INHERENTLY PARALLEL DATA STRUCTURES

By
Prabhat Totoo

Submitted in partial fulfilment of the requirements for the degree of

Master of Science
in
Advanced Internet Applications

Supervised by
Dr. Hans-Wolfgang Loidl

School of Mathematical and Computer Sciences
Heriot-Watt University
August 2011
Declaration

I confirm that this work submitted for assessment is my own and is expressed in my own words. Any uses made within it of the works of other authors in any form e.g., ideas, equations, figures, text, tables, programs etc are properly acknowledged at any point of their use. A list of the references employed is included.

(Prabhat Totoo)
Abstract

The classic design of efficient data structures has been strongly influenced by the sequential nature of processing them. Often linear data structures, such as lists or arrays, are used for high-performance computation, exploiting good data locality on algorithm level and good cache usage on system level.

With the advent of multi-core machines, data structures that do not force a sequential evaluation mechanism on the algorithms are highly desirable. They encourage the design of high-level data-oriented algorithms, specifying parallelism in a minimally intrusive way.

The purpose of this project is to develop a library of inherently parallel data structures, that is, data structures that are represented in a way that favours independent evaluation of sub-components and with implicitly parallel basic operations. The focus will be on standard textbook data structures and operations with a goal of exploiting moderate parallelism on multi-core machines. In the longer term, these data structures should be applied in large-scale, real-world applications, possibly developing specialised data structures for the concrete application domain.
Acknowledgements

Firstly, I wish to thank my supervisor, Dr. Hans-Wolfgang Loidl, for his assistance throughout this project. His guidance has been of tremendous help to me in building a good understanding of what was a rather new research area to me. I really appreciate his willingness to help at all stages of the project and his technical insights gave me the direction I needed in completing this work.

I also wish to thank Thomas Horstmeyer from Philipp University of Marburg who built the Eden executable on the department’s Beowulf cluster. Mustafa Aswad from the department for his initial help with Simon Marlow’s code. Chris Okasaki for pointing me to his paper and helping out choosing the right representation for list. I am also grateful to my programme coordinator for his advices and all my lecturers who have been of invaluable help from the beginning of my course. It has been an enriching experience to me due to their teaching. I also thank my fellow MSc students for their views on course related issues and in general.

Last but not least, a thought to my family back in Mauritius and Koosum for their constant support, patience and love.
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>iii</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Problem</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Aims and objectives</td>
<td>1</td>
</tr>
<tr>
<td>1.3 Professional, legal and ethical issues</td>
<td>2</td>
</tr>
<tr>
<td>1.4 Structure</td>
<td>3</td>
</tr>
<tr>
<td>2 Literature review</td>
<td>5</td>
</tr>
<tr>
<td>2.1 Parallelism background</td>
<td>5</td>
</tr>
<tr>
<td>2.2 Parallel machines classification</td>
<td>5</td>
</tr>
<tr>
<td>2.3 Classifications of parallel programming</td>
<td>6</td>
</tr>
<tr>
<td>2.3.1 Memory model</td>
<td>6</td>
</tr>
<tr>
<td>2.3.2 Explicitness</td>
<td>7</td>
</tr>
<tr>
<td>2.3.3 Problem decomposition</td>
<td>7</td>
</tr>
<tr>
<td>2.3.4 Embedding parallel constructs</td>
<td>8</td>
</tr>
<tr>
<td>2.4 Parallel programming languages</td>
<td>8</td>
</tr>
<tr>
<td>2.4.1 High Performance Fortran</td>
<td>8</td>
</tr>
<tr>
<td>2.4.2 C+MPI</td>
<td>9</td>
</tr>
<tr>
<td>2.4.3 C+OpenMP</td>
<td>9</td>
</tr>
<tr>
<td>2.5 New generation languages</td>
<td>9</td>
</tr>
<tr>
<td>2.6 Functional programming and higher-order parallelism</td>
<td>10</td>
</tr>
<tr>
<td>2.6.1 Haskell</td>
<td>10</td>
</tr>
<tr>
<td>2.6.2 GpH: Glasgow parallel Haskell</td>
<td>11</td>
</tr>
<tr>
<td>2.7 Parallel program design</td>
<td>12</td>
</tr>
<tr>
<td>2.8 Concurrent libraries</td>
<td>13</td>
</tr>
<tr>
<td>2.8.1 Java concurrent collections</td>
<td>13</td>
</tr>
<tr>
<td>2.8.2 .Net concurrent collections</td>
<td>14</td>
</tr>
<tr>
<td>2.8.3 Intel concurrent collections</td>
<td>15</td>
</tr>
</tbody>
</table>
2.9 Patterns for parallel programming ........................................... 15
  2.9.1 Algorithmic skeletons .................................................. 15
  2.9.2 MapReduce ................................................................. 16
  2.9.3 Intel’s TBB ................................................................. 17
  2.9.4 Microsoft’s TPL .......................................................... 17
2.10 Data structures and algorithms of interest .................................. 18
  2.10.1 Lists ........................................................................... 18
  2.10.2 Algorithms and structures for the n-body problem .................... 19
2.11 Summary ............................................................................ 20
3 List data structure .................................................................... 21
  3.1 Standard list ....................................................................... 21
    3.1.1 Alternative representation ............................................ 22
  3.2 RAList implementation .......................................................... 24
    3.2.1 Data type definition ...................................................... 25
    3.2.2 ParTree ....................................................................... 25
    3.2.3 Operations on RAList .................................................. 26
  3.3 Benchmark applications ....................................................... 29
  3.4 Results and evaluation ......................................................... 30
4 Algorithms and structures for the n-body problem ......................... 37
  4.1 The problem description ....................................................... 37
    4.1.1 Solving methods .......................................................... 37
  4.2 Implementation .................................................................... 40
    4.2.1 Approach .................................................................... 40
    4.2.2 Sequential algorithm ................................................... 41
    4.2.3 Optimisation ............................................................... 43
    4.2.4 Profiling ..................................................................... 45
    4.2.5 Parallel implementation .............................................. 48
    4.2.6 Performance tuning ..................................................... 50
  4.3 Related implementations ...................................................... 53
    4.3.1 Eden ........................................................................ 53
    4.3.2 Eval and Par monads .................................................... 54
  4.4 Results and evaluation .......................................................... 54
5 Conclusion ............................................................................ 61
  5.1 Summary ............................................................................ 61
  5.2 Evaluation .......................................................................... 61
  5.3 Contribution ....................................................................... 62
5.4 Further work ................................................................. 63

