Nepal – Nested Data Parallelism in Haskell

Manuel M. T. Chakravarty[1], Gabriele Keller[2], Roman Lechtchinsky[3], and Wolf Pfannenstiel[4]

[1] University of New South Wales
[2] University of Technology, Sydney
[3] Technische Universität Berlin
[4] IT Service Omikron GmbH, Berlin

Abstract. This paper discusses an extension of Haskell by support for nested data-parallel programming in the style of the special-purpose language NESL. The extension consists of a parallel array type, array comprehensions, and primitive parallel array operations. This extension brings a hitherto unsupported style of parallel programming to Haskell. Moreover, nested data parallelism should receive wider attention when available in a standardised language like Haskell.

1 Introduction

Most extensions of Haskell aimed at parallel programming focus on *control parallelism* [1,7], where arbitrary independent subexpressions may be evaluated in parallel. These extensions vary in their selection strategy of parallel subexpressions and associated execution mechanisms, but generally maximise flexibility as completely unrelated expressions can be evaluated in parallel. As a result, most of them require multi-threaded implementations and/or sufficiently course-grained parallelism, and they make it hard for both the programmer and the compiler to predict communication patterns.

There are, however, also a few *data parallel* extensions of Haskell [14,12,11]. They restrict parallelism to the simultaneous application of a single function to all elements of collective structures, such as lists or arrays. This restriction might be regarded as a burden on the programmer, but it allows both the programmer as well as the compiler to better predict the parallel behaviour of a program, which ultimately allows for a finer granularity of parallelism and more radical compiler optimisations. Furthermore, the single-threaded programming model is closer to sequential programming, and thus, arguably easier to understand.

The programming model of *nested data parallelism (NDP)* [5] is an attempt at maximising flexibility while preserving as much static knowledge as possible. It extends *flat* data parallelism as present in languages like High Performance Fortran (HPF) [13] and Sisal [8] such that it can easily express computations over highly irregular structures, such as sparse matrices and adaptive grids. NDP has been popularised in the language NESL [4], which severely restricts the range of available data structures—in fact, NESL supports only tuples in addition to parallel arrays. In particular, neither user-defined recursive nor sum

R. Sakellariou et al. (Eds.): Euro-Par 2001, LNCS 2150, pp. 524–534, 2001.

types are supported. This is largely due to a shortcoming in the most successful implementation technique for NDP—the *flattening* transformation [6,19], which maps nested to flat parallelism. Recently, we lifted these restrictions on flattening [16,10] and demonstrated that the combination of flattening with fusion techniques leads to good code for distributed-memory machines [15,17]. These results allow us to support NDP in Haskell and to apply flattening for its implementation. In the resulting system—which we call NEPAL (NEsted PArallel Language), for short—a wide range of important parallel algorithms (1) can be formulated elegantly and (2) can be compiled to efficient code on a range of parallel architectures.

Our extension is conservative in that it does not alter the semantics of existing Haskell constructs. We merely add a new data type, namely *parallel arrays*, parallel array comprehensions, and a set of parallel operations on these arrays. An interesting consequence of explicitly designating certain data structures as parallel and others as sequential is a type-based specification of data distributions. When compared to NESL, NDP in Haskell benefits from the standardised language, wider range of data types, more expressive type system, better support for higher-order functions, referential transparency, module system and separate compilation, and the clean I/O framework. In previous work [16,9], we have provided experimental data that supports the feasibility of our approach from a performance point of view.

This paper makes the following main contributions: First, we show how NESL's notion of nested data parallelism can be integrated into Haskell by adding parallel arrays. Second, we show how the combination of parallel and sequential types leads to a declarative specification of data distributions. Third, we demonstrate the feasibility of our approach by discussing well-known parallel algorithms.

2 Nested Data Parallelism Using Parallel Arrays

In this section, we briefly introduce the parallel programming model of nested data parallelism (NDP) together with our extension of Haskell by parallel arrays.

