Triolet: A Programming System that Unifies Algorithmic Skeleton Interfaces for High-Performance Cluster Computing

Christopher Rodrigues Thomas Jablin Abdul Dakkak Wen-Mei Hwu

University of Illinois at Urbana-Champaign

{cirodrig, jablin, dakkak, w-hwu}@illinois.edu

Abstract

Functional algorithmic skeletons promise a high-level programming interface for distributed-memory clusters that free developers from concerns of task decomposition, scheduling, and communication. Unfortunately, prior distributed functional skeleton frameworks do not deliver performance comparable to that achievable in a low-level distributed programming model such as C with MPI and OpenMP, even when used in concert with high-performance array libraries. There are several causes: they do not take advantage of shared memory on each cluster node; they impose a fixed partitioning strategy on input data; and they have limited ability to fuse loops involving skeletons that produce a variable number of outputs per input.

We address these shortcomings in the Triolet programming language through a modular library design that separates concerns of parallelism, loop nesting, and data partitioning. We show how Triolet substantially improves the parallel performance of algorithms involving array traversals and nested, variable-size loops over what is achievable in Eden, a distributed variant of Haskell. We further demonstrate how Triolet can substantially simplify parallel programming relative to C with MPI and OpenMP while achieving 23–100% of its performance on a 128-core cluster.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) languages, Concurrent, distributed and parallel languages; D.3.4 [*Programming Languages*]: Processors—Optimization

Keywords Algorithmic skeletons; Loop fusion; Parallel programming

PPoPP '14, February 15–19, 2014, Orlando, Florida, USA.

ACM 978-1-4503-2656-8/14/02. . . $15.00.

http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/2555243.2555268

1. Introduction

Clusters of multicore computers are an inexpensive way to furnish parallel computing capacity. Data-parallel algorithmic skeletons—higher-order functions capturing common patterns of parallel computation—have been proposed as a simple, high-level interface for multicore [6, 12, 16, 19] and distributed [11, 15, 22] parallel programming. Skeletons decompose work into parallel tasks and distribute that work across a parallel machine, allowing a programmer to focus on what parallelization strategy to use from a skeleton library, rather than how it is implemented. Additionally, distributed functional skeletons send data and collect results from tasks, allowing programmers to focus on what data to use rather than how to move it between cluster nodes. Separating out the low-level details of parallelism allows developers to write parallel code that resembles sequential code, wherein the choice of parallel skeletons determines the details of parallel execution. However, while prior distributed functional skeleton frameworks have demonstrated their ease of use, they do not approach lower-level parallel programming models in absolute performance [14, 21].

For example, the molecular modeling application `cutcp` has a computationally demanding loop nest computing a floating-point histogram. It loops over a collection of atoms, visits all grid points near the atom, and updates the grid point with the electric potential induced by the atom. In idiomatic sequential Haskell, this could be written using a list comprehension as shown below, assuming suitable definitions of `floatHist`, `f`, and `gridPts`.

```
floatHist [f a r | a <- atoms, r <- gridPts a]
```

In the list comprehension, the generator `a <- atoms` loops over a list of atoms, the generator `r <- gridPts a` computes and loops over the grid points near each atom, and the call `f a r` computes the electrostatic potential induced by an atom at a grid point. The list of results is collected into a histogram by `floatHist`. A naïve attempt at parallelization might replace `floatHist` and the traversal of `atoms` by a distributed implementation written in Eden, a distributed extension of Haskell [15]. One could write a distributed `floatHist` function that partitions an input list across tasks

and performs histogramming within each task and use it as shown below.

```
floatHistD (\x -> [f r x | r <- gridPts x]) atoms
```

This code demonstrates the attractive simplicity of algorithmic skeletons, but its per-thread performance is an order of magnitude lower than sequential C chiefly due to the overhead of list manipulation, indicating that there is substantial room for improvement.

A performance-oriented high-level programmer could optimize the parallelized code by writing custom skeletons that minimize network communication and implementing each task with imperative code manipulating unboxed arrays. Since the combination of Eden and high-performance array libraries has not been studied before, we evaluate this high-performance style as a reference point. These optimizations can indeed yield sequential performance within a small multiplicative factor of C. However, this kind of manual transformation is exactly what skeletons should make unnecessary. Moreover, scalability remains limited because Eden does not exploit sharing on each cluster node.

We identify and address three problems that cause distributed functional skeletons to fail to achieve high performance. First, irregular loops, where each element of an input data structure yields a dynamically determined number of outputs, cannot be fused and parallelized using prior loop representations. As in the example above, it is common to write a parallel loop as a multi-stage process of generating, then consuming a collection of temporary values. We use a novel representation of fusible, nested loops to isolate irregularity within inner loops, allowing irregular generation phases to be fused with consumers into a single parallel loop. Unlike prior methods, our representation balances support for random access to collections, needed to efficiently parallelize loops and implement index-based operations such as `zip`, with support for cheap opportunistic partitioning of collections, needed to fuse and parallelize irregular loops.

Second, we treat data distribution strategies separately from work distribution strategies, enabling a parallel skeletons to use a data distribution mechanism appropriate for their inputs. Prior data-parallel distributed skeleton frameworks performed both data and work distribution by partitioning a list or array of inputs across processors. For some algorithms, this may be inefficient due to the overhead of processing linked lists, because tasks can be implemented more efficiently on chunks of data than on individual elements, and/or because some input data are unnecessarily replicated for use in multiple loop iterations. Separating data distribution reduces the programmer effort and run-time overhead of reorganizing data before a parallel loop.

Third, skeletons need to be able to employ different algorithm implementations at the sequential, shared-memory, and distributed scales, taking advantage of in-place data structure updates at the sequential level and avoiding copying at the shared-memory level. Eden's scalability, in particular, is limited by its inability to take advantage of shared memory to reduce communication cost. We utilize a two-level parallel architecture, with message passing across nodes and work-stealing thread parallelism in each node. Our parallel skeletons follow this two-level architecture, exploiting shared memory and in-place updates.

We have implemented these solutions in the library, compiler, and runtime of the Triolet programming language. We evaluate the performance and scalability of four programs written in Triolet, Eden, and C+MPI+OpenMP on a 128-core (8 nodes $\times$ 16 cores per node) cluster. Each of these benchmarks presents challenges that the Eden code must work around, whereas Triolet's library and runtime allow a cleaner expression of parallelism to achieve high performance. The C+MPI+OpenMP implementations, with their highly efficient low-level communication and synchronization, provide a reference point for estimating the overhead in Triolet's algorithms and runtime system. Triolet consistently yields higher parallel performance than Eden, achieves 23–100% of the performance of C+MPI+OpenMP versions, and yields a speedup up to 9.6–$99\times$ relative to simple loops in sequential C.

2. Overview

In typical data-parallel algorithmic skeleton libraries, each skeleton is a higher-order function containing a carefully implemented pattern of parallel communication and work distribution. Invoking a skeleton effectively instantiates and inserts code into a parallel loop. To conform their algorithm to an available skeleton, users of a library often have to regularize their algorithms and reorganize their data structures. These adaptations add complexity and overhead to parallel programs.

