Haskell vs. F# vs. Scala:

A High-level Language Features and Parallelism Support Comparison

Prabhat Totoo Pantazis Deligiannis Hans-Wolfgang Loidl

School of Mathematical and Computer Sciences,
Heriot-Watt University,
Edinburgh, EH14 4AS, U.K.
{pt114, pd85, H.W.Loidl}@hw.ac.uk

Abstract

This paper provides a performance and programmability comparison of high-level parallel programming support in Haskell, F# and Scala. Developing several parallel versions, we employ skeleton-based, semi-explicit and explicit approaches to parallelism. We focus on advanced language features for separating computational and coordination aspects of the code and tuning performance. We also assess the impact of functional purity and multi-paradigm design of the languages on program development and performance. Basis for these comparisons are several Barnes-Hut implementations of the n-body problem in all three languages, on both Linux and Windows. Our performance measurements on state-of-the-art multi-cores achieve a speedup up to 5.62 (on 8 cores) with a highly-tuned Haskell version. For comparable implementations in Scala and F# we achieve speedups of 4.51 (on 8 cores) and 2.28 (on 4 cores), respectively. We observe that near best speedups are achieved using the highest level abstractions in these languages.

Categories and Subject Descriptors D.1.3 [*Programming techniques*]: Concurrent programming—Parallel programming

General Terms Languages, Performance

Keywords Haskell, F#, Scala, Parallelism, Barnes-Hut

1. Introduction

Because functional languages are not defined in terms of operations on a hidden global state, they avoid unnecessary sequentialisation and provide ample latent parallelism that can be exploited by compiler or runtime-system. This property makes them an attractive platform for exploiting common-place parallel hardware without imposing new concepts of explicit threads with explicit communication onto every parallel application.

With functional programming, the programmer needs only to specify *what* instead of *how* to compute something. Managing parallelism is all about *how* and therefore largely hidden from the programmer. However, for tuning parallel performance, some limited control of operational aspects is desirable. The approaches to efficiently exploit such latent parallelism, provided in the latest implementations of state-of-the art languages such as Haskell, F# and Scala, all aim to be minimally intrusive to the code, while giving the expert parallel programmer sufficient control to perform parallel performance tuning. In this paper we evaluate language mechanisms provided in each of these three languages, and present a head-to-head comparison of the resulting parallel performance.

A new generation of programming languages, such as F# and Scala, often take a multi-paradigm approach, embedding the advantages of functional languages into a main-stream, object-oriented language. They use existing, highly-optimised VM technology, .NET and JVM respectively, to combine the ease of expressing parallelism with efficient sequential execution. In this paper we perform a head-to-head programmability comparison between purely functional Haskell and multi-paradigm F# and Scala. We assess the impact of key language design issues, in particular laziness and mutable data-structures, on sequential and parallel performance. Finally, we give a head-to-head parallel performance comparison of Haskell, F# and Scala, using different techniques to expose parallelism at different levels of abstraction. The results from our measurements of a Barnes-Hut implementation of the n-body algorithm show that we achieve respectable speedups in all languages. We achieve a speedup of 5.62 (on 8 cores) with a highly-tuned Haskell version. With the implementations in Scala and F# we achieve speedups of 4.51 (on 8 cores) and 2.28 (on 4 cores), respectively.

The remainder of the paper is organised as follows: Section 2 presents related work; Section 3 presents the background behind Haskell, F# and Scala, and their support for parallelism; Section 4 presents our multiple Barnes-Hut algorithm implementations, both sequential and parallel; Section 5 presents results from our measurements on two different multi-core architectures, an 8-core Linux machine and a 4-core (with hyperthreading) Windows machine; and we summarise our findings in Section 6.

2. Related Work

We can broadly classify parallel declarative languages as implicitly parallel, without any explicit control of parallelism, semi-explicit, only exposing potential parallelism, and explicit, with constructs for the generation and handling of explicit threads. In this section we focus on and survey semi-explicit approaches, although the lower level constructs in Scala can also be classified as explicit. The general area of parallel declarative programming languages is surveyed in depth in [32]. A more focused comparison of parallel Haskell variants is given in [31].

Crucial to a high-performance implementation of a declarative language is an adaptive runtime-system, that can make good decisions about the management of parallelism, usually deferred to the programmer in explicit languages. Important concepts are futures

as handles for a data-structure, that might be evaluated in parallel and on which other threads should synchronise, first introduced in the Mul-T [13] variant of Lisp. Importantly for performance, this system introduced lazy task creation [22] 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: both F# and Polyml [21] provide language-level futures; *GpH*'s runtime system uses lazy task creation, by representing potential parallelism as "sparks" that can move freely and cheaply between processors and work represented by one spark can be subsumed by a running thread if no additional parallelism is required. Another key runtime-system design goal is to support light-weight threads, thus reducing the overhead for creating parallelism and encouraging a programming style that generates a massive amount of parallelism, giving the runtime-system the flexibility to arrange the parallelism in a way most suitable to the underlying hardware. Haskell/GHC excels at light-weight threads, as shown by the thread-ring benchmark of the Computer Language Shootout [1]. Filaments [17] and Cilk [11], now integrated in Intel's Cilk Plus compiler, are other examples of runtime-systems for light-weight threads.

Several experimental languages explored the use of high-level, parallelism language features in object-oriented languages: Fortress [28], X10 [7] and Chapel [6]. 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.

An increasingly important area of high-level abstractions for parallelism are parallel patterns or algorithmic skeletons [8], higher-order functions with pre-defined parallel computation structures. Because they can hide all complexities of the efficient, possibly hardware-dependent, handling of parallelism in a library, it is being picked up as technology of choice in main-stream languages without built-in high-level parallelism support. Prominent examples are Google's MapReduce [10] implementation, on large-scale, distributed architectures, Intel's Task Building Block [26] library, and to some extent the Task Parallel Library [15].

3. Background

3.1 Haskell

Haskell [20] is a statically typed, lazy and purely functional language. Haskell is strongly typed but type definitions are rarely needed as it uses type inference to deduce types automatically. Its advanced type system also supports algebraic data types and polymorphic functions. Type classes are used as interfaces with default implementations. Instances of a specific type class group types together e.g. the `Num` type groups `int`, `double` and other numerical types. One of its most distinctive features is its lazy semantics, which means that expressions are only ever evaluated when they are demanded. This demand-driven evaluation strategy makes it possible to have infinite data structures and circular programs in Haskell. As a pure language, Haskell prohibits side-effects, which guarantees that a call to a function only returns a value without causing any change to global states. Functions are first-class, they can be passed as argument, returned as results, and treated as values.

Haskell makes use of monads to separate pure from side-effecting (impure) computations. With no default evaluation order specified, computations are chained using monads and through use of syntactic sugar such as the `do` notation. This gives imperative flavour of coding, but underlying these constructs are simply functions. Haskell comes with a number of data structures, the most notable is the list. Lists fit well in a functional context and are omnipresent in Haskell codes. They are implemented as singly-linked lists. Other common data structures include arrays, both immutable and mutable, which provide element access time of *O(1)*.

GHC is the main implementation of the Haskell language. It consists of a highly optimising, transformation-based compiler and graph-reduction-based runtime system. GHC has good support for semi-explicit parallelism. Though evaluation of functions can happen independently and in any order, parallelising all of them often leads to too fine-grained parallelism. By default, GHC now includes parallel support through the GpH extension for shared-memory via GHC-SMP.

3.1.1 GpH

Glasgow parallel Haskell [18, 19, 33] is a minimal, conservative, parallel extension of Haskell, supported by the GHC compiler. It extends standard Haskell by providing two basic primitives for specifying and controlling parallelism.

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

`par` allows the programmer to annotate computations that can be usefully be evaluated in parallel. The first argument is sparked and may potentially be executed in parallel with the evaluation of the second argument. `pseq` enforces sequential ordering which is needed to arrange parallel computations.

