Nepal – Nested Data-Parallelism in Haskell

Manuel M. T. Chakravarty
University of New South Wales
School of Computer Science & Engineering
Sydney, Australia
chak@cse.unsw.edu.au

Gabriele Keller
University of Technology, Sydney
Faculty of Information Technology
Sydney, Australia
keller@it.uts.edu.au

Roman Lechtchinsky
Technische Universität Berlin
Fachbereich Informatik
Berlin, Germany
rl@cs.tu-berlin.de

Wolf Pfannenstiel
Technische Universität Berlin
Fachbereich Informatik
Berlin, Germany
wolfp@cs.tu-berlin.de

ABSTRACT

This paper discusses an extension of Haskell by support for nested data-parallel programming in the style of the special-purpose language Nesl. More precisely, the extension consists of a parallel array type, array comprehensions, and a set of 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. This paper outlines the language extension and demonstrates its usefulness with two case studies.

Keywords: Data parallelism; flattening; irregular parallelism; Haskell

1. INTRODUCTION

Most extensions of Haskell that are aimed at parallel programming focus on *control parallelism* [1; 32; 31; 8; 10], 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 [20; 18; 15]. 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.

Ultimately, the choice between control and data parallelism is a trade off between flexibility and static knowledge about the parallelism contained within a program. The programming model of *nested data parallelism (NDP)* [6] 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) [19] and Sisal [9] 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 [5], which severely restricts the range of available data structures—in fact, NESL supports only tuples in addition to parallel arrays (called *vectors* in NESL). In particular, neither user-defined recursive nor sum types are supported. This is largely due to a shortcoming in the most successful implementation technique for NDP—the *flattening* transformation [7; 29], which maps nested to flat parallelism. Recently, we lifted these restrictions on flattening [23; 12] and demonstrated that the combination of flattening with fusion techniques leads to good code for distributed-memory machines [22; 24].

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. This paper will illustrate the first point by describing the implementation of the Barnes-Hut hierarchical n-body algorithm [2] and Wang's algorithm for solving tridiagonal equations [33]. It will provide only a rough sketch of the flattening-based implementation method, but details can be found elsewhere [12; 11]. A similar combination of the Nesl parallel programming model with Standard ML [25] is investigated in the PSciCo project [3].

Our extension of Haskell 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. Parallel arrays combine properties of the standard Haskell list and array data types; furthermore, their particular semantic properties make them ideally suited for parallel processing. A particularly interesting consequence of explicitly designating certain data structures as parallel and others as sequential is a type-based specification of data distributions. We will demonstrate this during the presentation of the example algorithms in Sections 4 and 5.

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 this paper, we will not present any new benchmark figures. In previous work [23; 11], we have provided experimental data that supports the feasibility of our approach from a performance point of view; we will summarise some of the results when discussing the n-body code (Section 4). In short, this paper makes the following main contributions:

- We show how NESL's notion of nested data parallelism can be integrated into Haskell by adding parallel arrays.
- We show how the combination of parallel and sequential types leads to a declarative specification of data distributions.
- We demonstrate the feasibility of our approach by discussing two well-known parallel algorithms.

The remainder of this paper is structured as follows: Section 2 provides more detail on nested data prallelism and gives a brief overview over our extension of Haskell. Section 3 details our integration of parallel arrays into Haskell and briefly outlines the flattening-based implementation. Section 4 discusses the implementation of the Barnes-Hut hierarchical n-body code and Section 5 studies Wang's parallel algorithm for solving tridiagonal systems of linear equations. Section 6 discusses related work. Finally, Section 7 concludes.

2. NESTED DATA PARALLELISM

In this section, we briefly introduce the parallel programming model of nested data parallelism (NDP) together with our extension of Haskell by parallel arrays—for more details on NDP, see [6].

2.1 A New Data Structure: 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 which realises a good compromise between optimal load balance and minimal data re-distribution—see [22] 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 the following:

$$
\begin{array}{l}
mapP \;\; :: \; (\alpha \to \beta) \to [:\alpha:] \to [:\beta:] \\
\quad \text{— map a function over a parallel array} \\
replicateP :: \; Int \to \alpha \to [:\alpha:] \\
\quad \text{— create an array containing } n \text{ copies of a value}
\end{array}
$$

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 $xss = [:[:1, 2:], [:3, 4, 5:], [::], [:6:]:]$,

$$mapP\ (mapP\ (+1))\ xss$$

executes in one parallel step as well. The same holds for expressions such as

$$[:sumP\ xs \mid xs \leftarrow xss:]$$

(the behaviour of array comprehensions corresponds to that of list comprehensions)—each application of $sumP$ uses parallel reduction *and* all of these applications are executed simultaneously. The standard function $sumP$ is described in section *3.1.4*.