A Source listings .......................................................... 69
  A.1 List data structure .................................................. 69
    A.1.1 RAList module source ........................................ 69
    A.1.2 ParTree module source ....................................... 79
    A.1.3 StdList module source ....................................... 82
  A.2 N-body problem algorithms ...................................... 83
    A.2.1 Full all-pairs algorithm .................................... 83
    A.2.2 Triangular algorithm ....................................... 86
    A.2.3 Barnes-Hut algorithm ....................................... 89
# List of Figures

2.1 Alternative representations of linked list ........................................ 19
2.2 Quadtree with 3 levels ............................................................... 20

3.1 Haskell list representation ............................................................ 21
3.2 Append tree .................................................................................... 23
3.3 Random-access list ......................................................................... 25
3.4 ParTree functions ........................................................................... 26
3.5 Structure construction cost ............................................................... 31
3.6 Sequential performance measurements .......................................... 31
3.7 Parallel map and operations measurements .................................... 32
3.8 RAList and standard list parallel map runtime & speedup ................. 33
3.9 RAList and standard list parallel operations runtime ....................... 34
3.10 RAList and standard list parallel operations speedup ...................... 35

4.1 2D bounding box ............................................................................. 43
4.2 Seq. Profiling .................................................................................. 47
4.3 Threadscope (1024 bodies, 1 step) .................................................. 51
4.4 Threadscope (1024 bodies, 20 steps) ............................................... 52
4.5 Standard error summary ................................................................. 53
4.6 Allpairs measurements ................................................................. 55
4.7 Allpairs runtime and speedup graphs - 16k bodies, 1 time step .......... 56
4.8 Triangular vs Allpairs runtime and speedup graphs - 1024 bodies, 20 time steps . 57
4.9 Barnes-Hut measurements .............................................................. 58
4.10 Barnes-Hut runtime and speedup graphs - 16k bodies, 20 time steps .... 59
Chapter 1

Introduction

1.1 Problem

We used to depend on increasing CPU speed to make our programs run faster. With many
issues such as physical hardware limits hindering this trend, the focus has now switched to
increasing number of cores in a machine. Indeed, if we want to decrease the runtime of our
program, we have to exploit multi-cores which are readily available in our desktops and laptops.
For example, dual- and quad-core processors are very common now. This trend has only started,
with 10's of cores available on the hardware in the not-too-distant future.

Most traditional data structures and algorithms are written for single-processor machines.
Sequential code does not meet performance goals and so programs have to be written to exploit
multi-cores. Faster programs need parallel programming but it still remains a challenge for the
programmer to write parallel programs efficiently with all the difficulties associated to it. It
is a burden to the programmer to specify how parallel computations need to be implemented.
This task can be relieved if programmers are provided with appropriate libraries that takes
care of some inherent parallelism in it, or having high-level constructs that facilitates the job
of programmers to write efficient parallel algorithms.

