Flattening Trees

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

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

Abstract. Nested data-parallelism can be efficiently implemented by mapping it to flat parallelism using Blelloch & Sabot's *flattening* transformation. So far, the only dynamic data structure supported by flattening are vectors. We extend it with support for user-defined recursive types, which allow parallel tree structures to be defined. Thus, important parallel algorithms can be implemented more clearly and efficiently.

1 Introduction

The *flattening* transformation of Blelloch & Sabot [BS90,Ble90] implements *nested* data-parallelism by mapping it to *flat* data-parallelism. Compared to flat parallelism, nested parallelism allows algorithms to be expressed on a higher level of abstraction while providing a language-based performance model [Ble96]; in particular, algorithms operating on irregular data structures and divide-and-conquer algorithms benefit from nested parallelism. Efficient code can be generated for a wide range of parallel machines [BCH$^+$93,PP93,PPW95,KS96]. However, flattening supports only vectors (i.e., homogeneous, ordered sequences) as dynamic data structures. Thus, important parallel algorithms, like hierarchical n-body codes and other adaptive algorithms based on tree structures, are awkward to program—the tree structure has to be mapped to a vector structure, implying explicit index calculations, to keep track of the parent-child relation, and leading to a suboptimal data distribution on distributed-memory machines.

In this paper, we propose user-defined *recursive types* to tackle the mentioned problems. We extend flattening such that it maps the new structures to efficient, flat data-parallel code. Our extension fits easily into existing formalizations and implementations of flattening; in particular, the optimization techniques of previous work [PP93,PPW95,KS96,Cha93,Kel98] remain applicable. This paper makes the following three main contributions: (1) It demonstrates the usefulness of recursive types for nested data-parallel languages (Section 2), (2) it formally specifies our extension of flattening including user-defined recursive types (Section 3), and (3) it provides experimental results gathered with the resulting code on a Cray T3E (Section 4). Regarding point (2), as a side-effect of our extension, we contribute to a rigorous specification of flattening by formalizing the instantiation of polymorphic primitives. Thereby, we also introduce a new kind of primitives, so-called *chunkwise operations*, for more efficient data redistribution on distributed-memory machines. We use the functional language Nesl [Ble96] throughout this paper, but the discussed techniques also work for imperative languages [ACD+97]. Many details and discussions have been omitted in this paper due to shortage of space—this material can be found in [KC98].

Section 2 discusses the benefit of recursive types for tree-based algorithms in a purely vector-oriented language. Section 3 formalizes our extended flattening transformation. Section 4 presents benchmarks. Finally, Section 5 concludes.

2 The Problem: Encoding Trees by Vectors

Nesl [Ble96] is strict functional language featuring *nested vectors* as its central data structure. In addition to built-in parallel operations on vectors, the *apply-to-each* construct is used to express parallelism. In its general form `{`e `:` x_1 `in` e_1`, ... ,` x_n `in` e_n `|` f`}` we call e the *body*, the x_i `in` e_i the *generators*, and f (which is optional) the *filter*. The body e is evaluated for each element of the vectors e_i and the result is included in the result vector if f evaluates to `T` (true); the vectors e_i are required to be of equal length and are processed in lock-step. For example, `{x + y : x in [1, -2, 3], y in [4, 5, 6] | x > 0}` evaluates to `[5, 9]`. Nested parallelism occurs where parallel operations appear in the body of an apply-to-each, e.g., `{plus_scan (x) : x in xs}`, which for `xs = [[1, 2, 3], [4], [5, 6]]` yields `[[0, 1, 3], [0], [0, 5]]`. The built-in `plus_scan` is a prescan using addition [Ble90].

The implementation of a tree-based algorithm in such a language implies representing trees by nested vectors, obscuring the code with explicit index calculations, to keep track of the parent-child relation. Moreover, in an implementation based on flattening, these nested vectors are represented by a set of flat vectors in the target code. All *data elements* of the tree are mapped to a single vector and are uniformly distributed to achieve load balancing. This, however, leads to superfluous redistributions as those algorithms usually traverse the trees breadth-first, i.e., level by level, all nodes on one level are processed in parallel.

