On the Distributed Implementation of Aggregate Data Structures by Program Transformation

Gabriele Keller[1] and Manuel M. T. Chakravarty[2]

[1] Fachbereich Informatik, Technische Universität Berlin, Germany
keller@cs.tu-berlin.de
www.cs.tu-berlin.de/~keller/
[2] Inst. of Inform. Sciences and Electronics, University of Tsukuba, Japan
chak@is.tsukuba.ac.jp
www.score.is.tsukuba.ac.jp/~chak/

Abstract. A critical component of many data-parallel programming languages are operations that manipulate *aggregate* data structures as a whole—this includes Fortran 90, Nesl, and languages based on BMF. These operations are commonly implemented by a library whose routines operate on a distributed representation of the aggregate structure; the compiler merely generates the control code invoking the library routines and all machine-dependent code is encapsulated in the library. While this approach is convenient, we argue that by breaking the abstraction enforced by the library and by presenting some of internals in the form of a new intermediate language to the compiler back-end, we can optimize on all levels of the memory hierarchy and achieve more flexible data distribution. The new intermediate language allows us to present these optimisations elegantly as program transformations. We report on first results obtained by our approach in the implementation of nested data parallelism on distributed-memory machines.

1 Introduction

A *collection-oriented* programming style is characterized by the use of operations on *aggregate* data structures (such as, arrays, vectors, lists, or sets) that manipulate the structure as a whole [21]. In parallel collection-oriented languages, a basic set of such operations is provided, where each operation has a straight-forward parallel semantics, and more complex parallel computations are realized by composition of the basic operations. Typically, the basic operations include elementwise arithmetic and logic operations, reductions, prescans, re-ordering operations, and so forth. We find collection-oriented operations in languages, such as, Fortran 90/95 [18], Nesl [1], and GoldFISh [15] as well as parallel languages based on the Bird-Meertens Formalism (BMF) and the principle of algorithmic skeletons [9, 6]. Usually, higher-order operations on the aggregate structure are provided, e.g., apply-to-each in Nesl; `FORALL` in Fortran 90/95[1]; and map, filter, and so on in BMF. Collection-oriented parallelism supports clear code, the use of program transformation for both program development and compilation, and simplifies the definition of cost models [23, 1].

---

[1] Fortran's `FORALL` loops are index-based, which complicates the implementation in comparison to, e.g., Nesl's apply-to-each, but still, many of the basic principles are the same [8].

The aggregate data structures of collection-oriented languages are often implemented by a library whose routines are in functionality close to the basic operations of the source language. Consequently, the library is both high-level and complex, but carefully optimized for the targeted parallel hardware; furthermore, it encapsulates most machine dependencies of the implementation. We call this the *library approach*—for Nesl's implementation, see [4]; Fortran 90/95's collection-oriented operations are usually directly realized by a library, and even, `FORALL` loops can benefit from a collection-oriented implementation [8]. The library approach, while simplifying the compiler, blocks optimizations by the rigid interface imposed by the library—most importantly, it inhibits the use of standard optimizations for processor-local code and hinders the efficient use of the memory hierarchy, i.e., optimizations for the processor cache and for minimizing communication operations.

We propose to continue compilation, where the library approach stops and delegates an efficient implementation to the library. As a result, we can optimize on all levels of the memory hierarchy (registers, caches, local memory, and distributed memory)—after breaking the abstraction enforced by the library, optimizations across multiple operations become possible and data distribution gets more flexible. Our main technical contribution is an *intermediate language* that distinguishes local and distributed data structures and separates local from global operations. We present the internals of the library routines to the back-end of the compiler in the form of this intermediate language, which allows us to implement optimizations by program transformation—thus, the optimizations are easy to comprehend, the compiler can be well structured, optimizations are easily re-ordered and can be proved correct individually. Although we provide some technical detail, we do not have enough room for discussing all important points; the proposed technique has been applied to the implementation of nested data parallelism in the first author's PhD thesis [17], where the missing detail can be found.

Our main contributions are the following three:

- We introduce the notion of *distributed types*, to statically distinguish between local and distributed data structures.
- We outline an intermediate language based on distributed types that allows to statically separate purely local computations, where program fusion can be applied to improve register and cache utilization, from global operations, where program transformations can be used to minimize communication.
- We present experimental evidence for the efficiency of our method.

The paper is structured as follows: Section 2 details the benefits of an intermediate language that explicitly distinguishes between local and global data structures and computations. Section 3 introduces the concept of distributed types and outlines a declarative intermediate language based on these types. Section 4 briefly outlines the use of the intermediate language. Section 5 presents results obtained by applying our method to the implementation of nested data parallelism. Finally, Section 6 discusses related work and summaries.

2 Global versus Local Computations

Before outlining our proposal, let us discuss the disadvantages of the library approach.

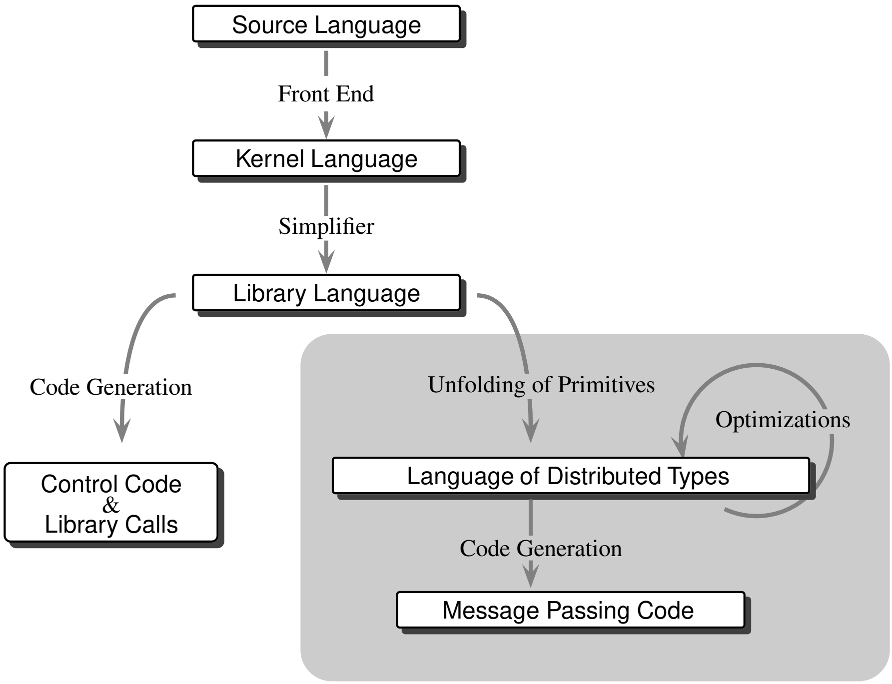


Fig. 1. Compiler structure (the grey area marks our new components)