In other words, the key property of nested data parallelism is that all parallelism can be exploited independent of the depth of nesting of data-parallel constructs. In fact, as we will see in the next subsection, this holds even for recursively nested divide&conquer algorithms, where the nesting is not even statically bound. As a result, the implementation of parallel algorithms is often straightforward, as illustrated by the following examples.

2.2 Using Nested Data Parallelism

Blelloch [6] introduced an elegant formulation of the multiplication of a sparse matrix with a dense vector, resulting in another dense vector. It is based on a well-known representation of general sparse matrices, the so-called *compressed row format*. Here, only non-zero elements of a matrix row are stored in an array of column-index/value pairs; a sparse matrix is represented by an array of such rows:

$$
\begin{array}{l}
\textbf{type}\ SparseRow \;\; = [:(Int,\ Float):] \quad \text{— index, value} \\
\textbf{type}\ SparseMatrix = [:SparseRow:]
\end{array}
$$

The multiplication of a sparse matrix with a vector can then be expressed by nesting three levels of parallel operations:

$$
\begin{array}{l}
smvm \;\; :: \; SparseMatrix \to [:Float:] \to [:Float:] \\
smvm\ sm\ vec = \\
\quad [:sumP\ \underbrace{[:x * (vec\ !:\ col) \mid (col,\ x) \leftarrow row:]}_{\text{products of one row}} \mid row \leftarrow sm:]
\end{array}
$$

The inner array comprehension computes all products of a single row of the matrix by indexing the input vector with the column index of the corresponding matrix element;

sumP adds the products in a parallel reduction; and the outer comprehension specifies that the products and sums for all rows should be computed in parallel. Since the algorithm makes full use of the parallelism inherent in the problem, its parallel depth complexity is proportional to the logarithm of the length of the longest row (cf. [6] for details). Moreover, a flattening-based implementation exploits all three levels of parallelism (the inner comprehension, *sumP*, and the outer comprehension) contained in the definition of *smvm*. While it is possible to achieve the same behaviour in a flat language, the code is significantly more involved.

The above 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.

Still, care has to be taken, so that computations that could be executed in parallel are not inadvertently sequentialised. The following definition of parallel quicksort is again very similar to the list-based version:

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

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, in principle, it would be possible to achieve the same parallel behaviour in Fortran—it is, however, astonishingly tedious.

3. PARALLEL ARRAYS IN HASKELL

We like to emphasise once more that we only add parallel arrays and associated operations to Haskell, but we leave the semantics of standard Haskell programs entirely intact. Consequently, we can make full use of existing source code, implementation techniques, and tools. In the following, we shall first discuss the details of our extensions, i.e, of parallel arrays, array comprehensions, and parallel array operations. Afterwards, we will briefly outline our implementation method.

3.1 The Details of the Extension

We merely introduce a single new polymorphic type, denoted $[:\alpha:]$, which represents parallel arrays containing elements of type α.

3.1.1 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 without any elements

$[:e_1, \ldots, e_n:] :: [:\tau:]$
— an array with n elements, where $e_i :: \tau$ for all i

$[:e_1..e_2:]$ $:: Enum\ \alpha \Rightarrow [:\alpha:]$
— an array enumerating the values between e_1 and e_2

$[:e_1, e_2..e_3:]$ $:: Enum\ \alpha \Rightarrow [:\alpha:]$
— enumerating 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. Note, how we distinguish between sequential types (e.g., lists) and parallel types (in our case, parallel arrays) here. We will reinforce the parallel flavour of values from $[:\alpha:]$ by requiring a particular evaluation strategy.

3.1.2 Evaluation Strategy

To guarantee the full exposure of nested parallelism and in order for 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 is strict in so far as all elements are evaluated to weak head-normal form, 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

$$\textbf{let}\ xs\ =\ [:1:] +:+ xs\ \textbf{in}\ xs$$

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; in other words, 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.