We illustrate these problems at the example of Barnes & Hut's hierarchical n-body code [BH86]. It minimizes the number of force calculations by grouping particles hierarchically into *cells* according to their spatial position. The hierarchy is represented by a tree. This allows approximating the accelerations induced by a group of particles on distant particles by using the centroid of that group's cell. The algorithm has two phases: (1) The tree is constructed from a particle set, and (2) the acceleration for each particle is computed in a down-sweep over the tree. The following Nesl function outlines the tree construction:

```
function bhTree (ps) : [MassPnt] -> [Cell] =
  if #ps == 1 then [Cell (ps[0], [])]            — cell has only one particle
  else let ⟨split particles ps into spatial groups pgs⟩
           subtrees  = {bhTree (pg): pg in pgs};
           children  = {subtree[0] : subtree in subtrees};                (⋆)
           cd        = centroid ({mp : Cell (mp, is) in children});
           sizes     = {#subtree : subtree in subtrees}                   (⋆)
  in [Cell (cd, sizes)] ++ flatten (subtrees)   — whole tree as flat vector  (⋆)
```

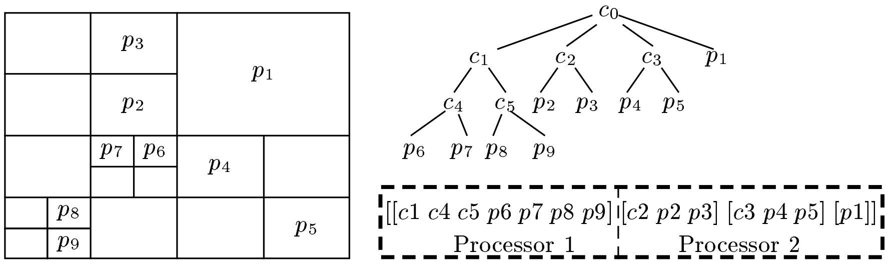

Fig. 1. Example of a Barnes-Hut tree and its representation as a vector.

We represent the tree as a *vector of cells*. Each cell of the form `Cell (`*mp*`,` *szs*`)` corresponds to a node of the tree and is a pair of a mass point *mp* and the sizes of the subtrees *szs* (tuples can be named in Nesl). Given such a vectorized `tree`, the acceleration of a set of mass points `mps` is computed by

```
function accels (tree, mps) : ([Cell], [MassPnt]) -> [Vec] =
  if #mps == 0 then []
  else let Cell (cd, chNos) = tree[0];              — get root          (⋆)
           ⟨split mps into closeMps and farMps (direct force calculation)⟩
           farAcs    = {accel (cd, mp) : mp in farMps};
           subTrees  = partition (drop (tree, 1), chNos);                  (⋆)
           closeAcss = {accels (tree, closeMps) : tree in subTrees};
  in ⟨combine farAcs and closeAcss⟩
```

It computes the acceleration for the mass points in `farMps` directly (using the function `accel`) and recurses into the tree for those in `closeMps`. The function `drop` omits the first element of a vector and `partition` forms a nested vector according to the lengths passed in the second argument (it is the same as $\mathcal{P}$ in the next section). The type `Vec` represents 'vectors' in the sense of physics.

The lines marked by $(\star)$ in the functions are (partially) artifacts of maintaining the tree as a vector. Figure 1 depicts the grouping of an exemplary particle distribution and the corresponding tree. The tree is both built and traversed level by level, i.e., all nodes in one level of the tree are processed in a parallel step. Let us consider the data layout (over two processors) for the example tree in Figure 1. To ensure proper load balancing, all cells of the already constructed subtrees have to be redistributed in each recursive step of `bhTree`. Similarly, `accels`, while descending the tree, has to redistribute those cells that correspond to one level of the tree. We will quantify these costs experimentally in Section 4.

It should be clear that it is more suitable to store the nodes of the tree in a distinct vector for each level of the tree, and then, to chain these vectors to represent the whole tree. At the source level, such a structure corresponds to regarding each node as being composed from some node data plus a vector of tree nodes that have exactly the *same* structure; in other words, we need a *recursive type*. For our example, we can use `datatype Node (MassPnt, [Node])`. Then, we simplify the function `bhtree` as follows: We omit computing `children` and `sizes`, we compute `cd` by `centroid ({mp : Node (mp, chs) in trees})`, and change the body of the `let` into `Node (cd, subtrees)`. In `accels`, computing `subtrees` becomes superfluous, and `closeAcss` is computed by `accels (tree, closeMps) : tree in subtrees}` where `Node (cd, subtrees) = tree`.