Naive usage of these primitives can lead to unexpected parallel behaviour, for example generating sparks for already evaluated data. *Evaluation strategies* are abstractions over the primitives to provide an even higher level of control of parallelism. Evaluation strategies provide a clean separation of the coordination aspects from the main computation. For example, `parList` can be used to demand parallel evaluation of each list element, separately from defining the contents of the list. Additionally, evaluation degree and evaluation order can be specified using evaluation strategies.

The following example uses the `parList` strategy to define a `parMap` skeleton:

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

3.2 F#

F# [29] combines the features of a strict, higher-order, impurely functional language of an ML-style, with features of main-stream object-oriented languages. Both paradigms are made available to the programmer, who can make a choice based on the suitability of the paradigm for the application and on his familiarity with the paradigm. On balance, F# emphasises its heritage from ML, though, demonstrated through ML compatibility in its light mode. The F# implementation compiles to .NET as the VM and can therefore build on highly-optimised VM implementations, and interact with libraries in other languages also targeting this intermediate platform. The latter is important and convenient when switching between data structures. For example, data structures like list and seq are instances of the .NET IEnumerable interface. Exposing functional code through objects is often useful when interacting with other, foreign language .NET objects.

From a pragmatic point of view the rich tool support through VisualStudio and its apparent backing by Microsoft make it a very attractive language in particular for programmers previously unaware

of functional programming. There is a tendency in its implementation to hide details of the parallel execution from the programmer. While this encourages a high-level of abstraction, best supported through data-parallel PLINQ, it also complicates the tuning of the program for the expert parallel programmer.

F# makes a clear distinction between between *value* and *variable*. The former, which includes functions, are immutable. The latter is defined using the `mutable` keyword which denotes a variable whose value can change using the left arrow operator, thus allowing mutable states. F# computation expressions hide the complexity of monadic syntax behind heavy use of syntactic sugar.

The most prominent F# collections are mutable arrays, ordered lists and ordered sequences, in which evaluation is demand-driven, but LazyList from the F# PowerPack are also available and similar to Haskell lists. It uses caching and allows pattern matching unlike sequence.

In terms of parallelism support, F# benefits from the .NET Parallel Extensions. The extensions provide a number of high-level constructs to write and execute parallel programs. The Parallel Extensions consist of the Task Parallel Library, Parallel LINQ and a set of *coordination data structures*. In addition to these, Asynchronous Workflows, which is intended mainly for operations involving I/O, provide basic parallelisation. These libraries can be combined to take advantage of potential parallelism in a program.

3.2.1 Asynchronous Workflows

Asynchronous workflows allow the creation of multiple threads in an otherwise single-threaded application in order to avoid blocking asynchronous operations such as I/O or interactions in user interfaces. They can be used to add basic parallelisation to the code with limited code changes. A computation can be executed in parallel by wrapping it inside an `async` block (a workflow) that will run asynchronously without blocking the current computation thread. For example, a web server can handle requests simultaneously. On a parallel machine, these requests are executed in parallel thus improving the performance. It is also possible to specify a continuation function when the computation ends.

```
let handleRq rq = async { (* some code here *) }
Async.RunSynchronously (handleRq request)
```

3.2.2 Task Parallel Library

The Task Parallel Library [5, 15] simplifies the process of adding parallelism to a program by abstracting over low-level mechanisms such as thread creation, management and scheduling behind a set of high-level APIs. TPL scales the degree of parallelism automatically depending on the number of available processors. The following constructs are either provided by TPL or built on top of it.

Tasks: The main construct for task parallelism is built around the concept of a *task* which represents a unit of work that can be executed independently. Tasks provides a high-level abstraction compared to working directly with threads. The `Task` type represents a computation that does not return any value. The `Task<TResult>` type represents an operation that calculates a value of type `TResult` eventually i.e. a *future*. The computation of a future happens in the background thread and synchronisation is implicit, i.e. if the result is not yet computed when requested, the asking thread will block until the result is available. TPL provides several methods for composing and working with tasks for e.g. task continuation and cancellation functions.

```
let tasks ts =
  [| for t in ts
       Task.Factory.StartNew(fun () ->
           process t)
  |]
Task.WaitAll tasks
```

Parallel Class: TPL supports *data parallelism* through the Parallel class which includes static methods such as Parallel.For and Parallel.ForEach for basic loops parallelisation. These methods provide an easy way of parallelising `for` loops.

```
Parallel.For(0, n, (fun i -> process i))
```

Parallel LINQ: PLINQ is a declarative model for data parallelism based on Language Integrated Query. It provides a shallow embedding of an SQL-like query language directly in the general purpose, host language (in this case F#) and allows to query XML data, databases or objects from standard data collections. The latter is how we use PLINQ. The implementation uses TPL internally for efficient implementation, and is the highest level language mechanism for parallelism in F#.

3.3 Scala

Scala is a general purpose, statically typed, strict, multi-paradigm programming language, combining functional and object-oriented features [24]. The language allows the expression of common programming patterns in a concise, elegant and type-safe manner. Scala's compiler targets the Java Virtual Machine (JVM) platform and its implementation is freely available as an open source project. Scala is fully interoperable with Java, thus empowering the programmer with the full range of Java libraries and frameworks. The language was also designed with extensibility in mind, meaning that new features can be easily added in the form of new libraries without the need to change the syntax of the language.

A main focus of Scala is to deliver state-of-the-art high-level constructs and abstractions for concurrent programming, emphasising large-scale distribution, scalability and fault-tolerance. Towards this goal it provides a number of parallel and distributed programming frameworks, most notably the Scala Parallel Collections [25] and the Scala Actors [12].

3.3.1 Scala Parallel Collections Framework

Scala 2.8 was a major milestone for the language as it introduced a new collections framework that focuses on providing an easy to use, concise, safe and uniform approach for interacting with the available data structures [23]. The framework is based on a sophisticated class hierarchy where each collection inherits and extends the functionality defined in its parent classes. The collection hierarchy splits into three main categories: `seq` (indexed collections), `map` (key and value pairs) and `set` (collections without duplicate elements). These three categories are blueprints for the rest of the available Scala collections. The framework also contains two major subpackages, `mutable` and `immutable`, each one providing a different set of data structure implementations. One of the main benefits of such a hierarchical approach is that the programmer can easily swap a collection for another and still using the same operations. This leads to easier and more approachable tuning, refactoring and maintenance of programs.

Scala 2.9 was released in 2011, enhancing the collections framework with semi-explicit parallelism [25]. Instead of defining a new set of parallel methods for each collection, the chosen design was to build upon the existing hierarchy of collections. The subpackage `parallel` was introduced, which defines parallel implementations for sequences, maps and sets, together with common parallel operations. The programmer can use the method `par` on a sequential collection to invoke its corresponding parallel implementation. With the method `seq` the parallel collection behaves again in a sequential manner. The benefit of this approach is that parallel oper-

ations can have the same names as their sequential versions, which means that the programmer can easily introduce parallelism by just providing the method par in the right places, as shown below.

```
xs.map( (x: Int) => x + 1 ) // sequential

xs.par.map( (x: Int) => x + 1 ).seq // parallel
```

The implementation of the parallel collections library is built on top of the Java Fork/Join framework [14]. This is basically a thread pool implementation that aims to efficiently schedule fork/join tasks among available processors. Inspired by divide and conquer and recursive approaches to parallelism, a fork/join task can spawn (fork) new tasks and wait for them to finish (join) before progressing with the execution. Currently the fork/join implementation uses two core techniques, adaptive work stealing [9, 14] and exponential task splitting [9], in order to efficiently control the task granularity. Although the programmer does not currently have much control over the level of parallelism provided by the Scala Parallel Collections Framework, the implementation has been carefully tuned to ensure a high parallel performance.