2.1 The Problems of High-level Libraries

The structure of a compiler based on the library approach is displayed in Figure 1 when the grey area is omitted. The source language is analyzed and desugared, yielding a kernel language, which forms the input to the back-end (optimizations on the kernel language are left out in the figure). The back-end simplifies the kernel language and generates the control code that implements the source program with calls to the library implementing the parallel aggregate structure.

As an example, consider the implementation of Nesl [1]. In the case of Nesl, the simplifier implements the *flattening transformation* [5, 20, 16], which transforms all nested into flat data parallelism. In CMU's implementation of the language [4], the library language is called *VCODE* [2]—it implements a variety of operations on simple and segmented vectors, which are always uniformly distributed over the available processing elements. In fact, CMU's implementation does not generate an executable, but instead emits VCODE and interprets it at runtime by an interpreter linked to their *C Vector Library (CVL)* [3]. Unfortunately, this approach, while working fine for vector computers, is not satisfying for shared-memory [7] and distributed-memory machines [14].

The problems with this approach are mainly for three reasons: (1) Processor caches are badly utilized, (2) communication operations cannot be merged, and (3) data distribution is too rigid. In short, the program cannot be optimized for the memory hierarchy.

2.2 Zooming Into the Library

The library routines provide a *macroscopic view* of the program, where many details of the parallel implementation of these routines are hidden from the compiler. We propose to break the macroscopic view in favour of a *microscopic view*, where each of the library routines is unfolded into some *local computations* and *global communication* operations. The grey area of Figure 1 outlines an improved compiler structure including this unfolding and an optimization phase, where the microscopic view allows new optimizations realized as program transformations that minimize access to data elements that are located in expensive regions of the memory hierarchy. In the following, we outline the most important optimizations. Please note that the optimizations themselves are not new—our contribution is to *enable* the automatic use of these optimizations for the parallel implementation of aggregate structures in collection-oriented languages.

Optimizing local computations. The microscopic view allows to jointly optimize adjacent local computations that were part of different library routines; in the extreme, purely local routines, such as, elementwise operations on aggregate structures, can be fully merged. Such optimizations are crucial to exploit machine registers and caches.

For example, the following Nesl [1] function summing all squares from `1` to `n` proceeds in three steps: First, a list of numbers from `1` to `n` is generated; second, the square for each number is computed; and third, all squares are summed up.

```
sumSq1 (n) : Int -> Int =
  let nums = [1:n+1];              — numbers from 1 to n
      sqs  = {i*i : i in nums}     — square each i from nums
  in
  sum (sqs);                       — perform a reduction on sqs
```

Each of the three steps corresponds to one VCODE operation, i.e., one library routine, in CMU's Nesl implementation. Even on a single processor machine, this program performs poorly when the value of `n` is sufficiently large, i.e., when the intermediate results do not fit into the cache anymore. Much more efficient is the following explicitly recursive version that can keep all intermediate results in registers.

```
sumSq2 (n) : Int -> Int = sumSq2' (1, n);

sumSq2' (x, n) : (Int, Int) -> Int =
  if (x > n) then 0
  else x*x + sumSq2' (x + 1, n);
```

In sequential programming, *fusion* (or deforestation) techniques are applied to derive `sumSq2` automatically from `sumSq1` [25, 11, 24, 19]. Unfortunately, these techniques are not directly applicable to parallel programming, because they replace the aggregate-based computations fully by sequential recursive traversals—`sumSq2` completely lost the parallel interpretation of `sumSq1`. However, as soon as we have clearly separated local from global computations, we can apply fusion and restrict it to the local computations, so that the global, parallel structure of the program is left intact.

Optimizing global communication. The separation of local and global operations, which becomes possible in the microscopic view, allows to reduce communication overhead in two important ways: (1) We can merge communication operations and (2) we can trade locality of reference for a reduction of communication needed for load balancing. A simple example for the first point is the Nesl expression for concatenating three vectors: `(xs ++ ys) ++ zs`. It specifies pure communication, which re-distributes `xs`, `ys`, and `zs` such that the result vector is adequately distributed. Hence, the intermediate result `xs ++ ys` should ideally not be computed at all, but `xs` and `ys` should immediately be distributed correctly for the final result. Unfortunately, the rigid interface of a high-level library hinders such an optimization.

Regarding the second point, i.e., trading locality of reference for reduced load balancing costs, it is well known that optimal load balance does not necessarily lead to minimal runtime. Often, optimal load balance requires so much communication that an imbalanced computation is faster. Again, the compiler would often be able to optimize this tradeoff, but it is hindered in generating optimal code in the library approach, as the library encapsulates machine specific factors and library routines require fixed data distributions for their arguments.

3 Distributed Types Identify Local Values

A macroscopic parallel primitive, i.e., an operation on an aggregate structure in the library approach, is in the microscopic view implemented as a combination of purely local and global subcomputations. The latter induce communication to either combine the local sub-results into a global result or to propagate intermediate results to other processors, which need this information to execute the next local operation. For example, consider the summation of all values of an aggregate structure on a distributed-memory machine. In a first step, each processor adds up all the values in its local memory; in the second step, the processors communicate to sum all local results, yielding the global result, which is usually distributed to all processors. To model local and global operations in the microscopic view, we need to distinguish between local and global values. We define a *global value* as a value whose layout is known by all processors; thus, each processor is able to access the components of a global value without explicitly co-operating with other processors.[2] In contrast, a *local value* is represented by an instance on every processor, where the various instances are completely independent and accessible only by the processor on which they reside. We use a type system including *distributed types* to statically model this difference.

