CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2012; **00**:1–30
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

Parallel Haskell implementations of the n-body problem

Prabhat Totoo, Hans-Wolfgang Loidl

School of Mathematical and Computer Sciences, Heriot-Watt University, Edinburgh, EH14 4AS, UK.

SUMMARY

This paper provides an assessment of the advantages and disadvantages of high-level parallel programming models for multi-core programming by implementing two versions of the n-body problem. We compare three different parallel programming models based on parallel Haskell, differing in the ways how potential parallelism is identified and managed. We assess the performance of each implementation, discuss the sequential and parallel tuning steps leading to the final versions, and draw general conclusions on the suitability of high-level parallel programming models for multi-core programming. We achieve speed-ups of up to 7.2 for the all-pairs algorithm and up to 6.5 for the Barnes-Hut algorithm on an 8-core machine.

Received . . .

KEY WORDS: high-level parallel programming models; functional programming; n-body problem;

1. INTRODUCTION

Modern multi-core architectures make the computational power of parallel hardware available to domain experts, who want to implement compute-intensive applications. However, most established parallel programming technology makes it challenging for non-specialists in parallel programming to exploit this potential. In contrast, high-level parallel programming models simplify the challenging tasks of parallel programming by offering powerful abstractions and by hiding the low level details of synchronisation and coordination from the programmer [30]. Writing parallel programs using these models usually involves only small changes to the sequential algorithm and therefore does not obscure the core computation. This comes, of course, at the expense of additional runtime overhead. Despite being high-level, the programming model should be detailed enough to allow for targeted performance tuning, enabling the domain expert to use algorithmic knowledge in order to improve the utilisation of multi- and many-cores.

Several high-level language models are built on the purely functional, non-strict programming language Haskell. While all of them are significantly higher than explicit, thread-based orchestration as in C+MPI, they vary in their support and flexibility. Completely implicit models of parallelism try to hide all aspects of parallelism to the programmer and rely on compilation and runtime system technology for parallelisation. Semi-explicit models of parallelism only require the programmer to identify the available parallelism. All aspects of synchronisation and communication are handled automatically by the runtime-system, the compiler or library code. Explicit models of parallelism expose the notion of independent threads to the application programmer and provide means of explicit synchronisation.

This paper considers three current variants of parallel Haskell including: the semi-explicit *GpH* [35, 50, 34], which provides basic primitives for introducing parallelism and controlling order and degree of evaluation; the semi-explicit *Eden* [32], which identifies parallelism through the notion of processes and is implemented for distributed-memory architectures; and the explicit *ParMonad* [36], which provides a new programming model for explicit, deterministic parallel programming in Haskell.

To compare these models we implement the n-body problem which represents an important class of problems in many different areas of science including molecular dynamics and astrophysics. It is the core of a real application and representative for a wide range of applications. As a baseline for comparison, we first implement a naive all-pairs algorithm exploiting obvious data-parallelism and tuning its parallel performance. As a more efficient algorithm, we then implement the more complex Barnes-Hut algorithm which presents more challenge in getting a good parallel implementation.

The development of the parallel implementations uses a proven methodology, we have developed for languages with semi-explicit parallelism on the basis of several symbolic computations in the past [31]. Notably, it makes use of a rich set of pre-defined parallel patterns, or skeletons, that encode optimised, parallel patterns of computation as higher order functions.

This work contributes to the wider effort put together by participants in the SICSA Multicore Challenge* where parallel implementations of the naive algorithm in a variety of languages were presented. The results varied from language to language. Our aim is to look into implementations using different parallel models in Haskell and evaluate which one is best suited for this particular problem. The comparison takes into account the usability of the models and the required tunings in order to get the optimal parallel implementation.

The structure of the paper is as follows. We first look at the problem description and solving methods, followed by a discussion of the three parallel programming models in which we cover the internal implementation of each as well as user-level functions. The two subsequent sections look at the sequential and parallel implementations respectively. The performance of the models is then discussed and we cover related work in the area before concluding.

2. THE PROBLEM

The n-body problem is the problem of predicting the motion of a system of N bodies that interact with each other gravitationally. In astrophysics, the bodies are galaxies or stars, and the movement of the bodies are affected by the gravitational force. The computation proceeds over time steps where the acceleration of each body with respect to the others is calculated and then used to update the velocity and position in each iteration. We look at two commonly used methods of solving the problem: the first method is used mainly for simulation consisting of up to a few thousands number of bodies while the second method is best suited for system of large number of bodies, for example, interactions between molecules in biological systems.

2.1. *All-Pairs Algorithm*

The all-pairs method is the traditional brute-force technique in which the pair-wise accelerations among the bodies are calculated. The body-to-body comparison requires a time complexity of $O(N^2)$ to complete and thus making it convenient for only small number of bodies. It is relatively simple to implement in an imperative language using a double nested loop with *inplace* update.

In pure Haskell, the absence of *destructive update* makes the implementation a bit different. Loops are implemented using recursive function calls. However, a nested mapping function can also be employed to get the same behaviour.

2.2. *Barnes-Hut Algorithm*

In a system consisting of huge number of bodies, the traditional approach may not be feasible. In this situation, using a hierarchical force-calculation algorithm for example Barnes-Hut [3] provides an approximate and efficient solution.

The tree-based algorithm proceeds by recursively sub-dividing the region containing the bodies into smaller regions and calculating the centre of mass and total mass of each region. A tree is

*SICSA Multicore Challenge: n-body computation.
http://www.macs.hw.ac.uk/sicsawiki/index.php/Challenge_PhaseII

DOI: 10.1002/cpe

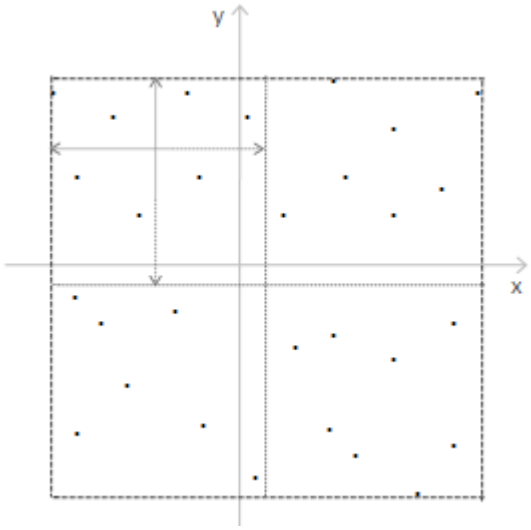


Figure 1. Bodies or points in a 2D simulation are contained in a region (bounding box) which is sub-divided recursively into smaller regions.

constructed with its root representing the entire region and children nodes as sub-region. The force-calculation for each body is done by traversing the tree and approximating bodies that are too far away (determined using a threshold) using the region centre of mass. The tree construction phase does not usually lead to a well-balanced structure which creates irregular parallelism.

3. TECHNOLOGY

While the problem has been implemented across a wide variety of programming languages covering many paradigms, using a functional approach raises the level of abstraction by making it easier to express the problem, for example, through use of higher-order functions. As a pure functional language, it is free of many of the inherent problems associated with imperative programming e.g it disallows side effects which leads to *referential transparency*, thus making it easier to parallelise programs.

Haskell had good support for parallelism but for *concurrent* programming, it provides concurrent execution of threads in an interleaved fashion using operations from the Control.Concurrent library [43]. This is a more explicit way of spawning multiple IO threads that execute at the same time. This approach is non-deterministic and can result in race conditions and deadlocks. This paper does not deal with concurrency but rather pure parallelism to speed up a program. But we will shortly see how the *ParMonad* builds on this technology to deliver a new deterministic model for parallel computation. We briefly describe *GpH*, the *ParMonad* and *Eden* as parallel programming models in Haskell. Data Parallel Haskell (DPH) provides a model of nested data parallelism in Haskell, and an implementation of the Barnes-Hut algorithm, with a discussion of flattening transformations to improve performance, is given in [25].

3.1. Parallelisation Methodology

We follow guidelines established in [31, Sec 3] with some flexibility. The approach we use is as follows:

- **Sequential implementation:** Start with the initial sequential algorithms.
- **Sequential optimisation:** Optimise the sequential algorithms e.g. improve heap consumption by identifying any space leaks and fix them; use tail recursive functions; and add strictness where necessary.

DOI: 10.1002/cpe

- **Time profile:** To point out "big eaters", i.e. parts of the programs that are compute-intensive.
- **Top-level parallelisation:** Parallelise top-level data independent operations e.g. map functions representing data-oriented parallelism, and independent tasks representing task-oriented parallelism, using high-level constructs provided in the parallel programming model.
- **Parallel execution and initial results:** Run parallel programs on multi-core machine or cluster to obtain initial results.
- **Parallel tuning:** To achieve better runtimes and speedups, use more advanced and explicit features of the model. This step also looks at scalability of the parallel programs by tuning for varying/increasing input sizes.

3.2. GpH*: Glasgow parallel Haskell*

GpH extends Haskell with two basic primitives to enable semi-explicit parallelism: `par` for parallel composition and `pseq` for sequential composition [50, 34, 35]. The `par` primitive "sparks" its first argument, i.e. it records it to potentially be evaluated in parallel. The `pseq` primitive evaluates its first argument to WHNF (Weak Head Normal Form) before continuing with the evaluation of its second argument and thus enforces sequential ordering. Both primitives return their second argument as the result.

```
-- parallel composition
par :: a -> b -> b
-- sequential composition
pseq :: a -> b -> b
```

For parallel execution, the program needs to be compiled with the `-threaded` option. The runtime option `-Nx` needs to be specified where `x` represents the number of processors.

Sparks are added to a spark pool and are taken up for execution by lightweight Haskell threads which in turn are mapped down to the underlying OS threads. Creating sparks using `par` is cheap, just a pointer, and thousands of them can be created. Converted sparks represent parallelism extracted from the algorithm, incurring the usual thread creation overhead.

Using the primitives only may often lead to wrongly specify parallelism and obscure the code. *Evaluation strategies* provide an abstraction over this level of programming, separating the coordination and computation concerns. An evaluation strategy is basically a function of type `a -> Eval a` that is executed for coordination effects.

```
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)

-- applying strategy e.g.
somefunc strat = someexpr `using` strat
```

The basic strategies `rpar` and `rseq` are defined directly in terms of their primitives. The `using` function applies a strategy to an expression. Since all parallelism, evaluation order and evaluation degree are specified within the `Eval` monad, an explicit `runEval` is used at the point where it is applied to a concrete expression. Using a monad helps to separate the purely functional aspects of the execution from the behavioural aspects of the execution. It also allows the programmer to use rich sets of libraries and abstractions available for monads in Haskell.

Strategies can be composed just like functions using the `dot` strategy combinator e.g. `(rpar `dot` rdeepseq)` sparks parallel evaluation of its argument and completely evaluates it to normal form. In this example `rdeepseq` is used to specify the evaluation degree. The expressive

DOI: 10.1002/cpe

power of evaluation strategies comes from the ability to compose them, as above, to separate the specification of parallelism from evaluation degree and other parallelism optimisations such as clustering, as we will see later, and the possibility to nest strategies, by providing other strategies as arguments, exploiting higher-order functions in the language.

Skeletons: A number of skeletons are implemented as higher-order functions e.g. parallel map, pipeline and divide&conquer. Parallel map is the most common one which specifies data-oriented parallelism over a list.

```
-- parallel map definition
parMap strat f xs = map f xs `using` parList strat

xs = parMap rdeepseq f [10..25]
```

In the version above, `parMap` exposes the maximal parallelism, creating a spark for each item to be evaluated in parallel. In Section 5 we will discuss several techniques for improving parallel performance by generating fewer, more coarse-grained threads.