3.3.2 Scala Actors Library

Scala aims to support concurrency by providing an explicit message passing programming model based on actors. Actors are first-class, light-weight processes that communicate with each other by exchanging asynchronous messages [2]. These messages are gathered in the receiving actor's mailbox. An actor is able to iterate through its mailbox and respond to the various messages it has gathered by using pattern matching, a staple approach in functional programming. Responses to messages include among other actions: creating a new actor; sending a new message to the sender; and changing the underlying behaviour of the receiving actor. This design is motivated by Erlang [3].

```
// actor sending a message
a ! msg

 // receiving msgs and responding with actions
receive {
  case msg_pattern_1 => action_1
  case msg_pattern_2 => action_2
  case msg_pattern_n => action_n
}
```

3.4 Summary

In summary, the following table compares and contrasts the key features of each of the three languages and their parallelism support listed in order of their level of abstraction.

	Key Features	Parallelism Support
Haskell	functional, lazy evaluation, static/inferred typing	Eval strategies (skeletons), par and pseq (semi-explicit)
F#	functional, imperative, object oriented, strict evaluation, static/inferred typing, .NET interoperability	TPL, PLINQ (skeletons), Async workflows (semi-explicit)
Scala	functional, imperative, object oriented, strict evaluation, static/inferred typing, Java interoperability	Parallel Collections (skeletons), Actors (explicit)

Haskell is designed as a purely functional language and therefore does not include features for object-oriented and imperative programming. However, it does support foreign language integration e.g. C through the FFI library, which can perform unsafe operations inside the IO monad. F# is mostly functional but its design aims at integration with other paradigms from the offset. Scala is mainly influenced by Java and so many of the language concepts are tied to objects. However, it does provide fairly good functional support, though the syntax differs from more traditional functional languages a bit.

All three languages provide advanced type systems with automatic type inference and support high-level approaches for multi-core parallelism. Platform-wise, Haskell has its own graph-reduction-based runtime-system, F# compiles down to .NET Common Intermediate Language (CIL) and then runs on the Common Language Runtime (CLR), Scala programs compile to Java Byte Code and runs on the Java Virtual Machine (JVM).

4. Implementation

In this section, we provide implementations for the n-body problem in Haskell, F# and Scala. A detailed implementation of the problem in Haskell is provided in [30]. We produce the F# and Scala versions based on this implementation. All versions are used to compare the facilities and difficulties in each language and their performance.

4.1 N-Body Problem

Informally, the N-body problem is the problem of predicting and simulating the motion of a system of N bodies that interact with each other gravitationally. It represents an important algorithm in many areas of science e.g. simulation of particles in astrophysics and molecules in biological systems. In direct gravitational N-body simulations, the bodies (e.g. stars, galaxies or molecules) interact with each other by applying the gravitational force, which affects the movement of both nearby and far away bodies in the system (in a different scale). The simulation proceeds over a specified number of time steps, where the acceleration of each body with respect to the other bodies is calculated and then used to update the corresponding position and velocity in each iteration.

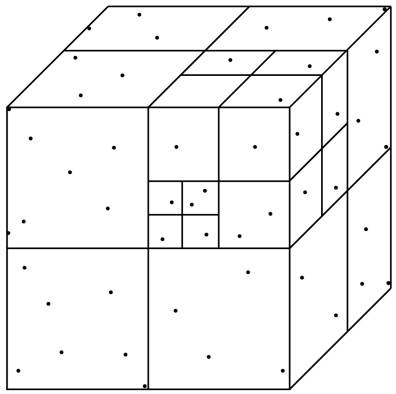


Figure 1. Points in a 3D Barnes-Hut nbody simulation are contained in a region (bounding box) which is sub-divided recursively into smaller regions.

The naive method of solving the problem consists of a body-to-body force comparison (see *all-pairs* implementation in [30]). This approach is not feasible for large number of bodies and therefore hierarchical force-calculation algorithm such as *Barnes-Hut* [4]

algorithm provides an efficient approximation solution. The core of the algorithm consists of two phases: tree construction and force calculation, the latter being the most compute-intensive phase. In the first phase, an octree is constructed from the list of bodies (see Figure 1). In the second phase, the acceleration due to each body is computed by traversing the tree and approximating bodies that are too far away by using the centre of mass of nearby bodies. In the algorithm, opportunity for parallelism exists at the tree construction and accelerations calculation stage. The tree construction phase, however, accounts for a small percentage of the overall time. So we focus on parallelising the force calculation phase which can be done independently for each body once the tree is constructed. The following is the algorithm for the Barnes-Hut nbody simulation:

```
function doStep (bs,n)
  if (n==0) return bs
  else
    //find the bounding box of the bodies
    bbox = findBounds bs
    //build the BH tree (also calculate centre
        of mass of each region)
    tree = buildTree (bbox,bs)
    //use tree to update velocities and
        positions
    foreach b in bs
      accel = calcAccel b tree
      //deduct acceleration from velocity of
          body b
      b = updateVel b accel
      //move body b
      b = updatePos b
    doStep(bs,n-1)
```

The *calcAccel* function calculates the acceleration of a given body against the others by traversing the Barnes-Hut tree. It is the main source of parallelism and can be performed independently for each body.

4.2 Sequential Implementations and Optimisations

The first step in writing parallel code is to come up with an efficient sequential implementation. Towards this goal, the opportunity to introduce parallelism should be preserved. Using mutable states to get the best sequential performance destroys any opportunity for parallelism. Instead, keeping the implementation pure usually eases the parallelisation step. In our case, the sequential Barnes-Hut algorithm is initially implemented in all three languages. The chosen data structure is *list*, as it is the most commonly used in functional languages. Next, we try to improve the sequential implementation by applying a number of generic and language specific optimisations.

4.2.1 Generic Optimisations

A number of general optimisation techniques can be used in the Barnes-Hut algorithm in order to get efficient sequential implementations across all languages. These optimisation techniques are (we assess their impact on performance in Section 5):

Deforestation can be used as an instance of (manual) program transformation. For example, elimination of multiple traversals of data structures can improve sequential runtimes by doing fusion, such as merging fold and map, and using function composition in map operations: `map (f . g) xs` in Haskell; `f << g` in F#; and `f andThen g` in Scala. Some of these transformations are done automatically, e.g. by GHC, with full optimisation enabled.

Tail-calls elimination ensures constant stack usage by making sure that recursive functions return an accumulated value in their last call without any further evaluation. This can be often achieved by using the right built-in functions in the language e.g. foldl instead of foldr. The former is tail recursive and uses an accumulator which is returned in the last call.

Compiler optimisations can be enabled selectively to perform automatic source-to-source transformations, typically on intermediate language code. Haskell's GHC compiler provides several levels of optimisations, enabled with `-O`, `-O1`, `-O2`. The F# compiler has similar options using the `--optimize+` flag during compilation. This performs JIT, cross-module, tail-calls optimisations. There is also the possibility of specifying on which platform (e.g. x86 or x64) the generated code will run. Scala has the flag `-optimise`. However, in the current version the Scala compiler does not seem to perform any significant optimisations, based on our measurements of the code. Notably, the F# and Scala compilers do most optimisations without having to specify any flag (enabled by default), whereas the GHC Haskell compiler allows selective enabling of optimisations and provides several optimisation levels.

4.2.2 Haskell

We start with an initial sequential implementation of the Barnes-Hut algorithm in Haskell, where the expressiveness of the language helps to easily translate the advanced problem into functional code. Initially, the program is not well optimised and is unable to execute on a large number of input due to stack overflow. Several iterations of optimisation are required in order to produce an efficient sequential version (reported in [30]). These optimisations are:

Strictness annotation: Being a lazy language, some optimisations are specific to Haskell. For example, often it is not necessary to delay evaluation of values to avoid unnecessary thunking of computations. A number of ways to force evaluation is available in Haskell: using the `pseq` primitive, strict application function (`$!`) or simply the exclamation (`!`) from BangPatterns extension. These annotations are typically placed in data type definitions, to effect every usage of such data. All data types are defined with strict data fields consequently reducing the heap consumption and runtime. Additionally, the `UNPACK` pragma is used to refer to the values directly instead of pointers, thus removing one level of indirection and reducing memory consumption.

foldr/build: Another important optimisation in Haskell is foldr/build short cut fusion. This eliminates the intermediate data structures produced by a build followed by a foldr. The compiler can spot this specific sequence and automatically fuse the code.