Data structures e.g. lists are reusable components used everyday by programmers. Many
of these structures are sequential but can be parallelised to improve the performance of a pro-
gram. Shavit argues that current data structures will disappear and be replaced by concurrent
constructs. “Multicore processors are about to revolutionize the way we design and use data
structures”[35].

1.2 Aims and objectives

The main aims of the project can be broken into two parts.

Firstly, we investigate conventional data structures and try to see how these can be adapted
to exploit multi-core platforms. We cover the design, implementation and assessment of a parallel data structure library in a high-level language. The idea is to get parallelism for free from using operations from the library. For this aim, we look at the commonly used list data structure and possible alternative underlying representations of such structure in order to expose parallelism.

The second part of the project investigates the n-body problem which represents an important class of parallel algorithms covering applications like simulation. The focus is more on algorithm to solve the problem with huge input that would otherwise not be feasible to do sequentially.

We set the following as a checklist to compare we have been successful in implementing most of them at the end of the project:

1. **Data structure focus**
   
   - Survey a range of data structures for their use in parallel applications, and select a subset as the scope of interest.
   
   - Investigate possible inherently parallel representations and implementations of these data structures.
   
   - Implement the data structures and common operations.
   
   - Assess the performance of the parallel libraries on a range of benchmark applications.

2. **Algorithm focus**
   
   - Investigate algorithms for a problem domain and possible implementations.
   
   - Implement sequential algorithms to solve the problem for small input set.
   
   - Parallelise and tune the performance of the algorithms to obtain better runtime.
   
   - Compare results with related implementations in the area.

### 1.3 Professional, legal and ethical issues

All the tools, compilers and libraries which are relevant and used are published in open source format. We do not require any license or permission to use them. A large part of the project will be based on previous research carried out within the department in the area of parallel and functional programming, most notably parallel strategies. There is also no particular ethical issues as no people or questionnaires are involved. Performance evaluation will be performed on the parallel data structures and algorithms implemented.
1.4 Structure

The content of the dissertation is structured as follows:

Chapter 1: Introduction gives an overview of the problem we attempt to solve and sets our aims.

Chapter 2: Literature review investigates the current state in parallel computing and towards the end it covers the data structures of interest.

Chapter 3: List data structure sees the implementation of a new list library with parallel operations.

Chapter 4: Algorithms and structure for the n-body problem looks at advanced structures and algorithms in solving a real-life application in parallel.

Chapter 5: Conclusion reflects on what have been achieved and future works.

Appendices: List of final versions of source code and any other additional materials referred in the main text.
Chapter 2

Literature review

2.1 Parallelism background

Before the introduction of multi-core machines, programmers had to write software for serial computation on single-CPU machines. Concurrency was achieved by breaking a problem into independent processes or threads which were executed simultaneously in an interleaved fashion on a single processor. This was not aimed at improving performance but rather to allow concurrent execution of instructions of different threads.

When problems are large or complex, it becomes impractical or impossible to solve them on a single computer especially given limited computer memory. Parallelism seeks to improve performance mainly in terms of runtime using multiple processors that are at our disposal. These compute resources may include a single computer with multiple processors, an arbitrary number of computers connected by a network, or a combination of both. The problem is broken into discrete parts than can be solved simultaneously on different processors. In theory, making more resources available to execute tasks or process data in parallel will shorten the time to completion with potential cost savings. However, in certain cases, parallelism may introduce additional overheads thus making parallelisation useless in its sense. It is therefore important to make the right decisions when parallelising a program. It involves a number of tasks and is harder than sequential coding.

2.2 Parallel machines classification

Parallel machines are usually classified according to Flynn’s Taxonomy [30] which is based upon the number of instruction and data streams available in the architecture.

Single instruction, single data (SISD) classification is equivalent to an entirely sequential program.

Single instruction, multiple data (SIMD) describes computers with multiple processing ele-
ments that perform the same operation on multiple data simultaneously.

Multiple instruction, single data (MISD) involves multiple processing units performing different operations on the same data.

Multiple instruction, multiple data (MIMD) involves processing elements each with its own stream of instructions operating on its own data. The vast majority of modern parallel systems fit into this category which are typically decomposed according to memory organisation: shared memory, distributed memory, or a hybrid architecture (combination of distributed-shared memory).

### 2.3 Classifications of parallel programming

While the previous section looks at hardware characteristics of parallel computers, at a programming level, there are several approaches to parallelism. It is desirable to have architecture-independent implementations of parallelism but often parallel programs are written for the targeted architecture. Parallel programming can be roughly classified according to the following four different criteria which are used in categorising the languages discussed later in this chapter.

#### 2.3.1 Memory model

The memory model characterises access to all memory locations.