Generally, we extend Nesl with named tuples that refer to themselves, but only in the element type of a vector (to avoid infinite types), as we can terminate recursion only by empty vectors—there are no arbitrary sum types. Handling sum types efficiently in flattening seems much harder than recursive types.

3 Flattening Trees by Program Transformation

After the source language extension, we proceed to the implementation of user-defined recursive types by flattening. The flattening transformation serves three aims: First, it replaces all nested parallel expressions by flat parallel expressions that contain the same degree of parallelism, second, it changes the representation of data structures such that vectors contain only elements of base type, and third, it replaces all polymorphic vector primitives with monomorphic instances.

In this section, we begin by introducing the flat data-parallel kernel language Fkl, which is the target language of the first part of the flattening transformation. We continue with a discussion of the target representation of data structures, and then, describe an instantiation procedure, which implements the change of the representation of data structures as well as the generation of all necessary monomorphic instances of the primitives. Due to changing the representation of data structures, the instantiation of the polymorphic primitives becomes technically challenging—especially so, in the presence of recursive types. The representation of recursive types together with the instantiation procedure are the central technical contributions of this paper.

Although the instantiation of the polymorphic primitives—excluding recursive types—is already implicit in previous work, it was never described in detail (for example, Blelloch [Ble90] merely gives some examples and leaves the non-trivial induction of a general algorithm to the inclined reader); in particular, we provide the first formal specification. Our treatment also leads to more efficient code on distributed-memory machines than previous approaches (which concentrated on vector and shared-memory machines). A central idea of our method is the fact that only the *primitive* vector operations actually access or manipulate elements of nested vectors. Therefore, we can regard nested vectors as an abstract data type with some freedom in the concrete implementation.

We will not discuss the first step of the flattening, as it is already detailed in previous work [Ble90,PP93,KS96] and not affected by the addition of recursive types—however, a complete specification of the flattening transformation can be found in an unabridged version of this paper [KC98].

3.1 The Flat Kernel Language

A kernel language (Fkl) program consists of a list of declarations produced by the rule $\mathsf{D} \to \mathsf{V}\ (\mathsf{V}_1, \ldots, \mathsf{V}_n) = \mathsf{E}$, with variables V and expressions produced by

$$\mathsf{E} \to \mathsf{C} \mid \mathsf{V} \mid \mathsf{V}\ (\mathsf{E}_1, \ldots, \mathsf{E}_n) \mid \textbf{let}\ \mathsf{V} = \mathsf{E}_1\ \textbf{in}\ \mathsf{E}_2 \mid \textbf{if}\ \mathsf{E}_1\ \textbf{then}\ \mathsf{E}_2\ \textbf{else}\ \mathsf{E}_3 \mid [\mathsf{E}_1, \ldots, \mathsf{E}_n]$$

where C are constants. We assume programs are typed, with types from

$$\mathsf{T} \to \mathit{Int} \mid \mathit{Bool} \mid \mathsf{V} \mid (\mathsf{T}_1, \ldots, \mathsf{T}_n) \mid [\mathsf{T}] \mid \mu\mathsf{V}.\mathsf{T}$$

For brevity, we only have *Int* and *Bool* as primitive types. In a recursive type $\mu x.T$, all occurrences of x in T must be within element types of vectors (to get a finite type). For example, the type `Node` of Section 2 is represented as $\mu x.(\mathit{MassPnt}, [x])$, where *MassPnt* abbreviates the tuple of a mass point.