3.1 Distributed Types

In the source language, we distinguish between scalar values of basic type and compound values of aggregate type, where the latter serve to express parallelism. Both kinds

---

[2] We assume a global address space, where a processor can get non-local data from another processor as long as it knows the exact layout of the whole structure across all processors. Such a global address space is supported in hardware by machines, such as the Cray T3E, and is generally available in MPI-2 by one-sided communication [13].

of values are global in the sense defined before; each processor either has a private copy of the value (this is usually the case for scalar values) or it is aware of the global layout and can, thus, access any component of the value if necessary. Such values are often not suitable for expressing the results computed by local operations, as for example, the local sums computed in the first step of the previous summation example. The result of the local summation is actually an ordered collection of scalar values, with as many elements as processors. The ordering, which is essential for some computations, corresponds to the ordering of the processors. We introduce *distributed types*, denoted by double angle brackets $\langle\!\langle\cdot\rangle\!\rangle$ to represent such local values; to be precise, such a value is an ordered collection of fixed, but unknown arity. The arity reflects the number of processors, which is unknown at compile-time, but fixed throughout one program execution. For any type α, $\langle\!\langle\alpha\rangle\!\rangle$ represents such a collection of processor-local values of type α.

The usefulness of distributed types for local results of basic type can easily be seen in the sum example. However, it is probably not as obvious for values of aggregate type, as those values are usually not replicated across processors, but distributed over the processors. Still, the distinction between global and local values is also important for aggregate data types. For example, consider a filter operation that removes all elements that do not satisfy a given predicate from a collection. After each processor computed the filter operation on its local share of the aggregate structure, the overall size of the result is not yet known; thus, the intermediate result of the local operation is not a global aggregate structure according to our definition, but a collection of local structures.

3.2 Distributed Computations

Together with the type constructor $\langle\!\langle\cdot\rangle\!\rangle$, we introduce a higher-order operation $\langle\!\langle\cdot\rangle\!\rangle$. For a function f, we denote by $\langle\!\langle f\rangle\!\rangle$ the operation that applies f in parallel to each local component of a value of distributed type. Thus, for

$$f : \alpha_1 \times \cdots \times \alpha_n \to \beta_1 \times \cdots \times \beta_m \tag{1}$$

the operation $\langle\!\langle f\rangle\!\rangle$ has type

$$\langle\!\langle f\rangle\!\rangle : \langle\!\langle\alpha_1\rangle\!\rangle \times \cdots \times \langle\!\langle\alpha_n\rangle\!\rangle \to \langle\!\langle\beta_1\rangle\!\rangle \times \cdots \times \langle\!\langle\beta_m\rangle\!\rangle \tag{2}$$

In other words, $\langle\!\langle f\rangle\!\rangle\ (a_1, \ldots, a_n)$, with the a_i of distributed type $\langle\!\langle\alpha_i\rangle\!\rangle$, applies f on each processor P to the n-tuple formed from the components of a_1 to a_n that are local to P and yields an m-tuple of local results on P. The jth components of the result tuples of all processors form, then, a local value of distributed type $\langle\!\langle\beta_j\rangle\!\rangle$. This formalism distinguishes local computations (inside $\langle\!\langle\cdot\rangle\!\rangle$) from global computations and allows to calculate with them, while still maintaining a high level of abstraction.

3.3 A Language of Distributed Types

Before continuing with the implementation of aggregate structures using distributed types, let us become more concrete by fixing a syntax for an intermediate language that

$$\begin{array}{lll}
\mathsf{F} \to & \mathsf{V}\ (\ \mathsf{V}_1\ ,\ \ldots\ ,\ \mathsf{V}_n\ )\ :\ \mathsf{T}\ =\ \mathsf{E} & \text{(function definition)}\\
\mathsf{T} \to & \langle\langle\ \mathsf{T}\ \rangle\rangle & \text{(distributed type)}\\
\mid & [\ \mathsf{T}\ ] & \text{(aggregate type)}\\
\mid & \mathsf{T}_1\ \times\ \cdots\ \times\ \mathsf{T}_n & \text{(tuple type)}\\
\mid & \mathsf{T}_1\ \to\ \mathsf{T}_2 & \text{(function type)}\\
\mid & \mathit{Int}\ \mid\ \mathit{Float}\ \mid\ \mathit{Char} & \text{(scalar types)}\\
\mathsf{E} \to & \mathbf{let}\ \mathsf{V}\ =\ \mathsf{E}\ \mathbf{in}\ \mathsf{E} & \text{(local definition)}\\
\mid & \mathbf{if}\ \mathsf{E}_1\ \mathbf{then}\ \mathsf{E}_2\ \mathbf{else}\ \mathsf{E}_3 & \text{(conditional)}\\
\mid & \mathsf{E}_1\ \mathsf{E}_2 & \text{(application)}\\
\mid & \langle\langle\ \mathsf{E}\ \rangle\rangle & \text{(replication)}\\
\mid & [\ \mathsf{E}_1\ ,\ \ldots\ ,\ \mathsf{E}_n\ ] & \text{(aggregate constructor)}\\
\mid & (\ \mathsf{E}_1\ ,\ \ldots\ ,\ \mathsf{E}_n\ ) & \text{(tuple)}\\
\mid & \mathsf{V} & \text{(variable)}
\end{array}$$

Fig. 2. Syntax of $\mathcal{L}_{\text{DT}}$

supports distributed types. The language $\mathcal{L}_{\text{DT}}$, whose syntax is displayed in Figure 2, allows a functional representation of the compiled program. A functional representation is well suited for optimization by program transformation and does not restrict the applicability of our approach, as parallel computations are usually required to be side-effect free, even in imperative languages (see, for example, the *pure functions* of Fortran 95 and HPF [10]).

A program is a collection of typed function definitions produced by F. Types T include a set of scalar types as well as more complex types formed by the tuple and function constructors. Furthermore, $[\alpha]$ denotes an aggregate type whose elements are of type α and $\langle\langle\alpha\rangle\rangle$ is a distributed type as discussed previously.

