Lazy Data-Oriented Evaluation Strategies

Prabhat Totoo
School of Mathematical and Computer Sciences,
Heriot-Watt University,
Edinburgh EH14 4AS, UK
pt114@hw.ac.uk

Hans-Wolfgang Loidl
School of Mathematical and Computer Sciences,
Heriot-Watt University,
Edinburgh EH14 4AS, UK
H.W.Loidl@hw.ac.uk

Abstract

This paper presents a number of flexible parallelism control mechanisms in the form of evaluation strategies for tree-like data structures implemented in Glasgow parallel Haskell. We achieve additional flexibility by using laziness and circular programs in the coordination code. Heuristics-based parameter selection is employed to auto-tune these strategies for improved performance on a shared-memory machine without programmer-specified parameters. In particular for unbalanced trees we demonstrate improved performance on a state-of-the-art multi-core server: giving a speedup of up to 37.5 on 48 cores for a constructed test program, and up to 15 for two other non-trivial applications using these strategies, a Barnes-Hut implementation of the n-body problem and a sparse matrix multiplication implementation.

Categories and Subject Descriptors D.3.2 [*Programming languages*]: Concurrent, distributed, and parallel languages; D.3.2 [*Programming languages*]: Applicative (functional) languages; E.1 [*Data structures*]: Trees; C.1.4 [*Processor architectures*]: Distributed architectures

Keywords Parallel Haskell, Dynamic parallelism control, Quad-trees.

1. Introduction

Evaluation strategies [17, 28] make it easy to specify parallel operations on flat data structures, for example, lists in Haskell. Indeed the existing library has a number of strategies for lists including `parList` for element-wise parallelism and `parListChunk` for grouping computations and thus improving the granularity of the parallelism. Parallel sub-components are usually homogeneous and work well with a basic implementation. The existing library also specifies generic strategies for *traversable* types. However, these achieve significantly worse performance on irregular input data, which is notoriously difficult to parallelise. In the Data Parallel Haskell (DPH) extension [23], for example, flattening transformation techniques are used for nested arrays and other irregular structures to enable even partitioning and hence even distribution of work across processors. A similar transformation is used in the Manticore implementation of Parallel ML [5]. While this is efficient, it necessitates change to the compiler and base libraries, heavy use of arrays and intermediate data structures, and often employs mutable operations to achieve the best results. Our design goal is to achieve improved performance through more flexible management of the available parallelism, without having to modify the structure of the program or relying on compiler-driven source code transformations.

In our approach to parallelism we take a data-centric view and provide strategies as traversals over tree-like data structures. This offers the perspective of good re-use of such strategies for different applications, and a clean separation of computation from coordination, which was one of the main design goals for the existing evaluation strategies module.

In this paper we develop novel parallelism control mechanisms, using circular programs, and embed them into evaluation strategies for tree-like data structures based on the version described in [17]. Most notably, we use the following core, functional programming techniques to achieve flexible parallelism in our implementation:

- We provide traditional parallelism control mechanisms, such as thresholding, to limit the amount of parallelism.
- We provide advanced *fuel splitting* mechanisms, which prove to be more flexible in throttling parallelism.
- For the administration of the parallel execution we add annotations to the data structure and perform lazy size computation.
- We use *circular programs* to pass fuel down and up a tree [6].
- We use *heuristics-based* parameter selection for advanced strategies to select their specific control parameters.
- We demonstrate our strategies on a *quad-tree* representation and implement a Barnes-Hut algorithm and sparse matrix multiplication using quad-trees.

More widely, the main contributions of this paper are:

- The development of advanced parallelism control mechanisms, using lazy evaluation techniques, and their integration into the evaluation strategies framework.
- Implementations of such strategies in Glasgow parallel Haskell, using a generic, k-ary tree representation.
- The comparative study of the performance of these strategies and heuristics, using a Barnes-Hut and a sparse matrix multiplication algorithm.
- The development of a data-centric approach to parallelisation that preserves existing data structures and controls parallelism dynamically.

FHPC '14, September 4, 2014, Gothenburg, Sweden.

ACM 978-1-4503-3040-4/14/09...$15.00.
http://dx.doi.org/10.1145/2636228.2636234

2. Background

Glasgow parallel Haskell (GpH) provides a single primitive to specify pure parallelism:

```
par::a->b->b
```

The `par` combinator simply *sparks* the computation of `a` to be potentially evaluated in parallel with `b`. A spark is only a pointer to potential work and there is no guarantee that `a` is computed in parallel. It is up to the system to decide if this annotation is converted to a thread. In addition to `par`, a coordination combinator `pseq` (having the same type definition as `par`) imposes a left-to-right order of evaluation, which is intentionally left unspecified in Haskell but required to arrange computations for parallel execution.

Evaluation strategies [17, 28] were introduced as a layer of abstraction on top of the basic primitives for parallel computation in Haskell. Strategies allow the separation of coordination from computation aspects, resulting in more structured parallel programs. A strategy is simply a function that takes an argument and lifts it to a context, `Eval` in this case. `Eval` encapsulates the parallel coordination of the argument in a monad. By using `runEval`, we can extract its value, which also launches any order of evaluation specified in the context. It is important that the extracted value is an identity of what was initially passed as argument. Basic strategies available include `r0` (no evaluation), `rseq` (evaluation to WHNF), `rdeepseq` (full evaluation). While these specify evaluation-degree and -order, the `rpar` strategy specifies parallelism, i.e. defining that an expression can be evaluated in parallel.

Listing 1: Evaluation Strategies

```
data Eval a = Done a

runEval::Eval a->a
runEval (Done x) = x

type Strategy a = a->Eval a

rseq, rpar::Strategy a
rseq x = x `pseq` Done x
rpar x = x `par` Done x

using::a->Strategy a->a
x `using` strat = runEval (strat x)

-- e.g. 1 evaluate x in parallel with y
compute x y = (x+y) `using` strat
  where
    -- custom strategy definition
    strat res = do
      rpar x
      rseq y
      return res

-- e.g. 2 parallel map
parMap strat f xs =
    map f xs `using` parList strat
```

There are a number of existing strategies defined in the library for basic types, and list data structure. The preferred way of applying a strategy to a value is through the `using` combinator. For instance, in the parallel map definition in Listing 1, `parList` specifies data-oriented parallelism over a list. It also demonstrates the use of higher-order functions to parameterise strategies. Strategies can be composed using the `dot` combinator (e.g. `strat2 `dot` strat1`). This is an important aspect in achieving clear separation of parallel specification in an algorithm.

3. Related Work

Our work follows the direction of high-level abstractions for efficient, flexible parallel execution, as spear-headed by algorithmic skeletons [11] and parallel patterns [18]. This direction has become more prominent in the search of an easy-to-use technique that can be applied to moderately parallel desktop machines. Successful parallel pattern libraries and packages are Microsoft's Task Parallel Library [8], Intel's Threading Building Blocks [24], and the Cilk language for light-weight parallelism [10], now also included in Intel's compiler distributions. These techniques are increasingly used by practitioners and promoted in textbooks such as in [19]. In comparison, the specific flavour of our abstractions is a data-centric one, aiming to provide parallel implementations of common operations over data structures, that can be re-used in a range of different applications.

Our approach shares the design goals of high-level data-parallel languages, such as the aforementioned DPH [23], the purely functional array programming languages SAC [14] and the earlier, influential NESL [7] language for nested data parallelism. However, we focus on structures with irregular data distribution, for which static decisions for managing parallelism are far more difficult. Therefore, our control mechanisms are mainly dynamic and low-level orchestration is entirely delegated to the runtime-system.

Okasaki [22] gives the most comprehensive treatment of purely functional data structures available. High sequential efficiency, as provided by these implementations, is the basis for efficient parallel implementations. Other work on data structures in a functional context, especially graphs, builds on a model of deterministic parallelism and the concept of LVars for explicit synchronisation [20]. This work uses a monadic programming style in Parallel Haskell, whereas our style of parallel programming focuses on pure functions and the clear separation of coordination from computation concerns.