3.3. ParMonad

The *ParMonad* offers a new parallel programming model which is implemented entirely as a Haskell library [36]. Programming in the *ParMonad* looks a lot like programming in Concurrent Haskell but it preserves determinism and is side-effect-free. `Par` is simply a type declared as a monad. `IVars` are used for communication, an implementation of the I-Structures, a concept from the pH and Id languages [40]. The basic operations use an explicit approach to specify parallel computations. Parallelism is introduced using `fork` which creates a parallel task. Tasks are scheduled using an optimised parallel scheduler among threads. The computation in the monad is extracted using `runPar`. The communication constructs include the following functions:

- `new` to create a new `IVar`.
- `put` to place some value in the `IVar`. It is executed once per `IVar` otherwise an error occurs.
- `get` to retrieve the value from the `IVar`. It is a blocking operation and waits until something is put into the `IVar`.

The derived `spawn` function hides the explicit `put` and `get` operations and therefore ensures that each `IVar` created is only ever put into once. This raises the level of abstraction provided by this model.

```
runPar :: Par a -> a
fork :: Par () -> Par ()
spawn :: NFData a => Par a -> Par (IVar a)

-- communication
data IVar a
new :: Par (IVar a)
put :: NFData a => IVar a -> a -> Par ()
get :: IVar a -> Par a
```

As the library is fairly recent, it has a limited number of higher-level function abstractions. The most obvious being a parallel map implementation, `parMap`. Work on more abstractions is in progress.

3.4. Eden

Eden extends Haskell by providing constructs for parallel *process definition*: abstracting a function that takes an argument and produces a value into a process with input and output that correspond to the function argument and result respectively; and *process instantiation*: evaluating the process in parallel [32].

```
-- process definition
process::(Trans a,Trans b) => (a->b) -> Process a b
-- process instantiation
(#)::(Trans a,Trans b) => Process a b -> a -> b
```


DOI: 10.1002/cpe

Building on these coordination constructs, a parallel function application operator and eager process creation function are derived. `spawn` is only denotationally specified, ignoring demand control.

```
-- parallel function application
($#)::(Trans a,Trans b) => (a->b) -> a -> b
f $# x = process f # x

-- eager process creation
spawn::(Trans a,Trans b) => [Process a b] -> [a] -> [b]
spawn = zipWith (#)
```

The parallel runtime system distributes the processes to the available processors. Since *Eden* has been designed for a distributed-memory model, processes communicate messages to each other to provide input and retrieve output. All this synchronisation and coordination is handled implicitly by the runtime-system. The programmer does not need to worry about low-level send and receive between parallel processes, and only uses process abstraction or skeletons built on top of these. *Eden* processes produce output eagerly with the argument to the process being evaluated locally in the parent process before sending. Lists are handled as streams and are sent element-by-element. This can cause significant overhead and techniques to avoid element-wise streaming are used.

***EdenSkel*:** *Eden* provides a rich set of higher-order functions that abstract common parallel patterns in its skeleton library *EdenSkel*. For instance, an implementation of parallel map uses `spawn` to eagerly instantiate a list of process abstractions.

```
-- parallel map definition in Eden
parMap f = spawn (repeat (process f))
```

`parMap` creates a process for each list element and this often results in far too many processes in comparison to the number of processing elements available.

Farm process: The farm process skeleton adapts the number of processes to the number of available processing elements (given by `noPe`). The input list is grouped into *noPe* sublists and then a process is created for each sublist instead of each individual element. The farm process rewritten below to provide a simpler interface and familiar name to the programmer specifies `unshuffle` as the distribution function and `shuffle` as the combination function.

```
parMapFarm f = shuffle . (parMap (map f)) . (unshuffle noPe)
```

`parMapFarm` creates (*noPe*+1) processes in total with *noPe* farm processes and 1 main process which means that one machine will be allocated two processes. A slight variation to this, `parMapFarmMinus`, where *noPe-1* processes are created so each processor gets exactly one process.

Chunking input stream: The farm process reduces the number of processes but does not have any effect on the messages exchanged between the processes. Each element of the list is sent as a single message by default. To improve process communication, the number of messages is reduced using a chunking policy. `parMapFarmChunk` is defined as a new function which decomposes the input list into chunks of a specified size e.g. 1000 then creates the farm processes and distributes the chunks to them. This reduces communication overhead.

```
parMapFarmChunk f xs = concat (parMapFarm (map f) (chunk size xs))
```

Offline processes: Another skeleton that is available modifies the communication behaviour: rather than evaluating the input by the parent process, the unevaluated data is sent (which is typically much smaller) and evaluated lazily by the new process. This reduces the combined effort of the main process having to completely evaluate all input to the farm processes.

DOI: 10.1002/cpe

```
-- x is strictly reduced and sent to child process
f $# x

-- parameter passing: input serialised and sent to remote PE
(\() -> f x) $# ()
```

3.5. Summary

In summary, all three models provide high-level constructs which can be used to get initial parallelism from the sequential algorithm. Both *GpH* and *Eden* represent semi-explicit parallelism, without an explicit notion of threads. Both implementations delegate the coordination of the parallelism to a sophisticated runtime system, in the case of *Eden* based on a distributed memory model, in the case of *GpH* based on a (virtual) shared memory model. Under closer examination, *Eden* can be considered slightly more explicit since process instantiation mandates the generation of parallelism, whereas all parallelism in *GpH* is advisory and may be discarded by the runtime-system. In contrast, *ParMonad* is explicit in creating parallel threads, using `spawn`, and it is up to the programmer to coordinate the parallel threads, using `IVars` on a physical shared memory system.

Skeletons: In order to raise the level of abstraction further, all three languages provide skeleton libraries. The most advanced of these libraries is the *EdenSkel* library. It uses internal primitives of the *Eden* implementation, to realise sophisticated topology skeletons, such as rings and tori, as well as optimised, high-level algorithmic skeletons for common patterns such as data parallelism, divide-and-conquer etc. Based on a long history of developed and comparative applications to a range of symbolic applications [30], it represents the best tuned skeleton library of these. The abstraction provided for *GpH* are evaluation strategies, which have proven to be very flexible and easily composable in describing patterns of parallelism [35]. They provide the least intrusive way of specifying parallelism, achieving the clean separation between coordination and computation. *ParMonad* is the youngest of these systems, and thus, it comes with only a small set of skeletons, most notably ones for data-parallelism. Being the most explicit of these languages, it provides the highest level of control to the skeleton programmer. This is reflected in a carefully tuned, work-inlining scheduler, that aims to minimise the parallelism overhead in massively parallel programs.

Runtime-system: The most significant difference between these three models is on the runtime-system level. *GpH* implements parallelism on top of a (virtual) shared-memory model. It is implemented very efficiently on physical shared-memory systems, via the GHC-SMP runtime-system, which is part of the main release of GHC. It is also implemented on distributed memory architectures, via the GHC-GUM runtime-system and explicit message passing. *Eden* has been designed for distributed memory architectures from the start but can also be used on shared memory machines by exploiting an optimised implementation of MPI or custom shared memory implementation of internal communication. This generality of design, however, bears the danger of runtime overhead due to duplication of data in different memory locations. The GHC-Eden runtime system is the only stable release of distributed memory parallelism for Haskell at the moment. The implementation of *ParMonad* supports only physical shared memory systems. In contrast to *GpH* and *Eden*, it does not require any runtime-system extension and is implemented entirely as a library. One advantage of this design is the decoupling from any internal changes to the GHC compile chain or runtime-system. As of the time of writing, the latest official GHC-based releases are 7.4 for GHC-SMP and 6.12.3 for GHC-Eden. A release candidate for GHC-Eden 7.4 and an unstable version of GHC-GUM 6.12.3 exist.

Because of the different advantages for the individual systems, a combination of some or all of them is desirable in particular for heterogeneous, hierarchical networks of multi-cores. The development of such a merged system is currently being pursued by several research groups.

DOI: 10.1002/cpe

4. SEQUENTIAL IMPLEMENTATION

In this section, we discuss the sequential implementation of the two algorithms. Starting with a naive initial implementation, we perform several sequential tuning steps in order to produce optimised versions of both all-pairs and Barnes-Hut algorithms. We consider the problem in three dimensional space. The runtimes exclude generation of input and show the main computation of the algorithms only.

4.1. All-Pairs

The all-pairs program proceeds by comparing each body with the rest in order to calculate the accelerations induced by the other bodies. The accelerations are then deducted from the body's initial velocity and the body is finally moved by updating its position. Two algebraic data types are defined to represent a body and the acceleration.

```
data Body   -- body type def
  = Body {  x::Double,  y::Double,  z::Double,
           vx::Double, vy::Double, vz::Double,
            m::Double }

data Accel  -- acceleration type def
  = Accel { ax::Double, ay::Double, az::Double }
```

The main part of the program is implemented using two map functions. The top-level map function applies the composite function `(updatePos . updateVel)` to each body in the list `bs`. The main computation happens in the `updateVel` function, which has another map function to calculate the accelerations against all the bodies. The fold function deducts the accelerations which gives the updated velocity. The code below shows the computation for one iteration.

```
doSteps::Int->[Body]->[Body]
doSteps 0 bs = bs
doSteps s bs = doSteps (s-1) new_bs
  where
    new_bs = map (updatePos . updateVel) bs

    updatePos (Body x y z vx vy vz m) = Body (x+timeStep*vx) (y+timeStep*vy)
        (z+timeStep*vz) vx vy vz m

    updateVel b = foldl deductChange b (map (accel b) bs)

    deductChange (Body x y z vx vy vz m) (Accel ax ay az) = Body x y z (vx-ax
        ) (vy-ay) (vz-az) m

    accel (Body ix iy iz ivx ivy ivz imass) (Body jx jy jz jvx jvy jvz jmass)
         = Accel (dx*jmass*mag) (dy*jm*mag) (dz*jm*mag)
      where
        mag = timeStep / (dSquared * distance)
        distance = sqrt (dSquared)
        dSquared = dx*dx + dy*dy + dz*dz + eps
        dx = ix - jx
        dy = iy - jy
        dz = iz - jz
```

4.2. Barnes-Hut

The Barnes-Hut implementation is more complicated and based on the 2D version of the algorithm from [7] and [42] that focus on nested-data parallelism.

The `Body` and `Accel` types from the all-pairs version remain unchanged. Three new data types are introduced to represent

1. `BHTree` the Barnes-Hut tree structure;
2. `Bbox` the bounding box representing a region in 3D space, and
3. `Centroid` the centroid of a region.

DOI: 10.1002/cpe

The `BHTree` data-structure implements a rose tree, with an arbitrary number of sub-trees, represented by a list. In our case of modelling a 3D space, this will not be more than 8 children per node, thus an oct-tree. The node consists of the size of a region, the centre of mass, total mass and sub-trees.

```
data BHTree
  = BHT { size::Double, centerx::Double, centery::Double,
          centerz::Double, totalmass::Double, subTrees::[BHTree] }

data Bbox
  = Bbox { minx::Double, miny::Double, minz::Double,
           maxx::Double, maxy::Double, maxz::Double }

data Centroid
  = Centroid { cx::Double, cy::Double, cz::Double, cm::Double }
```

The algorithm proceeds in two main phases:

tree construction first an oct-tree is constructed from the list of bodies (`buildTree`);

force calculation then all forces between bodies are calculated to update the velocities (`updateVel`), and their positions are updated (`updatePos`).