The second component of FKL are its primitive operations. Among the most important are the usual arithmetic and logic operations as well as construction of tuples $\tau^n : \alpha_1 \times \cdots \times \alpha_n \rightarrow (\alpha_1, \ldots, \alpha_n)$ and the corresponding projections $\pi^i_n : (\alpha_1, \ldots, \alpha_n) \rightarrow \alpha_i$ (the function type is given to the right of the colon). The vector operations include operations length $\# : [\alpha] \rightarrow \mathit{Int}$, concatenation $+\!\!+ : [\alpha] \times [\alpha] \rightarrow [\alpha]$, and indexing $\mathit{ind} : [\alpha] \times \mathit{Int} \rightarrow \alpha$. Moreover, we have distribution $\mathit{dist} : \alpha \times \mathit{Int} \rightarrow [\alpha]$, where $\mathit{dist}\ (a, n)$ yields $[a, \ldots, a]$ with length n, permutation $\mathit{perm} : [\alpha] \times [\mathit{Int}] \rightarrow [\alpha]$, where $\mathit{perm}\ (\mathit{xs}, \mathit{is})$ permutes xs according to the index vector is, packing $\mathit{pack} : [\alpha] \times [\mathit{Bool}] \rightarrow [\alpha]$, where $\mathit{pack}\ (\mathit{xs}, \mathit{fs})$ removes all elements from xs that correspond to a *false* value in fs. Furthermore, there are families of reduction $\oplus_\tau\text{-}\mathit{reduce} : [\tau] \rightarrow \tau$ and prescan $\oplus_\tau\text{-}\mathit{scan} : [\tau] \rightarrow [\tau]$ functions for associative binary primitives $\oplus_\tau$ operating on data of basic type τ. Finally, FKL contains three functions that form the basis for handling nested vectors: $\mathcal{F} : [[\alpha]] \rightarrow [\alpha]$ correspond to $+\!\!+\text{-}\mathit{reduce}$ and removes one level of nesting, e.g., $\mathcal{F}\ ([[1, 2, 3], [], [4, 5]]) = [1, 2, 3, 4, 5]$; $\mathcal{S} : [[\alpha]] \rightarrow [\mathit{Int}]$ corresponds to $\{\#\mathit{xs} : \mathit{xs} \leftarrow \mathit{xss}\}$ and returns the toplevel nesting structure, e.g., $\mathcal{S}\ ([[1, 2, 3], [], [4, 5]]) = [3, 0, 2]$; and $\mathcal{P} : [\alpha] \times [\mathit{Int}] \rightarrow [[\alpha]]$ reconstructs a nested vector from the results of the previous two function, e.g., $\mathcal{P}\ ([1, 2, 3, 4, 5], [3, 0, 2])$ $= [[1, 2, 3], [], [4, 5]]$. For each nested vector xs, we have $\mathcal{P}\ (\mathcal{F}\ (\mathit{xs}), \mathcal{S}\ (\mathit{xs})) = \mathit{xs}$. We write the application of primitives like $+\!\!+$ infix. Some of the primitives are polymorphic (those with α in the type) and, as said, we discuss their instantiation later, but we assume that in an FKL program all polymorphism in user-defined functions is already removed (by code duplication and type specialization).

To compensate the lack of general nested parallelism (i.e., no apply-to-each construct), FKL supports all primitive functions $p : T_1 \times \cdots \times T_n \rightarrow T$ in a vectorized form $p^\uparrow : [T_1] \times \cdots \times [T_n] \rightarrow [T]$, which applies p in parallel to all the elements of its argument vectors (which must be of the same length). For example, we have $[1, 2, 3] +^\uparrow_{\mathit{Int}} [4, 5, 6] = [5, 7, 9]$. In general, a primitive and its vectorized form relate through $p^\uparrow(\mathit{xs}_1, \ldots, \mathit{xs}_n) = \{p(x_1, \ldots, x_n) : x_1 \leftarrow \mathit{xs}_1, \ldots, x_n \leftarrow \mathit{xs}_n\}$. Note, the part of the transformation generating FKL guarantees that nothing like $(p^\uparrow)^\uparrow$ is needed. The next subsection introduces two additional sets of primitives: One handles recursive types and the other, although not strictly necessary, handles some operations on nested vectors more efficiently. We delay the discussion of these primitives as it benefits from knowing the target data representation.

We choose FKL as the target language as there are optimizing code-generation techniques mapping it on different parallel architectures [BCH+93,Cha93,Kel98].

3.2 Concrete Data Representation

Before presenting the instantiation procedure for polymorphic primitives, we discuss an efficient target representation of nested vectors, vectors of tuples, and vectors of recursive types.

Nested vectors. We can represent nested vectors of basic type using only flat vectors by separating the data elements from the nesting structure. For example, $[[1,2,3],[],[4,5]]$ is represented by a pair consisting of the data vector $[1,2,3,4,5]$ and the *segment descriptor* $[3,0,2]$ containing the lengths of the subvectors. The primitives $\mathcal{F}$ and $\mathcal{S}$ extract these two components, whereas $\mathcal{P}$ combines them into an *abstract* compound structure representing a nested vector. In general, this representation requires a single data vector and one segment descriptor per nesting level of the represented vector. Instances of polymorphic primitives operating on these nested vectors can be realised by combining primitives on vectors of basic type with the functions $\mathcal{F}$, $\mathcal{S}$, and $\mathcal{P}$, as we will see below.