Various modern parallel programming language designs emphasise the importance of inherently parallel data structures. One such language is X10 [9], which is an instance of the class of Partitioned Global Address Space (PGAS) languages. These languages provide a shared address space, mainly for storing large, flat data structures such as arrays, with language constructs for distributing the data structure over the available machines in a distributed memory setting. Predefined distribution policies are typically block-distribution or cyclic distribution. The algorithms tend to be data-centric, structured around this distribution, with transparent, system-controlled access to remote partitions, and implicit synchronisation and communication among the available threads. More main-stream instances of this programming model are Unified Parallel C (UPC) [13] and Co-Array Fortran [21].

Systems for implicit parallelism, such as [16], emphasise dynamic and automatic tuning of parallelism to achieve scalable performance. Heuristics to determine optimal program parameters for execution utilise information available at run-time and thus enable dynamic auto-tuning of parallel strategies. As an implicitly-threaded system, the Manticore implementation of parallel ML uses lazy tree splitting [4] to automatically manage size and number of the parallel threads in the system.

In our approach to control parallelism in a more flexible way, we use a *fuel splitting* technique to distribute resources to sub-computations. This technique is related to the use of *engines* in Scheme 84 as a notion of timed preemption for processes [15]. An engine is given a quantity of fuel and computation lasts until fuel runs out. In our context, parallelism generation, rather than evaluation, is based on fuel being available for a particular sub-tree.

In the age of multi-cores as standard CPU hardware, the need for concurrent data structures, that allow cheap, explicit multi-threading, or inherently parallel data structures, that provide im-

plicitly parallel operations over them, is widely recognised, and summarised in [25]. When using such data structures on system-level, for example, providing a stack, it is important that key properties are relaxed in order to minimise the need for explicit locking, to reduce overhead of basic operations and thus enable scalable parallelism.

4. Data structures

One of our main design goals is to develop data-centric parallelism control mechanisms that can be applied across a range of commonly used data structures and that are not tied to a particular application. We therefore focus on several variants of tree data structures, as widely used data structures with potentially irregular distribution of its contents. We note, however, that techniques such as fuel-based control, should apply to graph-structures as well.

Two important decisions in defining the tree data structure are the arity of the nodes and the value attribution (to nodes or leaves). In order to remain flexible, we parameterise our definition over both aspects, and arrive at the following generic definition of a k-ary tree [12]:

Definition 1 (k-ary Tree)**.** A tree of the form

```
data Tree k tl tn = E | L tl
                  | N tn (k (Tree k tl tn))
type QTree tl tn  = Tree Quad tl tn
```

with data elements of type `tl` in leaf nodes and data elements of type `tn` in the inner nodes is called a *k-ary tree*.

Note that in this definition the number of sub-trees is parameterised by introducing a type variable k to specify the sub-tree container. Using a 4-tuple, defined as `Quad`, gives the well-known quad-tree data structure, which we will use in the Barnes-Hut simulation [3] in Section 6. Other common choices for the container argument are: a 2-tuple, defined as `Bin`, for a binary tree; an 8-tuple, defined as `Oct`, for oct-tree (use in 3D nbody simulation); and `[]` for a rose tree with potentially varying arities in different nodes.

From now on we focus on defining strategies on this k-ary tree data structure in order to enable parallelism over the data structure, without fixing the amount of parallelism or tying the evaluation to one class of architectures.

4.1 Basic strategies

The basic strategy `parTree` creates a spark for every element in the tree. Depending on whether the tree is node- or leaf-valued, the variants `parTreeL` and `parTreeN` will spark just leaf or node elements, respectively. In our discussion we focus on the most generic version, `parTree`.

Note that the implementation of `parTree` in Listing 2, uses the `Traversable` class to arrange the traversal in a way that is not restricted to a tree data structure. Furthermore, the definition of `parTree` demonstrates that we can easily compose more complex strategies from simpler ones: we pass `(rpar ‘dot‘ strat)` as argument to the sequential `evalTree` strategy, specifying that each element should be evaluated in parallel, using the parameter `strat` to specify evaluation degree. This compositionality is inherited from the design of evaluation strategies as described in [17].

Listing 2: Data element sparking

```
evalTree :: (Traversable k) =>
            Strategy a -> Strategy (Tree k a a)
evalTree = traverse

-- parallel evaluation of inner node and leaf
    values
parTree  :: (Traversable k) =>
            Strategy a -> Strategy (Tree k a a)
parTree strat = evalTree (rpar ‘dot‘ strat)
```

Note that `parTree` does not attempt to control, or throttle, spark creation. Thus, if the tree is large this results in an abundance of parallelism, which can be detrimental to its performance due to the excessive overhead. Additionally, the current definition applies the same strategy to both inner and leaf nodes. This can be easily changed by adding a new parameter to the definition specifying different strategies for the two types of node, for example, `Strategy a -> Strategy b -> Strategy (Tree k a b)`, and tweaking the traversal function definition.

Node-level sparking In order to control granularity, we want to spark branches or sub-trees of appropriate size instead of each individual element in the tree. This is particularly useful for very large tree data structures where the overhead of element-wise sparking over-shadows the performance gain expected from parallelisation. The concept is analogous to *chunking* to ensure sufficient amount of work for each thread in the context of list data structures, but identifying which branches are of adequate size is tricky. We cover this discussion in the next section where we use a number of dynamic techniques to throttle the amount of parallelism that is generated.

Listing 3: Node-level or branch sparking

```

parTreeBranch :: Strategy (Tree Quad tl tn)
              -> Strategy (Tree Quad tl tn)
parTreeBranch strat (N n (Q nw ne sw se)) =
    (N n <$> (Q <$> parTreeBranch strat nw
                <*> parTreeBranch strat ne
                <*> parTreeBranch strat sw
                <*> parTreeBranch strat se))
    >>= (rpar ‘dot‘ strat)
parTreeBranch _ (L x)  = pure $ L x
parTreeBranch _ E      = pure E
```

In Listing 3, sparks are created to evaluate branches in parallel. In this version, no restriction is placed yet, so all branches (i.e. inner nodes) are sparked. Therefore, for quad-trees, a branch will compute between 1 and 4 elements in parallel. Note the strategy type is changed as the first argument is applied to a branch, i.e., of a `Tree` type. Note also that for this implementation we need to specify `k` is a `Quad` so pattern matching can be done, unlike the more generic implementation of `parTree` and variants.

5. Lazy Strategies

Uncontrolled parallelism creates overheads through generation of excessive sparks in GpH – many of which, if converted, will carry the usual thread management cost, and many will be overflown and never taken to execution. This negatively affects parallel performance. In this section, we present several mechanisms to throttle the amount of parallelism.

Table 1 gives an informal overview of the strategies that we develop, classifying them by some basic properties of their dynamic behaviour. The information flow column indicates whether administrative information for controlling parallelism is passed down or up. For example, depth-based thresholding passes a depth counter down the tree. Fuel-based version can also pass information up, if resources have been unused. The context column indicates how much of context information is required in a node to implement this strategy. For example, depth-based thresholding requires only information about the length of the path to the current node, whereas a lookahead strategy examines a fixed number of nodes in the sub-trees to make its decision. In the extreme, a perfect-split strategy requires complete (global) size information about the tree, and therefore incurs the highest amount of overhead. The final two columns

Strategy	Type	Info flow	Context	Parameter	Heuristics
parTree	element-wise sparks	-	-	-	-
parTreeDepth	depth threshold	down	path length	d	yes
parTreeSizeAnn	annotation	up	global	-	-
parTreeLazySize	lazy size check	down (lazy)	local	s	yes
parTreeFuelAnn	fuel with annotation			f	yes
- pure	equal fuel distr	down	local		
- lookahead	check next n nodes	down/limited	N	N	
- giveback	circular fuel distr	up/down (lazy)	local		
- perfectsplit	perfect fuel distr	down	global		

Table 1: Strategies overview and classification

specify the parameters that are used in the various strategies to control their behaviour and whether the parameter can be auto-specified by heuristics in our implementation. We will elaborate on these aspects in the following sections.

