold, non-highlighted instructions are those that must be hoisted into the first (home) region. The corresponding value dependence graph is shown in (b) with nodes for instructions labeled with the value they produce. Here boxes are drawn around regions – arrows that cross these boundaries indicate values that will go into continuations.

iteratively expand the region to include instructions that are valid to run on that locale. Any instructions proven to not touch memory are trivially allowed. Thanks to the previous analysis, any memory-access instructions, including calls to functions that may access memory, will be associated with a locality set. Any instructions in the same locality set as the current anchor are allowed. Symmetric pointers, explained in Section 3.2, are valid on any locale, so any symmetric anchors can also be included. For other accesses, we will attempt to hoist or localize them, which will be explained next. After determining that an instruction is valid, the cost function, explained below, is computed for the current region, and the minimum cost region is tracked.

Though regions may have multiple exits, in order for the continuation-passing transformation to work, they must have only a single entrance. To ensure this, basic blocks reached by the expanding region are visited in reverse postorder, and basic blocks with incoming edges not already in the region are disallowed and become exit points. This over-conservatively disallows loops from being subsumed within a region, which is a potential pitfall that could be remedied with further engineering.

4.2.2 Cost heuristic

The cost function attempts to encode the combination of communication and execution costs inherent in migrating the given region. In the coarsest view of the runtime system, the total amount of data moved is worth minimizing, but the execution overhead – time spent aggregating, sending, deaggregating, blocking and waking threads – is roughly *per application-level message*. These are aggregated into larger messages by the runtime, but overhead is associated with each independent task that issues a remote request. Therefore, our cost function has to take into account number of messages in addition to the amount of data moved.

Inputs and outputs to each region are computed from LLVM IR, which is in static single assignment (SSA) form by design, and used to compute the size of the continuation, or the total amount of data that needs to be moved, in each migration. This can be viewed as partitioning the program dependence graph, similar to how it is done in decoupled software pipelining [31], but attempting to minimize data crossing the partitions rather than exposing parallelism. Figure 1b shows the dependence graph for HOPS, with a node for each instruction labeled by the value it produces, and

arrows showing uses of those values. Arrows that cross a region's bounding box represent values that must go into a continuation. Grappa's communication mechanisms currently only support POD types, allowing Alembic to statically determine the precise amount of data to be moved. More dynamic object-oriented features, such as sub-type polymorphism or serialization of arbitrary additional data, would make this cost estimate more difficult.

Grouping two anchors with the same locale into one region eliminates a round-trip message. This is modeled in the cost heuristic by subtracting the cost of those messages for each anchor. If all exits from a region return `void`, this means it is the final region in the task and the return trip to get back to the task's home locale is unnecessary, saving an additional message, which we also model. The resulting heuristic equation is:

$$\begin{aligned} \mathit{cost} = \ & \texttt{sizeof}(\mathit{inputs} + \mathit{outputs}) \\ & - 2 * \mathit{messageCost} * \mathit{numAnchorsIncluded} \\ & - (\mathit{messageCost}, \texttt{if}\ \mathit{allExitsVoid}) \end{aligned}$$

In Section 5.3, we evaluate this tradeoff empirically to come up with a reasonable setting for *messageCost* for our experimental platform.

4.2.3 Hoisting anchors

We saw earlier, in Figure 1a, that sometimes the order in which memory accesses are scheduled is not ideal for migrating because the instruction scheduler is assuming a different cost model for memory accesses than what we have in mind. For example, the load of `%origin` in the exit block prevents what would otherwise be an asynchronous migration. It is a clear win in this case to hoist the load before both regions because it only costs the data movement of 2 additional bytes but saves in total messages sent by allowing an asynchronous migration. In the general case, one would need to explore every allowable reordering of anchor points to find the one that minimizes messages and continuation size. In our simplified search, we only attempt to move instructions into the first region (at the home locale), which is a clear case where reordering will be beneficial. Anytime an anchor with a different locality set is reached, we check whether it can be placed in the first region without violating dependences or locality. We do not consider opportunities to move the access into other regions, as it would greatly increase the complexity and search space, and we found it typically did not pay off in the situations we encountered.

We use LLVM's memory dependence analyses to determine if the memory operation clobbers or is clobbered by any instructions in the region or violates synchronization ordering. Additionally, to be hoisted, stores must not be conditional (must dominate all exits from the region). Typically this move has already been done by previous passes if it is possible. Finally, we must determine if, recursively, all of the operands that reside in the region can be hoisted.

If all of these criteria are met, then the operation can be marked as *hoistable.* When computing migration cost, candidate regions treat them as if they had been moved, but they are not actually moved unless the minimum-cost selection includes it. Hoisting instructions is done independent of prior region selections. There is a slight chance that this hoisting could have made prior migrations happen if they had known, but this is a performance, not a correctness, issue.