This representation allows an optimized treatment of costly reordering primitives, such as permutation. Consider the expression $perm'\ (as, is)$, where as is of type $[[Int]]$. Both the data vector and the segment descriptor of as have to be permuted. We have $perm'\ (as, is) = \mathcal{P}\ (perm\ (\mathcal{F}\ (as), is'), perm\ (\mathcal{S}\ (as), is))$, where $perm$ operates on vectors of type Int and is' is a new permutation vector computed from is and $\mathcal{S}\ (as)$. So, for example, $perm'\ ([[3,4,5],[1,3]],[1,0]) = \mathcal{P}\ (perm\ ([3,4,5,1,3],[2,3,4,0,1]), perm\ ([3,2],[1,0]))$.

This scheme is expensive, because (a) a new index vector is' is computed for each level of nesting, and moreover, (b) the data vector is permuted elementwise, whereas the original expression allows to deduce that several continuous blocks of elements (the subvectors) are permuted, i.e., we lose information about the structure of communication. We can prevent this behaviour by employing an additional set of primitive functions, the so-called *chunkwise operations distC*, *permC*, and *indC*. The operation *distC* gets a vector together with a natural number and yields a vector that repeats the input vector as often as specified by the second argument. The operations *permC* and *indC* both get three arguments: (1) a flat data vector, (2) a segment descriptor, and (3) an index vector or index position (depending on the function). They permute and index blocks of the data vector chunkwise, where the chunk size is specified by the segment descriptor. Their semantics is defined as $permC\ (xs, s, is) = perm'\ (\mathcal{P}\ (xs, s), is)$ and $indC\ (xs, s, i) = ind'\ (\mathcal{P}\ (xs, s), i)$, respectively, where $perm'$ and ind' operate on vectors of type $[[s]]$ when xs is of type $[s]$ and s is a basic type. Their implementation using blockwise communication is straightforward.

Vectors of tuples. Vectors of tuples are represented by tuples of vectors. Accordingly, applications of vector primitives operating on such vectors are pulled inside the tuple, as proposed by [Ble90].

Recursive Types. The most complicated case is the representation of vectors of recursive types by vectors of basic type in such a way that the nodes of each level of the represented tree structure are stored in a separate vector (as discussed in Section 2). In Subsection 3.1 we required that, in a recursive type $\mu x.T$, each occurrence of x in T has the form $[x]$. Let us consider the possible contexts of these occurrences. If the outermost type constructor of T is a primitive type (e.g., Int), there is no recursion and we represent T as usual. But, if the outermost type constructor of T is a vector, we have $[[x]]$, i.e., a nested vector of recursive type. Similar to nested vectors of basic type, we regard them as an abstract structure manipulated by the three functions $\mathcal{F}$, $\mathcal{S}$, and $\mathcal{P}$. Next, if the outermost type constructor of T is a tuple, we have to represent it as a tuple of vectors like in the non-recursive case, while treating the components of the resulting tuple in the same way as T itself. Finally, if the outermost type constructor of T is a recursive type of the form $\mu x'.T'$, no special treatment is necessary, apart from handling T' in the same way as T.

Overall, any occurrence of a recursive type variable x (except in the root of a tree) is represented by $[[x]]$, due to the propagation of vectors inside tuples and the requirement that x occurs only as $[x]$. Thus, these occurrences are manipulated by $\mathcal{F}$, $\mathcal{S}$, and $\mathcal{P}$. These functions allow us to represent a tree structure of potentially unbounded depth using only flat vectors by separating data from nesting structure. Nevertheless, a problem remains: For nested vectors, the nesting depth of the data is statically known (due to strong typing), but not so for the depth of a recursive type. Hence, we get a sequence of segment descriptors of statically unknown length. Each of these is accompanied by the data vector for one level of the tree. A value of recursive type $\mu x.T$ is always terminated by empty vectors of a recursive occurrence x in T, but due to the propagation of vectors inside tuples, we need a special value to identify the recursion termination. Hence, we wrap the representation of vectors of recursive type into a *maybe* type $\langle\alpha\rangle$, which is either $\langle x\rangle$ (where x is of type α) or $\langle\rangle$. To process values of type $\langle\alpha\rangle$, we add two primitives $(\cdot ?) : \langle\alpha\rangle \to \mathit{Bool}$ and $(\cdot\uparrow) : \langle\alpha\rangle \to \alpha$. Operation $(\cdot ?)$ yields true if the argument has the form $\langle x\rangle$, in this case, $(\cdot\uparrow)$ returns x (it is undefined, otherwise). The tree of Figure 1 is represented by (c_0, v) with

