Data-Only Flattening for Nested Data Parallelism

Lars Bergstrom  
John Reppy  
Stephen Rosen  
Adam Shaw  
University of Chicago  
{larsberg,jhr,sirosen,ams}@cs.uchicago.edu

Matthew Fluet  
Rochester Institute of Technology  
mfluet@cs.rit.edu

Mike Rainey  
Max Planck Institute for Software Systems  
mraineyp@mpi-sws.org

Abstract

Data parallelism has proven to be an effective technique for high-level programming of a certain class of parallel applications, but it is not well suited to irregular parallel computations. Blelloch and others proposed nested data parallelism (NDP) as a language mechanism for programming irregular parallel applications in a declarative data-parallel style. The key to this approach is a compiler transformation that flattens the NDP computation and data structures into a form that can be executed efficiently on a wide-vector SIMD architecture. Unfortunately, this technique is ill suited to execution on today’s multicore machines. We present a new technique, called data-only flattening, for the compilation of NDP, which is suitable for multicore architectures. Data-only flattening transforms nested data structures in order to expose programs to various optimizations while leaving control structures intact. We present a formal semantics of data-only flattening in a core language with a rewriting system. We demonstrate the effectiveness of this technique in the Parallel ML implementation and we report encouraging experimental results across various benchmark applications.

Categories and Subject Descriptors  
D.3.0 [Programming Languages]: General; D.3.2 [Programming Languages]: Language Classifications – Applicative (Functional) Programming, Concurrent, distributed, and parallel languages; D.3.4 [Programming Languages]: Processors – Compilers, Optimization

Keywords  
multicore, NESL, nested data parallelism, compilers

1. Introduction

Data-parallel computations are ones in which a function is applied to the elements of a collection (e.g., set or sequence) in parallel. Data parallelism is an effective technique to take advantage of parallel hardware and is especially suited to large-scale parallelism [10], but most languages that support data parallelism limit that support to flat data parallelism (FDP), where the computation being mapped over the collection does not contain nested data parallel computation. While FDP is very effective for many regular parallel applications, it is not well-suited for irregular parallel applications. To address this weakness, Blelloch and others proposed nested data parallelism (NDP) [4, 5, 8, 20].

The basic operation in both flat and nested data parallelism is the parallel map operation, which applies a function to the elements of a collection in parallel. What distinguishes NDP from FDP is that elements of the collection may themselves be collections, and the mapped computation may itself involve parallel maps over the nested collections. Because the nested collections in an NDP computation may vary in size, it is difficult to ensure a balanced partitioning of work across multiple processors and it is difficult to execute the parallel computation with Single-Instruction-Multiple-Data (SIMD) architectures.

Blelloch addressed these challenges with an approach that he called flattening. Flattening (also called vectorisation) is a technique for converting irregular nested computations into regular computations on flat arrays. This approach, which was invented for first-order NDP by Sabot and Blelloch [2, 5] and extended to full-featured higher-order functional languages by Keller, Chakravarty, and others [7, 8, 14, 16], transforms both the data representations and the code so that the computation can be executed by a SIMD machine. An alternative approach, which is used in the Manticore system, is to execute the parallel map operations as fork-join parallelism and to rely on efficient work-stealing techniques to handle load balancing [1, 23].

In this paper, we introduce a new approach to implementing NDP constructs that is based on the idea of flattening the nested data representations, but not vectorising the code [25]. In our prototype implementation, we build on the Manticore compiler for PML [11], a parallel dialect of Standard ML [17].

This paper makes the following contributions:

1. We introduce a novel approach to implementing NDP that is well suited to execution on MIMD architectures, such as modern multicore processors.
2. We provide a formalization of our approach using a core calculus.
3. We present empirical evidence that our approach improves the performance of code that executes over irregular data while also preserving performance on regular data.

2. Full flattening for NDP

As a motivating example, consider the NDP code to implement sparse-matrix times dense-vector multiplication. We represent a dense vector as a parallel array of floating-point values:

\[
\text{type dense_vec = float_parray}
\]

and a sparse vector is a parallel array of index-value pairs:
In this representation, we record only the non-zero entries of the sparse vectors: vector, paired with their indices. A sparse matrix is a parallel array of sparse vectors, illustrated in Figure 1(a) and (b) for a matrix and its sparse representation.

We define multiplying a dense vector by a sparse-matrix as the smvm (sm multiplicand) operator: a parallel reduction that computes the sum of the elements of a parallel array of floats:

$$\text{smvm (sm : sparse_mat, v : dense_vec) = } [\text{sumP (\| x \times \text{sv !} | (\text{sv in sm }) \|)}]$$

The technical challenge is to find the meeting point between this elegant declarative code and an efficient implementation that can exploit powerful multicore architectures.