3.1.3 Array Comprehensions

Experience with Nesl suggests that array comprehensions (called *apply-to-each* constructs in Nesl) 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

$$[:x + y \mid (x, y) \leftarrow zipP\ [:1, 2, 3:]\ [:4, 5, 6:]:]$$

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 [28, Section 3.11]):

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

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.

3.1.4 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

$$\begin{array}{l}
foldP :: (\alpha \to \alpha \to \alpha) \to \alpha \to [:\alpha:] \to \alpha \\
scanP :: (\alpha \to \alpha \to \alpha) \to \alpha \to [:\alpha:] \to [:\alpha:]
\end{array}$$

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. [4]). Other parallel reductions are defined in terms of these basic operations, e.g.,

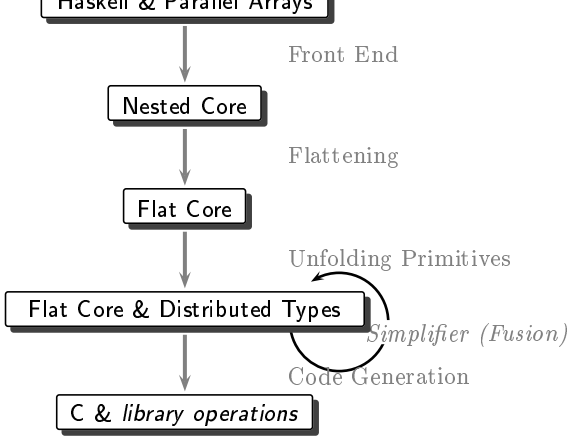

Figure 1: GHC with NDP extensions

$$\begin{array}{l}
sumP :: Num\ \alpha \Rightarrow [:\alpha:] \to \alpha \\
sumP = foldP\ (+)\ 0
\end{array}$$

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).

3.1.5 Open Problems

Currently, there are two open, but from a practical point of view not very serious, problems in the outlined design:

1. The pattern-matching suggested for arrays might be considered ad hoc, as it essentially allows to match only arrays of fixed sizes.

2. Expressions like $[:f\ a \mid f \leftarrow [:foo, bar:]:]$ essentially denote control parallelism, as the two unrelated functions *foo* and *bar* would—by what we have said so far—have to be evaluated in parallel.

The first problem is a consequence of not having an inductive definition for arrays. Thus, it could be argued that we should omit pattern-matching on arrays entirely. While this would certainly be feasible, it is often convenient to be able to test for parallel arrays containing zero, one, or two elements in a pattern.

The second problem is more serious. An obvious solution would be to forbid having functions as elements of parallel arrays. This is not so much of a restriction, as it might seem at first, as parallel arrays are for the expression of data parallelism only and there are not many meaningful data-parallel operations that can be defined on functions—all other uses of functions would, of course, not be restricted in any way. The main problem is that the obvious attempt of requiring all elements of parallel arrays to be part of a type class *NonFun* would lead to a proliferation of (rather trivial) contexts on all type declarations involving parallel computations. An alternative solution is to allow functions in parallel arrays, but to specify that expressions as the one stated above will lead to a sequential evaluation of the function applications. This, however, introduces a fair amount of complications into the formalisation of the flattening transformation, as discussed in [12].

3.2 Implementation of Nested Data-Parallelism

Let us now have a look at the implementation of NEPAL, which we realise by extending an existing Haskell system: the Glasgow Haskell Compiler (GHC), which is known to produce fast sequential code. The compilation process roughly consists of four major phases, which are depicted in Figure 1. The present paper only provides a sketch of each of the phases and of the techniques involved. More details can be found in [22; 24; 11; 12].

The first phase, the front end, simply converts Haskell code including parallel arrays into an intermediate language called *Nested Core*, i.e., the input is type checked and all syntactic sugar 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. Furthermore, all nested parallel data structures are transformed into isomorphic flat data structures. This is done by partially separating information about the structure from the data. Arrays with recursive element types are mapped onto recursive structures containing arrays with only simple element type. As, at some level, recursive structures have to be modelled using pointers, this step corresponds to converting an array of pointers into a pointer to an array. As a consequence of the type transformation, polymorphic operations on parallel arrays have to be replaced by corresponding operations on the new data structure.

