

Synthesizing MPI Implementations from Functional Data-Parallel Programs

Tristan Aubrey-Jones¹ · Bernd Fischer²

Received: 11 August 2014 / Accepted: 10 March 2015
© Springer Science+Business Media New York 2015

Abstract Distributed memory architectures such as Linux clusters have become increasingly common but remain difficult to program. We target this problem and present a novel technique to automatically generate data distribution plans, and subsequently MPI implementations in C++, from programs written in a functional core language. The main novelty of our approach is that we support distributed arrays, maps, and lists in the same framework, rather than just arrays. We formalize distributed data layouts as types, which are then used both to search (via type inference) for optimal data distribution plans and to generate the MPI implementations. We introduce the core language and explain our formalization of distributed data layouts. We describe how we search for data distribution plans using an adaptation of the Damas–Milner type inference algorithm, and how we generate MPI implementations in C++ from such plans.

Keywords Data parallelism · Data distribution · Type inference · Code generation · MPI

1 Introduction

Functional languages provide good high-level notations for data parallelism (e.g., [15,22,23,29,35]), but their automatic translation into efficient low-level code for

✉ Tristan Aubrey-Jones
taj105@ecs.soton.ac.uk

Bernd Fischer
bfischer@cs.sun.ac.za

¹ University of Southampton, Southampton, UK

² University of Stellenbosch, Stellenbosch, South Africa

distributed memory architectures remains a problem, due to the many possible data distributions. Many techniques only support a fixed model such as MapReduce [15] that is not necessarily suitable for all problems [18], or do not support distributed memory at all [29]. We present a flexible, type-based technique to search through the space of possible data distributions, and to generate efficient MPI implementations in C++ from the solutions.

We use a high-level core language called Functional language on compute clusters (Floc) to demonstrate our approach. Like PigLatin [27] and PQL [32], Floc takes inspiration from relational algebra. Figure 1 illustrates this data-parallel programming style with a matrix multiplication in SQL and relational algebra. Here, $A \bowtie_p B$ is the join of A and B over the predicate p ; R_1 thus contains all pairs of elements $(A_{i,k}, B_{k,j})$ that contribute to the result. $\rho_{e/x}$ is a renaming that creates a new column x with values e ; R_2 thus contains all products $A_{i,k} * B_{k,j}$. Finally, $G_{i,f}$ is a group-reduce operation that reduces the groups of all tuples that have the same values in the columns i using the function f ; C thus contains at (i, j) the sum of all products $A_{i,k} * B_{k,j}$. This formulation is implicitly parallel and abstracts away from global state, iteration and recursion, and individual element accesses, which helps us to derive data distribution plans. For example, we know from its syntactic structure that the operation $G_{\langle A.i, B.j \rangle, \text{sum}(v)}(R_2)$ only ever reduces groups of v -elements that come from tuples with the same values in the $A.i$ - and $B.j$ -columns of R_2 , respectively. Hence, if we distribute R_2 so that all these tuples are co-located, then the group reduce operation requires no further data exchanges and can run locally.

Our key insight is that we can use *types* to formalize this knowledge about the *data distribution characteristics* of data parallel operations (also called skeletons or combinators), as well as the distribution of the data itself, and *type inference* to derive *data distribution plans* for Floc programs, in a way that works for multiple collection types and not just arrays. We call these types *distributed data layout* (DDL) types; they combine the usual (functional) types with layout information. For example, the DDL type `DArr (Int, Int) Float fst D1 D2` characterizes a two-dimensional array of `Float` values that is partitioned by row and distributed and mirrored over different dimensions of a cluster's node topology. Here, the third argument `fst` of the DDL type constructor `DArr` is the *partition function* that describes which dimensions of the array are partitioned over the nodes in the cluster along the *dimension* `D1` given as the fourth argument. The final argument `D2` gives any dimensions along which each partition selected by the partition function is mirrored. Note that the DDL types are only used by the compiler, and not exposed to the programmer.

The compiler provides different functionally equivalent implementations of the combinators that work for different data distributions, and are thus characterized by different DDL types. For example, the declaration

<pre>SELECT A.i as i, B.j as j, sum (A.v * B.v) as v FROM A JOIN B ON A.j = B.i GROUP BY A.i, B.j;</pre>	$R_1 = A \bowtie_{A.j=B.i} B$ $R_2 = \rho_{A.v*B.v/v}(R_1)$ $C = G_{\langle A.i, B.j \rangle, \text{sum}(v)}(R_2)$
---	--

Fig. 1 Matrix multiplication: SQL (left) and Relational Algebra (right)

```
groupReduceArr2 ::  $\Pi$  (pf,_,_,_) :
  (i1->i2, (i1,v1)->v2, (v2,v2)->v2, DArr i1 v1 pf d m) -> DArr i2 v2 id d m
```

expresses that `groupReduceArr2` binds the value of its first parameter, which is an array partition function, to `pf` and then uses *this specific* `pf` as partition function for the array it reduces. The compiler instantiates the combinators in the user program with the different implementations, and then uses a variant of the standard Damas–Milner type inference algorithm to derive the DDL types that represent data distribution plans for these variants. In a final step, it generates efficient MPI implementations in C++ from these DDL plans.

1.1 Contributions

In this paper we describe the first technique (to our knowledge) to automate the data distribution of data-parallel programs that supports *multiple* distributed collection types, including arrays, lists, and maps. Our approach can easily be extended with further collection types and data-parallel combinators, making it flexible and applicable for a wide variety of data-parallel tasks. This is in contrast to existing approaches that focus on automatically finding the best data distributions for array-based algorithms, where input programs are either nested loops with affine array references [1,3,5], or combinations of array section and reduction operators [13]. We demonstrate our approach for a small domain-specific language for data-parallel functional programming that is inspired by relational algebra. In particular, we formalize distributed data layouts by polymorphic dependent type schemes and use a variant of the standard Damas–Milner type inference algorithm to infer DDLs for different plans in a type-directed way. We have implemented a prototype code generator for DDL plans that targets MPI and C++.

2 Data-Parallel Programming in Flocc

Flocc’s expression and type syntax are shown in Fig. 2. Expressions e can be identifiers, literals, function abstractions, function applications, tuples, let bindings, or if-then-else expressions. Function abstraction arguments and let expressions bind values to tuples of identifiers x . List, array, and map literal expressions are also supported but not shown. Flocc uses Damas–Milner type inference [14] to infer types for all expressions, though function abstractions and let-bindings support optional type declarations which the compiler checks.

At the high-level (i.e., executed on a single processor with a single address space), Flocc has a standard call-by-value reduction semantics. All parallelism in the language is expressed via data-parallel operations applied to the collections. These operations include predefined combinators for arrays, maps, and lists shown in Fig. 3; there are many further combinators not shown here for brevity. In the following we illustrate the language with several examples.

```

 $e ::= Id \mid v \mid (e_1, \dots, e_n) \mid \backslash x [:: t] \rightarrow e \mid e_1 \ e_2 \mid \text{let } x [:: t] = e_1 \text{ in } e_2$ 
 $\mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3$ 
 $x ::= Id \mid \_ \mid (x_1, \dots, x_n)$ 
 $v ::= Int \mid Float \mid True \mid False \mid ()$ 
 $s ::= \forall Id.s \mid t$ 
 $t ::= Id \mid Int \mid Float \mid Bool \mid Null \mid (t_1, \dots, t_n) \mid t_1 \rightarrow t_2 \mid \text{Map } t_1 \ t_2 \mid \text{Arr } Int^+ \ t \mid \text{List } t$ 

```

Fig. 2 Flocc expression and type syntax

```

subArr :: (i, i, Arr i v) -> Arr i v
shiftArr :: (i, Arr i v) -> Arr i v
mapArrInv :: (i->j, (i,v)->w, j->i, Arr i v) -> Arr j w
eqJoinArr :: (i->k, j->k, Arr i v, Arr j w) -> Arr (i,j) (v,w)
groupReduceArr :: (i->j, (i,v)->w, (w,w)->w, Arr i v) -> Arr j w

map :: ((i,v)->(j,w), Map i v) -> Map j w
eqJoin :: ((i,v)->k, (j,w)->k, Map i v, Map j w) -> Map (i,j) (v,w)
allPairs :: ((i,v)->k, Map i v) -> Map (i,i) (v,v)
reduce :: ((i,v)->s, (s,s)->s, Map i v) -> s
groupReduce :: ((i,v)->j, (i,v)->w, (w,w)->w, Map i v) -> Map j w
union :: (Map i v, Map i v) -> Map i v

zip :: (List v, List w) -> List (v,w)
mapList :: (v->w, List v) -> List w
reduceList :: ((v,v)->v, v, List v) -> v

```

Fig. 3 Predefined data-parallel combinators for arrays, maps, and lists

```

let mmul = (\(A,B) :: (Arr (Int,Int) Float, Arr (Int,Int) Float) ->
  -- zip all combinations of rows from A and cols from B
  let R1 = eqJoinArr (snd, fst A, B) in
  -- multiply values from A and B
  let R2 = mapArrInv (id, Float.*, id, R1) in
  -- group by dest & sum-reduce
  let C = groupReduceArr (\((ai,aj),(bi,bj)) -> (ai,bj),
    snd, Float.+, R2) in C) in ...