Tree construction: This phase constructs an oct-tree from the list of bodies, shown in the `buildTree` function below.

First, the bounding box representing the lower and upper coordinates of the region containing all the points is found (`findBounds`) and the size of the region calculated.

The centre of mass (`cx`, `cy`, `cz`) and total mass (`cm`) are calculated and stored at the root node of the tree to represent the whole space.

The centre of mass (R) and total mass (M) of a list of bodies (m) are given by:

$$M = \sum_{i=1}^{n} m_i \quad \text{where } n \text{ is number of bodies} \tag{1}$$

$$R = \frac{1}{M} \sum_{i=1}^{n} m_i \times r_i \tag{2}$$

The bounding box is used to subdivide the bodies into 8 smaller regions (`splitPoints`) and then the centre of mass and total mass of the bodies contained in each region are computed in the same way and stored in the children nodes. The process continues until a region has no body in it — `buildTree` is essentially a recursive function. The actual bodies are not stored in the tree structure as in some implementation as the centre and total mass are calculated in the tree construction phase.

```
doSteps 0 bs = bs
doSteps s bs = doSteps (s-1) new_bs
  where
    bbox = findBounds bs
    tree = buildTree (bbox,bs)
    new_bs = map (updatePos . updateVel) bs

-- build the Barnes-Hut tree
buildTree :: (Bbox,[Body])->BHTree
buildTree (bb,bs) = BHT size cx cy cz cm subTrees
  where
    subTrees =          if bs <= 1 then []
               else map buildTree (splitPoints bb bs)
    Centroid cx cy cz cm = calcCentroid bs
    size = calcBoxSize bs

findBounds::[Body]->Bbox
-- split bodies into subregions
splitPoints::Bbox->[Body]->[(Bbox,[Body])]
-- calculate the centroid of points
calcCentroid::[Body]->Centroid
```


DOI: 10.1002/cpe

```
-- size of the region
calcBoxSize::Bbox->Double
```

Force calculation: In this phase, the acceleration due to each body is computed by traversing the tree, shown in the `calcAccel` function below.

The traversal along any path is stopped as soon as a node is too far away to make a significant contribution to the overall force (`isFar`). This is determined by dividing the size *s* of the region by the distance *d* between the body and the node centre of mass coordinates. If the ratio $\frac{s}{d}$ is less than a certain threshold *t* where *0 <t <1* then the centroid is used as approximation as the point is far from the region. Setting *t* to zero degenerates to a brute force version while increasing the value improves performance at the expense of losing some precision.

The `accel` function here differs from the one in the all-pairs version in that instead of calculating the acceleration between two *bodies*, it uses the *centroid* of a region and a body. The `updateVel` function deducts the net acceleration due to each body.

```
updatePos (Body x y z vx vy vz m) = ... -- same as allpairs

updateVel b@(Body x y z vx vy vz m) = Body x y z (vx-ax) (vy-ay) (vz-az) m
  where
    Accel ax ay az = calcAccel b tree

calcAccel::Body->BHTree->Accel
calcAccel b tree@(BHT _ _ _ _ subtrees)
  | null subtrees     = accel tree b
  | isFar tree b      = accel tree b
  | otherwise = foldl addAccel (Accel 0 0 0) (map (calcAccel b) subtrees)
    where
      addAccel (Accel ax1 ay1 az1) (Accel ax2 ay2 az2) = Accel (ax1+ax2) (ay1
          +ay2) (az1+az2)

accel::BHTree->Body->Accel
isFar::BHTree->Body->Bool
```

4.3. Sequential tuning

A number of sequential optimisation techniques are used for improving the runtime, heap usage and fixing issues like stack overflow.

Optimisations: The following general optimisations apply to both algorithms:

Reducing stack consumption: The naive implementation of the algorithm suffers from an excessive stack consumption if huge number of bodies are used. Space profiling helps to understand the memory usage of each algorithm and to find any space leak, which led to a stackoverflow in the initial implementation. To fix the problem and improve general performance of the sequential algorithm, the following steps are taken:

- *Tail recursion:*
  Two general, well known techniques for reducing stack consumption in functional languages are to make the function tail recursive and to use accumulating parameters.
- *Strictness:*
  More specifically to a lazily evaluated language such as Haskell, strictness annotations can be added, where delaying evaluation is not necessary thus avoid unnecessary thunking of computations. This is achieved in a number of ways e.g. using the `pseq` primitive, strict application function (`$!`) or strict annotation (`!`) from the BangPatterns extension.
- *Types Definition:*
  Initially, type synonyms were used e.g. to represent position, velocity and mass as triple tuple: `type Pos = (Double,Double,Double)`. Through the use of more advanced data

DOI: 10.1002/cpe

types provided in Haskell e.g. with strict data fields, space leaks can be avoided and this improves performance considerably with high input sizes as we will see shortly.

Reducing heap consumption: While the GHC compiler performs numerous automatic optimisations, more opportunities can be exposed by specific code changes, in particular in fusing function applications where necessary. This removes any intermediate data structures that could potentially decrease performance.

```
-- A trivial example
map updatePos (map updateVel bs)

-- rewritten using function composition
map (updatePos . updateVel) bs
```

The composed version using a single `map` is easier to read and also does not depend on compiler optimisation.

The use of `foldr` in conjunction with list comprehension (`foldr`/`build`) also eliminates the intermediate lists produced by build and consumed by foldr. This was used in the Barnes-Hut tuning.

Quantifying sequential tuning: Table I shows the results of each step of tuning the sequential all-pairs and Barnes-Hut algorithms.

All-Pairs

- *Version 1:* the initial version of the all-pairs program uses type synonyms/tuples to represent position, velocity, mass and acceleration. Type synonyms are not new data types but are used mainly for code clarity. For example, it is easy to read that a position consists of 3 doubles, thus representing it using a tuple.

```
type Pos   = (Double, Double, Double)
type Vel   = (Double, Double, Double)
type Mass  = Double
type Accel = (Double, Double, Double)
```

- *Version 2:* change type synonyms to data types. This causes the initial runtime to go up by 60%. Using type synonym is usually more efficient as it incurs one less indirection than data type. However, data types are more powerful and can be used with further optimisation as we will see in the following versions. Data type necessitates deriving appropriate typeclasses e.g. `Eq` if we need to be able to compare them.

```
data Pos   = Pos Double Double Double
data Vel   = Vel Double Double Double
data Mass  = Mass Double
data Accel = Accel Double Double Double
```

- *Version 3:* add strictness to avoid unnecessary thunking of computation. For example, return type of the `accel` function below is `Accel`. By default the data fields of `Accel` are evaluated lazily, explaining why Version 2 takes up a lot of memory space. By making them strict, they are computed eagerly. The clearer way to achieve this is by using the (!) strictness annotation instead of the most explicit `pseq`. As the results show, this step accounts for the main reduction in heap and time by 78% and 96%.

```
-- data fields evaluated lazily
accel bodyi bodyj = Accel (dx*jm*mag) (dy*jm*mag) (dz*jm*mag)
-- add strictness annotation (!)
accel bodyi bodyj = Accel ax ay az
  where
    !ax = (dx*jm*mag)
    !ay = (dy*jm*mag)
    !az = (dz*jm*mag)
```


DOI: 10.1002/cpe

(a) All-Pairs (5000 bodies)

Version	Runtime (s)	Max Resi (KB)	Heap Alloc (%)
allpairs1	39.47	1976	+0.0
allpairs2	63.24	3533	+30.9
allpairs3	2.54	1273	-77.9
allpairs4	2.15	726	-28.9
allpairs5	2.14	726	-0.0
allpairs-final	1.94	69	-0.0

(b) Barnes-Hut (80000 bodies)

Version	Runtime (s)	Max Resi (KB)	Heap Alloc (%)
bh1	41.90	28	+0.0%
bh-final	33.37	27	-88.6%

Table I. Sequential tuning

- *Version 4:* Strict data fields. Making the data fields strict removes the need for the previous strictness annotation added inside the function. In addition to this, use of the `UNPACK` pragma indicates to the compiler that it should unpack the contents of the constructor field into the constructor itself, removing a level of indirection.

```
data Pos = Pos {-# UNPACK #-} !Double {-# UNPACK #-} !Double {-# UNPACK #
    -} !Double
```

- *Version 5:* use appropriate higher order functions from the standard libraries e.g. use `foldl'` instead of `foldl` for its strictness properties.
- *Final version:* use single `Body` data type. This removes the need to deal with many different data types, makes the program more compact, and as a result, reduces the runtime by 10%. The maximum residency sees an important decrease.

```
data Body
       = Body
       { x  :: {-# UNPACK #-} !Double  -- pos of x
       , y  :: {-# UNPACK #-} !Double  -- pos of y
       , z  :: {-# UNPACK #-} !Double  -- pos of z
       , vx :: {-# UNPACK #-} !Double  -- vel of x
       , vy :: {-# UNPACK #-} !Double  -- vel of y
       , vz :: {-# UNPACK #-} !Double  -- vel of z
       , m  :: {-# UNPACK #-} !Double } -- mass
```

Barnes-Hut In addition to the optimisations applied to all-pairs, the main tuning for Barnes-Hut sequential performance is the use of `foldr/build`.

- *Version 1:* The first Barnes-Hut version includes all all-pairs optimisations detailed earlier. This gives a good initial runtime.
- *Version 2:* Use `foldr/build` in the `calcAccel` which eliminates intermediate lists produced by `build` and consumed by `foldr`.

```
-- before
foldl' addAccel (Accel 0 0 0) [calcAccel st b | st <- subtrees]
-- after
foldr addAccel (Accel 0 0 0) [calcAccel st b | st <- subtrees]
```

Compiler optimisation: The sequential runtimes up to now are based on fully optimised code and used 5000 bodies. GHC's automatic optimisation already manages to improve time and heap performance significantly. Table II shows that without enabling compiler optimisation,

DOI: 10.1002/cpe

		allpairs1		allpairs-final	
Optimisation		Runtime (s)	Max Resi (KB)	Runtime (s)	Max Resi (KB)
`-O0`	(disable optimisations)	48.26	1667000	18.58	71
`-O1`	(standard optimisations)	3.76	1094	0.31	69
`-O2`	(full optimisations)	3.72	1117	0.31	69

Table II. Effect of compiler optimisation (2000 bodies)

the runtime may be very high, and actually too high that we use 2000 bodies here to show the difference in optimisation. This table shows, that GHC's aggressive optimisation machinery manages to automatically improve performance of the final version by a factor of 13.0, relative to the unoptimised version, but doesn't find many more optimisation sources when going to full optimisation. Algorithmic optimisation, represented by all the sequential tuning steps as discussed in this section, gives a performance gain of a factor of 12.0 (combined with full optimisations). Both taken together account for a sequential speedup of 155.7.

Baseline comparison: As comparison basis of our sequential all-pairs algorithm, we use the highly optimised nbody implementations from the language shootout website [†]. We first compare the performance of our all-pairs version with the corresponding Haskell implementation. The two programs implement the same algorithm. However, we note that our implementation is by a factor of 1.5 slower: 1.97 sec compared to 1.25 sec using 5000 bodies. The shootout Haskell version is highly optimised by programmers, who are both experts in Haskell and in the GHC compiler, using inplace update operations organised through monadic code and unsafe operations like `unsafePerformIO`. The disadvantage of this approach is that it introduces sequentialisation in the code as part of the optimisation and therefore loses a lot of potential for parallelism.

Comparing the performance of the Haskell version with other languages, the shootout Haskell version is by a factor of 2.3 slower than the fastest, Fortran version. Therefore, we observe as a baseline comparison a sequential overhead of a factor of 3.4 compared to the fastest available all-pairs algorithm.