The flattening step itself is similar to the technique described, for example, in [7; 29]. However, 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 [12].

In the third step (*Unfolding Primitives*) all the data parallel primitives are decomposed into their purely processor local and their global components—the latter are those requiring communication. The intermediate language "Flat Core & Distributed Types", which is the target language of this step, distinguishes between local and global values by the type system. In this 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 step transforms fine-grained vector loops into deep computations: This localises memory access, reduces synchronisation, and allows one to trade load balance for data redistribution.

Finally, the code-generation phase produces C or native code 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 [11].

The combination of flattening with array fusion and the communication library that contains only a small core of machine-dependent functions allows us to target a wide range of high-performance architectures. Furthermore, the components that are marked by use of an *italic* font in Figure 1 behave differently in dependence on the targeted architecture—we call them *target-dependent components*. However, the flattening transformation, while being essential for our approach to portability, operates in the same way for all kinds of target architectures; it does not specialise the code for an architecture, but generally brings it into a form that makes it easier for subsequent phases to generate good code. In contrast, the application of calculational fusion, the code generation, and our library have to be parametrised with information about the target architecture to generate good code.

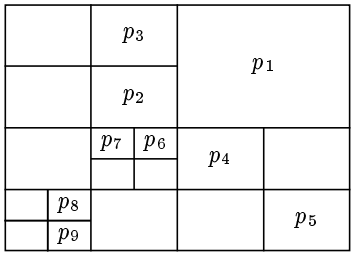

Figure 2: Hierarchical division of an area into subareas

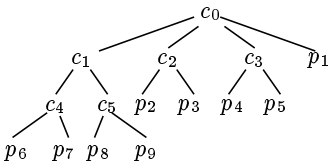

Figure 3: Example of a Barnes-Hut tree.

4. A SOLUTION TO THE N-BODY PROBLEM

This section presents a Nepal implementation of a simple version of the Barnes-Hut n-body algorithm[2], which is a representative of an important class of parallel algorithms covering applications like simulation and radiocity computations. These algorithms consist of two main steps: first, the data is clustered in a hierarchical tree structure; then, the data is traversed according to the hierarchical structure computed in the first step. In general, we have the situation that the computations that have to be applied to data on the same level of the tree can be executed in parallel.

The remainder of this section briefly describes the Barnes-Hut algorithm, the data structures that are required, and the NEPAL code. It addresses some implementation issues and discusses benchmarking results.

An n-body algorithm determines the interaction between a set of particles by computing the forces which act between each pair of particles. A precise solution therefore requires the computations of n^2 forces, which is not feasible for large numbers of particles. The Barnes-Hut algorithm minimizes the number of force calculations by grouping particles hierarchically into *cells* according to their spatial position. The hierarchy is represented by a tree. This allows approximating the accelerations induced by a group of particles on distant particles by using the centroid of that group's cell. The algorithm has two phases: (1) The tree is constructed from a particle set, and (2) the acceleration for each particle is computed in a down-sweep over the tree. Each particle is represented by a value of type *MassPoint*, a pair of position in the two dimensional space and mass:

```
type Vec      = (Double, Double)
type Area     = (Vec, Vec)
type Mass     = Double
type MassPoint = (Vec, Mass)
```

We represent the tree as a node which contains the centroid and a parallel array of subtrees:

```
data Tree = Node MassPoint [:Tree:]
```

Each iteration of *bhTree* takes the current particle set and the area in which the particles are located as parameters. It first splits the area into four subareas *subAs* of equal size. It then subdivides the particles into four subsets according to the subarea they are located in. Then, *bhTree* is called recursively for each subset and subarea. The resulting four trees are the subtrees of the tree representing the particles of the area, and the centroid of their roots is the centroid of the complete area. Once an area contains only one particle, the recursion terminates. Figure 2 shows such a decomposition of an area for a given set of particles, and Figure 3 displays the resulting tree structure.

```
bhTree :: [:MassPnt:] → Area → Tree
bhTree [:p:] area = Node p [::]
bhTree ps area =
  let
    subAs  = splitArea area
    pgs    = splitParticles ps subAs
    subts  = [:bhTree pg a | pg ← pgs, a ← subAs:]
    cd     = centroid [:mp | Node mp _ ← subts:]
  in
  Node cd subts
```

