

Runtime Optimizations for Irregular Applications in Chapel

Thomas B. Rolinger
tbrolin@cs.umd.edu
University of Maryland
College Park, MD, USA

Christopher D. Krieger
krieger@lps.umd.edu
Laboratory for Physical Sciences
College Park, MD, USA

Alan Sussman
als@cs.umd.edu
University of Maryland
College Park, MD, USA

ABSTRACT

Programming languages that implement the Partitioned Global Address Space (PGAS) model offer a simplified approach to writing parallel distributed applications, since explicit message passing is abstracted from the user. While the PGAS model offers high productivity, communication costs can still be a bottleneck for application performance. Applications that exhibit sparse and indirect memory accesses to distributed data pose a significant challenge to performance. These irregular applications lack spatial and temporal locality, leading to fine-grained remote communication that is not known until runtime. In this work, we investigate runtime optimizations for distributed irregular applications within the Chapel programming language. We focus on the inspector-executor technique, which evaluates a kernel of interest at runtime and constructs an optimized version of that kernel for execution. For our preliminary study, we hand-code the inspector and executor to demonstrate that runtime speed-ups as large as 224x, 13x and 96x are possible for Conjugate Gradient, a molecular dynamics simulation, and PageRank kernels, respectively.

KEYWORDS

PGAS, Chapel, irregular applications, runtime optimizations, inspector executor

1 INTRODUCTION

In a Partitioned Global Address Space (PGAS) model, memory that is physically distributed across a system is viewed logically as a single global address space, allowing programmers to develop parallel algorithms without expressing explicit data movement between remote processes or specifying low-level data distribution details. Languages that implement the PGAS model include Chapel [4], UPC [8] and GlobalArrays [14]. While these languages offer high productivity for developing distributed applications, achieving high performance still relies on optimizations that reduce remote communication. Such optimizations include proper data partitioning and message aggregation.

Applications that exhibit sparse, indirect memory accesses pose additional performance challenges for distributed systems, as they lead to fine-grained remote communication with little spatial and temporal locality. Such memory access patterns are not known until runtime, precluding many static compiler optimizations. While the PGAS model allows for irregular applications to be distributed relatively easily, it may hinder optimizations by abstracting the difference between a remote and local memory access at the source code level. This potentially puts more of a burden on the compiler and runtime system to perform optimizations, as they may have more insight into the underlying remote communication patterns than the application developer.

In this work, we investigate runtime optimizations for distributed irregular applications within the Chapel programming language. We use an inspector-executor technique [5, 15, 17]. In this approach, an *inspector* runs prior to a kernel of interest to gather memory access information, such as which accesses result in remote communication. A new version of the kernel is constructed that is optimized using the information obtained by the inspector. This optimized kernel is called the *executor*. The inspector-executor optimization is typically applied to applications that execute several iterations of a kernel, where the memory access pattern remains the same for each iteration. In such applications, the one-time cost of the inspector can be amortized over multiple iterations of the optimized kernel.

To evaluate our optimization, we implement three distributed irregular applications in Chapel and apply the inspector-executor technique. The design and implementation of these applications adhere to Chapel’s principle of separating the algorithm from the data distribution, so they are written with limited low-level details available. As a result, our evaluation shows that not only can the inspector-executor provide large performance gains but it is applicable to codes written by non-expert Chapel developers.

Our work makes the following contributions:

- Design of an inspector-executor that evaluates accesses to a distributed array and replicates remotely accessed data to be used locally.
- Application of the inspector-executor to three irregular codes: conjugate gradient (CG), molecular dynamics (moldyn), and PageRank.
- Performance evaluation of the inspector-executor across the three irregular applications on a distributed memory system. We demonstrate average runtime speed-ups of 27x for NAS-CG, 8x for moldyn, and 11x for PageRank.

While the inspector-executor technique can be implemented as a compiler optimization, we present hand-coded implementations in this preliminary study to demonstrate that runtime speed-ups are possible. We intend to have the compiler automatically generate the inspector and executor in future work.

The rest of this paper is outlined as follows. Section 2 describes our inspector-executor design. Section 3 presents the three irregular applications we study, including their implementation details and how the inspector-executor is applied in each case. We present a performance evaluation of our optimization across the applications in Section 4. Finally, we provide concluding remarks and plans for future work in Section 5.

2 INSPECTOR-EXECUTOR DESIGN

In this section, we describe the high-level design of our inspector and executor, as well as provide details of the data structures and

algorithms used to support our design. The goal of the inspector-executor is to reduce remote communication during kernel execution by replicating the data that is accessed remotely on each locale, thus enabling only local access during the kernel execution. At a high-level, our inspector-executor is similar to the work by Su and Yelick [18], which targeted the Titanium PGAS language, and the PARTI library [5], which is a set of primitives for generating inspector-executors for distributed irregular codes written in C and Fortran. Our work and theirs share the same goal of storing remotely accessed data for local use during the execution of a kernel of interest, but we target a more modern language, namely Chapel. The work by Kayraklioglu et al. [12] provides support for locality-aware prefetching of remote data in Chapel, as well as a means to log access patterns to remote data for offline analysis [11]. Our work differs from theirs in that we aim to perform analysis and optimizations at runtime.

2.1 High-level Design

Access patterns of the form $A[B[i]]$ are the target for our inspector-executor, where the elements of B are not known until runtime and result in indirect, and potentially remote, accesses to the block distributed array A . For example, A could be an array of vertices in a graph and B array could be the neighbor lists for the vertices. We will refer to B as the *interactions* array and A as the *data* array. The types of kernels on which we focus are those written in a data-parallel manner, such as `forall` loops that iterate over each element in a distributed array. To this end, our inspector-executor targets kernels that exhibit both shared- and distributed-memory parallelism.