Triolet's skeleton library has a more modular design that builds parallel loops from loosely coupled components managing data movement, work distribution, result collection, and the composition of loops. Support for composition of nested loops accommodates irregular and nested parallel loops. The data movement component accommodates different looping patterns. The work distribution component supports block-based decomposition of multidimensional arrays for memory locality. By implementing library functions in terms of these components rather than parallel loops, the library permits simple source code to execute efficiently.

From an application developer's perspective, Triolet presents an extensible set of data-parallel higher-order functions that help manipulate aggregate data structures. A Triolet parallel loop resembles sequential Python code that uses list comprehensions and higher-order functions to manipulate lists. For example, a function for computing the dot product of two vectors could be defined as follows.

```
def dot(xs, ys):
  return sum(x*y for (x, y) in par(zip(xs, ys)))
```

This statement defines `dot` as a function of `xs` and `ys`. The body of `dot` calls the library functions `zip`, `par`, `map`, and `sum` to sum the elementwise product of `xs` and `ys`. The call of `map` arises from desugaring the list comprehension (`...for...in...`) [27], used here to mutiply the input lists elementwise after zipping them together. Loops execute sequentially by default. The call to `par` designates the loop as parallel, directing the library to use all available parallelism when computing the dot product.

Though not computationally intensive enough to benefit from cluster parallelism, `dot` exhibits scaling and efficiency challenges that also limit the performance of realistic workloads. Multiple library calls commonly express what should be a single parallel loop. In this case, Triolet fuses the calls of `zip`, `map`, `par`, and `sum` into a single loop to minimize communication and memory traffic. Input data should be identified and, if appropriate, partitioned across cluster nodes. Triolet partitions the arrays `xs` and `ys` into subarrays and distributes them to cluster nodes. Results from each thread should be aggregated locally within each node to minimize the cost of communicating results. Each thread computes its own private sum, and these are summed on each node, produce a single value per node that is sent back to the main thread. While `dot` has a relatively simple looping pattern, some algorithms loop over irregular or multidimensional iteration spaces, which adds complexity to loop fusion, work distribution, and result collection.

Triolet uses a library-driven approach to generating efficient parallel loops. Optimized, modular components are assembled into loops by library code, effectively decoupling loop optimizations from the design of the compiler. Compile-time optimization on user code often yields statically generated loops. Library-driven loop optimizations have been introduced previously [5, 7, 12], and Triolet adopts some of these techniques. Triolet's compiler incorporates a suite of general-purpose optimizations on a typed, functional intermediate language [17, 26]. The library uses this optimization infrastructure like a metaprogramming framework: library functions examine their arguments and assemble functions embodying the code to execute, and as long as this assembly process depends only on statically known information, that code will be built and simplified during compile-time optimizations. In the simplified program, the optimizer is often able to replace costly dynamic features such as anonymous functions and heap allocation by cheaper control flow and register-allocatable local variables.

Some library functions return lazily evaluated arrays, called *iterators*, to enable loop fusion. An iterator represents a collection of tasks, each of which computes some result values. Typical parallel iterator representations bear a close connection to parallel loops: the collection of tasks is encoded as a function of the loop counter (the loop body) together with a range of loop counter values (the iteration space) [1, 12]. This is an *indexed* iterator representation, since elements are retrieved by index. In the parallel `dot` function, `zip`, `par`, and `map` return iterators. The call to `zip` represents a parallel loop whose body is the function $\lambda i \to \langle \texttt{xs}[i], \texttt{ys}[i] \rangle$ that, when called with a value for parameter i, retrieves the ith elements of both input arrays and tuples them together. (We use Haskell syntax for examples of code within the optimizer, and use brackets for array subscripting.) This iterator is passed to `par`, which leaves the loop body unchanged. The call to `map` constructs a new iterator whose body performs the work of `zip` (with the function call below), unpacks the result (with the case expression), and multiplies values:

$$\lambda k \to \textbf{case}\ (\lambda i \to \langle \texttt{xs}[i], \texttt{ys}[i] \rangle)\ k \\ \textbf{of}\ \langle x, y \rangle \to x * y$$

Compile-time optimizations eliminate the function call and temporary tuple, producing $\lambda k \to \texttt{xs}[k] * \texttt{ys}[k]$, which performs `zip`'s array lookups followed immediately by `map`'s multiplication. Thus, a carefully chosen iterator representation allows compile-time optimizations to fuse loops.

Indexed representations are not suited to parallelizing irregular loops, where the set of loop iterations to execute is determined dynamically. A common use of irregular parallelism is to generate many inputs, then conditionally skip inputs or expand them into multiple inputs for subsequent processing. This can be parallelized by partitioning the initial set of inputs evenly across parallel processes. To support this work distribution, a loop body should not implement the operation "get the nth intermediate result," which requires a communication phase to count intermediate results, but rather "get each intermediate result generated from the nth input." Each parallel process loops over a subset of inputs and, in an inner loop, generates and processes intermediate results from each input.

To provide the convenience of a shared memory programming interface on a cluster, iterator-based parallelization needs a way to automatically send data over the network. Prior iterator-based parallelization either assumes a shared memory for all parallel tasks [1, 12], sends each distributed task a copy of all objects that are referenced by its input [15, 22], or statically analyzes memory references to find the subarrays used by distributed tasks [11]. While the last approach has the potential to maximize performance by eagerly sending each task only the subset of data it needs, it is challenging to provide a static array reference analysis that yields precise results in the presence of higher-order functions and pointer data structures.

Triolet's library takes advantage of the access patterns implicit in higher-order function calls to partition arrays across parallel tasks without the need for strong compile-time analysis. Many algorithms employ regular array traversals that can be cleanly expressed as compositions of library calls. To track traversal patterns, each iterator contains a representation of both a set of parallel tasks and a set of input data.

Iterators include methods for extracting a subset, or *slice*, of the input. When a subset of tasks is sent to a cluster node, the corresponding slice of the input data is extracted and sent as well. This feature enables a parallel 2D block decomposition of dense matrix multiplication to be written in two lines of code. The matrix product AB is computed (after transposing B for faster memory access) by evaluating dot products of rows of A with rows of B^T:

```
zipped_AB = outerproduct(rows(A), rows(BT))
AB = [dot(u, v) for (u, v) in par(zipped_AB)]
```

Here, `dot` is taken to be defined as sequential code so that the computation of a single output block is sequential. The calls to the library function `rows` reinterpret the two-dimensional arrays A and B^T as one-dimensional iterators over array rows, where each array row is a one-dimensional iterator over elements. The zip-like library function `outerproduct` creates a 2D iterator paring rows of A with rows of B^T. Together, the functions on the first line determine a block distribution of input data. Iterators returned by `rows` associate each task with the corresponding array row. From these iterators, `outerproduct` associates each 2D matrix block with the rows of A and B corresponding, respectively, to the block's vertical and horizontal extent. When parallel tasks are launched by the comprehension on the second line of code, each task will be sent only the array rows that it needs to compute its output.

The library uses several techniques to efficiently execute parallel loops for multidimensional, nested, and irregular problems. Operations that compute a new value or array, such as `dot`'s summation or matrix multiplication's construction of the output array, loop over and execute an iterator's tasks and collect results. Nested and irregular iteration spaces are treated as a loop nest, and the library chooses a sequential or parallel implementation for each nesting level individually.