Expressions E include the standard constructs for local definitions, conditionals, and function application as well as tuple and aggregate construction. Moreover, we have $\langle\langle E\rangle\rangle$ for some expression E, where we require that E is of functional type and the function represented by E is made into a parallel operation as discussed in the previous subsection. Furthermore, we use $E_1 \circ E_2$ to denote function composition and $E_1 \times E_2$ to denote the application of a pair of functions to a pair of arguments, i.e., $(f \times g)\,(x,\, y) \equiv (f\,(x),\, g\,(y))$. We omit a formal definition of $\mathcal{L}_{\text{DT}}$'s type system, as it is standard apart from the addition of distributed types, which essentially means to cast Equations 1 and 2 into a typing rule. Replication distributes over $\circ$ and $\times$, i.e., $\langle\langle f\rangle\rangle \circ \langle\langle g\rangle\rangle = \langle\langle f \circ g\rangle\rangle$ and $\langle\langle f\rangle\rangle \times \langle\langle g\rangle\rangle = \langle\langle f \times g\rangle\rangle$.

3.4 Operations on Distributed Types

To re-phrase the primitives of the library approach in the microscopic view, we need operations that convert values of global type into values of distributed type and vice versa. In general, we call a function of type $\alpha \to \langle\langle\alpha\rangle\rangle$ a *split operation* and a function of type $\langle\langle\alpha\rangle\rangle \to \alpha$ a *join operation*. Intuitively, a split operation decomposes a global value into its local components for the individual processors. Conversely, a join operation combines the components of a local value into one global value. Note that especially split operations do not necessarily perform work; sometimes they merely shift the perspective from a monolithic view of a data structure to one where the distributed components of the same structure can be manipulated independently.

The operation $\mathit{split\_scalar}_A : A \rightarrow \langle\langle A\rangle\rangle$ takes a value of basic type A (i.e., *Bool*, *Int*, or *Float*) and yields a local value whose components are identical to the global value. It is an example of a function, which typically does not imply any work, as it is common to replicate scalar data over all processors, to minimize communication; but, the shift in perspective is essential, as $\langle\langle A\rangle\rangle$ allows to alter the local copies of the value independently, whereas A implies the consistent change of all local copies.

On values of type integer, we also have $\mathit{split\_size} : \mathit{Int} \rightarrow \langle\langle \mathit{Int}\rangle\rangle$. The argument, say n, is assumed to be the size of an aggregate structure; $\mathit{split\_size}$ yields a collection of local integer values, whose sum is equal to n. Each component of the collection is the size of the processor-local chunk of an aggregate structure of size n when it is distributed according to a fixed load balancing strategy.

$\mathcal{L}_{\mathrm{DT}}$ requires a join operation for each reduction on the aggregate structure:

$$\begin{array}{lll} \mathit{join\_+}_A & : \langle\langle A\rangle\rangle \rightarrow A & \text{— for integers and floats} \\ \mathit{join\_max}_A & : \langle\langle A\rangle\rangle \rightarrow A & \text{— for integers and floats} \\ \mathit{join\_and} & : \langle\langle \mathit{Bool}\rangle\rangle \rightarrow \mathit{Bool} & \text{— and so on} \ldots \end{array}$$

They are used to combine the local results of the parallel application of the reduction to yield a global result—in the summation example from the beginning of this section, we would need $\mathit{join\_+}_{\mathit{Int}}$ if we sum integer values. Furthermore, we can use $\mathit{join\_+}_{\mathit{Int}} \circ \langle\langle \# \rangle\rangle$ to compute the number of elements in a structure of type $\langle\langle [\alpha]\rangle\rangle$, where $\#$ gives the size of a structure $[\alpha]$.

Furthermore, for aggregate structures, the operation $\mathit{split\_agg} : [\alpha] \rightarrow \langle\langle [\alpha]\rangle\rangle$ provides access to the local components of the distributed structure and, conversely, the operation $\mathit{join\_agg} : \langle\langle [\alpha]\rangle\rangle \rightarrow [\alpha]$ combines the local components of a structure into a global structure. The operation $\mathit{split\_agg}$ induces no communication or other costs; it merely provides an explicit view on the distribution of a global structure. In contrast, $\mathit{join\_agg}$ re-arranges the elements, if applied to a structure of distributed type, whose components are not properly distributed according to the fixed load balancing strategy. Thus, the combined application $\mathit{split\_agg} \circ \mathit{join\_agg}$, while computationally being the identity, ensures that the resulting structure is distributed according to the load-balancing strategy of the concrete implementation. To illustrate the use of split and join operations for the definition of more complex routines, we formalize the summation that we already discussed in the beginning of this section:

$$\mathit{sum}\,(\mathit{xs}) : [\mathit{Int}] \rightarrow \mathit{Int} = (\mathit{join\_+}_{\mathit{Int}} \circ \langle\langle \mathit{reduce\_+}_{\mathit{Int}}\rangle\rangle \circ \mathit{split\_agg})\ \mathit{xs}$$

After obtaining the local view of the aggregate structure xs with $\mathit{split\_agg}$, summation proceeds in two steps: First, the purely local reduction $\langle\langle \mathit{reduce\_+}_{\mathit{Int}}\rangle\rangle$, which yields a local integer result on each processing element, i.e., a value of type $\langle\langle \mathit{Int}\rangle\rangle$; and second, the global reduction of the local results with $\mathit{join\_+}_{\mathit{Int}}$.

Apart from join and split operations, which compute global values from distributed values, and vice versa, we need operations that *propagate* information over the components of a distributed value; they receive a value of distributed type as input parameter and also return a distributed type. Although the parallel application of local operations