```

Fig. 4 Matrix–matrix multiplication program

2.1 Matrix Multiplication

In Flocc, the matrix multiplication (cf. Fig. 4) closely follows the relational algebra version (cf. Fig. 1). Here, A and B are arrays with pairs of integers as indices, and floating point values. The array join `eqJoinArr` computes the Cartesian product of both arrays, restricted to entries where the `snd` index from A is equal to the `fst` index from B. It thus returns an array with four indices that contains all pairs of Floats that contribute to the result. `mapArrInv` multiplies each of these pairs (like the renaming), and the aggregation `groupReduceArr` then groups these values using new keys `(ai, bj)` (i.e., the row from A and column from B), and sums up all the values in each group using `Float.+`. We use a standard distributed-memory implementation of this dense matrix multiplication, which partitions A by row, B by column, and C by A's row and B's column, as our main running example in this paper.

```

let hist = (\(N,D) :: (Int, Map k Float) ->
  -- use min/max vals as x-axis bounds
  let (minV, maxV) = reduce (\(_,v) -> (v,v),
    \((x1,y1),(x2,y2)) -> (Float.min(x1,x2),Float.max(y1,y2)),D) in
  -- scaling coefficient to get bucket ids
  let i = Float./ (toFloat (Int.- (N,1)), Float.- (maxV,minV)) in
  let D' = map (\(k,v) -> (k, toInt (Float.* (v,i))), D) in
  -- group by bucket & count group sizes
  groupReduce (snd, \_ -> 1, addi, D')) in ...

```

Fig. 5 N-bucket histogram

```

let dotp = (\(A,B) :: (List Float, List Float) ->
  let AB = mapList (Float.*, zip (A,B)) in
  reduceList (Float.+, 0.0, AB)) in ...

```

Fig. 6 Dot product

2.2 Histograms

The function `hist` (cf. Fig. 5) shows a use of maps in Flocc. It takes a pair of arguments `N` and `D`, where `D` is a map from keys of arbitrary type `k` to floating point values, and computes a histogram of these values. This histogram has `N` equally spaced buckets such that bucket 0 contains the minimum value in `D` and bucket `N-1` contains the maximum. The `reduce` combinator projects the values from the map `D` into pairs and finds the minimum and maximum values. These values are used to calculate the scaling coefficient `i`, which in turn is used to calculate each value's bucket index with the `map` combinator. Here, the key remains unchanged, so `map`'s first argument is `id`. `groupReduce` then uses these bucket indices as the keys for the result map, where `snd` projects them out of the original key-value pairs. For each key-value pair a 1 is projected out (using `_ -> 1`), and then each group of ones is aggregated using `Float.+`, thus counting the entries in each bucket.

2.3 Dot Product

The function `dotp` (cf. Fig. 6) shows a use of lists in Flocc. It takes a pair of lists of floats, and returns their dot product, computed by zipping together the lists, multiplying the pairs, and then sum reducing them.

2.4 Comparison of Code Sizes

Figure 7 compares the code sizes for a number of programs written in Flocc and other languages.¹ The Flocc implementations are between 3 (Histogram) and 32 % (K-means) of the size of the comparative implementations (12 % on average). This illustrates the potential productivity gains of such a high-level language approach.

¹ See <http://www.flocc.net/hlpp14/codesizes.html> for details.

Problem	Flocc	Comparison		Types
Matrix multiply (cf. Fig 4)	5	C/MPI	89	Arr
Floyd's all pairs shortest path	15	C/MPI	88	Arr
Jacobi 2D	8	C++/MPI	120	Arr
SOR red/black	18	C/MPI	289	Arr
N-body (gravitational)	38	C/MPI	153	Arr
K-means clustering	36	C/MPI	114	Map
Triangle enum (cf. Fig 15)	12	C++/MR-MPI	263	Map
R-MAT graph generation	35	C++/MR-MPI	148	Map
PageRank	11	Java/Hadoop	157	Map
Histogram (cf. Fig 5)	6	C++/MPI	204	Map
Apriori association mining	14	Java/Hadoop	371	Map
Dot product (cf. Fig 6)	3	C++/MPI	35	List
Standard deviation	6	C/MPI	38	List
Simple linear regression	10	C++/MPI	47	List
Word count	3	Java/Hadoop	48	List & Map
Grep	2	Java/Hadoop	59	List

Fig. 7 Comparative code sizes (code lines without comments and IO code)

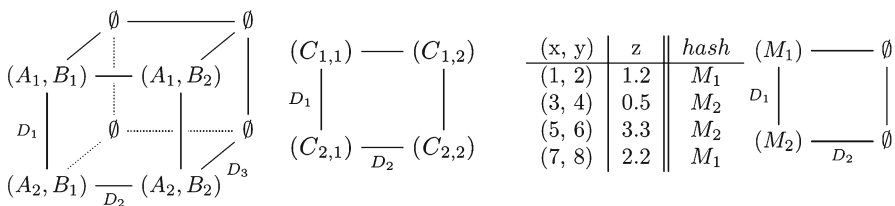


Fig. 8 Array distributions (left, center). Map distribution (right)

3 Distributed Data Layouts as Types

3.1 Distributing Collections on Clusters

MPI allows the definition of virtual node topologies where nodes are addressable via Cartesian coordinates. The MPI implementation then decides how best to map these onto physical nodes. This abstraction is useful, since it allows us to describe where collections are stored and replicated relative to each other, without considering the physical interconnect. We therefore identify nodes using n -dimensional grids with dimensions D_1 to D_n .

Collections can be split into partitions and distributed over the nodes in some of the dimensions, replicated across other dimensions, and are stored at the nodes on the axis of any remaining dimensions. Figure 8 illustrates on the left an input distribution for a matrix multiply on an 8-node cluster organized as a 3D grid. Matrix A is split into two partitions A_1 and A_2 distributed along D_1 , and mirrored across D_2 , but only stored at the axis of D_3 . B is partitioned along D_2 , and mirrored across D_1 , also only at the axis of D_3 . Hence, the node $(0, 1, 0)$ contains the partitions A_2 and B_2 while $(1, 1, 1)$ remains empty. This (without D_3) illustrates the data layouts of our standard `mmul` implementation's input matrices A and B , where A is partitioned by row, and B by column. This `mmul` implementation is the main running example in the sections that follow. Figure 8 (center) illustrates the 2D partitioning of our `mmul` example's

$$\begin{aligned}
dts &::= \forall Id \cdot dts \mid \Pi x : dt_1 \rightarrow dt_2 \mid dt \\
dt &::= Id \mid \text{Int} \mid \text{Float} \mid \text{Bool} \mid \text{Null} \mid (dt_1, \dots, dt_n) \mid dt_1 \rightarrow dt_2 \\
&\quad \mid \text{Arr } i \ t \mid \text{Map } t_1 \ t_2 \mid \text{List } t \\
&\quad \mid \text{DArr } i \ t \ f \ m? \ d_1 \ d_2 \mid \text{DMap } t_1 \ t_2 \ f \ d_1 \ d_2 \mid \text{DList } t \ m? \ d_1 \ d_2 \mid \dots \\
m &::= \text{blk} \mid \text{cyc} \mid (m_1, \dots, m_n) \\
i &::= \text{Int} \mid (i_1, \dots, i_n) \\
x &::= Id \mid _ \mid (x_1, \dots, x_n) \\
d &::= Id \mid (d_1, \dots, d_n) \\
f &::= \backslash x [:: t] \rightarrow e \mid g \mid f_1 \cdot f_2 \mid f_1 \otimes f_2 \mid \text{id} \mid \Delta \mid \text{nullF} \mid \text{fst} \mid \text{snd} \mid \text{lft} \mid \text{rht} \mid \text{hash}(d) \\
g &::= \text{fstFun } f \mid \text{sndFun } f \mid \text{lftFun } f \mid \text{rhtFun } f
\end{aligned}$$

Fig. 9 Distributed data layout (DDL) type syntax

output array C , and shows (on the right) a map M partitioned along D_1 and only at the axis of D_2 . We use such node arrangements to describe data distributions in the sections that follow.