As one solution to this challenge, Blelloch and Sabot introduced the NESL language and the flattening transformation to compile nested-data-parallel programs for wide-vector parallel machines [2, 5]. NESL is an ML-like language with nested-data-parallel features. NESL’s provides only scalars and associated operators, sequences, simple datatypes, conditionals, let bindings, top-level function definitions, and a parallel apply-to-each construct. Blelloch’s flattening transformation then substantially changes both the data structures and code. This transformation is especially effective in treating irregular parallelism: certain operations like the parallel segmented sum operation, where sums are computed over an array of arrays of numbers, complete in the same number of steps regardless of the irregularity of the shapes of the segments.

In NESL, a segment-descriptor is always a flat vector of integers, and a flattened array carries with it one segment-descriptor for each level of nesting. Figure 1(c) illustrates the flat representation of the example sparse matrix.

### 2.1 Challenges for full flattening

Full flattening, while a successful innovation, can produce inefficient code in common cases, including conditionals and certain regular nested parallel programs. When full flattening is applied to conditionals, the resulting code generates many intermediate vectors, as we demonstrate here in a simplified example. Here is a function that replaces the zeroes in a vector with ones:

```haskell
fun g (xs : dense_vec) = |
  | if x=0.0 then 1.0 else x |
  | in in xs |
```

The transformed code (which uses standard split and combine operations [2]) builds intermediate vectors to handle partitioning the data elements, performs the appropriate conditional work on each partition of the data, and then combines them:

```haskell
fun g_full_flat (xs : dense_vec) = let
  val flags = |
    | x=0.0 |
    | in in xs |
  val (zs, nz, flags) = split (xs, flags)
  val zs' = |
    | 1.0 |
    | in in zs |
  in
  combine (zs', nz, flags)
end
```

This approach makes sense for SIMD architectures with large penalties for failing to keep the vector registers full, but on multicore machines those benefits can be overwhelmed by the large number of memory operations. There are many more extended examples of this kind of transformation in the literature [2, 13]. Our approach, by contrast, will transform g to operate on a flattened dense vector, but will not otherwise transform the code; to the point, it will not necessitate generating any intermediates.

With respect to regular nested parallel programs, problems similar to dense matrix multiplication suffer from a polynomial space increase under full flattening, which is a known serious problem [26] also due to excessive data copying. The present work describes a system designed to address these problems with traditional full flattening by avoiding extra data copies, both due to splitting of conditional branches and duplication of vectors within nested parallel applications.

### 3. Data-Only Flattening

We present data-only flattening in the context of a broader system for hybrid flattening. Hybrid flattening is a program representation that allows both flat and nested representations of parallel arrays and which has coercions for transforming between representations.
This representation allows flexibility in choosing when it is profitable to use a flat representation vs. a nested representation. Hybrid flattening does not itself express or embody a particular transformation policy.

Whereas Manticore without flattening compiles nested parallel arrays to nested ropes [6], with flattening it compiles nested parallel arrays to flattened arrays. Flattened arrays, like nested arrays as compiled by NESL, consist of two pieces: a flat data vector, and a value representing the structure of the nested array called a shape tree. By means of standard unzipping transformations, nested arrays of tuples are compiled to arrays of flattened arrays. In our implementation, flattened arrays are represented by the polymorphic farray datatype. To represent the flat data vector part of flattened arrays, we use ropes, exploiting Manticore’s existing rope infrastructure to compute in parallel with them. Figure 2 presents the PML datatype definitions for rope, shape, for shape trees, and farray. Note that ropes and shape trees are internal representations; the programmer uses them only indirectly.

Shape trees are our adaptation of segment descriptors in the NESL tradition. A shape is an n-ary tree whose leaves store integer pairs. Each leaf contains the starting index and the index of the element forming the segment of data in an farray. The shape Shape.Lf (i, i+n) describes a length-n segment starting at i and ending at the last position before i+n.1

A simple, flat parallel array of integers such as

\[
\text{FArray (Rope.Leaf [1,2,3], Shape.Lf (0,3))}
\]

has the following farray representation:2

\[
\text{FArray (Rope.Leaf [1,2,3], Shape.Lf (0,3))}
\]

The data in the original sequence appears here in the original order as a Rope.Leaf and the accompanying shape indicates that the flattened array’s only segment begins at position 0 and ends at position 2.

Nested parallel arrays are translated as follows. Consider the following nested array:

\[
\text{[[[ [1, 1, 2, 3 ]], [ 1, 3, 4, 5, 6 ] ]]}
\]

Its flattened array representation is the following:

\[
\text{FArray (Rope.Leaf [[1, 1, 2, 3 ], [1, 3, 4, 5, 6 ]], Shape.Nd (Shape.Lf (0,2), Shape.Lf (2,2), Shape.Lf (2,2))} \]

The flat data appears in order in a Rope.Leaf. The shape is a N d with three leaves: this means that the parallel array consists of three subsequences. The leaves tell us that the first sequence begins at position 0 and ends at 1, the second sequence is empty at position 2, and the third sequence begins at position 2 and ends at 5. This representation naturally scales to any nesting depth.