The purpose of the inspector is to analyze the memory access pattern of the kernel to determine which accesses will result in remote communication to a data array. Elements that result in remote accesses are replicated on the locale that issued the access, and the inspector builds a data structure on each locale that stores a mapping from $B[i]$ to $A[B[i]]$. A key feature of the inspector is that it does not actually perform the remote access, but rather checks whether the access would be remote based on the distribution information of the data array, which is available through Chapel's array/domain interface.

Another crucial feature is that remote elements are only replicated once, regardless of the number of times they are remotely accessed. This allows the executor to fully take advantage of any data reuse in the kernel. The purpose of the executor is to perform the same computation as the original kernel, but redirect all remote accesses to the locally replicated copies. Prior to executing the kernel, the replicated elements on each locale need to be updated to reflect the current values in the original data array, as they may have been changed outside of the kernel (e.g., in a prior loop iteration). As the inspector only allocates a single space for a remote value, the executor incurs the cost of one remote read per replicated element, but then performs local accesses for however many times the element is accessed in the kernel.

Another case that needs to be considered is if the elements in the data array are written to during the kernel, which may result in writes being issued to the locally stored copies on the locales. In

this scenario, the executor is responsible for pushing these updates to the original data array at the end of the kernel.

2.2 Replicating Remote Elements

To implement the inspector and executor, we need a suitable data structure to replicate the remote indices/values that each locale accesses. For this purpose, we use Chapel's associative arrays, as they provide a method for storing sparse indices with relatively fast access times. They also automatically ignore duplicate indices and avoid the need to construct new index mappings. Listing 1 presents the definition of our `SparseBuffer` record, which is used by each locale to store its replicated values.

```

1 record SparseBuffer {
2   type elem_type;
3   var spD : domain(int);
4   var arr : [spD] elem_type;
5   var start_idx, end_idx, num_elems : int;
6   var D : domain(1) = {0..#num_elems};
7   var indices : [D] int; // sorted indices
8 }
```

Listing 1: `SparseBuffer` record to store remote elements for a locale

An alternative approach to associative arrays would be a normal Chapel array, which would provide better spatial locality and faster access times overall. However, that approach would require remapping $B[i]$ to the indices in the array that hold the replicated values. In future work, we intend to implement this approach and evaluate it, as we suspect it will provide performance improvements over the associative array implementation.

2.3 Updating Replicated Elements

As noted in Section 2.1, the executor needs to update the remote elements that are replicated on each locale prior to executing the kernel. We employ a *gather* strategy, which is straightforward to implement using the `SparseBuffer` record described in Section 2.2. Prior to the kernel code, the executor inserts a loop that iterates through each locale's set of replicated indices and issues remote reads to the corresponding values in the data array. These gathers can be performed in parallel across all locales. We suspect that aggregation optimizations can improve the gathers, which we intend to explore in future work.

Some kernels may issue writes to the data array, which in turn can lead to replicated elements with different values on each locale. It is the executor's responsibility to ensure that the original data array is updated with these partial results so that it is identical to an execution without the inspector-executor optimization. To do this, the executor adds a loop at the end of the kernel that pushes each replicated value to the original data array. Since multiple locales can replicate the same element, these updates must be performed atomically. Currently, we only consider updates that can be formulated as reduction operations. We show an example in Section 3.2.

2.4 Effective Use of Remote Cache

Chapel includes a per-core runtime managed cache for remote data, based on the work by Ferguson and Buettner [9]. Each cache entry corresponds to a 1024 byte cache page, where each cache line in the page is 64 bytes. Without any user intervention, remote

reads will be rounded up to fill a cache line and remote writes will be aggregated if they occur nearby. Enabling this remote cache for a program can significantly improve performance, even in the presence of irregular access patterns. However, as is the case with a traditional CPU cache, performance can be further improved by effectively utilizing the remote cache.

The approach described in Section 2.3 for updating replicated elements iterates over the indices of each locale’s associative array. Since indices of associative arrays are not sorted, the updates perform irregular remote reads to the data array, making poor use of the remote cache. A simple but effective optimization is to include a step in the inspector to create a sorted array of the indices for each locale’s associative array. The sorted indices are then iterated over for the updates in the executor, rather than the associative array itself. We evaluate the effectiveness of this sorting in Section 4.

3 APPLYING THE INSPECTOR-EXECUTOR

In this section we describe specific examples of the inspector-executor technique applied to three irregular applications: conjugate gradient, molecular dynamics simulation, and PageRank. We note that the target kernels in these applications are performance bottlenecks, constituting more than 95% of the applications’ total runtime. We also provide details of these applications in terms of their data structures, indirect memory accesses, and parallelization. Some code syntax and details that are not crucial to understanding our work are omitted for brevity.

3.1 Conjugate Gradient

The conjugate gradient (CG) method solves the equation $Ax = b$ for x , where A is a symmetric positive-definite matrix and x and b are vectors. The CG method can be used to solve unstructured optimization problems and partial differential equations. The iterative approach to CG is used when A is large and sparse. The bottleneck for CG is repeated sparse matrix-vector multiplies (SpMV), where the memory access pattern remains the same across each SpMV. This allows for the inspector-executor technique to be applied.