4.2.3 F#

The F# version is a direct translation of the Haskell code and the changes are mainly syntactic In contrast to Haskell's lazy list, we use the default strict list in F#. Other versions using sequence and array are implemented and the results are given in the next section.

F# does not require any optimisations related to laziness as the language is strict by default. After translation from Haskell, the main optimisation involves manually merging fold/map in F# which is done by using foldr/map in Haskell. For e.g. in the following the operation done in the map is moved into the lambda function of fold.

```
List.fold g acc (List.map (fun x -> f x) xs)
-- becomes
List.fold (fun state x -> g state (f x)) acc xs
```

Inlining functions is another way of improving performance. The `inline` annotation indicates that a function definition should be embedded into any code which uses it.

4.2.4 Scala

The Scala version is largely based on the Haskell and the F# implementations and the main differences are in the syntax. This requires to wrap functions inside classes or singleton objects. The `val` keyword is used to indicate immutability. Similar to F#, Scala provides by default strict evaluation, using strict lists, and, thus, does not require any optimisations related to laziness. Following the initial implementation, which was largely translated from Haskell and F#, the two main sequential optimisations are:

Tail recursion optimisation: This generic optimisation technique seems to play a significant role towards achieving good performance in Scala in the case of the Barnes-Hut algorithm. Tail calls are not natively supported in the JVM, as opposed to .NET, which explains why such a source code transformation is more important than in F#. Although some tail-call optimisation was recently introduced to the Scala compiler, it is still quite basic and only able to convert simple recursive functions into loops during compile time. Quite handy, the annotation `@tailrec` can be used to check if a recursive function is tail-call optimised during compilation.

Unnecessary object initialisations removal: Object allocation in Scala is very light-weight, something which is very important as objects are an integral part of the language. Object initialisation, though, causes some additional performance overhead especially when used inside heavy numerical computations such as in the recursive `calcAccel` function, which is the main worker function in the Barnes-Hut code.

4.3 Parallel Implementations and Tuning

The main source of parallelism occurs in the acceleration calculation step as shown by time profiling the program in all languages. The tree construction stage is insignificant in terms of the overall percentage of time spent in it. Thus, we focus our efforts in parallelising the top level map function that uses the constructed tree and computes the acceleration for each body in the list. The computation of acceleration for each body is independent from the other bodies, which means no synchronisation locks are required.

4.3.1 Haskell

The strategies library provides a parallel map implementation `parMap` which is implemented using `parList`, a high-order composable strategy that applies a given evaluation degree to each element in the list in parallel. `parMap` is the starting point to introduce data parallelism in the code. Usually it will give good parallel performance if the list to which it is applied is not too big and the function applied to each list element does enough work to cover the overhead of creating a spark for each list item. With a large input size, `parMap` is inefficient as a spark is generated for each elements in the list structure resulting in more overheads than actual benefit of parallelism especially if the work is too fine-grained.

It is usually not a problem to create many sparks in GpH as it amounts to a pointer for each spark created only. However this may lead to too fine-grained parallelism and poor performance.

Parallel tuning: The right balance of spark creation to match the number of cores on the system is important in order to achieve good parallel performance. If too many sparks are created, they might not end up being taken for execution by the runtime; while too few of them may result in under exploitation of processing units.

We use strategic chunking as a method to control the number of sparks in Haskell. Explicit chunking and clustering are covered in more detail in [30]. Strategic chunking makes use of the high-order strategy from the library which performs the chunking implicitly. This involves using `parListChunk`, which takes an additional *chunksize* parameter, instead of `parList`. The chunk size specifies the size of the sublists and is calculated depending on the number of processor cores (using `numCapabilities` in GpH) and input size. This ensures that the work is properly balanced among the processors. The two lines of code below are all that is needed to make the parallel implementation scale.

```
chunksize = (length bs) `quot` (numCapabilities *
    4)
new_bs = map f bs `using` parListChunk chunksize
    rdeepseq
```

4.3.2 F#

We first use asynchronous workflow to implement a parallel map in F#. By marking the function application to each element in the list with the `async` keyword, the sequential map operation is made concurrently, with each function application not blocking each other. Adding Async.Parallel to the pipeline enables the function applications to run in parallel if multiple cores are used. Async.RunSynchronous waits to synchronise at the end.

```
let pmap_async f xs =
  seq { for x in xs -> async { return f x } }
  |> Async.Parallel
  |> Async.RunSynchronously
  |> Seq.toList
```

While asynchronous workflow is a fairly easy way to introduce parallelism and get initial speedup, it is also fairly intrusive and changes the code structure. If the main source of parallelism can be identified in one specific higher-order function, or skeleton, one would typical use tasks from the TPL library for independent operations. For instance, similar to the Haskell initial parMap implementation where a spark is created for each list element, we try creating a task for each list element using the task factory from TPL. This surely incurs overhead if the cost of creating a task for an element is higher than the cost of processing (applying a function to) the element.

```
let pmap_tpl_tasks f (xs:list<_>) =
    let createTask x = Task<_>.Factory.StartNew(
        fun () -> f x).Result
    let tasks = xs |> List.map createTask
    tasks

// chunking
let pmap_tpl_tasks_chunk f (xs:list<_>) =
    let chunks = chunksOf (xs.Length / (numProc *
        2)) xs
    let chunkTask chunk = Task<_>.Factory.
        StartNew(fun () ->
        List.map f chunk).Result
    let tasks = List.map chunkTask (chunks |> Seq
        .toList)
    tasks |> List.concat
```

The code extract shows explicit task creation for each list element in a naive parallel map implementation (comparable to parMap in Haskell). The intention is to compare the overhead of spark versus task creation in Haskell and F# respectively. As we expected this does not give good performance. Thus we use a chunking mechanism to try to limit the number of tasks created. The results are discussed in Section 5.

Using tasks directly is not a good fit for our data-parallel problem. Many higher-level constructs are provided in TPL to achieve a more declarative way of enabling data parallelism. These are typically implemented on top of tasks. PLINQ presents the best choice. It hides the details of task creation and management in its implementation and provides a nice, familiar interface to easily express

parallel queries. For example, doing an operation on each element in a list in parallel is enabled by simply marking the container *as parallel* which hints to the underlying system that the latter is to be processed in parallel i.e. converting it into a parallel query. PLINQ uses TPL tasks in the background and handles load balancing across the cores implicitly, though it also offers some limited control. In the PLINQ-style parallel map implementation, the Select actually performs a map operation on each element.

```
let pmap_plinq f (xs:list<_>) =
    xs.AsParallel()
      .Select(fun x -> f x) |> Seq.toList
```

Imperative style programming: The other main construct for data parallelism is Parallel.For/ForEach. However, this does not prove to be convenient with lists but is mostly useful with mutable arrays, where the action inside the loop is to update elements in the array. Implementing a different version of the algorithm that uses arrays enables us to use Parallel.For to introduce parallelism and thus examining the effect of inplace update.

```
let pmap_tpl_parfor f (xs:array<_>) =
    let new_xs = Array.zeroCreate xs.Length
    Parallel.For(0, xs.Length, (fun i ->
        new_xs.[i] <- f (xs.[i]) )) |> ignore
    new_xs
```

Alternatively, there already exists a parallel map in the Array.Parallel namespace which is a basic implementation and uses uses Parallel.For behind the scene.

```
let res = Array.Parallel.map f arr
```

Tuning: PLINQ and TPL provide some options for tuning, although we find that the default settings are usually sufficient.

Maximum degree of parallelism: A thread pool is used to schedule tasks and the number of threads can be controlled by using maximum degree of parallelism. This configuration specifies the maximum number of concurrently executing tasks.