3.2 Distributed Data Layout Types

In our system, every high-level collection has a corresponding distributed collection type which, in addition to describing the data type, has extra parameters which specify how it should be distributed on the cluster. It is important to note that the user does not see these types, but the compiler uses them to plan the data distribution. The syntax for these *distributed data layout (DDL) types* is given in Fig. 9. In addition to standard type schemes these include a polyadic version of dependent Π -types that we call *dependent type schemes* and explain in Sect. 3.4. The DDL types dt extend types t with distributed arrays, maps, and lists (DArr , DMap , and DList). Additional type parameters include partition functions f , distribution modes m , and dimension identifiers d , which are described below. Partition functions can contain function generators g which are described in Sect. 3.5.

The DArr type formally describes how to store an array on a cluster. It takes a partition function f , an optional tuple of distribution modes m , and two tuples of dimension identifiers d_1 and d_2 . f is an actual (projection) function that consists of lambda terms from the input program and the operators listed under f in Fig. 9, and defined in Fig. 10. It identifies the dimensions of the array along which it should be partitioned. Dimension identifiers d_1 and d_2 are just tuples of type variables. d_1 has the same arity as f 's co-domain, and specifies over which dimensions of the cluster the partition dimensions should be distributed. d_2 specifies over which dimensions to mirror. Array partitions only exist at the zero-position position in any remaining dimensions. m is an optional tuple of distribution modes (blk or cyc ; default blk), with the same arity as d_1 ; it specifies a mode for each of the array dimensions returned by f . Here blk describes contiguous blocks of an array on successive nodes (e.g. $a[0:9]$ on n_0 , $a[10:19]$ on n_1), and cyc describes storing alternate elements on successive nodes (e.g. $a[0]$ on n_0 , $a[1]$ on n_1 , $a[5]$ on n_0 , where $|d_1| = 5$). For example, the arrays for our `mmul` implementation example have DDL types

$$\begin{array}{ll}
\beta \cdot \alpha & \Rightarrow \backslash x \rightarrow (\beta(\alpha \ x)) \\
\alpha \otimes \beta & \Rightarrow \backslash (x,y) \rightarrow (\alpha \ x, \ \beta \ y) \\
\text{id} & \Rightarrow \backslash x \rightarrow x \\
\Delta & \Rightarrow \backslash x \rightarrow (x,x) \\
\text{nullF} & \Rightarrow \backslash _ \rightarrow () \\
\text{fst} & \Rightarrow \backslash (x,y) \rightarrow x \\
\text{snd} & \Rightarrow \backslash (x,y) \rightarrow y \\
\text{lft} & \Rightarrow \text{fst} \cdot \text{fst} \otimes \text{fst} \cdot \text{snd} \\
\text{rht} & \Rightarrow \text{snd} \cdot \text{fst} \otimes \text{snd} \cdot \text{snd}
\end{array}$$
Fig. 10 Built-in functions

```

A :: DArr (Int,Int) Float fst blk D1 D2
B :: DArr (Int,Int) Float snd blk D2 D1
C :: DArr (Int,Int) Float id blk (D1,D2)

```

where A is 1D partitioned by row using `fst`, B by column using `snd`, and C is 2D partitioned by row and column using `id`, which returns the row and column of C. Similarly, DMaps describe how to store Maps on clusters. d_1 and d_2 work in the same way as for DArr, but f takes key-value pairs rather than indices, mapping them onto specific node indices in d_1 . f can use the function `hash (d)` which takes a tuple of dimension identifiers d , and returns a function from any value type to node indices for the dimensions in d . For example, the value M in Fig. 8 (right) is partitioned by z and so has type

```
DMap (Int,Int) Float (hash(D1) · snd) D1 ()
```

DList's parameters d_1 and d_2 work in the same way, but instead of a partition function, DLists just have a partition mode m .

Top-level scalars and lambda terms are always mirrored on all nodes in the cluster. For DArrs and DMaps, if the partition function is `nullF` the collection is not partitioned. The `·`-operator sequentially composes two functions, and the \otimes -operator composes pairwise, as defined in Fig. 10.

3.3 Distributed Function Types

For each high-level combinator (cf. Fig. 3), the compiler internally provides different functionally equivalent implementations that work for different data distributions. We use DDL types to characterize how these different implementations store their inputs and outputs. These types then allow us to infer concrete DDLs for programs that use these different implementations. For example, our `mmul` implementation's DDLs can be inferred using `eqJoinArr3`, `mapArrInv2`, and `groupReduceArr2`'s DDL types. The implementations and their types are hidden from the user; they only see the high-level combinators.

The DDL type schemes for some of these implementations are shown in Fig. 11, where different implementations of the same combinator are distinguished using suffix numbers. For example, `groupReduce1` locally groups and reduces the values stored

```

mapArrInv1 ::  $\Pi(f, \_, \_, \_) : (i \rightarrow j, (i, v) \rightarrow w, j \rightarrow i,$ 
  DArr i v (g · f) d m) -> DArr j w g d m
mapArrInv2 ::  $\Pi(\_, \_, f^{-1}, \_) : (i \rightarrow j, (i, v) \rightarrow w, j \rightarrow i,$ 
  DArr i v g d m) -> DArr j w (g · f-1) d m
eqJoinArr1a ::  $\Pi(f, g, \_, \_) : (i \rightarrow k, j \rightarrow k, \text{DArr } i \text{ v f d m},$ 
  DArr j w g d m) -> DArr (i, j) (v, w) (f · fst) d m
eqJoinArr1b ::  $\Pi(f, g, \_, \_) : (i \rightarrow k, j \rightarrow k, \text{DArr } i \text{ v f d m},$ 
  DArr j w g d m) -> DArr (i, j) (v, w) (g · snd) d m
eqJoinArr2 :: (i → k, j → k, DArr i v fstFun(f) d m,
  DArr j w nullF () (d, m)) -> DArr (i, j) (v, w) f d m
eqJoinArr3 :: (i → k, j → k, DArr i v fstFun(f) d1 (d2, m),
  DArr j w sndFun(f) d2 (d1, m)) -> DArr (i, j) (v, w) f (d1, d2) m
groupReduceArr1 :: (i → j, (i, v) → w, (w, w) → w,
  DArr i v f d1 m1) -> DArr j w id d2 m2
groupReduceArr2 ::  $\Pi(pf, \_, \_, \_) : (i \rightarrow j), (i, v) \rightarrow w, (w, w) \rightarrow w,$ 
  DArr i v pf d m) -> DArr j w id d m

map ::  $\Pi(f, \_) : ((i, v) \rightarrow (j, w),$ 
  DMap i v (g · f) d m) -> DMap j w g d m
eqJoin1a ::  $\Pi(f, g, \_, \_) : ((i, v) \rightarrow k, (j, w) \rightarrow k, \text{DMap } i \text{ v f d m},$ 
  DMap j w g d m) -> DMap (i, j) (v, w) (f · lft) d m
eqJoin1b ::  $\Pi(f, g, \_, \_) : ((i, v) \rightarrow k, (j, w) \rightarrow k, \text{DMap } i \text{ v f d m},$ 
  DMap j w g d m) -> DMap (i, j) (v, w) (g · rht) d m
eqJoin2 :: ((i, v) → k, (j, w) → k, DMap i v lftFun(f) d m,
  DMap j w nullF () (d, m)) -> DMap (i, j) (v, w) f d m
eqJoin3 :: ((i, v) → k, (j, v) → k, DMap i v lftFun(f) d1 (d2, m),
  DMap j w rhtFun(f) d2 (d1, m)) -> DMap (i, j) (v, w) f (d1, d2) m
allPairsa ::  $\Pi(f, \_) : ((i, v) \rightarrow k, \text{DMap } i \text{ v f d m}) \rightarrow$ 
  DMap (i, i) (v, v) (f · lft) d m
allPairsb ::  $\Pi(f, \_) : ((i, v) \rightarrow k, \text{DMap } i \text{ v f d m}) \rightarrow$ 
  DMap (i, i) (v, v) (f · rht) d m
groupReduce1 :: ((i, v) → j, (i, v) → w, (w, w) → w,
  DMap i v f d1 m1) -> DMap j w (hash(d) · fst) d2 m2
groupReduce2 ::  $\Pi(f, \_, \_, \_) : ((i, v) \rightarrow j, (i, v) \rightarrow w, (w, w) \rightarrow w,$ 
  DMap i v (hash(d) · f) d m) -> DMap j w (hash(d) · fst) d m
reduce :: ((k, v) → s, (s, s) → s, DMap k v f d m) -> s
union :: (DMap k v fst d m, DMap k v fst d m) -> DMap k v fst d m

zip :: (DList v cyc d m, DList w cyc d m) -> DList (v, w) cyc d m
mapList :: (v → w, DList v q d m) -> DList w q v m
reduceList :: ((v, v) → v, v, DList v q d m) -> v

redistArr :: DArr i v f1 d1 m1 -> DArr i v f2 d2 m2
redistMap :: DMap k v f1 d1 m -> DMap k v f2 d2 m
redistList :: DList v q1 d1 m1 -> DList v q2 d2 m2