For our Chapel implementation of CG, we store the vectors x and b as one-dimensional block distributed arrays. For the sparse matrix A , we represent each row as a record, which contains a sorted array of the row’s column indices and an array of the corresponding non-zero values. All row records are stored in a one-dimensional block distributed array, with the same distribution as the x and b arrays. The baseline SpMV kernel is shown in Listing 2, where `Rows` is the block distributed array of row records. As can be seen on line 4, the indirect accesses are to x , as dictated by the `col_idx` array. All writes to the array b on line 6 are local, as the locale that owns row also owns the corresponding element of b .

```

1 forall row in Rows {
2   var accum : real = 0;
3   for k in 0..#row.nnz {
4     accum += row.value[k] * x[row.col_idx[k]];
5   }
6   b[row.id] = accum;
7 }
```

Listing 2: Baseline SpMV kernel

The inspector we generate for SpMV is shown in Listing 3. The `localeBuffers` array is the distributed array of `SparseBuffer` instances. Lines 1–11 perform the SpMV kernel, but without actually executing the accesses to x . Instead, when a given index is determined to be remote, the index is added to the locale’s associative domain `spD` (lines 6–9). The bounds check on line 7 is specifically designed for block distributed arrays. A more general method would be to use the `.contains()` method on the locale’s local subdomain. However, doing so is significantly slower than the bounds check. We plan to investigate approaches to this that are both general and efficient. Line 12 creates a sorted array of the remote indices on each locale, as per the optimization described in Section 2.4. The generated executor for the SpMV kernel is shown in Listing 4. Lines 1–5 perform the gather strategy discussed in Section 2.3. Lines 6–21 perform the SpMV kernel, with the modification of accessing remote elements in the locale’s associative array `arr` rather than x .

```

1 forall row in Rows {
2   const start = localeBuffers[here.id].start_idx;
3   const end = localeBuffers[here.id].end_idx;
4   ref spD = localeBuffers[here.id].spD;
5   for k in 0..#row.nnz {
6     const idx = row.col_idx[k];
7     if idx < start || idx > end {
8       spD += idx;
9     }
10  }
11 }
12 sort_indices(localeBuffers);
```

Listing 3: Inspector for SpMV kernel

```

1 forall buff in localeBuffers {
2   forall idx in buff.indices {
3     buff.arr[idx] = x[idx];
4   }
5 }
6 forall row in Rows {
7   const start = localeBuffers[here.id].start_idx;
8   const end = localeBuffers[here.id].end_idx;
9   ref arr = localeBuffers[here.id].arr;
10  var accum : real = 0;
11  for k in 0..#row.nnz {
12   const idx = row.col_idx[k];
13   if idx < start || idx > end {
14     accum += row.value[k] * arr[idx];
15   }
16   else {
17     accum += row.value[k] * x[idx];
18   }
19 }
20 b[row.id] = accum;
21 }
```

Listing 4: Executor for SpMV kernel

3.2 Moldyn

Molecular dynamics simulations analyze how a system of particles evolves over time, where particles interact with other particles to influence their positions, forces and velocities. Particles can be atoms, molecules, or even large astronomical objects. The underlying kernel for molecular dynamics simulations is essentially operating on a graph, where an edge exists between two particles that interact with each other. In this work, we implement the moldyn benchmark

from the CHARMM molecular dynamics application, which has been used in prior work for data/computation reordering optimizations [10, 16]. The inspector-executor technique can be applied to moldyn because there are several time-step iterations that are performed before the particle interactions are updated. As a result, the memory access pattern remains fixed during these time steps.

```

1 forall pi in data {
2   var i = pi.id;
3   for j in 0..#pi.num_neighbors {
4     ref pj = data[pi.neighbors[j]];
5     const rd = // distance between pi and pj
6     if rd < cutoff {
7       acquireLock(i); pi.forces += ...; releaseLock(i);
8       acquireLock(j); pj.forces -= ...; releaseLock(j);
9     }
10  }
11 }

```

Listing 5: Baseline moldyn kernel

We focus on the kernel that computes the interaction force updates, as shown in Listing 5, where `data` is a one-dimensional block distributed array of records, each representing a particle. Each record has an array of sorted indices that correspond to the particles it interacts with (`neighbors`) along with fields for its forces, position and velocity. Each particle i only stores interactions with particle j if $i < j$ (i.e., only one side of the undirected edge). For a given set of locales, the locales with higher IDs will be assigned fewer interactions, leading to less remote communication. In the case of the “last” locale, it will never require remote communication. The kernel operates on a per-particle basis, iterating over each particle in `data` and then over the particle’s interactions. Line 4 contains the indirect access to the data array. We utilize a pool of locks to address the race condition on lines 7 and 8.

```

1 forall buff in localeBuffers {
2   forall idx in buff.indices {
3     buff.arr[idx].forces = data[idx].forces;
4     buff.arr[idx].coords = data[idx].coords;
5   }
6 }
7 forall pi in data {
8   const start = localeBuffers[here.id].start_idx;
9   const end = localeBuffers[here.id].end_idx;
10  ref arr = localeBuffers[here.id].arr;
11  for j in 0..#pi.num_neighbors {
12    const idx = pi.neighbors[j];
13    ref pj : Particle;
14    if idx < start || idx > end {
15      pj = arr[idx];
16    }
17    else {
18      pj = data[idx];
19    }
20    // rest of loop is the same
21  }
22 }
23 forall buff in localeBuffers {
24   for i in buff.indices {
25     acquireLock(i);
26     data[i].forces += buff.arr[i].forces;
27     releaseLock(i);
28   }
29 }

```

Listing 6: Executor for moldyn kernel

The inspector for moldyn follows the same structure as shown for SpMV in Listing 3, except that the `SparseBuffer` instances store records (particles) instead of reals. The executor for moldyn is shown in Listing 6. Lines 1–6 use the gather-based approach to update each locale’s replicated elements. We only require specific fields of the record to be replicated, namely the forces and coordinates. Those fields are the only ones required for reading and/or writing during the kernel. It is crucial that the executor only copy over these fields, as copying the entire record, which includes the neighbors array, would induce significant overhead. Lines 23–29 update the original data array after the kernel, scattering the values of the replicated elements since they were modified during the kernel. This scatter requires synchronization, as a given particle may have been replicated by more than one locale. For that reason, we opt to not perform the typical nested `forall` loop as we do on lines 1–6, as it leads to more lock contention rather than improved performance. Worth noting is that prior to the kernel, the moldyn application sets each particle’s forces to 0. This allows the executor to perform the sum reduction across the replicated copies into the global version.