In the running example, hoisting both of the loads in HOPS leaves us with two contiguous regions back-to-back, allowing us to migrate directly between them. The phaser is symmetric, so calling `complete` on it can be done anywhere, so the migration can be asynchronous.

4.2.4 Localizing allocas

Rather than hoisting loads and stores before the region, in some cases it can be possible to instead change what memory they are referring to. In particular, temporary objects are typically allocated on the stack on entry to the function and used later. If we can prove that a piece of stack-allocated memory is only used inside a single region, then we can *localize* that temporary storage and put it in the migrated region, so that the loads and stores using it can be done locally after migration. To determine if this is the case, we examine all accesses to the region of memory specified by an `alloca` instruction, including double-checking with the alias analysis to ensure nothing else may be using that memory. If all of the accesses are resolvable, and they all occur in one region, then we can move the `alloca` inside the region.

This check can only be done after the region has been expanded to its maximum extent (see below). Our analysis speculatively allows the region to include accesses to stack-allocated memory, expands as far as possible, then does this check. If any allocated regions are not localizable, we mark them and redo the region expansion, this time not including those accesses. This may iterate more times, but each iteration will remove at least one speculatively-included anchor, so it will terminate quickly.

4.2.5 Chaining regions

After finding the maximum extents of all single-locale regions, we start evaluating how to stitch these regions together to form a single migrating thread of execution. We could simply migrate back to the home locale of the task after each region, and we would still benefit from moving multiple anchors on a single locale. However, if two regions to different locales are adjacent, additional benefit could come from hopping directly between the two. Migrating directly saves costly messages and wake-ups but may increase the size of the continuation.

To evaluate whether continuing directly to the next region will be beneficial, we use the same cost heuristic. We compute the continuation needed to execute the combined region, the continuation from the first region to the second, and the outputs of the combined region. If this combined cost is less

than the sum of the individual region costs, which amounts to whether the continuation between the two regions is smaller than the cost of an additional message, then the two regions are chained. As we continue to consider adjacent pairs, longer chains of linked regions may be generated, resulting in a task that will seem to hop around between locales, following where its data is.

4.3 Transforming tasks

This section will explain at a high level how the original task is transformed according to the choices made by the analyses above, at the level of LLVM IR. Migration is done by extracting all of the instructions in the region into a separate function, sending the continuation in a message to the remote locale, which, on receipt, invokes the extracted function, and the output values from the region, needed for the next continuation, are collected. If the next region is to be executed back on the original task, these outputs are sent back to rejoin the rest of the stack. Otherwise, if the next hop is directly to another locale, then the continuation is constructed, and another migration is done.

All of the data movement is handled by a generic `migrate` call in the runtime. This function takes as input the destination locale, the function to run, a struct for the continuation, and a pointer to a struct for storing the outputs. The calling task blocks until the sequence of migrations returns to rejoin the stack. In the case where a migrated region includes the end of the task, the variant `migrate_async` is used, which immediately frees the worker to start another task while the migrated continuation finishes the previous task's execution remotely.

Extraction is done using a modified version of the LLVM `CodeExtractor` utility. All the basic blocks of the region to be extracted are cloned into a new function. All of the exits are redirected to a single `return` block which returns the output of a `phi` to differentiate which exit was taken. At the call site, this return value is used in a `switch` to jump to the correct exit. Before the call, the continuation is constructed on the task's stack, and after the call the outputs are loaded from the other struct passed to `migrate`.

Figure 2 shows how the HOPS code ends up being transformed. The initial task constructs a continuation with the values needed for both migrations, and computes the destination locale. Inside each migrated region, we load the inputs from the continuation. Finally, in each region, we extract and use the local pointer from global pointers which are now local. Because the two migrations make up the rest of the task, `migrate_async` can be used, which allows the initial task to return immediately, though the enclosing parallel loop waits for the final migrated region to signal `complete`.

4.4 Globalizing functions

As mentioned back in Section 3.1, methods can be called on objects via global pointers. However, this is not expressible in C++. We allow the C++ frontend to generate these method calls anyway and fix them ourselves.

To handle method calls on global pointers correctly, they must be made parametric on the pointer type of the receiver. This means constructing a new version of the method where the receiver is a global pointer instead. To do this, we clone the function, then propagate the changed pointer type through all the instructions, which may cause other pointer values to become global. Any local pointers referenced inside the method are wrapped up in a new global pointer with the locale of the receiver pointer, including the return value if the method returns a local pointer. Finally, we replace all calls where the receiver was cast from a global pointer with a call to the new *globalized* version.

In fact, because methods are really just functions, we apply this same transformation on any functions that accept a local pointer but are passed a global pointer instead.