A *parallel array* is an ordered, homogeneous sequence of values that comes with a set of parallel collective operations. We require parallel arrays to be distributed across processing nodes if they occur in a program executed on a distributed memory machine. It is the responsibility of the execution mechanism to select a distribution that realises a good compromise between optimal load balance and minimal data re-distribution—see [15] for the corresponding implementation techniques. The type of a parallel array containing elements of type τ is denoted by $[:\tau:]$. This notation is similar to the list syntax and, in fact, parallel arrays enjoy the same level of syntactic support as lists where the brackets [: and :] denote array expressions. For instance, $[:a_1, \ldots, a_n:]$ constructs a parallel array with n elements. Furthermore, most list functions, such as *map* and *replicate*, have parallel counterparts distinguished by the suffix P, i.e., the standard prelude contains definitions for functions such as $mapP :: (\alpha \to \beta) \to [:\alpha:] \to [:\beta:]$

to map a function over a parallel array or $replicateP :: Int \to \alpha \to [:\alpha:]$ to create an array containing n copies of a value. The infix operators (!:) and (+:+) are used to denote indexing and concatenation of parallel arrays.

In contrast to sequential list operations, collective operations on parallel arrays execute in parallel. Thus, $mapP\ (+1)[:1, 2, 3, 4, 5, 6:]$ increments all numbers in the array in a single parallel step. The nesting level of parallel elementwise operations does not affect the degree of parallelism available in a computation so that if $xss = [:[:1, 2:], [:3, 4, 5:], [::], [:6:]:]$, then $mapP\ (mapP\ (+1))\ xss$ executes in one parallel step as well. The same holds for expressions such as $[:sumP\ xs \mid xs \leftarrow xss:]$. Each application of *sumP* uses parallel reduction *and* all of these applications are executed simultaneously. The standard function *sumP* is described in section 3. The behaviour of array comprehensions corresponds to that of list comprehensions. The key property of nested data parallelism is that all parallelism can be exploited independent of the depth of nesting of data-parallel constructs.

2.1 Using Nested Data Parallelism

Consider the following definition of parallel quicksort:
$qsort :: Ord\ \alpha \Rightarrow [:\alpha:] \to [:\alpha:]$.

$$
\begin{array}{lcl}
qsort\ [::] & = & [::] \\
qsort\ xs & = & \textbf{let} \\
 & & \quad m = xs\ !:\ (lengthP\ xs\ \text{`}div\text{`}\ 2) \\
 & & \quad ss = [:s \mid s \leftarrow xs,\ s < m:] \\
 & & \quad ms = [:s \mid s \leftarrow xs,\ s == m:] \\
 & & \quad gs = [:s \mid s \leftarrow xs,\ s > m:] \\
 & & \quad sorted = [:qsort\ xs' \mid xs' \leftarrow [:ss,\ gs:]:] \\
 & & \textbf{in} \\
 & & (sorted\ !:\ 0) +:+ ms +:+ (sorted\ !:\ 1)
\end{array}
$$

The above NEPAL program looks strikingly similar to the sequential list-based implementation of this algorithm. This is not surprising since our approach seamlessly supports the usual functional programming style and integrates well into Haskell. This is mainly due to (1) the use of collection-based operations which are ubiquitous in sequential Haskell programs as well and (2) the absence of state in parallel computations. Note, however, that the recursive calls to *qsort* are performed in an array comprehension ranging over a nested array structure and are thus executed in parallel. This would *not* be the case if we wrote $qsort\ ss +:+ ms +:+ qsort\ gs$! The parallelism in *qsort* is obviously highly irregular and depends on the initial ordering of the array elements. Moreover, the nesting depth of parallelism is statically unbounded and depends on the input given to *qsort* at runtime. Despite these properties, the flattening transformation can rewrite the above definition of *qsort* into a flat data parallel program, while preserving all parallelism contained in the definition. Thus, it would be possible to achieve the same parallel behaviour in Fortran, too—it is, however, astonishingly tedious.