$$\begin{array}{ll|ll}
v = \langle(([c_1, c_2, c_3, p_1])\rangle, & \mathcal{P}\ (w, [2, 2, 2, 0])) & x = \langle(([p_6, p_7, p_8, p_9], & \mathcal{P}\ (y, [0, 0, 0, 0]))\rangle \\
w = \langle(([c_4, c_5, p_2, p_3, p_4, p_5], & \mathcal{P}\ (x, [2, 2, 0, 0, 0, 0]))\rangle & y = \langle(([], & \mathcal{P}\ (\langle\rangle, []))\rangle
\end{array}$$

3.3 Instantiation of Polymorphic Functions

Now, we are in a position to define the instantiation of the polymorphic uses of primitives. We denote the type of an instance of a primitive by annotating the type *substituted for the type variable* (α in the signatures from Subsection 3.1) as a subscript. For example, $+\!\!+_{[\mathit{Int}]}$ has type $[[\mathit{Int}]] \times [[\mathit{Int}]] \to [[\mathit{Int}]]$, and we have to generate instances for all occurrences apart from $+\!\!+_{\mathit{Int}}$ and $+\!\!+_{\mathit{Bool}}$. The generic primitives τ^n, π^n_i, $\#$, $\mathcal{F}$, $\mathcal{S}$, and $\mathcal{P}$ are not instantiated as their code is independent of the type instance.

The transformation algorithm. In the presence of recursive types, we cannot transform the program by replacing uses of polymorphic primitives with expressions that merely use primitives on basic type; instead, we add new declarations to a program, until all uses of primitives are either directly supported or covered by a previously added declaration. The addition of a declaration may introduce primitives occurring at new instances, which in turn triggers the addition of declarations for these instances. We discuss the termination of this process later.

Each equation says that if a primitive occurs at a type matching the left hand-side, add a declaration that is an instance of the right hand-side for that type. We start with the distribution *dist*, together with its chunkwise variant:

$$\begin{aligned}
\mathit{dist}_{(T_1,\ldots,T_n)}\ (x,n) &= (\mathit{dist}_{T_1}\ (\pi^n_1(x),n),\ldots,\mathit{dist}_{T_n}\ (\pi^n_n(x),n))\\
\mathit{dist}_{\mu x.T}\ (x,n) &= \langle \mathit{dist}_{T\{\mu x.T/x\}}\ (x,n)\rangle\\
\mathit{dist}_{[T]}\ (x,n) &= \mathcal{P}(\mathit{distC}_{[T]}\ (x,n),\mathit{dist}_{\mathit{Int}}\ (\#x,n))\\
\mathit{distC}_{\mu x.T}\ (\mathit{xs},n) &= \textbf{if } \mathit{xs}?\ \textbf{then } \mathit{distC}_{T\{\mu x.T/x\}}\ (\mathit{xs}{\uparrow},n)\ \textbf{else } \langle\rangle\\
\mathit{distC}_{[T]}\ (\mathit{xs},n) &= \mathcal{P}(\mathit{distC}_{[T]}\ (\mathcal{F}\ (\mathit{xs}),n),\mathit{distC}_{\mathit{Int}}\ (\mathcal{S}\ (\mathit{xs}),n))
\end{aligned}$$

The rules for tuples propagate the function inside. Vectors are distributed using the chunkwise version, of which the second rule demonstrates the handling of the maybe type $\langle\alpha\rangle$. We omit the rules for tuples in the following, as they have the same structure as above. The next set of rules covers chunkwise concatenation:

$$\begin{aligned}
\mathit{xs} +\!\!+_{\mu x.T}\ \mathit{ys} &= \textbf{if } \mathit{xs}?\ \textbf{then } (\textbf{if } \mathit{ys}?\ \textbf{then } \mathit{xs}{\uparrow} +\!\!+_{T\{\mu x.T/x\}}\ \mathit{ys}{\uparrow}\ \textbf{else } \mathit{xs})\ \textbf{else } \mathit{ys}\\
\mathit{xs} +\!\!+_{[T]}\ \mathit{ys} &= \mathcal{P}(\mathcal{F}(\mathit{xs}) +\!\!+_T \mathcal{F}(\mathit{ys})),\mathcal{S}(\mathit{xs}) +\!\!+_{\mathit{Int}} \mathcal{S}(\mathit{ys}))
\end{aligned}$$