4.5 Put/get generation

For comparison, we also implement a version of our compiler that generates just `put` and `get` operations. This is a fairly standard baseline for PGAS languages without any optimizations enabled. Each global memory access is replaced with a call to a corresponding remote operation in the API. After fixing up function calls with global pointer parameters (as described in Section 4.4), all of the global memory accesses are clearly delineated in the LLVM IR. We then simply find all instances of `load`, `store`, `cmpxchg`, and `atomicrmw` which have a global pointer operand and replace them with calls to the underlying PGAS library (in our case, `grappa_get`, `grappa_put`, `grappa_compare_and_swap`, etc). To maintain the sequential semantics implied by the original memory operations, these operations all block the calling task.

Beyond the generic memory access optimizations applied by LLVM, our compiler generates fairly naïve puts and gets compared to optimized communication generated by other PGAS systems (see Section 6.3). However, the Grappa runtime dynamically aggregates messages from multiple tasks and tolerates remote access latency using massive multithreading which give much of the performance benefit of those other techniques but with some runtime cost.

5. Evaluation

Our goal in this evaluation is to quantify the extent to which these static migration analyses and transformations are able to match the performance of hand-tuned locality optimizations. First, we evaluate the performance of Alembic on 4 irregular application kernels. Then we probe more deeply into the effect of each optimization using the HOPS case study. Finally, we explore the tradeoff between asynchronous and blocking migrations in order to empirically choose a value for *messageCost.*

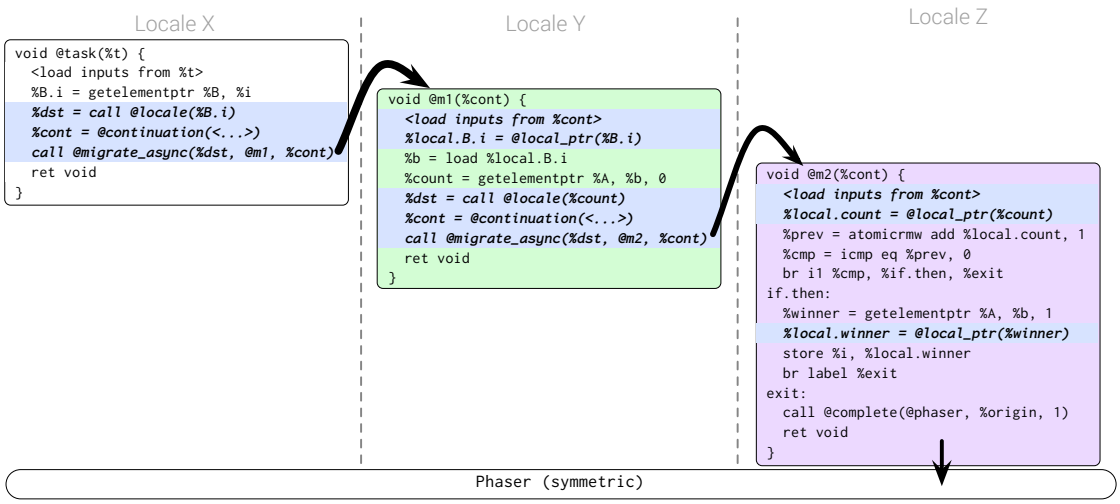


Figure 2: Alembic transformation of the HOPS task doing multi-hop migration (much-simplified LLVM IR with types elided). Code added to do the transformation (bold and highlighted blue) includes: for each migration, construct a continuation and find the destination locale, and in each migrated region, extract local pointers from global pointers.

5.1 Application performance

The purpose of Alembic is to be able to automatically generate task migrations that are onerous to do by hand. We evaluate the analyses on 4 representative irregular application kernels which were implemented and optimized in previous work evaluating the Grappa runtime [29]. The existing implementations have explicit delegate calls to do communication and move parts of the computation to different locales. These delegates calls were tuned by hand to get the best performance out of the Grappa system, including changes to make them asynchronous, reduce the number of messages, and minimize data transferred.

In each application, we ported the most performance-critical sections, removing all explicit communication and instead using the C++ extensions described in Section 3 (e.g., `global*`). These sections now rely on Alembic to automatically generate communication for them. In the following sections we will briefly describe the applications, the sections ported, and the regions identified by Alembic.

BFS Breadth-first-search is a common kernel used to evaluate irregular application scaling, and is the primary benchmark for the Graph500 rankings [16]. The benchmark does a search starting from a random vertex in a synthetic graph and constructs a tree out of parent vertices for each vertex traversed. We port the entire timed region; a snippet which does a single level of the traversal is shown in Listing 2.

Alembic determines that the atomic compare-and-swap and everything after it can be in an asynchronous migration. This includes pushing the vertex onto the next frontier, which can be moved because `GlobalQueue` is symmetric and safely handles push operations from any locale.

```
symmetric GlobalQueue frontier, next;

void bfs_level(Graph symmetric* g) {
  Vertex global* vs = g->vertices();
  while ( !frontier.empty() ) {
    VertexID i = frontier.pop();
    forall<async,&phaser>(adj(g,vs+i),
                          [=](VertexID j){
      if (cmp_swap(&vs[j]->parent, -1, i))
        next.push(j);
    });
  }
  phaser.wait();
}
```

Listing 2: Code from BFS which does a single level of the traversal. Alembic identifies and transforms the highlighted region into an asynchronous migration.

