Control Replication: Compiling Implicit Parallelism to Efficient SPMD with Logical Regions

Elliott Slaughter
Stanford University
SLAC National Accelerator Laboratory
slaughter@cs.stanford.edu

Wonchan Lee
Stanford University
wonchan@cs.stanford.edu

Sean Treichler
Stanford University
NVIDIA
sean@nvidia.com

Wen Zhang
Stanford University
zhangwen@cs.stanford.edu

Michael Bauer
NVIDIA
mbauer@nvidia.com

Galen Shipman
Los Alamos National Laboratory
gshipman@lanl.gov

Patrick McCormick
Los Alamos National Laboratory
pat@lanl.gov

Alex Aiken
Stanford University
aiken@cs.stanford.edu

ABSTRACT

We present *control replication*, a technique for generating high-performance and scalable SPMD code from implicitly parallel programs. In contrast to traditional parallel programming models that require the programmer to explicitly manage threads and the communication and synchronization between them, implicitly parallel programs have sequential execution semantics and by their nature avoid the pitfalls of explicitly parallel programming. However, without optimizations to distribute control overhead, scalability is often poor.

Performance on distributed-memory machines is especially sensitive to communication and synchronization in the program, and thus optimizations for these machines require an intimate understanding of a program's memory accesses. Control replication achieves particularly effective and predictable results by leveraging language support for first-class data partitioning in the source programming model. We evaluate an implementation of control replication for Regent and show that it achieves up to 99% parallel efficiency at 1024 nodes with absolute performance comparable to hand-written MPI(+X) codes.

CCS CONCEPTS

• **Software and its engineering → Parallel programming languages**; **Compilers**;

KEYWORDS

Control replication; Regent; Legion; regions; task-based runtimes

SC17, November 2017, Denver, CO, USA

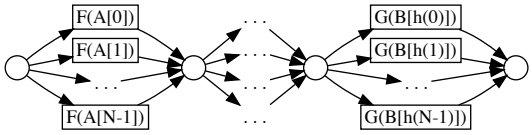
ACM ISBN 978-1-4503-5114-0/17/11…$15.00
https://doi.org/10.1145/3126908.3126949


```
1 for t = 0, T do
2   for i = 0, N do -- Parallel
3     B[i] = F(A[i])
4   end
5   for j = 0, N do -- Parallel
6     A[j] = G(B[h(j)])
7   end
8 end
```

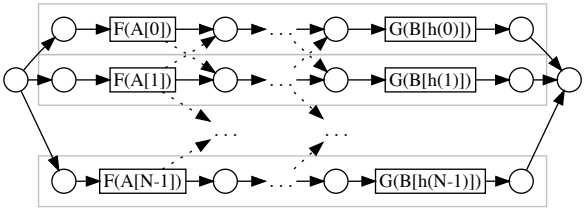
(a) Original program.

```
1 for i = 0, N do -- Parallel
2   for t = 0, T do
3     B[i] = F(A[i])
4     -- Synchronization needed
5     A[i] = G(B[h(i)])
6   end
7 end
```

(b) Transposed program.



(c) Implicitly parallel execution of original program.



(d) SPMD execution of transposed program.

Figure 1: Comparison of implicit and explicit parallelism.

1 INTRODUCTION

Programs with sequential semantics are easy to read, understand, debug, and maintain, compared to explicitly parallel codes. In certain cases, sequential programs also lend themselves naturally to parallel execution. Consider the code in Figure 1a. Assuming there are no loop carried dependencies, the iterations of each of the two inner loops can be executed in parallel on multiple processors in a straightforward fork-join style. As illustrated in Figure 1c, a main thread launches a number of worker threads for the first loop, each

of which executes one (or more) loop iterations. There is a synchronization point at the end of the loop where control returns to the main thread; the second loop is executed similarly. Because the second loop can have a completely different data access pattern than the first (indicated by the arbitrary function h in `B[h(j)]`), complex algorithms can be expressed. With considerable variation, this *implicitly parallel* style is the basis of many parallel programming models. Classic parallelizing compilers [10, 25, 29], HPF [30, 36] and the data parallel subset of Chapel [16] are canonical examples; other examples include MapReduce [18] and Spark [48] and task-based models such as Sequoia [23], Legion [6], StarPU[4], and PaRSEC [13].

In practice, programmers don't write highly scalable high performance codes in the implicitly parallel style of Figure 1a. Instead, they write the program in Figure 1b. Here the launching of a set of worker threads happens once, at program start, and the workers run until the end of the computation. We can see in Figures 1c and 1d that conceptually the correspondence between the programs is simple. Where Figure 1a launches N workers in the first loop and then N workers in the second loop, Figure 1b launches N long-lived threads that act as workers across iterations of the inner loops of the program. This *single program multiple data*, or SPMD, programming style is the basis of MPI [42], UPC [2] and Titanium [1], and also forms a useful subset of Chapel, X10 [17] and Legion.

While Figure 1a and Figure 1b are functionally equivalent, they have very different scalability and programmer productivity properties. Figure 1b is much more scalable, and not just by a constant factor. To see this, consider what happens in Figure 1a as the number of workers N (the "height" of the execution graph in Figure 1c) increases. Under weak scaling, the time to execute each worker task (e.g., `F(A[i])` in the first loop) remains constant, but the main control thread does $O(N)$ work to launch N workers. (In some cases, a broadcast tree can be used to reduce this overhead to $O(\log N)$.) Thus, for some N, the runtime overhead of launching workers exceeds the individual worker's execution time and the program ceases to scale. While the exact scalability in practice always depends on how long-running the parallel worker tasks are, our experience is that many implicitly parallel programs don't scale beyond 10 to 100 nodes when task granularities are on the order of milliseconds to tens of milliseconds. In contrast, the SPMD program in Figure 1b, while it still must launch N workers, does so only once and in particular the launch overhead is not incurred in every time step (the `T` loop). Programs written in SPMD style can scale to thousands or tens of thousands of nodes.

On the other hand, the implicitly parallel program in Figure 1a is much easier to write and maintain than the program in Figure 1b. While it is not possible to give precise measurements, it is clear that the difference in productivity is large: In our experience an implicitly parallel program that takes a day to write will require roughly a week to code in SPMD style. The extra programming cost is incurred because the individual workers in Figure 1b each compute only a piece of the first loop of Figure 1a, and thus explicit synchronization is required to ensure that all fragments of the first loop in all workers finish before dependent parts of the second loop begin. Furthermore, because the access patterns of the two loops in Figure 1a need not be the same, data movement is in general also needed to ensure that the values written by the various distributed pieces of the first loop are communicated to the threads that will read those values in the distributed pieces of the second loop. In most SPMD models (and specifically in MPI) this data movement must be explicitly written and optimized by the programmer. The synchronization and the data movement are by far the most difficult and time consuming parts of SPMD programs to get right, and these are exactly the parts that are not required in implicitly parallel programs.

This paper presents *control replication*, a technique for generating high-performance and scalable SPMD code from implicitly parallel programs with sequential semantics. The goal of control replication is to both "have our cake and eat it", to allow programmers to write in the productive implicitly parallel style and to use a combination of compiler optimizations and runtime analysis to automatically produce the scalable (but much more complex) SPMD equivalent.

Control replication works by generating a set of *shards*, or long-running tasks, from the control flow of the original program, to amortize overhead and enable efficient execution on large numbers of processors. Intuitively, a control thread of an implicitly parallel program is *replicated* across the shards, with each shard maintaining enough state to mimic the decisions of the original control thread. An important feature of control replication is that it is a local transformation, applying to a single collection of loops. Thus, it need not be applied only at the top level, and can in fact be applied independently to different parts of a program and at multiple different scales of nested parallelism.

