Data Parallel Haskell: a status report

Manuel M. T. Chakravarty[†] Roman Leshchinskiy[†] Simon Peyton Jones[‡]
Gabriele Keller[†] Simon Marlow[‡]

[†]Computer Science & Engineering
University of New South Wales
{chak,rl,keller}@cse.unsw.edu.au

[‡]Microsoft Research Ltd
Cambridge, England
{simonpj,simonmar}@microsoft.com

Abstract

We describe the design and current status of our effort to implement the programming model of nested data parallelism into the Glasgow Haskell Compiler. We extended the original programming model and its implementation, both of which were first popularised by the NESL language, in terms of expressiveness as well as efficiency. Our current aim is to provide a convenient programming environment for SMP parallelism, and especially multicore architectures. Preliminary benchmarks show that we are, at least for some programs, able to achieve good absolute performance and excellent speedups.

1. Introduction

One of the most promising approaches to making efficient use of parallel hardware is *data parallelism*, in which a single computation is performed in parallel across a large number of data elements. For example, High-Performance Fortran (HPF) and OpenMP exploit data parallelism by employing many processors to process different parts of a single array; and SMPD programming with MPI extends the same idea to a distributed setting.

Typically, the arrays are required to be *flat* (arrays of floats, for example), but that is often quite inconvenient for the programmer. In ground-breaking work in the 90's, the NESL language and its implementation offered data-parallel operations over *nested data structures* (such as arrays of variably-sized subarrays), and allowed all subarrays to be simultaneously computed in data-parallel [2, 1].

This paper describes our progress in taking NESL's good ideas and incorporating them in Haskell, a widely-used, purely functional programming language. Doing so required us to generalise many aspects of NESL's design, and the project we describe here represents the culmination of a ten-year research programme. We make several significant contributions:

- We are building support for nested data parallelism in Haskell, a fully-fledged programming language; and we are doing so for the world's leading Haskell compiler, GHC. GHC's implementation of Haskell already supports two different paradigms for parallel programming: (a) explicit control parallelism, coordinated with transactional memory [20, 10], and (b) semi-implicit concurrency, based on annotations [28]. Our goal is to add a third paradigm, data parallelism; we believe that there is no one silver bullet for expressing parallelism.
- The crucial breakthrough in NESL was the *flattening* or *vectorisation* transformation, which transforms the nested program such that it manipulates only flat arrays. We have extended this transformation in several directions: we support not only built-in types, product, and array types (like NESL), but also handle user-defined, sum, and function types [5, 14]—the latter are particularly challenging (Sections 4 and 5).
- Data parallel programs typically generate many intermediate arrays. We improved on the implementation of NESL by developing *fusion techniques* that completely eliminate many of these intermediates, which dramatically reduces the constant-factor overhead [6, 7] (Section 6.3).
- When fusing *parallel* computations, we need to be careful not to reduce opportunities for parallel execution too much, and we need to minimise communication operations where possible. To guide fusion in the presence of parallelism and to structure the mapping of array operations onto concurrent threads, we developed type-directed data distribution [11] (Sections 6.1 and Section 6.2).
- Our entire implementation is strongly typed, including the compiler intermediate language. This is a real challenge, because the representation of data-parallel arrays is *non-parametric*; for example, an array of pairs is represented as a pair of arrays. We have developed an extension to Haskell's type system called associated types [4, 3], and a new extension of the typed intermediate language [26] that together solve this problem (Section 4).
- Our implementation is carefully structured so that most of it is presented as a stack of Haskell *libraries*, rather than as pervasive modifications to a Haskell *compiler*. The only aspects that must be built into the compiler itself are the support for array comprehensions (syntax, type checking, desugaring; Section 2), and the vectorisation transformation (Section 5).

In combination, these innovations lead to a very efficient implementation of a highly expressive form of data parallelism. To be fair, this claim remains to be tested. We are currently implementing a fully-fledged version of Data Parallel Haskell in the Glasgow Haskell Compiler (GHC) – a state of the art implementation of Haskell – but the implementation is still incomplete.

This paper describes the current state of play: it sketches our main contributions, gives some details of the implementation, and

DAMP 2007 16 January 2007, Nice, France.

presents some first benchmarks.[1] It does not attempt to describe the various program transformations employed in our implementation in technical, or even, formal detail—a comprehensive treatment is well beyond the scope of a workshop paper. Instead, we illustrate all major components of our approach by example and provide references to publications focusing on particular sub-problems, where such publications are already available.

2. What the programmer sees

In Data Parallel Haskell, parallelism is expressed implicitly, by operations over a built-in type of *parallel arrays*, denoted by `[:.:]`. For example, `[:Float:]` is the type of (dense) vectors of floating point values:

```
type Vector = [: Float :]
```

As another example, a sparse vector can be efficiently represented by a (dense) array of (index,value) pairs:

```
type SparseVector = [:(Int, Float):]
```

The dot product of a sparse and a dense vector is easily computed:

```
dotp :: SparseVector -> Vector -> Float
dotp sv v = sumP [: x * (v!:i) | (i,x) <- sv :]
```

Here, the subexpression `[:x * (v!:i) | (i,x) <- sv:]` is an *array comprehension*, of type `[:Float:]`. Its value is obtained by computing, for each element `(i,x)` of the sparse vector `sv`, the product of `x` with the element of the dense vector `v` at position `i`, where `(!:)` denotes indexing. Finally, we compute the sum of the intermediate vector of products, using the function `sumP`[2]

```
sumP ::  [:Float:] -> Float
```

The operational intuition is that `dotp` is executed by a *gang* of threads, one per processing element. Each thread in the gang, a *gang member*, is responsible for a chunk of the sparse vector; the gang member computes the appropriate part of the intermediate vector, adds it up, and the sub-totals are combined to give the final result. By using a tree-like reduction algorithm, `sumP` ensures that the parallel step complexity of `dotp` is logarithmic in the size of the sparse vector. (Of course, it is essential that addition is associative.) It is also essential that, unlike Haskell's normal arrays, parallel arrays are head-strict; that is, if the array is computed at all, then all of its elements are computed. This property allows us to compute the whole array at once, in data-parallel.