3.3 PageRank

The PageRank algorithm operates over a graph and provides a ranking of its vertices in terms of their perceived importance [3]. A vertex’s importance is determined by both the number of incoming edges as well as the importance of the vertices incident to those edges. The original application of PageRank targeted web graphs, where vertices are web pages and a directed edge from web page i to web page j indicates there is a hyperlink on i to j . The output of PageRank is an assignment of probabilities to each vertex, which represents the likelihood that a web surfer randomly clicking links will arrive at a given vertex. These probabilities are referred to as pageranks.

The PageRank algorithm is iterative, where each iteration updates the pagerank value for all vertices based on the previous iteration’s results. The inspector-executor approach is applicable to PageRank because there are indirect accesses to the graph to access a given vertex’s neighbors, and the structure of the graph remains fixed throughout the entire algorithm. For our implementation, we represent the input graph as a block distributed array of `Vertex` records, where each record contains a sorted array of the vertex’s incoming neighbors as well as other metadata such as its pagerank value and in/out degree. This is nearly identical to the adjacency list data structure used to represent particles in the moldyn application.

```

1 var isolated_val = calc_sinks();
2 forall v in A {
3   var val = 0.0;
4   for i in 0..#v.in_degree {
5     ref t = A[v.in_neighbors[i]];
6     if t.out_degree > 0 {
7       val += t.pr_read/t.out_degree;
8     }
9   }
10  v.pr_write = (val*d)+((1-d)/num_vertices) + isolated_val;
11 }

```

Listing 7: Baseline PageRank kernel

Listing 7 presents the baseline PageRank kernel. Line 1 handles what are called *sink vertices*, which are vertices with no outgoing edges. If a random web surfer were to end up on such a page, they would not have any links to click on. The PageRank algorithm handles this by assuming the surfer would randomly jump to another web page, where there is an equal probability to go to any page. These probabilities are accumulated into `isolated_val` and factored into every vertex’s new pagerank score (line 10). Worth noting is that there are no remote accesses issued during the computation on line 1. Lines 2–11 perform the main PageRank kernel, iterating over each vertex in the distributed array `A` and computing its new pagerank. Line 5 contains the indirect memory access to `A`. As PageRank is iterative and uses the previous iteration’s results in the current iteration, each vertex stores the previous iteration’s pagerank in `pr_read` and the current iteration’s in `pr_write`. This also allows each vertex to be updated independently, and thus updates can be performed in parallel. The variable `d` is a constant known as the damping factor and is set to 0.85.

The inspector for PageRank follows the same concept as SpMV and moldyn, so we omit its description. We do note, however, that the inspector does not need to perform the sink vertices computations, as those do not contain any indirect remote accesses. Also, the inspector performs the initialization of each replicated element’s out-degree field, as it is needed during the kernel but never changes throughout the application. The executor for PageRank is straightforward and follows the same concepts described for SpMV and moldyn, so we also omit its description.

4 PERFORMANCE EVALUATION

In this section we evaluate our inspector-executor optimization on the three irregular applications described in Section 3. For each application we present runtime speed-ups achieved by our optimizations when compared to the original code, factoring in any overhead introduced by the inspector and executor. We also discuss the runtime overhead of the inspector and memory consumption increases as they relate to our optimization.

4.1 Experimental Setup

For all experiments, we run our applications on a 16-node FDR Infiniband cluster. Each node consists of two 10-core Xeon E5-2650v3 CPUs with hyper-threading enabled and 512 GB of DDR4 memory. All applications are built and compiled using Chapel 1.24.1 and LLVM 11.0.1 with the `-fast` and `-cache-remote` flags set. The communication configuration used is GASNet over Infiniband. Each node is designated as a locale, and we will use the terms interchangeably. Each experiment involving runtime measurements ran for multiple trials, and the averages of those trials are presented. The coefficient of variation across all experiments and trials did not exceed 0.07. The coefficient of variation is the standard deviation divided by the mean and provides a measure of variability. Unless stated otherwise, all experiments utilize the sorting optimization described in Section 2.4.

To provide a comparison to the inspector-executor, we also evaluate a “replicate everything” approach, which we refer to as *replicate-all*. In *replicate-all*, instead of using the inspector to determine

Table 1: Data Sets for NAS-CG

Name	Rows	Non-zeros	Density (%)	Memory
C	150k	39M	0.17	0.8 GB
D	150k	73M	0.32	1.3 GB
E	9M	6.6B	$8.1e-5$	115 GB

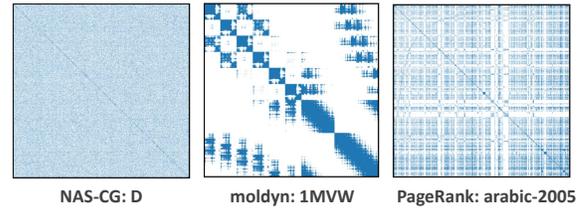


Figure 1: Examples of the sparsity structure for different data sets evaluated.

which accesses are remote, we simply replicate the entire data array on each locale. This provides a comparison point for memory consumption and the overhead of the inspector’s memory access analysis. For the remainder of this section, we will refer to the inspector-executor as “I/E”.