Due to the differences in communication costs over shared memory and over a network, many algorithms benefit from different inter-node and intra-node parallelization strategies. Unfortunately, the majority of existing programming languages and libraries either present a "flat" view of parallelism where all cores are equally remote from one another, as in Eden, or require developers to explicitly manage the distribution of work across levels, as in C with MPI and OpenMP. Triolet uses a two-level work distribution policy that first distributes large units of work to cluster nodes, then subdivides this work among cores within a node. Because library code cannot examine user code to decide whether a loop is worth paralellizing, it relies on user hints. The library functions `par` and `localpar` set a flag in an iterator to indicate that it should be parallelized across the entire system or across a single node, respectively. Single-node parallelization leverages shared memory to obtain speedup on loops that do very little work per byte of data, such as matrix transposition. A skeleton in the library consists of code that, depending on the input iterator's parallelism hint, invokes low-level skeletons for distributing work across nodes, cores within a node, and/or sequential loop iterations in a task.

3. Implementation

This section describes how Triolet's library, compiler, and runtime execute algorithmic skeletons efficiently. Section 3.1 explains prior approaches to loop fusion. Section 3.2 explains how Triolet's iterators build on prior loop fusion techniques. Section 3.3 generalizes iterators for programming with multidimensional arrays. Section 3.4 describes how Triolet manages parallel tasks and communication.

3.1 Loop Fusion Background

Triolet builds on several existing loop fusion strategies in order to avoid the shortcomings of each. This section introduces these fusion strategies.

At the most basic level, a library may operate only on data structures whose contents are stored in memory, such as arrays. Each skeleton is a loop that reads its entire input and writes its entire output. Compilers may fuse loops by rewriting known patterns of function calls [2, 6, 20]. For example, any pair of calls to filter matching the pattern filter g (filter f a) can be fused by rewriting it to filter $(\lambda x \rightarrow f\ x\ \&\&\ g\ x)$ a. The rewritten code does the work of both filter calls in one pass over a. Because this approach involves designing ad-hoc fusion rules for each pattern, its effectiveness is limited by a library implementor's ability to anticipate and make rules for all combinations of function calls. A more systematic approach is necessary to fuse a larger inventory of loops, especially as nested loops multiply the space of possible looping patterns.

In an imperative setting, loop fusion interleaves the execution of loops on an iteration-by-iteration basis [13]. However, loops with variable numbers of outputs, such as filter, have dependence patterns that can't be expressed purely in terms of loop iterations. In the example above, the first iteration of the second call to filter may depend on any iteration of the first call to filter. Consequently, iteration-based loop fusion techniques cannot fuse this example.

Triolet uses a relatively simple and robust approach to loop fusion that depends only on general-purpose compile-time optimizations. This approach uses what we call *virtual* data structures in place of some of a program's arrays. Several virtual data structure encodings, listed as the rows of Figure 1, have been developed to enable loop fusion. What they have in common is that they all contain a function that is called to compute the data structure's contents. Use of a function effectively defers computation until results are needed. General-purpose compile-time optimizations inline the function at the site where it is used, typically in a loop body, fusing loops.

The rest of this section presents the virtual data structures in Figure 1 and explains why each encoding has limited

	Parallel	Zip	Filter	Nested traversal	Mutation
Indexer	yes	yes	no	no	no
Stepper	no	yes	yes	slow	no
Fold	no	no	yes	yes	no
Collector	no	no	yes	yes	yes

Figure 1. Features of fusible virtual data structure encodings. A "no" means the feature cannot be used or its output is not fusible. A "slow" means the feature may be much less efficient than a handwritten loop.

applicability. We then introduce a new encoding used by Triolet that builds on these encodings to work around their limitations. We use as an example a list holding consecutive integers $[0, 1, 2]$. When used as the input to a skeleton, this list is analogous to a counted loop with three iterations.

Indexers An indexer encoding consists of a size and a lookup function. The ith element of a data structure is retrieved by calling the lookup function with argument i. The example list would be encoded as the pair $\langle 3, \lambda i \to i \rangle$: the list's size is 3, and its ith element is i. Mapping a function f over this data structure builds a new virtual list whose lookup function calls the original lookup function, then calls f on the result: $\langle 3, \lambda j \to f\ ((\lambda i \to i)\ j) \rangle$, which the compiler simplifies to $\langle 3, f \rangle$. Summing the elements of this data structure proceeds by looping over all indices less than 3, calling the lookup function on each, and summing the results. The `map` function and many other indexer-based skeletons consist of straight-line code that builds an indexer. Loop fusion becomes a function inlining task, which is typically easier for compilers to accomplish than traditional loop transformations.

Since indexers allow any element to be retrieved independently of the others, indexers can be used in parallel loops. In C++, readable random access iterators fill the role of indexers. Thrust [1] and Repa [12] use indexers internally to generate fused parallel loops. Parallel loop bodies in functional languages [8, 23] resemble indexers, though they are special syntactic forms rather than ordinary functions.

While the random-access nature of indexers affords parallelism, it also makes indexers unsuitable for irregular loops or loops that write mutable data structures. The user of an indexer is free to choose how to execute the indexer's lookup function, making it difficult to predict side effects when an indexer is created. Triolet prevents such unpredictability by disallowing parallel access to mutable data structures. Functions that produce a variable number of outputs per input cannot be fused by encoding them with indexers. This includes the skeletons concatMap for nested traversal and filter for conditionally skipping elements. To retrieve a value at one index, one must compute some information about all lower indices, which wastes work. For instance, to look up the output at index 10, it's necessary to find the producers of all output elements up to index 10. The usual solution is to precompute the necessary index information using a parallel scan, but because parallel scan is a multipass algorithm, fusion is impossible; all temporary values have to be saved to memory at some point.

Steppers A stepper encoding is a coroutine that returns one data structure element each time it is run, until all elements have been extracted. Steppers are not parallelizable since it is only possible to retrieve the "next" element at any given time. In C++ and other imperative languages, readable forward traversal iterators play the role of steppers. The Haskell vector library, which provides high-performance sequential loops over arrays, uses the fusible stepper encoding presented by Coutts et al. [7]. In this encoding, a stepper is a data structure containing a suspended loop state and a function for stepping to the next loop iteration. The stepper function returns a result value holding the loop's result and the starting state of the next iteration, or else indicating that the last iteration has completed. Similarly to indexers, loop fusion extends a stepper with code that does further processing on the stepper's result and state.

Steppers are a fairly versatile sequential encoding. Their main drawback is that, although nested traversals are fusible, they are not reliably optimized to nested loops [7]. In our two Eden applications that use nested loops, using steppers was roughly a factor of two to five slower than imperative loop nests.

Folds A data structure can be encoded as a function that folds over its elements in some predetermined order. The function calls a given worker function on each data structure element to update an accumulator. The list $[0, 1, 2]$ has the following fold encoding.

$$\begin{aligned}\lambda w\ z \to \textbf{let}\ \mathsf{loop}\ i\ x = \textbf{if}\ i == 3 \\ \qquad\qquad\qquad\qquad\quad \textbf{then}\ x \\ \qquad\qquad\qquad\qquad\quad \textbf{else}\ \mathsf{loop}\ (i+1)\ (w\ i\ x) \\ \qquad\quad \textbf{in}\ \mathsf{loop}\ 0\ z\end{aligned}$$

Its meaning is clearer after unrolling the loop to get $\lambda w\ z \to w\ 2\ (w\ 1\ (w\ 0\ z))$, which calls w to update an accumulator with the values 0, 1, and 2 in turn. Nested traversals do not pose the same optimization trouble for folds that they do for steppers. In a nested traversal, the worker function passed into w calls another fold function that contains its own loop. Inlining moves the value of w to its callsite in the body of loop, bringing the inner fold function along to produce a nested loop.