```

Fig. 11 DDL types for combinator implementations

at each node, exchanges the results between nodes to co-locate by key, and then group-reduces again at each node. This implementation works no matter how the input is partitioned. In the DDL type we therefore use the universally quantified type variable f to specify that the input can be partitioned by any partition function. The output is always partitioned by key, which we specify by using `fst` in the return type.

In situations where collections are partitioned by multiple partition functions (e.g., the result of two aligned collections can be viewed as partitioned by either/both of the input's partition functions), we use multiple types for the same template, differentiated by alphabetic subscripts (e.g., `eqJoin1a/b`).

3.4 Dependent Type Schemes

In addition to classic type schemes, we also have a polyadic version of dependent Π -types *dfs*, similar to those used in dependent ML [44]. These are not full dependent types, so that we can keep our system decidable. They are similar to type schemes but the type variables are now rigidly bound to the members of the argument tuples at function applications. For example, $\Pi(f, _) : (x \rightarrow y, S\ f) \rightarrow T\ f$ is like a normal function type, except the runtime value of the first parameter, which is itself a function of type $x \rightarrow y$, is bound to the variable f , and then used in the other parameter's type $S\ f$, and the return type $T\ f$. Hence, rather than representing *any* value, such type variables are bound to the *actual* values of parameters at runtime, or more precisely, the AST terms that represent them. These variables can then be used in the input and output types. This allows us to place context-dependent constraints on data distributions, to specify when different combinator implementations can be used.

Our example `mmul` implementation uses `groupReduceArr2`. In contrast to `groupReduceArr1` and `groupReduce1`, `groupReduceArr2` and `groupReduce2`, `group-reduce` just once, locally at each node. To yield a valid result all the input values for a given group must be co-located on the same node. We specify this constraint using a Π -type. `groupReduce2`'s type

```
groupReduce2 ::  $\Pi(f, \_, \_, \_)$  :
  ((k1, v1) -> k2, (k1, v1) -> v2, (v2, v2) -> v2, DMap k1 v1 (hash(d)·f) d m)
  -> DMap k2 v2 (hash(d)·fst) d m
```

thus binds the value of its first parameter, the function that generates the result keys, to f . This f is then used to define the input map's partition function. All values for a given key produce the same hash, and will therefore be stored on the same node. `groupReduceArr2` uses the same technique. Array `R2` in our `mmul` example implementation is therefore constrained to be partitioned by $\backslash((a_i, a_j), (b_i, b_j)) \rightarrow (a_i, b_j)$.

`eqJoin1a/b` and `eqJoinArr1a/b` work in a similar way. Here, if we know that the values that yield a given key are co-located on the same node then no inter-node communication is necessary and local joins will suffice. To specify this we partition both inputs by their respective join-key projection functions f and g . So in `eqJoinArr1`, the output will be partitioned by $f \cdot \text{fst}$ and $g \cdot \text{snd}$. Therefore either the type labeled `eqJoinArr1a` or `eqJoinArr1b` can be used respectively, since both are valid.

We also use Π -types to propagate partitioning information between inputs and outputs for structure-preserving transformations, like `mapArrInv1`. Here, to ensure the output is partitioned by g , the input must be partitioned by g applied after the index transformer function f . In the other direction, if the inverse transformer function f^{-1}

is known, and the input is partitioned by g , then the output of `mapArrInv2` will be partitioned by g applied after f^{-1} . This forces the distribution of $R1$ in our `mmul` example to be $\backslash((ai, aj), (bi, bj)) \rightarrow (ai, bj) \cdot id$. Both these implementations work the same way, but they propagate distribution information in different directions.

3.5 Function Generators

By expressing output partition functions as compositions of input ones, our type schemes allow us to carry DDL information *forwards* (from inputs to outputs) through the programs. However, the analysis also needs to work *backwards*, in order to automatically find input partition functions which combine to yield a given output partitioning. For unary combinators like `mapArrInv1` we can use the existing DDL information, but for binary combinators like `eqJoinArr` we need to *decompose* output partition functions. For example, our `mmul` example implementation needs DDLs for `eqJoinArr3`'s inputs A and B that will make the output $R1$ partitioned by $\backslash((ai, aj), (bi, bj)) \rightarrow (ai, bj) \cdot id$. For this we use *function generators* (cf. Fig. 9, `fstFun` to `rhtFun`).

Function generators analyze at compile-time the abstract syntax trees (ASTs) of their arguments (which are partition functions), and derive new partition functions that depend only on a subset of the inputs. If no such AST terms exist then the `nullF` function $\backslash_ \rightarrow ()$ is generated. For instance, `fstFun` takes a function with a domain (x, y) , and generates a function with domain x by retaining all the parts of the AST that depend only on x , and throwing away those terms that also depend on y ; `sndFun` works accordingly on the y domain. For example, given a partition function $f = \backslash(a, (b, c)) \rightarrow (a, c)$, `fstFun(f)` equals $\backslash a \rightarrow a$ and `sndFun(f)` equals $\backslash(b, c) \rightarrow c$. These can be used together to give an output partitioning equivalent to a partition function that subsumes the original. For example, `eqJoinArr3` uses function generators so that it can partition its output by any f . To ensure that the output is partitioned by f , the inputs must be partitioned by `fstFun(f)` and `sndFun(f)` along dimensions $d1$ and $d2$. This gives us the required DDLs for our `mmul` implementation.

3.6 Local Data Layouts

In addition to distribution information, we also use DDL types to specify how to store collections in memory. We have omitted this from the types in Fig. 11 to simplify the presentation, but briefly sketch the mechanism here.