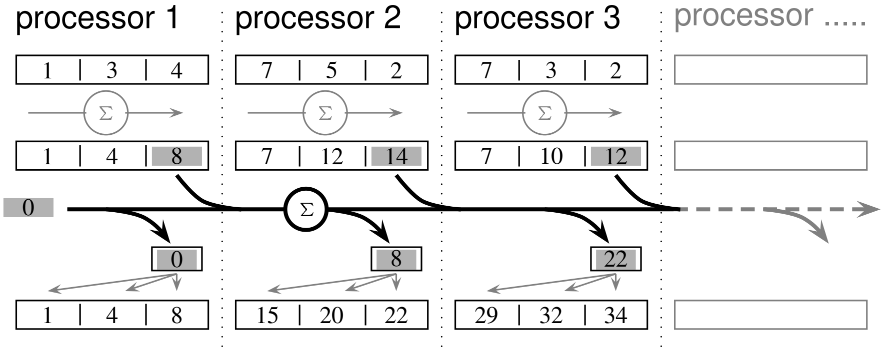


Fig. 3. Local and global operations in *prescan*_+

has the same type, namely $\langle\langle[\alpha]\rangle\rangle \rightarrow \langle\langle[\beta]\rangle\rangle$, propagation is fundamentally different, since, in the case of the parallel application of local operations, there is no communication between the processing elements. Similar to the join operations, where $\mathcal{L}_{\mathrm{DT}}$ offers an operation for every form of reduction, there is a propagate operation in $\mathcal{L}_{\mathrm{DT}}$ for each prescan operation that we need on the aggregate structure. A propagate operation prescans over the components of the distributed type according to the implicit order of the components. Some examples of this type of operation are the following:

$propagate\_+_A \ : \langle\langle A\rangle\rangle \rightarrow \langle\langle A\rangle\rangle$ — for integers and floats
$propagate\_max_A : \langle\langle A\rangle\rangle \rightarrow \langle\langle A\rangle\rangle$ — for integers and floats
$propagate\_and \ : \langle\langle Bool\rangle\rangle \rightarrow \langle\langle Bool\rangle\rangle$ — and so on . . .

Formally, the semantics of the split, join, and propagate operations can be characterized by a set of axioms [17]

4 Optimizing with Distributed Types

With $\mathcal{L}_{\mathrm{DT}}$ we can realize the three compiler phases marked by the grey area in Figure 1: unfolding of the library primitives, optimizations, and code generation. We do not have the space to discuss them in detail, but like to outline some important points of the first two phases, i.e., unfolding and optimizations, in the next two subsections. Afterwards, we discuss a small example to illustrate the technique. More details can be found in [17].

4.1 Unfolding Library Routines

The whole point of $\mathcal{L}_{\mathrm{DT}}$ is to allow defining the internal behaviour of the routines that are primitive in the library approach (the unfolding step in Figure 1). We already defined the summation function sum in $\mathcal{L}_{\mathrm{DT}}$; now, we discuss the related, but more complex pre-scan operation. We do not detail the local operations, because they depend on the implemented aggregate type and source language—in [17], the details for the case of nested parallel languages are given.

Figure 3 sketches the steps of a parallel *prescan*_+ routine. The dotted lines mark processor boundaries and indicate that each processor has a local copy of a part of the aggregate structure. Local computations are marked by grey, global computations by black arrows. The $\mathit{prescan}\_+$ algorithm proceeds in three steps:

1. Each processor computes a local pre-scan on its share of the aggregate structure. The local sum of each processor is marked with a grey box.
2. A global pre-scan (the black arrow) over all local sums provides each processor with the sum of all elements residing on preceding processors.
3. Each processor adds its result of the previous operation to each element of the intermediate vector computed in the first step.

Overall, the $\mathit{prescan}\_+$ operation consists of two purely local computations (Step 1 and 3) and one global operation (Step 2), entailing communication.

Next, we express this computation in terms of distributed types, *split*, *join*, *propagate*, and local operations. Before applying the first local operation, we have to use $\mathit{split}\_\mathit{agg}$ to obtain the local view on the input structure xs. Then, we apply the local pre-scan operation, $\mathit{prescan}\_+\_\mathit{local}$ to each component of the local value. As can be seen in Figure 3, $\mathit{prescan}\_+\_\mathit{local}$ has to return two values on each processor: the new local structure and the local sum—that is, $\mathit{prescan}\_+\_\mathit{local} : [\mathit{Int}] \rightarrow ([\mathit{Int}] \times \mathit{Int})$. Overall, Step 1 can be expressed in $\mathcal{L}_{\mathrm{DT}}$ by the composition $\langle\langle \mathit{prescan}\_+\_\mathit{local} \rangle\rangle \circ \mathit{split}\_\mathit{agg}$.

Step 2 essentially is the previously discussed $\mathit{propagate}\_+ : \langle\langle \mathit{Int} \rangle\rangle \rightarrow \langle\langle \mathit{Int} \rangle\rangle$. It realizes the black arrow of Figure 3, propagating the local results from left to right. In Step 3, we add, on each processor, the local result of the previous step to each element of the local fragment of the intermediate aggregate structure that we obtained in Step 1. Therefore, we define an auxiliary function $+\_\mathit{offset}$ that gets the results of the previous two steps as arguments.

$$+\_\mathit{offset}\ (zs,\ \mathit{off}) : [\mathit{Int}] \times \mathit{Int} \rightarrow [\mathit{Int}] = \mathit{elem}\_+\ (zs,\ \mathit{dist}\ (\mathit{off},\ \#zs))$$

The expression $\mathit{dist}\ (\mathit{off},\ \#zs)$ replicates the value off as often as the size of zs requires. Elementwise addition elem_+ computes the local contributions to the overall result, which in turn is obtained from the local results with $\mathit{join}\_\mathit{agg}$.

Overall, we have the following definition for $\mathit{prescan}\_+$, where local and global computations are explicit—local computations are enclosed in $\langle\langle \cdot \rangle\rangle$:

$$\begin{aligned} &\mathit{prescan}\_+\ (xs) : [\mathit{Int}] \rightarrow [\mathit{Int}] = \\ &\quad (\mathit{join}\_\mathit{agg} \circ \langle\langle +\_\mathit{offset} \rangle\rangle \circ (\mathit{id} \times \mathit{propagate}\_+) \\ &\quad\ \circ \langle\langle \mathit{prescan}\_+\_\mathit{local} \rangle\rangle \circ \mathit{split}\_\mathit{agg})\ xs \end{aligned}$$

4.2 Optimizations on $\mathcal{L}_{\mathrm{DT}}$

The unfolding of the library routines in $\mathcal{L}_{\mathrm{DT}}$ enables new kinds of optimizations, which we can categorize as follows: (1) optimizations of local computations, (2) optimizations of global operations, and (3) enabling optimizations. We outlined the first two kinds already informally in Section 2 and refer to [17] for a formal definition of applicable fusion techniques, optimizations minimizing communication, and example derivations. In the previous subsection, we avoided fixing a notation for specifying local computations, as there is a wide design space, which is largely independent from the main points of this paper. One possibility is to provide a set of primitives functions for manipulating local fragments of the aggregate structure; another possibility is the use of loop constructs that define iterators over the local fragments of aggregate structures (this the approach taken in [17]). In any case, a set of equational transformation rules should be provided that implements fusion (see Section 2) and possibly also other optimizations.