A crucial advantage of our approach is the ability to *nest* parallel arrays and computations without restricting the available parallelism or introducing undue inefficiencies. For instance, a sparse matrix can be represented naturally as an array of rows where each row is a sparse vector:

```
type SparseMatrix = [:SparseVector:]
```

To multiply such a matrix with a dense vector, we simply compute the dot product for each row (this formulation of the algorithm is due to [1]):

```
smvm :: SparseMatrix -> Vector -> Vector
smvm sm v = [: dotp row v | row <- sm :]
```

This is a nested parallel computation: we apply `dotp` to each row of the matrix in parallel, but `dotp` is already a parallel operation. This kind of parallelism is notoriously difficult to deal with. In particular, it would be naive to divide the work among the gang by simply giving an equal number of rows to each gang member, since the number of elements in each row may vary greatly. However, by combining a specialised representation of nested arrays with code vectorisation as we describe in Section 4, we are able to automatically translate the above program into code containing only flat parallelism that can be executed efficiently on stock hardware.

The rest of this paper uses `dotp` as a running example to explain the successive steps through which the program is compiled to run efficiently on parallel shared-memory machines. The example is small enough that it can be understood, and the techniques scale to real programs, as NESL has shown.

3. The big picture

The translation of high-level nested data parallel programs, as described in the previous section, into efficient low-level code involves a large number of source-to-source program transformations. Many of these transformations have been part of GHC's optimiser for a long time, in particular a sophisticated inliner, worker-wrapper unboxing, and constructor specialisation [19, 22, 21, 18]. In the course of the Data Parallel Haskell project, we are adding more, array-specific transformations. Due to GHC's generic support for program transformations — specifically, the inliner and rewrite rules [22, 18] — we can implement most of these new transformations as library code, as opposed to extending the compiler itself. The exception being the vectorisation transformation described in Section 5.

Figure 1 illustrates the anatomy of the array-specific transformations. The three major components are (1) *flattening* that transforms nested into flat data parallelism; (2) *fusion* that eliminates redundant synchronisation points and intermediate arrays, and thus drastically improves locality of reference; and (3) *gang parallelism* that maps parallel operations to a gang of threads. All transformations apply to both data types (labelled "– Data –") and operations on these data types (labelled "– Control –"). The figure names these data types and operations and lists the sections in which they are introduced and explained.

The basic idea is that, after flattening has eliminated all nesting, we use a notion of *distributed types* to explicitly distinguish by type the sequential and the parallel components of a data parallel program. Both components are subjected to their own fusion mechanism; i.e., *stream fusion* and *communication fusion*, respectively. Finally, we distribute the fused parallel components across a gang of threads, each of which operates on a chunk of each array.

4. Non-parametric array representation

Standard arrays in Haskell are parametric; i.e., the array representation is independent of the type of array elements. This is achieved by using arrays of pointers refereeing to the actual element data. Such a *boxed* representation is very flexible, but it is also detrimental to performance. The indirections consume additional memory, increase memory traffic, and decrease locality of memory access. The resulting runtime penalty can be of two orders of magnitude.

Consequently, GHC already offers non-standard *unboxed* arrays for applications where array performance matters. However, these unboxed variants are only available for arrays of basic type. If we want efficiency and convenience for parallel arrays, we need the performance of unboxed arrays for arrays of arbitrary user-defined algebraic datatypes.

Hence, a key aspect of our compilation strategy is a *non-parametric representation of parallel arrays* — for each array, we select an efficient representation based on the type of its elements [5]. For instance, a value of type `[:Int:]` is held as a

[1] All code of our system is publicly available. For details, please refer to `http://haskell.org/haskellwiki/GHC/Data_Parallel_Haskell`

[2] In reality, `sumP` is overloaded using type classes, to have type `Num a => [:a:] -> a`. In this paper, though not in the implementation, we ignore that complication.

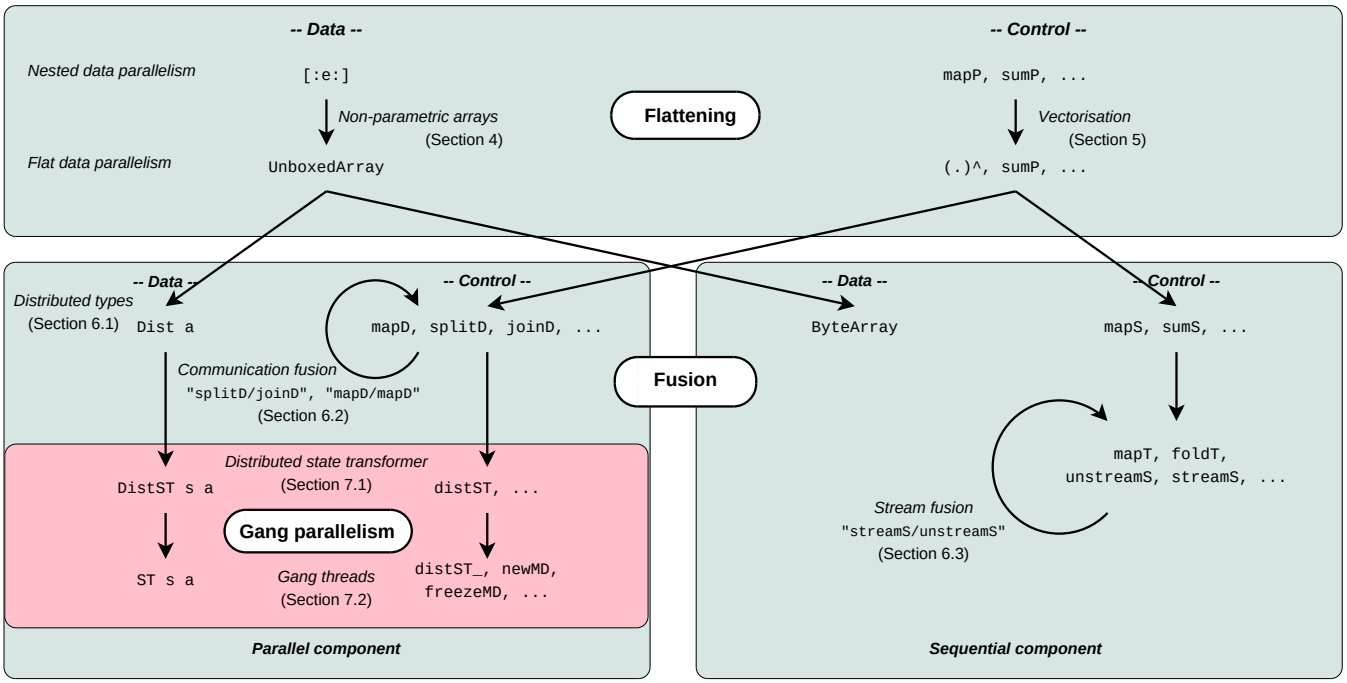

Figure 1. Structure of program transformations

contiguous memory area containing unboxed 32-bit integer values — not as a block of pointers to `Int`-valued thunks, as is the case in vanilla Haskell. In our notation of *associated types* [4], we declare a type whose representation varies in dependence on a type argument as part of a Haskell type class that also contains elementary operations on that type; for arrays we have

```
class ArrElem e where
  data [:e:]
  (!:) :: [:e:] -> Int -> e
```

Our concrete implementation is more complex with more operations, but the code shown here conveys the basic idea. The class instance for integers takes the following form:

```
class ArrElem Int where
  data [:Int:] = ArrInt ByteArray
  (ArrInt ba) !: i = indexIntArray ba i
```

We represent the array by a contiguous region of bytes (aka `ByteArray`) with an indexing primitive `indexIntArray` that extracts a single 32-bit integer from a `ByteArray`. (The code again abstracts over the concrete implementation by omitting the use of unboxed types.)