Connected Components Another core graph analysis kernel is Connected Components (CC). We implement the three-phase CC algorithm [4] designed for the massively-parallel MTA-2 machine. The first phase does multiple recursive traversals in parallel, each labeling vertices with a color. Whenever two traversals encounter each other, an edge between the two colors is inserted in a global set. The second phase performs the classical Shiloach-Vishkin parallel algorithm [34] on the reduced graph formed by the edge set from the first phase, and the final phase propagates the component labels back out to the graph.

We port the first phase, which does the traversals and insertion into the hash set and takes the majority of execution time; a snippet is shown in Listing 3. Most of the iteration is able to be subsumed in a single asynchronous migration because the stack-allocated lambda which is passed to `spawn`

```
GlobalHashSet symmetric* set;
Graph symmetric* g;

void explore(VertexID r, color_t color) {
  Vertex global* vs = g->vertices();
  phaser.enroll(vs[r].nadj)
  forall<async>(adj(g,vs+r), [=](VertexID j){
    auto& v = vs[j];
    if (cmp_swap(&v.color, -1, color)){
      spawn([=]{ explore(j, color); });
    } else if (v.color != color) {
      Edge edge(color, v.color);
      set->insert(edge);
      phaser.complete(1);
    }
  });
  phaser.complete(1);
}
```

Listing 3: The first phase of Connected Components where we assign colors and insert an edge into the set whenever two traversals conflict. Alembic detects 2 migrations, highlighted above. The second region is only able to be asynchronous because the alloca for `spawn` could be localized.

```
void spmv(Graph symmetric* g, double global* X,
          double global* Y) {
  forall(g, [vx,vy](VertexID i, Vertex& v) {
    forall<async>(adj(g,v), [=,&v]
          (int64_t localj, VertexID j){
      Y[i] += X[j] * v->weights[localj];
    });
  });
}
```

Listing 4: Ported code from Pagerank. The index into weights is local, so just two chained migrations are needed to visit the element in `X` and then update the element in `Y`.

is able to be localized, the set is symmetric, and `spawn` and `complete` are annotated with *anywhere*.

Pagerank This kernel is a common centrality metric for graphs which iteratively computes the weighted sum of neighbors until convergence. The computation essentially amounts to a sparse matrix dense vector multiply for each iteration, which in our implementation is parallelized over vertices in the graph as well as over the adjacencies for each vertex. We report performance as throughput, comparable to Graph500's TEPS measure, computed as the number of edges in the graph over the average time per iteration.

We port just this multiply section, shown in Listing 4, which makes up nearly all of the communication and execution time. This kernel is able to benefit from doing two continuation-passing migrations back-to-back to go from the original spawned task which is executed at the source vertex where the edge weight is, to the corresponding element in the source vector, and finally to the element in the resulting vector. That multi-hop migration can all be done with asynchronous migrations, eliminating the need for any blocking calls (except of course the main task which blocks on the phaser used to synchronize all this work).

IntSort This benchmark comes from the NAS Parallel Benchmark Suite [2, 30]. The second-largest problem size, class D, ranks 0.5 billion random integers sampled from a gaussian distribution using a bucket sort algorithm. The performance metrics for NAS Parallel Benchmarks, including IntSort, are "millions of operations per second" (MOPS). For IntSort, this "operation" is ranking a single key, so it is roughly comparable to our TEPS measure.

We port the phase which scatters elements into buckets. This is done by essentially just appending individual elements to pre-allocated buckets, which involves a remote fetch-and-increment and a store. The entire remote end of the scatter is able to be done with an asynchronous migration.

5.1.1 Performance comparisons

To evaluate the impact automatic migration has on application performance, we performed experiments comparing compiler-generated communication against manually-optimized explicit delegate calls. As explained earlier, the manual implementations were optimized in previous work evaluating the Grappa runtime itself, so though they may not be the best possible implementation, they are the best known so far. For each application, we compare 2 variants of compiler-generated communication: individual puts and gets, as described in Section 4.5, and Alembic with the *messageCost* which will be chosen empirically in Section 5.3.