By remaining faithful to a purely functional programming model, our implementation provides opportunities for parallelism, that do not exist in the lower-level implementations and have to be refactored in a time-consuming and error prone parallelisation methodology. By exploiting high-level parallelism, we can compensate for the sequential overhead, using a fairly small number of processors, and achieve high scalability of our code, through the usage of more massively parallel hardware. In particular, the final parallel program will not be tied to one particular class of architectures, nor to a certain number of processors.

5. PARALLEL IMPLEMENTATION

The parallel implementation is based on high-level constructs provided in each of the programming models. This should not necessitate major changes to the sequential algorithms to get initial parallel versions of both algorithms. Still, the small set of available constructs makes it possible to improve runtime and speedup in a parallel tuning phase of program development.

Time profiling: Identifying the source of parallelism is the first step in writing the parallel algorithms. Time profiling points out the "big eaters," that is, functions that take the largest percentage of the total time. Listing 1 shows the time and allocation profiling for both algorithms.

[†]The Computer Language Benchmarks Game [1] `http://shootout.alioth.debian.org/`

DOI: 10.1002/cpe

Listing 1: Time and Allocation Profiling Report

```
                                                  individual     inherited
COST CENTRE          MODULE  entries  %time %alloc   %time %alloc

-- All-Pairs (2000 bodies, 1 iteration)

doSteps              Main          1   0.0   0.0   100.0   99.8
 updatePos           Main       2000   0.0   0.0     0.0    0.0
 updateVel           Main       2000   3.1   1.9    99.9   99.8
  accel              Main    4000000  23.5  37.2    94.5   93.3
  deductChange       Main    4000000   2.4   4.5     2.4    4.5


-- Barnes-Hut (8000 bodies, 1 iteration)

doSteps              Main          1   0.0   0.0    99.9   99.8
 findBounds          Main          1   0.0   0.0     0.0    0.0
 buildTree           Main      11804   0.0   0.0     0.2    0.2
  splitPoints        Main       3804   0.0   0.0     0.1    0.2
  calcCentroid       Main      11804   0.0   0.0     0.0    0.0
 updatePos           Main       8000   0.0   0.0     0.0    0.1
 updateVel           Main       8000   0.0   0.0    99.7   99.5
  calcAccel          Main   14893157   4.8   7.1    99.7   99.5
   accel             Main   12193706  12.9  16.1    65.1   63.3
   isFar             Main   10247211   7.1   9.2    29.7   29.1
```

In both algorithms, the top-level `doSteps` function performs the iterations and inherits the largest percentage of time. The main source of parallelism arises from the update velocity function `updateVel` which is used as the function argument of a map operation in both all-pairs and Barnes-Hut. It accounts for almost 100% of the inherited overall time. As it is used in a map, it presents data-oriented parallelism.

```
new_bs = map (updatePos . updateVel) bs
```

While the tree construction phase can normally be done in parallel, the time profile indicates that `buildTree` accounts for less than 1% of the time in the Barnes-Hut algorithm. Parallelising it may not cause any significant improvement but could, on the other hand, create overheads. However, with the same number of bodies as used for all-pairs, the time percentage spent in `buildTree` reaches approximately 12%. This is explained by lesser computation involved in the acceleration calculation phase and thus better distribution of the time between the two phases. Also, depending on the distance threshold used to determine when to consider a body far enough, the time in `updateVel` can varies significantly. For instance, if the distance threshold is high (closer to 1), the traversal is very fast, as a result of little computation involved. This is not very good for parallelism as the cost of creating parallelism may be higher than the actual computation. Smaller thresholds for example 0.1 runs slower, while 0 degenerates to pair-wise comparison. Ideally, we use 0.25 which gives a reasonable approximation, accuracy and speed.

Both all-pairs and Barnes-Hut implementations turn out to be data-parallel algorithms as the same operation is applied to the list of bodies in order to update them. All of the three models provide some sort of parallel map for data parallelism.

5.1. GpH

All-Pairs: Initial parallelism is obtained by replacing `map` with `parMap` such that each body velocity is computed in parallel.

```
new_bs = parMap rdeepseq (updatePos . updateVel) bs

-- equivalent to
new_bs = map (updatePos . updateVel) bs `using` parList rdeepseq
```

The composition in the `map` function argument can be turned into a pipeline using the parallel `.||` combinator that arranges for a parallel evaluation of both functions being combined. In this case, the result of `updateVel` is evaluated in parallel with the application of the first

DOI: 10.1002/cpe

function. However, given that the `updatePos` function does negligible computation as opposed to `updateVel`, this is not a useful source of parallelism and therefore not considered any further. It demonstrates, however, that this programming model makes it easy to compose different sources of parallelism, and to prototype alternative parallelisations, without having to restructure the existing code in a fundamental way.

The performance results from this naive version are disappointing: in the best case we observe a speedup of 1.4 on 4 processors, but for 8 processors we actually encounter a slow-down. The reason for this poor performance is considerable overhead associated with generating a thread for every list element, potentially 16000 in total. While the generation of sparks is cheap, it amounts to adding a pointer to a queue, the generation of a thread requires the allocation and initialisation of a thread state object (TSO), which among other data contains the stack used by the thread. In this case, the computation performed by one thread, namely updating the velocity and position of one body, is too small in comparison with the overhead for TSO initialisation and for scheduling the available threads. The following statistics summarises the execution on 2 cores: in total 16000 sparks are created, one for each list element, and of these 8192 are converted into threads. This lower number is due to the limited size of the spark pool, which is 4k by default. Since the nature of the parallelism is data-parallel, no work can be subsumed by a sibling-thread, and thus lazy task creation is not effective in automatically increasing thread granularities.

Listing 2: Global statistics of a parallel run on 2 cores

```
./allpairs 16000 1 +RTS -N2 -s
56026.00329381344
54897.906546913
time taken: 16.76s
  31,145,652,016 bytes allocated in the heap
      27,366,360 bytes copied during GC
       2,999,520 bytes maximum residency (5 sample(s))
         517,760 bytes maximum slop
              10 MB total memory in use (0 MB lost due to fragmentation)

  Generation 0: 44953 collections, 44952 parallel,  2.70s,  1.22s elapsed
  Generation 1:     5 collections,     5 parallel,  0.05s,  0.03s elapsed

  Parallel GC work balance: 1.12 (3256443 / 2904329, ideal 2)

                        MUT time (elapsed)       GC time  (elapsed)
  Task  0 (worker) :    4.90s    ( 15.63s)       1.51s    (  0.82s)
  Task  1 (worker) :    6.21s    ( 15.63s)       0.52s    (  0.08s)
  Task  2 (bound)  :    9.96s    ( 15.63s)       0.72s    (  0.36s)
  Task  3 (worker) :    0.00s    ( 15.63s)       0.00s    (  0.00s)

  SPARKS: 16000 (8192 converted, 0 pruned)

  INIT  time    0.00s  (  0.01s elapsed)
  MUT   time   21.07s  ( 15.63s elapsed)
  GC    time    2.75s  (  1.25s elapsed)
  EXIT  time    0.00s  (  0.00s elapsed)
  Total time   23.83s  ( 16.89s elapsed)

  %GC time      11.6%  (7.4% elapsed)

  Alloc rate    1,477,931,568 bytes per MUT second

  Productivity  88.4% of total user, 124.8% of total elapsed
```

In order to tune the parallel performance, we control the number of sparks created by grouping elements into larger units, into “chunks.” Instead of creating a spark for each element in the list, the list is broken down into chunks and a spark is created for each chunk, thus significantly reducing the thread creation overhead. The number of chunks is determined by the number of available processors. Having too few chunks may result in some processors not getting enough work while too many chunks create excessive overhead.

DOI: 10.1002/cpe

As often in data-parallel programs, a careful balance between low thread management overhead and massive parallelism is crucial. There is no dynamic parallelism here, i.e. all parallelism is generated at the beginning. Thus a low, fixed number of chunks is likely to be the best choice for performance. Each processor does not necessarily get the same number of chunks. Using more chunks retains more flexibility for the runtime system, because a faster or more lightly loaded processor can pick-up new work after having finished its initial work allocation.

We now consider three ways to introduce chunking (or clustering) into the algorithm (the language-level differences between these approaches are discussed in more detail in [35]).

Explicit Chunking: The most obvious way of performing chunking, is to explicitly apply functions performing chunking before and de-chunking after the data-parallel core of the application (see below). Explicit chunking will be used in the *ParMonad* version, and its performance discussed in Section 5.2.

```
s = 1000 -- chunk size
new_bs = concat (map (map (updatePos . updateVel)) (chunk s bs) `using`
    parList rdeepseq)
```

Used directly in the application code, this technique obfuscates the computational core of the application, and introduces an intermediate data structure that is only needed in order to increase thread granularity.

Strategic Chunking: Another skeleton-based approach to introducing chunking is to modify the definition of the strategy and to encode chunking additionally to the specification of parallelism inside this skeleton. Thus, we change `parList` to `parListChunk`, which takes an additional argument, specifying the chunk size. The `parListChunk` strategy applies the given strategy, in this case `rdeepseq`, to each chunk. This achieves a cleaner separation of computation and coordination, leaving the core code unchanged, and hiding the intermediate data structure in a custom strategy. However, this strategy is now fixed to one parallel pattern and one way of chunking.

```
s = (length bs) `quot` (numCapabilities * 4) -- 4 chunks/PE
new_bs = map (updatePos . updateVel) bs `using` parListChunk s rdeepseq
```

Table X (a) in Appendix 10 shows the runtime and speedup results for different number of chunks per processor using `parListChunk` strategy. While generating exactly 1 chunk per processor might intuitively seem to be the best choice, it is also the least flexible one, because it deprives the runtime-system from distributing parallelism in the case where one processor suffers from a high external load. Therefore, a small number of chunks greater than 1 is usually a good choice. In this case, the right balance is to have approximately 4 chunks per processor.

Implicit Clustering: A more compositional way to introduce chunking is to delegate it to an instance of a new `Cluster` class, with functions for performing clustering and unclustering (or flattening). We can use available abstractions of performing an operation on each element of a cluster (`lift`) and of flattening the resulting data structure (`decluster`). Thus, to define an instance of this class the programmer only needs to define `cluster` in such a way, that the specified proof obligation is fulfilled e.g. an instance for lists as given below requires us only to define `cluster`.

```
class (Traversable c, Monoid a) => Cluster a c where
  cluster     :: Int -> a -> c a
  decluster   :: c a -> a
  lift        :: (a -> b) -> c a -> c b

  lift = fmap       -- c is a Functor, via Traversable
  decluster = fold  -- c is Foldable,  via Traversable
  -- we require: decluster . cluster n == id

instance Cluster [a] [] where
  cluster = chunk
```


DOI: 10.1002/cpe

	nochunk		parListChunk		evalCluster	
no. PE	Runtime (s)	Speedup	Runtime (s)	Speedup	Runtime (s)	Speedup
Seq.	20.04	1.00	20.06	1.00	20.02	1.00
1	20.64	0.97	22.10	0.91	19.71	1.02
2	16.76	1.20	11.33	1.77	10.93	1.83
4	13.89	1.44	5.97	3.36	5.83	3.43
8	15.64	1.28	3.29	6.10	3.28	6.10

Table III. All-Pairs: GpH runtime and speedup

Based on this class definition, we can then separately define an `evalCluster` strategy, which uses these functions before and after applying its argument strategy to each cluster, thus separating the definition of parallelism from any form of clustering.

```
evalCluster :: Cluster c => Int->Strategy a->Strategy a
evalCluster n s x = return (decluster (cluster n x `using` cs))
  where cs = evalTraversable s :: Strategy c
```

Using this approach, we can add clustering to the basic data-parallel strategy, without changing the original strategy at all. We simply replace `evalList (rpar `dot` rdeepseq)`, which is the definition of `parList`, by `evalCluster s (rpar `dot` rdeepseq)`. In short, the compositionality of this style of programming allows to specify a parallel strategy combined with a clustering strategy. This provides more flexibility in aggregating collections in ways that cannot be expressed using only strategies.