5.1 Mechanisms

Thresholding: A common technique is depth-based thresholding in order to throttle parallelism. This involves specifying an additional parameter d used to limit sparks creation to the top d levels of a tree. This is most effective for a regular tree layout and small d as the number of sparks increases exponentially at each level.

Alternatively, size-based thresholding checks sub-nodes for a minimum size s before sparking. For this, the size information needs to be readily encoded in the inner nodes of the tree, which might be done by the application anyway. Otherwise, a first pass to annotate the tree is required.

Fuel Splitting: Depth-based thresholding works well under the assumption that the major source of parallelism occurs within depth d in the tree. Spark creation is controlled, but still statically determined. The mechanism to control the amount of parallelism is fairly crude, since the number of sparks is exponential in the depth of the tree.

Fuel splitting is based on the notion of *fuel* – a limited, explicit resource that is used to throttle parallelism more flexibly. Fuel splitting offers the flexibility of defining custom functions specifying how fuel is distributed among sub-nodes, thus influencing which path in the tree will benefit most of the parallel evaluation.

Bi-directional fuel transfer: In order to enhance flexibility in splitting and transferring fuel, a bi-directional mechanism of transfer is advantageous. This way, fuel that is unused in one sub-tree, can be used in another sub-tree. To achieve this behaviour, we need *lazier* representation of numbers, for example, implementing Peano number sequence through the use of list of unit type for fuel instead of integer type. This enables us to check sub-node bounds, for example, if it has at least *n* elements. This does not force the entire sub-node to *normal form* to return true or false.

One instance of strategy definition relies on circular program definition [2] enabled only in a lazy language. Specifically, we use it in a fuel distribution function with a *giveback* technique, i.e. unused fuel by sub-nodes is pushed back up in the tree to be re-distributed elsewhere. We discuss this technique in detail in Section 5.5.

Abstraction: By defining splitting functions separate from the strategy definition, we can parameterise strategy to custom-defined split functions. Similar concept as used for clustering strategies in the original library.

Annotations: Size (if not readily encoded) and fuel information need to be attached to the tree in an annotation run. Therefore, some strategies are defined over an annotated tree type (`AnnTree`), which is the same as `Tree`, except with an added type constructor on the node of a tree, as opposed to adding another field in its definition. Size annotations are synthesised from bottom-up, making sure that the tree is annotated in a single pass. Fuel annotations depend on the distribution function. For this reason, the context for the fuel-based strategies differ. Some require size information, for example, perfect split, and others require only local lookup. Depending on how the splitting works, the annotation function will assign an amount of fuel at each node, until fuel runs out. At present, we settle for a simple annotation function, with parameterised split function for fuel distribution. More generic annotation implementations, for example, as used for the AST for Hume space analysis, or attribute-grammar [26] style have been investigated.

Heuristics: The advanced strategies are parameterised by additional variables to specify the depth d, size s, and fuel f thresholds. These can be programmer-specified or determined through a heuristics-based parameter selection based on a number of other parameters, for instance, input size, number of PEs, etc. The latter ensures that available information are used to tune the strategies.

5.2 Depth-thresholding (`parTreeDepth`)

The simple `parTreeDepth` strategy, shown in Listing 4, introduces some degree of control of spark creation by using depth as a threshold for parallelism generation. This technique is frequently used for throttling parallelism on regular trees. The depth threshold limits sparking of sub-nodes to evaluate in parallel at the top d levels in the tree.

Listing 4: Depth-based thresholding

```
1  parTreeDepth::Int->Strategy (QTree tl)
2                   ->Strategy (QTree tl)
3  parTreeDepth 0 _     t = return t
4  parTreeDepth d strat (N (Q nw ne sw se))  =
5      (N <$> (Q <$> parTreeDepth (d-1) strat nw
6                <*> parTreeDepth (d-1) strat ne
7                <*> parTreeDepth (d-1) strat sw
8                <*> parTreeDepth (d-1) strat se))
9      >>= rparWith strat
10 parTreeDepth _ _     t = return t
```

The strategy is a recursively-defined function which stops generating sparks when depth 0 is reached. It is useful to have sparks as early as possible, that is why sparks are created from the root node to the specified level d.

The main advantage of this strategy is the simplicity of its implementation, low overhead, and predictable parallelism. However, it lacks flexibility in particular for unbalanced trees, where potentially useful parallelism may reside outside the given depth threshold. Though we gain improved control over parallelism as opposed to element-wise sparking, the amount of sparks usually remains flat as an upper bound for d is specified to avoid excessive sparks. For

instance, for a regular tree, d_{max} can be set at 6 to generate a maximum of 4^6 sparks.

In the following sections we look at other techniques to automatically select the depth threshold d to improve performance.

5.3 Synthesised size info (parTreeSizeAnn)

Thresholding can be based on sub-tree sizes. The thresholding mechanism checks if the size of the sub-tree is more than s, then it creates a spark. This ensures that for smaller sub-trees in the tree, sparks are not created. If size information is not encoded in the inner-nodes, an initial traversal is needed to annotate the tree. In this particular implementation, size information is synthesised from bottom up. In the lazy size strategy (`parTreeLazySize`) we remove the need for an initial traversal, by lazily checking the size of sub-trees using Peano numbers.

5.4 Lazy size check (parTreeLazySize)

Using size threshold, as in `parTreeSizeAnn`, sparks are created for sub-trees that have at least s nodes. The idea is similar for this lazier variant except that the size check is performed by using lazy size computation for a given sub-tree, thus, not forcing complete evaluation of it. The default size function makes a full traversal, deconstructing the entire tree structure when size is demanded at the top-level. The lazy size function allows to test bounds without a full deconstruction. For instance, a function `isBoundedSize s` can return true when it has established that the sub-tree contains at least `s` nodes, without traversing the rest of the tree.

5.5 Fuel-based control (parTreeFuel)

`parTreeFuel` is an annotation-based strategy. The strategy function itself is similar to `parTreeDepth` — where instead of a depth threshold, the strategy stops creating sparks once fuel runs out (as seen in the pattern match for fuel check).

```
parTreeFuel :: Strategy (AnnQTree Fuel tl)
            -> Strategy (AnnQTree Fuel tl)
parTreeFuel strat t@(N (AQ (A f) nw ne sw se))
  | f>minfuel = (N <$> ( AQ (A f)
                    <$> parTreeFuel strat nw
                    <*> parTreeFuel strat ne
                    <*> parTreeFuel strat sw
                    <*> parTreeFuel strat se))
                >>= rparWith strat
  | otherwise = return t
parTreeFuel _     t = return t
```

A variant of this definition (which we append with `marked`) sparks only when the condition `f>minfuel` is met. With this variant the total number of sparks generated is not cumulative from the root, but only at specified (or marked) node within the tree. Thus, the number closely corresponds to the amount of fuel distributed, assuming we allocate one unit of fuel per node. This allows to have a better estimate on spark creation for a given amount of fuel. However, spark creation is delayed, which may not be desirable in certain cases, but results show that this variant (`fuelpuremarked`) performs well compared to other fuel-based strategies in our test applications.