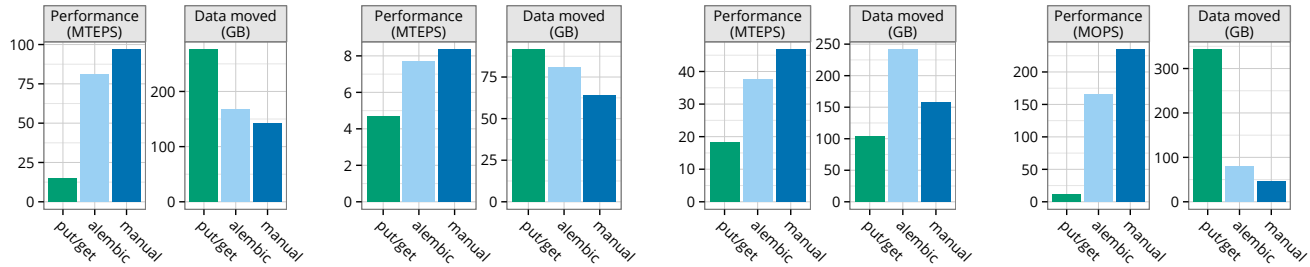
Experiments were run on a small cluster with 12 nodes, each with two 6-core Intel Westmere 2.66 GHz Xeon processors with hyperthreading disabled, 24 GB of memory, and 40 Gbit Mellanox ConnectX-2 InfiniBand interconnect. For these experiments we run 8 Grappa processes per node as this gives more reliable performance. The results in Figure 3 show the performance of each application as a throughput measurement (bigger is better). Also plotted is the total number of bytes transferred during execution, which is dominated by application data such as puts and gets or continuations.

It is clear that the naïve put/get model is insufficient, despite the the runtime's efforts to mask latency. On average, manual delegates performed 7.6x better than put/get. This vast performance pitfall is due to the much larger number of round-trip messages that must be sent, and is echoed in the larger total amount of data moved. By looking at the static migration metrics in Table 2, we can get a sense for how many messages are saved. For instance, IntSort performs 5 remote accesses per scatter operation, which can be done manually with a single async delegate, a ratio of 10 messages to 1, so the performance difference should be drastic.

One the other hand, for Pagerank the discrepancy is smaller. The three remote accesses are transformed into two chained migrations, but the second one is back at the source locale, so the put/get implementation, which only does one get, moves less data than the transformed version. However,

Application	Anchors (global)	Migrations (blocking)	Migrations (async)	Hoisted accesses	Allocas localized	Symmetric accesses	"Anywhere" calls
HOPS	3	0	2	2	0	1	0
BFS	4	2	1	1	0	5	0
CC	5	2	1	4	1	13	3
Pagerank	3	0	2	2	0	0	0
IntSort	5	0	1	0	0	0	0

Table 2: Static metrics: frequency of each optimization in each benchmark. Counts are for unique source-code instances, so more than one inlining location does not count in these metrics. Only the ported part is counted. HOPS and Pagerank's two asynchronous migrations are each chained.



(a) BFS (scale-23 graph) (b) CC (scale-23 graph) (c) Pagerank (scale-23 graph) (d) IntSort (Class D, 0.5M keys)

Figure 3: Application kernel performance: comparing manually-optimized movement against compiler-generated migration. Experiments were done on 12 nodes with 8 cores per node. Overall, Alembic performs competitively with manually-optimized communication, and significantly better than naïve puts and gets. This performance is due in part to reduced data movement, which is also shown. The outlier, Pagerank, is explained in Section 5.1.1.

the additional scheduling overhead of waking the blocked task is such that the asynchronous version is still faster.

Alembic-generated migrations perform favorably with manual delegates, on average achieving 82% of their performance. The cause of this shortfall is visible in the total data moved metric – Alembic moves more data in each of the applications. Rather than doing template specialization and inlining as the C++ code does, Alembic currently uses a C-style interface for `migrate` which requires an additional function pointer and phaser pointer in each message, which, for messages on the order of 16-32 bytes, is significant.

Another situation where Alembic-generated continuations are larger than necessary is when it includes values which could be re-computed. One example is in IntSort, where rather than computing two field offsets from the base pointer, it includes both pointers in the continuation.

These shortcomings can of course be remedied with some engineering effort. A technique analogous to C++ template specialization could be used by the code extraction pass to make optimized versions of `migrate`, eliminating the need for the extra arguments and allowing opportunities to pass arguments through registers. Common register allocation techniques could be applied to determine when to *rematerialize* [6] values to save space.

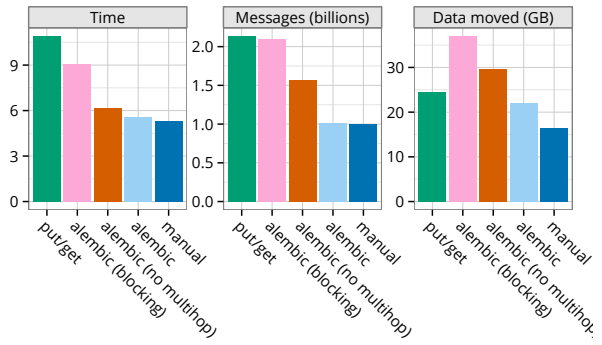


Figure 4: Performance of HOPS using manual communication, naïve puts and gets, or Alembic migration, with various features disabled. We can see that only with all features enabled does Alembic produce the same number of messages as the manual version.