Unlike indexers and steppers, the fold encoding offers no flexibility in execution order. A fold processes each data structure element in sequence without interruption. This inflexibility rules out fusion of zip skeletons, which pair up elements at corresponding indices in multiple data structures. A fused zip skeleton would read from each of several data structures in an interleaved fashion. It is a common pattern to store data in a structure-of-arrays format, then zip the ar-

rays together in preparation for a loop that uses all the fields. Folds do not support this pattern.

Collectors A collector is an imperative variant of a fold. Instead of updating an accumulator, the worker function uses side effecting operations to update its output value. Collectors are used by Scala's collection library [20] and SkeTo [25]. Triolet uses collectors in sequential code for histogramming and for packing variable-length output skeletons' results into an array.

Conversions The rows of Figure 1 are ordered by how much the user of a virtual data structure can control its execution order. Indexers offer the greatest control, steppers offer less, and folds and collectors offer no control. A higher-control encoding can be converted to a lower-control one. Although no encoding supports zips and mutation, for instance, one could fuse `histogram(n, map(f, zip(a, b)))` by zipping and mapping over indexers, converting the result to a collector, and computing a histogram of the result. A collector that is created from indexer $\langle n, f \rangle$ loops over all indices up to n, calls f on each index, and passes the generated values to the worker function:

$$
\begin{aligned}
&\mathsf{idxToColl}\ \langle n, f\rangle = \\
&\quad \lambda w\ s \rightarrow \textbf{let}\ \mathsf{loop}\ i\ s_2 = \textbf{if}\ i == n \\
&\qquad\qquad\qquad\qquad\quad \textbf{then}\ s_2 \\
&\qquad\qquad\qquad\qquad\quad \textbf{else}\ \mathsf{loop}\ (i+1)\ (w\ (f\ i)\ s_2) \\
&\qquad\quad \textbf{in}\ \mathsf{loop}\ 0\ s
\end{aligned}
$$

However, this conversion removes the potential for parallelization, since a collector's use of side effects is not compatible with parallel execution.

Triolet's iterator library is layered on top of a library of fusible operations for manipulating each of these virtual data structures. We use conventional names for these library functions along with a subscript to indicate what encoding they are implemented for, e.g., $\mathsf{map}_{\mathsf{Idx}}$, $\mathsf{map}_{\mathsf{Step}}$, $\mathsf{map}_{\mathsf{Fold}}$, and $\mathsf{map}_{\mathsf{Coll}}$ are map functions over indexers, steppers, folds, and collectors. We use conversion functions named by their input and output encoding, such as idxToColl.

3.2 Hybrid Iterators

There is at least one fusible encoding supporting every desirable feature in Figure 1, and this suggests that a hybrid encoding could overcome the limitations in the previous section. Triolet's iterator encoding represents a loop nest with either an indexer or stepper encoding at each nesting level. To illustrate, consider the computation of `sum(filter(lambda x: x > 0), xs)`, which selects the positive numbers in array `xs` and sums them. Suppose `xs` has the value $[1, -2, -4, 1, 3, 4]$. The call to `filter` returns $[1, 1, 3, 4]$. For the implementation of `sum`, indexers are the only parallelizable, fusible form at our disposal so far. Using indexers, each thread is assigned a specific number of elements to process. For instance, one thread may sum the first two values while the other sums the last two. Unfortunately, computing which index each output of `filter` resides at requires a complete pass through the data, making a fusible indexer encoding impossible.

A better fusion strategy is to partition the input array `xs` across threads and have each thread sequentially filter and sum one partition. The key to fusion is that our implementation of `filter` does not reassign indices, but rather produces either zero or one output at each index so that it is compatible with indexer-based parallelization and fusion. Conceptually, the call to `filter` transforms $[1, -2, -4, 1, 3, 4]$ into the nested list $[[1], [], [], [1], [3], [4]]$, then the call to `sum` partitions this nested list into $[[1], [], []]$ and $[[1], [3], [4]]$ and sums the two parts in parallel. By encoding the nested list as an indexer of steppers, we ensure that the filter computation is fused with the summation.

In general, a loop may have arbitrarily nested filter operations and/or traversals. Each level of nesting may produce a predetermined number of values using an indexer, or a variable number of values using a stepper. Thus an iterator can consist of an indexer containing values, a stepper containing values, an indexer containing iterators, or a stepper containing iterators. We name these cases IdxFlat, StepFlat, IdxNest, and StepNest and make them the constructors of the Iter data type:

$$
\begin{aligned}
&\textbf{data}\ \mathsf{Iter}\ \alpha\ \textbf{where} \\
&\quad \mathsf{IdxFlat} :: \mathsf{Idx}\ \alpha \rightarrow \mathsf{Iter}\ \alpha \\
&\quad \mathsf{StepFlat} :: \mathsf{Step}\ \alpha \rightarrow \mathsf{Iter}\ \alpha \\
&\quad \mathsf{IdxNest} :: \mathsf{Idx}\ (\mathsf{Iter}\ \alpha) \rightarrow \mathsf{Iter}\ \alpha \\
&\quad \mathsf{StepNest} :: \mathsf{Step}\ (\mathsf{Iter}\ \alpha) \rightarrow \mathsf{Iter}\ \alpha
\end{aligned}
$$

Nested iterators can be understood as loop nests where all loops work together to produce a sequence of values.

Triolet's iterators are flexible enough to fuse all the difficult patterns in Figure 1, while also keeping indexer-based loops available so that they can be distributed across parallel tasks. Figure 2 shows Iter-based implementations of some common skeleton functions. The functions zip, filter, concatMap, and collect implement four of the five features from Figure 1. Section 3.4 addresses the remaining feature, parallelism. Reductions are illustrated by sum. Each function examines its input iterator's constructor (i.e., what loop structure was passed in), and executes code suitable for handling that loop structure. Thus, most functions in Figure 1 are defined by four equations, one for handling each constructor. A function's output loop structure is always determined solely by its input loop structure, ensuring that any composition of known function calls can be simplified statically. In the sum-of-filter example, iterating over the input array produces an IdxFlat term. The compiler inlines the implementation of `filter` for this form of iterator, which yields an IdxNest term as the argument to sum. The compiler inlines sum for this form of iterator, exposing a recursive call

```
zip (IdxFlat xs) (IdxFlat ys) = IdxFlat (zip_Idx xs ys)
zip xs ys = StepFlat (zip_Step (toStep xs) (toStep ys))
  where
    toStep (IdxFlat xs) = idxToStep xs
    toStep (StepFlat xs) = xs
    toStep (IdxNest xss) =
        concatMap_Step toStep (idxToStep xss)
    toStep (StepNest xss) = concatMap_Step toStep xss

filter f (IdxFlat xs) =
    IdxNest (map_Idx (StepFlat ∘ filter_Step f ∘ unit_Step) xs)
filter f (StepFlat xs) = StepFlat (filter_Step f xs)
filter f (IdxNest xss) = IdxNest (map_Idx (filter f) xss)
filter f (StepNest xss) = StepNest (map_Step (filter f) xss)

concatMap f (IdxFlat xs) = IdxNest (map_Idx f xs)
concatMap f (StepFlat xs) = StepNest (map_Step f xs)
concatMap f (IdxNest xss) =
    IdxNest (map_Idx (concatMap f) xss)
concatMap f (StepNest xss) =
    StepNest (map_Step (concatMap f) xss)

collect (IdxFlat xs) = idxToColl xs
collect (StepFlat xs) = stepToColl xs
collect (IdxNest xss) =
    λw s1 → idxToColl xss (λxs s2 → collect xs w s2) s1
collect (StepNest xss) =
    λw s1 → stepToColl xss (λxs s2 → collect xs w s2) s1

sum (IdxFlat xs) = sum_Idx xs
sum (StepFlat xs) = sum_Step xs
sum (IdxNest xss) = sum_Idx (map_Idx sum xss)
sum (StepNest xss) = sum_Step (map_Step sum xss)
```