We continue with plain and chunkwise indexing. Note that indexing an empty vector of recursive type leads to an error as the index is out of bounds.

$$\begin{aligned}
\mathit{ind}_{\mu x.T}\ (\mathit{xs},i) &= \textbf{if } \mathit{xs}?\ \textbf{then } \mathit{ind}_{T\{\mu x.T/x\}}\ (\mathit{xs},i)\ \textbf{else error}\\
\mathit{ind}_{[T]}\ (\mathit{xs},i) &= \mathit{indC}_T\ (\mathcal{F}(\mathit{xs}),\mathcal{S}(\mathit{xs}),i)\\
\mathit{indC}_{\mu x.T}\ (\mathit{xs},s,i) &= \textbf{if } \mathit{xs}?\ \textbf{then } \mathit{indC}_{T\{\mu x.T/x\}}\ (\mathit{xs},s,i)\ \textbf{else } \langle\rangle\\
\mathit{indC}_{[T]}\ (\mathit{xs},s,i) &= \mathcal{P}(\mathit{indC}_T\ (\mathcal{F}(\mathit{xs}),+_{\mathit{Int}}\text{-}\mathit{reduce}^{\uparrow}\ (\mathcal{P}\ (\mathcal{S}(\mathit{xs}),s)),i),\\
&\qquad\quad \mathit{indC}_{\mathit{Int}}\ (\mathcal{S}(\mathit{xs}),s,i))
\end{aligned}$$

For $\mathit{indC}_{[T]}$, the segment descriptor passed to the recursive call is computed by summing over the segments of the current level. For space reasons, we omit the rules for *perm*, *pack*, and *combine*. They are similar to *ind*, but, e.g., permuting an empty vector does not raise an error, but returns the empty vector. Finally, considering vectorized primitives, the rules for types $(T_1,\ldots,T_n)$ and $\mu x.T$ are as above; for $[T]$, the vectorized version is generated by vectorizing the above rules—this vectorization is essentially the same as the first step of the flattening transformation that was mentioned in the beginning of the present section. The details of this and of handling *perm*, *pack*, and *combine* can be found in [KC98].

Termination. Considering the termination of the above process, we see that for each primitive p, the rules for instances $p_{(T_1,\ldots,T_n)}$ and $p_{[T]}$ decrease the structural depth of the type subscript. Merely the rules for $p_{\mu x.T}$ may be problematic as they recursively unfold the type in their right hand-side. However, in the definition of the kernel languages, we required that type variables bound by μ occur only as the element type of a vector. So, for all occurrences where the type unfolding substitutes the recursive type, we get an expression that requires $p_{[\mu x.T]}$. If p is a chunk operation or $+\!\!+$, the right hand-side of an instance of the rule for $p_{[T']}$ with $T' = \mu x.T$ requires $p_{\mu x.T}$, which is exactly the instance we

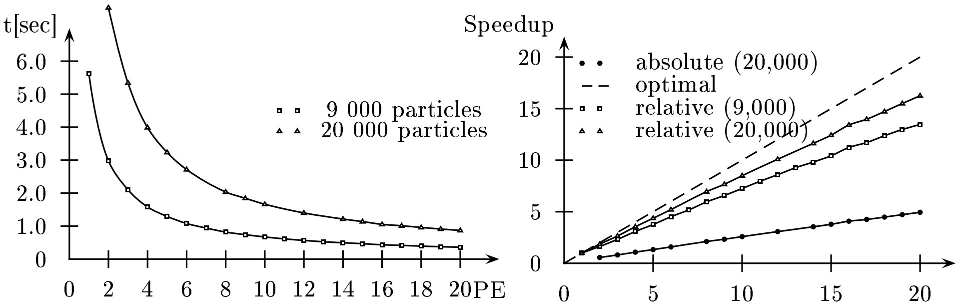

Fig. 2. Absolute time and relative speed-up for Barnes-Hut on a Cray T3E.

started with and which, therefore, is already defined. If p is neither a chunk operation nor ++, it requires the corresponding chunkwise operation and we already discussed the termination for chunkwise operations.