In summary, the code below shows the use cases for all three clustering techniques:

```
-- explicit clustering
concat (map (map f) (chunk s bs) `using` parList rdeepseq)
-- strategic clustering
map f xs `using` parListChunk s rdeepseq
-- combining parallel and clustering strategies
map f xs `using` evalCluster s (rpar `dot` rdeepseq)
```

Table III summarises the parallel results of using: no chunk, parListChunk, and evalCluster with the last two using 4 chunks per PE. Most notably, the implicit evalCluster version achieves the same performance as the strategic parListChunk. Thus, using this more compositional version, that makes it easy to introduce and modify clustering strategies separately from specifying parallelism over the data structure, does not incur a significant performance penalty.

Barnes-Hut: Sequential profiling of the Barnes-Hut algorithm identifies the same function, `updateVel`, as the main eater of compute time. As the call count for this function shows, this is due to the iterative use in the top level map. Therefore, the Barnes-Hut algorithm is parallelised in the same, data-parallel way as the all-pairs version. Since an abundance of fine-grained parallelism is also a problem in this version, we use the same form of chunking in order to tune the parallel performance.

In this version a natural parallel phase is `buildTree`, where sub-trees can be constructed in parallel. But as the profiling report showed earlier, it does not account for a big percentage of the overall time, and therefore benefits from parallelising this stage are limited. However, it is cheap to mark the stage as parallel computation, and whether to take the spark for parallel execution is up to the runtime-system.

Another generic optimisation that is applied in the `buildTree` function is "thresholding." By adding an explicit argument to the function that represents the current level of the tree, the generation of parallelism can be restricted to just the top levels. This makes sure there are not too many parallel threads for the tree construction otherwise it would cause overheads with large number of bodies.

As expected, Table IV shows that parallel `buildTree` does not improve performance significantly but it does not cause additional cost either. The main observation though is that the algorithm does not achieve as good speedup as the all-pairs algorithm. This is expected as all parallel

DOI: 10.1002/cpe

	Top-level map only		Parallel buildTree	
no. PE	Runtime (s)	Speedup	Runtime (s)	Speedup
Seq.	33.31	1.00	33.31	1.00
1	35.77	0.93	35.77	0.93
2	21.50	1.55	21.61	1.54
4	11.00	3.03	10.77	3.09
8	6.77	4.92	6.11	5.45

Table IV. Barnes-Hut: GpH runtime and speedup

tasks in the all-pairs algorithm have the same amount of computation to perform i.e. the parallelism is regular; whereas the acceleration calculation steps in the Barnes Hut algorithm varies for each body depending on its location. Some bodies require traversing deeper inside the tree to calculate the net acceleration, while for some, it may not require to do so. For instance, quantifying the irregularity of the computation involved in Barnes-Hut, random generation of 80000 bodies gives an unbalanced tree with minimum tree depth 6 and maximum depth 9.

5.2. ParMonad

All-Pairs: We use the pre-defined parallel map provided in *ParMonad* to add parallelism the same way as we did in *GpH*. In contrast to *GpH*, the parallel computation happens in a monad and therefore the result has to be extracted using `runPar`.

```
new_bs = runPar $ parMap (updatePos . updateVel) bs
```

The initial results in Table X (b), without using chunks, already show good performance: a speedup of 6.17 on 8 cores. The reason for this efficient behaviour is the work-inlining scheduler, which distributes the potential parallel tasks to a number of implicitly created threads and then executes the task within the existing thread. This dramatically reduces the thread creation overhead, at the expense of less flexibility in how to distribute the tasks in the first place. This model is well suited for homogeneous, multi-core architectures and no explicit chunking is needed to improve parallel performance. However, as shown in Table V the use of chunking reduces the maximum residency by 50% from 4822MB to 2417MB for parallel run on 8 cores.

Table X (b) also shows that the number of chunks causes negligible change to the runtime and speedup. However, in order to maintain low memory residency, and to facilitate scalability beyond the number of cores available for these measurements, a chunking policy is preferred.

ParMonad, however, does not come with a pre-defined parallel map function with chunking. So, we use explicit chunking, as discussed above: *s* is the chunk size, and it is adjusted to the number of cores, in the same way as in *GpH* in order to produce an appropriate number of chunks.

```
new_bs = parMapChunk (updatePos . updateVel) s bs

parMapChunk f n xs = concat ( runPar $ parMap (map f) (chunk n xs) )
```

Barnes-Hut: For the Barnes-Hut algorithm we note that chunking causes a noticeable improvement in the speedup from 5.29 to 6.50 on 8 cores (using parallel `buildTree`). This could be due to the fact that a large number of bodies are used in this algorithm and the memory usage is more significant.

	nochunk	chunking
copied during GC (MB)	31527	16171
max residency (MB)	4822	2417

Table V. ParMonad: nochunk vs 4chunks/PE

DOI: 10.1002/cpe

	Top-level map only		Parallel buildTree	
no. PE	Runtime (s)	Speedup	Runtime (s)	Speedup
Seq.	33.39	1.00	33.65	1.00
1	34.49	0.97	33.96	0.99
2	17.72	1.88	17.79	1.89
4	9.21	3.63	8.97	3.75
8	5.91	5.65	5.18	6.50

Table VI. Barnes-Hut: ParMonad runtime and speedup

The large number of bodies used in Barnes-Hut makes the heap usage more significant compared to the all-pairs algorithm. Without chunking, the maximum residency is 83MB and productivity is at 63%. With chunking, residency is 55MB and improved productivity by 10%. This reduced percentage of garbage collection time has an immediate impact on the performance of the parallel program.

Similarly to the *GpH* version, we also try to parallelise the `buildTree` function and the difference is insignificant, as we expected due to `buildTree` not representing a large part of the overall computation (Table VI).

5.3. Eden

All-Pairs: As with the previous two models, we only need a parallel map implementation to get data-oriented parallelism from the algorithm. *Eden* offers several skeletal approaches and in particular has several implementations of parallel map as described earlier. The default `parMap` implementation creates a process for each list element causing far too much overheads in terms of number of processes and messages communicated between them (16001 and 64000 respectively as shown in Appendix Table XI). Observing number of processes and communications is motivation for picking a different strategy, and with it numbers drop significantly and speedup improves.

The farm process skeleton creates the same number of processes as the number of available processing elements. But the message overheads remain. Each list element is communicated as a single message which generates 32048 messages in total. This represents a high number but still the performance is considerably improved compared to the naive parallel map and good speedup is noted. This indicates that process creation overheads is far more important than the number of messages.

Doing further parallel tuning in order to reduce message overheads, we use chunking to break the stream into chunks of size 1000 items which are then sent as one message, thus enabling the process to do more computation at one time rather than having to send and receive messages in between. The chunking reduces the total number of messages communicated in the `parMapFarm` version (farm process) from 32048 to just 80 messages. As a result of this, the runtime and speedup is improved as indicated in Table VII.

The offline farm process, where process input is evaluated by the child process instead of the parent process, causes a small performance improvement compared to the farm process. Sending process input to child processes to be evaluated is intended to reduce the combined time the parent

	farm processes		with stream chunking	
no. PE	Runtime (s)	Speedup	Runtime (s)	Speedup
Seq.	22.13	1.00	22.13	1.00
1	23.67	0.93	22.91	0.97
2	11.91	1.86	11.57	1.91
4	6.08	3.64	5.82	3.80
8	3.41	6.49	3.09	7.16

Table VII. All-Pairs: Eden `parMapFarm` vs. `parMapFarmChunk` runtime and speedup

DOI: 10.1002/cpe

	Top-level map only		Parallel buildTree	
no. PE	Runtime (s)	Speedup	Runtime (s)	Speedup
Seq.	35.34	1.00	35.32	1.00
1	33.11	1.07	33.89	1.04
2	18.41	1.92	18.73	1.89
4	10.62	3.33	11.24	3.14
8	10.37	3.41	10.71	3.30

Table VIII. Barnes-Hut: Eden (parMapOfflineFarmChunk) runtime and speedup

process spent in reducing all inputs. However, in the algorithm, the inputs to farm processes is Body type with strict fields. So there is no much reduction happening after sending it to the child processes.

The measurements using all skeletons are available in Appendix Table X (c) and (d) for all-pairs and Table XI summarises the overheads of each skeleton.

Barnes-Hut: Though the *Eden* all-pairs implementation has given the best performance so far compared to the other two models, the performance for the Barnes-Hut algorithm using *Eden* is not as good as the other models. The speedup is roughly the same on 1 to 4 cores but then there is no further speedup upto 8 cores. The best speedup achieved is using offline process with chunking as seen in Table VIII. This is partially due to the high maximum residency caused by all PEs combined due to the large number of bodies used. Furthermore, this indicates that spark-oriented parallelism in *GpH* and *ParMonad* parallel tasks deal better with dynamic and irregular parallelism. We compare the performance of all implementations in more detail in Section 6.

6. PERFORMANCE EVALUATION

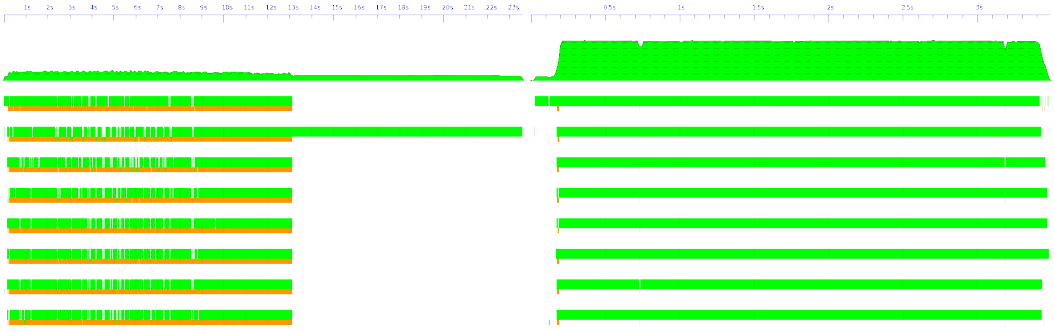
Experimental Setup: The machine used for taking the measurements contains a 64-bit Intel Xeon CPU E5410 2.33 GHz processor with 8 cores, 8GB RAM and 12MB L2 cache. The machine runs Linux and the version of GHC used for *GpH* and *ParMonad* is 7.0.1 while *Eden* uses 6.12.3. A newer version of *Eden* is under development and an attempt to get measurements using it was not successful. Comparison between *GpH*/*ParMonad* and *Eden* is therefore on the basis of speedup.

The input size for all-pairs measurements is 16000 bodies, while for Barnes-Hut, being a more efficient algorithm and able to cope with high number of bodies, we use 80000 bodies. The input sizes ensure the runtimes of one iteration are within a minute for both algorithms. Larger input sizes are used for doing scalability tests and smaller sizes in the sequential optimisation phase. Measurements for the challenge input specification of 1024 bodies and 20 iterations are given at the end of the section, for comparison with other systems. All speedups in the tables and graphs are *absolute*.