4.2 Conjugate Gradient: NAS-CG

To evaluate the I/E on conjugate gradient, we utilize the NAS Parallel Benchmarks specifications [1], which we refer to as NAS-CG. NAS-CG provides a specification for generating sparse matrices of different sizes but with similar non-zero structure. Each problem size is specified to run for a certain number of iterations. Regardless of the problem size, each iteration of the NAS-CG benchmark executes 25 sparse matrix-vector multiply (SpMV) kernels. We focus our evaluation on problem sizes C, D and E, with the properties described in Table 1. The reported memory footprint in Table 1 is the total amount of memory required to represent the sparse matrix and dense input/output vectors in Chapel. The density column refers to the percentage of elements that are non-zero. The sparsity pattern of these matrices can be described as random with a uniform distribution of non-zeros per row, as shown on the left in Figure 1. Other benchmarks for CG, such as HPCG [7], use sparse matrices that are not purely random and can exhibit different communication behavior. We leave evaluation of those benchmarks for future work.

In our experiments, we run each data set for the full number of iterations as specified by the NAS benchmark. Problem C is specified to execute for 75 iterations and problem sizes D and E both execute for 100 iterations.

4.2.1 Memory Usage. For the I/E, we observe that the geometric increase in memory usage over the baseline implementation is 6% across the different data sets and locale counts, with a minimum of 0.5% and a maximum of 21%. In counting the raw number of elements replicated, both the I/E and *replicate-all* perform a near equal amount of replication. This is due to the nature of the sparse matrices, which have a uniformly random sparsity structure. As a result, each locale requires access to virtually every element of the input vector, leading to a full copy of the vector per locale. However,

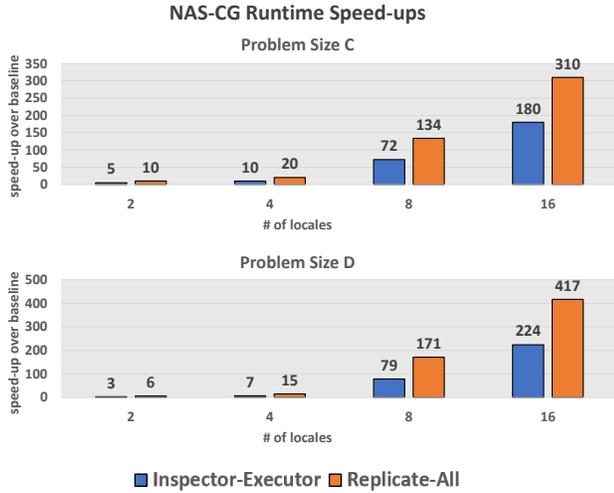


Figure 2: Runtime speed-ups on NAS-CG for the C and D data sets in Table 1. Speed-ups are shown for the inspector-executor and replicate-all relative to the baseline performance for each locale count. Higher bars represent better performance. Values on top of each bar show the speed-up achieved.

the replicate-all approach increases memory usage over the baseline by an average of 1%. There is a larger overhead for the I/E since it uses an associative array to store the per-locale replicated elements, while replicate-all uses a normal array on each locale. While these uniformly random matrices reduce the impact of the I/E, they serve the useful purpose of highlighting the performance differences between the I/E and replicate-all that may otherwise be obscured by the I/E replicating fewer elements.

For the I/E, the memory usage overhead for a given problem size increases with the number of locales, as each locale essentially introduces a full copy of the vector. Indeed, we observe an average increase in memory usage of 7% for 16 locales when compared to two locales for all problem sizes. The increase is relatively small due to the fact that the input vector, which is what is replicated, contributes only a small portion to the overall memory usage and is overshadowed by the sparse matrix.

4.2.2 Inspector Runtime Overhead. Across the different data sets and locale counts, we observe a geometric mean runtime overhead of 4% for the inspector relative to the total execution time, with a minimum of 2% and a maximum of 6%. As the locale count increases for a given data set, the inspector’s runtime does generally decrease, since it is parallelized. Indeed, for problem size E, the inspector is 4.6x faster on 16 locales compared to two locales. However, the scalability can be limited due to an increase in overall remote communication, which results in more work performed by the inspector.

4.2.3 Runtime Performance. Figure 2 shows the runtime speed-ups relative to the baseline achieved by the I/E and replicate-all for the C and D data sets from Table 1. These speed-up results are for the total runtime for NAS-CG (excluding I/O), which includes the cost of the inspector. The I/E achieves a geometric mean speed-up of 27x across the C and D data sets for all locales, with a minimum speed-up of 3x and a maximum of 224x. For replicate-all, the geometric mean speed-up achieved is 53x, with a minimum of 6x and a maximum of 417x.

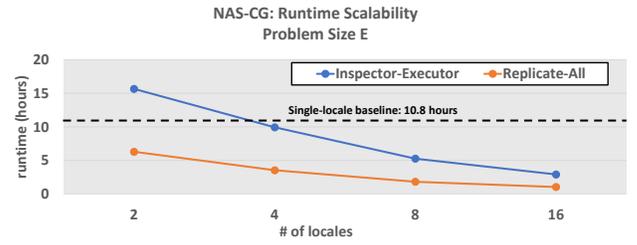


Figure 3: Runtime performance (in hours) on problem size E for inspector-executor and replicate-all. The horizontal dotted line represents the single-locale baseline runtime performance.

While both replicate-all and the I/E replicate the same amount of data, as described in Section 4.2.1, replicate-all uses normal Chapel arrays for accessing the replicated data while the I/E uses associative arrays. Because associative arrays have a much higher access time when compared to normal arrays, replicate-all outperforms the I/E. Furthermore, the inspector phase requires significantly more time than what is required for replicate-all, which is simply allocating arrays on each locale. However, as the locale count increases, the inspector phase becomes faster, leading to a smaller difference in speed-ups.