The `ArrElem` instance for `Float`, and other primitive types, follows the same pattern. But what about more complex data structures, such as `SparseVector`, which is a parallel array of *pairs*? It is quite unacceptable to represent it by an array of pointers to (heap-allocated) pairs, because the indirection costs would be too heavy. Instead, we represent it by a *pair of arrays*:

```
class (ArrElem a, ArrElem b) => ArrElem (a, b) where
  data [: (a,b) :] = ArrPair [:a:] [:b:]
  (arr1, arr2) !: i = (arr1 !: i, arr2 !: i)
```

Thus, a `SparseVector` is represented by two unboxed arrays, one storing the indexes of non-zero elements and one the actual floating point values of those elements. Crucially, the two arrays must have the same length, a constraint which cannot be expressed in Haskell's type system but is maintained by our implementation. Notice that the representation is *compositional*; that is, the representation of an array of pairs is given by combining the representations of an array of the first and second elements of the pair, respectively. This representation also allows us to combine two arrays elementwise into an array of pairs in constant time.

More interesting is the representation of nested arrays. Since ultimately, our goal is to eliminate nested parallelism, it is not surprising that we also want to represent nested arrays in terms of flat ones. Indeed, a nested array `[:[:a:]:]` can be encoded by

- a flat *data array* `[:a:]` which contains the data elements and
- a *segment descriptor* of type `[:(Int, Int):]` which stores the starting position and length of the subarrays embedded in the flat data array.

This is captured by the following instance:

```
class ArrElem a => ArrElem [:a:] where
  data [:[:a:]:] = ArrArr [:a:] [:(Int, Int):]
  (ArrArr arr segd) !: i = sliceP arr (segd !: i)
```

where `sliceP` extracts a subarray from a larger array in constant time. Thus, the sparse matrix

```
[:[:(0,15),(2,9),(3,20):], [::], [:(3,46):]:]
```

will be represented as

```
ArrArr (ArrPair [:0,2,3,3:]     -- data
                [:15,9,20,46:])
       (ArrPair [:0,3,3:]       -- segment
                [:3,0,1:])      -- descriptor
```

where the first array contains all the column indexes, the second one all the `Floats`, and the third and fourth the start indexes and lengths of the segments, respectively. Since all four arrays will be unboxed, programs which process such matrices can be compiled to highly efficient code.

5. Vectorisation

It is important to understand that the programmer does not need to be concerned with how parallel arrays are represented. This greatly simplifies the programmer's task, at the expense of increased complexity in the compiler, which has to generate code working on flat, unboxed arrays from nested programs written under the assumption of a parametric, boxed representation. This is the job of *code vectorisation* (also called *flattening*), a compiler transformation which eliminates nested parallelism.

In the literature, vectorisation is often performed on the array-comprehension syntax directly, but we break it into two steps: first we desugar the comprehension syntax into ordinary function applications, and then we vectorise the latter. For us this is crucial, because otherwise vectorisation would have to treat the *entire* Haskell language (before desugaring), whereas with our approach we can defer vectorisation until the Haskell program has been desugared into GHC's small intermediate lambda-calculus language [26].

For the `dotp` example of Section 2, the desugared form is as follows:

```
dotp :: SparseVector -> Vector -> Float
dotp sv v = sumP (mapP (\(x,i) -> x * (v!:i)) sv)
```

The main idea of the vectorisation step is to generate for each function in the program a *lifted* version which works on arrays instead of individual values. For instance, where *scalar* multiplication, `(*)`, computes the product of two `Floats`, *lifted* multiplication computes the element-wise products of two arrays of `Floats`:

```
(*^) :: [:Float:] -> [:Float:] -> [:Float:]
```

Whenever a function is used in a parallel context, code vectorisation replaces it by its lifted version. Vectorisation turns the desugared definition of `dotp` into this:

```
dotp :: SparseVector -> Vector -> Float
dotp (ArrPair is xs) v = sumP (xs *^ bpermuteP v is)
```

Here we can see that the *scalar* multiplication and array indexing, clearly visible in the desugared version of `dotp`, are replaced by *lifted* multiplication and backwards permutation, `bpermuteP`:

```
bpermuteP :: [:a:] -> [:Int:] -> [:a:]
bpermuteP v is = [: v!:i | i <- is :]
```

Moreover, notice that since the first argument of `dotp` is a vector of pairs — recall the definition of `SparseVector` in Section 2 — its representation is a pair of vectors `(ArrPair is xs)`, and `dotp` works directly on this representation. The indices `is` are used to perform a back permute on the dense vector, extracting those values for which the corresponding elements in the sparse vector are non-zero. The result is then multiplied elementwise with `xs` and, finally, the sum of the array of products is computed.

Strictly speaking, we should have used lifted indexing, `(!:^)`, instead of back permute in the above code. The use of back permute is an optimisation, which may be realised by a specialised transformation rule triggered by `v` being free in the lambda abstraction passed to `mapP`. However, in our implementation, it is effectively realised by the fusion transformation described in the next section. We can't discuss fusion in sufficient detail to demonstrate the `bpermute` optimisation in this paper, though — which is why we take the liberty to introduce back permute here.

Since `dotp` appears in the sparse-matrix vector multiplication `smvm` inside another array comprehension, we also need a lifted version, `dotp^`, of `dotp`. We omit `dotp^` here for space reasons. More details are in [14], which discusses vectorisation of arbitrary nestings in a higher-order language. Ultimately, we need basic operations, such as `(*)` and `sumP`, in their original and their lifted form. Everything else can be broken down to those two variants of the basic operations.