**Shared memory model:** all processes share a single address space and communicate with each other by writing and reading shared variables. Changes in a memory location effected by one processor are visible to all other processors. Communication is fast but this model does not scale very well. As the number of processors increases, contention rises which becomes a bottleneck.

It is relatively easy to program since all processors share a single view of data. The programmer is responsible for correct access of global memory. Various mechanisms such as locks/semaphores are used to control access to the shared memory. There is no need to specify explicit communication of data between tasks, this is achieved via memory reads/writes (one-way communication) instead of send/receive operations.

**Distributed memory model:** processors have their own local memory. Memory addresses in one processor do not map to another processor, so there is no concept of global address space across all processors. Distributed memory systems require a communication network to connect inter-processor memory. Data has to be distributed over memories and it can be difficult to map existing data structures to this memory organisation.

This model uses message passing to send and receive messages between tasks. This requires cooperative operations (two-way communication) to be performed by each process e.g. a send
operation must have a matching receive operation. Distributed memory model can be organised in a virtual shared memory setup by introducing a layer of abstraction. The programmer then perceives this as a shared memory model. This variation has the characteristics of the PGAS model.

**PGAS memory model:** Partitioned Global Address Space (PGAS) supports a shared namespace, like shared memory. It supports a strong sense of ownership and locality. Each variable is stored in a particular memory segment. Tasks can access any visible variable, local or remote. Local variables are cheaper to access than remote ones. PGAS retains many of the downsides of shared memory.

### 2.3.2 Explicitness

Parallelism can be achieved either implicitly or explicitly.

**Implicit Parallelism:** also known as auto-parallelisation in which the compiler automatically exploits the parallelism inherent to the computations expressed by some of the language’s constructs. A pure implicitly parallel language does not need special directives, operators or functions to enable parallel execution. However, this is hard to implement. Implicit parallelism reduces the control that the programmer has over the parallel execution of the program which results in less efficient parallelism. It typically looks for parallelism in loops by checking that each iteration is independent of the next.

**Explicit Parallelism:** the programmer nominates program components for parallel execution. Parallelisation involves several steps, such as decomposition of tasks, mapping tasks to processing elements and managing communication of data between processing elements. The programmer has more control over the parallel execution which helps in producing more efficient code.

### 2.3.3 Problem decomposition

A problem can usually be split into parallel components which are data- or task-oriented.

**Data parallelism:** focuses on distributing the data across different parallel computing nodes. Often this involves splitting a data structure into equal size according to the available number of processors then each processor running the same function on their portion of data. The output from each processor is then aggregated to get the final result. The operation is similar to applying the same function to each item in a list in parallel.

Example: *sum of squares*

\[
f \{1,2,3\} = \text{sum} \left( \text{map square} \ (1,2,3) \right) = \text{sum}(\{1,4,9\}) = 14
\]

**Task parallelism:** focuses on distributing execution processes (threads) across different parallel computing nodes. This is different from data parallelism in the sense that each compute
node gets a different function to apply to its local data. Those functions can be applied in parallel and then composed to produce the final outcome.

Example: \textit{sum of fibonacci and euler functions}

\begin{verbatim}
sumFibEuler = fib o sumEuler
\end{verbatim}

2.3.4 Embedding parallel constructs

Parallelism can be embedded into the language in the following ways.

- \textit{Using language primitives} the user does not have to learn a whole new language. Parallelism is language dependant.

- \textit{Compiler directives} for example using C pragmas to provide the compiler with additional information on how execution should proceed.

- \textit{Annotations} to specify behaviour of the program during runtime e.g. in GpH.

- \textit{Separate coordination language} language deals with coordination aspect of parallelism.

- \textit{Libraries} use language and platform independent libraries.

2.4 Parallel programming languages

Concurrent programming languages, libraries, APIs, and parallel programming models (such as Algorithmic Skeletons [13]) have been created for programming parallel computers. These can generally be divided into classes based on the assumptions they make about the underlying memory architecture - shared memory, distributed memory, or distributed-shared memory. Shared memory programming languages communicate by manipulating shared memory variables. Distributed memory programming languages use message passing to send data between processes. POSIX Threads and OpenMP [4] are two of the most widely used shared memory APIs, whereas Message Passing Interface (MPI) [28] is the most widely used message-passing system API.

2.4.1 High Performance Fortran

High Performance Fortran (HPF) [6] provides a portable syntax for expressing data-parallel computations in Fortran. HPF uses compiler directives to distribute a single array computation over multiple processors. HPF does not have support for task parallelism and irregular data distribution.

\begin{center}
\begin{tabular}{|l|}
\hline
Classifications \\
Level of abstraction: high level \\
Memory model: shared, distributed \\
Explicitness: implicit \\
Type: data parallel \\
Embedding: compiler directives \\
\hline
\end{tabular}
\end{center}
and more complex data structures in general. New standard expects to consider these missing features but HPF still requires a complex compiler to provide a good implementation.