Apart from the performance obtained from using the different models for the parallel implementations, other factors are equally important. For example, each model is built around different concepts and the underlying implementation is fairly technical. However, exposing high-level functions with simple interfaces to the casual programmer is an important part of any programming model. All of them do provide similar interfaces familiar to the programmer, for example, parallel map.

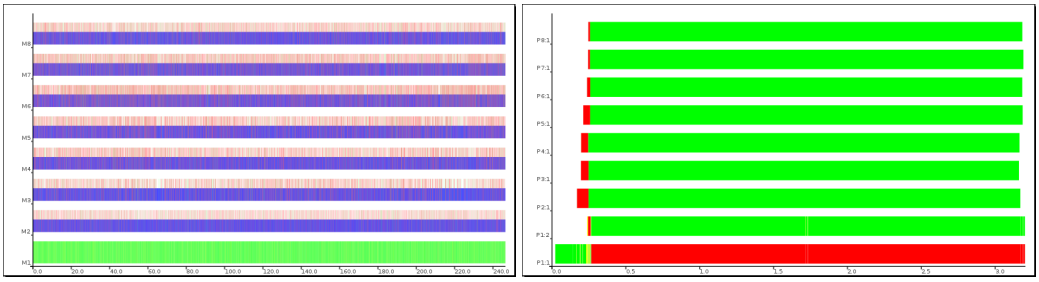
Tuning: While an initial parallel version was easily produced with only a one-line program change, *GpH* required some parallel performance tuning, in particular by using chunks to generate a bounded number of threads of suitable granularity. Selecting a good chunk size required a series of experiments, establishing four threads per processor to be the best balance between massive parallelism and coarse thread granularity. The more irregular nature of the parallelism in the Barnes-Hut version, compared to the naive all-pairs version, diminishes the achieved speedup, but also

DOI: 10.1002/cpe



(a) nochunk (b) 4chunksPerPE

Figure 2. Threadscope of parallel run on 8 cores



(a) parMap (b) parMapFarmChunk

Figure 3. Eden Trace Viewer of parallel run on 8 PE

demonstrates that the runtime-system effectively manages and distributes the available parallelism, without requiring further application code changes from the programmer.

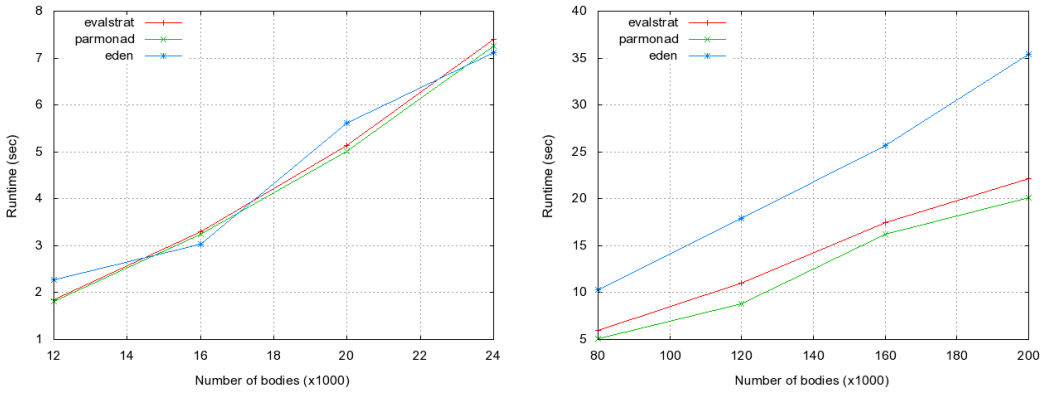
The *ParMonad* version uses a highly tuned, data-parallel map skeleton, and thus can efficiently handle a large number of parallel tasks already in its initial version, eliminating the need for explicit chunking. However, chunking does improve the maximum residency and therefore the scalability of the application.

Eden provides the richest set of skeletons available to implement both versions of the algorithm. In this model, parallelisation amounts to selecting the most suitable skeleton for the main worker function. Ample literature on the advantages and disadvantages of different skeletons helps in making the best decision for a specific application and architecture. For fine tuning the parallel performance, however, an understanding of the process creation and message communication is required to minimise the amount of communication in this distributed memory model.

For any high-level language model, good tool support is crucial in order to understand the concrete dynamic behaviour of a particular algorithm and to tune its performance. Threadscope helps to visualise the work distribution among the available processors for *GpH* and *ParMonad*. Figure 2 shows the work distribution of running the all-pairs *GpH* program on 8 cores before and after parallel tuning using chunking. The upper portion of the graph shows overall activity, measured in terms of the total number of active processors at each time in the execution. The lower portion of the graph shows the activity for each of the processors, with green representing "running" and red representing "idle". Additionally, the time spent in garbage collection is indicated in orange. The number of sparks is given in the runtime statistics for *GpH*. For *ParMonad*, however, the exact number of threads created is not given.

EdenTV (Eden Trace Viewer) gives useful information about processes, their placement, conversations and messages between processes. Figure 3 compares the use of the naive parallel map (`parMap`) against an alternative implementation using farm process and stream chunking (`parMapFarmChunk`). It shows the overheads of too many processes, and consequently messages, being generated in the former. The overheads are eliminated in the tuned version. Each line

DOI: 10.1002/cpe



(a) All-Pairs (b) Barnes-Hut

Figure 4. *Scale-up* graphs of both algorithms for parallel runs on 8 cores

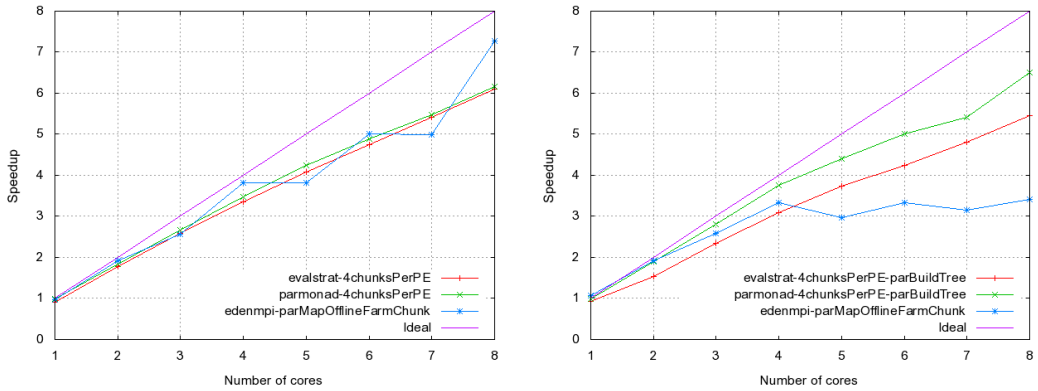
represents the activity on one processor, with green representing "running", while blue represents "waiting for data". While the first trace shows frequent changes between running and waiting states, reflecting the element-by-element transfer of the input data from the master to the workers, the second trace shows much better utilisation as uninterrupted activity once the entire block of input data has been received by a worker. The master remains idle, while the workers produce their results. A related set of skeletons allows a dual usage of the master process as worker in such a case, and can be used to improve performance further.

Scale-up: Both algorithms have $O(N^2)$ asymptotic complexity, with a smaller, tunable factor for the Barnes-Hut version. The optimisations carried out in the sequential tuning phase play an important role in ensuring that the algorithms can be executed on a large number of bodies by maintaining low heap consumption. Figure 4 assesses the scalability of all implementations by plotting the runtimes against increasing input sizes on 8 cores: 12-24 for all-pairs and 80-200 for Barnes-Hut. The graph indicates almost linear scale-up within this window by all models. The *Eden* Barnes-Hut runtime is higher, reflecting a significantly higher memory consumption, which comes from *Eden's* distributed memory model, which duplicates data more often than needed on a shared memory architecture. As expected, Barnes-Hut is also able to cope with very large number of bodies e.g. 1 million.

Speed-up: The head-to-head comparison of speedups for the all-pairs versions of the code in Figure 5a show that, despite a higher variability, the *Eden* implementation performs best, even though it was designed for distributed memory architectures. This indicates that message passing can work well on shared-memory architectures. Using a highly tuned skeleton that avoids synchronisation bottlenecks on high-latency, distributed memory systems, is beneficial even on a single multi-core. The support for light-weight parallelism in all three runtime-systems helps to reduce the overhead that has to be paid for exposing parallelism. The *GpH* version is potentially more flexible and adaptive, through its dynamic, spark-based load distribution policy. This, is beneficial in particular in heterogeneous environments, with dynamically changing external load. On an otherwise idle machine as used for these measurements, however, these benefits cannot be capitalised on, while the overhead still has to be paid for. The *ParMonad* version performs very well already in its initial, unoptimised version, but does not exceed the performance of the other systems in its final version. In this case, the overhead of encoding scheduling and other dynamic machinery in the application, rather than the runtime-system, is higher compared to the other two systems.

The speedup results for the Barnes-Hut algorithm in Figure 5b show a significantly different picture. The dynamic behaviour of the Barnes-Hut algorithm differs from that of the all-pairs version, in that the parallel threads vary significantly in their granularities. The amount of work

DOI: 10.1002/cpe



(a) All-Pairs (b) Barnes-Hut

Figure 5. *Speed-up* graphs of both algorithms on up to 8 cores

is significantly higher when calculating the impact of a densely populated cube in the oct-tree representation. In contrast, the parallelism in the all-pairs version was more regular, with parallel tasks taking approximately the same amount of time to execute on different processors. The irregular parallelism in the Barnes-Hut version is more challenging to manage efficiently. The underlying runtime-system of *GpH* and the application-level implementation of scheduling for *ParMonad*, are designed to be very flexible and dynamic in their management of parallelism, in particular allowing for cheap transfer of potential parallelism. Considering the more challenging nature of the parallelism, *GpH* and *ParMonad* achieve good speedups. The *Eden* version, however, suffered most severely from the irregular parallelism. This case shows the limitations of a purely skeleton-based approach, that relies on the existence of a wide range of skeletons for many different architectures. Since *Eden* is not limited to such a pure skeleton-based approach, but is a skeleton implementation language in its own right, further optimisation should be possible, by fine tuning an existing skeleton for this application. We have not explored a cluster configuration of the *Eden* execution in sufficient detail to report any solid results on it here.

MultiCore Challenge input specification: Finally, Table IX shows the speed-up results for the tuned versions of all-pairs and Barnes-Hut, when using the MultiCore Challenge input specification of 1024 bodies and 20 iterations. As expected, the speed-ups are slightly lower for the smaller input set and for an execution which requires synchronisation between the iterations. Still, the speed-ups of 5.23 for *GpH* and 5.63 for *ParMonad* for the Barnes-Hut version are remarkable, for less than a dozen lines of code changes, and no structural changes to the original Haskell implementation. In particular, we surpass the calculated sequential overhead of a factor 3.4, compared to Fortran, already on a moderate multi-core architecture and deliver superior, scalable performance with a high-level language model.