Hierarchical n-body codes, such as the Barnes-Hut algorithm [2], exhibit a similar parallel structure as qsort and there is considerable interest in their high-performance implementation. We showed that rose trees of the required form lend themselves to a particularly elegant nested data parallel implementation of the Barnes-Hut algorithm, which can be expressed elegantly in Nepal and compiled into efficient code [15].

3 Parallel Arrays in Haskell

As we leave the semantics of standard Haskell programs entirely intact, we can make full use of existing source code, implementation techniques, and tools. We merely introduce a single new polymorphic type, denoted $[:\alpha:]$, which represents parallel arrays containing elements of type α.

Construction and Matching Construction of parallel arrays is defined analogous to the special bracket syntax supported in Haskell for lists. In particular, we have:

$[:]$	$::$	$[:\alpha:]$	– nullary array, i.e., an array with no elements
$[:e_1, \ldots, e_n:]$	$::$	$[:\tau:]$	– an array with n elements, where $e_i :: \tau$ for all i
$[:e_1..e_2:]$	$::$	$Enum\ \alpha \Rightarrow [:\alpha:]$	– enumerate the values between e_1 and e_2
$[:e_1, e_2..e_3:]$	$::$	$Enum\ \alpha \Rightarrow [:\alpha:]$	– enumerate from e_1 to e_3 with step $e_2 - e_1$

Moreover, we introduce $[:p_1, \ldots, p_n:]$ as a new form of patterns, which match arrays that (1) contain exactly n elements and (2) for which the ith element can be bound to the pattern p_i.

In contrast to lists, parallel arrays are not defined inductively, and thus, there is no constructor corresponding to (:). From the user's point of view, parallel arrays are an abstract data type that can only be manipulated by array comprehensions and the primitive functions defined in the following. An inductive view upon parallel arrays, while technically possible, would encourage inefficient sequential processing of arrays. Usually, lists are a better choice for this task.

Evaluation Strategy To guarantee the full exposure of nested parallelism and to enable the compiler to accurately predict the distribution of parallel structures and the entailed communication requirements, we impose some requirements on the evaluation of expressions resulting in a parallel array. In essence, these requirements guarantee that we can employ the flattening transformation for the implementation of all nested data parallelism contained in a Nepal program.

We require that the construction of a parallel array is strict in so far as all elements are evaluated to WHNF, i.e., $[:e_1, \ldots, e_{i-1}, \bot, e_{i+1}, \ldots, e_n:] = \bot$. Moreover, parallel arrays are always finite, i.e., an attempt to construct an infinite array like **let** $xs = [:1:] +:+ xs$ **in** xs diverges.

As a result, the execution mechanism can evaluate all elements of an array in parallel as soon as the array itself is demanded. Moreover, elements of primitive type (like *Int*) can always be stored unboxed in parallel arrays, so we can implement a value of type $[:Int:]$ as a flat collection of whatever binary representation

the target machine supports for fixed-precision integral values. This is certainly much more efficient than having to heap-allocate each individual *Int* element, and thus, beneficial for most numerical applications. These properties of parallel arrays are what prevents us from using the *Array* type provided by Haskell's standard library for expressing NDP.

Array Comprehensions Experience with NESL suggests that array comprehensions are a central language construct for NDP programs. Parallel array comprehensions are similar to list comprehensions, but again use [: and :] as brackets. However, we extend the comprehension syntax with the new separator | that simplifies the elementwise lockstep processing of multiple arrays. For instance, the expression $[:x + y \mid x \leftarrow [:1, 2, 3:] \mid y \leftarrow [:4, 5, 6:]:]$ evaluates to $[:5, 7, 9:]$, and thus, is equivalent to $[:x + y \mid (x,\ y) \leftarrow \mathit{zipP}\ [:1, 2, 3:]\ [:4, 5, 6:]:]$. Therefore, the introduction of | is strictly speaking redundant. However, in contrast to the typical list processing usage of list comprehensions, experience with NDP code suggests that lockstep processing of two and more parallel arrays occurs rather frequently—moreover, the application of these comprehensions tends to be nested. For the sake of orthogonality, we also allow | to be used in list comprehensions. The semantics of array comprehensions is defined as follows (in correspondence to [18]):