6. Fusion

Code that uses vector operations suffers from a major source of inefficiency: it introduces too many intermediate arrays and consequently thread synchronisation points. The vectorisation transformation makes matters worse, by introducing yet more intermediate arrays. For instance, the vectorised version of `dotp` creates two temporary arrays containing the results of the back permute and the elementwise multiplication, respectively. On parallel hardware, where all array operations are executed by a gang of threads, these threads must synchronise to signal the completion of each temporary. Both the temporary arrays and the resulting thread synchronisation are unnecessary: ideally, we would like each gang thread to traverse its local chunk of the sparse vector, extracting values from the dense vector, multiplying and adding in one go.

The process of collapsing a pipeline of collective operations into a single loop is called *fusion*, and has been extensively studied (e.g. [30, 23, 16, 12, 15]). It is much easier to achieve in a purely-functional context, where most prior work concerns fusion of *list* operations, often called *deforestation* (e.g. [29, 9, 27, 24]). In our approach, however, we have to deal with arrays, with the substantial additional complication of having to fuse *parallel* operations. Our running `dotp` example demonstrates the latter tension: on a sequential machine one would fuse the entire `dotp` algorithm into a single sequential loop, but that would obviously be hopeless on a parallel machine. What we must do instead is *first* to split the computation into chunks, one chunk for each gang thread, and *then* to perform aggressive fusion. In turn, that requires us to express the "chunking" strategy explicitly in the intermediate language, rather than hide it in the runtime system, so that the post-chunking program is exposed to fusion transformations.

Our mechanism for exposing the chunking is called *distributed types* [11], which we discuss next in Subsection 6.1, followed by removal of synchronisation points by fusing phases of parallel computations in Subsection 6.2, and finally removal of temporary arrays by array fusion in Subsection 6.3.

6.1 Distributed types

Our main vehicle for distinguishing between synchronisation and computation is the type `Dist a` of distributed values. For instance, `Dist Int`, pronounced "distributed `Int`", denotes a collection of *local* integers, such that there is one local integer value per gang member. Arrays can be distributed, too: `Dist [:Float:]` is a collection of local array *chunks*, again one per gang member, which together make up the array. Arrays are distributed across gang members and joined back together by the following functions:

```
splitD :: [:a:] -> Dist [:a:]
joinD  :: Dist [:a:] -> [:a:]
```

Distributed values support a number of operations, most importantly mapping:

```
mapD :: (a -> b) -> Dist a -> Dist b
```

While `splitD` and `joinD` denote synchronisation, `mapD` is the main means of implementing parallel computation phases: the gang members concurrently apply the (purely sequential) function to their respective local values.

The above is sufficient to express a wide range of parallel operations. For instance, `bpermuteP` can be implemented as

```
bpermuteP :: [:a:] -> [:Int:] -> [:a:]
bpermuteP xs is =
  joinD (mapD (bpermuteS xs) (splitD is))
```

Here, we distribute the index array across the gang, then apply a sequential backpermute (`bpermuteS`) concurrently to each chunk, and join the results back together. Analogously, elementwise multiplication is defined as

```
(*^) :: [:Float:] -> [:Float:] -> [:Float:]
xs *^ ys =
  joinD (mapD mult (zipD (splitD xs) (splitD ys)))
  where mult (lxs, lys) = zipWithS (*) lxs lys
```

The argument arrays are distributed with `splitD` and then chunkwise combined by `zipD`:[3]

```
zipD :: Dist a -> Dist b -> Dist (a, b)
```

Thus, each gang member processes a pair of local chunks, computing the local products, which are then joined together.

Note that while `(*^)` distributes both arrays, in the case of `bpermuteP`, the data array `xs` is *not* distributed because the entire array is required for each local computation. Distributed types permit us to make explicit the distinction between local values, which are accessible only by a particular thread, and global ones, which are available to all threads.

By now, the definition of `sumP` in terms of distributed types should hold no surprises:

```
sumP :: [:Float:] -> a
sumP xs = sumD (mapD sumS (splitD xs))
```

It distributes the argument array `xs`, has each gang member sum its local portion sequentially with `sumS`, and finally sums up the local results with `sumD :: Dist Float -> Float`.

In this exposition, we only discuss distributed types and fusion for operations on flat arrays. However, the presented approach scales to nested arrays, using the representation based on segment descriptors outlined in Section 4.

6.2 Removing synchronisation points

Distributed types make the structure of parallel computations quite explicit, but how can they be used for fusing pipelines of such computations? Let us revisit the vectorised code for `dotp` from Section 5. By inlining[4] the definitions of `(*^)` and `sumP`, it can be rewritten as follows. (In Haskell, `(.)` denotes function composition and `($)` is function application with very low operator precedence.)

```
dotp (ArrPair is xs) v =
  sumD . mapD sumS . splitD . joinD . mapD mult
  $ zipD (splitD xs) (splitD (bpermuteP v is))
```

Ignoring the call to `bpermuteP` for the moment, the above contains two parallel computation phases (the two `mapD`) with a join/distribute phase (the `splitD . joinD`) in between. It is easy to see that the latter is unnecessary — instead of first joining and then again distributing the result of the elementwise multiplication, it can be used directly to compute the sum. Thus, we can replace the subexpression `splitD . joinD` by the identity.

In this example, `splitD . joinD` has no effect at all. In general, a `splitD/joinD` combination may perform load balancing (e.g., after filtering a distributed array). However, to keep matters simple, we ignore load balancing for the moment and assume that we want to apply the following rewrite rule whenever it matches:

```
"splitD/joinD" forall xs. splitD (joinD xs) = xs
```

GHC has support for specifying such rewrite rules directly in libraries in the form of source code pragmas [18], and this is how we implement the various rewrite rules that we need for fusion.

Applying the `splitD/joinD` rule to `dotp`, we get

```
dotp (ArrPair is xs) v =
  sumD . mapD sumS . mapD mult
  $ zipD (splitD xs) (splitD (bpermuteP v is))
```

This code still contains a superfluous synchronisation: `mapD mult` causes each gang thread to compute its share of elementwise products and then to synchronise with the rest of the gang. However, there is no need for this, as the next operation `mapD sumS` is also purely thread-local. Hence, we want to apply a distributed types version of the well known map fusion law:

```
"mapD/mapD" forall f g xs.
  mapD f (mapD g xs) = mapD (f . g) xs
```

Operationally, this means that two adjacent computation phases with no global operations in between can be combined into a single one. Applying the `mapD/mapD` rule, we get

```
dotp (ArrPair is xs) v =
  sumD . mapD (sumS . mult)
  $ zipD (splitD xs) (splitD (bpermuteP v is))
```

Now, we are left with a synchronisation between `mapD` and `sumD`. This is genuinely required, since the global reduction needs to access all local sums.