5.5.1 Fuel splitting methods

The general usage of the fuel-based function is as follows, with the specification of the amount of fuel and distribution function (`SplitFunc`):

```
t' = (unann
      . withStrategy strat
      . fmap f
      . ann) t
```

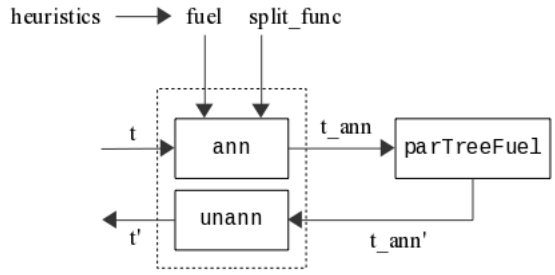

Figure 1: Annotation-based strategies, such as `parTreeFuel`, require annotating the tree in a first pass before performing the main traversal of the tree. The strategy may use heuristics to determine the amount of fuel under different parameters.

```
-- pure fuel annotation example
ann = annFuel (annFuel_pure fuel)
```

A split function can be generalised with the type `SplitFunc`, then the actual implementation is parameterisable, to easily switch between different distribution modes.

```
type SplitFunc = Fuel->[Fuel]

annFuel::SplitFunc->Fuel->QTree tl
                   ->AnnQTree Fuel tl
annFuel splitfunc _    E               = E
annFuel splitfunc _    (L x)           = (L x)
annFuel splitfunc fuel (N (Q a b c d)) =
    let (f1:f2:f3:f4:_) = splitfunc fuel
    in N $ AQ (A fuel) (annFuel splitfunc f1 a)
                       (annFuel splitfunc f2 b)
                       (annFuel splitfunc f3 c)
                       (annFuel splitfunc f4 d)

annFuel_pure::Fuel->QTree tl->AnnQTree Fuel tl
annFuel_pure = annFuel (fuelsplit_pure
    _numSubnodes)
```

The following gives implementation details of each distribution.

Pure splits fuel evenly among the sub-nodes, ignoring the node type. Fuel is lost on hitting outer nodes (empty and leaf nodes), and on division. A version that avoids any such loss has been implemented, but does not perform significantly better and is therefore not discussed any further.

```
type Fuel=Int   -- fuel as int

fuelsplit_pure::Int->Fuel->[Fuel]
fuelsplit_pure numnodes fuel =
    replicate numnodes (fuel `div` numnodes)
```

Lookahead/LookaheadN As the name suggests, this fuel distribution method *looks ahead* one level down the tree before distributing unneeded fuel to outer nodes. In the second variant, *lookaheadN*, we can specify how far down the tree we can look.

```
fuelsplit_lookaheadN::Int->QTree tl
                         ->Fuel->[Fuel]
fuelsplit_lookaheadN n (N (Q a b c d)) fuel =
   [f1,f2,f3,f4]
 where
  Q na nb nc nd = fmap (numInnerNodesUntil n)
                       (Q a b c d)
  numsubnodes     = na + nb + nc + nd
  (f1:f2:f3:f4:_) = fuelsplit_perfect fuel
                    numsubnodes [na,nb,nc,nd]
fuelsplit_lookaheadN _ _ _ = [0,0,0,0]
```

Giveback is the same idea as lookahead where we avoid losing fuel on meeting outer nodes. But instead of looking ahead down *n* level the tree, giveback employs a circular programming technique to allow passing fuel up in the tree, if it has not been used in a sub-tree. Thus, information flow is bi-directional in this implementation. This technique depends on laziness to enable circular reference, as discussed below.

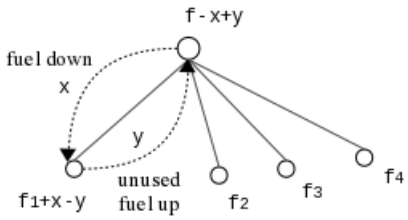

Figure 2: Fuel giveback mechanism where fuel is represented using list of units to work with the circular nature of its definition.

We note that unused fuel is passed to the next node on the right, and, if it is still unused, is passed up in the tree to be re-used in another, typically deeper, sub-tree, as depicted in Figure 2. We achieve this behaviour by using a circular definition [6]. The input to the annotation function (`ann`) takes an initial "share" of the fuel and any fuel that is returned from the left. In the code of Listing 5, the definition `f1_out` (Line 14) depends on `f4_out`, which, in three steps, depends again on `f1_out` (Line 15). In order to guarantee that this definition is productive, fuel must not be represented as an (atomic) integer, but needs to be a list of values, which is expanded by this circular definition and requires lazy evaluation.

Listing 5: Fuel with giveback annotation

```
1  -- | Fuel with giveback annotation
2  annFuel_giveback::Fuel->QTree tl
3                       ->AnnQTree Fuel tl
4  annFuel_giveback f t = fst $ ann (fuelL f) t
5   where
6    ann::FuelL->QTree tl->(AnnQTree Fuel tl,
         FuelL)
7    ann f_in E                = (E,f_in)
8    ann f_in (L x)            = (L x,f_in)
9    ann f_in (N (Q a b c d)) =
10    (N (AQ (A (length f_in)) a' b' c' d'),
          emptyFuelL)
11    where
12     (f1_in:f2_in:f3_in:f4_in:_) =
13       fuelsplit_unitlist _numSubnodes f_in
14     (a', f1_out ) = ann (f1_in++ f4_out ) a
15     (b', f2_out ) = ann (f2_in++ f1_out ) b
16     (c', f3_out ) = ann (f3_in++ f2_out ) c
17     (d', f4_out ) = ann (f4_in++ f3_out ) d
```

Auxiliary functions used in this strategy are defined below:

```
type FuelL=[()] -- fuel as unit list

emptyFuelL=[]   -- empty fuel list

fuelL::Fuel->FuelL
fuelL x = replicate x ()

fuelsplit_unitlist::Int->FuelL->[FuelL]
fuelsplit_unitlist numnodes fuel =
 split 0 fuel [emptyFuelL,emptyFuelL,
               emptyFuelL,emptyFuelL]
 where
  split _ []     xs = xs
  split x (_:fs) xs =
    split (x+1) fs (addfuelat (x `mod`
        numnodes) xs)

  addfuelat 0 [a,b,c,d] = [():a,b,c,d]
  addfuelat 1 [a,b,c,d] = [a,():b,c,d]
  addfuelat 2 [a,b,c,d] = [a,b,():c,d]
  addfuelat 3 [a,b,c,d] = [a,b,c,():d]
  addfuelat _ xs        = xs
```

Perfect fuel splitting distributes fuel based on sub-node sizes. It depends on the size information being available, otherwise an annotation run, requiring a full traversal of the tree, is needed before any parallel sub-computation (spark) is generated.

```
fuelsplit_perfect::Fuel->Size->[Size]->[Fuel]
fuelsplit_perfect fuel s ss =
  fmap (\x -> (x*fuel) `div` s) ss
```

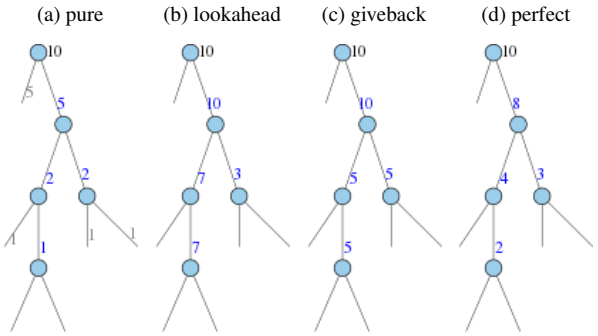

Figure 3: Example of fuel distribution methods on a binary tree

5.6 Heuristics

Using the strategies with the right parameters is crucial for optimal performance. While this can be programmer-specified, it is difficult to deduce the right values for best performance given the number of variables involved. Table 2 lists a number of variables that are used in determining an optimum parameter for the strategies during execution.

Variable name	Description
T_{in}	number of inner nodes in tree T
T_{out}	number of outer nodes
T_l	number of non-empty (leaf) outer nodes
T_e	number of empty outer nodes i.e. $T_{out} - T_l$
H_{min}	shortest path from root to any outer node
H_{max}	longest path from root to any outer node
S	number of sparks
S_{max}	max num of sparks
P	Number of processing elements

Table 2: Heuristic parameters

S is the number of sparks generated which roughly corresponds to the "amount" of parallelism desired. The granularity of these parallel computations will vary.