$$
\begin{array}{lcl}
[:e \mid :] & = & [:e:] \\
[:e \mid b, Q:] & = & \textbf{if}\ b\ \textbf{then}\ [:e \mid Q:]\ \textbf{else}\ [::] \\
[:e \mid p \leftarrow l, Q:] & = & \textbf{let} \\
 & & \quad \mathit{ok}\ p\ =\ [:e \mid Q:] \\
 & & \quad \mathit{ok}\ \_\ =\ [::] \\
 & & \textbf{in} \\
 & & \mathit{concatMapP}\ \mathit{ok}\ l \\
[:e \mid p_1 \leftarrow l_1 \mid & & \\
\quad p_2 \leftarrow l_2 \mid Q_1,\ Q_2:] & = & [:e \mid (p_1, p_2) \leftarrow \mathit{zipP}\ l_1\ l_2 \mid Q_1,\ Q_2:] \\
[:e \mid \textbf{let}\ \mathit{decls}, Q:] & = & \textbf{let}\ \mathit{decls}\ \textbf{in}\ [:e \mid Q:]
\end{array}
$$

As with list comprehensions, the above merely defines the declarative semantics of array comprehensions. An implementation is free to choose any optimising implementation that preserves this semantics.

Standard Operations on Parallel Arrays Besides supporting the entire Haskell prelude, NEPAL also provides a comprehensive set of functions for manipulating arrays. Most of these, such as *mapP*, *filterP*, *zipP*, and *concatMapP*, have sequential list-based counterparts with nearly identical denotational semantics. However, the definitions of some list functions, most notably of reductions and scans, preclude an efficient or even meaningful parallel implementation of their semantics. Consequently, no parallel versions of functions such as *foldr* are provided. Instead, the NEPAL prelude contains definitions of parallel reduction and scan functions, such as $\mathit{foldP} :: (\alpha \to \alpha \to \alpha) \to \alpha \to [:\alpha:] \to \alpha$ and $\mathit{scanP} :: (\alpha \to \alpha \to \alpha) \to \alpha \to [:\alpha:] \to [:\alpha:]$. The order in which individual array elements are processed is unspecified and the binary operation is required

to be associative, thus permitting a tree-like evaluation strategy with logarithmic depth (cf. [3]). Other parallel reductions are defined in terms of these basic operations, e.g., $sumP :: Num\,\alpha \Rightarrow [:\alpha:] \to \alpha$ with $sumP = foldP\,(+)\,0$. For these specialized reductions, the semantical differences between the parallel and the corresponding list-based versions, such as *sum*, are minimal and reflected in the definition of the more primitive operations (*foldP* in the above case).

3.1 Implementation of Nested Data Parallelism

We implement Nepal by extending an existing Haskell system: the Glasgow Haskell Compiler (GHC), which is known to produce fast sequential code. The present paper only provides a sketch of each of the compiler phases and of the techniques involved. More details can be found in [15,17,9,10].

The first phase, the front end, simply converts Haskell code including parallel arrays into an intermediate language, i.e., syntactic sugar is removed. The second phase, the flattening transformation, maps all nested computations to flat parallel computations, preserving the degree of parallelism specified in the source program. As already mentioned, due to the presence of recursive data types in a parallel context, the type transformation, as well as the instantiation of polymorphic functions on arrays, requires special consideration—we present the complete transformation in a form suitable for the Haskell Kernel in [10]. In the third step, all the data parallel primitives are decomposed into their purely processor local and the communication components. In this unfolded representation, we apply GHC's simplifier, which has been extended with rules for array and communication fusion to optimise local computations and communication operations for the target architecture. This localises memory access, reduces synchronisation, and allows one to trade load balance for data re-distribution. Finally, the code-generation phase produces C or native code that uses our collective-communication library to maintain distributed data structures and to specify communication. The library internally maps all collective communication to a small set of one-sided communication operations, which makes it highly portable [9].