Let us now consider the subexpression to the right of the `($)`. When we inline `bpermuteP`, we notice that its outermost `joinD` cancels the enclosing `splitD` by the `splitD/joinD` rule discussed previously; hence we get (for just the `zipD` subexpression)

```
zipD (splitD xs) (mapD (bpermuteS v) (splitD is))
```

Applying the rewrite rule

```
"zipD/mapD" forall xs f ys.
  zipD xs (mapD f ys) =
    mapD (\(x, y) -> (x, f y)) (zipD xs ys)
```

followed by the `mapD/mapD` rule, inlining `mult`, and performing two standard simplifications, we end up with the following optimised code for the dot product:

```
dotp (ArrPair is xs) v =
    sumD
  . mapD (\(lxs, lis) ->
          sumS . zipWithS (*) lxs (bpermuteS v lis))
  $ zipD (splitD xs) (splitD is)
```

Assuming constant-time implementations of the distributed types primitives `splitD` and `zipD`[5] all that is missing for an efficient, parallel implementation of `dotp` is to fuse the purely sequential array operations `sumS`, `zipWithS`, and `bpermuteS`. After sequential array fusion, which we discuss in the following subsection, each gang member essentially forms and sums up the local products, and then, `sumD` combines the individual contributions into the final result.

6.3 Removing temporary arrays

We implement sequential array fusion using the same mechanism for rewrite rules that we employ to remove synchronisation points. However, given the plethora of collective array operations, we need to reduce those to a very small set of elementary fusable recursive

[3] We could write this slightly more concisely with `zipWithD`, but the given form is more useful for the following fusion discussion.

[4] Much of what follows depends crucially on inlining library code into user-written programs. Fortunately, GHC supports cross-module inlining, allowing the libraries to be pre-compiled while still retaining the high-level form for later inlining by the library's clients.

[5] Zipping of distributed values is a constant-time operation since, similarly to parallel arrays, a `Dist` of pairs is internally represented as a pair of `Dists`; `splitD` is implemented in terms of constant-time array slicing (cf. Section 4).

array operations — not unlike foldr/build or destroy/unfoldr fusion for lists. The crucial point here is that we minimise the number of different *recursive* functions that partake in fusion. There is no problem in having a wide range of non-recursive interface functions on top of the small set of fusable, recursive ones. Starting from our earlier work on equational array fusion [6], we recently developed a new fusion framework, which we call *stream fusion*, as it models array traversals as a stream of array elements.

Here, we can only sketch the basic ideas behind stream fusion. However, we previously presented a particular instance of stream fusion, namely stream fusion for byte strings, which has been highly successful and produces code competitive with C [7]. Nevertheless, fully fledged array fusion presents additional challenges; in particular, streaming of segmented arrays and the stream-based implementation of permutation operations. We are planning to explain the details in a forthcoming paper.

The essential abstraction behind stream fusion is the notion of a lazy stream, `Stream e`, of array elements `e`:

```
data Stream e = forall s. -- existential type
                Stream (s -> Step s e) !s Int
data Step s e = Done
              | Skip     !s
              | Yield !e !s
```

Such a stream consists of a *seed* of type `s`, a *stepping function* of type `(s -> Step s e)`, and a *size hint* of type `Int`. The stepping function incrementally produces the stream of elements from the seed. The size hint bounds the number of elements that may be produced, which is useful to optimise memory allocation. The stream contains three types of elements: `Done` flags the end of the stream, `Skip` indicates dropped array elements, and `Yield` gives a single array element of type `e`.

We move between arrays and streams with the two functions

```
streamS   :: [:e:] -> Stream e
unstreamS :: Stream e -> [:e:]
```

which give rise to the following main *stream fusion rule*:

```
"streamS/unstreamS" forall s.
  streamS (unstreamS s) = s
```

We manipulate streams with stream operators, including

```
mapT     :: (a -> b) -> Stream a -> Stream b
foldT    :: (b -> a -> b) -> b -> Stream a -> b
zipWithT :: (a -> b -> c)
         -> Stream a -> Stream b -> Stream c
```

that enable us to express array processing as stream processing. For example, the sequential array operations used in our running example `dotp` are implemented as follows:

```
sumS             = foldT (+) 0 . streamS
bpermuteS a      = unstreamS . mapT (a!:) . streamS
zipWithS f a1 a2 =
  unstreamS (zipWithT f (streamS a1) (streamS a2))
```

After inlining these definitions into the optimised definition of `dotp` given at the end of the previous subsection, the stream fusion rule can be applied twice to eliminate the two temporary arrays, and so turn the lambda abstraction into

```
\(lxs, lis) -> foldT (+) 0
  $ zipWithT (*) (streamS lxs)
                 (mapT (a!:) (streamS lis))
```

All temporary arrays are gone. We are left with streams only. Why is this better? Its better as the stream processors are not recursive and so easily optimised by inlining. The optimised implementation of streams is beyond this paper, but the results of stream fusion for byte strings [7] and our preliminary results for arrays in Section 8 support the claim of the efficiency of streams.

7. Gang parallelism

In the previous section, we discussed how we decompose data-parallel array operations into genuinely parallel operations (such as `splitD`, `joinD`, `mapD`) and purely sequential operations (such as `mapS`, `zipWithS`, `bpermuteS`) which are executed simultaneously by all members of a thread gang. Overall, this leaves us with three kinds of code: (1) non-array Haskell code, (2) parallel operations on distributed types, and (3) sequential array code distributed across a gang. The first category, non-array Haskell code, can include explicit concurrency operators, such as `forkIO`, which implies that we may get data parallelism in multiple, explicitly forked threads.

We deal with this situation by mapping both array parallelism and GHC's explicit parallelism to the same set of thread and synchronisation primitives provided by the runtime. In particular, the thread gangs executing data-parallel operations consist of standard GHC threads. This allows us to delegate the problem of scheduling multiple gangs which compete with each other and with non-gang threads for a limited numbers of processing elements to GHC's scheduler. It remains to be seen how the scheduler must be improved to efficiently handle advanced scenarios involving multiple gangs and user-created threads.

7.1 Distributed state

Parallel arrays are two phase data structures: they are initialised by destructive updates, but are restricted to read-only access as soon as initialisation is complete. Thus, while parallel arrays present the user with a purely functional interface, their initialisation is *stateful* behind the scenes. Haskell provides support for such structures in the form of the *state transformer monad* `ST` which captures stateful computations in a referentially transparent manner [13]. A computation of type `ST s a` transforms a state indexed by the type `s` and produces a value of type `a`. The `ST` monad provides operations for allocating and updating mutable arrays, which are finally *frozen*, which is when they become immutable and can be used outside of their state transformer. The implementation of `unstreamS` follows this pattern: it fills a newly allocated mutable array with elements produced by the stream and then freezes it.