The third form of optimizations, *enabling optimizations*, are independent of the notation for local computations; they use essential properties of $\mathcal{L}_{\mathrm{DT}}$ to re-order local and global computations such that subcomputations are nicely clustered for the first two forms of optimizations. Space limitations again force us to refer to [17] for the complete set of these optimization rules; however, we want to state one crucial point: As mentioned, $\mathit{split\_agg} \circ \mathit{join\_agg}$ does not change the contents of the value of type $\langle\!\langle[\alpha]\rangle\!\rangle$ to which it is applied, but it ensures that the aggregate structure is well balanced. After unfolding the library primitives, all intermediate results are re-balanced in this way, which does not necessarily minimize the runtime (see Section 2). Thus, the optimizer removes many of these re-distributions, which saves communication and moves local computations, which were originally separated by re-balancing, next to each other. Thus, new local optimizations becomes possible, after applying the enabling optimization $\langle\!\langle f\rangle\!\rangle \circ \langle\!\langle g\rangle\!\rangle = \langle\!\langle f \circ g\rangle\!\rangle$.

4.3 An Example

We use `sumSq1` from Section 2.2 for an example transformation. Let us start with the definition of the function as it appears in the library language, i.e., after the font-end:

$$\mathit{sumSq}1\,(n) : \mathit{Int} \to \mathit{Int} = (\mathit{sum} \circ \mathit{map}\,(\mathit{sq}) \circ \mathit{range})\,(1, n)$$

where we have $\mathit{sq}\,(x) : \mathit{Int} \to \mathit{Int} = x * x$. We already provided the definition for sum in Section 3.4 and define map and range as follows:

$$\begin{array}{l}
\mathit{map}\,(f) : (a \to b) \to [a] \to [b] = \mathit{join\_agg} \circ \langle\!\langle \mathit{seq\_map}\,(f)\rangle\!\rangle \circ \mathit{split\_agg} \\
\quad \mathbf{where} \\
\qquad \mathit{seq\_map}\,(f)\,[x_1, \ldots, x_n] = [f\,(x_1), \ldots, f\,(x_n)] \\
\mathit{range} : \mathit{Int} \times \mathit{Int} \to [\mathit{Int}] = \mathit{join\_agg} \circ \langle\!\langle \mathit{seq\_range}\rangle\!\rangle \circ \mathit{split\_range} \\
\quad \mathbf{where} \\
\qquad \mathit{seq\_range}\,(n, m) = [n, n+1, \ldots, m]
\end{array}$$

For the local functions $\mathit{seq\_map}$ and $\mathit{seq\_range}$, we only provide a specification; the exact definition depends on how we choose to represent the processor-local, sequential code. However, the function $\mathit{split\_range}$ deserves a little more attention; it computes for each processing element the local range of values and may be defined as

$$\begin{array}{l}
\mathit{split\_range}\,(\mathit{from}, \mathit{to}) : \mathit{Int} \times \mathit{Int} \to \langle\!\langle \mathit{Int}\rangle\!\rangle \times \langle\!\langle \mathit{Int}\rangle\!\rangle = \\
\quad \mathbf{let} \\
\qquad \mathit{lens} \;\;\, = \mathit{split\_size}\,(\mathit{to} - \mathit{from} + 1) \\
\qquad \mathit{froms} = \langle\!\langle + \rangle\!\rangle\,(\mathit{split\_scalar}_{\mathit{Int}}\,(\mathit{from}), \mathit{propagate\_}+\,(\mathit{lens})) \\
\quad \mathbf{in} \\
\quad (\mathit{froms}, \langle\!\langle \lambda\, f, l. f + l - 1\rangle\!\rangle\,(\mathit{froms}, \mathit{lens}))
\end{array}$$

Nevertheless, it makes sense to regard this function as a primitive, as it can, in dependence on the distribution policy for the concrete aggregate structure, usually be implemented without any communication.

When the compiler unfolds the library primitives, it can transform the function compositions within the body of $\mathit{sumSq}1$ as follows:

$$\begin{aligned}
&(\mathit{join\_{+}}_{\mathit{Int}} \circ \langle\!\langle \mathit{reduce\_{+}}_{\mathit{Int}} \rangle\!\rangle \circ \mathit{split\_agg}) \circ (\mathit{join\_agg} \circ \langle\!\langle \mathit{seq\_map}\ (\mathit{sq}) \rangle\!\rangle \\
&\quad \circ \mathit{split\_agg}) \circ (\mathit{join\_agg} \circ \langle\!\langle \mathit{seq\_range} \rangle\!\rangle \circ \mathit{split\_range}) \\
= &\ \{\text{eliminate } \mathit{split\_agg} \circ \mathit{join\_agg} \quad \text{(there is no load imbalance anyway)}\} \\
&\mathit{join\_{+}}_{\mathit{Int}} \circ \langle\!\langle \mathit{reduce\_{+}}_{\mathit{Int}} \rangle\!\rangle \circ \langle\!\langle \mathit{seq\_map}\ (\mathit{sq}) \rangle\!\rangle \circ \langle\!\langle \mathit{seq\_range} \rangle\!\rangle \circ \mathit{split\_range} \\
= &\ \{\text{distribution law for replication} \quad \text{(Section 3.3)}\} \\
&\mathit{join\_{+}}_{\mathit{Int}} \circ \langle\!\langle \mathit{reduce\_{+}}_{\mathit{Int}} \circ \mathit{seq\_map}\ (\mathit{sq}) \circ \mathit{seq\_range} \rangle\!\rangle \circ \mathit{split\_range}
\end{aligned}$$