5.2 HOPS Case Study

Our goal with this study is to explore how various optimizations implemented by Alembic affect its performance. Put/get does 3 blocking remote accesses, while the manually-

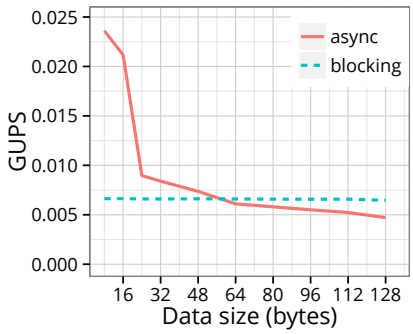


Figure 5: Exploring the tradeoff between asynchronous and blocking migrations. The async version must carry additional data with it, while blocking can leave its data behind. Past 64 bytes, blocking wins out, but performance degrades slowly.

optimized version and the Alembic version both do two chained asynchronous migrations. Figure 4 shows three metrics: execution time, total number of messages, and total data movement. We can see that blocking migrations end up moving more data even than put/get. This is because of the additional function pointer and phaser pointer explained earlier, which results in the greater amount of data for Alembic compared with manual. The message count metric matches our expectations closely: both blocking versions have the same number of messages, the next bar is allowed to do an async migration, saving a return trip, and the full Alembic additionally avoids yet another message by hopping directly between two locales. In the end, Alembic achieves 95% of hand-tuned performance for HOPS.

5.3 Measuring message cost

The heuristic which drives region selection, described in detail in Section 4.2.2, relies on having an estimate of the relative cost of each message. To get a rough idea of what a good setting for this message cost may be, we construct another variant of GUPS. The goal is to measure the tradeoff between making larger continuations, requiring larger messages for each migration, compared to the benefits of async migrations. For this experiment, GUPS is modified to do additional work after the increment to `A[B[i]]` – it copies an array of randomly-generated values into a static variable on the locale.

For the first experimental condition, we manually do an asynchronous migration containing the GUPS increment and the array computation, so the statically-sized data array must be included in the continuation, and synchronization is done via the default phaser. Alternatively, the second condition leaves the data array on the original stack, does a blocking migration to do the increment, and returns to do the array computation on the original locale. We then vary the size of the data array and measure the performance.

The results, shown in Figure 5, show that *blocking* performance is flat, because the communication, which dominates execution time, is constant. The asynchronous migration, however, varies greatly as the continuation's size changes. For smaller amounts of data, avoiding the return message and task wakeup is a clear win (3.5x better than blocking). As continuation size increases, there is an initial drastic drop in performance. This is due to some logic in Grappa's communication layer that optimizes for fitting messages plus some additional metadata in a single cacheline. After that initial drop, however, performance continues to degrade as more memory and network bandwidth is consumed. In these experiments on GUPS, the advantage shifts to the blocking version around 64 bytes of additional data.

Because of this slow degradation, it is safe to err on the larger side when choosing what to set *messageCost* to. In our case, we have chosen to set *messageCost* to 80, which is large enough that in our applications, whenever it is possible to migrate asynchronously, the compiler chooses to do so.

6. Related Work

Program partitioning to reduce communication has been explored in a variety of systems previously. These can broadly be separated into solutions related to moving computation closer to data, offloading computation to a more capable locale, and other communication optimization techniques.

6.1 Computation migration

Early DSM systems Computation migration was employed in multiple early DSM systems, most notably MCRL [20, 21] and Olden [7, 32], to improve performance for unpredictable access patterns. MCRL, and prior simulation work in the Prelude language, generated a continuation and appropriate messages to perform a lightweight migration, but only at user-annotated procedure calls. On the other hand, Olden performed a relatively heavyweight thread migration (registers and top stack frame) at every remote memory access. Both used heuristics similar to Alembic's to predict when to migrate – Olden used static analysis and annotations to determine how many accesses are co-located, and MCRL used the dynamic read and write load to determine how to balance work. Alembic's migrations are both lightweight like MCRL's, and may happen anywhere in a program, as in Olden. Alembic's aggressive instruction reordering and alloca localizing further improve the effectiveness of computation migration.

Traveling threads The traveling thread execution model [28] is another notable instance of moving execution context to data. In this execution model, threads are split up into much smaller *threadlets,* consisting of just a few instructions, which represent a migration to execute that part of the code closer to the memory it accesses. This work is part of a larger effort to overcome the von Neumann bottleneck by leveraging processing-in-memory (PIM) technology [26]. Aimed at of-

floading small snippets of execution to the memory system, their notion of locale is extremely fine-grained, at the level of banks of physical memory. Some of the analyses they describe use a similar minimum-cut optimization strategy over the dataflow graph to determine where to split threadlets. Our work could be seen as implementing a form of traveling thread architecture in software on commodity clusters.