Literal values can be flattened at transformation time directly, with the representation change incurring no runtime cost. In general, however, the compiler must cope with arbitrarily nested arrays whose dimensions are known only at runtime. In such cases, the compiler needs to arrange for flattening to take place at runtime. We handle this issue, by inserting coercion operators that perform runtime flattening. When nested arrays are transformed into flattened arrays, all operations applied to those array values must be correspondingly transformed. Our approach to this problem is to provide a core group of type-indexed families of array operators, each of which performs its operation at every array type in its family.

1 In practice, this choice of bounds is a convenient convention that guards against common fencepost errors.

2 For brevity, we present the sequence in the Rope.Leaf node and the sequence of leaves in the Shape.Nd with list syntax.

\[
\tau ::= \text{g} \quad \text{ground types}
\| \langle \tau, \tau \rangle \quad \text{pairs}
\| \tau \rightarrow \tau \quad \text{functions}
\| \{ \tau \} \quad \text{parallel arrays}
\| \{ \tau ; \nu \} \quad \text{flattened parallel arrays}
\]

\[
\nu ::= \text{If} \quad \text{structure of flat arrays}
\| \text{nd}(\nu) \quad \text{structure of nested arrays}
\]

\[
g ::= \text{int} \mid \text{bool}
\]

\[
t ::= e^\nu \quad \text{ground terms}
\]

\[
e ::= b \quad \text{variables}
\| \text{if } t \text{ then } t \text{ else } t \quad \text{conditionals}
\| \text{let } x = t \text{ in } t \quad \text{let expressions}
\| \text{fun } x = t \quad \text{function expressions}
\| t \circ t \quad \text{function composition}
\| t \circ t \quad \text{application}
\| (t, t) \quad \text{pairs}
\| \pi_i t \quad \text{projection } (i \in \{1, 2\})
\| \{t, \ldots, t\} \quad \text{flattened arrays}
\| \text{nd}(s) \quad \text{arrays}
\| \text{map}(\tau) \quad \text{array subscript}
\| \text{map}(\tau, \tau, \tau, \tau) \quad \text{array map}
\| \text{map}(\tau, \tau, \tau, \tau) \quad \text{array filter}
\| \tau \circ \tau \quad \text{array reduction}
\| \text{red}(\tau) \quad \text{type coercions}
\]

\[
s ::= \text{If} (t, t) \quad \text{leaves}
\| \text{nd}(s, \ldots, s) \quad \text{nodes}
\]

\[
b ::= \text{true} \mid \text{false}
\| 0 \mid 1 \ldots
\| \text{not} \mid + \ldots
\]

Figure 3. Flatland: types.

Figure 4. Flatland: terms.

This group contains parallel array subscripting and parallel maps, filters, and reductions over parallel arrays. All operations on parallel arrays are either members of this core group or are built from members of this group. As such, transformation of the type-indexed operators matching the transformations of data structures is sufficient to preserve the program’s behavior.

4. Formalization

The system presented here, Flatland, consists of a model language and a variety of rewriting systems and judgments. Its model language is an explicitly-typed, monomorphic, strict, pure functional language. Flattened and non-flattened terms commingle in Flatland; there is no inherent distinction between source language and target language.

Figure 3 presents Flatland’s types, ranked over by the metavariable \( \tau \). We use subscript indices \( (\tau) \) and overbars \( (\overline{\tau}) \) to distinguish types from one another. The type language consists of ground types \( \text{int} \) and \( \text{bool} \), pairs, functions, parallel arrays, and flattened parallel arrays. Flattened-parallel types include \( \text{shape} \) types as subcomponents. Shape types, ranked over by the metavariable \( \nu \) (for “nesting”), record an array’s nesting depth.
\[ \Gamma \vdash e_1 \text{bool} \quad \Gamma \vdash e_2 \text{ok} \quad \Gamma \vdash e_3 \text{ok} \]

\[ \Gamma \vdash (\text{if } e_1 \text{bool} \text{ then } e_2 \text{ok} \text{ else } e_3 \text{ok}) \]

\[ \Gamma \vdash \text{map}(e_2 \text{ok}) \quad \tau_1 \text{ A} \tau_3 \]

\[ \Gamma \vdash \text{filt}(e_2 \text{ok}) \quad \tau_2 \text{ A} \tau_1 \]

\[ \Gamma \vdash \text{red}(e_2 \text{ok}) \quad \tau_2 \text{ A} \tau_1 \]

\[ \Gamma \vdash \tau \vdash \tau \text{ ok} \]

\[ \Gamma \vdash [(\tau, \tau_2)] \vdash [(\tau_1, \tau_2)] \text{ ok} \]