Now, the local computation $\mathit{reduce\_{+}}_{\mathit{Int}} \circ \mathit{seq\_map}\ (\mathit{sq}) \circ \mathit{seq\_range}$ is a slight generalization of the original $\mathit{sumSq}1$ function (it has an explicit start value instead of starting with 1). As it is purely local, we can fuse it into essentially the same code as `sumSq2'` (the auxiliary function in the fused version of Section 2.2), i.e., we get

$$\mathit{sumSq}1\,(n) \,:\, \mathit{Int} \to \mathit{Int} = (\mathit{join\_{+}}_{\mathit{Int}} \circ \langle\!\langle \mathit{sumSq}2' \rangle\!\rangle \circ \mathit{split\_range})\,(1,\ n)$$

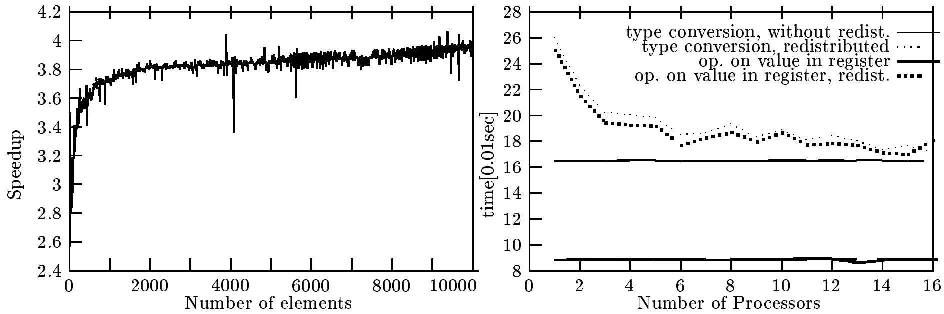
We achieved our goal of applying fusion to the local code in a controlled manner, which leaves the parallel semantics of our code intact.

5 Benchmarks

We summarize results collected by applying our method to the implementation of Nesl-like [1] nested data parallelism [17]. However, we do not directly compare our code and that of CMU's implementation of Nesl [4], because the implementation of CMU's vector library CVL [3] is already an order of magnitude slower and scales worse than our vector primitives on the Cray T3E, which we used for the experiments. Our code is hand-generated using the compilation and transformation rules of [17]—we are currently implementing a full compiler.

Local optimizations. In Section 2, we mentioned fusion as an important technique for optimizing local computations. For sufficiently large vectors, the optimization achieves a speedup linear to the numbers of vector traversals that become obsolete as main memory access is reduced. Comparing `sumSq1` and `sumSq2` from Section 2, we find a speedup of about a factor four for large vectors—see Figure 4 (left part). Fusion is particularly important for compiling nested parallelism, as the flattening transformation [5, 20, 16] leads to extremely fine-grained code.

Global optimizations. Concerning global optimizations, we measured the savings from reducing load balancing, i.e., a $\mathit{split\_agg} \circ \mathit{join\_agg}$ combination (see previous section). In Figure 4 (right part), we applied a filter operation removing all elements from a vector except those on a single processor, thus, creating an extreme load imbalance. After filtering, we applied one scalar operation elementwise. The figure shows that, if this operation (due to fusion) finds its input in a register, re-distribution is up to 16 processors significantly slower than operating on the unbalanced vector. In the case of a more complex operation, the runtime becomes nearly identical for 16 processors. As the flattening transformation introduces many filter operations it is essential to avoid re-distribution where possible.



The left figure displays the speedup of *sumSq2* over *sumSq1* (by loop fusion). The right figure contains the runtime of filtering plus one successive operation; with and without re-distribution.

Fig. 4. Effects of local and global optimizations

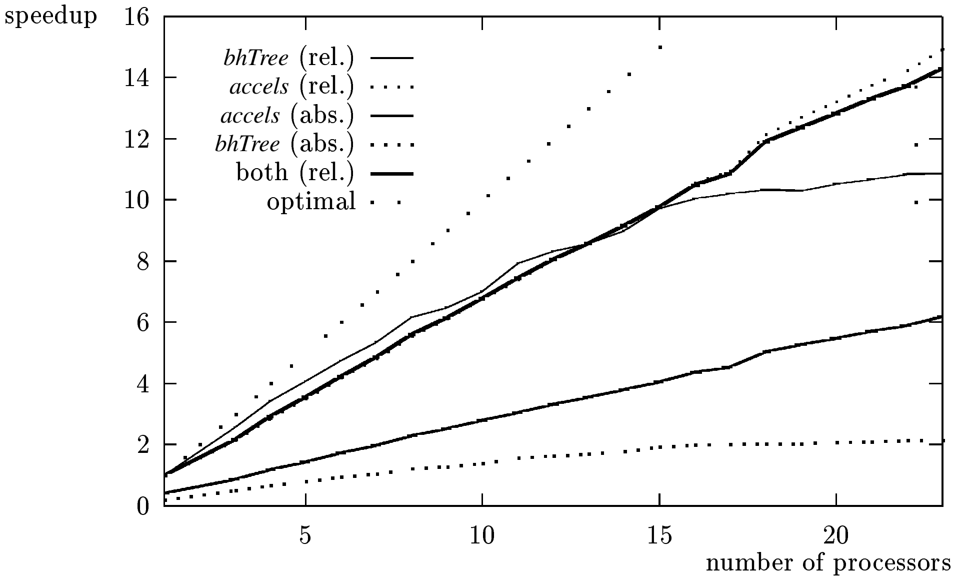


Fig. 5. Relative and absolute speedup of n-body algorithm for 20,000 particle

A complete application. We implemented an n-body code, the 2-dimensional Barnes-Hut algorithm, to get a feeling for the overall quality of our code. The absolute and relative speedup for 20,000 particles are shown in Figure 5. The runtime is factored into the two main routines, which build the Barnes-Hut tree (*bhTree*) and compute the accelerations of the particles (*accels*), respectively. The routine *bhTree* needs a lot of communication, and thus, scales worse, but as it contributes only to a small fraction of the overall runtime, the overall algorithm shows a speedup nearly identical to *accels*.

The absolute speedup is against a purely sequential C implementation of the algorithm. We expect to be able to improve significantly on the absolute speedup, which is about a factor of 6 for 24 processors, because we did not exploit all possible optimizations. To see the deficiency of the library approach, consider that in the two-dimensional Barnes-Hut algorithm, we achieve a speedup of about a factor of 10 by fusion.