Parallel.For and similar methods take an additional parameter specifying this parameter. In PLINQ, the parameter AsParallel().MaxDegreeOfParallelism can be set explicitly to the same effect. As an upper bound this parameter is used to restrict the amount of parallelism at one time in the execution, but does not ensure that the specified number is generated.

Chunking/Partitioning: We implement custom chunking to control the granularity of tasks created explicitly as above — to achieve similar effect as in Haskell.

There is also a Partitioner class with static methods to partition collections. It supports chunking partitioning with dynamic allocation, but also range partitioning with static allocation. A partitioner can be passed as argument to Parallel.For to specify a custom partitioning. Therefore, it is best used with arrays as it gives the intervals for each partitions.

4.3.3 Scala

Through the use of Parallel Collections in Scala, parallelisation is semi-explicit by using the keyword `par` to call a parallel version of the list which implements parallel operations.

```
nbody.par.map((b: Body) => new Body(b.mass,
    updatePos(b), updateVel(b))).seq
```

The method `par` is applied on the list of bodies to call `scala.collection.parallel.immutable.ParSeq`, which is the default parallel implementation of the list. When we subsequently apply the `map` function on the parallel list, the parallel map function is invoked on the list elements. In this way, it achieves a similar separation of coordination and computation as `parList` in GpH. Finally, we have to convert the result to a sequential list by applying the method `seq` on the results of the mapping calling `scala.collection.immutable.List`.

Parallel Collections requires small changes to the code to achieve initial speedup. The main disadvantage is that the framework does not currently provide much control over the parallelism. As an example, it is not possible to control how many threads are spawned or define the size of the underlying thread pool. Instead, these details are handled by the underlying implementation, which uses sophisticated work stealing and chunking techniques.

In a second approach, we implement parallel map skeletons using the `scala.actors.Futures` package from the Actors library. A Future abstracts over send and receive primitives and represents an object that is created to store a result that has not yet been computed. The result is computed concurrently at a later time and can be collected on demand. Our first skeleton is the classic parallel map function:

```
def pmap[T](f: T => T) (xs: List[T]): List[T] = {
  val tasks = xs.map((x: T) => Futures.future { f
      (x) })
  tasks.map(future => future.apply())
}
```

In `pmap` a `future` is explicitly created for each element in the given list, mapping the given function on the corresponding list element. The results of the parallel map are then returned to the user as the output of the skeleton.

The second skeleton we implement is a parallel map using chunking to explicitly control the granularity of the parallelism, directly corresponding to the initial Haskell implementation:

```
def chunk[T](xs: List[T], size: Int): List[List[T
    ]] = xs.isEmpty match {
  case true => List()
  case false =>
    val split = (xs.take(size), xs.drop(size))
    (split._1) +: chunk(split._2, size)
}

def pmap_chunk[T](f : T => T, size : Int) (xs:
    List[T]): List[T] = {
  val chunks = chunk(xs, size)
  val task_chunks = chunks.map((c: List[T]) =>
      Futures.future { c.map((x: T) => f(x)) })
  val tasks = task_chunks.map(future => future.
      apply())
  tasks.flatten
}
```

5. Results

5.1 Experimental Setup

Platforms: All three languages are supported both on Linux and Windows platforms either natively or through independent (open-source) implementations. This provides ground for comparison of the language implementations across the two platforms and discuss the results on each. Haskell's GHC implementation is cross-platform. GHC offers the option to compile code down to C and run on a standard C compiler. F#'s official Microsoft implementation is intended to run on the .NET Framework on machines running Windows. The open-source implementation of the runtime, Mono, is available to compile and run F# code under Linux. Scala runs on JVM, which has good support on both platforms. The following are the machines used for the experiments:

	Linux	Windows
Version	CentOS 5.8	XP
Architecture	64-bit	32-bit
CPU	Intel Xeon E5410 2.33GHz	Intel Core i7 860 2.80GHz
# cores	8	4 (8 HyperThreads)
RAM	7986 MB	3520 MB

Language Implementations: The following are the language implementations used with the corresponding version numbers on each platform:

	Linux	Windows
Haskell	GHC 7.4.1	GHC 7.4.1
F#	F# 2.0 / Mono 2.11.1	F# 2.0 / .NET 4.0
Scala	Scala 2.10 / JVM 1.7	Scala 2.10 / JVM 1.7

Input: All parallel measurements for the Barnes-Hut algorithm are taken using 80,000 bodies as the input and the execution is based on 1 iteration of the nbody simulation. We focus on a single iteration to assess the potential for parallelism in this application core for each language, rather than an application tuning exercise.

5.2 Baseline

We use the all-pairs implementations from the Computer Language Shootout website[1] as baseline for comparison. The implementations on the shootout webpage are highly-optimised by experts in each language community. However, the implementations are impure and they make heavy use of inplace updates and other unsafe constructs provided in the languages in order to get the best performance out of the implementation. This approach however destroys the possibility to add parallelism easily to the sequential code. Often the whole program would have to change in order to parallelise it. Therefore, we do not take these versions as starting points for our parallelisation. The original shootout runtimes on a Linux x64 Intel Q6600 machine are taken for 5 bodies and 50 million iterations. We take measurements on our machines using 16000 bodies and 1 iteration only, and we exclude the input generation and energy calculation times as we are interested in the main iteration and parallelising it. The runtimes are given in Table 1.

Table 1. Baseline (a) — language shootout results (all-pairs). The numbers in brackets are slow-downs w.r.t. the Haskell version.

	Linux (Original) 5 bodies, 50M iterations	Linux (16k bodies, 1 iteration)	Windows (16k bodies, 1 iteration)
Haskell	25.23 (1.00)	9.24 (1.00)	7.66 (1.00)
F#	41.36 (1.63)	9.37 (1.01)	4.88 (0.63)
Scala	23.47 (0.93)	5.51 (0.59)	14.25 (1.86)

Both the original runtimes and those taken on our Linux machine show that the Scala implementation is fastest, followed by Haskell, then F#. However, on Windows platform, interestingly, the slowest of the three, F#, performs the best. This highlights that the F# .NET implementation on that platform is very well-tuned and Microsoft technologies integrate well together. On the other hand, Scala goes from best to worst performance under Windows.

We also use our pure all-pairs implementations, which do not use destructive updates, as a baseline. The results are shown in Table 2, Haskell gives the best performance under Linux, and second best under Windows, though closely followed by F# under the same platform. Scala gives decent performance on both platforms. F# on Linux is very slow by factor of 8.5 compared to Windows for the pure baseline, as opposed to 1.9 for the impure baseline. We also observe that, as expected, the pure implementations are slower than the impure ones between 1.7 to 3.9 times. We intentionally not use impure features to enable easy parallelisation.

[1] `http://shootout.alioth.debian.org/`

Table 2. Baseline (b) — our pure all-pairs implementations (16k bodies, 1 iteration). The numbers in brackets are slow-downs w.r.t. the Haskell version.

	Linux	Windows
Haskell	20.77 (1.00)	15.55 (1.00)
F#	123.22 (5.93)	14.45 (0.92)
Scala	21.88 (1.05)	24.71 (1.58)

Comparing the runtimes from Tables 1 and 2, we can also assess the sequential performance of our all-pairs implementations, giving an estimate of the quality of our sequential code: 49.3% for Haskell, 33.8% for F#, 57.5% in Scala. Avoiding any use of impure features we lose some performance initially, but we gain ample opportunities for parallelisation in exchange. Later in this section we will show that selective usage of impure features *after* parallelisation can further enhance performance. We believe that efficiencies for the Barnes-Hut algorithm are similar, but in the absence of similarly tuned implementations we cannot make a direct comparison for this algorithm.

In the following subsections, we see how these figures relate to the pure Barnes-Hut implementations, where the algorithm is more complex than the all-pairs, on the two platforms. The discussions focus on 3 metrics: performance, programmability and pragmatic aspects of the languages.