2.5 NEW GENERATION LANGUAGES

The Message Passing Interface (MPI) [1] is the standard programming environment for distributed memory parallel computers. It provides a set of library routines for process management, message passing and some collective communication operations. MPI programs can be difficult to write because the programmer is responsible for data distribution and explicit interprocess communicating using messages.

2.4.2 C+MPI

Classifications

- **Level of abstraction:** low level
- **Memory model:** shared, distributed
- **Explicitness:** explicit
- **Type:** data, task parallel
- **Embedding:** library

The Message Passing Interface (MPI) [1] is the standard programming environment for distributed memory parallel computers. It provides a set of library routines for process management, message passing and some collective communication operations. MPI programs can be difficult to write because the programmer is responsible for data distribution and explicit interprocess communicating using messages.

2.4.3 C+OpenMP

Classifications

- **Level of abstraction:** mid level
- **Memory model:** shared
- **Explicitness:** explicit
- **Type:** data, task parallel
- **Embedding:** pragmas

OpenMP [5] is a set of language extensions implemented as compiler directives. It is supported on the most widely used native programming languages such as Fortran, C and C++. OpenMP offers a common specification that lets software programmers easily design new parallel applications or parallelise existing sequential applications to take advantage of multi-core systems configured with shared memory.

By adding a compiler directive around a loop, for example, the compiler can be instructed to generate code to execute the iterations of the loop in parallel. The compiler takes care of most of the details of thread creation and management [29].

However, both C+MPI and C+OpenMP provide low- to mid-level constructs to coordinate the parallel execution ("goto" of parallel programming). They do not help structuring the parallel programming.

2.5 New generation languages

A new generation of parallel programming languages strives to provide a higher level of expression [9]. Languages such as X10 [10], Fortress [36] and Chapel [9] 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 [11]. PGAS (Partitioned Global Address Space) is the basis of these new parallel languages which adopt high-level language concepts, pioneered by parallel functional languages:
• Static type systems play a central role in all of these languages. X10 even uses dependent
types in order to express constraints on the location of computations [10]. Fortress,
designed as a “secure Fortran,” puts an emphasis on the security gained from a static
type system [36].

• Higher-order functions are successfully used in several parallel libraries, such as Intel’s
thread library [15] or in Google’s map-reduce framework [17].

• Declarative or rule-based computation, with its inherently decentralised nature of compu-
tation, has been identified as one promising direction for (semi-) automatically exploiting

• Generic programming offers the potential of code-reuse and is used for example in Chapel [9].

Our approach to parallel functional language design supports these concepts and takes an even
more radical high-level approach with almost all details of the parallel coordination delegated
to the parallel runtime environment. The key technology for the efficient, parallel execution
is an architecture-transparent runtime-environment, which uses an abstract machine model in
deciding about the generation, distribution and coordination of parallel computations.

2.6 Functional programming and higher-order parallelism

Parallel programming is inherently harder than sequential programming as the programmer
not only must describe what to compute in the form of correct algorithm, but also how to do
effective parallel coordination [24].

2.6.1 Haskell

Functional programming languages represent potential advantages for prototyping and develop-
ing parallel programs. They provide a powerful abstraction over computation and coordina-
tion; eliminate dependencies; and provide an architecture-independent style of parallelism [24].
Declarative languages e.g. Haskell [21] do not operate on a shared program state, and therefore
provide a high degree of inherent parallelism.

Like other modern functional languages e.g. SML or Scheme, Haskell [39] includes advanced
features:

• Good abstraction mechanism.

• Sophisticated type system, with type inference.

• Pattern matching.

• Higher-order functions.
Haskell frees the programmer from specifying low-level coordination details. The key feature of Haskell as a pure functional language is referential transparency. As defined by Stoy (1977), the only thing that matters about an expression is its value, and any subexpression can be replaced by any other equal in value. Moreover, the value of an expression is, within certain limits, the same wherever it occurs. This implies that two expressions are equal if they have the same value e.g. \( \sin(6) = \sin(1+5) \). An expression can be replaced with its value without changing the program (in other words, yielding a program that has the same effects and output on the same input). Referential transparency makes it possible for a programmer not to specify an execution order. Non-strictness allows for lazy evaluation where an expression is only evaluated when and if it is needed. Haskell’s purity greatly simplifies reasoning about parallel programs.

**Haskell’s Parallel Extension** Engineering a parallel program entails specifying computation i.e. a correct, efficient algorithm, and coordination. The coordination aspects involve partitioning, synchronisation, load management, communication, etc which represent a burden to the programmer. Many approaches including skeletons (described later) and parallelising compilers exist. GpH aims to provide a high level parallel programming model.