We observe that both the I/E and replicate-all achieve higher speed-ups as the number of locales increase. This is due to the baseline runtime increasing with more locales, while the I/E and replicate-all both achieve lower runtimes with more locales. Indeed, the baseline performance at 16 locales is 11x and 16x slower than the two locale performance for the C and D matrices, respectively. On the other hand, the I/E is 3x and 5x faster on 16 locales versus two locales for the C and D matrices, respectively.

Not included in Figure 2 are results for problem size E. We are not able to execute the multi-locale baseline implementation on problem size E for a single iteration of the NAS-CG benchmark in a reasonable amount of time. We found that a single SpMV kernel for problem size E with the baseline code requires several hours to complete on two locales (a single iteration of NAS-CG performs 25 SpMV kernels). However, the baseline can execute in a reasonable amount of time on a single locale. For the I/E and replicate-all, we are able to run problem size E to completion on multiple locales. Figure 3 shows the runtime scalability of the I/E and replicate-all on problem size E. The horizontal dotted-line represents the single-locale baseline runtime. Worth noting is that both approaches can outperform the single-local baseline performance for relatively small locale counts. This further highlights the advantage of the I/E to not only improve baseline multi-locale performance but to provide gains over single-locale performance as well.

4.2.4 Effects of Sorting for Remote Cache Locality. In Section 2.4 we described the importance of effectively utilizing the remote cache by sorting the remote indices that are accessed in the update routine within the executor. To demonstrate this, we performed NAS-CG using the I/E without sorting enabled. We found that for problem sizes C and D across different locale counts, sorting provides a geometric mean speed-up of 8.5x over not sorting, with a minimum of 2x and a maximum of 22x. This is largely due to the difference in hit

Table 2: Data Sets for Moldyn

Name	Particles	Interactions	Density (%)	Memory
4B2Q	70k	163M	6.6	1.4 GB
4BOG	90k	310M	7.7	2.6 GB
1M8Q	75k	468M	16.7	3.8 GB
1MVW	92k	600M	14.1	4.9 GB

rates for the remote cache. When sorting is disabled, the update routine achieves an average hit rate of 17%, versus 98% with sorting enabled.

4.3 Moldyn

We execute the moldyn application for 10 time steps across real-world data sets obtained from the RCSB Protein Data Bank [2], presented in Table 2. The density column refers to the density of the adjacency matrix representation of the interactions (i.e., percentage of non-zeros). The memory footprint reported is the total memory required to store the particles and interactions. While the sparsity structure of the moldyn data sets is highly irregular, as illustrated in the middle of Figure 1 for the 1MVW data set, interactions tend to be clustered in groups that results in blocks of contiguous memory accesses to the data array.

4.3.1 Memory Usage. For the data sets in Table 2 across all locale counts, we observe that the I/E has an average memory overhead of 4% over the baseline, with a minimum of 0% and a maximum of 13%. The 4B2Q data set is perfectly partitioned for two locales, where there is no remote communication, so there is no replication performed. On the other hand, replicate-all has an average memory overhead of 21%, with a minimum of 4% and a maximum of 80%. There is a stark difference between the I/E and replicate-all because each locale does not issue remote accesses to each particle, resulting in less replication. This is in contrast to what we observed for NAS-CG.

For the I/E, increasing the number of locales leads to a larger memory footprint, as we observed with NAS-CG. This is because increasing the number of locales results in more remote communication, as the data partitioning becomes more fine-grained and each locale owns a smaller portion of the distributed array. On average, we observe an 8% increase in memory usage for 16 locales across the data sets when compared to two locales.

4.3.2 Inspector Runtime Overhead. The average runtime overhead of the inspector for the moldyn experiments is 24%, with a minimum of 0.2% and a maximum of 46%. We observe much higher overheads in moldyn than for NAS-CG because of the amount of remote communication required, as well as the relatively few number of iterations performed. As mentioned in Section 3.2, for a given set of locales, each locale requires less remote communication than the previous, with the “last” locale requiring no remote communication. This is because we store the upper-triangle of the adjacency matrix that represents the interactions, and we distribute the matrix in blocks of contiguous rows. Naturally, locales that own rows closer to the “bottom” of the matrix are responsible for fewer interactions. As a result, there is not as much remote communication to exploit to amortize the cost of the inspector overhead over the 10 iterations.



Figure 4: Runtime speed-ups on the moldyn application for the data sets in Table 2. Speed-ups are shown for the inspector-executor and replicate-all relative to the baseline performance for each locale count. Higher bars represent better performance. Values on top of each bar show the speed-up achieved.

4.3.3 Runtime Performance. Figure 4 shows the runtime speed-ups relative to the baseline achieved by the I/E and replicate-all. These speed-up results are for the total runtime (excluding I/O), which includes the cost of the inspector. Across all data sets and locale counts, we observe a geomean speed-up of 8x for both the I/E and replicate-all. Both approaches achieve no speed-ups on the 4B2Q data set for two locales, since there is no remote communication.

In general, the I/E outperforms replicate-all due to the time required to perform the updates before and after the kernel, which becomes more apparent at higher locale counts. While replicate-all stores a full copy of the data array on each locale and must iterate over all of the elements when updating, the I/E is only replicating the elements that are remotely accessed. We observe, on average, that replicate-all spends 5x longer performing the updates than the I/E. However, the I/E’s overall runtime performance is negatively impacted by the overhead of the inspector as well as the slow associative array accesses, leading to cases where replicate-all outperforms the I/E on the 1M8Q and 1MVW data sets.

In terms of scalability, the baseline performance achieves an average speed-up of 1.2x from two locales to 16 locales. For the I/E and replicate-all, the average speed-up is 2.1x and 1.6x, respectively. The scalability of all approaches is hindered by the use of locks within the kernel. In future work, we intend to instead use atomic variables.