5.3 Performance Evaluation

Tables 3 and 4 summarise the runtimes and speedups in each language on Linux and Windows respectively. As ancillary data, Table 5 shows the peak memory usage under the Windows platform, as observed by an external, OS-level task manager.

Table 5. Peak memory usage on a Windows machine

	Haskell	F#	Scala
Sequential	57 MB	32 MB	58 MB
Parallel (4 cores)	71 MB	36 MB	62 MB

We use the same data structure, in this case list, across the different implementations to have comparable results. The results highlight a number of interesting points.

5.3.1 Sequential Performance

Tables 3 and 4 show that Haskell gives the best sequential performance on both platforms. This has been made possible due to extensive sequential optimisations, using a range of techniques.

Most notably, the initial, naive Haskell version — without strict data fields — gives a runtime of 479.94s. By using strict data fields and `UNPACK` pragma, the runtime goes down to 33.02s, amounting to a sequential speedup of 14.5. Enabling foldr/build optimisations by code restructuring, as a GHC specific compiler optimisation, gives a further 23% reduction in runtime resulting in 25.28s. All results are measured under Linux. A detailed discussion of both implementations is given in [30].

The F# sequential runtime is slightly better than that of Haskell under Windows. This directly corresponds to the lower memory footprint of F# as shown in Table 5.

Table 3. Runtimes (in seconds) on Linux (8 cores)

	Haskell		F#				Scala		
	GpH		AsyncWork	TPL			ParColl	Actors	
# cores	parMap	parMapChunk		Tasks	Tasks/chunk	PLINQ		pmap	pmapchunk
seq	25.39	25.28	118.12	118.12	118.12	118.12	39.04	39.04	39.04
1	25.91	26.38	211.78	197.72	209.76	196.14	48.02	45.38	40.01
2	25.77	14.48	129.07	154.09	162.32	120.78	25.72	25.18	22.34
4	22.69	7.41	89.63	128.99	134.54	80.91	16.41	16.42	14.88
8	23.17	**4.50**	**70.41**	120.26	122.45	70.67	13.48	14.34	**13.26**

Table 4. Runtimes (in seconds) on Windows (4 cores plus hyperthreading)

	Haskell		F#				Scala		
	GpH		AsyncWork	TPL			ParColl	Actors	
# cores	parMap	parMapChunk		Tasks	Tasks/chunk	PLINQ		pmap	pmapchunk
seq	17.64	17.64	21.12	21.12	21.12	21.12	66.65	66.65	66.65
1	17.77	18.05	21.26	23.31	21.10	21.39	66.96	68.30	67.24
2	17.61	9.41	16.96	26.47	21.36	17.32	57.96	58.63	58.66
4	16.94	**6.80**	**10.18**	36.06	21.50	10.56	34.48	**33.64**	33.84
8 (HT)	*17.61*	*4.77*	*8.82*	*36.28*	*21.05*	*8.64*	*26.18*	*24.74*	*25.28*

The F# version is a direct translation from Haskell with some F# specific optimisations. Some optimisations native to Haskell e.g. strictness annotations, are not required in F#. Other program transformations, in particular merging fold and map operations thereby eliminating intermediate data structures, have to be done manually in F# and improve the runtime from 28.43 to 22.15s. Inlining of functions reduces the runtime by 5% to 21.12s.

Another general observation is that the Mono implementation of the .NET runtime, used under Linux, is not as well optimised as the corresponding Microsoft .NET implementation, used under Windows. Since the Mono project is now focusing on providing a .NET infrastructure on embedded systems, rather than focusing on high-performance computing, this is not surprising. However, this is one of the first systematic comparisons of both platforms for parallel computation. The difference in runtime is particularly remarkable, since the hardware used for running the F#/Mono instance was faster than the hardware for running F#/.NET.

The Scala version is the slowest one under Windows, but significantly better under Linux. This behaviour is consistent with the all-pairs baseline results in Tables 3 and 4. We attribute this difference mainly to the Scala compiler and the memory management overhead imposed by it. It has been reported elsewhere [27] that Scala makes heavy use of boxed types, resulting in a fairly high memory footprint, as shown in Table 5 where the peak memory consumption of the strict Scala implementation (58 MB) is even higher than the lazy Haskell implementation (57 MB). On the Windows machine the smaller amount of main memory will cause this high memory consumption to have a stronger impact on total runtime. Additionally, as a mixed paradigm language, Scala makes heavy use of objects, resulting in initialisation overhead. Under Linux, the initial Scala implementation, without the optimisations described in Section 4.2.4, gives a runtime of 55.44s. By using tail recursion optimisation, the runtime goes down to 45.48s (-18%). Removing unnecessary object instantiations helps to further improve the performance to 39.04s (-14%).

Using Lazy Data Structures: All 3 languages support lazy data structures and here we compare their sequential performance.

As reported above, the unoptimised Haskell version, with lazy data structure by default loses a factor of 14.5 performance.

The LazyList version in F#, as a direct comparison with this initial Haskell version, exhibits an increase in sequential runtimes from 118.12 to 604.19s (Linux), and from 21.12 to 81.68s (Windows), representing a factor of 5 and 4, respectively. The memory usage under Windows peaks at 115 MB (from 32 MB using the default strict list). This suggests that this library is not well-optimised for LazyList and fewer compiler optimisations, aiming at eliminating unnecessary laziness are applied. It is worth noting that this structure is available as part of the PowerPack, is not officially supported and is still under development.

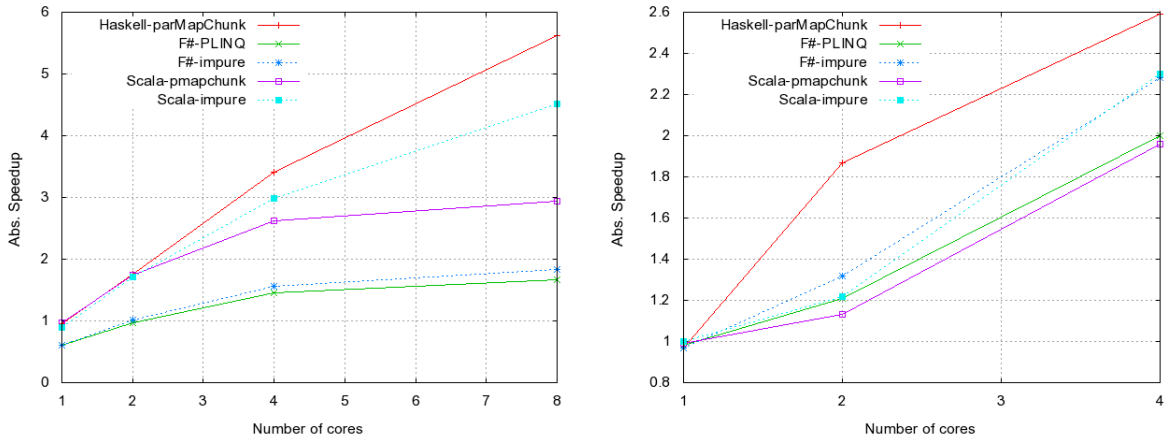
Similarly, in Scala, the use of streams as lazy structures in lieu of lists results in an increase in sequential runtimes from 39.04 to 89.35s (Linux), and from 66.65 to 92.24s (Windows). The memory usage under Windows peaks at 100 MB, which is a 72% increase from the 58 MB using the default strict list.

5.3.2 Parallel Performance

Figures 2(a) and 2(b) summarise the speedups obtained with the best versions of each language implementation on Linux and Windows, respectively. For F#, we use PLINQ speedups instead of Async Workflow as the differences are small and we want to compare *parallel* construct in the plots. Tables 3 and 4 elaborate on the runtimes of several versions with the best parallel runtimes in each language highlighted. Comparing the performance of the languages, Haskell displays the best speedups, up to 5.6 on 8 cores, and remains scalable. This is achieved through strategic chunking to improve thread granularity.