Charm++ & ParalleX Charm++ [25] and ParalleX [24] are event-driven distributed-memory programming models based on sending messages between dynamically movable objects. These models allow for a form of computation migration via fine-grained asynchronous active messages. While these models provide opportunities for latency tolerance and scalability, reasoning about sequential control flow can be difficult. Alembic comes from the opposite direction, taking sequential tasks and turning them into asynchronous messages.

6.2 Computation offload

Automatic program partitioning has also been explored in the domain of mobile application offloading, where the goal is to reduce the load on resource-constrained clients. Wang and Li [37] partition statically based on a cost heuristic, but rather than using a fixed cost, specialize for multiple cost ranges and select among them at runtime. Other work in dynamic object-oriented languages [36, 38] has modeled communication patterns with object relation graphs, assigning costs according to a target platform and doing min-cut analyses to partition computation and place objects. In the interest of keeping sensive data on the server, Chong et al. [13] used a similar notion of "anchoring" computation and optimizing communication based on those constraints, in this case for security.

6.3 Communication optimization

UPC Unified Parallel C (UPC) [8] is a PGAS language with a number of compiler optimizations to make communication more efficient for the runtime. In addition to common optimizations such as redundancy elimination, the UPC compiler coalesces puts and gets [12] and tolerates latency by automatically making some remote memory operations asynchronous [11]. The latter optimization involves aggressive reordering of memory accesses and coordination of data dependences. Expressing global accesses as C++ pointer dereferences allows us to leverage built-in optimizations, such as simple redundancy elimination, but we do not do static coalescing. The Grappa runtime dynamically aggregates requests and uses programmer-specified parallel tasks to tolerate latency. These techniques, while improving performance by making communication more efficient, do not significantly affect total data movement as migration has the potential to.

FortranD An early PGAS-like programming language, FortranD [17], used layout information to partition straight-line programs to place computation where its data is. Like modern PGAS languages, FortranD has ways to express at a high level how data is distributed across locales, which it uses to determine where to run iterations of loops and generate communication, using what they call the *owner computes* rule. For these techniques to be effective, they need global knowledge of layout, which is not always possible, especially for workloads where layout is dependent on the data.

Chapel & X10 Chapel [9] and X10 [10], two PGAS languages in active development, employ a mix of techniques leveraging high-level information about data layout to optimize communication, such as coalescing communication into bulk operations and spawning tasks with their data [3, 33]. These languages also support explicitly running blocks of code on other locales (via `on` or `at` statements) which operate the same as Grappa's delegates. To the best of our knowledge, that work has not included automatically splitting up tasks and migrating them to improve locality, which is important when there is no "good" initial task placement, and allows the code to remain readable – free of combersome nested migration blocks.

7. Conclusion

Alembic automatically extracts locality in PGAS programs by migrating execution, which achieves better performance and a reduction in messages and total data movement over other optimization techniques which only move data. The technique, which finds co-located memory accesses and chooses migrations that minimize communication, should be generally applicable to other PGAS environments. On the set of irregular application kernels evaluated in this work, Alembic provides performance close to that of hand-optimized migration – on average within 18%, and is 5.8× faster than naïvely generated communication.

Acknowledgements

We thank Michael Ferguson for his early input on the work and the idea of expressing global memory accesses using LLVM address spaces, and for providing his implementation generating LLVM IR for Chapel. We also thank the reviewers for their detailed feedback and useful suggestions for improvement.

References

[1] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 1–11. ACM, 1988.

[2] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks. *International Journal of High Performance Computing Applications*, 5:63–73, 1991.

[3] R. Barik, J. Zhao, D. Grove, I. Peshansky, Z. Budimlic, and V. Sarkar. Communication optimizations for distributed-

memory X10 programs. In *Parallel Distributed Processing Symposium (IPDPS)*, pages 1101–1113, May 2011.

[4] J. W. Berry, B. Hendrickson, S. Kahan, and P. Konecny. Software and algorithms for graph queries on multithreaded architectures. In *Parallel and Distributed Processing Symposium. IPDPS 2007. IEEE International*, pages 1–14. IEEE, 2007.

[5] P. Briggs, K. D. Cooper, and L. T. Simpson. Value numbering. *Software – Practice and Experience*, 27(6):701–724, 1997.

[6] P. Briggs, K. D. Cooper, and L. Torczon. Rematerialization. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI '92, pages 311–321. ACM, 1992.

[7] M. C. Carlisle and A. Rogers. Software caching and computation migration in Olden. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 29–38, New York, NY, USA, 1995. ACM.

[8] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to UPC and Language Specification. Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, 1999.

[9] B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the Chapel Language. *International Journal of High Performance Computing Application*, 21(3):291–312, Aug. 2007.

[10] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 519–538. ACM, 2005.