\[ \Gamma \vdash \tau_1 \vdash \tau_3 \text{ ok} \quad \Gamma \vdash \tau_2 \vdash \tau_3 \text{ ok} \]

\[ \Gamma \vdash [(\tau_1, \tau_2)] \vdash [(\tau_1, \tau_2)] \text{ ok} \]

\[ \Gamma \vdash \tau_1 \vdash \tau_2 \text{ ok} \quad \Gamma \vdash \tau_2 \vdash \tau_3 \text{ ok} \]

\[ \Gamma \vdash \tau_1 \vdash \tau_3 \text{ ok} \]

\[ \Gamma \vdash e_1 \text{int} \text{ ok} \quad \Gamma \vdash e_2 \text{int} \text{ ok} \]

\[ \Gamma \vdash \text{If}(e_1 \text{int}, e_2 \text{int}) \text{ : if} \]

\[ \Gamma \vdash s_1 : \nu \quad \Gamma \vdash s_n : \nu \quad \Gamma \vdash \text{nd}(s_1, \ldots, s_n : \nu) \]

**Figure 7.** Calculation of shape types.

\[ !\tau = \tau \]

\[ !\{\tau; \nu\} = \tau \]

\[ !\{\tau; nd(\nu)\} = \{\tau; \nu\} \]

\[ !\{\tau_1, \tau_2\} = (!\tau_1, !\tau_2) \]

**Figure 8.** Array-type subscripting.

\[ \tau \text{ L} \tau \equiv \tau_1 \text{ L} \tau_2 \quad \tau_2 \text{ L} \tau_1 \quad \tau_1 \text{ L} \tau_2 \quad \tau_2 \text{ L} \tau_1 \]

\[ \{\{\tau; \nu\} ; \text{lf}\} \text{ L} \tau_3 \]

\[ \{\{\tau; \nu\} ; nd(\nu')\} \text{ L} \{\{\tau; nd(\nu)\} ; \nu'\} \]

**Figure 9.** Definition of \( \tau_1 \text{ A} \tau_2 \) and its auxiliary relation \( \tau_1 \text{ L} \tau_2 \).

\( \Downarrow \) the element selected by subscript out of a value of type \( \tau \). The return type of a particular subscript operator is calculated from its domain. Figure 8 gives the definition of \( !\tau \). If \( !\tau \) cannot be computed from these rules, it is undefined.

In order to understand the typing of the type-indexed operators, one needs to understand the relation \( \text{A} \). For a type \( \tau \), we cannot definitively write \( ![\tau] \) to mean "array of \( \tau", because \( !\{\tau; \text{lf}\} \) is also an array of \( \tau \), albeit in a different representation. Furthermore, if \( \tau \) is a pair type \( \tau_1, \tau_2 \), then \( ![\{\tau_1, \tau_2\}] \) and \( ![\{\tau_1, \tau_2\}] \) (and more) are also arrays of \( \tau \), and so on. Thus we appeal to the relation \( \tau_1 \text{ A} \tau_2 \) for "\( \tau_1 \) is an array of \( \tau_2 \)." For arrays of \( \text{int}, \text{int} \), for example, all

\footnote{Due to space limitations, only selected rules are presented; the full rule sets appear in [25].}
of the following are established by A:

\[ \begin{align*}
& [(\text{int}, \text{int})] \quad \text{A} \quad (\text{int}, \text{int}) \\
& [(\text{int}, \text{int})] \quad \text{A} \quad (\text{int}, \text{int}) \\
& \{ (\text{int} \mid \text{int}) ; \text{lf} \} \quad \text{A} \quad (\text{int}, \text{int}) \\
& \{ (\text{int} \mid \text{int}) ; \text{lf} \} \quad \text{A} \quad (\text{int}, \text{int})
\end{align*} \]

These types correspond to arrays of integer pairs and all equivalent representations under well-formed coercions. The relation A is defined in Figure 9. A is defined by defining an auxiliary relation L that verifies that two types are both representations of the same scalar or array type. We use the name L for the relation because it tells us that one type is “on the same level as” another. All arrays of type τ can be coerced between one another. Formally, if \( \tau_1 \vdash \tau_2 \) and \( \eta_1 \vdash \eta_2 \) (proven elsewhere [25]),

We specify array operators by writing their type indices in a subscript. \( \tau \) is the operator that selects elements from an array type \( \tau \). \text{map}_{\tau_1, \tau_2, \tau_3, \tau_4} \) takes two arguments, a function of type \( \tau_1 \rightarrow \tau_2 \) and a term of array type \( \tau_3 \), and produces a term of array type \( \tau_4 \). \text{filt} and \text{red} carry similar type subscripts. The typing rules of \text{map}, \text{filt} and \text{red} all appeal to A.