The tree computed by *bhTree* is then used to compute the forces that act on each particle by a function *accels*. It first splits the set of particles into two subsets: *fMps*, which contains the particles far away (according to a given criteria), and *cMps*, which contains those close to the centroid stored in the root of the tree. For all particles in *fMps*, the acceleration is approximated by computing the interaction between the particle and the centroid. Then, *accels* is called recursively for with *cMps* and each of the subtrees. The computation terminates once there are no particles left in the set.

```
accels :: Tree → [:MassPoint:] → [:Vec:]
accels _              [::]  = [::]
accels (Node cd subts) mps =
  let
    (fMps, cMps) = splitMps mps
    fAcs         = [:accel cd mp | mp ← fMps:]
        — forces for particles far from current p.
    cAcs         = [:accels t cMps | t ← subts:]
  in
  combine farAcs closeAcs

accel :: MassPoint → MassPoint → Vec
    — given two particles, the function accel
    — computes the acceleration that one particle
    — exerts on the other
```

The tree is both built and traversed level by level, i.e., all nodes in one level of the tree are processed in a single parallel step, one level after the other. This information is important for the compiler to achieve good data locality and load balance, because it implies that each processor should have approximately the same number of masspoints of each level. We can see the tree as having a sequential dimension to it, its depth, and a parallel dimension, the breadth, neither of which can be predicted statically. The programmer conveys this information to the compiler by the choice the data structure: By putting all subtrees into a parallel array in the type definition, the compiler assumes that all subtrees are going to be processed in parallel. The depth of the tree is modelled by the recursion in the type, which is inherently sequential. The type transformation in the compilation phase, then, transforms the tree into a list of arrays connected by global pointers, where each of the arrays is distributed over the processors involved in the computation. The local portions of the arrays (on for each level of the tree) are interconnected on each processor in the form of a linked list. In [23], we discussed why the above encoding based on recursive types is not possible in Nesl and what its advantages are compared to a possible Nesl implementation

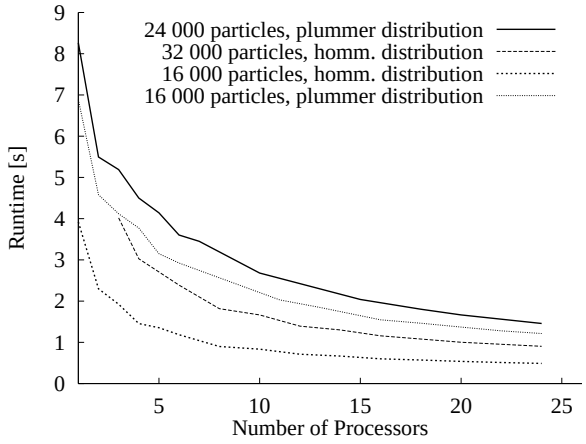

Figure 4: Runtime of the Barnes-Hut NBody algorithm on the Cray T3E

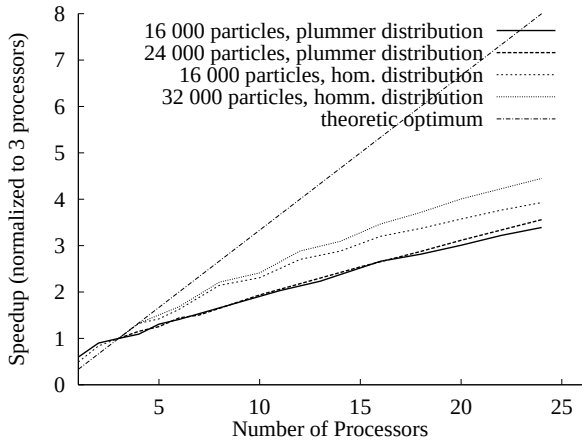

Figure 5: Speedup of the Barnes-Hut NBody algorithm on the Cray T3E

of the algorithm.