### 2.6.2 GpH: Glasgow parallel Haskell

GpH is an extension of Haskell which simplifies parallel programming by requiring the programmer to specify only a few key aspects of parallel programming. It provides thread-based semi-explicit parallelism. The language implementation hides the details and automatically manage the rest. The coordination aspects involve using the two primitives: `par` and `pseq` [25, 38, 26].

`par` indicates that the first argument may be executed in parallel. The runtime system will decide if a spark is created to perform execution in parallel. `pseq` causes the first argument to be evaluated (to WHNF - weak head normal form) before the second.

\[
\begin{align*}
par &: a \rightarrow b \rightarrow b & \text{-- parallel composition} \\
pseq &: a \rightarrow b \rightarrow b & \text{-- sequential composition}
\end{align*}
\]

Basic parallelisation can be specified using these two primitives. All the programmer has to do is to expose the parallelism by annotating expressions that can be evaluated in parallel, without having to control the details of parallel execution. The following example shows the calculation of Fibonacci numbers taken from [26]:

\[
\begin{align*}
nfib :: Int \rightarrow Int \\
nfib n | n <= 1 &= 1 \\
| otherwise &= x \ `par` (y `pseq` x + y)
\end{align*}
\]
where \( x = \text{nfib} \ (n-1) \)
\[ y = \text{nfib} \ (n-2) \]

The example shows how easy it is to reason about a program written in functional language. The parallelism is also easily exposed. The same program in an imperative language like C would normal require more than double the number of lines of code as the above.

**Evaluation Strategies**

Strategies provide a better way to specify parallelism in which the coordination and computation concerns are separated. Evaluation strategies raise the level of abstraction over \texttt{par} and \texttt{pseq}. The \texttt{using} construct is used to apply a strategy to an expression, for example:

\[
\text{expr} \ 'using' \ \text{strat}
\]

Evaluation degree specifies the degree to which an expression should be evaluated e.g. WHNF (Weak Head Normal Form) or normal formal (completely evaluated). Evaluation order specifies in which order evaluation should proceed. This is particularly important in specifying parallelism in an algorithm.

Simple strategies are defined in the \texttt{Control.DeepSeq} and \texttt{Control.Parallel.Strategies} modules e.g. \texttt{r0} performs no reduction at all, used to evaluate only the first element but not the second of a pair. \texttt{rseq} reduces its argument to WHNF.

\[
\texttt{-- performs no evaluation.}
\texttt{r0 :: Strategy a}
\texttt{r0 x = Done}
\texttt{-- evaluates its argument to weak head normal form.}
\texttt{rseq :: Strategy a}
\texttt{rseq x = x 'seq' Done}
\]

The following uses the \texttt{rdeepseq} strategy which fully evaluates its argument to specify data-oriented parallelism over a list.

\[
\text{map f [12 .. 30] 'using' parList rdeepseq}
\]

The system will spark evaluation of each item in the list in parallel. It is up to the runtime to pick up those sparks and transform into threads of execution to run on different processors. \texttt{parList} is an example of strategy composition.

### 2.7 Parallel program design

Designing parallel programs is hard. We can benefit by following a methodical approach. Foster (1995) proposed four main steps to facilitate the design [19]. The approach involves identifying parallel parts of a program, partitioning a problem into components, controlling granularity of the components and mapping them to the compute resources available.
2.8. CONCURRENT LIBRARIES

**Partitioning:** identify parallel tasks - the more the better.

Two methods for doing so can be used:

1) data or domain decomposition: involves dividing the data and operating upon the divided data concurrently. This can be applied to an array or list data structure, for example.
2) functional decomposition: divides the program into independent functions (computations) and executes the functions concurrently.

It is important to make sure that the number of tasks scale with the problem size, and to avoid redundant computation/storage. The tasks should also be of comparable size in order to facilitate load balancing.

**Communication:** identify channels - the less the better.

Communication is more costly than computation, and we should aim to reduce it. Local communication is preferred over global communication. Good design requires that all tasks perform about the same number of communication operations. If not, try to distribute communication operations more equitably. Communication can be made to proceed concurrently by using divide-and-conquer.

**Agglomeration:** combine tasks - to improve performance or reduce dev cost.

Scalability should be maintained while:

1) increasing locality. Areas of the problem that use the same data can be grouped together and executed sequentially to reduce communication.
2) and decreasing granularity. Granularity is the ratio of computation to communication. It is possible to vary the granularity to increase computation or reduce communication.