	All-Pairs				Barnes-Hut			
	GpH		ParMonad		GpH		ParMonad	
no. PE	Runtime (s)	Speedup	Runtime (s)	Speedup	Runtime (s)	Speedup	Runtime (s)	Speedup
Seq.	1.67	1.00	1.70	1.00	1.36	1.00	1.35	1.00
1	1.71	0.98	1.66	1.02	1.40	0.97	1.38	0.98
2	0.94	1.78	0.93	1.83	0.78	1.74	0.72	1.88
4	0.51	3.27	0.52	3.27	0.44	3.09	0.42	3.21
8	0.30	5.57	0.30	5.67	0.26	5.23	0.24	5.63

Table IX. Runtime and speedup for 1024 bodies and 20 iterations

DOI: 10.1002/cpe

7. RELATED WORK

The area of declarative programming languages provides a particularly rich source for parallelism, stemming from the referentially transparent nature of the evaluation in a (pure) language [51]. This property guarantees that any order of evaluating an expression will deliver the same result. In particular, independent parts of the program can be evaluated in parallel. A straightforward implementation of this approach would yield an abundance of parallelism, with too fine grained threads to be efficient. Therefore most modern parallel functional languages take an approach of semi-explicit parallelism: they provide a means of identifying those sources of parallelism that are likely to be useful. Still, most of the synchronisation, communication and thread management is hidden from the programmer and done automatically in a sophisticated runtime-system.

An influential, early system for parallel functional programming was Mul-T [28] using Lisp. It introduced the concept of futures as handles for a data-structure, that might be evaluated in parallel and on which other threads should synchronise. Importantly for performance, this system introduced lazy task creation [39] as a technique, where one task can automatically subsume the computation of another task, thus increasing the granularity of the parallelism. Both, the language- and the system-level contributions have been picked up in recent implementations of parallel functional languages.

One prominent example of this approach is the Manticore system [19] using a parallel variant of ML that includes futures and constructs for data parallelism. It allows to specify parallelism on several levels in a large-scale applications, typically using explicit synchronisation and coordination on the top level [45] and combining it with implicit, automatically managed, fine-grained threads on lower levels [18].

Another ML extension is Polyml [37], which also supports futures, light-weight threads and implicit scheduling in its implementation. Reflecting its main usage as an implementation language for automated theorem provers, such as Isabelle, it has been used to parallelise its core operations.

Another parallel extension of Haskell is Data Parallel Haskell [25], which has evolved out of the Nepal language [7]. It provides language, compiler and runtime-system support for data-parallel operations, in particular parallel list comprehensions. Several program transformations, e.g. data-structure flattening, are performed in the compiler to optimise the parallel code. Its design is heavily based on the Blelloch's NESL language [5], a nested, data-parallel language with an explicit cost model to predict parallel performance.

SAC [46] is another functional, data-parallel language. Its syntax is based on C, but its semantics is based on single assignment, and therefore referentially transparent. It mainly targets numerical applications and achieves excellent speedups on the NAS benchmark suite.

Microsoft's Accellerator [47] is another prominent data-parallel system. It supports high-level, language-independent program development, and generates parallel code to be executed on GPGPUs or on multi-cores with vector-processing extensions.

Several experimental languages explored the use of high-level, parallelism language features in object-oriented languages: Fortress [48], X10 [9] and Chapel [8]. Of these, Chapel is currently best supported, in particular on massively parallel supercomputers. These languages introduce high-level constructs such as virtual shared memory (X10), structured programming constructs for parallel execution (Chapel), and software transactional memory (Fortress) to avoid a re-design of the software architecture due to specifics of the underlying, parallel architecture [10].

Based on the experiences with these languages, high-level abstractions are now entering main-stream languages for parallel programming. In particular, the concept of partitioned global address spaces (PGAS) enables the programmer to use the abstraction of virtual shared memory, while providing possibilities for co-locating data on specific nodes and thereby tune the parallel execution. The most prominent languages in this family are Unified Parallel C (UPC) [17] and Co-Array Fortran [41].

Increasingly, these high-level abstractions are also used in main-stream languages to facilitate parallel programming. The latest version of the .NET platform, comes with the Task Parallel Library [6], which provides a set of parallel patterns, in particular for divide-and-conquer and pipeline parallelism, and some advanced parallel programming constructs such as futures. Several

DOI: 10.1002/cpe

languages implemented on top of .NET, including C# and F#, can make direct use of these library functions in order to introduce parallelism, without extending the language itself. Intel's Task Building Blocks [44], also a collection of parallel patterns, has been successfully used on a range of parallel architectures.

Crucial for the feasibility of a high-level language approach to parallelism, is an efficient and flexible implementation of basic resource management policies, such as load distribution and scheduling. All three Haskell variants used in this paper profit from light-weight threads as managed in the runtime system. *GpH*'s runtime system manages work distribution through "sparks", effectively handles to potential parallelism [49]. Work represented by one spark can be subsumed by a running thread, effectively achieving lazy task creation [39]. *Eden*'s runtime system is more prescriptive in the way it manages the parallelism, which allows for the specification of skeletons implementing specific topologies [27]. *ParMonad*'s approach builds on the existing mechanisms for thread creation and synchronisation as initially developed for Concurrent Haskell [43].

Several other systems have taken similar design decision in producing a system for dynamic and adaptive management of parallelism. Filaments [33] was an early system focusing on light-weight threads, encouraging an approach of parallelisation that exposes massive amounts of parallelism and deciding at runtime whether or not to exploit specific parallelism, rather than restricting it on application level. The Charm++ system [26] builds on top of C++ and provides asynchronous message-driven orchestration together with an adaptive runtime system. It has been used on numerous, large-scale applications, for example biomolecular simulations from the domain of molecular dynamics. The Cilk system [21] achieved excellent results on physical shared memory systems, in particular for the FFT application. Its C and C++ language extensions are now supported both by Intel's Cilk Plus compiler and by GCC 4.7. Goldstein's thesis [22] provides a detailed study of different representations of light-weight threads and their impact on parallel performance, e.g. in the context of the TAM system [13].

8. CONCLUSIONS

In this paper we have used a very high level language approach to the challenge of obtaining efficient parallelism from a typical, compute-intensive application: the n-body problem. As host language we used Haskell, the de-facto standard, non-strict, purely functional language, that is increasingly used in academia and beyond to achieve a high level of programmer productivity.

We studied three variants of parallel Haskell: evaluation strategies, built on top of Glasgow parallel Haskell (*GpH*), *Eden*, which provides process abstractions akin to lambda abstractions to define parallel computations, and *ParMonad* with an explicit way of controlling threads. Common to all variants is the design philosophy to minimise the code changes that are needed in order to achieve parallel execution. Ideally, the specification of the parallel execution is orthogonal, and separate from the specification of the code. Indeed, the initial parallel versions of both algorithms required only one-line code changes. All of these languages build on a sophisticated runtime-system that automatically manages the synchronisation, coordination and communication necessary to achieve high parallel performance. The resulting programming model is one of semi-explicit parallel programming for *GpH* and *Eden*, where the programmer only has to identify the useful sources of parallelism, but explicit for *ParMonad*, which allows to encode archetypical runtime-system functionality as high-level Haskell code. More commonly, however, pre-defined parallel skeletons are used to simplify the parallelisation.

The three variants differ in the way they facilitate tuning of the initial parallel algorithm, though. Being first class objects, evaluation strategies can be easily modified to enable different dynamic behaviour. For example, adding chunking to a data parallel algorithm can be easily done by composing strategies. This modularity is one of its main advantages. However, control of data locality is significantly more difficult, because *GpH* relies on an implicit, work stealing scheduler to distribute the work. In contrast, in *Eden* thread creation is mandatory on process application, and it provides finer control of co-location, by using partial applications. These features provide more opportunities for tuning the parallel program without abandoning the high level of

DOI: 10.1002/cpe

abstraction. Finally, *ParMonad* is the most explicit form of controlling parallelism. Here, threads are explicit entities in the program, that have to be synchronised using `IVars`, which raises all the usual issues about parallel programming. However, by providing parallel patterns of computation, skeletons, these low-level issues can be hidden to the programmer. By implementing runtime-system functionality on Haskell level, an expert parallel Haskell programmer can also tailor the application, e.g. by implementing a custom scheduling algorithm.

Another current parallel Haskell extension is Data parallel Haskell (DPH), which implements a model of implicit, nested data-parallelism on top of parallel arrays. For a predecessor of DPH, Nepal, it is shown how a Barnes-Hut algorithm can be implemented in this language [25]. Concrete performance comparisons with this Haskell variant would be interesting future work.

Despite the high-level of abstraction, the performance results show good *speedups* for all systems: 5.45 for *GpH*, 6.50 for *ParMonad*, and 3.62 for *Eden*, always using the Barnes-Hut algorithm. Most notably, these results were achieved changing only a few lines of code. Introducing top-level data-parallelism changes only one line of the original code. Introducing the chunking optimisation adds less than a dozen lines of auxiliary functions, and modifies this one line code change.

Our main conclusions from this challenge implementation are:

- All three parallel Haskell variants are able to achieve competitive multi-core speed-ups not only for the simple, regular all-pairs algorithm but also for the more sophisticated, irregular Barnes Hut algorithm.
- The performance of the parallel all-pairs version surpasses the calculated performance of the fastest sequential Fortran version from the language shootout [1] and achieves scalable performance up to the maximum of 8 cores.
- The ease of parallelisation allowed us to develop 6 versions of the challenge, using three different variants of parallel Haskell and implementing both an all-pairs and a Barnes-Hut version.
- Well documented program transformations, in the form of local changes to the sequential program, reduce both heap and stack space consumption considerably and improve sequential performance by a factor of 12.0.
- Established techniques for tuning parallel performance, in particular chunking, were important to tune the *GpH* and *Eden* implementations of the algorithms.
- The *ParMonad* version already achieves good parallel performance in its initial version, due to a highly optimised, work-inlining scheduler. In contrast, both the *GpH* and *Eden* versions require explicit chunking to achieve the same level of performance, but allow for more flexible tuning of performance.
- Interestingly *Eden*, which is designed for distributed memory architectures, performs very well on a shared memory setup using message passing, in particular for the all-pairs version.

We found this exercise of implementing one agreed challenge application, and comparing the parallelisation methodology, the tool support and the achieved performance results with other implementations, that were presented at the workshops, an extremely valuable experience, gaining insights in the relative advantages of each of the approaches. These results also help to focus our research efforts in developing the underlying systems further. In particular, one main direction of further work is to improve our runtime support for hierarchical, heterogeneous parallel architectures, e.g. clusters of multi-cores, and to integrate the different Haskell variants into one unified language that makes use of these variants on different levels in the hierarchy. *Eden*, based on a distributed memory model, is a natural match with clusters, whereas *GpH* and *ParMonad* are natural matches for physical shared memory architectures. *GpH* also supports virtual memory, which can be efficiently exploited on closely connected clusters. For the next challenge application we hope to also cover clusters of multi-cores to address the issue of the scalability of the parallelism in more detail. *Eden* already provides a suitably platform for such a comparison, and we hope to also have a stable cluster-version of *GpH* ready to use for this next stage.

DOI: 10.1002/cpe

9. ACKNOWLEDGMENTS

The authors would like to thank: the participants in the MultiCore Challenge who took up the challenge and provided useful insight of the all-pairs algorithm performance in other technologies at the workshops; researchers in the Dependable Systems Group at Heriot-Watt University for their continued work on GpH; the Eden Group in Marburg for many useful technical discussions and for their support in working with Eden; and SICSA (the Scottish Informatics and Computer Science Alliance) for sponsoring the first author through a PhD studentship.