To perform a flattening step, we insert a coercion. A standard step in flattening transformations is to unzip arrays of pairs of scalars—that is, to reshape an array of pairs into a pair of arrays. The coercion that unzips an array of integer pairs is written \([(\text{int}, \text{int})] \vdash [(\text{int}, \text{int})]) \). Its inverse coercion \([(\text{int}, \text{int})] \vdash [(\text{int}, \text{int})]) \) is also part of the language of coercions. If it is ever the case that a pair of inverse coercions, like these two, are successively applied to a value, they may be rewritten to an identity coercion, which can in turn be removed from the program.

Well-formed programs are guaranteed to remain well-formed under any legal transformation in our rewriting system. The top-level type of the whole program remains fixed under transformation, but the types of the subexpressions within a program may change. The well-formedness guarantee is maintained by introducing inverse coercions for every coercion introduced into the program, to maintain the stability of the types in the program. For example, every time a coercion is applied to the value to which a variable \( x \) is bound, the inverse coercion is introduced at every use of coerced \( x \). This ensures that no clients of \( x \) are put in a position to compute with a term of the wrong type, post-coercion. Type-indexed operators such as \text{map} and \text{filt} have the ability to absorb type coercions, since, for each one, there are many implementations from which to choose, and this helps reduce the number of type coercions in the transformed program.

To provide intuition about how this machinery works, here is an example of Flatland in action. Consider the following program:

\[ \text{let } \text{ns } = [1, 2, 3] \text{ in } (\text{ns} \downarrow [\text{int}] 0) \]

For brevity, we let

\[ \downarrow = [\text{int}] \triangleright \{ \text{int} ; \text{lf} \} \]

and

\[ \uparrow = \{ \text{int} ; \text{lf} \} \triangleright [\text{int}] \]

Think of \( \downarrow \) as “flatten” and \( \uparrow \) as “unflatten.” The step-by-step transformation of this let-expression appears in Figure 13. The parallel array \([1, 2, 3] \) is transformed to its flattened array equivalent, \( \downarrow [1, 2, 3] \), which evaluates to \( \{1, 2, 3\} \). Furthermore, by rewriting, we exchange one type-indexed subscript operator for another, thereby eliminating coercion operations. The coercions \( \downarrow \) and \( \uparrow \) are inverse coercions. Note by the fourth rule in Figure 10 we have

\[ ([\text{int}] \triangleright \{ \text{int} ; \text{lf} \}) \circ \{ \text{int} ; \text{lf} \} \triangleright \{ \text{int} ; \text{lf} \} \triangleright \{ \text{int} ; \text{lf} \} \]
We now present data-only flattening in the context of the Flatland system, as a transformation from a source language to a target language. A source type is a type that is neither a flattened-array type, nor contains any flattened-array types, nor contains any flattened-array types, generated by the grammar

\[
\tau ::= g | \tau \rightarrow \tau | (\tau \circ \tau) | [\tau]
\]

We define source programs as a term \( e^\tau \) for source type \( \tau \), all of whose subterms have source types and contain no coercions. By this definition, we have made it illegal to write down flattened arrays anywhere in a source program. The transformation will introduce all flattened-array values and all coercions.

The target language is defined in terms of flat types. A type \( \tau \) is flat if

- it is a ground type \( g \),
- it is a function type \( \tau_1 \rightarrow \tau_2 \) and \( \tau_1 \) and \( \tau_2 \) are flat,
- it is a pair type \( (\tau_1, \tau_2) \) and \( \tau_1 \) and \( \tau_2 \) are flat, or
- it is an array type \( \{\tau ; \nu\} \) and \( \tau \) is a ground type or a flat function type.

If a type is not flat, we say it is nonflat. Note that source types and nonflat types are not the same. For example, \( \{\text{int, bool}\} \) is a source type, and the related type \( \{\text{int ; list}, \text{bool ; list}\} \) is a flat type. But the related type \( \{\text{int, bool ; list}\} \) is neither a source type nor a flat type; it is disqualified as a source type since it is a flattened-array type and disqualified as a flat type since it includes a pair inside an array. A target program is an expression whose outermost type is a source type, yet all of whose subexpressions have flat types. The restriction on its outermost type is a consequence of the type-preservation property of Flatland’s rewriting. The restriction is enforced by the application of one last “unflattening” coercion to the transformed program at the top level. Within the program, all subexpressions are flattened.

Flattening of whole programs is written as a type-preserving relation \( \downarrow \). Figure 15 gives the sole judgment for \( \downarrow \), which immediately delegates its work to an auxiliary relation \( \preceq \). Whole-program flattening consists of transforming a program \( e^\tau \) of source type \( \tau \) to a program \( e^{\tau^\ast} \) of flat type \( \tau^\ast \) and then coercing the transformed program to original type \( \tau \) at the top level. Note that in the cases where \( \tau \) is, for example, a ground type (see Figure 13), the outermost coercion is an identity coercion and has no effect.