As suggested above, the heart of the control replication transformation depends on the ability to analyze the implicitly parallel program with sufficient precision to generate the needed synchronization and data movement between shards. Similar analyses are known to be very difficult in traditional programming languages. Past approaches that have attempted optimizations with comparable goals to control replication have relied on either very sophisticated, complex and therefore unpredictable static analysis (e.g., HPF) or have relied much more heavily on dynamic analysis with associated run-time overheads (e.g., inspector-executor systems [35]).

A key aspect of our work is that we leverage recent advances in parallel programming model design that greatly simplify and make reliable and predictable the static analysis component of control replication. Many parallel programming models allow programmers to specify a *partition* of the data, to name different subsets of the data on which parallel computations will be carried out. Recent proposals allow programmers to define and use *multiple* partitions of the same data [6, 11]. For example, returning to our abstract example in Figure 1a, one loop may be accessing a matrix partitioned by columns while the other loop accesses the same matrix partitioned by rows. Control replication relies on the programmer to declare the data partitions of interest (e.g., rows and columns). The static analysis is carried out only at the granularity of the partitions and determines which partitions may share elements and therefore might require communication between shards. The dynamic analysis optimizes the communication at runtime by computing exactly which elements they share.

An important property of this approach is that the control replication transformation is guaranteed to succeed for any programmer-specified partitions of the data, even though the partitions can be

arbitrary. Partitions name program access patterns, and control replication reasons at the level of those coarser collections and their possible overlaps. This situation contrasts with the static analysis of programs where the access patterns must be inferred from individual memory references; current techniques, such as polyhedral analyses, work very well for affine index expressions [12], but do not address programs with more complex accesses.

This paper makes the following contributions:

- We describe the design and implementation of control replication in the context of the Regent programming language [41]. As noted above, the critical feature of Regent for control replication is support for multiple partitions; the technique should be applicable to any language with this feature.
- To the best of our knowledge, we are the first to demonstrate the impact of programming model support for multiple partitions on a compiler analysis and transformation. We show that this feature can be leveraged to provide both good productivity and scalability.
- We evaluate control replication using four codes: a circuit simulation on an unstructured graph, an explicit solver for the compressible Navier-Stokes equations on a 3D unstructured mesh, a Lagrangian hydrodynamics proxy application on a 2D unstructured mesh, and a stencil benchmark on a regular grid. Our implementation of control replication achieves up to 99% parallel efficiency on 1024 nodes (12288 cores) on the Piz Daint supercomputer [3] with absolute performance comparable to hand-written MPI(+X) codes.

In the following section, we describe the relevant features of Regent for control replication and give a motivating example. We then describe the transformation, and an evaluation of our implementation, before discussing related work and concluding.

```
1  task TF(B : region(SU, ...), A : region(SU, ...))
2  where reads writes(B), reads(A) do
3    for i in SU do
4      B[i] = F(A[i])
5    end
6  end
7
8  task TG(A : region(SU, ...), B : region(SQ, ...))
9  where reads writes(A), reads(B) do
10   for j in SU do
11     A[j] = G(B[h(j)])
12   end
13 end
14
15 -- Main Simulation:
16 var U = ispace(0..N)
17 var I = ispace(0..NT)
18 var A = region(U, ...)
19 var B = region(U, ...)
20 var PA = block(A, I)
21 var PB = block(B, I)
22 var QB = image(B, PB, h)
23 for t = 0, T do
24   for i in I do
25     TF(PB[i], PA[i])
26   end
27   for j in I do
28     TG(PA[j], QB[j])
29   end
30 end
```

Figure 2: Regent version of program with aliasing.

2 REGENT

Regent is a programming language with support for both implicit and explicit parallelism, making it possible to describe the control replication transformation entirely within one system. In particular, Regent's support for multiple partitions of data collections enables a particularly straightforward analysis of data movement required for efficient SPMD code generation. In this section, we discuss the relevant features of Regent and a number of preliminary steps to control replication.

2.1 Data and Execution Model

A central concern of the Regent programming language is the management and partitioning of data. Data in Regent is stored in *regions*. A region is a (structured or unstructured) collection of objects and may be *partitioned* into subregions that name subsets of the elements of the parent region.

Figure 2 shows a Regent version of the program in Figure 1a. The two inner loops with calls to point functions `F` and `G` have been extracted into tasks `TF` and `TG` on lines 1-6 and 8-13, respectively, and the new main simulation loop is preceded by an explicit partitioning of the data on lines 16-22.

Lines 18 and 19 declare two regions `A` and `B` that correspond to the arrays by the same name in the original program. These regions contains elements of some data type indexed from 0 to `N` − 1. (The element data type does not matter for the purposes of this paper.) The declaration of the *index space* `U` on line 16 gives a name to the set of indices for the regions; symbolic names for sets of indices are helpful because in general regions may be structured or unstructured, and optionally sparse. In Regent, memory allocation for regions is decoupled from their declaration. No actual memory allocation occurs at lines 18-19. Instead the program proceeds to partition the regions into subregions so that the eventual memory allocations are distributed across the machine.

Lines 20-22 contain calls to partitioning operators. The first two of these, on lines 20 and 21, declare block partitions of the regions `A` and `B` into roughly equal-sized subregions numbered 0 to `NT` − 1. (As before, a variable `I` is declared on line 17 to name this set of indices.) The variables `PA` and `PB` name the sets of subregions created in the respective partitioning operations. For convenience, we name the object which represents a set of subregions a *partition*.

Line 22 declares a second partition `QB` of the region `B` based on the image of the function `h` over `PB`. That is, for every index `b` in region `B`'s index space (`U`), `h(b)` ∈ `QB[i]` if `b` ∈ `PB[i]`. This partition describes exactly the set of elements that will be read inside the task `TG` on line 11. Importantly, there are no restrictions on the form or semantics of `h`. As a result, `QB` may not be a partition in the mathematical sense; i.e. the subregions of `QB` are not required to be disjoint, and the union of subregions need not cover the entire region `B`. In practice this formulation of partitioning is

extremely useful for naming the sets of elements involved in e.g. halo exchanges.

Regent supports a number of additional operators as part of an expressive sub-language for partitioning [44]. In the general case, Regent partitions are extremely flexible and may divide regions into subregions containing arbitrary subsets of elements. For the purposes of this paper, the only property of partitions that is necessary to analyze statically is the disjointness of partitions. A partition object is said to be *disjoint* if the subregions can be statically proven to be non-overlapping, otherwise the partition is *aliased*. For example, the block partition operators on lines 20-21 produce disjoint partitions as the subregions are always guaranteed to be non-overlapping. For the image operator on line 22, the function `h` is unconstrained and thus Regent assumes that the subregions may contain overlaps, causing the resulting partition to be considered aliased.

The main simulation loop on lines 23-30 then executes a sequence of task calls with the appropriate subregions as arguments. Tasks declare privileges on their region arguments (*read*, *write*, or *reduce* on an associative and commutative operator). Execution of tasks is apparently sequential: two tasks may execute in parallel only if they operate on disjoint regions, or with compatible privileges (e.g. both read, or both reduce with the same operator). Regent programs are typically written such that the inner loop can execute in parallel; in this case the loops on lines 24-26 and 27-29 both execute in parallel.

Note that in Regent, unlike in the fork-join parallel execution of Figure 1c, there is not an implicit global synchronization point at the end of each inner loop. Instead, Regent computes the dependencies directly between pairs of tasks (as described above) and thus tasks from different inner loops may execute in parallel if doing so preserves sequential semantics.

An important property of Regent tasks is that privileges are *strict*. That is, a task may only call another task if its own privileges are a superset of those required by the other task. Similarly, any reads or writes to elements of a region must conform to the privileges specified by the task. As a result, a compile-time analysis such as control replication need not consider the code inside of a task. All of the analysis for control replication will be at the level of tasks, privileges declared for tasks, region arguments to tasks, and the disjointness or aliasing of region arguments to tasks.