While this is sufficient for sequential code, parallel arrays are initialised *simultaneously* by several gang members, each of which initialises its local chunk. In other words, the gang threads transform the same *distributed* state in parallel. We capture this idea in the form of a *distributed state transformer monad* `DistST`, which we embedded into `ST` by the following operation:

```
distST :: DistST s a -> ST s (Dist a)
```

Given a stateful distributed computation of type `DistST s a`, `distST` executes it concurrently once with each gang thread. The local results (of type `a`) of the gang threads are collected in a distributed value `Dist a`. The semantics is quite similar to that of `mapD` which executes a *pure* computation on each gang thread and, indeed, `mapD` is implemented in terms of `distST`:

```
mapD :: (a -> b) -> Dist a -> Dist b
mapD f d = runST $
  distST (do {x <- myD d; return (f x)})
```

The function `runST :: (forall s. ST s a) -> a` encapsulates the execution of a state transformer in a purely functional context, in a safe manner. Moreover, `myD` extracts the current thread's local value of the distributed value `d`:

```
myD :: Dist a -> DistST s a
```

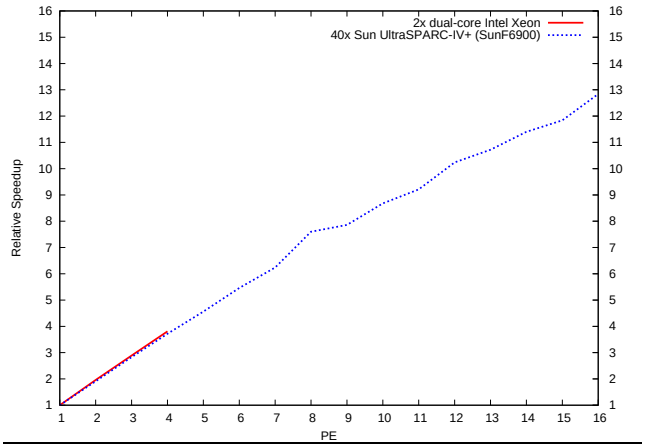

Figure 2. Speedup of smvm on Intel Xeon and Sun Fire (UMA)

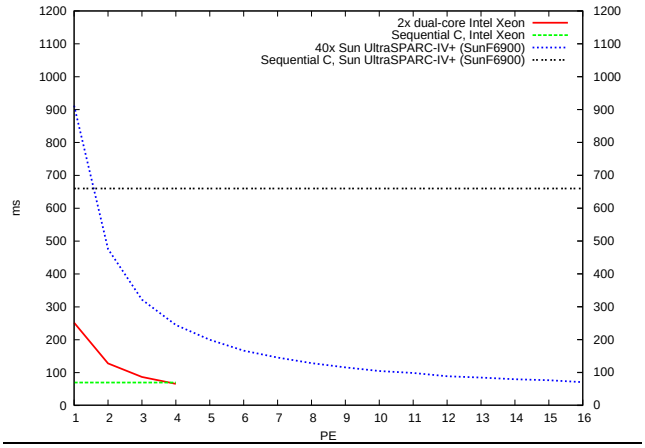

Figure 3. Run time of smvm on Intex Xeon and Sun Fire (UMA)

The function `myD` is stateful, because conceptually, the identity of the thread it is executed on forms part of the distributed state.

7.2 Gang threads

Like parallel arrays, distributed values are two phase data structures, which are initialised destructively. Thus, `distST` itself is implemented in terms of more primitive combinators:

```
distST p = do
  d <- newMD g                  -- alloc dist val
  distST_ (do { x <- p          -- run gang thread
              ; writeMyMD d x}) -- write result
  freezeMD d                    -- make immutable
```

Here, a new mutable distributed value is allocated (`newMD`), filled by the gang members (`writeMyMD`), and then frozen (`freezeMD`) to be consumed by pure code. The workhorse of the implementation is `distST_`: it passes a computation to each gang thread and then blocks until all threads have completed. Its signature is

```
distST_ :: DistST s () -> ST s ()
```

In our set up, a gang thread goes through the following simple loop: (1) it waits for a `DistST s ()` computation, to be issued by a call to `distST_` in a vanilla Haskell thread; (2) executes the computation, (3) signals its completion, and (4) then blocks until the next computation arrives.

This simple work distribution scheme is possible due to a beneficial interaction between code vectorisation and gang parallelism: vectorisation eliminates nested parallelism, thus ensuring that the computations executed by gang threads never need to perform parallel operations themselves. This crucial property removes the need for a work queue which would be necessary if parallel operations could be nested at this lowest level.

8. Preliminary results

Data Parallel Haskell is very much work in progress. For instance, the syntactic sugar and the compiler transformations described in Sections 2 and 5 have not been implemented yet. Moreover, the creation and scheduling of gangs is, so far, rather ad hoc. The library of parallel arrays and algorithms, however, is already quite usable, including automatic fusion of both sequential and parallel computations. This allows us to validate our approach by *manually* vectorising our running example and measuring its performance.

Since our implementation is entirely based on concurrency primitives provided by the production version of GHC, it is fairly portable and we were able to run the benchmarks on three quite

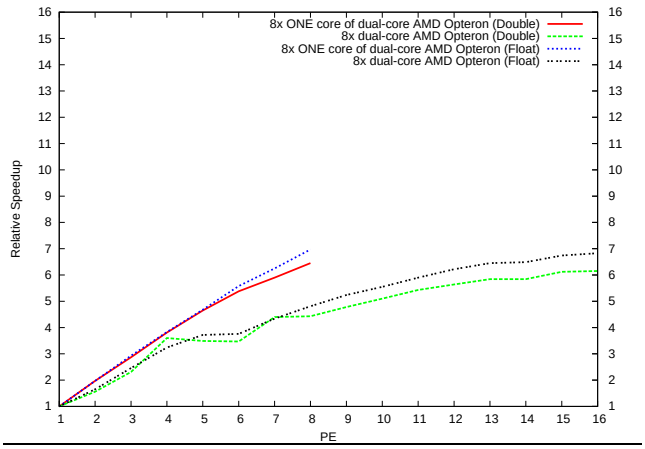

Figure 4. Speedup of smvm on AMD Opteron (NUMA)

different architectures: (1) an Intel-based SMP with two dual-core Xeon CPUs running under Linux with a 2.6.15 kernel, (2) a Sun Fire E6900 with 32 Sparc processors running under Solaris 5.10 (but due to system resource allocation policies, we were only able to use 16 of the 32 PEs at a time), and (3) an AMD64 machine (using a HyperTransport bus) with eight dual-core Opteron CPUs running under Linux with a 2.6.19 kernel. All Haskell code was compiled with GHC 6.7 and the sequential C reference implementation was compiled with gcc 4.1.