Figure 2. Triolet iterator functions.

to sum that is also inlined:

```
sum (filter f (IdxFlat ys))
 = sum (IdxNest (map_Idx (StepFlat ∘ filter_Step f ∘ unit_Step) ys))
 = sum_Idx (map_Idx (sum ∘ StepFlat ∘ filter_Step f ∘ unit_Step) ys)
 = sum_Idx (map_Idx (sum_Step ∘ filter_Step f ∘ unit_Step) ys)
```

Iterators are completely eliminated, leaving behind indexer and stepper code that further simplifies to a simple loop nest.

Zipping together two loops involves pairing up corresponding iterations. Since indexers allow elements to be retrieved by iteration number, flat indexers can be zipped into a new indexer. This preserves the potential parallelism in regular loops, such as when zipping arrays together. Other forms of iterator involve variable-length outputs and require scanning through outputs to find corresponding elements. These are zipped together sequentially using steppers. The variable-output functions filter and concatMap work similarly to each other. They add a level of loop nesting in order to preserve potential outer-loop parallelism and avoid the overhead of stepper-based nested traversals. Functions that consume iterators, like collect and sum, transform each level of nesting into a loop.

The functions in Figure 2 need to be inlined to enable subsequent optimizations. Compilers are normally reluctant to inline recursive functions, as doing so can blow up code size and/or execution time. We implement constructor-aware inlining control to inline recursive functions only when it would benefit subsequent optimization. We manually annotate library functions that should be inlined only when the compiler knows their Iter argument's constructor, which ensures that inlining only occurs when it would expose further optimization opportunities. Inlining eventually terminates because each level of recursion consumes one level of statically known loop nesting.

3.3 Multidimensional Iterators

So far, the Iter data type is good for variable-length and nested looping patterns, but is awkward for looping over multidimensional arrays. On the other hand, indexer-based libraries like Repa and loop-based functional languages like Single Assignment C are well-suited to loops over multidimensional arrays, but they do not support fusion of irregular loops. This section generalizes Iter for multidimensional loops and arrays.

Matrix transposition is an example of an algorithm that is awkward to write using one-dimensional arrays. The transpose of a matrix `A` can be defined by giving a loop that retrieves input matrix element `A[x,y]` for a given output position `(y, x)`. For a given matrix `A` whose dimensions are `w` and `h`, this would be written `[A[x,y] for (y, x) in arrayRange((0,0), (h, w))]`. The functions `map` (implicitly called by the comprehension) and `arrayRange` are overloaded for multidimensional iteration spaces.

Simulating a multidimensional loop using one-dimensional iterators would introduce overhead. Expressing transposition in flattened form, using a 1D loop over a 1D array, would require expensive division and modulus operations to reconstruct the 2D indices `x` and `y` from a 1D loop index. Alternatively, using an array of arrays adds an additional pointer indirection to subsequent lookups.

We introduce a type class called Domain to characterize index spaces. Each index space is a type that is a member of Domain. One-dimensional organizations of data, as we have been discussing up until now, have type Seq. A value of type Seq holds an array length. Two-dimensional arrays have a width and a height, so a two-dimensional domain Dim2 holds a pair of integers.

```
data Seq = Seq Int
data Dim2 = Dim2 Int Int
```

Each domain type d has an associated type Index d whose values identify individual indices within a domain. An Index Seq is an Int and an Index Dim2 is an $\langle \mathsf{Int}, \mathsf{Int} \rangle$.

Functions that deal with the indices represented by a domain, for instance by looping over a domain's indices,

are overloaded for different domain types. The definition of class Domain, below, lists overloaded types and functions that are used in this paper.

```
class Domain d where
    type Index d
    idxToFold :: (α → β → β) → β → Idx d α → β
    idxToColl :: (α → State → State) → Idx d α →
                 State → State
    zipWith :: (α → β → γ) → Iter d α → Iter d β →
               Iter d γ
```

The functions idxToFold and idxToColl convert an indexer to a fold or collector that loops over all points in the domain. The function zipWith visits all points in the intersection of two domains.

We also generalize Idx to arbitrary domains d:

```
type Idx d α = ⟨d, Index d → α⟩
```

Finally, we generalize Iter over arbitrary domains. Every Idx α is converted to a Idx d α, producing the following generalized algebraic data type.

```
data Iter d α where
    IdxFlat :: Idx d α → Iter d α
    StepFlat :: Step α → Iter Seq α
    IdxNest :: Domain d ⇒ Idx d (Iter Seq α) → Iter Seq α
    StepNest :: Step (Iter Seq α) → Iter Seq α
```

Only the IdxFlat constructor can create iterators of arbitrary domain types. It simply allows an Idx d α to be used through the Iter interface. The other three constructors contain variable-length traversals, which do not preserve array dimensionality—removing arbitrary elements of a 2D array does not in general yield a 2D array, for instance—so it does not make sense for them to build multidimensional iterators.

3.4 Parallelism

Triolet's runtime uses Threading Building Blocks for thread parallelism and MPI for distributed parallelism. We implement wrapper functions that expose these interfaces as generic parallel skeletons. On top of these wrappers, we layer high-level skeletons that allow users to select what degree of parallelism to use.

We add a field to Iter holding a flag to indicate what degree of parallelism to use. Users designate an iterator as parallel by calling `localpar` (for thread parallelism) or `par` (for distributed and thread parallelism) on it, thereby setting the flag. Parallel skeletons inspect the flag and invoke the appropriate distributed, threaded, and sequential functions. For instance, a distributed-parallel histogram performs a distributed reduction, which performs one threaded reduction per node, which sequentially builds one histogram per thread.

Triolet includes runtime facilities for serializing and deserializing objects to byte arrays. The compiler automatically generates serialization code from the definitions of algebraic data types. Functions are represented by heap-allocated closures and are also serialized. Serializing an object transitively serializes all objects that it references. Pointers to global data are serialized as a segment identifier and offset. Since the majority of serialized data typically resides in pointer-free arrays, such arrays are serialized using a block copy to minimize serialization time.