2.2 Target Programs

In this paper we consider programs containing forall-style loops of task calls such as those on lines 24-26 and 27-29 of Figure 2. Control replication is a local optimization and need not be applied to an entire program to be effective. The optimization is applied automatically to the largest set of statements that meet the requirements described below. In the example, control replication will be applied to lines 23-30 of the program.

Control replication applies to loops of task calls with no loop-carried dependencies except for those resulting from reductions to region arguments or scalar variables. Arbitrary control flow is permitted outside of these loops, as are statements over scalar variables.

No restrictions are placed on caller or callee tasks; control replication is fully composable with nested parallelism in the application.

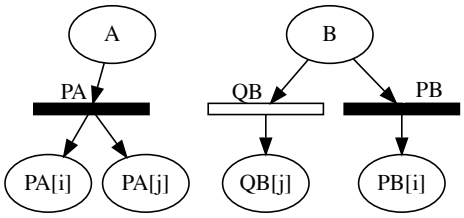


Figure 3: Region tree for the example. Filled boxes are disjoint partitions.

The compiler analysis for control replication need not be concerned with the contents of called tasks because the behavior of a task is soundly approximated by the privileges in the task's declaration (a property enforced by Regent's type system). Similarly, any caller task is completely agnostic to the application of control replication because any possible transformation of the code must be consistent with the task's privileges.

The region arguments of any called tasks must be of the form `p[f(i)]` where `p` is a partition, `i` is the loop index, and `f` is a pure function. Any accesses with a non-trivial function `f` are transformed into the form `q[i]` with a new partition `q` such that `q[i]` is `p[f(i)]`. Note here that we make essential use of Regent's ability to define multiple partitions of the same data.

2.3 Region Trees

The semantics of Regent enables a straightforward analysis of aliasing based on the relationships between regions and partitions. To determine whether two regions may alias, the compiler constructs a *region tree* that describes these relationships. This tree is a compile-time adaptation of the runtime data structure described in [6].

Figure 3 shows the region tree for the code in Figure 2. Note that regions in this formulation are *symbolic*, that is, the indices used to identify subregions are either constants or unevaluated loop variables. A dynamic evaluation of this program would result in an expansion of this tree for the various iterations of the loops (resulting in e.g. `PA[0]`, `PA[1]`, ..., `PA[NT-1]` under the `PA` partition). However, the number of iterations is not available at compile-time, making the symbolic version necessary.

The region tree is convenient because it provides a natural test to determine whether any two regions may alias: For any pair of regions R and S, find the least common ancestor A with immediate children R' and S' (along the path to R and S, respectively). If A is a disjoint partition and R' and S' are indexed by constants, then R and S are guaranteed to be disjoint regions at runtime; otherwise they may alias.

Region trees can be constructed by walking a task's body from top to bottom. Each newly created region becomes the root of a fresh region tree. Partitions are inserted under the region they partition, and expressions that access subregions of partitions result in the corresponding subregion nodes, tagged with the index expression used.

```
1 -- Initialization:
2 for i in I: PA[i] ← A
3 for i in I: PB[i] ← B
4 for i in I: QB[i] ← B
5
6 -- Transformed code:
7 for t = 0, T do
8   for i in I: TF(PB[i], PA[i])
9   for i, j in I × I: QB[j] ← PB[i]
10  for j in I: TG(PA[j], QB[j])
11 end
12
13 -- Finalization:
14 for i in I: A ← PA[i]
15 for i in I: B ← PB[i]
```

(a) Code after data replication.

```
1 -- Initialization:
2 for i in I: PA[i] ← A
3 for i in I: PB[i] ← B
4 for i in I: QB[i] ← B
5 var IQPB = {i, j | QB[j] ∩ PB[i] ≠ ∅}
6
7 -- Transformed code:
8 for t = 0, T do
9   for i in I: TF(PB[i], PA[i])
10  for i, j in IQPB: QB[j] ← PB[i]
11  for j in I: TG(PA[j], QB[j])
12 end
13
14 -- Finalization:
15 for i in I: A ← PA[i]
16 for i in I: B ← PB[i]
```

(b) Code with intersections.

```
1 -- Initialization:
2 for i in I: PA[i] ← A
3 for i in I: PB[i] ← B
4 for i in I: QB[i] ← B
5 var IQPB = {i, j | QB[j] ∩ PB[i] ≠ ∅}
6
7 -- Transformed code:
8 for t = 0, T do
9   for i in I: TF(PB[i], PA[i])
10  barrier()
11  for i, j in IQPB: QB[j] ← PB[i]
12  barrier()
13  for j in I: TG(PA[j], QB[j])
14 end
15
16 -- Finalization:
17 for i in I: A ← PA[i]
18 for i in I: B ← PB[i]
```

(c) Code with synchronization.

```
1 -- Shard task:
2 task shard(SI, SIQPB, PA, PB, QB)
3   for t = 0, T do
4     for i in SI: TF(PB[i], PA[i])
5     barrier()
6     for i, j in SIQPB: QB[j] ← PB[i]
7     barrier()
8     for j in SI: TG(PA[j], QB[j])
9   end
10 end
11 -- Initialization as before
12 -- Transformed code:
13 var X = ispace(0..NS)
14 var SI = block(I, X)
15 for x in X do
16   var SIQPB =
17     {k, j | k, j ∈ IQPB ∧ k ∈ SI[x]}
18   shard(SI[x], SIQPB, PA, PB, QB)
19 end
20 -- Finalization as before
```

(d) Code with shards.

Figure 4: Regent program at various stages of control replication.

3 CONTROL REPLICATION

In this section we describe the program transformations that comprise the control replication optimization. The optimization proceeds in phases, first inserting communication, then synchronization, and finally replicating the control flow to produce long-running shard tasks to reduce runtime overhead.

Consider a subregion S and its parent region P. Semantically, S is literally a subset of P: an update to an element of S also updates the corresponding element of P. There are two natural ways to implement this region semantics. In the *shared memory implementation* the memory allocated to S is simply the corresponding subset of the memory allocated to P. In the *distributed memory implementation*, S and P have distinct storage and the implementation must explicitly manage data coherence. For example, if a task writes to region S, then the implementation must copy S (or at least the elements that changed) to the corresponding memory locations of P so that subsequent tasks that use P see those updates; synchronization may also be needed to ensure these operations happen in the correct order. Intuitively, control replication begins with a shared memory program and converts it to an equivalent distributed memory implementation, with all copies and synchronization made explicit by the compiler.

3.1 Data Replication

The first stage of control replication is to rewrite the program so that every region and subregion has its own storage, inserting copies between regions where necessary for correctness. We use the shorthand $R_1 \leftarrow R_2$ for an assignment between two regions: R_1 is updated with the values of R_2 on the elements $R_1 \cap R_2$ they have in common. Figure 4a shows the core of the program in Figure 2 after three sets of copies have been inserted. Immediately before the code to which the optimization is applied (lines 7-11), the various partitions are initialized from the contents of the parent regions (lines 2-4). Symmetrically, any partitions written in the body of the transformed code must be copied back to their respective parent regions at the end (lines 14-15). Finally, inside the transformed code, writes to partitions must be copied to any aliased partitions that are also used within the transformed code. Here `PB` and `QB` are aliased (i.e. subregions of `PB` may overlap subregions of `QB`), so `PB` must be copied to `QB` on line 9 following the write to `PB` on line 8. Note that `PA` is also written (on line 10) but can be proven to be disjoint from `PB` and `QB` using the region tree analysis described in Section 2.3, thus no additional copies are required.