S_{max} refers to an upper bound that we may set in order not to have excessive sparks created by a strategy, which would otherwise cause a higher overhead. In practice, we set this to about 8000 sparks which is the maximum number of sparks that can be fitted

in the spark pool at one time. However, S_{max} can be set higher, given that sparks are created and converted during execution, and the total number of sparks that may have been created at the end of execution could be greater than 8000, but the spark pool was never overflowed. This depends on the sequence in which parallel computations happen and are specified in the algorithm.

The programmer-specified parameters are d, s and f, for the depth-threshold, lazy size and fuel-based strategies, respectively. For good performance it is important that these parameters fulfill basic properties on the tree structure. For instance, the selected value of a parameter may be out of range for a given tree. In the following sub-sections, we lay down these properties — what we expect from each strategy — and elaborate how the heuristics preserve them.

The heuristics work based on the assumption of how much information about the tree is available. In some cases, a traversal to extract this information is justifiable if the computation involved in the nodes is substantial enough. For instance, one version of the depth heuristics (**D2**) works well if the number of nodes at each level is known.

5.6.1 Determining d

The heuristics should guarantee and satisfy the following properties:

Invariant for d

1. d should be within the range
$$0 < d < H_{max} \wedge d \leq d_{max}$$
d cannot be outside the depth bounds of a tree. d_{max} is a maximum set by the programmer (hard-coded) or estimated in the more advanced heuristic functions.
2. For any selected d, $S < T_{in}$
3. For any selected d, $S < S_{max}$
The number of sparks generated for a given d should not exceed the maximum spark limit. S_{max} is the sparks cutoff point.

We now define some of the heuristics used to determine d.

D0 $d = H_{max}/2$
This assumes that half-way down the tree, we have sufficient parallelism to create. This version does not take P into account and generates a fixed number of sparks for any P.

D1 $d = min\ (P-1)\ d_{max}$
where d_{max} in this version is hard-coded. It selects the depth parameter, based on the maximal depth of the input tree.
The heuristic is implemented through a counter that starts with $d = 0$ on 1 PE and increases d by one from 2 PEs onward until d_{max} is met. This leads to $\sum_{x=0}^{P-1} 4^x$ sparks on P PEs. There is no maximum sparks control (Property 3) in the version.

D2 $d = min\ (P-1)\ d_{max}$, where cumulative nodes at d is less than S_{max}
In this version, d_{max} is computed (not specified) to enforce the "where" condition. d_{max} is dependent on the input size and is determined by building a table consisting of number of nodes and cumulative number of nodes at each level. d_{max} is the level at which the cumulative number of nodes is just before S_{max}. Note that $d_{max} < H_{max}$.

D2 is based on information obtained from an initial traversal of the tree. However, where not possible, we work on an estimate. For any tree, there is at most (upper bound) 4^i nodes at level i, and the cumulative nodes at this level is $\sum_{x=0}^{i} 4^x$. For instance, the upper bound for number of nodes at level 5 is 1024, and cumulative nodes is 1365. The cumulative nodes corresponds to the upper bound of sparks that is to be created at level. We work within these bounds to determine a maximum d. Any d greater than this will generate sparks in excess. For instance, at level 6, at most 5461 sparks are created, and at level 7, 21845 sparks.

5.6.2 Determining s

At present we use the same heuristics to determine s for both `parTreeSizeAnn` and `parTreeLazySize`. The choice of an s can limit or create more parallelism. Small s will create more sparks, while big s will create less. When P increases, we want to be able to have more sparks, thus smaller s.

Invariant for s

1. s should be within the range $0 < s < T_{in}$

The number of sparks created for the size annotation and lazy size strategies is directly related to s. If s is big, fewer sparks are likely to be created. s is seen as the minimum size threshold for `parTreeSizeAnn`. Sparks are created until a sub-tree size is less than s, which translates to the amount of computation in that node is small given its size.

Used in a slightly different "context" with the `parTreeLazySize` strategy, s refers to the minimum number of nodes check, performed lazily by the strategy in order to decide whether to create a spark or not.

S0
$$s = \begin{cases} \frac{T_l}{(P \times X)} & \text{if } P > 1 \\ 0 & \text{otherwise} \end{cases}$$
where X is an approximate number of sparks per PE.

We use an estimate function to classify computation in nodes as S (small), M (medium), and L (large), and based on this "weight" we determine X. For instance, for a small amount of computation, it is fine to have many sparks per core (for example, 100–200), but for a large amount of computation, sparks are restricted to ca 5–10.

5.6.3 Determining f

As the number of cores (P) increases, we want to provide more fuel such that more sparks can be created. f_{min} is the minimum amount of fuel that is needed for a spark.

Properties for f In principle, f should be within the range $0 < f < T_{in}$ (same as s), to effectively control the potential parallelism. If $f > T_{in}$, all nodes have at least 1 fuel. Thus, the default fuel check of at least 1 ($f > 0$) changes, for example, to $f > 5$ in order to ensure the mechanism works. Otherwise sparks are created at each inner node in the tree, defaulting to the naive `parTreeBranch` strategy. Thus, we require these properties:

1. if $f < T_{in}$, then $f_{min} < 0$
This means that fuel will run out during distribution, and thus nodes with 0 fuel (or negative) will not be sparked.
2. if $f > T_{in}$, then $f_{min} > 0$
This means that potentially all nodes will have some fuel, and we need to determine f_{min}, that is, the new minimum fuel threshold a node needs to have to be eligible for a spark.
3. f should be less than sparks upper bound ($f < S_{max}$)

We explored the following heuristic formulae for computing the fuel:

F0 The initial heuristics is designed to be simple:
$$f = \begin{cases} \frac{T_l}{X} \times P & \text{if } P > 1 \\ 0 & \text{otherwise} \end{cases}$$

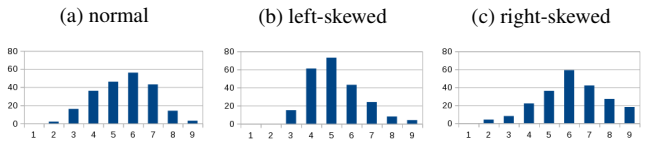

Figure 5: Depth distribution for `testprog` input

F1 The final heuristics is defined as follows: $f = a^{h(P)}$ where h is a function over the number of processors and the constant a is the arity of the tree, i.e. 4 for quad-trees. We use h in the exponent of this formula to reflect the exponential number of potential parallelism in the tree structure.

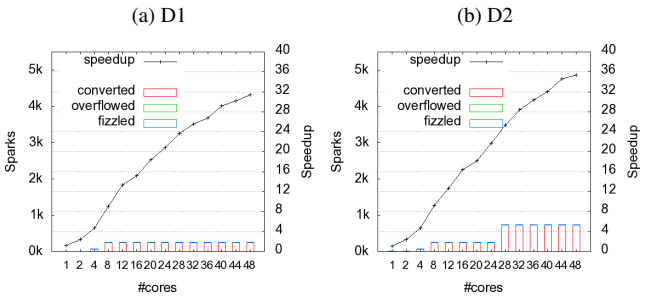

Figure 6: Depth heuristics performance comparison: D1 vs D2

6. Evaluation

6.1 Experiment environment

Machines: Initial test runs were carried on a *desktop-class* 8-core machine — arranged in 2 sockets x Intel Xeon CPU E5410 (4 cores) @ 2.33GHz — with 7870MB memory and 2 levels of cache hierarchy.

We also use a *server-class* many-core machine consisting of 48 cores — arranged in 4 sockets each with 2 NUMA nodes and each node with 6 AMD Opteron 6348 CPU cores (1400 MHz). Two cores share a 64kB L1 and a 2MB L2 cache, and the 6MB L3 cache is shared by all cores in one NUMA region. Each region has 64GB memory, amounting to a total of 512GB RAM. Both machines run CentOS 6.5 64-bit Linux with kernel version 2.6.32.

Being a NUMA architecture, memory latencies vary depending on the region. Using the `numactl` tool shows that access to a remote region is by a factor of 2.2 more expensive than access to a local region.