4 Solving Tridiagonal Systems of Linear Equations

In addition to the obvious uses of sum types, the extension of flattening to the full range of Haskell types allows a declarative type-based control of data distribution. Consider the operational implications for an array of arrays $[:[:Int:]:]$ versus an array of (sequential) lists $[:[Int]:]$. On a distributed memory machine, values of the former will be evenly distributed over the processing elements; in particular, if the subarrays vary substantially in size, they may be split up across processor boundaries to facilitate parallel operations over all elements of the nested array simultaneously. In contrast, arrays of lists are optimised for sequential operations over the sublists; although, the sequential processing of all the sublists is expected to proceed in parallel. One application where the

distinction of parallel and sequential data-structures is useful is the parallel solution of tridiagonal systems of linear equations as proposed by Wang [21].

Tridiagonal systems of linear equations are a special form of sparse linear systems occuring in numerous scientific applications. Such system can be solved sequentially in linear time by first eliminating the elements of the lower diagonal by a top-down traversal, and then eliminating the upper diagonal by traversing the matrix from bottom to top. Unfortunately, in each step a pivot row is needed that is computed just in the step before, so the algorithm is completely sequential. In the parallel solution proposed by Wang, the matrix is subdivided into blocks of consecutive rows, which are then processed simultaneously. The algorithm runs in three phases. First, all rows of a block are traversed top-down and then bottom-up to eliminate the lower and upper diagonal, respectively. However, since the first row in each but the first block still contains the lower diagonal element, a vertical chain of fill-in elements appears in this column. As the matrix is symmetric, a chain of fill-ins also occurs on the right in all but the last block in the bottom-up traversal. The non-zero elements of the matrix after the first phase are shown in Fig. 1. To diagonalise the matrix, the left and right chains of fill-ins must be eliminated. The first block's last row contains non-zeros

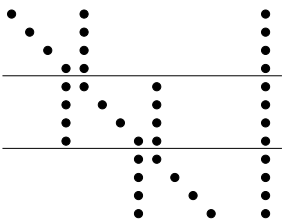

Fig. 1. Situation with 3 blocks after first parallel phase in Wang's algorithm

suitable for elimination of all left fill-ins in the second block. Once the left chain element of the second block's last row has been eliminated, this updated row can be used as a pivot for the elimination of the left fill-in chain in the third block etc. Thus, a pipelining phase is necessary over all blocks to propagate suitable pivot rows for the elimination of the left chains of fill-ins. Analogously, pivots can be propagated upwards starting with the last block to eliminate the right chains of extra non-zeros.

Elimination of the left chain can start only after the pivot row from the previous block is available, but this is the case only after the left fill-in of the previous block's last row has been eliminated already. Thus, it is important that during pipelining, only the first and last rows of each block are touched, because eliminating all fill-ins first before propagating pivots to the next block would mean a completely sequential traversal of the matrix.

After the pipelining phase, there are pivot rows for each block that can be used to eliminate both the left and the right chains of fill-ins. Like in the first phase, all blocks can be processed in parallel.

4.1 Encoding Wang's Algorithm in Nepal

In Nepal, we model an equation with a tuple-type *TRow* containing the three diagonal elements, the two potential chain elements, and the right-hand side.

```
type TRow = (Float, Float, Float, Float, Float, Float)
              — left, lower, main, upper, right, rhs
```

A row block is a list of rows, i.e., of type $[TRow]$. The whole matrix is a parallel array of row blocks of type $[:[TRow]:]$. The following encodes the top-level function of Wang's algorithm.

```
solve     :: [:[TRow]:] → [:[Float]:]
solve m =
  let
    res       = [:elimLowerUpper x | x ← m:]                    — phase 1
    frv       = [:f | (_, f, _) ← res:]
    lrv       = [:l | (_, _, l) ← res:]
    rowv      = [:r | (r, _, _) ← res:]
    (fpl, lpl) = pipeline (pArrayToList frv) (pArrayToList lrv)  — phase 2
    (fpv, lpv) = (listTopArray fpl, listTopArray lpl)
    dm        = [:elimLR r fp lp | r ← rowv | fp ← fpv | lp ← lpv:]  — phase 3
  in
  mapP (map (λ(TRow _ _ maine _ _ rhs) → rhs/maine))
```