3.2 Copy Placement

The placement of the copies in Figure 4a happens to be optimal, but in general the algorithm described in Section 3.1 may introduce redundant copies and place those copies suboptimally. To improve copy placement, we employ variants of partial redundancy elimination and loop invariant code motion. The modifications required to the textbook descriptions of these optimizations are minimal. Loops such as lines 8-10 of Figure 4a are viewed as operations on partitions. For example, line 8 is seen as writing the partition `PB` and reading `PA` (summarizing the reads and writes to individual subregions). Note that the use of standard compiler techniques is only possible because of the problem formulation. In particular, aliasing between partitions is removed by the data replication transformation in Section 3.1, and program statements operate on partitions which hide the details of individual memory accesses.

3.3 Copy Intersection Optimization

Copies are issued between pairs of source and destination regions, but only the intersections of the regions must actually be copied. The number, size and extent of such intersections are unknown at

compile time; this is an aspect of the analysis that is deferred until runtime. For a large class of high-performance scientific applications, the number of such intersections per region is $O(1)$ in the size of the overall problem and thus for these codes an optimization to skip copies for empty intersections is able to reduce the complexity of the loop on Figure 4a line 9 from $O(N^2)$ to $O(N)$. Figure 4b shows the code following this optimization; note the changes on lines 5 and 10. For clarity of presentation the intersections on line 5 are written in pseudocode. In the implementation the compiler generates equivalent Regent code.

To avoid an $O(N^2)$ startup cost in comparing all pairs of subregions in the computation of intersections at line 5 in Figure 4b, we apply an additional optimization (not shown in the figure). The computation of intersections proceeds in two phases. First, we compute a shallow intersection to determine which pairs of regions overlap (but not the extent of the overlap). For unstructured regions, an interval tree acceleration data structure makes this operation $O(N \log N)$. For structured regions, we use a bounding volume hierarchy for this purpose. Second, we compute the exact set of overlapping elements between known-intersecting regions. Following the creation of shard tasks in Section 3.5 these operations are performed inside the individual shards, making them $O(M^2)$ where M is the number of non-empty intersections for regions owned by that shard.

Note that while shallow intersections are initially placed immediately prior to the transformed code, the placement may subsequently be altered by standard optimizations such as loop-invariant code motion. In the applications tested in Section 5, the shallow intersections were all lifted up to the beginning of the program execution.

In practice, at 1024 nodes, the impact of intersection computations on total running time is negligible, especially for long-running applications. Section 5.5 reports running times for the intersection operations of the evaluated applications.

3.4 Synchronization Insertion

When moving to multiple shards, it is necessary to synchronize on copies performed between shards. Shards are created in Section 3.5, and thus the inserted synchronization is initially redundant, but becomes necessary in the final transformed code.

A naive version of this synchronization is shown in Figure 4c. The copy operations on line 11 are issued by the producer of the data. Therefore, on the producer's side only, copies follow Regent's normal sequential semantics. Explicit synchronization is therefore only required for the consumer. Two barriers are used in Figure 4c on lines 10 and 12. The first barrier on line 10 preserves write-after-read dependencies and ensures that the copy does not start until all previous consumers of `QB` (i.e. `TG` tasks from the previous iteration of the outer loop) have completed. The second barrier on line 12 preserves read-after-write dependencies and ensures that subsequent consumers of `QB` (i.e. subsequent `TG` tasks) do not start until the copy has completed.

As an additional optimization (not shown), these barriers are replaced with point-to-point synchronization. In particular, the tasks which require synchronization are exactly those with non-empty intersections computed in Section 3.3, thus the sets of tasks that much synchronize are computed dynamically from the intersections above. The placement of synchronization operations in the transformed code is determined as follows. A simple dataflow analysis determines all consumers of `QB` preceding the copy on line 11 and all those following; these tasks synchronize with copies on line 11 as determined by the non-empty intersections computed in `IQPB`. This form of synchronization in Regent has the additional benefit that the point-to-point synchronization operators can be added as a direct precondition or postcondition to a task and therefore do not block the main thread of control as would a traditional barrier.

3.5 Creation of Shards

In the final stage of the transformation, control flow itself is replicated by creating a set of shard tasks that distribute the control flow of the original program. Figure 4d shows the code after the completion of the following steps.

First, the iterations of the inner loops for `TF` and `TG` must be divided among the shards. Note this division does not determine the mapping of a task to a processor for execution, which is discussed in Section 4.2. This simply determines ownership of tasks for the purposes of runtime analysis and control flow. The assignment is decided by a simple block partition of the iteration space into `NS` roughly even blocks on line 14. Second, the compiler transforms the loops so that the innermost loops are now over iterations owned by each shard, while the new outermost loop on line 15 iterates over shards.

Third, the compiler extracts the body of the shard into a new task on lines 2-10. This task is called from the main loop on line 18.

4 IMPLEMENTATION

We have implemented control replication as a compiler plug-in for Regent that adds control replication as an additional optimization pass to the main compiler. As with Regent itself, control replication is implemented in the Terra [20] and Lua [28] languages. This combination gives Regent (and our plugin) the ability to drive compilation with a high-level scripting language, while generating efficient low-level code via LLVM [32].

4.1 Runtime Support

Regent targets Legion, a C++ runtime system for hierarchical task parallelism [6]. In non-control replicated Regent programs, Legion discovers parallelism between tasks by computing a dynamic dependence graph over the tasks in an executing program. Control replication removes the need to analyze inter-shard parallelism, but Legion is still responsible for parallelism within a shard as well as any parallelism in the code outside of the scope of control replication.

A notable feature of Legion is its deferred execution model. All operations (tasks, copies, and even synchronization) execute asynchronously in the Legion runtime. This is an important requirement for supporting task parallelism, as it guarantees that the main thread of execution does not block and is subsequently able to expose as much parallelism as possible to the runtime system.

Legion targets Realm, a low-level runtime that supports execution on a wide variety of machines [43]. Realm uses GASNet [47] for active messages and data transfer.

4.2 Mapping Tasks to Processors

All tasks in Regent, including shard tasks, are processed through the Legion *mapping interface* [6]. This interface allows the user to define a *mapper* that controls the assignment of tasks to physical processors. (At the user's discretion, these decisions may be delegated to a library implementation. Legion provides a default mapper which provides sensible defaults for many applications.) A typical strategy is to assign one shard to each node, and then to distribute the tasks assigned to that shard among the processors of the node. However, substantially more sophisticated mapper implementations are also possible; in general mappers are permitted to be stateful and/or dynamic in their decision making. The techniques described in this paper are agnostic to the mapping used.

4.3 Region Reductions

Control replication permits loop-carried dependencies resulting from the application of associative and commutative reductions to region arguments of tasks. These reductions require special care in an implementation of control replication.

The partial results from the reductions must be stored separately to allow them to be folded into the destination region, even in the presence of aliasing. To accomplish this, the compiler generates a temporary region to be used as the target for the reduction and initializes the contents of the temporary to the identity value (e.g., 0 if the reduction operator is addition). The compiler then issues special *reduction copies* to apply the partial results to any destination regions which require the updates.

4.4 Scalar Reductions

In control replication, scalar variables are normally replicated as well. This ensures, for example, that control flow constructs behave identically on all shards in a SPMD-style program. Assignments to scalars are restricted to preserve this property; for example, scalars cannot be assigned within an innermost loop (as the iterations of this loop will be distributed across shards during control replication).

However, it can be useful to perform reductions on scalars, for example, to compute the next timestep in a code with dynamic time stepping. To accommodate this, control replication permits reductions to scalars within inner loops. Scalars are accumulated into local values that are then reduced across the machine with a Legion *dynamic collective*, an asynchronous collective operation that supports a dynamically determined number of participants. The result is then broadcast to all shards.