4.4 PageRank

For PageRank, we evaluate a set of real-world web graphs from the SuiteSparse matrix collection [6] as well as generated graphs from the Graph500 benchmark [13]. These graphs are described in Table 3, where the “g500” graphs are the Graph500 related graphs. The sparsity structure of these graphs is highly irregular, as shown on the right in Figure 1 for the arabic-2005 graph. The Graph500 graphs are similar in structure to the uniformly random sparse NAS-CG matrices. Table 3 also provides the number of iterations required for the PageRank algorithm to converge for each graph, assuming a tolerance of $1e-10$ and a damping factor of 0.85. Our choice of damping factor matches the value used commonly in

Table 3: Data Sets for PageRank

Name	Vertices	Edges	Density (%)	Memory	Iterations
arabic-2005	23M	630M	1.2e-4	26 GB	94
sk-2005	51M	1.9B	7.5e-5	63 GB	82
g500_scale-26	67M	2.1B	4.7e-5	79 GB	29
g500_scale-28	268M	8.5B	1.2e-5	318 GB	20

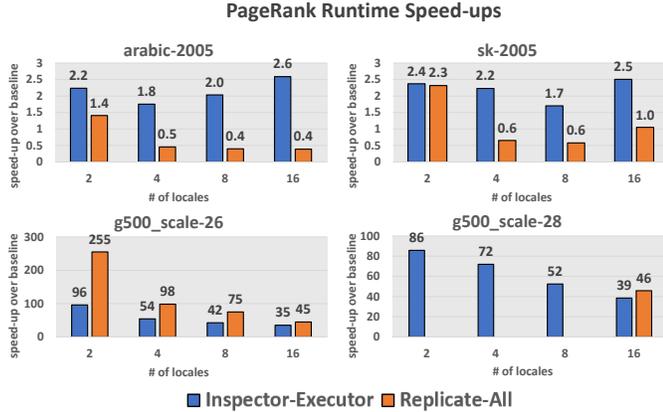


Figure 5: Runtime speed-ups on the PageRank application for the data sets in Table 3. Speed-ups are shown for the inspector-executor and replicate-all relative to the baseline performance for each locale count. Higher bars represent better performance. Values on top of each bar show the speed-up achieved. For the g500_scale-28 graph, replicate-all cannot run on 2, 4 and 8 locales because of out-of-memory errors.

practice. For the Graph500 graphs, we only run a single iteration for the baseline, extrapolating the runtime for the full iterations. This is because the baseline requires as much as 24 hours to run a single iteration.

4.4.1 Memory Usage. We observe that on average, the I/E has a memory usage overhead of 80% across all locale counts and data sets in Table 3 when compared to the baseline, with a minimum of 17% and a maximum of 360%. The high average overhead mainly comes from the large Graph500 data sets. If we focus on the real-world web graphs, we observe that the I/E’s average memory overhead is 42%.

For replicate-all, the average memory overhead across all data sets is 606%, with a minimum of 250% and a maximum of 1,400%. However, replicate-all cannot perform PageRank on the g500_scale-28 graph on fewer than 16 locales, as it runs out of memory. On the other hand, the I/E is able to process the g500_scale-28 graph on all locale counts. If we exclude the Graph500 graphs and only look at the real-world web graphs, replicate-all’s overhead is 565% versus the I/E’s 42%. This indicates that the real-world web graphs exhibit structure that the I/E is able to exploit without having to replicate all of the elements, highlighting the benefits of runtime analysis.

4.4.2 Inspector Runtime Overhead. The inspector incurs, on average, a 5% runtime overhead across all data sets and locale counts, with a minimum of 1.5% and a maximum of 14%. The runtime overhead is relatively low because the cost of the inspector is amortized over many iterations, as is the case for the arabic-2005 and sk-2005 graphs which execute for 94 and 82 iterations, respectively.

4.4.3 Runtime Performance. Figure 5 shows the runtime speed-ups relative to the baseline achieved by the I/E and replicate-all for the data sets in Table 3. We observe that the geomean speed-up across all data sets and locale counts for the I/E is 11x, with a minimum of 1.7x and a maximum of 96x. For replicate-all, the geomean speed-up is 5x, with a minimum of 0.4x and a maximum of 255x. However, as noted previously, replicate-all runs out of memory for the g500_scale-28 graph on 2, 4 and 8 locales. If we also exclude those results for the I/E, the geomean is 7x versus replicate-all’s 5x. In general, both approaches achieve smaller speed-ups as the number of locales increase, due to the baseline performance scaling well.

For the arabic-2005 and sk-2005 graphs, we observe that the I/E always outperforms replicate-all. This is because replicate-all stores a full copy of the array on each locale, requiring 3x more time on average to perform updates. This leads to replicate-all performing worse or the same as the baseline on the arabic-2005 and sk-2005 graphs for 4, 8 and 16 locales. For the g500_scale-26 and g500_scale-28 graphs, the I/E essentially replicates the same number of elements as replicate-all due to their uniformly random sparsity structure. As we saw with NAS-CG, replicate-all will outperform the I/E due to faster accesses into normal Chapel arrays versus associative arrays. However, as we also observe with NAS-CG, the performance becomes closer for more locales.

Worth noting is the wall clock time required by the baseline and I/E for the g500_scale-28 graph. On two locales, the baseline is projected to require 20 days to run the full 20 iterations, while the I/E only requires 6 hours. When using 16 locales, the runtime is 41 hours and one hour for the baseline and the I/E, respectively. The large difference in performance is due to the high amount of data reuse that the I/E can exploit, where it performs a single remote get for each replicated element but performs several local reads throughout the kernel.