The functions *elimLowerUpper* and *elimLR* are normal, recursive list-traversals, eliminating elements on each row both in the descending and ascending phase of recursion—we omit the details of their definition here, as they do not use parallelism. However, these traversals are executed in parallel for all blocks. The function *elimLowerUpper* is of type $[TRow] \to ([TRow], TRow, TRow)$. It returns the updated row block plus the two rows needed for the pipelining phase. As the pipelining is sequential, lists are used and so the arrays with the first and last pivot rows are converted by the primitive *pArrayToList*. The function *pipeline* is again an ordinary list traversal, realizing the desired pivot generation and propagation. The lists of new pivot rows are transformed into parallel arrays using *listTopArray*, so that the third phase can work in parallel on all blocks to eliminate the fill-in values.

Controlling the degree of parallelism While it is possible to implement this algorithm in Nesl, the trade-off between the computational depth of pipelining and the parallelism available in the other phases cannot be expressed cleanly in that language due to its lack of sequential types. Nepal's richer type system, on the other hand, allows us to make an explicit distinction between parallel and sequential computations. In the above example, we represent individual

blocks by sequential lists which, in turn, are stored in a parallel array. Thus, the structure of the algorithm is reflected in the structure of the data it operates upon. This makes the code more readable and allows the compiler to optimize more aggressively since more static information is available.

5 Related Work

We can categorise the extensions of Haskell as either data or control parallel as well as either preserving Haskell's semantics or altering it.

The approach that is probably the one closest related to Nepal is Jonathan Hill's data-parallel extension of Haskell [14]. The main difference between his and our approach is that he maintains the laziness of the collective type that is evaluated in parallel. The trade off here is, once more, one between flexibility and static information that can be used for optimisations. We chose to maximise the latter, he emphasised the former.

Two other approaches that do not alter the Haskell semantics and do, in fact, not extend the language at all are [12,11]. In both approaches, certain patterns in Haskell programs are recognised and treated specially—i.e., they are being given a parallel implementation. Both approaches choose to maximise static knowledge and are only applicable to regular parallelism, where the space-time mapping can be determined at compile time. This allows a maximum of optimisation by the compiler, but prevents the implementation of irregular parallelism.

Parallel Haskell (pH) [1] is an implicitly parallel approach that makes a fundamental change to Haskell's semantics: Instead of lazy evaluation, it requires *lenient* (non-strict, but eager) evaluation. Moreover, it introduces additional constructs that ultimately compromise referential transparency, but allow the programmer to maximise the available parallelism.

Glasgow Parallel Haskell (GPH) and the associated *evaluation strategies* [20] extend standard Haskell by a primitive `par` combinator that allows the programmer to designate pairs of expressions that may be evaluated in parallel. Based on this primitive, evaluation strategies allow to specify patterns of parallelism in the form of meaning-preserving annotations to sequential Haskell code.

6 Conclusion

We have presented Nepal, a conservative extension of the standard functional language Haskell, which allows the expression of nested data-parallel programs. Parallel arrays are introduced as the sole parallel datatype together with data-parallel array comprehensions and parallel array combinators. In contrast to some other approaches, the parallel operational semantics of Nepal does not compromise referential transparency.

Other than Nesl, Nepal supports the full range of both sequential and parallel data-types and computations, enlarging the class of algorithms suitable for a nested data-parallel programming style and allowing a declarative, type-based specification of data-distribution. In the context of NDP, Nepal is the

first flattening-based language that allows separate compilation in the presence of polymorphic functions on parallel arrays.