Multidimensional arrays can be stored different ways in memory, e.g., in row-major or in column-major order. We specify the layout of an n -dimensional `DArray` by adding a *layout function* to the type. This maps the array's indices to an n -tuple, whose order dictates how to order the indices in memory. Hence, $\backslash(x, y) \rightarrow (x, y)$ means row-major order, and $\backslash(x, y) \rightarrow (y, x)$ means column-major. This can express very similar constraints to partition functions. For example, `groupReduceArr2` has the full type

$$\Pi(f, _, _, _) : \\ (i1 \rightarrow i2, (i1, v1) \rightarrow v2, (v2, v2) \rightarrow v2, \text{DArr } i1 \ v1 \text{ pf } (f \otimes \text{rem}(f)) \cdot \Delta) \ d \ m \\ \rightarrow \text{DArr } i2 \ v2 \text{ id id } d \ m$$

where `rem` is a function generator that takes a function `f` and returns another function that is the complement of `f`, i.e., projects all the parts of the input tuple that `f` does not already project. Here, we force the first indices to be the group's key indices (projected by `f`), followed by the rest `rem(f)`. For our `mmul` example this means storing matrix `B` in transposed form. This improves cache-line usage by ensuring that elements in the same group are adjacent in memory. We use the same technique to specify the indexing schemas of `DMaps`. We also use flags in the types to specify the local storage modes (e.g. hash table/binary tree/sorted vector/stream of values) for `DMap` and `DList`.

3.7 Extensibility

A major strength of our approach is its extensibility. New combinators can be added simply by declaring their functional types, and the DDL types and back-end templates of their implementations. Furthermore, the system can be extended with new types without altering the underlying framework. For example, collections like spatially indexed maps (`Spatial`), or trees (`Tree`), and their distributed equivalents (`DSpatial` and `DTree`), can be added by simply adding them to a configuration file. `DArrs` can also be extended to support block-cyclic distributions, and ghosting, by adding more function parameters to specify the offsets and bounds. In fact irregular (i.e., master/slave) distribution algorithms can also be modeled in a similar way to the “stream of values” (`Stm`) local storage modes, where type variables identify irregular partition mappings, populated at runtime. This extensibility is a clear benefit of this approach over collection-specific techniques.

4 Automatic DDL Planning and Code Generation

4.1 DDL Type Inference

Now that we have characterized the DDLs of combinator implementations, we can search for distributed implementations of input programs by exploring different combinations of combinator implementations. Here each combinator application in a program can use a different implementation. We use type inference to find a valid assignment of data distributions for a given choice of combinator implementations, if one exists. For a concrete example, see the DDL types inferred for our running `mmul` example in Sect. 5.

Figure 12 shows some of the typing rules for our DDL type system, omitting rules for literals, and identifier patterns in let- and lambda-expressions for brevity. The rules derive typing judgments for programs. The judgments also include a set of constraints (denoted by “ $\hookrightarrow C$ ”) which must be satisfiable for the derived judgment to be valid. The rules closely mirror the standard polymorphic lambda calculus with conditionals and tuples [30], apart from `DAPP` which deals with dependent type schemes. `DAPP`

$$\begin{array}{c}
\text{(DVAR)} \frac{x : T \in \Gamma \quad T' = \text{instScheme}(T)}{\Gamma \vdash x : T' \hookrightarrow \{\}} \quad \frac{\Gamma \vdash e_1 : T_1 \hookrightarrow C_1 \quad \Gamma \vdash e_2 : T_2 \hookrightarrow C_2 \quad \Gamma \vdash e_3 : T_3 \hookrightarrow C_3}{C' = C_{1,2,3} \cup \{T_1 = \text{Bool}, T_2 = T_3\}} \text{(DIF)} \\
\text{(DTUP)} \frac{\Gamma \vdash e_1 : T_1 \hookrightarrow C_1 \dots \Gamma \vdash e_n : T_n \hookrightarrow C_n}{C' = C_1 \cup \dots \cup C_n} \quad \frac{\Gamma \vdash e_1 : T_1 \quad \Gamma, x : T_1 \vdash e_2 : T_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : T_2 \hookrightarrow \{\}} \text{(DLET)} \\
\text{(DABS)} \frac{\text{X fresh var} \quad \Gamma, x : X \vdash e_1 : T \hookrightarrow C}{\Gamma \vdash \lambda x. e_1 : X \rightarrow T \hookrightarrow C} \quad \frac{\text{X fresh var} \quad \Gamma \vdash e_1 : T_1 \hookrightarrow C_1 \quad \Gamma \vdash e_2 : T_2 \hookrightarrow C_2}{C' = C_{1,2} \cup \{T_1 = T_2 \rightarrow X\} \cup \text{gdc}(T_1, e_2)} \text{(DAPP)}
\end{array}$$

Fig. 12 DDL type rules

applies the *gdc* (generate dependent constraints) function to return additional constraints which marry up any Π -bound type variables (in T_1), with their respective AST terms at function applications (e_2), so uses of these variables must match the AST terms specified.

Our type inference algorithm extends a constraint based version of Damas and Milner's Algorithm W [14] in [30]. This version traverses the AST to construct initial types, and then uses Robinson's unification algorithm [33] to find substitutions that unify the constraints. Our version extends the one in [30] to support tuples, deal with dependent type schemes, and unify functions embedded in the types. To handle dependent type schemes, we extend the function application case of the initial AST pass to instantiate any Π -bound variables using *gdc*. To unify functions in the types, we need to make two alterations to the unification algorithm, to deal with constraints involving functions, and to change the order in which it solves constraints.

Testing for function equality is undecidable in the general case. Therefore, in order to make our type system decidable we adopt a sound but incomplete approximate solution to unify partition functions. Our current solution tries to unify them syntactically, and if this fails normalizes them and tests for syntactic equality. Normalization replaces all functions and function compositions with concrete lambda abstractions, and then simplifies the resulting lambda abstractions using Flocc's evaluation rules. This works in many situations, including the examples in this paper. However, we are working on a more nuanced approach which relies on the fact that almost all of these functions are projection functions.

Then, unlike standard syntactic unification, we cannot just stop unifying when a constraint does not unify immediately. In our system, constraints between partition functions may fail initially, but could succeed later on, since they might by substitution become terms which will equate when normalized. Similarly, function generators can only be applied when their argument functions have become concrete. To address this problem, we make our unification algorithm iterative. In each iteration we try to unify as many constraints as possible, and postpone constraints that currently fail until the next iteration. The algorithm succeeds when all constraints have been solved, and fails if no more progress is made during an iteration i.e., no more constraints are solved, and so the unification is stuck. If there exists a sequence of unifications that satisfies

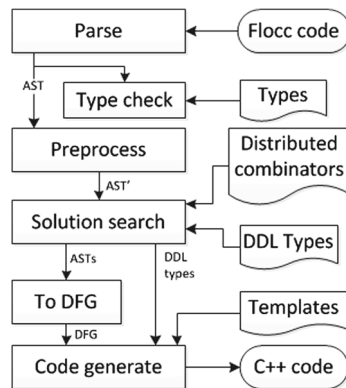


Fig. 13 Compiler architecture

the constraints this algorithm will find it. These modifications allow us to lift concrete functions into the types and solve equations involving them via unification.

4.2 Distribution Search

Now that we can synthesize data distributions for programs that use some choice of combinator implementations for their combinator applications, we can search through different choices of combinator implementations to explore different data distribution plans. Each combinator has a list of combinator implementations, with different communication patterns, and therefore different performance characteristics. The combinator implementations to use for the combinator applications in the examples in Sect. 5 were chosen manually, but the process can be automated. Our implementation does this via an exhaustive depth-first search to try different implementations for each combinator application in a program. This search prioritizes faster combinator implementations (based on an a priori ordering of implementations by descending performance) to try and find the optimal solution as soon as possible. This does not yet consider plans involving redistribution functions, so if the DDL types for a plan do not type check, the plan is discarded. However, possible redistributions could also be enumerated exhaustively. This distribution search technique is sufficient for the example programs and corresponding implementations presented in this paper. However, we are working on a better solution that automatically inserts redistributions at appropriate locations, and uses more sophisticated goal-directed search heuristics.