[11] W.-Y. Chen, D. Bonachea, C. Iancu, and K. Yelick. Automatic nonblocking communication for partitioned global address space programs. In *International Conference on Supercomputing, Proceedings*, pages 158–167. ACM, 2007.

[12] W.-Y. Chen, C. Iancu, and K. Yelick. Communication optimizations for fine-grained UPC applications. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, PACT '05, pages 267–278, Washington, DC, USA, 2005. IEEE Computer Society.

[13] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. In *ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 31–44, New York, NY, USA, 2007. ACM.

[14] C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, F. Cantonnet, T. El-Ghazawi, A. Mohanti, Y. Yao, and D. Chavarría-Miranda. An evaluation of global address space languages: Co-Array Fortran and Unified Parallel C. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 36–47. ACM, 2005.

[15] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, San Francisco, 1979.

[16] Graph 500. `http://www.graph500.org/`, July 2012.

[17] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Commun. ACM*, 35(8):66–80, Aug. 1992.

[18] B. Holt, J. Nelson, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Flat combining synchronized global data structures. In *International Conference on PGAS Programming Models (PGAS)*, Oct 2013.

[19] HPCC. HPCC random-access benchmark `http://icl.cs.utk.edu/hpcc/hpcc_results.cgi`.

[20] W. C. Hsieh, M. F. Kaashoek, and W. E. Weihl. Dynamic computation migration in DSM systems. In *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*, Supercomputing '96, Washington, DC, USA, 1996. IEEE Computer Society.

[21] W. C. Hsieh, P. Wang, and W. E. Weihl. Computation migration: Enhancing locality for distributed-memory parallel systems. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '93, pages 239–248, New York, NY, USA, 1993. ACM.

[22] ISO/IEC. Programming languages - C - Extensions to support embedded processors. Technical Report 18037, 2006.

[23] S. Jagannathan. Communication-passing style for coordination languages. In *Coordination Languages and Models*, pages 131–149. Springer, 1997.

[24] H. Kaiser, M. Brodowicz, and T. Sterling. Parallex: An advanced parallel execution model for scaling-impaired applications. In *Parallel Processing Workshops, 2009. ICPPW'09. International Conference on*, pages 394–401. IEEE, 2009.

[25] L. V. Kale and S. Krishnan. CHARM++: A portable concurrent object oriented system based on C++. In *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '93, pages 91–108, New York, NY, USA, 1993. ACM.

[26] P. M. Kogge. Of piglets and threadlets: Architectures for self-contained, mobile, memory programming. In *Innovative Architecture for Future Generation High-Performance Processors and Systems, Proceedings*, pages 130–138. IEEE, 2004.

[27] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO, pages 75–88. IEEE Computer Society, 2004.

[28] R. C. Murphy. *Traveling Threads: A New Multithreaded Execution Model*. PhD thesis, University of Notre Dame, 2006.

[29] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Grappa: A latency-tolerant runtime for large-scale irregular applications. Technical Report UW-CSE-14-02-01, University of Washington, 2 2014.

[30] NAS parallel benchmark suite 3.3. `http://www.nas.nasa.gov/publications/npb.html`, 2012.

[31] E. Raman, G. Ottoni, A. Raman, M. J. Bridges, and D. I. August. Parallel-stage decoupled software pipelining. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '08, pages 114–123, New York, NY, USA, 2008. ACM.

[32] A. Rogers, M. C. Carlisle, J. H. Reppy, and L. J. Hendren. Supporting dynamic data structures on distributed-memory

machines. *ACM Transactions on Programming Languages and Systems*, 17(2):233–263, Mar. 1995.

[33] A. Sanz, R. Asenjo, J. Lopez, R. Larrosa, A. Navarro, V. Litvinov, S.-E. Choi, and B. Chamberlain. Global data re-allocation via communication aggregation in Chapel. In *Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 235–242, Oct 2012.

[34] Y. Shiloach and U. Vishkin. An O(N log(N)) parallel max-flow algorithm. *Journal of Algorithms*, 3(2):128–146, 1982.

[35] J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer. Phasers: A unified deadlock-free construct for collective and point-to-point synchronization. In *International Conference on Supercomputing*, ICS '08, pages 277–288. ACM, 2008.

[36] E. Tilevich and Y. Smaragdakis. J-Orchestra: Automatic Java application partitioning. In B. Magnusson, editor, *ECOOP 2002 Object-Oriented Programming*, volume 2374 of *Lecture Notes in Computer Science*, pages 178–204. Springer Berlin Heidelberg, 2002.

[37] C. Wang and Z. Li. Parametric analysis for adaptive computation offloading. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 119–130, New York, NY, USA, 2004. ACM.

[38] L. Wang and M. Franz. Automatic partitioning of object-oriented programs for resource-constrained mobile devices with multiple distribution objectives. In *International Conference on Parallel and Distributed Systems (ICPADS'08)*, pages 369–376. IEEE, 2008.