There are several hand-compiled examples such as the Barnes-Hut code or sparse-matrix vector multiplication delivering promising performance [16,9]. As we do not change Haskell as the sequential part of Nepal, existing implementation techniques and compiler code for Haskell can be re-used.

References

1. S. Aditya, Arvind, L. Augustsson, J.-W. Maessen, and R. S. Nikhil. Semantics of pH: A parallel dialect of Haskell. In P. Hudak, editor, *Proc. Haskell Workshop, La Jolla, CA USA*, YALEU/DCS/RR-1075, pages 35–49, June 1995. 524, 532
2. J. Barnes and P. Hut. A hierarchical $O(n \log n)$ force calculation algorithm. *Nature*, 324, December 1986. 527
3. G. E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, Nov. 1990. 529
4. G. E. Blelloch. NESL: A nested data-parallel language. Technical Report CMU-CS-93-129, School of Computer Science, Carnegie Mellon University, 1993. 524
5. G. E. Blelloch. Programming parallel algorithms. *CACM*, 39(3):85–97, 1996. 524
6. G. E. Blelloch and G. W. Sabot. Compiling collection-oriented languages onto massively parallel computers. *Journ. o. Par. and Distr. Comp.*, 8:119–134, 1990. 525
7. S. Breitinger, U. Klusik, and R. Loogen. From (sequential) Haskell to (parallel) Eden: An implementation point of view. *LNCS*, 1490:318–328, 1998. 524
8. D. Cann. Retire fortran? A debate rekindled. *CACM*, 35(8):81, Aug. 1992. 524
9. M. M. T. Chakravarty and G. Keller. How portable is nested data parallelism? In *6th Australasian Conf. on Par. a. Real-Time Sys.*, pages 284–299. Springer, 1999. 525, 529, 533
10. M. M. T. Chakravarty and G. Keller. More types for nested data parallel programming. In P. Wadler, editor, *ACM SIGPLAN Conference on Functional Programming (ICFP '00)*, pages 94–105. ACM Press, 2000. 525, 529
11. N. Ellmenreich, C. Lengauer, and M. Griebl. Application of the polytope model to functional programs. In J. Ferrante, editor, *Languages and Compilers for Parallel Computing*. Computer Science and Engineering Department, UC San Diego, 1999. 524, 532
12. C. Herrmann and C. Lengauer. Parallelization of divide-and-conquer by translation to nested loops. *Journ. o. Functional Programming*, 9(3):279–310, May 1999. 524, 532
13. High Performance Fortran Forum. High Performance Fortran language specification. Technical report, Rice University, 1993. Version 1.0. 524
14. J. M. D. Hill. *Data-parallel lazy functional programming*. PhD thesis, Department of Computer Science, Queen Mary and Westfield College, London, 1994. 524, 532
15. G. Keller. *Transformation-based Implementation of Nested Data Parallelism for Distributed Memory Machines*. PhD thesis, Technische Universität Berlin, 1999. 525, 527, 529
16. G. Keller and M. M. T. Chakravarty. Flattening trees. In D. Pritchard and J. Reeve, editors, *Euro-Par'98*, number 1470 in LNCS, pages 709–719. Springer, 1998. 525, 533

17. G. Keller and M. M. T. Chakravarty. On the distributed implementation of aggregate data structures by program transformation. In J. Rolim et al., editors, *Parallel and Distributed Processing, HIPS Workshop*, number 1586 in LNCS, pages 108–122. Springer, 1999. 525, 529
18. Haskell 98: A non-strict, purely functional language. `http://haskell.org/definition/`, February 1999. 528
19. J. Prins and D. Palmer. Transforming high-level data-parallel programs into vector operations. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 119–128. ACM, 1993. 525
20. P. W. Trinder, K. Hammond, H.-W. Loidl, and S. L. Peyton Jones. Algorithm + strategy = parallelism. *Journal of Functional Programming*, 1998. 532
21. H. H. Wang. A parallel method for tridiagonal equations. *ACM Transactions on Mathematical Software*, 7(2):170–183, June 1981. 530