To get a feeling for the behaviour of our implementation technique, we tested hand-compiled code produced according to the compilation rules presented in [22; 23]. We ran benchmarks for two different types of particle sets: a homogeneous distributed set, where the particles are spread evenly over the area, and a so-called plummer distribution, where the particles center around one point of the area. The Barnes-Hut algorithm requires less computation steps for a homogeneous distribution, as the tree that stores the particles has depth of about $\log n$ for n particles. Roughly speaking, the algorithm has to compute twice as much particle-particle interactions for a set with plummer distribution than for a homogeneously distributed particle set with the same number of elements. The runtimes for particle sets of 16 000, 24 000, and 32 000 elements, which are displayed in Figure 4, show the higher absolute runtime of the plummer distribution. The diagram in Figure 5 reveals another effect: Not only is the absolute runtime of the regular case better, but we also obtain better speed up. On first sight, this might be surprising, as a higher number of computations often leads to programs with better relative speed up. In this case, though, we not only have more computations, but we also have more communication due to the high degree of irregularity. However, the diagram also shows that for 24 processors the speedup for the plummer set is still linear, while it already slows down slightly for the homogeneous sets.

5. 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 available 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 [33].

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. 6. To diagonalise the matrix, the left and right chains

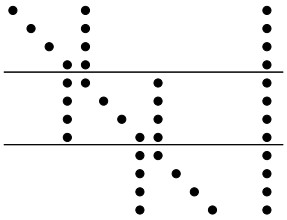

Figure 6: Situation with 3 blocks after first parallel phase in Wang's algorithm

of fill-ins must be eliminated.

The first block's last row contains non-zeros 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.

In each block, once the pivot row from the preceding block is available, the fill-in elements may be eliminated in any order. There are no sequential inner-block dependencies. However, as described above, there is a sequential dependency among the blocks. 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. Again, one top-down and one bottom-up traversal are necessary to obtain the desired diagonal structure.

5.1 Encoding Wang's Algorithm in NEPAL

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

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

A row block is a list of rows, i.e., of type $[TriRow]$. The whole matrix is a parallel array of row blocks, abbreviated by the type *Matrix*.

```
type Matrix = [: [TriRow] :]
                — a parallel array of lists of rows
```

The following encodes the top-level function of Wang's algorithm.

```
solve :: Matrix → [:[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       = [:elimLeftRight r fp lp | r ← rowv &
                fp ← fpv & lp ← lpv:]            — Phase 3
  in
  mapP (map (λ(TriRow _ _ maine _ _ rhs) → rhs/maine))
```

The functions *elimLowerUpper* and *elimLeftRight* are ordinary, 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 $[TriRow] \to ([TriRow], TriRow, TriRow)$. 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.

5.2 Controlling the degree of parallelism

The parallelism available in the algorithm depends on the number of blocks as these are processed in parallel in Phases 1 and 3. In the pipelining phase, however, pivot rows must be propagated sequentially across all blocks, making the depth of this phase proportional to the number of blocks. Consequently, parallelising the first and the third phases completely by setting the block size to 1 leads to a pipelining phase that needs linear time, which implies no speedup against the sequential version. Obviously, the best solution is to create one block per processor, thus minimising the costs of pipelining while still fully utilising the target machine.

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.

6. RELATED WORK

The relative merits of NDP when compared to other parallel programming models have already been covered elsewhere—e.g., [6]. Hence, in the following, we will concentrate on parallel functional languages and, in particular, on parallel extensions of Haskell—instead of discussing parallel programming languages in general. Generally, we can categorise the extensions of Haskell as either data or control parallel as well as either preserving the semantics of existing Haskell programs or altering it. Interestingly, it seems as if all data-parallel extensions maintain Haskell's original semantics, whereas control-parallel extensions tend to modify it—if only in a subtle way.

6.1 Data Parallel Extensions

NEPAL does not affect the semantics of standard Haskell programs, i.e., only the newly introduced types and operations have a parallel semantics. This guarantees maximal compatibility to existing Haskell code. An approach that follows the same goal and is probably the one closet related to NEPAL is Jonathan Hill's data-parallel extension of Haskell [20]. 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 of the programming model and static information that can be used for optimisations. We chose to maximise static information, he emphasised flexibility.

Two other approaches that do not alter the Haskell semantics and do, in fact, not extend the language at all are [18; 15]. In both approaches, certain patterns in Haskell programs are recognised and treated specially—i.e., they are being given a parallel implementation. In the first approach, these patterns have to be specified explicitly by means of coding parallel algorithms using specialised divide&conquer skeletons. 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. In fact, it is not entirely clear, whether these two approaches should be categorised as data or control parallel. They do not explicitly restrict the range of parallelised expressions, but due to their focus on array-based algorithms, they certainly operate in the realm of data parallelism.