The auxiliary relation of data-only flattening is given in Figures 16 and 17. The syntax of \( \preceq \) is as follows:

\[
\Delta \vdash e^\tau \preceq (\tau \circ \tau) \circ \tau^\ast
\]

\( \Delta \) is a finite map from variable terms to variable terms; it is used to implement propagations through let-expressions and functions. On the right-hand side of the relation, a diamond \( (\circ) \) is used to construct a pair out of a coercion and a transformed expression. The relation produces an unflattening coercion, along with the transformed expression, for use in one of the following ways. If the expression transformed is the whole program, the \( \downarrow \) relation applies the coercion to preserve the program’s original type (as per the rule in Figure 15). If the expression is not the whole program, the accompanying coercion is used as a building block for further coercions as program transformation proceeds outward.

The important work in data-only flattening takes place at array terms; see the second rule in Figure 17 (which in turn appeals to...


5. Implementation

Data-only flattening in PML is accomplished in three successive phases: an abstract flattening phase, whereby abstract flattening operations — symbolic values that stand in for actual implementations — are inserted throughout the code; a fusion phase where canceling coercions (adjacent coercions that undo one another’s work) are eliminated; and a concrete flattening phase, where symbolic flattening operations are replaced by monomorphic code. Due to space limitations, the detailed operations of these phases is not presented here and may be found elsewhere [25].

5.1 Optimizations

Flattened PML programs are amenable to various optimizations that cannot be applied to non-flattened ones. In this section, we discuss several such optimizations, each of which is responsible, in part, for the performance improvements we report in our benchmark results.

Monomorphization. Monomorphization is an optimization whereby a polymorphic data structure containing uniformly-represented (i.e., boxed) elements is transformed to a representation containing raw (unboxed) elements in their place. The flattening transformation, by virtue of unzipping arrays of tuples, exposes more opportunities for monomorphization than otherwise. In PML, arrays of double pairs, for example, become pairs of double arrays, which in turn become farrays, each containing a specialized rope of doubles as its flat data vector. Monomorphization is well known to be valuable even outside the context of nested data parallel compilation. MLton [18], an optimizing whole-program SML compiler, performs monomorphization to generate better-performing sequential code. PML stands to benefit from monomorphization even without flattening (PML currently does no monomorphization unless flattening is enabled), although it will never be the case that, without unzipping tuples, non-flattened PML will have as many opportunities to do it.

Tab flattening. Nested parallel comprehensions over ranges have regular structure: at each dimension, the length of every array is fixed a constant. The regularity of such structures can be exploited by the tab flattening optimization, which performs simple integer arithmetic operations to collapse multidimensional tabulations into linear ones.

Every one-dimensional parallel comprehension of scalars is trivially regular:

\[
\text{val xs} = \{ \text{Double.fromInt i | i in [| 0 to 9 |]} \}
\]

The straightforward, and inefficient, implementation of this parallel comprehension is to translate it to a map over the parallel array containing the integers from 0 to 9.

\[
PArray.map \text{Double.fromInt [| 0 to 9 |)}
\]

This naïve translation entails building an ephemeral data structure that is immediately computed with and discarded. To save the cost associated with this intermediate structure, the compiler rewrites parallel comprehensions over ranges as tabulations:

\[
PArray.tabulate (10, \text{Double.fromInt})
\]

Tabulating over integer intervals requires no intermediate data structures, and realizes a performance improvement over the build-and-map strategy outlined above.

Nested parallel comprehensions naturally give rise to nested tabulations. The computation of \(\text{xss}\) in this excerpt

\[
\text{val xss} = \{ | i*10+j | | j in [| 0 to 9 |]}\}
\]

\[
| i \in [| 0 to 9 |] \}
\]

can be naturally expressed by a tabulate within a tabulate as follows:

\[
PArray.tabulate (10, \text{Double.fromInt})
\]

\[
PArray.tabulate (10, \text{fn i => PArray.tabulate (10, \text{fn j => (i*10 + j))}}
\]

This translation is already better than using maps with ephemeral structures, but the shape of our flattened array representations allows us to use tab flattening to improve on nested tabulations. Recall our evaluation of \(\text{xss}\) results in an farray containing a flat data vector and a shape tree. We name the result \(\text{xssF}\) and sketch it as follows:
val xssF = FArray (Rope.Leaf [0, 1, 2, ..., 99],
Shape.Nd [Shape.Lf (0,10),
  ... Shape.Lf (90,100)])

We can generate the flat data vector of xssF in one tabulation,
over a single counter representing the total number of elements in
the nested array, by performing the appropriate index arithmetic on
the counter:

let fun \( f \) = let
  val (i, j) = (k div 10, k mod 10)
  in
  \( (i+10) + j \)
  end

in
PArray.tabulate (10*10, f)
end

The shape tree in rectangular cases has a simple regular structure as
well, and be computed from the dimensions of a regular array in a
straightforward way. Tab flattening operation scales to any number
of dimensions for regular nested arrays.