Compiler and libraries The Haskell compiler used is `ghc-7.6.1` with the `parallel-3.2.0.3` package and our initial `pardata-0.1` package consisting of the new strategies. All programs are compiled with optimisation flag `-O2` on.

Tools We identified useful visualisation tools to help in the implementation of the strategies and verify the actual with expected behaviour which is important in a non-strict language:

GHood allows to observe intermediate states in the data structure as evaluation proceeds. Different colour scheme is used to highlight unevaluated thunks and evaluated structures.

ghc-vis is a tool to visualise live Haskell data structures in GHCi. Evaluation is not forced and you can interact with the visualised data structures. This allows seeing Haskell's lazy evaluation and sharing in action.

6.2 Test Program

The strategies are first tested on a constructed program taking algorithmic complexity out of the picture and focusing on the strategies' behaviour. The test program performs a parallel map on irregular trees with different depth distribution — normal, left-skewed and right-skewed — and fixed (homogeneous) or variable (heterogeneous) computations in the element.

Initial results on a desktop machine with a small input size show good speedups on up to 8 cores with all the strategies. However, the advanced strategies do not outperform the naive element-wise `parTree`, as expected. The 8-core machine run was mainly intended as a check for initial performance for the advanced strategies.

Figure 4 shows the absolute speedup graphs of the different strategies against the sequential runtime of the test program on the server machine. For this many-core machine, we use a larger input size of 100k tree elements. The depth-based thresholding strategy works well with a speedup close to 32 on 48 cores. However, the main problem with this basic thresholding method is that the number of sparks generated remains flat after 8 cores. This is due to the maximum depth we encode in our heuristics as an upper bound in order to avoid excessive parallelism. Additionally, we see an improvement from using heuristics D2 over D1 as seen in Figure 6. This demonstrates that for irregular trees, we can have a high depth threshold determined dynamically to go deeper down the tree in order to generate sufficient parallelism, while avoiding a hard-coded maximum d.

The lazy size strategy performs well up to 32 cores with a speedup of 28 compared to 26 on the same core count for the depth strategy. This is explained by the fact that spark creation grows with increasing number of cores. The speedups range from 28 to 30 on 36 to 48 cores, which may be attributed to many more sparks being created on higher core numbers, which introduces some overhead and motivates the need for throttling parallelism.

The pure fuel strategy also gives good results compared to depth-based thresholding. This result is further improved by extending the fuel strategy with a lookahead mechanism — in this case, the amount of sparks generated is the same as the pure version, however, the performance gain comes from the improved efficiency in fuel distribution, marking the most eligible nodes to be sparked based on additional information from the lookahead mechanism.

The giveback fuel strategy has performance close to the depth-based thresholding strategy, even though we note that the giveback technique generates far more sparks than the remaining strategies with the same amount of fuel. In analysing the giveback mechanism, we measure how often fuel was given back to a parent node. With $f = 50$, the fuel hit-rate (the number of times an outer node is hit, thus fuel is passed back upward) is 247. For $f = 100, 500$ and 1000, the hit-rates are 478, 2357 and 4417, respectively. These numbers demonstrate that the giveback mechanism is effective in enabling additional parallelism for irregular trees. Due to the circular distribution of fuel, we expect more fuel to be available for nodes inside the tree — that is, distribution carries on deeper inside the tree, explaining why higher numbers of sparks are generated.

6.3 Barnes-Hut Algorithm

The Barnes-Hut algorithm for the n-body problem is used as a more realistic application to assess the performance of our new strategies. The algorithm simulates the movement of objects in space and uses a quad-tree representation for 2D space. Its implementation is in two phases:

Tree construction: a tree containing all the bodies in the given space is constructed.

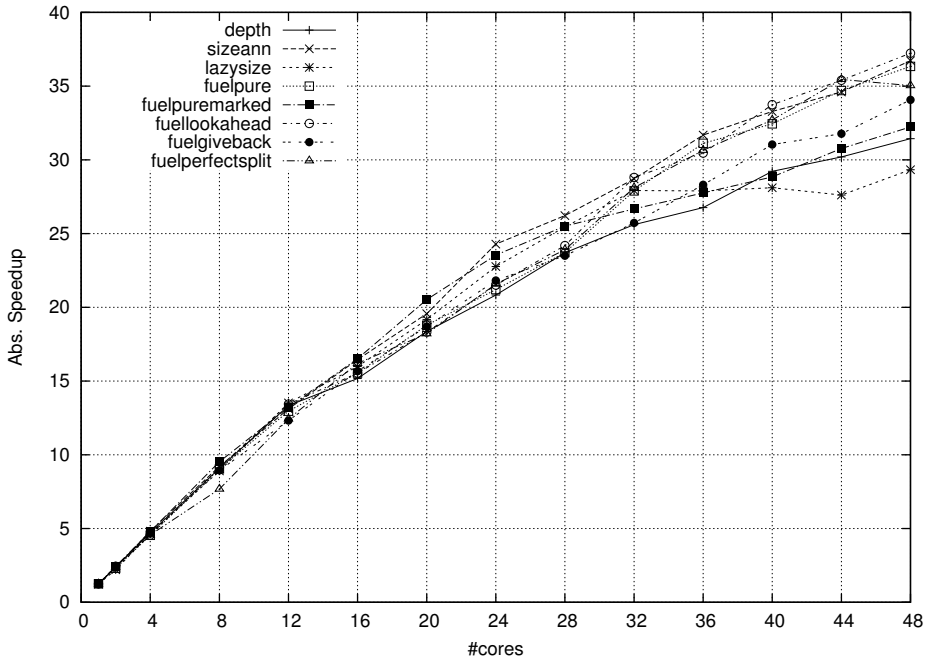

Figure 4: `testprog` speedups on 1-48 cores. 100k elements.

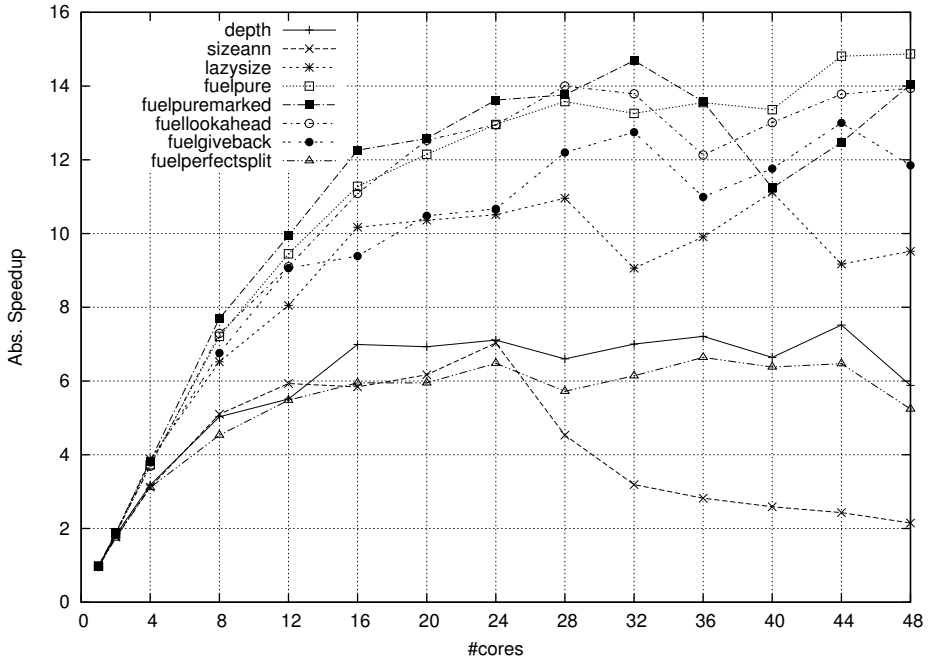

Figure 7: Barnes-Hut speedups on 1–48 cores. 2 million bodies, 1 iteration.

Force calculation: iteration-wise force calculation for each body with respect to the rest, followed by positions change.