5 CONCLUSIONS AND FUTURE WORK

In this work, we designed and evaluated the inspector-executor optimization for irregular applications in Chapel. Across three different applications, we observed speed-ups as high as 224x over baseline implementations. Furthermore, we showed that the inspector-executor optimization allows for applications to run on large-scale data sets in a matter of hours while the baseline implementation would require several days. Additionally, the inspector-executor is applied to codes that adhere to Chapel’s principle of separating the algorithm from the data distribution. As a result, our optimization does not require low-level details to be provided within the code. Instead, our inspector leverages the built-in mechanisms in the Chapel language to perform memory access analysis at runtime.

For future work, we plan to investigate optimizations to the inspector-executor, such as reducing memory access time by not using associative arrays. Additionally, we will incorporate aggregation-based optimizations to improve the updates performed by the executor. We also intend to reach out to the Chapel community to gather more applications to evaluate. Finally, our ultimate goal is to automate the generation of the inspector and executor within the compiler.

REFERENCES

- [1] David Bailey, Tim Harris, William Saphir, Rob Van Der Wijngaart, Alex Woo, and Maurice Yarrow. 1995. *The NAS Parallel Benchmarks 2.0*. Technical Report. Technical Report NAS-95-020, NASA Ames Research Center.
- [2] Helen M. Berman, John Westbrook, Zukang Feng, Gary Gilliland, T. N. Bhat, Helge Weissig, Ilya N. Shindyalov, and Philip E. Bourne. 2000. The Protein Data Bank. *Nucleic Acids Research* 28, 1 (01 2000), 235–242. <https://doi.org/10.1093/nar/28.1.235> arXiv:<https://academic.oup.com/nar/article-pdf/28/1/235/9895144/280235.pdf>
- [3] Monica Bianchini, Marco Gori, and Franco Scarselli. 2005. Inside Pagerank. *ACM Transactions on Internet Technology (TOIT)* 5, 1 (2005), 92–128.
- [4] Bradford L Chamberlain, Elliot Ronaghan, Ben Albrecht, Lydia Duncan, Michael Ferguson, Ben Harshbarger, David Iten, David Keaton, Vassily Litvinov, Preston Sahabu, et al. 2018. Chapel Comes of Age: Making Scalable Programming Productive. https://cug.org/proceedings/cug2018_proceedings/includes/files/pap130s2-file1.pdf
- [5] Raja Das, Mustafa Uysal, Joel Saltz, and Yuan-Shin Hwang. 1994. Communication Optimizations for Irregular Scientific Computations on Distributed Memory Architectures. *Journal of parallel and distributed computing* 22, 3 (1994), 462–478.
- [6] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (Dec. 2011), 25 pages. <https://doi.org/10.1145/2049662.2049663>
- [7] Jack Dongarra, Michael A Heroux, and Piotr Luszczek. 2016. High-performance Conjugate-gradient Benchmark: A New Metric for Ranking High-performance Computing Systems. *The International Journal of High Performance Computing Applications* 30, 1 (2016), 3–10.
- [8] Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick. 2005. *UPC: Distributed Shared Memory Programming*. Vol. 40. John Wiley & Sons, New York, NY, USA.
- [9] M. P. Ferguson and D. Buettner. 2015. Caching Puts and Gets in a PGAS Language Runtime. In *2015 9th International Conference on Partitioned Global Address Space Programming Models*. IEEE Xplore, USA, 13–24.
- [10] Hwansoo Han and Chau-Wen Tseng. 2000. A Comparison of Locality Transformations for Irregular Codes. In *International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*. Springer, Germany, 70–84.
- [11] Engin Kayraklioglu and Tarek El-Ghazawi. 2018. APAT: An Access Pattern Analysis Tool for Distributed Arrays. In *Proceedings of the 15th ACM International Conference on Computing Frontiers (Ischia, Italy) (CF '18)*. Association for Computing Machinery, New York, NY, USA, 248–251. <https://doi.org/10.1145/3203217.3203266>
- [12] Engin Kayraklioglu, Michael P. Ferguson, and Tarek El-Ghazawi. 2018. LAPPS: Locality-Aware Productive Prefetching Support for PGAS. *ACM Trans. Archit. Code Optim.* 15, 3, Article 28 (Aug. 2018), 26 pages. <https://doi.org/10.1145/3233299>
- [13] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. 2010. Introducing the Graph 500. *Cray Users Group (CUG)* 19 (2010), 45–74.
- [14] Jaroslaw Nieplocha, Robert J Harrison, and Richard J Littlefield. 1996. Global Arrays: A Nonuniform Memory Access Programming Model for High-performance Computers. *The Journal of Supercomputing* 10, 2 (1996), 169–189.
- [15] Joel H. Saltz, Ravi Mirchandaney, and Kay Crowley. 1991. Run-Time Parallelization and Scheduling of Loops. *IEEE Trans. Comput.* 40, 5 (May 1991), 603–612.
- [16] Michelle Mills Strout, Larry Carter, and Jeanne Ferrante. 2003. Compile-Time Composition of Run-Time Data and Iteration Reorderings. *SIGPLAN Not.* 38, 5 (May 2003), 91–102. <https://doi.org/10.1145/780822.781142>
- [17] Michelle Mills Strout, Mary Hall, and Catherine Olschanowsky. 2018. The Sparse Polyhedral Framework: Composing Compiler-generated Inspector-executor Code. *Proc. IEEE* 106, 11 (2018), 1921–1934.
- [18] J. Su and K. Yelick. 2005. Automatic Support for Irregular Computations in a High-level Language. In *19th IEEE International Parallel and Distributed Processing Symposium*. IEEE, USA. <https://doi.org/10.1109/IPDPS.2005.118>