6.2 Control Parallel Extensions

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. The most interesting feature of pH is probably that, despite being a control-parallel language, it allows very fine-grained parallelism—to a degree that is usually reserved for data parallel languages.

Glasgow Parallel Haskell (GPH) and the associated *evaluation strategies* [32; 31] 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 normal (sequential) Haskell code. There is, however, a slight modification of Haskell's original semantics hidden in these strategies. They can increase the strictness of functions, and thus, lead to non-termination

of programs that do terminate under the purely sequential execution model.

Two more radical control-parallel extensions of Haskell are Eden [8] and Goffin [10]. Both follow the idea of the seperation of computation and co-ordination, where the latter describes the parallel behaviour of a given program. Eden specifies co-ordination as a set of *stream processors* and introduces a notion of *process abstractions*, whereas Goffin uses a small set of *constraint-logic combinators* and *constraint abstractions* for the same purpose. Eden ultimately breaks referential transparency, and thus, Haskell's original semantics, whereas Goffin does not alter the standard Haskell portion of the language at all.

6.3 Other Parallel Functional Languages

Generally, there exists a wide range of parallel languages that are based on the model of functional programming—as, for example, witnessed in [17]. Ranging from languages that just support purely regular computations, such as Sisal [16] and SAC [30], over languages based on the idea of *skeletons* [13], such as [14], to control-parallel languages, such as Concurrent Clean [26].

The one parallel language that is closest to NEPAL in terms of the parallel programming model is certainly Nesl [5], which has been the starting point of our research. In essence, it has been our aim to take the novel functionality of Nesl and develop it to a point where it could be integrated in a standard functional language like Haskell. As a result, we could improve on the range of data types and the support for higher-order functions, and moreover, NEPAL has inherited from Haskell a module system with support for separate compilation and a clean I/O framework. This has only been possible due to the progress that we recently made in extending the scope of the flattening transformation [12].

7. 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. NEPAL is intended as a step towards bridging the gap between high-level parallel programming models and high performance, and it is our feeling that nested data-parallelism in Haskell together with the flattening transformation and appropriate optimisations bear a potential to achieve this goal.

Among smaller examples, we have presented two parallel applications that demonstrate the expressiveness of nested data-parallel programming based on Haskell. 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.

We are currently implementing a full compiler, which uses a transformation-based approach. We will integrate several optimisation techniques in the compiler that have been developed and investigated for nested data-parallelism [22; 27]. There are several hand-compiled examples such as the Barnes-Hut code or sparse-matrix vector multiplication delivering promising performance [23; 11]. As we do not change Haskell as the sequential part of NEPAL, existing implementation techniques and compiler code for Haskell can be reused.

7.1 Future Work

As an important piece of future work, we will develop a language-based cost model based on the common measures work and depth. The core rules of NESL's cost model will be re-used for NEPAL as far as possible. However, the adaptation to Haskell's powerful type system requires significant extensions to the cost model. Using Hinze's approach to generic functional programming as a starting point, we will develop a cost measure for polymorphic primitives [21].

In addition to the standard prelude, we will define a set of library functions for parallel arrays. Where useful, we will adapt the functions from the list and array libraries. New functions will probably be introduced for the interplay between parallel arrays and sequential collection types.

8. 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.

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

[3] G. Blelloch et al. The PSciCo project. `http://www.cs.cmu.edu/~pscico/`.

[4] G. E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, Nov. 1990.

[5] G. E. Blelloch. NESL: A nested data-parallel language (version 2.6). Technical Report CMU-CS-93-129, School of Computer Science, Carnegie Mellon University, April 1993.

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

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

[8] S. Breitinger, U. Klusik, and R. Loogen. From (sequential) Haskell to (parallel) Eden: An implementation point of view. *Lecture Notes in Computer Science*, 1490:318–??, 1998.

[9] D. Cann. Retire fortran? A debate rekindled. *Communications of the ACM*, 35(8):81, Aug. 1992.

[10] M. M. T. Chakravarty, Y. Guo, M. Köhler, and H. C. R. Lock. Goffin: Higher-order functions meet concurrent constraints. *Science of Computer Programming*, 30(1–2):157–199, 1998.