Figure 2 shows the speedups of vectorised `smvm` with a $10,000 \times 10,000$ sparse matrix with approximately 1 million non-zero `Double` elements on the Intel Xeon and Sun Fire machines. The speedups are linear as expected, since after fusion the algorithm runs almost entirely in parallel. Moreover, again due to the fusion framework, our library does not introduce any significant inefficiencies — the difference between the running times of the parallel algorithm on one PE and a purely sequential version are fairly small (912 vs. 830ms on the Sun Fire and 252 vs. 249 ms on Intel).

Even more importantly, we did not only achieve good speedup, but the absolute running time of our implementation is not far from hand-coded C. Figure 3 shows absolute running times of the same benchmark including the running time of a sequential C reference implementation. On the Sun Fire, a sequential C implementation of the benchmark runs in approximately 660ms; the parallel Haskell version requires 920ms on one PE and is already faster with two

PEs, where it takes 476ms. These results are for using gcc for *both* compiling the C program and as a backend for GHC. Sun's cc runs our C implementation of the algorithm four times faster than gcc on the Sun Fire, but we cannot use it as a backend for GHC, as GHC uses some gcc-specific language extensions. On the Intel Xeon the parallel Haskell program runs in 252ms on one core, compared to 70ms needed by the C version, and needs four PEs to outperform the sequential C code. The performance difference between the C and Haskell code is largely due to inadequacies of GHC's backend code generator, which has never been optimised for numerically intensive code; in particular, it uses the fairly few registers available on IA32 very ineffectively.[6] Register pressure is less of an issue on the Sparc architecture, which we believe is the main reason for the better relative performance of GHC on that architecture. The good news is that we believe that this performance gap can be closed, or at least strongly reduced, by adding some well understood backend optimisations to GHC — a task that is largely independent of our approach to compiling nested data parallelism.

The two machines discussed so far have a uniform memory access (UMA) architecture, whose memory latency is uniform across the whole address space. Furthermore, the Sun Fire provides a very high memory bandwidth, which is particularly important in `smvm` since only few arithmetic operations are performed for every load and store. In contrast, the AMD Opteron machine has a *non*-uniform memory acccess (NUMA) architecture, so that access to memory "near" to (i.e. physically adjacent to) a processor is faster than access to "distant" memory (i.e. memory attached to other processors). The effect of NUMA becomes apparent in the speedup graph for for the Opteron, which is is displayed in Figure 4. Here we use a $40,000 \times 20,000$ sparse matrix with approximately 8 million non-zero elements. We run this benchmark in four variants by varying it along two dimensions: (a) we use `Float` or `Double` and (b) we use one or two cores per CPU. As long as we use only one core per CPU, the speedup is not far from that achieved on the Sun Fire. However, if we use both cores of every CPU, we see a significantly reduced speedup. After some experiments with small kernels, we believe that we can attribute this behaviour to two properties of the hardware: (1) the main problem is that the memory bandwidth of the HyperTransport-based bus is simply not sufficient to saturate the arithmetic capacity of two cores for `smvm` and, (2) to a lesser extent, we see some reduced performance due to a large proportion of the memory traffic being to "distant" memory. We ensured load balancing of memory traffic by using the Linux NUMA utility `numactl` to set a memory interleave policy. However, if memory allocation in GHC were NUMA-aware (which it currently is not), we could optimise memory allocation for arrays by setting suitable memory affinity. This would surely improve matters, but probably not dramatically, because the main limitation of `smvm` on this hardware is simply the available memory bandwidth. Interestingly, the absolute performance of the parallel Haskell code is 1298ms, on one core, and 900ms for the purely sequential C reference code, which is much closer than on the Intel Xeon. We still need to investigate the reason for this in detail. It is most likely due to differences between the x86-64 and IA32 architecture: either gcc's backend for the x86-64 is less optimised, or the x86-64 architecture is an easier target for GHC, especially as it eases the register pressure compared to IA32.

Finally, we would like to emphasise that the results presented here are preliminary. Small changes, be it to our library implementation, to GHC, or to the compilation strategy, can have surprisingly large effects. Nevertheless, our results constitute a constructive proof that it is possible to compile at least one data-parallel Haskell program with a standard compiler onto a range of shared-memory multiprocessors such that (a) it is competitive in absolute performance, and (b) it scales with adding processors. Our goal is to extend this result to a much wider variety of programs.

[6] This is even the case when using gcc as the backend, as GHC forces gcc to reserve some registers for global use.

9. Related Work

We have drawn our inspiration from the seminal work on NESL and its implementation by program transformation [1, 2]. To this groundbreaking work, we added (a) the integration of NESL's parallel programming model into a fully-fledged functional language, (b) the generalisation of vectorisation to Haskell's full type structure including functionals, (c) a comprehensive fusion framework for distributed arrays, (d) the lifting of compiler magic into library code with associated types and rewrite rules, and (e) a type-preserving translation.

Prins et al. worked on various aspects of the vectorisation of nested data parallel programs; see, e.g., [17]. Most of their work was also in the context of a functional language, but one that like NESL lacks many of Haskell's features. Their work is largely orthogonal to ours.

So et al. [25] developed a parallel library of immutable arrays for C/C++ supporting what they call *sub-primitive fusion*. The goals and ideas behind this fusion framework are rather similar to those discussed in Section 6. However, So et al. do not require inlining of user-defined functions for fusion and they also introduce a light-weight synchronisation mechanism. Like us, they also strive for a seamless integration of data parallelism and explicit concurrency within a single program.

Fluet et al. [8] recently started the manticore project, where they combines CML-style explicit concurrency with nested data-parallelism. They introduce an approach to support multiple scheduling disciplines in one runtime system, much like the scheduling we will also need to efficiently combine explicit concurrency with nested data parallelism. We expect to to be able to directly benefit from the results of that project.

There is a rich body of work on parallel programming models and implementation techniques for functional languages (both data parallel and task parallel). However, a comprehensive discussion of this work is beyond the present paper.