Segmented reductions. NEST’s fast segmented operations are an
important element of NEST’s ability to perform well on irregular
designed data parallel programs, and an important one for PML to
emulate. NEST’s segmented sum operation, for example, is able to
compute the sums of a nested array of numbers in a fixed number
of steps regardless of the irregularity of the array’s structure. This
operation is critical to the performance of sparse-matrix/vector
multiplication (see below). Here is an example of an irregular sum
computation in PML:

let val nss = [1 [1, 2],
  [1],
  [1, 3, 4, 5, 6]]

in
PArray.reduce add 0.0 nss
end

If this parallel comprehension is rewritten such that the sum
operation is simply mapped over the array-valued elements of nss, the
irregularity of the structure of nss, in which there is wide variation in
the lengths of its elements, is bound to affect load balancing adversely.
As such, the compiler replaces nested reductions like these

val ss = [| PArray.reduce add 0.0 ns | ns in nss |]

with “segmented reductions”

val ss = PArray.segreduce add 0.0 nss

The non-flattened implementation of PArray.segreduce
oper ident is simply PArray.map (PArray.reduce
oper ident). As part of the flattening transformation, the non-
flattened segreduce implementation is replaced by a faster
segregate written to exploit the shape of farray data struc-
tures.

6. Evaluation

These benchmarks compare the performance and scalability of
Manticore’s current implementation of parallel arrays [1] against
data-only flattening. The baseline performance is from the sequen-
tial version of the benchmark, which runs on a single processor and
eliminates all overhead from the parallel language constructs and
associated runtime features. The sequential version does not take
advantage of any of the data transformations provided by data-only
flattening, relying on the default polymorphic array representation.

These benchmarks were selected to show that we gain perform-
ance on code that executes over irregular data, as in SMVM,
while preserving most of the performance of benchmarks over
regular data. Flattening happens at compile time in the follow-
ing sense: nested arrays such as those in the main expressions of
mandelbrot and raytracer are created flat in the first place,
not constructed nested and then flattened.

6.1 Experimental method

Our benchmark machine is a Dell PowerEdge R815 server, outfitted
with 48 cores and 128 GB physical memory. This machine runs
x86_64 Ubuntu Linux 10.04.2 LTS, kernel version 2.6.32-42. The
48 cores are provided by four 12 core AMD Opteron 6172 “Magny
Cours” processors. Each core operates at 2.1 GHz and has 64 KB
each of instruction and data L1 cache and 512 KB of L2 cache.
There are two 6 MB L3 caches per processor, each of which is
shared by six cores, for a total of 48 MB of L3 cache.

We ran each experiment configuration 30 times, and we report
the average performance results in our graphs and tables. Times are
reported in seconds.

6.2 Mandelbrot

We compute the Mandelbrot set by means of a function elt which
consumes a pair of integers and produces an integer. The argument
to elt represents a location in the complex plane. Its return value
is the number of iterations required, according to the standard
iterating Mandelbrot set membership test, for a given point to
diverge outside the set (by having a modulus greater than 2). A
point is a member of the Mandelbrot set if it fails to diverge before
reaching a fixed upper limit of iterations (we use 1000). We execute
this simple function in parallel over a 2048 \( \times \) 2048 range using the
following PML code:

fun mandelbrot n = let
  val rng = [| 0 to (n-1) |
  in
  [| elt (i, j) \| j in rng \| i in rng |]
end

Figure 18(a) shows PML speedups, with and without flattening,
against the sequential baseline. Due to the relatively small amount
of computation at each element, the benefits of the data-only flat-
tening transformation provide only a 5% speedup at 48 cores.
These benefits come from the reduced amount of memory traffic
when using a monomorphic array represetation, avoiding an ex-
tra allocation per result element and associated garbage collector
pressure.

6.3 Raytracer

Our ray tracing benchmark computes the image of a scene graph
consisting of a group of overlapping spheres with transparency and
reflection. The code is translated from a parallel program in the
implicitly-parallel language Id90 [19]. It is a brute-force imple-
mentation and does not use any acceleration data structures. The
Raytracer benchmark renders a 2048 \( \times \) 2048 image in parallel as a
two-dimensional sequence.

Similar to the Mandelbrot benchmark, we write the body of the
main function as a nested parallel comprehension:

fun raytracer n = let
  val ns = [| 0 to (n-1) |
  in
  [| trace (i, j) \| j in ns \| i in ns |]
end

In all versions of the program, the Manticore compiler transforms
this nested parallel comprehension into a two-dimensional tabu-
lation over fixed ranges, avoiding creation of an intermediate ar-ray of values to iterate over. In the data-only version of this pro-
gram, that two-dimensional tabulation is instead performed as a
Figure 18. Comparison of flattened and non-flattened performance.