3.5 Array partitioning

In parallel array traversals, the library identifies what array subset a task will use and extracts the subset to send over the network. Consider the loop `sum(par(xs))`. If parallelized on two nodes, each node should receive half of the array `xs`. Iterating over xs produces an indexer function $\lambda i \rightarrow xs[i]$ that reads an element of xs, and loop fusion merges this with additional code from `sum`. Using the compiler-generated serialization, when the function is serialized and sent to cluster nodes, the function's reference to xs would drag the entire array along with it.

We enhance the functionality of indexers so that the library can partition arrays that are traversed in parallel. First, we reorganize indexers' lookup functions into a (potentially large) data source and a value-extracting function. In the example, the function $\lambda i \rightarrow xs[i]$ is reorganized into the source array xs and an extractor that takes the source as an additional parameter, $\lambda xs\ i \rightarrow xs[i]$. This function is cheap to serialize since it does not directly reference the array.

Then, we extend the indexer type with a method for extracting a data subset or *slice*. An indexer's slice method builds a new indexer whose data source holds only the data used by the extracted slice. When a distributed parallel loop partitions work across nodes, it extracts and sends the slice needed for each node's chunk of work. The slice extraction process extracts subarrays, which are serialized and sent by the runtime. This approach is flexible enough to efficiently handle common cases of regular array traversal. No copying overhead is introduced in inner loops since data sources are used in-place and an indexer's functions are typically inlined into their use sites. Data sources may involve multiple arrays, such as in the result of a call to `zip` or `outerproduct`, without requiring a step of data copying and reorganization.

4. Evaluation

To show how a solid skeleton framework can deliver high performance without burdening programmers, we have converted four Parboil benchmarks [24] into Triolet, Eden and C+MPI+OpenMP. For each benchmark, we normalize performance as speedup against sequential C to provide a measure of absolute performance. As a highly efficient implementation layer, the C+MPI+OpenMP serves as a useful reference point against which to evaluate the scalability and

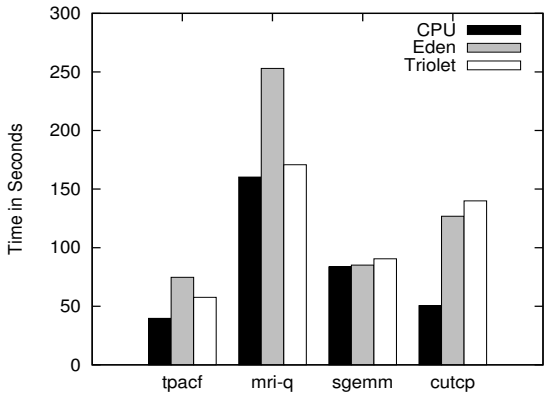

Figure 3. Sequential execution time of benchmarks.

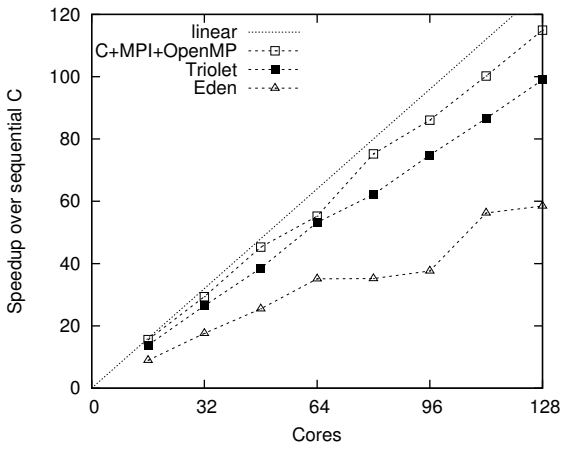

Figure 4. Scalability and performance of `mri-q` implemented in different languages.

parallel overhead of the high-level languages. We measure sequential execution time (Figure 3) to compare the efficiency of the code without communication overhead. We use similar parallelization strategies across languages in order to reveal the overhead imposed by each language.

C code is compiled with GCC 4.7.3 `-O3`. Eden code is compiled with GHC-Eden 7.6.1 `-O2` with LLVM 3.2 as the backend. All three versions use OpenMPI as a distributed communication layer. Tests are run on a group of 8 Amazon EC2 cluster compute nodes with two 8-core Xeon E5-2670 processors per node (a total of 16 cores per node). Hyperthreading is disabled. Parboil includes a range of input problem sizes for each benchmark. We select data sets with a sequential C running time between 20 and 200 seconds. This is large enough for the C+MPI+OpenMP code to scale up to the full test system. Parallel times are the average of 5 runs.

4.1 Eden Overview

Eden is a distributed parallel extension of the Glasgow Haskell Compiler (GHC) [15]. Eden uses either MPI or PVM to create and communicate among parallel execution contexts. Processes do not share memory.

We write parallel loops using map and reduce skeletons with sequential tasks that manipulate unboxed arrays. In array manipulation code, we follow a high-level Haskell programming style using the vector and Repa libraries where possible. Where this is inefficient, we rewrite tasks to use imperative loops and mutable arrays: for nested loops that build histograms in `tpacf` and `cutcp`, and for performing random-access array writes when building a 2D array from subarrays in `sgemm`.

We implement parallel skeletons that use a two-level work distribution similar to Triolet and C+MPI+OpenMP. The main process distributes work to one process in each node, which further distributes work to other processes in the same node. This avoids the communication bottleneck with the main process in Eden's skeleton library, where the main process directly communicates with all other processes.

4.2 MRI-Q

The main loop of `mri-q` computes a non-uniform 3D inverse Fourier transform to create a 3D image. This computation is straightforward to parallelize. In Triolet, it can be distilled down to the following two lines of code.

```
[sum(ftcoeff(k, r) for k in ks)
 for r in par(zip3(x, y, z))]
```

This consists of a parallel map over image pixels, summing contributions from all frequency-domain samples. Although this code contains only a call to `par` to control parallelization, it yields parallel performance nearly on par with manually written MPI and OpenMP (Figure 4).

In Eden, we build arrays in chunked form, as lists of 1k-element vectors, so that the runtime can distribute subarrays to processors while still benefiting from efficient array traversal. Unfortunately, Eden loses performance across the entire range. Eden's backend misses a floating-point optimization on `sinf` and `cosf` calls, resulting in about 50% longer run time on a single thread (Figure 3). While Eden scales fairly well, tasks occasionally run significantly slower than normal. With more nodes, it is more likely that a task will be delayed, reducing the observed scalability.

C+MPI+OpenMP is the most verbose, dedicating more code to partitioning data across MPI ranks than to the actual numerical computation. While `mri-q`'s communication pattern fits MPI's scatter, gather, and broadcast primitives, these were not as efficient as the Triolet code; the fastest version used nonblocking, point-to-point messaging.

4.3 SGEMM

The scaled product αAB of two 4k by 4k-element matrices is computed in `sgemm`. We parallelize the multiplication after transposing matrices so that the innermost loop accesses contiguous matrix elements.