4.3 Code Generation

We have implemented a prototype Flocc code generator in Haskell that produces MPI implementations in C++. The high-level compilation process is illustrated in Fig. 13. The generator parses the input program, and performs type inference for the functional

types. It then preprocesses the AST to expand all tuple-typed variables to tuples of variables, and to replace all function-typed variables with the lambdas they are bound to. This ensures that all λ -bound lambdas are directly available at function application expressions. Then it expands lambda term applications, so that different applications can have different DDLs.

At this point the generator loads the lists of combinator implementations and their DDL types, and then uses the technique described in Sect. 4.2 to find possible distributed solutions, with their corresponding DDL types. For a chosen implementation, it converts the AST into a data flow graph (DFG), replacing all literals, tuple expressions and function applications etc. with nodes, and `let`-bindings with edges.

The generator then traverses the DFG, performing dead code elimination and applying expression templates for each library function/distributed combinator application. Templates take their function application's concrete DDL types, and their input and output nodes. They output blocks of C++ to perform the corresponding operation, where blocks may be nested in, and consume values from loops, for combinator implementations that take streams of values. Most lambda-expressions are inlined by the code generator, apart from those that are passed as custom reduction operators to `MPI::Reduce` and `alike`.

5 Example Derivations

We now discuss some generated DDL plans for the example programs. We list the distributed implementation used for each combinator application, and the DDL types that result. We use `IP` as shorthand for `(Int, Int)`.

5.1 Matrix Multiplication—Partition for `groupReduceArr`

The first solution is driven by an optimized partition for the group-reduce operation, which yields the usual implementation of matrix multiplication. It uses `groupReduceArr2` to avoid inter-node communication by requiring `R2` to be partitioned using its index projection function, which projects `A`'s row and `B`'s column from the array's indices. `R1`, `mapArrInv2`'s input, must thus be partitioned using this function too. `eqJoinArr3` satisfies this constraint, by partitioning `A` by row (using `fst`) along one dimension, and `B` by column (using `snd`) along an orthogonal one, and then mirroring both along their respective orthogonal dimensions. This yields a 2D grid, enumerating all combinations of partitions, i.e., the Cartesian product.

```
A :: DArr IP Float fstFun(\((ai,aj),(bi,bj))->(ai,bj)).id d1 (d2,m)
  = DArr IP Float fst d1 (d2,m)
B :: DArr IP Float sndFun(\((ai,aj),(bi,bj))->(ai,bj)).id d2 (d1,m)
  = DArr IP Float snd d2 (d1,m)
R1 :: DArr (IP,IP) (Float,Float) (\((ai,aj),(bi,bj))->(ai,bj)).id (d1,d2) m
R2 :: DArr (IP,IP) Float \((ai,aj),(bi,bj))->(ai,bj) (d1,d2) m
C :: DArr IP Float id (d1,d2) m
```

5.2 Matrix Multiplication—Partition for eqJoinArr

The next solution is more unusual. It uses `eqJoinArr1a` to avoid mirroring A and B, by aligning them to co-locate partitions with common key values. A and B are partitioned by column (`snd`) and row (`fst`) respectively, and thus the join result R1 is partitioned by both the column of A (`snd·fst`) and the row of B (`fst·snd`), although we only infer the former property since we use `eqJoinArr1a` rather than `b.mapArrInv2` then constrains R2 to have this partitioning as well, and so `groupReduceArr2` cannot be used without inserting a redistribution. Instead `groupReduceArr1` is used, as it accepts any input partitioning, at the expense of having to exchange intermediates between nodes. With dense matrices R1 will be much larger than A and B, and so this solution will perform poorly, but if A and B are large and sufficiently sparse, exchanging the intermediates could outperform mirroring.

```
A :: DArr IP Float snd d m
B :: DArr IP Float fst d m
R1 :: DArr (IP,IP) (Float, Float) (snd·fst) d m
R2 :: DArr (IP,IP) Float (snd·fst)·id d m
C :: DArr IP Float id d m
```

5.3 Matrix Multiplication: Mirror One Matrix

This also uses `groupReduceArr2`, but unlike the first solution, it uses `eqJoinArr2` to give the required data distribution. This partitions A across all the nodes in `d`, and mirrors B on all of them. This solution is better than the first solution if B is much smaller than A so that it is less expensive to replicate all of B than partitions of A.

```
A :: DArr IP Float fstFun(\((ai,aj),(bi,bj))->(ai,bj))·id d m = fst d m
B :: DArr IP Float nullF () (d,m)
R1 :: DArr (IP,IP) (Float, Float) (\((ai,aj),(bi,bj))->(ai,bj))·id d m
R2 :: DArr (IP,IP) Float \((ai,aj),(bi,bj))->(ai,bj) d m
C :: DArr IP Float id d m
```

5.4 Histogram: Group Locally Before Exchange

The first Histogram solution uses `groupReduce1` so that the input D does not have to be partitioned by its `Float` value. The output R is partitioned by bucket `id` (`fst`), and a concrete function must still be chosen for `f`. Since in this example the type `k` is still abstract the two possibilities for `f` are `fst` and `snd`. In a concrete program `k` is a concrete type and so `f` could have more possible values. For `f=fst`, D is partitioned by `hash(d)·fst`, which is a valid solution. However, for `f=snd`, D is partitioned by `hash(d)·\v->toInt(Float.* (v,i))`, which is not valid, as it references `i` before it has been declared, and so this solution is discarded by the compiler.

```
D :: Map k Float hash(d)·f·(id·fst⊗\(_,v)->toInt(Float.* (v,i)))·Δ d m
D' :: Map k Int hash(d)·f d m
R :: Map Int Int hash(d)·fst d m
```

5.5 Histogram: Exchange Before Group

The second plan uses `groupReduce2` by repartitioning D' by `hash(d)·snd`. However, this plan will be sub-optimal unless the number of buckets is close to the number of data points, since the partitions of D' that `redistMap` communicates will be larger than the results of the local group-reduces that `groupReduce1` communicates.

```
D :: DMap k Float hash(d)·f·(id·fst⊗\(_,v) -> toInt(Float.*(v,i)))·Δ d m
D' :: DMap k Int hash(d)·snd d m
R :: DMap Int Int hash(d)·fst d m
```

5.6 Dot Product: Cyclic Distribution

In this plan A and B are aligned since they both have cyclic distributions over the same dimension d , so `zip` can be used without any communication. However, if `dotp` was used in a context where A or B had a different distribution, `redistList` would be automatically used to convert it into the required cyclic distribution.

```
A :: DList Float cyc d (); B :: DList Float cyc d ();
AB :: DList Float cyc d ()
```

5.7 Performance of Generated Code

We have generated implementations of the running examples using our prototype tool and compared them (cf. Fig. 14) to PLINQ [16] (using all cores on a 64-bit/quad-core/2.67 GHz workstation with 12GB memory), and hand-coded MPI implementations (averaged over 1,2,3,4,8,9,16,32 nodes on a 12k-core/16 × 2.67 GHz core per node cluster, with 4GB memory per node, and InfiniBand interconnect). We compare with PLINQ, even though it does not support distributed memory, because it also auto-parallelizes programs written in a functional language inspired by relational algebra, and so is the most closely related approach. All Flocc programs drastically outperformed the PLINQ implementations, most likely because PLINQ chose poor job partitionings, does not inline lambdas, and does not distinguish between, and so cannot optimize for, arrays, lists, and maps. The generated programs also came within 51 % of the speed of manually implemented textbook-style MPI versions. The dot product and simple linear regression compiled with ICC, and the standard deviation, were nearly identical to the hand-coded versions. The linear regression when compiled with ICC was 39 % slower because it used an array of structs, rather than a struct of arrays. The histogram was 27 % slower, because it used a hash table, and the comparison used an array. The matrix multiply was 6× faster than the hand written code when compiled with GCC, since our tool optimized the layout of B to be column-major, but was 51 % slower when compiled with ICC. This is because the manual version iterates over global arrays, and ICC seems to optimize for this case. An additional Global array storage mode (see Sect. 3.6) and corresponding templates, would cater for this situation. The results could be improved further by optimizing the existing and adding additional back-end templates, but they are sufficient to indicate that the approach is viable in practice.