The main source of parallelism is in the force calculation step, where the force for each body can be computed independently from the other bodies. We adapt the list-based algorithm used in [27] by performing the main map operation in the second phase over a quad-tree data structure. This enables us to use our more advanced strategies defined here, while comparing the performance with an already well-tuned implementation. Most notably, no restructuring of the sequential code was necessary to enable parallelism. We also extend our experiments to include a number of different body distributions — single uniform cluster, single normally distributed cluster, and multiple clusters of bodies, as depicted in Figure 8. This

is aimed to study the performance of our strategies with irregular data distributions.

(a) uniform cluster (b) normal cluster (c) multiple clusters

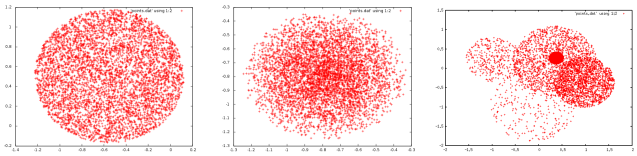

Figure 8: Bodies distribution

Performance Results: We focus our discussion on a particular set of results obtained for a single cluster input of normally distributed bodies across the space and compare these with results from a multiple clusters input. The latter is an example of irregularly distributed data.

The speedup results in Figure 7 show that across the range of core numbers, the fuel-based strategies are consistently more efficient than `parTreeDepth` or `parTreeSizeAnn`, for depth-based and size-based thresholding, respectively. In particular, a pure fuel version, using just simplistic but cheap fuel splitting, performs best on 48 cores, exhibiting a speedup of 15, and the marked variant of the fuel strategy performs best for core numbers up to 36. These results indicate that a fuel-based approach to controlling parallelism is more flexible than a thresholding approach. The latter performs very poorly from 16 cores onwards and drops in performance on the high end of the spectrum. One inherent problem with depth-based thresholding is that it provides only a very crude mechanism for controlling the amount of parallelism that is generated, because the number of sparks is exponential in the depth that is used as threshold. Furthermore, it misses out on opportunities of re-using potential parallelism late in the computation, where parallelism typically diminishes, due to having hit the depth threshold at this point in the computation. This shows up as a step function in the profile plotting sparks over cores (similar to the graph in Figure 6). In contrast, the same profile for the fuel-based strategy shows a continuous function, where parallelism steadily increases over the number of cores.

Among the fuel-based strategies, the pure variant performs best, but other variants remain fairly close to it. In particular, the give-back variant is within 20% and the lookahead variant is within 6% of the best result.

These versions invest more work into orchestrating the parallelism, by passing fuel through the tree: measuring this overhead, as discussed in Section 5, shows that for an input of 2 million bodies, annotating the tree takes about 11% and unannotating the tree takes about 4% of the time needed to build the tree. The overall performance is therefore a balance between this overhead and the more flexible distribution of fuel. For example, using a give-back mechanism to distribute the fuel both down-wards and up-wards, shows that for a tree with 100 thousand elements, and a fuel of 2000, there are 7682 instances of give-back, due to the irregular distribution of the data in the input.

We also observe that a perfect split strategy performs poorly in Figure 7. Again we believe that this is due to the additional overhead incurred by this strategy. Notably, the three worst performing strategies in this figure, are the ones that need a global context in order to decide how to arrange the parallelism (this can be seen in Column 4 of Table 1).

The limited scalability beyond 28 cores can be attributed to specifics of the algorithm, the runtime system and the hardware. The algorithm itself is a standard Barnes-Hut algorithm without further optimisations to minimise data exchange and facilitate scalability. Using an increasing number of cores will naturally generate a higher number of threads in our programming model, which increases the amount of live data and thus the garbage collection overhead. Additionally, global synchronisation across all cores is necessary to perform major collections. The underlying physical shared-memory hardware is a NUMA architecture with higher memory latencies across remote NUMA regions, of 6 cores. Thus, involving several regions in the computation will result in a significant percentage of expensive memory accesses. This effect is even more pronounced in Haskell programs, since the underlying graph reduction machinery typically requires frequent memory accesses across a large, dynamic heap. For a more detailed study of the parallel memory management performance see [1].

6.4 Sparse Matrix Multiplication

Wainwright et al [29] report good sequential performance and reduced space overhead for sparse input data, using a quad-tree representation for sparse matrix multiplication. We adapt the implementation in Haskell and aim for further performance gains through parallelisation using our new strategies. This is achieved by demanding the result matrix in parallel. Again, we do not have to change the sequential algorithm to obtain a parallel version.

Listing 6: Sparse matrix multiplication

```
qmul::(Eq a,Num a)=>QTree a->QTree a->QTree a
...
let res = qmul ma mb
in res `using` strat
```

Early results for 4096x4096 input matrices with 5% of all elements containing non-zero values (sparsity), in Figure 9, show fairly good performance on core numbers up to ca. 12 or 20, i.e. typical sizes for current desktop machines. However, there is a significant drop in performance thereafter and therefore poor scalability for now. One specific characteristic for this application is its fairly high memory allocation throughout the execution: total allocation is 4 times and memory residency is 3 times that of the Barnes-Hut algorithm. As a result, garbage collection (GC) overhead is high and steadily increasing for higher core numbers. We note that at the point where speedups drop, around 16 to 18 core, the GC% in the execution surpasses the MUT%, i.e. the mutation time spent doing actual graph reduction. This is an indication that all versions suffer from high GC overhead. Thus, it would be profitable to throttle the parallelism more aggressively, and this is the direction we want to explore in the future.

Comparing the performance of the different strategies reveals that again the marked variant of a fuel strategy performs best for core ranges between 12 and 20. The depth-based thresholding variant performs significantly better in this application, probably because the result of matrix multiplication is much denser than its input. Since our lazy strategies typically generate parallelism by traversing, and thus forcing evaluation of the result data structures, this means that the data is more regularly distributed compared to the Barnes-Hut program. Because we are not performing any operation on the elements of the result matrix, it is expected that `sizeann` and `fuelperfectsplit` do not give good performance as both require a first pass over the result matrix to attach administrative information.

7. Conclusion

We have presented new parallelism control mechanisms, building on laziness to achieve additional flexibility. We have encoded these as evaluation strategies over tree-like data structures and demonstrated improved parallel performance over established methods for throttling parallelism on a 48-core shared-memory server using three benchmark programs. Our new strategies are more flexible in controlling the available parallelism, by re-using previously

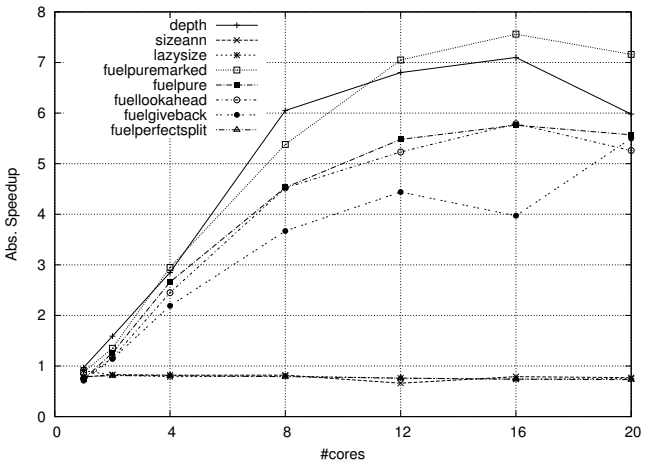

Figure 9: `spmatmult` speedups on 1-20 cores; 4096x4096 input matrices, sparsity 5%.

unused potential parallelism and thus obtaining better performance on structures with irregular data distribution. Our performance results show that they out-perform classic techniques that are often used, such as depth based thresholding. Core techniques that we use in the implementation to gain this added flexibility are circular programs, requiring lazy evaluation, and annotating the tree structure with administrative information. The best-performing strategy is based on the notion of *fuel* that is passed down the tree and controls whether parallelism should be generated or not. While our implementation and performance results have been obtained from an quad-tree data structure, the techniques, such as bi-directional flow of fuel, are not restricted to trees nor to specific applications.