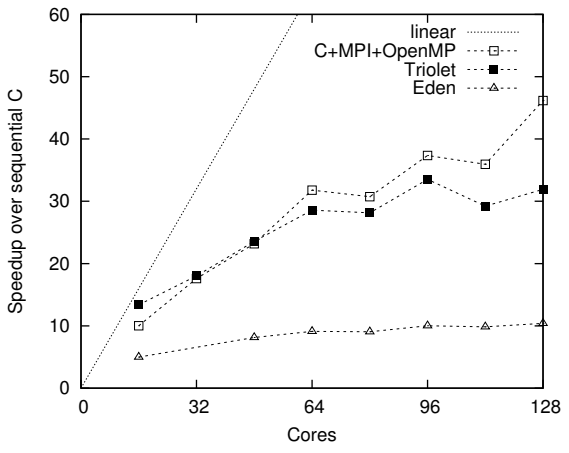

Figure 5. Scalability and performance of `sgemm` implemented in different languages.

```
1  def correlation(size, pairs):
2    values = (score(size, u, v)
3              for (u, v) in pairs))
4    return histogram(size, values)
5
6  def randomSetsCorrelation(size, corr1, rands):
7    empty = [0 for i in range(size)]
8    def add(h1, h2):
9      return [x + y for (x, y) in zip(h1, h2)]
10   return reduce(add, empty,
11                 par(corr1(r) for r in rands))
12
13 def selfCorrelations(size, obs, rands):
14   def corr1(rand):
15     indexed_rand = zip(indices(domain(rand)), rand)
16     pairs = localpar((u, v)
17                      for (i, u) in indexed_rand
18                      for v in rand[i+1:])
19     return correlation(size, pairs)
20   return randomSetsCorrelation(size, corr1, rands)
```

Figure 6. Triolet code of `tpacf`'s self-correlation loop

All three versions use a 2D block-based parallel decomposition that sends each worker only the input matrix rows that it needs to compute its output block. As discussed in Section 2, this data decomposition can be written using Triolet's `rows` and `outerproduct` library functions. Similar decompositions are written as part of the parallel C+MPI+OpenMP and Eden code. This took over 120 lines of code in each language, adding development complexity and detracting from the code's readability.

Transposition is a sequential bottleneck in Eden since it does too little work to parallelize profitably on distributed memory. We parallelize it over shared memory on a single node in Triolet and C+MPI+OpenMP. At 128 cores, transposition takes 35% of Eden's execution time.

All versions of the code exhibit limited scalability due to transposition time and communication time (Figure 5). C+MPI+OpenMP and Triolet spend similar amounts of time in communication and in parallel computation, resulting in similar performance. Triolet's performance stops rising toward 8 nodes as it spends more time constructing messages. At 8 nodes, 40% of Triolet's overhead relative to C+MPI+OpenMP is attributable to the garbage collector [3], which is slow when allocating objects comprising tens of megabytes. The garbage collection overhead was determined by comparing to the run time when libc `malloc` was substituted for garbage-collected memory allocation. The Eden code fails at 2 nodes because the array data is too large for Eden's message-passing runtime to buffer.

4.4 TPACF

The `tpacf` application analyzes the angular distribution of observed astronomical objects. It uses histogramming and nested traversals, presenting a challenge for conventional fusion frameworks. Three histograms are computed using different inputs. One loop compares an observed data set with itself; one compares it with several random data sets; and one compares each random data set with itself. We parallelize across data sets and across elements of a data set.

Triolet allows the common code of the three loops to be factored out and written once. The function on lines 1–4 of Figure 6) contains the common part of all three histogram computations, dealing with correlating pairs of values taken from a pair of data sets. This code maps `score` over all given pairs of objects to compute a similarity between members of each pair and collects the results into a new histogram. On lines 6–11, `randomSetsCorrelation` computes a parallel histogram over a collection of random data sets. Parameter `corr1` computes a histogram from one random data set, and `rands` is an array of random data sets. The function body consists of a parallel reduction that computes histograms of individual data sets and adds them together.

The `selfCorrelations` of random data sets are computed in lines 13–20. The function `corr1` computes the self-correlation of one data set `rand` (lines 14–18). Self-correlation examines all unique pairs of values `(rand[i], rand[j])` where `j > i`. Lines 15–18 define a triangular loop building all unique pairs of elements `(u, v)` from `rand`. Line 19 computes a correlation histogram from these pairs. Line 20 runs `corr1` in parallel over the random data sets and sums the generated histograms. The other two parallel histogramming loops are defined similarly to `selfCorrelation`.

Triolet abstracts away the number of threads in the system, while the Eden and C+MPI+OpenMP contain additional code to adapt to the number of threads. The Eden code subdivides data in order to produce enough work to occupy all threads. The C+MPI+OpenMP code examines the number of threads in order to privatize histograms. For a pro-

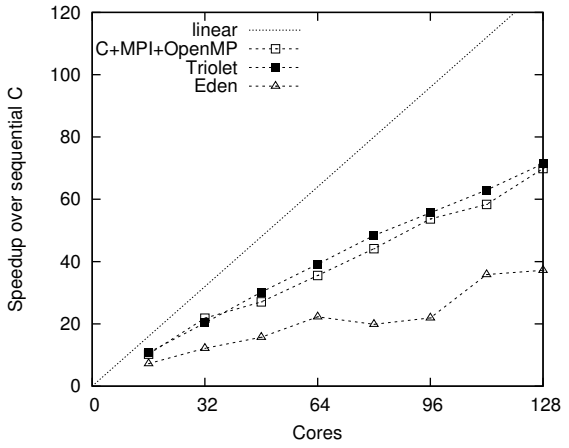

Figure 7. Scalability and performance of `tpacf` implemented in different languages.

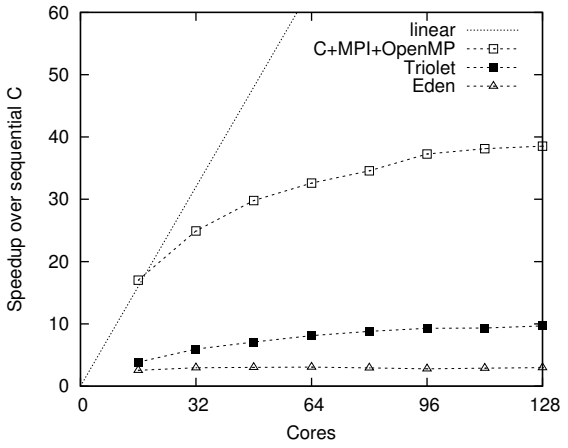

Figure 8. Scalability and performance of `cutcp` implemented in different languages.

grammer, identifying and inserting this code entails one or more iterations of performance optimization.

Triolet and C+MPI+OpenMP scale similarly (Figure 7). Triolet is slightly faster due to a more even distribution of computation time across nodes. Eden has somewhat worse sequential performance and a higher communication overhead.

4.5 CUTCP

The `cutcp` benchmark is taken from a molecular modeling application. It computes the electrostatic potential induced by a collection of charged atoms at all points on a grid. An atom's charge affects the potential at grid points within a distance c. The body of the computation is essentially a floating-point histogram: it loops over atoms, loops over nearby grid points, skips points that are not within distance c, and updates the grid at the remaining points. This computation is done by nested loops and conditionals in the C code or nested traversals in Triolet. Subsets of atoms are processed in parallel.