10. Conclusions

We described the design and implementation status of our current effort to support nested data parallelism in the highly optimising Glasgow Haskell Compiler, such that it co-exists elegantly with existing support for two forms of more explicit parallel programming. Our implementation is partial, but we chose a bottom up approach to implementation, where we can conduct rigorous performance tests of more low-level components of our implementation before moving further up. Currently, we have a parallel library of flat and segmented arrays, including automatic fusion of both sequential and parallel computations. For a sparse-matrix vector multiplication benchmark, this library achieves good absolute performance and excellent speedups on Intel IA32, Sun Sparc, and AMD x86-64 SMP machines.

We still have a lot of work ahead of us before we have a complete system, but the current results indicate that we are so far in good shape. Our current focus is on SMP, and especially, multicore machines. However, nested data parallelism, and we believe also our implementation strategy, extend to both distributed-memory parallel systems as well as to SIMD and stream processors (such as GPUs) — or even to heterogeneous systems combing them.

References

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

[2] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, April 1994.

[3] Manuel Chakravarty, Gabriele Keller, and Simon Peyton Jones. Associated type synonyms. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'05)*, Tallin, Estonia, 2005.

[4] Manuel Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. Associated types with class. In *ACM Symposium on Principles of Programming Languages (POPL'05)*. ACM Press, 2005.

[5] Manuel M. T. Chakravarty and Gabriele Keller. More types for nested data parallel programming. In Philip Wadler, editor, *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, pages 94–105. ACM Press, 2000.

[6] Manuel M. T. Chakravarty and Gabriele Keller. Functional array fusion. In Xavier Leroy, editor, *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP'01)*, pages 205–216. ACM Press, 2001.

[7] Duncan Coutts, Don Stewart, and Roman Leshchinskiy. Rewriting Haskell strings. In *Practical Aspects of Declarative Languages 8th International Symposium, PADL 2007*, LNCS. Springer-Verlag, 2007.

[8] Matthew Fluet, Mike Rainey, John Reppy, Adam Shaw, and Yingqi Xiao. Manticore: A heterogeneous parallel language. In *Proceedings of the Workshop on Declarative Aspects of Multicore Programming (DAMP 2007)*. ACM Press, 2007.

[9] A Gill, J Launchbury, and SL Peyton Jones. A short cut to deforestation. In *ACM Conference on Functional Programming and Computer Architecture (FPCA'93)*, pages 223–232, Cophenhagen, 1993. ACM Press. ISBN 0-89791-595-X.

[10] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable memory transactions. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'05)*, June 2005.

[11] Gabriele Keller and Manuel M. T. Chakravarty. On the distributed implementation of aggregate data structures by program transformation. In José Rolim et al., editors, *Parallel and Distributed Processing, Fourth International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'99)*, number 1586 in LNCS, pages 108–122, Berlin, Germany, 1999. Springer-Verlag.

[12] Ken Kennedy and Kathryn S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *1993 Workshop on Languages and Compilers for Parallel Computing*, number 768, pages 301–320. Springer Verlag, 1993.

[13] J Launchbury and SL Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–342, December 1995.

[14] Roman Leshchinskiy, Manuel M. T. Chakravarty, and Gabriele Keller. Higher order flattening. In *Third International Workshop on Practical Aspects of High-level Parallel Programming (PAPP 2006)*, number 3992 in LNCS. Springer-Verlag, 2006.

[15] E Christopher Lewis, Calvin Lin, and Lawrence Snyder. The implementation and evaluation of fusion and contraction in array languages. In *Proc ACM Conference on Programming Language Design and Implementation*, pages 50–59, 1998.

[16] Naraig Manjikian. Combining loop fusion with prefetching on shared-memory multiprocessors. In *Proceedings of the 1997 International Conference on Parallel Processing (ICPP '97)*. IEEE Computer Society Press, 1997.

[17] Daniel Palmer, Jan Prins, and Stephan Westfold. Work-efficient nested data-parallelism. In *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Processing (Frontiers 95)*. IEEE Press, 1995.

[18] Simon Peyton Jones, Tony Hoare, and Andrew Tolmach. Playing by the rules: rewriting as a practical optimisation technique. In *Proceedings of the ACM SIGPLAN 2001 Haskell Workshop*, 2001.

[19] SL Peyton Jones. Compilation by transformation: a report from the trenches. In *European Symposium on Programming*, volume 1058 of *LNCS*, pages 18–44. Springer Verlag, 1996.

[20] SL Peyton Jones, AJ Gordon, and SO Finne. Concurrent Haskell. In *23rd ACM Symposium on Principles of Programming Languages (POPL'96)*, pages 295–308, St Petersburg Beach, Florida, January 1996. ACM Press.

[21] SL Peyton Jones and J Launchbury. Unboxed values as first class citizens. In RJM Hughes, editor, *ACM Conference on Functional Programming and Computer Architecture (FPCA'91)*, volume 523 of *Lecture Notes in Computer Science*, pages 636–666, Boston, 1991. Springer.

[22] SL Peyton Jones and S Marlow. Secrets of the Glasgow Haskell Compiler inliner. *Journal of Functional Programming*, 12:393–434, 2002. First published at Workshop on Implementing Declarative Languages, Paris, Sept 1999.

[23] Gerald Roth and Ken Kennedy. Loop fusion in High Performance Fortran. In *Conference Proceedings of the 1998 International Conference on Supercomputing*, pages 125–132. ACM Press, 1998.

[24] SB Scholz. Single Assignment C – efficient support for high-level array operations in a functional setting. *Journal of Functional Programming*, 13:1005–1059, 2003.

[25] Byoungro So, Anwar Ghuloum, and Youfeng Wu. Optimizing data parallel operations on many-core platforms. In *First Workshop on Software Tools for Multi-Core Systems (STMCS)*, 2006. `http://www.isi.edu/~kintali/stmcs06/prog.html`.

[26] Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System F with type equality coercions. In *Proc ACM SIGPLAN Workshop on Types in Language Design and Implementation*, Nice, France, January 2007. ACM.

[27] Josef Svenningsson. Shortcut fusion for accumulating parameters and zip-like functions. In *Proc ACM International Conference on Functional Programming*, pages 124–132, Pittsburgh, 2002.

[28] PW Trinder, K Hammond, H-W Loidl, and SL Peyton Jones. Algorithm + strategy = parallelism. *Journal of Functional Programming*, 8:23–60, January 1998.

[29] Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.

[30] M. Wolf and M. Lam. An algorithmic approach to compound loop transformations. In T. Gross A. Nicolau, D. Gelernter and D. Padua, editors, *Advances in Languages and Compilers for Parallel Computing*, pages 243–259. The MIT Press, 1991.