Since this is a data-parallel application, with limited scope for thread-subsumption, such explicit granularity control is crucial, as can be seen from the poor performance of the naive parMap implementation, which generates a spark for each list element. In terms of absolute parallel performance, the Haskell version is 3 times faster than the Scala version on an 8-core Linux machine, and 1.5 times faster than F# on an 4-core Windows machine.

Due to the poor performance of Mono, for F# the Windows version is the more interesting. Here the implementation achieves a respectable speedup of 2 on 4 cores, which increases to 2.4 when using hyperthreading. Notably, the highest-level PLINQ implementation is the best performing on this platform, although speedup in



(a) Linux (8 cores)

(b) Windows (4 cores)

Figure 2. Absolute speedups on the two platforms for the best versions in all languages and impure versions in F# and Scala.

itself is not as good as in the Haskell version. Interestingly, the heap consumption of F# is significantly lower, even with the unoptimised version where it remains the same, than that of the Haskell version (Table 5), but this does not translate into faster runtimes. We conjecture that this is mainly due to GHC performing more aggressive optimisations than F#. We note that the task-based implementations in effect result in a slow-down, mainly due to the high task management overhead, which is reduced when employing chunking. In contrast to the GpH version, F# tasks are mandatory, so this overhead is more pronounced than in GpH. The highest level PLINQ implementation, best suited for data-parallelism gives almost as good a result as the lower level asynchronous workflow implementation (-3.7% on 4 cores). Given the simplicity of the PLINQ code (see Section 3.2.1) this is a strong argument in favour of this abstraction mechanism for data-parallel code.

For the Scala version we focus on the better performing Linux version. All three parallel versions exhibit good speedups, although trailing the Haskell results. In this implementation, chunking has a far lower impact on performance compared to the Haskell version. Together with the good 1 processor performance this indicates very efficient task management for actor code in Scala. The highest level parallel collection implementation is within 1.7% of the 8-core performance, almost as good as a tuned actor implementation. The parallel performance of the pure version tails out for higher core counts, with an 8-core speedup of 2.9 using the Parallel Collections, but the impure implementation, discussed below, achieves a speedup of 4.5. This simplest version achieves almost as good a performance as the lower-level actor-based implementations.

Mutable Data Structures: The remaining results use impure language features, in particular mutable data structures, to further improve performance. The use of arrays in F# which update the body inplace gives only a small performance gain under Windows, a 1.7% decrease in sequential runtime from 21.12s to 20.76 (Table 6). Interestingly, the memory usage remains the same as using list. This might be due to the imprecision of using an external, OS-level tool to determine peak usage, as opposed to maximum residency. Under Linux, where the sequential runtime is already slower by a high factor, arrays give a 19% improvement from 118.12 to 95.33s in runtime. The main advantage of using arrays, though, are as mentioned earlier: the use of the built-in parallel map from Array.Parallel module, and Parallel.For with the default or custom partitioning. The default partitioner, which creates partitions based on the processor count and input size, exhibits very good parallel performance that does not improve significantly with a custom partitioner.

We also developed an impure functional implementation in Scala using inplace updates. In this case, the maximum residency indeed drops from 114.75 MB to 91.72 MB (20% reduction) on the Linux platform, as obtained by internal, JVM-level monitoring. This directly translates into faster sequential and parallel runtimes than the pure versions under Linux (see Table 7). In Scala this difference is quite remarkable, improving speedup to 4.51 (on 8 cores), as opposed to 2.9 in the pure version (Figure 2(a)). This improvement in parallel speedup is most likely due to the reduction in heap contention on this shared memory architecture, in this more memory efficient version. The impure Scala implementation also gains significantly better sequential performance on the Windows platform: 47.86s (28% runtime reduction from the pure Scala implementation) and a 4–cores parallel runtime of 20.78s (39% runtime reduction and 14% speedup improvement from the parallel pure implementation using `pmap_chunk`).

Table 6. F# runtime results (in seconds) using arrays on *Windows* (4 cores plus hyperthreading).

# cores	Best Pure	Array.Par.map	Parallel.For	
			default	partitioner
seq	21.12	20.76	20.76	20.76
1	21.39	21.56	21.2	21.35
2	17.32	15.83	15.66	16.05
4	10.56	**9.09**	**9.09**	9.32
8	*8.64*	*7.33*	*7.33*	***7.15***

Table 7. Runtime results (in seconds) for the impure Scala implementation on *Linux* (8 cores).

# cores	Best Pure	ParColl	pmap	pmapchunk
seq	39.04	32.81	32.81	32.81
1	40.01	36.92	34.86	36.09
2	22.34	17.66	22.05	19.14
4	14.88	11.23	12.55	10.96
8	13.26	8.47	8.57	**7.27**

5.4 Programmability

Due to the high-level nature of all three languages, introducing parallelism to a pure version is easy and often amounts only to using a suitable data-parallel skeleton, e.g. parMap instead of a map in the computational core of the application. In the impure versions, we start from a pure version to introduce parallelism and then add inplace updates selectively in such a way that does not require locks as the operations are independent.

In the Haskell version we use evaluation strategies, which separate the parallel code from main computation logic. In another paper [30] we compared this implementation with alternatives for introducing parallelism in Haskell, in particular with Eden [16] and the ParMonad [18], achieving similar performance. However, Haskell requires more parallel performance tuning to achieve good parallel performance. In particular, the naive `parMap` does not work well when used directly, because its implementation does not automatically introduce chunks. Instead the programmer has to choose the `parMapChunk` variant or, to tune performance further, can implement customised chunking to control thread granularity.

F# supports several paradigms of parallel programming at several different levels of abstraction, making it easier for programmers to make the transition from sequential to parallel programming. The preferred mechanism for data-parallelism is to use the SQL-like PLINQ language, which represents the highest level of abstraction, and still gives close to optimal performance. Scala offers parallelism support through two libraries. On a data structure level parallelism is very easily introduced through the keyword `par`, which calls a parallel version of a collection causing most subsequent functions applied on it to be executed in parallel. The lower-level Actors message passing model offers explicit messaging between actors, but also allows to build higher-level solutions on top of Futures for introducing parallelism.

Optimisations are important in getting good performance but most be chosen carefully to avoid sequentialisation. Due to laziness, Haskell requires some optimisations to work around laziness to get good sequential results. In F# and Scala, some optimisations are necessary and easily introduced with the functional style e.g. ensuring recursive functions make tail-calls and avoiding multiple traversal of data structures using function composition in function argument to a map. Several of these (manual) optimisations are performed automatically by GHC.

Other language features such as purity in each language helps to structure the code so that the main n-body simulation code is written in an entirely pure functional way. This provides a good separation between pure and impure code, for example, the main function which generates the input, and output the simulation result.

Both F# and Scala allow integration of object-oriented features with functional programming. Although, we tried to integrate some of the available object-oriented features (e.g., classes) in our F# and Scala implementations, we did not notice any difference in the sequential or parallel performance. That makes intuitive sense because object-orientation is mainly used to help towards large-scale software development, by using features such as polymorphism, inheritance and encapsulation. In a high performance computing application, though, we found that we did not really require these features to achieve good performance.

One of the main difficulties of Haskell is to understand the implications of laziness. It is hard to predict when expressions are evaluated, and to estimate how much work is involved. However, laziness allows us to make use of infinite data structures. F# and Scala on the other hand have strict evaluation by default. This makes it easier to reason about the program.

5.5 Pragmatics

Haskell comes with powerful tool support which is helpful in optimising both sequential and parallel algorithms. As an example, time and heap allocation profiling reports, both textual and visual, are useful in identifying the hot-spot of the execution and potential space leaks. Threadscope is a parallel visualisation tool particularly useful to see work distribution across the number of cores. Although F# comes with very good tool support, such as a profiler, unfortunately it is only available on the ultimate version of Microsoft's Visual Studio. Due to the lack of these tools, it was difficult for us to find out why the same code in Haskell performs badly in F#. Free tools are difficult to find, as most third party options are commercial. Scala is based on the JVM and, thus, enjoys a wide range of both monitoring and analysis tools, such as VisualVM, which we used as a JVM-level monitor.