4 Experimental Results

We measured the accumulated runtime of the expressions in the `let`-binding of `bhtree` (Section 2) for the *original* NESL-*program* using 15,000 particles and CMU-NESL [BCH+93] (on a single processor, 200MHz Pentium/Linux machine), to quantify the overhead of mapping the tree structure to a vector. From the overall runtime of 1.75 seconds, 0.5 seconds (29%) are spend on the operations introduced by the mapping. We gathered this data on a sequential machine, due to inaccurate profiling information from CMU-NESL on the Cray T3E. The inefficiencies would even be worse in a parallel run, since these operations include algorithmically unnecessary reordering operations, which cause communication.

To measure our proposed techniques for implementing recursive types, we timed an implementation of Barnes-Hut generated using the presented rules.[1] Figure 2 shows the timing of a single simulation step on a Cray T3E for 9000 and 20000 particles as well as the relative speedup and the absolute speed up (we only had access to 20 processors). The relative speedup is already quite close to the theoretical optimal speedup, but the absolute speedup shows that there is still room for improvement and we already know some possible optimizations. We did not compare CMU-NESL and our implementation on the Cray T3E directly, since our code generation techniques for the flat language already outperform CMU-NESL by more than an order of magnitude [Kel98].

5 Conclusion

After its introduction by Blelloch & Sabot, flattening has received considerable attention, but to the best of our knowledge, we are the first to extend flattening to tree structures. There are other approaches to implementing nested data-parallelism, and of course, there is a wealth of literature on trees in parallel

---

[1] The code was 'hand-generated'; we are currently implementing a compiler.

computing, but we do not have the space to discuss this work here—some of it is discussed in [KC98]. Our extension is easily integrated into existing systems and the gathered benchmarks support the efficiency of the generated code.

Acknowledgements. We are indebted to Martin Simons and Wolf Pfannenstiel for fruitful discussions and helpful comments on earlier versions of this paper. Furthermore, we are grateful to the anonymous referees of EuroPar'98 for their valuable comments and the members of the CACA seminar (University of Tokyo) for lively discussions. The first author receives a PhD scholarship from the DFG (German Research Council). She thanks the SCORE group of the University of Tsukuba, especially, Tetsuo Ida, for the hospitality while being a guest at SCORE.

References

[ACD+97] P. K. T. Au, M. M. T. Chakravarty, J. Darlington, Yike Guo, Stefan Jähnichen, G. Keller, M. Köhler, W. Pfannenstiel, and M. Simons. Enlarging the scope of vector-based computations: Extending Fortran 90 with nested data parallelism. In W. K. Giloi, editor, *Proc. of the Intl. Conf. on Advances in Parallel and Distributed Computing*. IEEE Computer Society Press, 1997.

[BCH+93] G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. In *4th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 1993.

[BH86] J. Barnes and P. Hut. A hierarchical $O(n \log n)$ force calculation algorithm. *Nature*, 324, December 1986.

[Ble90] G. E. Blelloch. *Vector Models for Data-Parallel Computing*. The MIT Press, 1990.

[Ble96] G. E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, 1996.

[BS90] G.E. Blelloch and G. W. Sabot. Compiling collection-oriented languages onto massively parallel computers. *Journal of Parallel and Distributed Computing*, 8:119–134, 1990.

[Cha93] S. Chatterjee. Compiling nested data-parallel programs for shared-memory multiprocessors. *ACM Trans. on Prog. Lang. and Systems*, 15(3), 1993.

[KC98] Gabriele Keller and Manuel M. T. Chakravarty. Flattening trees—unabridged. Forschungsbreicht 98-6, Technical University of Berlin, 1998. http://cs.tu-berlin.de/cs/ifb/TechnBerichteListe.html.

[Kel98] G. Keller. *Transformation-based Implementation of Nested Parallelism for Parallel Computers with Distributed Memory.* PhD thesis, Technische Universität Berlin, Fachbereich Informatik, 1998. Forthcoming.

[KS96] 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*, LNCS. Springer Verlag, 1996.

[PP93] J. Prins and D. 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.

[PPW95] D. Palmer, J. Prins, and S. Westfold. Work-efficient nested data-parallelism. In *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Processing (Frontiers 95)*. IEEE, 1995.