As future work we plan to apply these techniques to graph data structures, which are increasingly used in symbolic high-performance computing to model huge dependency sets, often labeled as "big data" computing. The emergence of the Graph500 benchmarks as an alternative to the established numerical high-performance benchmarks, shows a movement of the community in this direction. More specifically to the results in this paper, we plan to enhance the heuristics for parameter selection and to explore the prospect of using a attribute-grammar style of specifying the propagation of synthesised and inherited parameters through the data structure. This style could provide a more user-friendly and familiar framework for fine-tuning parallelism, while still defining behaviour and desired properties on a high level. Finally, we want to further improve the compositionality of the mechanisms discussed in this paper. It would be desirable to provide high-level constructors, to freely combine a general fuel mechanism, for driving the parallelism, with a lookahead mechanism, for controlling the precision of contextual information, and a giveback mechanism, for adding flexibility in managing the parallelism.

An online version of this paper, together with the source code for the `pardata` package and applications, is available at: http://www.macs.hw.ac.uk/~dsg/gph/papers/abstracts/fhpc14.html

Acknowledgments

We would like to thank SICSA (The Scottish Informatics and Computer Science Alliance) for funding the first author through a PhD studentship.

References

[1] M. Aljabri, H.-W. Loidl, and P. Trinder. Distributed vs. Shared Heap, Parallel Haskell Implementations on Shared Memory Machines. In *Draft Proc. of Symp. on Trends in Funct. Program.*, TFP'14, Univ. of Utrecht, The Netherlands, 2014.

[2] L. Allison. Circular programs and self-referential structures. *Soft.: Prac. and Exp.*, 19(2):99–109, 1989. ISSN 1097-024X. URL http://dx.doi.org/10.1002/spe.4380190202.

[3] J. Barnes and P. Hut. A hierarchical o(n log n) force-calculation algorithm. *Nature*, 324(6096):446–449, Dec. 1986. URL http://dx.doi.org/10.1038/324446a0.

[4] L. Bergstrom, M. Fluet, M. Rainey, J. Reppy, and A. Shaw. Lazy Tree Splitting. *J. of Funct. Program.*, 22(4–5):382–438, 2012. URL http://dx.doi.org/10.1017/S0956796812000172.

[5] L. Bergstrom, M. Fluet, M. Rainey, J. Reppy, S. Rosen, and A. Shaw. Data-Only Flattening for Nested Data Parallelism. In *Proc. of the ACM SIGPLAN Symp. on Princ. Pract. of Par. Program.*, PPoPP'13, pages 81–92, Feb. 2013.

[6] R. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, 21(3):239–250, 1984. ISSN 0001-5903. URL http://dx.doi.org/10.1007/BF00264249.

[7] G. E. Blelloch. NESL: A Nested Data-Parallel Language. Technical report, Carnegie Mellon University, Pittsburgh, PA, USA, 1992.

[8] C. Campbell, R. Johnson, A. Miller, and S. Toub. *Parallel Programming with Microsoft .NET — Design Patterns for Decomposition and Coordination on Multicore Architectures*. Microsoft Press, Aug. 2010. URL http://msdn.microsoft.com/en-us/library/ff963553.aspx.

[9] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An Object-Oriented Approach to Non-uniform Cluster Computing. In *Proc. of the ACM Conf. on OO Prog. Sys. Lang. and App.*, OOPSLA'05, pages 519–538, 2005. URL http://dx.doi.org/10.1145/1094811.1094852.

[10] Cilk. *Cilk 5.4.6 Reference Manual*. MIT, Supercomputing Technologies Group MIT Laboratory for Computer Science, 1998. URL http://supertech.lcs.mit.edu/cilk.

[11] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, Cambridge, MA, USA, 1991. ISBN 0-262-53086-4.

[12] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, third edition, 2009. ISBN 978-0262033848.

[13] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick. *UPC: Distributed Shared Memory Programming*. John Wiley and Sons, 2005. ISBN 9780471478362. URL http://dx.doi.org/10.1002/0471478369.fmatter.

[14] C. Grelck and S.-B. Scholz. SAC: A Functional Array Language for Efficient Multi-threaded Execution. *Int. J. Parallel Program.*, 34(4):383–427, Aug 2006. URL http://dx.doi.org/10.1007/s10766-006-0018-x.

[15] C. T. Haynes and D. P. Friedman. Engines build process abstractions. In *Proc. of the ACM Symp. on LISP and Funct. Program.*, LFP'84, pages 18–24, New York, NY, USA, 1984. ACM. ISBN 0-89791-142-3. URL http://doi.acm.org/10.1145/800055.802018.

[16] L. Huelsbergen and J. Larus. Dynamic Program Parallelization. In *Proc. of the ACM Conf. on LISP and Funct. Program.*, LFP'92, pages 311–323, New York, NY, USA, 1992. ISBN 0-89791-481-3. URL http://doi.acm.org/10.1145/141471.141567.

[17] S. Marlow, P. Maier, H.-W. Loidl, M. K. Aswad, and P. Trinder. Seq no More: Better Strategies for Parallel Haskell. In *Proc. of the 3rd ACM Haskell Symp.*, Haskell '10, pages 91–102, New York, NY, USA, 2010. ISBN 978-1-4503-0252-4. URL http://doi.acm.org/10.1145/1863523.1863535.

[18] T. G. Mattson, B. A. Sanders, and B. L. Massingill. *Patterns for Parallel Programming*. Addison-Wesley, 2004. ISBN 978-0321228116.

[19] M. McCool, A. Robison, and J. Reinders. *Structured Parallel Programming*. Morgan Kaufmann, 2012. ISBN 978-0-12-415993-8.

[20] P. Narayanan and R. Newton. Graph Algorithms in a Guaranteed-Deterministic Language. In *5th Workshop on Deter. and Correct. in Par. Program.*, WODET 2014, March 2, Salt Lake City, UT, USA, 2014. URL http://wodet.cs.washington.edu/.

[21] R. Numrich and J. Reid. Co-arrays in the Next Fortran Standard. *ACM SIGPLAN Fortran Forum*, 24(2):4–17, Aug. 2005. ISSN 1061-7264. URL http://doi.acm.org/10.1145/1080399.1080400.

[22] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1999. ISBN 978052166350. Sept.

[23] S. Peyton Jones. Harnessing the Multicores: Nested Data Parallelism in Haskell. In *Program. Lang. and Sys.*, LNCS 5356, pages 138–138. Springer, 2008. ISBN 978-3-540-89329-5. URL http://dx.doi.org/10.1007/978-3-540-89330-1.

[24] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly, 2007.

[25] N. Shavit. Data structures in the multicore age. *Commun. ACM*, 54(3): 76–84, Mar. 2011. ISSN 0001-0782. URL http://doi.acm.org/10.1145/1897852.1897873.

[26] S. Swierstra, P. Azero Alcocer, and J. Saraiva. Designing and Implementing Combinator Languages. In *Ad. Funct. Program.*, LNCS 1608, pages 150–206. Springer, 1999. ISBN 978-3-540-66241-9. . URL http://dx.doi.org/10.1007/10704973_4.

[27] P. Totoo and H.-W. Loidl. Parallel Haskell Implementations of the N-body Problem. *Conc. and Comp.: Prac. and Exp.*, 26(4):987–1019, Mar. 2014. URL http://dx.doi.org/10.1002/cpe.3087.

[28] P. Trinder, K. Hammond, H.-W. Loidl, and S. Peyton Jones. Algorithm + Strategy = Parallelism. *J. Funct. Program.*, 8(1):23–60, Jan. 1998. URL http://dx.doi.org/10.1017/S0956796897002967.

[29] R. L. Wainwright and M. E. Sexton. A study of sparse matrix representations for solving linear systems in a functional language. *J. of Funct. Program.*, 2(01):61–72, 1992. URL http://dx.doi.org/10.1017/S0956796800000265.