4.5 Hierarchical Region Trees

Regent permits recursive partitioning of regions. Among many other uses, this feature enables a common idiom in which the programmer constructs a top-level partition of a region into two subsets of elements: those which are guaranteed to never be involved in communication, and those which may need to be communicated. This design pattern, in combination with the region tree analysis described in Section 2.3, enables an important communication optimization that reduces data movement for distributed-memory execution, and also substantially reduces the cost of the dynamic computation of intersections described in Section 3.3.

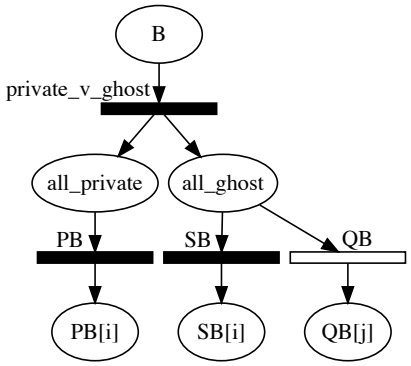


Figure 5: Region tree with hierarchical partitions.

Figure 5 shows a possible modification to the region tree from Figure 3 that uses this optimization. The top-level region `B` has been partitioned into two subregions that represent all the *private* elements (i.e. those never involved in communication) and *ghost* elements (i.e. those that are involved in communication). The new partition `SB` represents the subset of elements of the original `PB` partition involved in communication. Similarly, the new `PB` and `QB` partitions have been intersected with the regions `all_private` and `all_ghost`.

Notably, the top-level partition in this new region tree is disjoint, and thus by consulting the region tree the compiler is able to prove that the partition `PB` is disjoint from `QB` and `SB`. As a result, the compiler is able to prove that the subregions of `PB` are not involved in communication (as they are provably disjoint from all other subregions), and can avoid issuing copies for `PB`. Additionally, because `PB` has been excluded from the set of partitions involved in communication, the compiler is able to skip any intersection tests with `PB` and other partitions. As in most scalable applications the set of elements involved in communication is usually much smaller than those not involved in communication, so placing the private data in its own disjoint subregion can reduce the runtime cost of computing intersections.

5 EVALUATION

We evaluate performance and scalability of control replication in the context of Regent with four applications: a stencil benchmark on a regular grid; MiniAero, an explicit solver of the compressible Navier-Stokes equations on a 3D unstructured mesh; PENNANT, a Lagrangian hydrodynamics simulation on a 2D unstructured mesh; and a circuit simulation on a sparse unstructured graph. For each application we consider a Regent implementation with and without control replication and when available a reference implementation written in MPI or a flavor of MPI+X.

For each application, we report weak scaling performance on up to 1024 nodes of the Piz Daint supercomputer [3], a Cray XC50 system. Each node has an Intel Xeon E5-2690 v3 CPU (with 12 physical cores) and 64 GB of memory. Legion was compiled with GCC 5.3.0. The reference codes were compiled with the Intel C/C++ compiler 17.0.1. Regent used LLVM for code generation: version 3.8.1 for Stencil and PENNANT and 3.6.2 for MiniAero and Circuit.

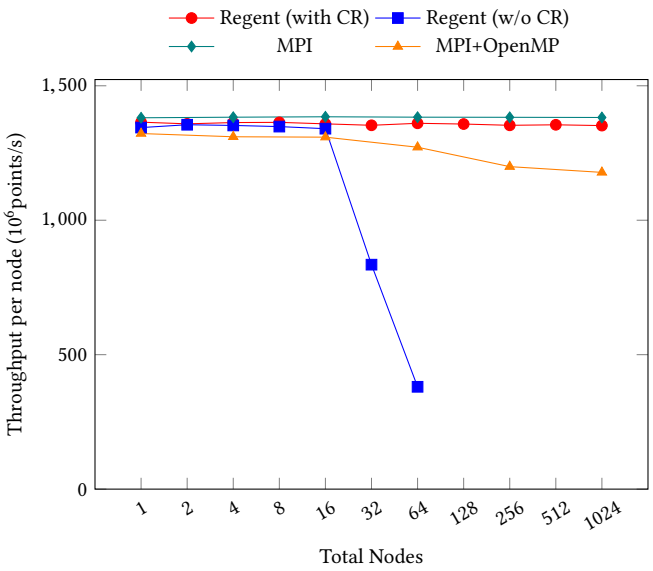


Figure 6: Weak scaling for Stencil.

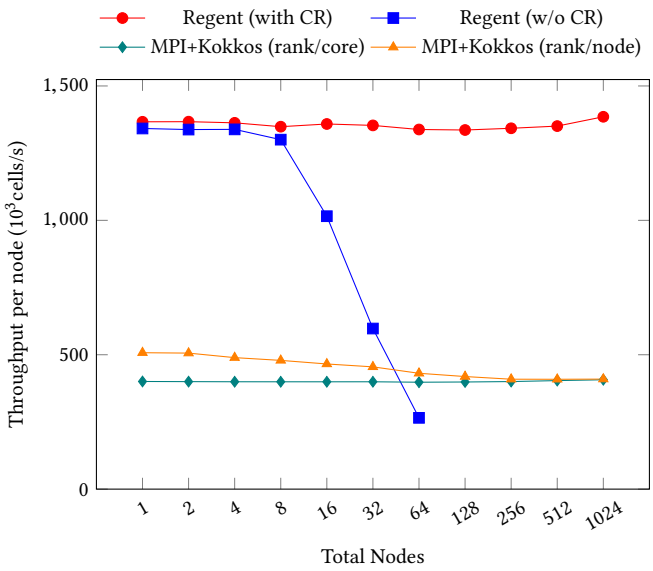


Figure 7: Weak scaling for MiniAero.

Finally, we report the running times of the dynamic region intersections for each of the applications at 64 and 1024 nodes.

5.1 Stencil

Stencil is a 2D structured benchmark from the Parallel Research Kernels (PRK) [45, 46]. The code performs a stencil of configurable shape and radius over a regular grid. Our experiments used a radius-2 star-shaped stencil on a grid of double-precision floating point values with $40k^2$ grid points per node. We compared against the MPI and MPI+OpenMP reference codes provided by PRK. Both reference codes require square inputs and thus were run only at node counts that were even powers of two.

As noted in Section 2.2, all analysis for control replication was performed at the task and region level. Control replication was able to optimize code containing affine access patterns, without requiring any specific support for affine reasoning in the compiler.

Figure 6 shows weak scaling performance for Stencil up to 1024 nodes. (In the legend control replication is abbreviated as CR.) Control replication achieved 99% parallel efficiency at 1024 nodes, whereas Regent without control replication rapidly drops in efficiency when the overhead of launching an increasing number of subtasks begins to dominate the time to execute those subtasks, as discussed in Section 1.

5.2 MiniAero

MiniAero is a 3D unstructured mesh proxy application from the Mantevo suite [26] developed at Sandia National Laboratories, implementing an explicit solver for the compressible Navier-Stokes equations. The mini-app is written in a hybrid style, using MPI for inter-node communication and Kokkos for intra-node parallelism. (Kokkos is a portability layer for C++ that compiles down to pthreads (on CPUs), and is also developed at Sandia [22].) We ran the reference in two configurations: one MPI rank per core, and one MPI rank per node (using Kokkos support for intra-node parallelism).

Figure 7 shows weak scaling absolute performance for the various implementations of MiniAero on a problem size of 512k cells per node. As demonstrated in [41], Regent-based codes out-perform the reference MPI+Kokkos implementations of MiniAero on a single node, mostly by leveraging the improved hybrid data layout features of Legion [7].

Control replication achieves slightly over 100% parallel efficiency at 1024 nodes due to variability in the performance of individual nodes; as before, Regent without control replication struggles to scale beyond a modest number of nodes. Although the rank per node configuration of the MPI+Kokkos reference provides initial benefits to single-node performance, performance eventually drops to the level of the rank per core configuration.