6 Related Work and Conclusions

With regard to implementing nested parallelism, Chatterjee et al's work, which covers shared-memory machines [7], is probably closest to ours; however, their work is less general and not easily adapted for distributed memory. In the BMF community, *p-bounded lists* are used to model block distributions of lists [22]; this technique is related to distributed types, but the latter are more general as they (a) can be applied to any type and (b) do not make any assumption about the distribution, especially, $split\_agg \circ join\_agg$ is in general not the identity. Gorlatch's *distributed homomorphisms* appear to be well suited to systematically derive the function definitions, which we need for the library unfoldings—like, e.g., the $prescan\_+$ definition [12]. To the best of our knowledge, we are the first to propose a *generic, transformation-based* alternative to the library approach. Our method enables a whole class of optimizations by exposing the internals of the library routines to the compiler's optimizer; the transformation-based approach systematically distinguishes between local and global operations using a type system and makes it easy to formalize the optimizations. Most other work on implementing aggregate structures either focuses on the library approach or describes language-specific optimizations. We applied our method to the implementation of nested data parallelism and achieved satisfying results in first experiments.

Acknowledgements. We thank the Nepal project members, Martin Simons and Wolf Pfannenstiel, for providing the context of our work and feedback on a first version of the paper. We are also grateful to Sergei Gorlatch and the anonymous referees for their comments on an earlier version. The first author likes to thank Tetsuo Ida for his hospitality during her stay in Japan. Her work was funded by a PhD scholarship of the Deutsche Forschungsgemeinschaft (DFG).

References

1. Guy E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, 1996.
2. Guy E. Blelloch and Siddhartha Chatterjee. VCODE: A data-parallel intermediate language. In *Proceedings Frontiers of Massively Parallel Computation*, pages 471–480, October 1990.
3. Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Margaret Reid-Miller, Jay Sipelstein, and Marco Zagha. CVL: A C vector library. Technical Report CMU-CS-93-114, Carnegie Mellon University, 1993.
4. Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. Implementation of a portable nested data-parallel language. In *4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1993.
5. Guy E. Blelloch and Gary W. Sabot. Compiling collection-oriented languages onto massively parallel computers. *Journal of Parallel and Distributed Computing*, 8:119–134, 1990.
6. George Horatiu Botorog and Herbert Kuchen. Using algorithmic skeletons with dynamic data structures. In *IRREGULAR 1996*, volume 1117 of *Lecture Notes in Computer Science*, pages 263–276. Springer Verlag, 1996.
7. S. Chatterjee. Compiling nested data-parallel programs for shared-memory multiprocessors. *ACM Transactions on Programming Languages and Systems*, 15(3):400–462, july 1993.

8. S. Chatterjee, Jan F. Prins, and M. Simons. Expressing irregular computations in modern Fortran dialects. In *Fourth Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, Lecture Notes in Computer Science. Springer Verlag, 1998.
9. J. Darlington, A. J. Field, P. G. Harrison, P. H. J. Kelly, D. W. N. Sharp, Q. Wu, and R. L. While. Parallel programming using skeleton functions. In A. Bode, M. Reeve, and G. Wolf, editors, *PARLE '93: Parallel Architectures and Languages Europe*, number 694 in Lecture Notes in Computer Science, pages 146–160, Berlin, Germany, 1993. Springer-Verlag.
10. High Performance Fortran Forum. High Performance Fortran language specification. Technical report, Rice University, 1993. Version 1.0.
11. Andrew J. Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In Arvind, editor, *Functional Programming and Computer Architecture*, pages 223–232. ACM, 1993.
12. S. Gorlatch. Systematic efficient parallelization of scan and other list homomorphisms. In L. Bouge, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Euro-Par'96, Parallel Processing*, number 1124 in Lecture Notes in Computer Science, pages 401–408. Springer-Verlag, 1996.
13. William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir. *MPI: The Complete Reference*, volume 2—The MPI-2 Extensions. The MIT Press, second edition, 1998.
14. Jonathan C. Hardwick. An efficient implementation of nested data parallelism for irregular divide-and-conquer algorithms. In *First International Workshop on High-Level Programming Models and Supportive Environments*, 1996.
15. C. Barry Jay. Costing parallel programs as a function of shapes. Invited submission to Science of Computer Programming, September 1998.
16. G. Keller and M. Simons. A calculational approach to flattening nested data parallelism in functional languages. In J. Jaffar, editor, *The 1996 Asian Computing Science Conference*, Lecture Notes in Computer Science. Springer Verlag, 1996.
17. Gabriele Keller. *Transformation-based Implementation of Nested Data Parallelism for Distributed Memory Machines*. PhD thesis, Technische Universität Berlin, Fachbereich Informatik, 1998. To appear.
18. Michael Medcalf and John Reid. *Fortran 90 Explained*. Oxford Science Publications, 1990.
19. Y. Onue, Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. A calculational fusion system HYLO. In R. Bird and L. Meertens, editors, *Proceedings IFIP TC 2 WG 2.1 Working Conf. on Algorithmic Languages and Calculi, Le Bischenberg, France, 17–22 Feb 1997*, pages 76–106. Chapman & Hall, London, 1997.
20. Jan Prins and Daniel Palmer. Transforming high-level data-parallel programs into vector operations. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 119–128, San Diego, CA., May 19-22, 1993. ACM.
21. Jay M. Sipelstein and Guy E. Blelloch. Collection-oriented languages. *Proceedings of the IEEE*, 79(4):504–523, April 1991.
22. D. Skillicorn and W. Cai. A cost calculus for parallel functional programming. *Journal of Parallel and Distributed Computing*, 28:65–83, 1995.
23. D.B. Skillicorn. *Foundations of Parallel Programming*. Cambridge Series in Parallel Computation 6. Cambridge University Press, 1994.
24. Akihiko Takano and Erik Meijer. Shortcut deforestation in calculational form. In *Conf. Record 7th ACM SIGPLAN/SIGARCH Intl. Conf. on Functional Programming Languages and Computer Architecture*, pages 306–316. ACM Press, New York, 1995.
25. Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.