[11] M. M. T. Chakravarty and G. Keller. How portable is nested data parallelism? In *Proc. of 6th Annual Australasian Conf. on Parallel And Real-Time Systems*, pages 284–299. Springer-Verlag, 1999.

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

[13] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. The MIT Press, 1989.

[14] J. Darlington, A. J. Field, P. G. Harrison, P. H. J. Kelly, D. W. N. Sharp, Q. Wu, and R. L. While. Parallel programming using skeleton functions. In A. Bode, M. Reeve, and G. Wolf, editors, *PARLE '93: Parallel Architectures and Languages Europe*, number 694 in Lecture Notes in Computer Science, pages 146–160, Berlin, Germany, 1993. Springer-Verlag.

[15] N. Ellmenreich, C. Lengauer, and M. Griebl. Application of the polytope model to functional programs. In J. Ferrante, editor, *Proc. 12th Int. Workshop on Languages and Compilers for Parallel Computing (LCPC'99)*. Computer Science and Engineering Department, UC San Diego, 1999.

[16] J. T. Feo, D. C. Cann, and R. R. Oldehoeft. A report on the Sisal language project. *Journal of Parallel and Distributed Computing*, December 1990.

[17] K. Hammond and G. Michaelson, editors. *Research Directions in Parallel Functional Programming*. Springer-Verlag, 1999.

[18] C. A. Herrmann and C. Lengauer. Parallelization of divide-and-conquer by translation to nested loops. *Journal of Functional Programming*, 9(3):279–310, May 1999.

[19] High Performance Fortran Forum. High Performance Fortran language specification. Technical report, Rice University, 1993. Version 1.0.

[20] J. M. D. Hill. *Data-parallel lazy functional programming*. PhD thesis, Department of Computer Science, Queen Mary and Westfield College, London, 1994.

[21] R. Hinze. A new approach to generic functional programming. In *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Language*. ACM Press, 2000.

[22] G. Keller. *Transformation-based Implementation of Nested Data Parallelism for Distributed Memory Machines*. PhD thesis, Technische Universität Berlin, Fachbereich Informatik, 1999.

[23] G. Keller and M. M. T. Chakravarty. Flattening trees. In D. Pritchard and J. Reeve, editors, *Euro-Par'98, Parallel Processing*, number 1470 in Lecture Notes in Computer Science, pages 709–719, Berlin, 1998. Springer-Verlag.

[24] 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, Fourth International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'99)*, number 1586 in Lecture Notes in Computer Science, pages 108–122, Berlin, Germany, 1999. Springer-Verlag.

[25] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, 1990.

[26] E. G. J. M. H. Nöcker, J. E. W. Smetsers, M. C. J. D. van Eekelen, and M. J. Plasmeijer. Concurrent Clean. In *Proceedings of PARLE '91*, number 505/506 in Lecture Notes in Computer Science, pages 202–220. Springer-Verlag, 1991.

[27] W. Pfannenstiel. Combining fusion optimizations and piecewise execution of nested data-parallel programs. In J. R. et al., editor, *IPDPS 2000 Workshops (HIPS)*, Lecture Notes in Computer Science 1800, pages 324–331. Springer-Verlag, 2000.

[28] Haskell 98: A non-strict, purely functional language. `http://haskell.org/definition/`, February 1999.

[29] J. Prins and D. Palmer. Transforming high-level data-parallel programs into vector operations. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 119–128, San Diego, CA., May 19-22, 1993. ACM.

[30] S.-B. Scholz. On defining application-specific high-level array operations by means of shape-invariant programming facilities. In *Proceedings of APL'98*, pages 40–45. ACM Press, 1998.

[31] P. W. Trinder, K. Hammond, H.-W. Loidl, and S. L. Peyton Jones. Algorithm + strategy = parallelism. *Journal of Functional Programming*, 1998.

[32] P. W. Trinder, K. Hammond, J. S. Mattson Jr, A. S. Partridge, and S. L. Peyton Jones. GUM: a portable parallel implementation of Haskell. In *Proceedings of Programming Languages Design and Implementation*, 1996.

[33] H. H. Wang. A parallel method for tridiagonal equations. *ACM Transactions on Mathematical Software*, 7(2):170–183, June 1981.