5.3 PENNANT

PENNANT is a 2D unstructured mesh proxy application from Los Alamos National Laboratory simulating Lagrangian hydrodynamics [24]. The application represents a subset of the functionality that exists in FLAG, a larger production code used at the lab [14].

The reference PENNANT implementation applies a cache blocking optimization that substantially improves the computational intensity of the overall application. This optimization impacts even the data structure layouts, as the (otherwise unordered) mesh elements are grouped into chunks to be processed together. In spite of this, control replication applied seamlessly to the code, as the details of the cache blocking optimization are limited to the structure of the region tree (which subsumes the chunk structure of the original code) and the bodies of tasks (whose details are accurately summarized by the privileges declared in the task declaration).

Figure 8 shows weak scaling performance for PENNANT on up to 1024 nodes, using a problem size of 7.4M zones per node. The

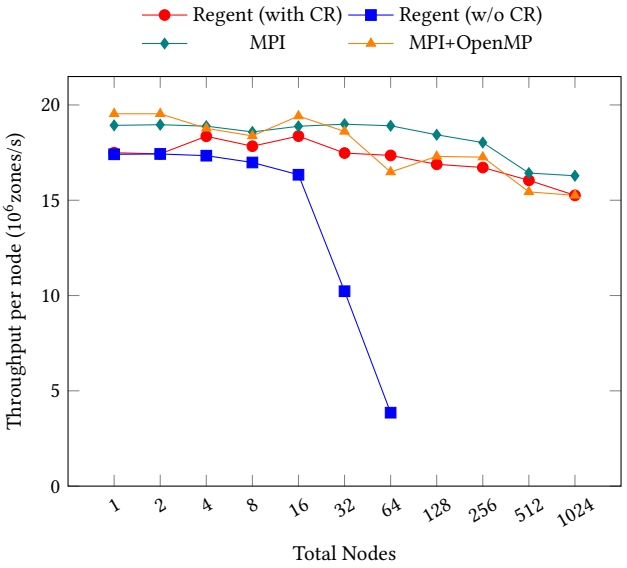


Figure 8: Weak scaling for PENNANT.

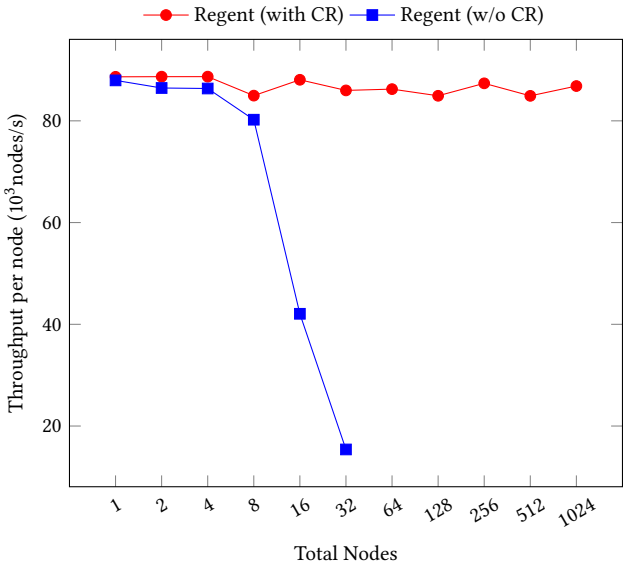


Figure 9: Weak scaling for Circuit.

single-node performance of the Regent implementation is less than the reference because the underlying Legion runtime requires a core be dedicated to analysis of tasks. This effect is noticeable on PENNANT because, due to the cache blocking optimization above, PENNANT is mostly compute-bound.

However, this performance gap closes at larger node counts as Regent is better able to achieve asynchronous execution to hide the latency of the global scalar reduction to compute the *dt* in the next timestep of the application. At 1024 nodes, control replication achieves 87% parallel efficiency, compared to 82% for MPI and 64% for MPI+OpenMP.

5.4 Circuit

We developed a sparse circuit simulation based on [6] to measure weak scaling performance on unstructured graphs. The implicitly parallel version from [6] was already shown to be substantially communication bound at 32 nodes and would not have scaled to significantly more nodes, regardless of the implementation technique. The input for this problem was a randomly generated sparse graph with 100k edges and 25k vertices per compute node; the application was otherwise identical to the original.

Figure 9 shows weak scaling performance for the simulation up to 1024 nodes. Regent with control replication achieves 98% parallel efficiency at 1024 nodes. As with the other applications, Regent without control replication matches this performance at small node counts (in this case up to 16 nodes) but then efficiency begins to drop rapidly as the overhead of having a single master task launching many subtasks becomes dominant.

5.5 Dynamic Intersections

As described in Section 3.3, dynamic region intersections are computed prior to launching a set of shard tasks in order to identify the communication patterns and precise data movement required for control-replicated execution. Table 1 reports the running times of the intersection operations measured during the above experiments while running on 64 and 1024 nodes. Shallow intersections are performed on a single node to determine the approximate communication pattern (but not the precise sets of elements that require communication); these required at most 259 ms at 1024 nodes (15 ms at 64 nodes). Complete intersections are then performed in parallel on each node to determine the precise sets of elements that must be communicated with other nodes; these took at most 124 ms. Both times are much less than the typical running times of the applications themselves, which are often minutes to hours.

Application	Nodes	Shallow (ms)	Complete (ms)
Circuit	64	7.8	2.7
	1024	143	4.7
MiniAero	64	15	17
	1024	259	43
PENNANT	64	6.8	14
	1024	125	124
Stencil	64	2.7	0.4
	1024	78	1.3

Table 1: Running times for region intersections on each application at 64 and 1024 nodes.

6 RELATED WORK

Several approaches have been considered for compiling sequential, shared-memory parallel, and/or data-parallel programs for execution on distributed-memory machines.

Inspector/executor (I/E) methods have been used to compile a class of sequential programs with affine loops and irregular accesses for distributed memory [34, 35]. As in control replication, a necessary condition for I/E methods is that the memory access

patterns are fixed within the loop, so that the inspector need only be run once. Use of an inspector allows the read/write sets of program statements to be determined dynamically when the necessary static analysis is infeasible in the underlying programming language, enabling distributed, parallel execution of codes written in conventional languages. This approach has been demonstrated to scale to 256 cores. However, the time and space requirements of the inspector limit scalability at very large node counts. Also, the I/E approach relies on generic partitioning algorithms such as automatic graph partitioning [15, 39].

Kwon et al. describe a technique for compiling OpenMP programs with regular accesses to MPI code [31]. A hybrid static/dynamic analysis is used to determine the set of elements accessed by each parallel loop. For efficiency, the dynamic analysis maintains a bounded list of rectangular section fragments at communication points. As a result, non-affine accesses cause analysis imprecision that results in replicated data, increased communication, and limited scalability. The approach has been demonstrated to scale to 64 cores.

Like the two approaches above, control replication uses a combined static/dynamic analysis to obtain precise information about access patterns. At a high level, the key difference is that control replication leverages a programming model with explicit support for coarse-grain operations (tasks), data partitioning (of regions into subregions), and the simultaneous use of multiple partitions of the same data. Control replication performs analysis at this coarsened level rather than at the level of individual loop iterations, resulting in a more efficient dynamic analysis and in-memory representation of the access patterns of each loop without any loss of precision. Furthermore, hierarchically nested partitions enable control replication to skip analysis at runtime for data elements not involved in communication (further reducing memory usage for the analysis). Finally, explicit language support for partitioning allows control replication to leverage application-specific partitioning algorithms, which are often more efficient and yield better results than generic algorithms. As a result, control replication is able to support more complex access patterns more efficiently, resulting in better scalability.