6. Conclusion

In this paper we compare parallel implementations of the Barnes-Hut algorithm for solving the N-body problem, implemented in the purely functional language Haskell and in the modern multi-paradigm languages F# and Scala. We assess both programmability and parallel performance, when executing on a state-of-the-art multi-core. We use a number of alternative parallel programming constructs provided in each language, in particular GpH in Haskell, Asynchronous Workflows and TPL in F#, and Parallel Collections and Actors in Scala.

The type of parallelism in this application is data-oriented and to this end, we implemented a number of parallel map skeletons in F# and similarly in Scala. We achieve speedups up to 5.62 in Haskell (8 cores), 2.28 in F# (4 cores) and 4.51 in Scala (8 cores).

With a caveat that our observations are based on parallel variants of only one application, we draw the following conclusions:

- Across all languages, the version using the highest abstraction level also produced the (near) best runtimes.
- Providing first-class parallelism support in the language, through primitives rather than annotations or libraries, is important in the Haskell version in order to explicitly tune thread granularity, e.g. using higher-order functions in a chunking `parMap`.
- Careful (sequential) optimisation of the lazy Haskell version results in sequential performance, surpassing that of the strict F# and Scala versions.
- Aggressive, sequential code optimisations, using impure language features early on, seriously hamper the parallelisation of the code, as can be seen from the (sequential) implementations on the language shootout page, which are a poor starting point for parallelisation due to enforced sequentialisation on mutable data structures. However, selective use of impure features at the end of the parallelisation process can gain notable additional performance, demonstrated, e.g. in the Scala implementation.
- The poor performance of F# on Linux, due to a fairly low-performance .NET implementation provided by Mono, does not make F# a viable choice for parallelism on Linux at the moment.
- The additional expressive power provided by lower-level Actor-based code in Scala does not manage to improve performance significantly and therefore the extra programming effort is not justified in this example.

In comparison to other main-stream parallel languages, we found that all languages provide fairly high-level constructs for parallelism but the degree of control provided in them differs. For instance, Haskell allows initial parallelism to be easily specified,

as parallel versions of well-known higher-order functions. Since parallelism is provided through primitives, rather than annotations or libraries, the full power of the languages is available and user-customisable to tune parallel performance using chunking. On the other hand, F# and Scala confine most control of parallelism inside the runtime-system implementation, aiming for automatic management without any programmer input. Therefore tuning parallelism is more difficult for the expert parallel programmer in these languages.

Acknowledgments

We would like to thank our colleagues in the Dependable Systems Group of Heriot-Watt University for their helpful discussions and feedback. We also thank the contributors to the SICSA MultiCore challenge (`http://www.macs.hw.ac.uk/sicsawiki/index.php/MultiCoreChallenge`) for their comparative work on parallel n-body implementations.

References

[1] The Computer Language Benchmarks Game. Website `http://shootout.alioth.debian.org`, Feb. 2012.

[2] G. Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, 1986.

[3] J. Armstrong, R. Virding, and M. Williams. *Concurrent programming in ERLANG*. Prentice Hall, 1993.

[4] J. Barnes and P. Hut. A hierarchical o(n log n) force-calculation algorithm. *Nature*, 324:446–449, 1986.

[5] 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, 2010.

[6] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications*, 21(3):291–312, Aug. 2007.

[7] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of OOPSLA '05*, pages 519–538, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-031-0. doi: http://doi.acm.org/10.1145/1094811.1094852.

[8] M. Cole. *Algorithmic skeletons: structured management of parallel computation*. Pitman, 1989.

[9] G. Cong, S. Kodali, S. Krishnamoorthy, D. Lea, V. Saraswat, and T. Wen. Solving Large, Irregular Graph Problems Using Adaptive Work-Stealing. In *Proceedings of the 2008 37th International Conference on Parallel Processing*, ICPP '08, pages 536–545, 2008.

[10] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/1327452.1327492.

[11] M. Frigo, C. E. Leiserson, and K. 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. URL `http://supertech.csail.mit.edu/papers/cilk5.pdf`. Proceedings published ACM SIGPLAN Notices, Vol. 33, No. 5, May, 1998.

[12] P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci*, 410:202–220, 2009.

[13] D. Kranz, R. 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.

[14] D. Lea. A Java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, JAVA '00, pages 36–43, 2000.

[15] D. Leijen, W. Schulte, and S. Burckhardt. The design of a task parallel library. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 227–242. ACM, 2009.

[16] R. Loogen, Y. Ortega-mallén, and R. Peña marí. Parallel functional programming in eden. *J. Funct. Program.*, 15:431–475, May 2005. ISSN 0956-7968. doi: 10.1017/S0956796805005526. URL `http://dl.acm.org/citation.cfm?id=1067405.1067409`.

[17] D. K. Lowenthal and V. W. F. G. R. Andrews. Using Fine-grain Threads and Run-time Decision Making in Parallel Computing. *Journal of Parallel and Distributed Computing*, 37(1), 1996. URL `http://dx.doi.org/10.1006/jpdc.1996.0106`. Special issue on multithreading for multiprocessors.

[18] S. Marlow, S. Peyton Jones, and S. Singh. Runtime Support for Multicore Haskell. In *ICFP '09: Proceeding of the 14th ACM SIGPLAN International Conference on Functional Programming*, 2009.

[19] S. Marlow, P. Maier, H.-W. Loidl, M. K. Aswad, and P. W. Trinder. Seq no more: better strategies for parallel Haskell. In *Proceedings of the third ACM Haskell symposium on Haskell*, Haskell '10, pages 91–102, 2010.

[20] S. Marlow et al. Haskell 2010 language report. *Available online http://www. haskell.org/(May 2011)*, 2010.

[21] D. C. J. Matthews and M. Wenzel. Efficient Parallel Programming in Poly/ML and Isabelle/ML. In *DAMP10: Declarative Aspects of Multicore Programming*, Madrid, Spain, Nov. 2010.

[22] E. Mohr, D. Kranz, and R. 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. URL `ftp://cag.lcs.mit.edu/pub/papers/futures.ps.Z`.

[23] M. Odersky and A. Moors. Fighting bit Rot with Types (Experience Report: Scala Collections). In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2009)*, volume 4 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 427–451, 2009.

[24] M. Odersky, P. Altherr, V. Cremet, et al. An Overview of the Scala Programming Language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[25] A. Prokopec, P. Bagwell, T. Rompf, and M. Odersky. A generic parallel collection framework. In *Proceedings of the 17th international conference on Parallel processing - Volume Part II*, Euro-Par'11, pages 136–147, 2011.

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

[27] A. Sewe, M. Mezini, A. Sarimbekov, and W. Binder. Da capo con scala: design and analysis of a scala benchmark suite for the java virtual machine. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, pages 657–676. ACM, 2011.

[28] Sun. The Fortress Language. Talks and Posters. available at `http://research.sun.com/projects/plrg`.

[29] D. Syme, A. Granicz, A. Cisternino, and L. MyiLibrary. *Expert F#*. Apress, 2007.

[30] P. Totoo and H.-W. Loidl. Parallel Haskell Implementations of the n-body Problem. *Concurrency and Computation: Practice and Experience.*, 2012. Submitted.

[31] P. Trinder, H.-W. Loidl, and R. Pointon. Parallel and Distributed Haskells. *J. of Functional Programming*, 12(4&5):469–510, July 2002. URL `http://www.macs.hw.ac.uk/~dsg/gph/papers/ps/jfp01.ps.gz`.

[32] P. Trinder, K. Hammond, and H.-W. Loidl. *Encyclopedia of Parallel Computing*, chapter Parallel Functional Languages. Springer Verlag, 2011. ISBN 978-0-387-09844-9.

[33] P. W. Trinder, K. Hammond, H.-W. Loidl, and S. L. Peyton Jones. Algorithm + Strategy = Parallelism. *J. Funct. Program.*, 8:23–60, 1998.