Program	PLINQ			Manual MPI		
	Speedup	Compiler	Data	Speedup	Compiler	Data
Dot product	4.96×	gcc -Ofast	2.2GB	0.99×	icc -O3	4.5GB
Simple linear regression	137×	gcc -Ofast	3GB	0.89×	icc -O3	3GB
Standard deviation	98.6×	gcc -Ofast	3GB	0.61×	gcc -O3	3GB
				0.88×	icc -O3	3GB
				1.00×	gcc -O3	3GB
Histogram	31.5×	gcc -Ofast	32MB	0.73×	icc -O3	8GB
Matrix multiply	342×	gcc -Ofast	1.4MB	6.14×	gcc -O3	140MB
				0.49×	icc -O3	140MB

Fig. 14 Performance comparison of Flocc generated code versus others

6 Related Work

Traditionally most high-performance computing (HPC) applications were programmed with MPI [41] or High Performance Fortran (HPF) [26]. MPI specifies message passing primitives for programming clusters. It is very versatile, but it has no automatic data layout, and requires very verbose, hard-to-debug implementations. We use it as target language in our work. HPF extends Fortran 95 with directives to specify how to distribute arrays. It supports a limited number of data-parallel operations for flat multi-dimensional arrays. Distribution directives were originally specified manually but a tool was developed to optimize them for different programs [24]. Similar techniques were developed to find data distributions for programs with array sections [13] and affine loop-nests [1,3,5]. However our approach is more general, supporting collections other than arrays, and an extensible set of data-parallel combinators.

MapReduce [15] and Hadoop [43] are frameworks for performing aggregations on huge datasets, hosted on large-scale clusters. They primarily rely on a *map* function that projects key-value pairs from a dataset, and a *reduce* function that aggregates a sorted list of values for each key. They handle all communication, scheduling, and failure recovery, and so greatly simplify data-parallel programming. However, they have a single restricted programming model, and a single distributed implementation which is not suitable for all applications. For example, one investigation showed a Hadoop K-means clustering program performed 20× slower than an MPI version [17]. For this reason numerous alternates have been suggested to allow, e.g., iteration [6,19], different file types [7,20], accepting multiple inputs [45], removal of intermediate files [17], and supporting different architectures [11,31]. However, each of these also has a (different) single programming model, and implementation, specialized for one particular task, and so can still suffer from the same inflexibility as the original MapReduce. By contrast, our approach has many input combinators, with many possible distributed implementations, that can be combined in numerous ways, to yield implementations optimized for specific applications. In particular, our technique supports iteration, multiple collection types, and structured data distributions.

Parallel databases can also be used for some distributed data-parallel tasks. Like Flocc programs, parallel SQL query plans [9] are synthesized by enumerating differ-

ent combinations of plan operators to minimize the overall cost [39]. SQL queries are also based on relational algebra, though they have a weak type system, no support for array-based computation, and cannot be extended with new operators. Furthermore, parallel databases typically do not generate standalone code, and the distributed schemas must be designed manually, though a tool to assist with this has been proposed [28]. DryadLINQ [23] is a framework for cluster computation in .NET languages, that also takes SQL queries, optimizing them at runtime to query large distributed datasets. However it suffers from many of the same problems as parallel databases.

Chapel [8] is a Partitioned Global Address Space (PGAS) language for HPC that evolved from ZPL [25], a language for working with multidimensional arrays, which featured named index sets called *regions*, so that arrays that shared a region were aligned/distributed in the same way. Chapel improves on ZPL and HPF by supporting data-parallel operations on maps and graphs as well as arrays. It also includes some built in data distributions for these types, but these must still be chosen by the programmer. X10 [10] is similar, although it only supports distributed arrays and runs on the Java VM. Fortress [42] was never fully implemented and is now dormant.

A number of functional data-parallel languages have been developed that target shared memory parallelism. PLINQ [16] is like DryadLINQ for multicores. NESL [4] specialized in nested data-parallel vector operations on vector machines. Data Parallel Haskell [29] is an extension to Haskell based on NESL, but for modern multi-cores. Single Assignment C (SAC) [22] supports n-dimensional array computations, with an impressive implementation that has outperformed Fortran in some cases. Some recent work on SAC has also used a type system to reason about local array layouts, to detect when they can be transformed to permit vectorization for SIMD instructions [40]. However none of these currently support distributed memory data parallelism, or suggest how such support could be implemented.

One functional language that does target distributed memory architectures is Sisal [21]. Sisal supports data-parallel *for*-expressions, which range over index spaces accessing array elements, generating intermediate values, and aggregating them. Although it only included this one data-parallel construct, it did synthesize distributed memory implementations and seek to optimize them by collocating tasks that would perform a lot of intercommunication [36]. However, Sisal did not support structured data partitionings, alignments, or data replication etc, and so was very limited in its ability to optimize data layout. Furthermore, it only supported 1D arrays.

Finally, our data-parallel combinator implementations are very similar to *algorithmic skeletons* [12]. Skeletons encode patterns of parallel processing and communication, which can be composed and parameterized with concrete functions, leading to networks of processes that perform a parallel task. For example, skeletons have been implemented as C++ templates to allow users to quickly trial different process networks on the CELL processor [34]. They have also been used in a parallelizing SML compiler to implement list-combinators by synthesizing process networks during an AST pass [37], and in an image processing DSL which avoids redundant communication steps using a technique similar to our automatic redistribution insertion [38]. However, none of these approaches support different data partitionings, multiple collection types, or automatically explore different data distributions.

```

let triEnum = (\E :: Map (Int,Int) () ->
  -- find degree of all vertices
  let D1 = groupReduce (fst.fst, \_>1, addi, E) in
  let D2 = groupReduce (snd.fst, \_>1, addi, E) in
  let D = map (\(k,v)->(k,addi v), eqJoin (fst,fst,D1,D2)) in
  -- identify edges by vertex with lower degree
  let E1 = eqJoin (fst, fst, E, D) in
  let E2 = eqJoin (snd.fst, fst, E1, D) in
  let E3 = map (\(((_,v1),v2),((_,d1),d2)) ->
    (if lti (d1,d2) then (v1,v2) else (v2,v1), ()), E2) in
  -- for each edge, find all angles
  let A = allPairs (fst, E3) in
  -- for each angle, see if it is closed
  let T1 = eqJoin (\(((a,_),(_ ,b)),_)->(a,b), fst, A, E3) in
  let T2 = eqJoin (\(((a,_),(_ ,b)),_)->(b,a), fst, A, E3) in
  map (\((((v1,_),_),((v2,v3)),_)-> ((v1,v2,v3),()), union (T1,T2)))

```

Fig. 15 Triangle enumeration (MinBucket algorithm)

7 Conclusions and Future Work

7.1 Conclusions

Existing languages for data-parallel programming rarely target distributed-memory architectures, and those that do are restricted to a fixed distribution model (MapReduce), and only support a limited set of operators (SQL/LINQ/HPF). In this paper we have presented a more general approach, where distributed-memory implementations are automatically synthesized from data-parallel programs written in Flocc, a high-level functional core language. To our knowledge this is the first approach that captures data distribution as a typing problem. In particular, we formalized distributed data layouts by polymorphic dependent type schemes and used a variant of the standard Damas–Milner type inference algorithm to search for different DDL plans in a type-directed way.

Unlike similar work, our approach supports multiple collection types (i.e., arrays, maps, and lists) and thus works for a wide variety of programs (cf. Fig. 7), and can easily be extended with more data types, data distributions, and data-parallel operators, without changing the core framework. Our approach can boost programmer productivity and program reliability through the conciseness of input programs (cf. Fig. 7), fully automatic generation of distribution plans and code, and the reduced number of possible bugs compared to low level languages (i.e. no pointers/explicit message passing). Finally, initial performance results (cf. Fig. 14) are substantially better than PLINQ, a similar tool for multi-cores, and are close to manual MPI implementations, indicating that the approach is viable in practice.