A number of efforts to support OpenMP on distributed-memory machines target software distributed shared-memory (DSM) systems [5, 27, 38]. These approaches have limited scalability due to the limitations of page-based DSM systems.

Efforts in data-parallel languages such as High Performance Fortran (HPF) [30, 36] pioneered compilation techniques for a variety of machines, including distributed-memory. In HPF, a single (conceptual) thread of control creates implicit data-parallelism by specifying operations over entire arrays in a manner similar to traditional Fortran. This data parallelism is then mapped to a distributed-memory system via explicit user-specified data distributions of the arrays—though the compiler is still responsible for inferring *shadow regions* (i.e. halos that must be communicated) from array accesses. Several implementations of HPF achieve good scalability on structured applications [37, 40].

ZPL, an implicitly parallel array language, provides a very general array remap operator [19] that permits the redistribution of data via (conceptually) scatters and gathers. The ZPL compiler optimizes cases of the remap operator that commonly appear in structured programs to avoid paying the cost of the general case. However, when these optimizations fail, ZPL falls back to an inspector/executor approach similar to the one described above, with similar tradeoffs.

The Chapel [16] language supports a variety of styles of parallelism, including implicit data parallelism and explicit PGAS-style parallelism. This *multiresolution* design reduces the burden placed on the compiler to optimize Chapel's data parallel subset because users can incrementally switch to other forms of parallelism as needed. However, use of Chapel's explicitly parallel features expose users to the hazards of traditional explicitly parallel programming.

Compared to Regent, Chapel's data parallel subset (which is most similar to Regent's implicit parallelism) only supports a single distribution of elements for the lifetime of a given array, and limited task parallelism. Regent's support for multiple and hierarchical partitions are critical to enabling control replication to optimize implicitly parallel programs for efficient execution on distributed memory machines.

The multiresolution approach is also taken in Legion and Regent, which support both a straightforward implicitly parallel style of programming that scales to modest numbers of nodes as well as more involved explicit communication constructs that can be used to scale to very large node counts [7]. In particular, the explicit approach can be time-consuming and error-prone, and was identified in a recent study [8] as a challenge for this class of programming systems. Our work can be seen as greatly increasing the performance range of the implicit style, allowing more whole codes and subsystems of codes to be written in this more productive and more easily maintained style.

MapReduce [18] and Spark [48] support implicitly parallel execution of pure functions composed via data-parallel operators such as *map* and *reduce*. Both MapReduce and Spark use a centralized scheduler that becomes a bottleneck at large node counts with tasks on the order of milliseconds to tens of milliseconds. Originally these systems were intended for use in I/O-intensive applications where the input is initially read from disk, such that the overhead of centralized scheduling was not a concern. More recently, improvements in the efficiency of these codes have resulting in more fine-grained tasks. The use of execution templates to reduce control overhead [33] has been explored as a way to improve the scalability of a centralized scheduler.

Liszt [21] is an implicitly parallel domain-specific language for solving partial differential equations on unstructured meshes. By leveraging domain-specific knowledge, the Liszt compiler is able to generate optimized distributed code that scales to large numbers of nodes. However, the domain-specific reasoning used in the compiler limits the applicability of the approach. In contrast, control replication is general-purpose and not limited to a specific domain.

X10 [17] is an explicitly parallel programming language with places and hierarchical tasks. Flat X10 [9] is a subset of this language that restricts programs to a two-level task hierarchy where the top level consists of forall-style parallel loops. A compiler for Flat X10 is able to transform the program into a SPMD-style X10 program with explicit synchronization between tasks. However, as the original Flat X10 program already contains explicit communication, the compiler need not make changes to the structure of communication in the program. In contrast, control replication is

able to automatically generate efficient explicit communication for an implicitly parallel program with implicit data movement.

7 CONCLUSION

Control replication automatically generates efficient SPMD code for implicitly parallel programs by leveraging language support for multiple partitions of data. We have presented an implementation of control replication for Regent and evaluated it on three unstructured proxy applications and a structured benchmark. In our experiments, control replication achieves up to 99% parallel efficiency on 1024 nodes.

ACKNOWLEDGMENT

This work was supported by the Army High Performance Computing Research Center, by the Department of Energy National Nuclear Security Administration under Award Number DE-NA0002373-1, by the Advanced Simulation and Computing Program, Advanced Technology Development and Mitigation element administered by Thuc Hoang under contract DE-AC52-06NA25396 with Los Alamos National Laboratory, by the Department of Energy Office of Science, Office of Advanced Scientific Computing Research under the guidance of Dr. Lucy Nowell, by an internship at NVIDIA Research, and by a grant from the Swiss National Supercomputing Centre (CSCS) under project ID d51.

The authors would like to thank Albert Sidelnik and Norm Rubin for their mentorship and support, Charles R. Ferenbaugh for his advice and assistance with the PENNANT code, and Janine C. Bennett, Greg Sjaardema, Kenneth Franko, Hemanth Kolla, Jeremiah Wilke, David Hollman, and the Mantevo project [26] for support with the MiniAero code.

REFERENCES

[1] 2006. Titanium Language Reference Manual. http://titanium.cs.berkeley.edu/doc/lang-ref.pdf. (2006).

[2] 2013. UPC Language Specifications, Version 1.3. http://upc.lbl.gov/publications/upc-spec-1.3.pdf. (2013).

[3] 2016. Piz Daint & Piz Dora - CSCS. http://www.cscs.ch/computers/piz_daint. (2016).

[4] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2011. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience* 23 (Feb. 2011), 187–198. Issue 2.

[5] Ayon Basumallik, Seung-Jai Min, and Rudolf Eigenmann. 2007. Programming Distributed Memory Sytems Using OpenMP. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*. IEEE, 1–8.

[6] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: Expressing Locality and Independence with Logical Regions. In *Supercomputing (SC)*.

[7] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2014. Structure Slicing: Extending Logical Regions with Fields. In *Supercomputing (SC)*.

[8] Janine Bennett, Robert Clay, Gavin Baker, Marc Gamell, David Hollman, Samuel Knight, Hemanth Kolla, Gregory Sjaardema, Nicole Slattengren, Keita Teranishi, Jeremiah Wilke, Matt Bettencourt, Steve Bova, Ken Franko, Paul Lin, Ryan Grant, Si Hammond, Stephen Olivier, Laxmikant Kale, Nikhil Jain, Eric Mikida, Alex Aiken, Mike Bauer, Wonchan Lee, Elliott Slaughter, Sean Treichler, Martin Berzins, Todd Harman, Alan Humphrey, John Schmidt, Dan Sunderland, Pat McCormick, Samuel Gutierrez, Martin Schulz, Abhinav Bhatele, David Boehme, Peer-Timo Bremer, and Todd Gamblin. 2015. ASC ATDM Level 2 Milestone #5325: Asynchronous Many-Task Runtime System Analysis and Assessment for Next Generation Platforms. SAND2015-8312 (2015).

[9] Ganesh Bikshandi, Jose G. Castanos, Sreedhar B. Kodali, V. Krishna Nandivada, Igor Peshansky, Vijay A. Saraswat, Sayantan Sur, Pradeep Varma, and Tong Wen. 2009. Efficient, Portable Implementation of Asynchronous Multi-place Programs. In *PPoPP*. ACM, 271–282.

[10] William Blume, Rudolf Eigenmann, Keith Faigin, John Grout, Jay Hoeflinger, David Padua, Paul Petersen, William Pottenger, Lawrence Rauchwerger, Peng Tu, and Stephen Weatherford. 1995. Effective Automatic Parallelization with Polaris. In *International Journal of Parallel Programming*. Citeseer.

[11] Robert L. Bocchino Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. 2009. A Type and Effect System for Deterministic Parallel Java. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

[12] Uday Bondhugula. 2013. Compiling Affine Loop Nests for Distributed-Memory Parallel Architectures. In *Supercomputing (SC)*. ACM, 33.