Performance of Triolet and C+MPI+OpenMP saturates quickly (Figure 8), as the overhead of summing the large output arrays dominates execution time. As in `sgemm`, Triolet has significant garbage collection overhead. Approximately 60% of Triolet's execution time at 8 nodes arises from allocation overhead.

5. Related Work

The use of higher-order functions as a parallel programming interface has a long history. Vector parallel languages extract fine-grained parallel loops from programs, then fuse them into larger tasks [2, 6]. The vector execution model maximizes load balancing and parallelism in irregular workloads, sometimes at the cost of high communication and memory overheads. Later work reduces these overheads in some cases [4, 9]. Another line of development follows a *loop* execution model [8, 12, 23], related to or based on indexers.

Loop fusion has been achieved through iteration-based loop transformations [8, 23], compile-time deforestation [28], expression rewriting [2, 9, 18, 20], and virtual data structures [7, 12]. Delite [20] performs compiler-driven fusion and parallelization of nested traversals.

Algorithmic skeletons abstract out the communication and coordination aspects of parallel programs. Gonzalez [10] surveys recent work on algorithmic skeletons. Triolet's programming model is most similar to the data-parallel, distributed, functional algorithmic skeleton frameworks HDC [11], PMLS [22], and Eden [15]. These frameworks incorporate compile-time generation or library implementations of skeleton code for higher-order functions. Eden also provides low-level primitives for implementing new skeletons. These frameworks do not address irregular loops, do not combine shared memory with message passing, and (in PMLS and Eden) do not provide skeletons for array computation.

6. Conclusion

Functional algorithmic skeletons are a simple, high-level interface to parallel programming, but their overhead relative to lower-level programming models has limited their usefulness. Triolet demonstrates that the most severe performance limitations can be eliminated, yielding superior performance without increased programming complexity. On code that is not communication-bound, performance rivals that of C with MPI and OpenMP. The necessary changes to runtime behavior are familiar to low-level programmers: storing data in arrays, fusing loops, partitioning arrays across distributed tasks, and utilizing shared memory and in-place array updates. Triolet allows skeletons to make those changes under the hood through library enhancements, coupled with compiler and runtime support for data serialization and shared memory parallelism.

Acknowledgments

This project was partly supported by the STARnet Center for Future Architecture Research (C-FAR) and the DoE Vancouver Project (DE-FC02-10ER26004/DE-SC0005515).

References

[1] N. Bell and J. Hoberock. Thrust: A productivity-oriented library for CUDA. *GPU Computing Gems Jade Edition*, page 359, 2011.

[2] G. Blelloch, J. Hardwick, J. Sipelstein, M. Zagha, and S. Chatterjee. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1): 4–14, 1994.

[3] H. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software—Practice & Experience*, 18(9): 807–820, 1988.

[4] B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: Compiling an embedded data parallel language. In *Proc. ACM Symposium on Principles and Practice of Parallel Programming*, pages 47–56, 2011.

[5] M. Chakravarty and G. Keller. Functional array fusion. In *Proc. ACM International Conference on Functional Programming*, pages 205–216, 2001.

[6] M. Chakravarty, R. Leshchinskiy, S. Peyton Jones, G. Keller, and S. Marlow. Data Parallel Haskell: A status report. In *Proc. Workshop on Declarative Aspects of Multicore Programming*, pages 10–18, 2007.

[7] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion: From lists to streams to nothing at all. In *Proc. ACM International Conference on Functional Programming*, pages 315–326, 2007.

[8] J. T. Feo, D. C. Cann, and R. R. Oldehoeft. A report on the sisal language project. *Journal of Parallel and Distributed Computing*, 10(4):349–366, 1990.

[9] M. Fluet, M. Rainey, J. Reppy, and A. Shaw. Implicitly threaded parallelism in manticore. *Journal of Functional Programming*, 20(5–6):537–576, 2010.

[10] H. González-Vélez and M. Leyton. A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. *Software: Practice and Experience*, 40(12): 1135–1160, 2010.

[11] C. Herrmann and C. Lengauer. HDC: A higher-order language for divide-and-conquer. *Parallel Processing Letters*, 10(2–3): 239–250, 2000.

[12] G. Keller, M. Chakravarty, R. Leshchinskiy, S. Peyton Jones, and B. Lippmeier. Regular, shape-polymorphic, parallel arrays in Haskell. In *Proc. ACM International Conference on Functional Programming*, pages 261–272, 2010.

[13] K. Kennedy and K. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Proc. International Workshop on Languages and Compilers for Parallel Computing*, pages 301–320, 1994.

[14] H.-W. Loidl, F. Rubio, N. Scaife, K. Hammond, S. Horiguchi, U. Klusik, R. Loogen, G. Michaelson, R. Peña, S. Priebe, Á. Rebón, and P. Trinder. Comparing parallel functional languages: Programming and performance. *Higher-Order and Symbolic Computation*, 16(3):203–251, 2003.

[15] R. Loogen, Y. Ortega-mallén, and R. Peña-marí. Parallel functional programming in Eden. *Journal of Functional Programming*, 15(3):431–475, 2005.

[16] E. Meijer. Confessions of a used programming language salesman. In *Proc. ACM Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 677–694, 2007.

[17] S. Peyton Jones. Compiling Haskell by program transformation: A report from the trenches. In *Proc. European Symposium on Programming*, pages 18–44, 1996.

[18] S. Peyton Jones, A. Tolmach, and T. Hoare. Playing by the rules: Rewriting as a practical optimisation technique in GHC. In *Proc. ACM Haskell Workshop*, pages 203–233, 2001.

[19] A. Prokopec, T. Rompf, P. Bagwell, and M. Odersky. A generic parallel collection framework. In *Euro-Par'11: Proceedings of the International Conference on Parallel Processing*, pages 136–147, 2011.

[20] T. Rompf, A. Sujeeth, N. Amin, K. Brown, V. Jovanovic, H. Lee, M. Jonnalagedda, K. Olukotun, and M. Odersky. Optimizing data structures in high-level programs: new directions for extensible compilers based on staging. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 497–510, 2013.

[21] N. Scaife, G. Michaelson, and S. Horiguchi. Comparative cross-platform performance results from a parallelizing SML compiler. In *Proc. International Workshop on Implementation of Functional Languages*, pages 138–154, 2002.

[22] N. Scaife, S. Horiguchi, and G. M. P. Bristow. A parallel SML compiler based on algorithmic skeletons. *Journal of Functional Programming*, (4):615–650, 2005.

[23] S.-B. Scholz. Single Assignment C—efficient support for high-level array operations in a functional setting. *Journal of Functional Programming*, 3(6):1005–1059, 2003.

[24] J. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. Liu, and W.-M. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. Technical Report IMPACT-12-01, University of Illinois at Urbana-Champaign, 2012.

[25] H. Tanno and H. Iwasaki. Parallel skeletons for variable-length lists in SkeTo skeleton library. In *Euro-Par 2009 Parallel Processing*, volume 5704 of *Lecture Notes in Computer Science*, pages 666–677. 2009.

[26] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *Proc. ACM Conference on Programming Language Design and Implementation*, pages 181–192, 1996.

[27] P. Wadler. List comprehensions. In S. Peyton Jones, editor, *The Implementation of Functional Programming Languages*, pages 127–138. Prentice Hall, 1987.

[28] P. Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, 1988.