7.2 Future Work

We are currently optimizing and implementing more back-end templates for our code generator. In addition, there are several interesting fundamental research directions. First, we are developing more efficient goal-directed searches using cost estimates and

performance feedback for our data distribution planning. Second, we are developing a refined notion of function comparison so that our distribution search algorithm can find solutions with non-trivial partition function equalities. For example, Fig. 15 shows an algorithm for enumerating all triangles in a graph. The constraints caused by the `eqJoins` and `union` at the end of this example preclude the use of local `eqJoins` under syntactic equality. However, we work towards a more nuanced system that finds the most general unifier of two partition functions, and so permits this. Third, since arrays are “just” maps with dense integer domains we plan to detect such maps in input programs, and to infer their bounds and strides. This can then, for example, be used to derive dense and sparse matrix algorithms from the same high-level programs using maps.

Additional Information. Additional information, including code for the examples in Figs. 7 and 14 are available at <http://www.flocc.net/hlpp14/>, and in the first author’s PhD thesis [2].

References

1. Anderson, J.M., Lam, M. S.: Global optimizations for parallelism and locality on scalable parallel machines. In: PLDI ’93, pp. 112–125 (1993)
2. Aubrey-Jones, T.: Synthesizing imperative distributed-memory implementations from functional data-parallel programs. PhD thesis, submitted at University of Southampton, UK (2015)
3. Bixby, R.E., Kennedy, K., Kremer, U.: Automatic data layout using 0–1 integer programming. In: IFIP Trans ’94, pp. 111–122 (1994)
4. Blelloch, G., Hardwick, J., Chatterjee, S., Sipelstein, J., Zagha, M.: Implementation of a portable nested data-parallel language. In: PPOPP ’93, pp. 102–111 (1993)
5. Bondhugula, U.: Compiling affine loop nests for distributed-memory parallel architectures. In: SC ’13, pp. 1–12 (2013)
6. Bu, Y., Howe, B., Balazinska, M., Ernst, M. D.: Haloop: efficient iterative data processing on large clusters. In: PVLDB ’10, pp. 285–296 (2010)
7. Buck, J.B., Watkins, N., LeFevre, J., Ioannidou, K., Maltzahn, C., Polyzotis, N., Brandt, S.: SciHadoop: array-based query processing in Hadoop. In: SC ’11, pp. 1–11 (2011)
8. Chamberlain, B., Callahan, D., Zima, H.: Parallel programmability and the Chapel language. In: IJHPCA ’07, pp. 291–312 (2007)
9. Chamberlin, D., Boyce, R.: Sequel: A structured english query language. In: SIGFIDET ’74, pp. 249–264 (1974)
10. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioğlu, K., von Praun, C., Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing. In: OOPSLA ’05, pp. 519–538 (2005)
11. Chen, R., Chen, H., Zang, B.: Tiled-mapreduce: optimizing resource usages of data-parallel applications on multicore with tiling. In: PACT ’10, pp. 523–534 (2010)
12. Cole, M.: Algorithmic Skeletons: Structured Management of Parallel Computation. MIT Press, Cambridge (1991)
13. Chatterjee, S., Gilbert, J.R., Schreiber, R., Teng, S.-H.: Automatic array alignment in data-parallel programs. In: POPL ’93, pp. 16–28 (1993)
14. Damas, L., Milner, R.: Principal type-schemes for functional programs. In: POPL ’82, pp. 207–212 (1982)
15. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. In: OSDI ’04 , USENIX (2004)
16. Duffy, J., Essey, E.: Parallel LINQ: running queries on multi-core processors. In: MSDN Magazine ’07, pp. 70–78 (2007)

17. Ekanayake, J., Pallickara, S., Fox, G.: Mapreduce for data intensive scientific analyses. In: *eScience '08*, pp. 277–284 (2008)
18. Ekanayake, J., Gunarathne, T., Fox, G., Balkir, A., Poulain, C., Araujo, N., Barga, R.: DryadLINQ for scientific analyses. In: *e-Science '09*, pp. 329–336 (2009)
19. Ekanayake, J., Li, H., Zhang, B., Gunarathne, T., Bae, S.-H., Qiu, J., Fox, G.: Twister: a runtime for iterative mapreduce. In: *HPDC '10*, pp. 810–818 (2010)
20. Fadika, Z., Dede, E., Govindaraju, M., Ramakrishnan, L.: Mariane: Mapreduce implementation adapted for hpc environments. In: *GRID '11*, pp. 82–89 (2011)
21. Feo, J., Cann, D., Oldehoeft, R.: A report on the Sisal language project. *JPDC* **10**(4), 349–366 (1990)
22. Grelck, C.: Shared memory multiprocessor support for functional array processing in SAC. *JFP* **15**(03), 353–401 (2005)
23. Isard, M., Yu, Y.: Distributed data-parallel computing using a high-level programming language. In: *SIGMOD '09*, pp. 987–994 (2009)
24. Kennedy, K., Kremer, U.: Automatic data layout for high performance fortran. In: *Supercomputing '95* (1995)
25. Lin, C., Snyder, L.: Zpl: an array sublanguage. In: *LCPC '94*, LNCS 768, pp. 96–114 (1994)
26. Loveman, D.: High performance fortran. *PDS* **1**(1), 25–42 (1993)
27. Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A.: Pig latin: a not-so-foreign language for data processing. In: *SIGMOD '08*, pp. 1099–1110 (2008)
28. Papadomanolakis, S., Ailamaki, A.: Autopart: automating schema design for large scientific databases using data partitioning. In: *SSDBM '04*, pp. 383–392 (2004)
29. Peyton Jones, S.: Harnessing the multicores: nested data parallelism in haskell. In: *APLAS '08*, LNCS 5356, pp. 138–138 (2008)
30. Pierce, B.: *Types and Programming Languages*. MIT Press, Cambridge (2002)
31. Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G., Kozyrakis, C.: Evaluating mapreduce for multi-core and multiprocessor systems. In: *HPCA '07*, pp. 13–24 (2007)
32. Reichenbach, C., Smaragdakis, Y., Immerman, N.: PQL: a purely-declarative java extension for parallel programming. In: *ECOOP '12*, LNCS 7313, pp. 53–78 (2012)
33. Robinson, J.A.: A machine-oriented logic based on the resolution principle. *J. ACM* **12**(1), 23–41 (1965)
34. Saidani, T., Falcou, J., Tadonki, C., Lacassagne, L., Etiemble, D.: Algorithmic skeletons within an embedded domain specific language for the cell processor. In: *PACT '09*, pp. 67–76 (2009)
35. Sarkar, V., Cann, D.: Posc - a partitioning and optimizing sisal compiler. In: *ICS '90*, pp. 148–164 (1990)
36. Sarkar, V., Hennessy, J.: Compile-time partitioning and scheduling of parallel programs. In: *CC '86*, pp. 17–26 (1986)
37. Scaife, N., Horiguchi, S., Michaelson, G., Bristow, P.: A parallel sml compiler based on algorithmic skeletons. *JFP* **15**, 615–650 (2005)
38. Seinstra, F., Koelma, D., Bagdanov, A.: Finite state machine-based optimization of data parallel regular domain problems applied in low-level image processing. *TPDS* **15**(10), 865–877 (2004)
39. Selinger, P.G., Astrahan, M.M., Chamberlin, D.D., Lorie, R.A., Price, T.G.: Access path selection in a relational database management system. In: *SIGMOD '79*, pp. 23–34 (1979)
40. Sinkarovs, A., Scholz, S.-B.: Semantics-preserving data layout transformations for improved vectorisation. In: *FHPC '13*, pp. 59–70 (2013)
41. Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J.: *MPI—The Complete Reference*. The MIT Press, Cambridge (1996)
42. Weiland, M.: Chapel, Fortress and X10: novel languages for hpc. In: Technical report, HPCx Consortium, University of Edinburgh, Oct 2007
43. White, T.: *Hadoop: The Definitive Guide* (2010)
44. Xi, H.: Dependent ML: an approach to practical programming with dependent types. *JFP* **17**, 215–286 (2007)
45. Yang, H.-C., Dasdan, A., Hsiao, R.-L., Parker, D.S.: Map-reduce-merge: simplified relational data processing on large clusters. In: *SIGMOD '07*, pp. 1029–1040 (2007)