[13] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, and Jack J. Dongarra. 2013. PaRSEC: Exploiting Heterogeneity to Enhance Scalability. *Computing in Science & Engineering* 15, 6 (2013), 36–45.

[14] Donald E. Burton. 1994. *Consistent Finite-Volume Discretization of Hydrodynamics Conservation Laws for Unstructured Grids*. Technical Report UCRL-JC-118788. Lawrence Livermore National Laboratory, Livermore, CA.

[15] Ümit Çatalyürek and Cevdet Aykanat. 2011. PaToH (Partitioning Tool for Hypergraphs). In *Encyclopedia of Parallel Computing*. Springer, 1479–1487.

[16] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. 2007. Parallel Programmability and the Chapel Language. *Int'l Journal of HPC Apps*. (2007).

[17] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph Von Praun, and Vivek Sarkar. 2005. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 519–538.

[18] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *Opearting Systems Design & Implementation (OSDI)*. 10–10.

[19] Steven J. Deitz, Bradford L. Chamberlain, Sung-Eun Choi, and Lawrence Snyder. 2003. The Design and Implementation of a Parallel Array Operator for the Arbitrary Remapping of Data. In *PPoPP*, Vol. 38. ACM, 155–166.

[20] Zachary DeVito, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan Vitek. 2013. Terra: A Multi-Stage Language for High-Performance Computing *(PLDI)*.

[21] Zachary DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, Eric Darve, Juan Alonso, and Pat Hanrahan. 2011. Liszt: A Domain Specific Language for Building Portable Mesh-based PDE Solvers. In *Supercomputing (SC)*.

[22] H. Carter Edwards and Christian R. Trott. 2013. Kokkos: Enabling Performance Portability Across Manycore Architectures. In *Extreme Scaling Workshop (XSW), 2013*. 18–24.

[23] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. 2006. Sequoia: Programming the Memory Hierarchy. In *Supercomputing*.

[24] Charles R. Ferenbaugh. 2014. PENNANT: an unstructured mesh mini-app for advanced architecture research. *Concurrency and Computation: Practice and Experience* (2014).

[25] Mary W. Hall, Jennifer-Ann M. Anderson, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, Edouard Bugnion, and Monica S. Lam. 1996. Maximizing Multiprocessor Performance with the SUIF Compiler. *IEEE Computer* 29, 12 (1996), 84–89.

[26] Michael A. Heroux, Douglas W. Doerfler, Paul S. Crozier, James M. Willenbring, H. Carter Edwards, Alan Williams, Mahesh Rajan, Eric R. Keiter, Heidi K. Thornquist, and Robert W. Numrich. 2009. *Improving Performance via Mini-applications*. Technical Report SAND2009-5574. Sandia National Laboratories.

[27] Jay P. Hoeflinger. 2006. Extending OpenMP to Clusters. *White Paper, Intel Corporation* (2006).

[28] Roberto Ierusalimschy, Luiz Henrique De Figueiredo, and Waldemar Celes Filho. 1996. Lua – An Extensible Extension Language. *Software: Practice & Experience* (1996).

[29] François Irigoin, Pierre Jouvelot, and Rémi Triolet. 1991. Semantical Interprocedural Parallelization: An Overview of the PIPS Project. In *Supercomputing (SC)*.

[30] Ken Kennedy, Charles Koelbel, and Hans Zima. 2007. The Rise and Fall of High Performance Fortran: An Historical Object Lesson. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. ACM, 7–1.

[31] Okwan Kwon, Fahed Jubair, Rudolf Eigenmann, and Samuel Midkiff. 2012. A Hybrid Approach of OpenMP for Clusters *(PPoPP)*. ACM, 75–84.

[32] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. Palo Alto, California.

[33] Omid Mashayekhi, Hang Qu, Chinmayee Shah, and Philip Levis. 2017. Execution Templates: Caching Control Plane Decisions for Strong Scaling of Data Analytics. In *USENIX Annual Technical Conference (USENIX ATC)*.

[34] Mahesh Ravishankar, Roshan Dathathri, Venmugil Elango, Louis-Noël Pouchet, J. Ramanujam, Atanas Rountev, and P. Sadayappan. 2015. Distributed Memory Code Generation for Mixed Irregular/Regular Computations *(PPoPP)*. ACM, 65–75.

[35] Mahesh Ravishankar, John Eisenlohr, Louis-Noël Pouchet, J. Ramanujam, Atanas Rountev, and P. Sadayappan. 2012. Code Generation for Parallel Execution of a Class of Irregular Loops on Distributed Memory Systems. In *Supercomputing (SC)*.

[36] Harvey Richardson. 1996. High Performance Fortran: History, Overview and Current Developments. *Thinking Machines Corporation* 14 (1996), 17.

[37] Hitoshi Sakagami, Hitoshi Murai, Yoshiki Seo, and Mitsuo Yokokawa. 2002. 14.9 TFLOPS Three-Dimensional Fluid Simulation for Fusion Science with HPF on the Earth Simulator. In *Supercomputing, ACM/IEEE 2002 Conference*. IEEE, 51–51.

[38] Mitsuhisa Sato, Hiroshi Harada, Atsushi Hasegawa, and Yutaka Ishikawa. 2001. Cluster-enabled OpenMP: An OpenMP Compiler for the SCASH Software Distributed Shared Memory System. *Scientific Programming* 9, 2, 3 (2001), 123–130.

[39] Kirk Schloegel, George Karypis, and Vipin Kumar. 2002. Parallel Static and Dynamic Multi-Constraint Graph Partitioning. *Concurrency and Computation: Practice and Experience* 14, 3 (2002), 219–240.

[40] Yoshiki Seo, Hidetoshi Iwashita, Hiroshi Ohta, and Hitoshi Sakagami. 2002. HPF/JA: Extensions of High Performance Fortran for Accelerating Real-World Applications. *Concurrency and Computation: Practice and Experience* 14, 8-9 (2002), 555–573.

[41] Elliott Slaughter, Wonchan Lee, Sean Treichler, Michael Bauer, and Alex Aiken. 2015. Regent: A High-Productivity Programming Language for HPC with Logical Regions. In *Supercomputing (SC)*.

[42] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. 1998. *MPI-The Complete Reference*. MIT Press.

[43] Sean Treichler, Michael Bauer, and Alex Aiken. 2014. Realm: An Event-Based Low-Level Runtime for Distributed Memory Architectures. In *Parallel Architectures and Compilation Techniques (PACT)*.

[44] Sean Treichler, Michael Bauer, Rahul Sharma, Elliott Slaughter, and Alex Aiken. 2016. Dependent Partitioning. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 344–358.

[45] Rob F. Van der Wijngaart, Abdullah Kayi, Jeff R. Hammond, Gabriele Jost, Tom St. John, Srinivas Sridharan, Timothy G. Mattson, John Abercrombie, and Jacob Nelson. 2016. Comparing Runtime Systems with Exascale Ambitions Using the Parallel Research Kernels. In *International Conference on High Performance Computing*. Springer, 321–339.

[46] Rob F. Van der Wijngaart and Timothy G. Mattson. 2014. The Parallel Research Kernels. In *HPEC*. 1–6.

[47] Katherine Yelick, Dan Bonachea, Wei-Yu Chen, Phillip Colella, Kaushik Datta, Jason Duell, Susan L. Graham, Paul Hargrove, Paul Hilfinger, Parry Husbands, Costin Iancu, Amir Kamil, Rajesh Nishtala, Jimmy Su, Michael Welcome, and Tong Wen. 2007. Productivity and Performance Using Partitioned Global Address Space Languages. In *PASCO*. 24–32.

[48] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. *HotCloud* 10 (2010), 10–10.