REFERENCES

1. The Computer Language Benchmarks Game. Website `http://shootout.alioth.debian.org`, February 2012.
2. B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P^3L: a Structured High-level Programming Language and its Structured Support. *Concurrency and Computation: Practice and Experience*, 7(3):225–255, 1995. doi:10.1002/cpe.4330070305.
3. J. Barnes and P. Hut. A hierarchical O(N log N) force-calculation algorithm. *Nature*, 324:446–449, December 1986.
4. A. Benoit, M. Cole, S. Gilmore, and J. Hillston. Flexible Skeletal Programming with eSkel. In *EuroPar'05*, LNCS 3648, pages 761–770. Springer, 2005.
5. Guy Blelloch. NESL: A Parallel Programming Language. Website `http://www.cs.cmu.edu/~scandal/nesl.html`, February 2012.
6. 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, August 2010.
7. Manuel M.T. Chakravarty, Gabriele Keller, Roman Lechtchinsky, and Wolf Pfannenstiel. Nepal – nested data-parallelism in haskell. In *IN EURO-PAR 2001*, pages 524–534. Springer-Verlag, 2001.
8. Bradford L. Chamberlain, David Callahan, and Hans P. Zima. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications*, 21(3):291–312, August 2007.
9. Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of OOPSLA '05*, pages 519–538, New York, NY, USA, 2005. ACM Press.
10. Andrew Chien. Parallelism Drives Computing. Talk given at Manycore Computing Workshop, 2007 June.
11. P. Ciechanowicz, M. Poldner, and H. Kuchen. The Münster Skeleton Library: Müsli. Technical Report ERCIS Working Paper No. 7, University Münster, 2010.
12. M. Cole. *Algorithmic Skeletons: Structural Management of Parallel Computation*. Pitman/MIT Press, 1989.
13. D.E. Culler, S.C. Goldstein, K.E. Schauser, and T. von Eicken. TAM — A Compiler Controlled Threaded Abstract Machine. *Journal of Parallel and Distributed Computing*, 18:347–370, June 1993.
14. Daytona — Iterative MapReduce on Windows Azure, February 2012.
15. J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI '04 — Symp. on Operating System Design and Implementation*, pages 137–150, 2004.
16. J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.
17. Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick. *UPC: Distributed Shared Memory Programming*. John Wiley and Sons, May 2005. ISBN 0-471-22048-5.
18. M. Fluet, M. Rainey, J. Reppy, and A. Shaw. Implicitly-threaded parallelism in Manticore. *Journal of Functional Programming*, 20(5–6):537–576, 2010.
19. M. Fluet, M. Rainey, J. Reppy, A. Shaw, and Y. Xiao. Manticore: A heterogeneous parallel language. In *DAMP 2007: Workshop on Declarative Aspects of Multicore Programming*, pages 37–44, Nice, France, January 2007. ACM Press.
20. Apache Foundation. Apache hadoop nextgen mapreduce (yarn). web page, February 2012.
21. Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI98: Conference on Programming Language Design and Implementation*, pages 212–223, Montreal, Quebec, Canada, June 1998. Proceedings published ACM SIGPLAN Notices, Vol. 33, No. 5, May, 1998.
22. Seth Copen Goldstein. *Lazy Threads: Compiler and Runtime Structures for Fine-rained Parallel Programming*. PhD thesis, University of California at Berkeley, 1997.
23. Horacio Gonzalez-Velez and Mario Leyton. A Survey of Algorithmic Skeleton Frameworks: High-Level Structured Parallel Programming Enablers. *Software: Practice and Experience*, 2010. To appear.
24. Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *European Conference on Computer Systems (EuroSys)*, Lisbon, Portugal, March 2007.
25. Simon Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel M. T. Chakravarty. Harnessing the Multicores: Nested Data Parallelism in Haskell. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2008)*, IBFI, Schloss Dagstuhl, 2008.
26. Laxmikant V. Kale and Sanjeev Krishnan. Charm++: a portable concurrent object oriented system based on c++. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and*

applications, OOPSLA '93, pages 91–108, New York, NY, USA, 1993. ACM.
27. U. Klusik, R. Loogen, S. Priebe, and F. Rubio. Implementation Skeletons in Eden: Low-effort Parallel Programming. In *IFL'00: International Workshop on the Implementation of Functional Languages*, LNCS 2011. Springer-Verlag, 2000.
28. D.A. Kranz, R.H. Halstead Jr., and E. Mohr. Mul-T: A High-Performance Parallel Lisp. In *PLDI'91 — Programming Languages Design and Implementation*, volume 24(7) of *SIGPLAN Notices*, pages 81–90, Portland, Oregon, June 21–23, 1989.
29. Mario Leyton and Jose M. Piquer. Skandium: Multi-core programming with algorithmic skeletons. In *IEEE Euromicro PDP 2010*, 2010.
30. H.-W. Loidl, F. Rubio, N. Scaife, K. Hammond, S. Horiguchi, U. Klusik, R. Loogen, G. J. Michaelson, R. Peña, S. Priebe, Á J. Rebón, and P. W. Trinder. Comparing parallel functional languages: Programming and performance. *Higher Order Symbol. Comput.*, 16:203–251, September 2003.
31. H-W. Loidl, P.W. Trinder, K. Hammond, S.B. Junaidu, R.G. Morgan, and S.L. Peyton Jones. Engineering Parallel Symbolic Programs in GPH. *Concurrency and Computation: Practice and Experience*, 11:701–752, 1999.
32. Rita Loogen, Yolanda Ortega-mallén, and Ricardo Peña marí. Parallel functional programming in eden. *J. Funct. Program.*, 15:431–475, May 2005.
33. David K. Lowenthal and Vincent W. Freeh Gregory R. Andrews. Using Fine-grain Threads and Run-time Decision Making in Parallel Computing. *Journal of Parallel and Distributed Computing*, 37(1), 1996. Special issue on multithreading for multiprocessors.
34. Simon Marlow, Simon Peyton Jones, and Satnam Singh. Runtime support for multicore haskell. In *ICFP '09: Proceeding of the 14th ACM SIGPLAN International Conference on Functional Programming*, August 2009.
35. Simon Marlow, Patrick Maier, Hans-Wolfgang Loidl, Mustafa K. Aswad, and Phil Trinder. Seq no more: better strategies for parallel haskell. In *Proceedings of the third ACM Haskell symposium on Haskell*, Haskell '10, pages 91–102, New York, NY, USA, 2010. ACM.
36. Simon Marlow, Ryan Newton, and Simon Peyton Jones. A monad for deterministic parallelism. In *Proceedings of the 4th ACM symposium on Haskell*, Haskell '11, pages 71–82, New York, NY, USA, 2011. ACM.
37. David C. J. Matthews and Makarius Wenzel. Efficient Parallel Programming in Poly/ML and Isabelle/ML. In *DAMP10: Declarative Aspects of Multicore Programming*, Madrid, Spain, November 2010.
38. T. G. Mattson, B. A. Sanders, and B. L. Massingill. *Patterns for Parallel Programming*. Addison-Wesley, 2004. ISBN 978-0321228116.
39. E. Mohr, D.A. Kranz, and R.H. Halstead Jr. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, July 1991.
40. Rishiyur Nikhil and Arvind. *Implicit Parallel Programming in pH*. Morgan Kaufmann, 2001. ISBN 978-1558606449.
41. Robert W. Numrich and John Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17:1–31, August 1998.
42. Simon Peyton Jones. Harnessing the multicores: Nested data parallelism in haskell. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, APLAS '08, pages 138–138, Berlin, Heidelberg, 2008. Springer-Verlag.
43. S.L. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *POPL'96 — Symposium on Principles of Programming Languages*, pages 295–308, St Petersburg, Florida, January 1996. ACM.
44. James Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly, 2007.
45. John Reppy, Claudio Russo, and Yingqi Xiao. Parallel Concurrent ML. In *International Conference on Functional Programming (ICFP 2009)*, September 2009.
46. Sven-Bodo Scholz. Single Assignment C – Efficient Support for High-level Array Operations in a Functional Setting. *Journal of Functional Programming*, 13(6):1005–1059, 2003.
47. Satnam Singh. Declarative Data-Parallel Programming with the Accellerator System. In *DAMP10: Declarative Aspects of Multicore Programming*, Madrid, Spain, November 2010.
48. Sun. The Fortress Language. Talks and Posters. available at `http://research.sun.com/projects/plrg`.
49. P. Trinder, K. Hammond, J.S. Mattson Jr., A.S Partridge, and S.L. Peyton Jones. GUM: a Portable Parallel Implementation of Haskell. In *PLDI'96 — Programming Languages Design and Implementation*, pages 79–88, Philadelphia, PA, USA, May 1996.
50. P. W. Trinder, K. Hammond, H.-W. Loidl, and S. L. Peyton Jones. Algorithm + Strategy = Parallelism. *J. Funct. Program.*, 8:23–60, January 1998.
51. P.W. Trinder, K. Hammond, and H-W. Loidl. *Encyclopedia of Parallel Computing*, chapter Parallel Functional Languages. Springer Verlag, 2011. ISBN 978-0-387-09844-9.

DOI: 10.1002/cpe

10. APPENDIX

(a) GpH

	nochunk		1chunk/PE		2chunks/PE		4chunks/PE	
no. PE	Runtime (s)	Speedup	Runtime (s)	Speedup	Runtime (s)	Speedup	Runtime (s)	Speedup
Seq.	20.04	1.00	20.05	1.00	20.03	1.00	20.06	1.00
1	20.64	0.97	27.80	0.72	24.07	0.83	22.10	0.91
2	16.76	1.20	12.73	1.58	11.98	1.67	11.33	1.77
4	13.89	1.44	6.42	3.12	6.15	3.26	5.97	3.36
8	15.64	1.28	3.40	5.90	3.35	5.98	3.29	6.10

(b) Par monad

	nochunk		1chunk/PE		2chunks/PE		4chunks/PE	
no. PE	Runtime (s)	Speedup	Runtime (s)	Speedup	Runtime (s)	Speedup	Runtime (s)	Speedup
Seq.	20.30	1.00	20.06	1.00	20.03	1.00	20.04	1.00
1	20.48	0.99	20.16	1.00	20.08	1.00	20.19	0.99
2	10.96	1.85	10.91	1.84	10.98	1.82	10.94	1.83
4	5.93	3.42	5.85	3.43	5.82	3.44	5.78	3.47
8	3.29	6.17	3.24	6.19	3.22	6.22	3.25	6.17

(c) Eden 1

	parMap		parMapFarm		parMapFarmMinus	
no. PE	Runtime (s)	Speedup	Runtime (s)	Speedup	Runtime (s)	Speedup
Seq.	22.11	1.00	22.13	1.00	22.12	1.00
1	362.38	0.06	23.67	0.93	23.59	0.94
2	294.99	0.07	11.91	1.86	23.33	0.95
4	259.19	0.09	6.08	3.64	7.73	2.86
8	245.72	0.09	3.41	6.49	3.49	6.34

(d) Eden 2

	parMapFarmChunk		parMapOfflineFarmChunk		workpoolSortedChunk	
no. PE	Runtime (s)	Speedup	Runtime (s)	Speedup	Runtime (s)	Speedup
Seq.	22.13	1.00	22.13	1.00	22.12	1.00
1	22.91	0.97	23.02	0.96	23.06	0.96
2	11.57	1.91	11.53	1.92	11.59	1.91
4	5.82	3.80	5.80	3.82	5.84	3.79
8	3.09	7.16	3.04	7.28	3.11	7.11

Table X. All-Pairs: Measurements (16k bodies, 1 iteration)

	processes	threads	conversations	messages
parMap	16001	32001	64000	64000
parMapFarm	9	17	48	32048
parMapFarmMinus	8	15	42	32042
parMapFarmChunk	9	17	48	80
parMapOfflineFarmChunk	9	17	40	56
workpoolSortedChunk	9	17	48	80

Table XI. All-Pairs: Eden skeleton overheads - 16000 bodies/par. run on 8 cores

DOI: 10.1002/cpe