6.4 Sparse-Matrix Vector Multiplication

Among our benchmarks, sparse-matrix vector multiplication profits most from the data-only flattening transformation. Sparse-matrix vector multiplication is expressed concisely in the following PML code:

```pml
fun smvm (sm, v) = let
  fun add (a,b) = a+b
  in
    [ | PArray.reduce add 0.0
      [ | x*(v!i) | (i,x) in sv |]
    | sv in sm |]
  end
```

The inner parallel array expression computes the dot-product of the sparse matrix and the vector. The outer parallel array then computes the sum of each of those resulting dot-products. The summation is implemented on top of `PArray.reduce`, which allows the runtime to process the reduction in parallel.

In both versions of the benchmark, the Manticore compiler automatically optimizes the parallel reduction over multiple values.
into a single segmented reduction. Segment operators have been shown by Blelloch and others to result in more balanced chunks of parallel work, across a variety of platforms [3, 24]. All versions are transformed into the following PML code:

```pml
fun smvm (sm, v) = let
  val prods = products (sm, v)
  val sums = segsum prods
  in
    sums
  end
```

The flattened version of `smvm` uses monomorphic vectors for both the intermediate representation of the dot-products and the final result of the segmented reduction. Further, the implementation of segmented reduction over monomorphic vectors takes advantage of the layout, performing the segmented reduction with far less overhead than the sequential and non-flattened versions.

Figure 18(c) gives the speedups of PML over its sequential baseline, both with and without flattening. The sparse matrix is 10,000 × 10,000, with a random number of entries between 100 and 500 in each row. Flattened `smvm` is substantially faster than non-flattened `smvm` for all numbers of processors up to 48, and furthermore has much better performance with respect to the sequential baseline. The super-linear speedups are due to the relatively small amount of work performed on each element compared to the improvement due to the representation change. Above 36 processors, our performance improvement flattens due to having insufficient data to take advantage of the processors. Unfortunately, limitations in the Manticore runtime currently prevent us from further increasing the size of the data.

6.5 Dense Matrix Multiplication

The dense matrix multiplication (DMM) benchmark is a dense-matrix by dense-matrix multiplication in which each matrix is 600 × 600. As mentioned in Section 2, this benchmark has traditionally had extremely poor performance under flattening. As shown in Figure 18(d), our approach does still result in a slowdown in performance due to the creation of some intermediate arrays (resulting in a factor of 3 increase in memory usage). This penalty is roughly 13% and could be reduced through the introduction of additional fusion operations to avoid those intermediate arrays.

6.6 Conclusion

These benchmarks demonstrate that the data-only flattening transformation significantly improves a benchmark with irregular data (SMVM), does not experience the polynomial blowup typical to full flattening on DMM, and does not dramatically change the performance of other programs.

7. Related work

The incremental extension of NESL’s foundation to a more feature-rich platform has ultimately taken the form of Data Parallel Haskell [9]. Chakravarty et al. first present the language Nepal in 2001 [8], characterized as a version of Haskell including nested data parallelism. Nepal is succeeded by Chakravarty et al.’s Data Parallel Haskell [9], bringing together Nepal-style nested data parallelism with Haskell as implemented in the Glasgow Haskell Compiler (GHC) [12]. In 2008, Peyton Jones et al. give a thorough overview of the Data Parallel Haskell language and its compilation [22], including an updated account of how Data Parallel Haskell uses the flattening transformation in their implementation. The especially germane question is this: what happens after a system has compiled a Data Parallel Haskell program in the manner of NESL, and yet executes on an SMP rather than a SIMD machine? Their answer to this technical problem is to adapt the `split` and `join` mechanisms originally presented in Keller’s dissertation [13] to implement NESL-style operations across the multiple processing elements on a multicore computer. Parallel computations are `split` across processing elements and subsequently `joined` on completion. To eliminate unnecessary synchronization points, GHC uses rewrite rules [21] to erase successive applications of `split` and `join` (per the identity equivalence rule originally given by Keller). (In addition to this, various advanced fusion techniques are employed to streamline the resulting post-flattened program, an overview of which is given in their paper.)

Though Data Parallel Haskell represents broad advances in NESL-style flattening in numerous ways, and although it has been adapted to run on multicore machines, its compilation strategy continues to reflect the SIMD orientation of its predecessors, though there has been recent work on vectorisation in DPH to avoid excessive flattening [15]. This work differs from that line of research by never performing the full vectorisation transformation on the code, though both approaches flatten nested data.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grants CCF-0811389 and CCF-1010568, and upon work performed in part while John Reppy was serving at the National Science Foundation. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations or the U.S. Government.

References