**Mapping:** assign tasks to processor - maximise utilisation and minimise communication.

This step is not required if the number of tasks is the same as the number of processors available. Two methods of mapping can be used: static task allocation for regular computation cost or dynamic task allocation for irregular computation cost.

### 2.8 Concurrent libraries

With the advent of multi-core processors, many known data structures have concurrent versions that allow multiple threads to access the data structure simultaneously in a *thread-safe* way.

#### 2.8.1 Java concurrent collections

In Java, the Collections Framework greatly simplifies the organisation and manipulation of in-memory data by providing implementations of commonly used data structures.

Similarly its Concurrency Utilities [7, 37] aims to simplify the development of concurrent classes by providing implementations of building blocks commonly used in concurrent designs. The Concurrency Utilities include a high-performance, flexible thread pool; a framework for
asynchronous execution of tasks; a host of collection classes optimized for concurrent access; synchronization utilities such as counting semaphores; atomic variables; locks; and condition variables.

Several new Collections classes have been added, including the new Queue and BlockingQueue interfaces, and high-performance, concurrent implementations of Map, List, and Queue. For example, ArrayList implementation is not synchronized in the regular collection package. If multiple threads access an ArrayList instance concurrently, and at least one of the threads modifies the list structurally, it must be synchronized externally. A thread-safe variant of ArrayList (CopyOnWriteArrayList) in which all mutative operations (add, set, and so on) are implemented by making a fresh copy of the underlying array is available in the concurrent libraries. This is ordinarily too costly, but may be more efficient than alternatives when traversal operations vastly outnumber mutations, and is useful when you cannot or do not want to synchronize traversals, yet need to preclude interference among concurrent threads.

The focus of this library is concurrency but not parallelism. We seek to implement data structures that offer parallelism for free. The Haskell-based library, we will provide a parallel implementation of, for example, map which applies a function \( f \) in parallel to all list elements.

\[
\text{map } f \left[1, 2, 3\right] \Rightarrow \left[2, 3, 4\right] \quad \text{-- } f \text{ is an increment function}
\]

An internal representation of the list as an append-tree would enable divide-and-conquer parallelism.

### 2.8.2 .Net concurrent collections

The .NET Framework version 4 introduces several new types that are useful in parallel programming, including a set of concurrent collection classes, lightweight synchronization primitives, and types for lazy initialization.

The collection classes in the System.Collections.Concurrent namespace provide thread-safe add and remove operations that avoid locks to avoid overhead wherever possible and use fine-grained locking where locks are necessary. Unlike collections that were introduced in the .NET Framework versions 1.0 and 2.0, a concurrent collection class does not require user code to take any locks (completely hidden in the library) when it accesses items. The concurrent collection classes can significantly improve performance over types such as System.Collections.ArrayList (with user-implemented locking) in scenarios where multiple threads add and remove items from a collection [31, 32].

Additionally, C# provides several lower-level coordination mechanisms for parallel programming. In a shared-memory model, the user gets full control over the parallel code by specifying explicit threads with synchronisation via locks, critical regions, etc. But orchestrating the parallel threads is tricky and error prone. In a distributed-memory setting, threads use
message-passing library to communicate by explicitly sending messages between the compute nodes. The programmer has to write code for (un)serialising data sent between machines. The difficulty with threads are the same.

C# supports both data and task parallelism in its Parallel Extensions library which we will discuss shortly.

2.8.3 Intel concurrent collections

Intel Concurrent Collections is a new parallel programming model for mainstream programmers that differs from other approaches. The programmer does not specify parallel operations; instead, only specifies semantic ordering constraints. This provides a separation of concerns between the domain expert and the tuning expert, simplifying the job of the domain expert while providing more flexibility to the tuning expert [18].

The CnC notion of a step is a pure function. Haskell is a pure functional language, making it a great match with CnC.

2.9 Patterns for parallel programming

Patterns have proved to be very useful in object-oriented programming. They describe good solutions to recurring problems in a particular context. In their book, Mattson T.G. et al. describe a pattern language for parallel programming that provides several benefits [29]. It is organised in four design spaces:

- **Finding Concurrency** Start with a specification that solves the original problem, finish with a decomposition of the problem into tasks, plus an analysis of shared data and task dependencies (partial ordering).

- **Algorithm Structure** Select overall program organisation to exploit concurrency identified in previous step.

- **Supporting Structures** High-level constructs affecting large-scale organisation of the source code.

- **Implementation Mechanisms** The "primitives" of parallel computing. Examples are in Java, OpenMP, and MPI.

2.9.1 Algorithmic skeletons

Algorithmic skeletons abstract commonly used patterns of parallel computation, communication, and interaction [13, 14].

This high-level parallel programming technique, known as structured parallelism, enables the composition of skeletons for the development of programs where the control is inherited...
through the structure, and the programmer adheres to top-down design and construction [20].
To the programmer, most parallelism is implicit. The program has to use a parallel pattern
to exploit parallelism. Using such patterns requires advanced language features, in particular
delegates (higher-order functions).

Skeletons can be categorised based on their functions: data parallel, task parallel, and
resolution.

Common skeletons examples:

- **Data**
  
  - *Map* - applies a function simultaneously to all the elements of a list to achieve
    parallelism.
  
  - *Reduce* - iterates an arbitrary function e.g. summation over a data structure in some
    order and build up a return value.

- **Task**
  
  - *Task Farm* - also known as master-slave/worker. The master process distributes
    tasks to worker processes which in turn can distribute to other workers. Tasks are
    executed in parallel. This can also be data parallel where the farmer distributes
    input to a pool of N identical workers.
  
  - *Pipeline* - enables staged (pipelined) computations. Different tasks can be computed
    simultaneously on different pipe stages.

- **Resolution**
  
  - *Divide-and-conquer* - recursively divide the task into sub-tasks until a condition is
    met. The sub-tasks are executed in parallel and the results are merged.

Other approaches are backed by the industry which include Google’s MapReduce, Intel’s
TBB and Microsoft’s TPL.

2.9.2 MapReduce

Google’s MapReduce model is derived from the *map* and *reduce* combinators from functional
programming and supports distributed computing on large data sets on clusters of comput-
ers [12, 23]. Libraries have been written in many programming languages. The programmer
only writes the map and reduce functions, and the runtime system takes care of parallel exe-
cution and fault tolerance. In the execution steps, M map workers and R reduce workers are
created. Master assigns tasks to workers. Apache Hadoop provides an open source implemen-
tation of the framework [2].

Haskell defines the appropriate combinators in its prelude standard library [23].
Example: *map* - returns a list constructed by applying a function (the first argument) to all items in a list passed as the second argument. In this example, the function to apply is "multiply by 2".

```
Prelude> map ((*) 2) [1,2,3]
[2,4,6]
```

Example: *foldl* - takes the second argument and the first item of the list and applies the function to them, then feeds the function with this result and the second argument and so on. *foldl* is left-associative. A right-associative equivalent is the *foldr* function. In the example, the expression "(+)" denotes addition and the constant "0" is the default value.

```
Prelude> foldl (+) 0 [1,2,3]
6
```

### 2.9.3 Intel's TBB

Intel's Threading Building Blocks (TBB) offers a rich and complete approach to expressing parallelism in a C++ program. It is a library that helps take advantage of multi-core processor performance without having to be a threading expert. TBB represents a higher-level, task-based parallelism that abstracts platform details and threading mechanisms for scalability and performance [16]. It makes developing multithreaded applications easy by reducing the number of lines of code and programming complexity. Thread management is handled internally so the programmer does not have to specify anything.

### 2.9.4 Microsoft’s TPL

The Task Parallel Library (TPL) is a library for .NET that makes it easy to take advantage of potential parallelism in a program. It exposes parallel constructs like parallel For and ForEach loops, using regular method calls and delegates. The constructs can be used from any language supporting the .NET framework. The library takes care of threads creating, management and termination, in addition to scaling the number of threads according to the number of available processors.

TPL also includes other constructs such as *Task* (light-weight thread), which is an action that can be executed independent of the rest of the program, and *Future*, which is a task that returns a result. The result is computed in a background thread encapsulated by the Future object. If the result is not yet computed when requested, the asking thread will block until the result is available.

The other construct of TPL is *Parallel* class. TPL provides a basic form of structured parallelism via three static methods in the Parallel class:
**Parallel.Invoke** Executes an array of Action delegates in parallel, and then waits for them to complete

**Parallel.For** Parallel equivalent of a C# for loop

A sequential for loop in C#:

```csharp
int n = ...
for (int i = 0; i <= n; i++)
{
    // ...
}
```

A parallel for loop in C#:

```csharp
int n = ...
Parallel.For(0, n, i =>
{
    // ...
});
```

**Parallel.ForEach** Parallel equivalent of a C# foreach loop

### 2.10 Data structures and algorithms of interest

Data structures are reusable components that programmers rely on to build software systems so they do not have to write code from scratch. Most of the common structures used in mainstream languages are sequential by nature and not built to exploit multi-cores e.g. lists. While Java and C# have concurrent versions of these structures (as discussed earlier), they mainly offer thread-safe operations by dealing with synchronisation, locking and other issues; they do not offer parallelism.

Our data-centric approach to parallelism will look at a commonly used data structure: list. We investigate how to get parallelism for free from alternative representations and parallel implementations in a functional language (Haskell) and high-level approach to specifying parallelism (parallel strategies).

#### 2.10.1 Lists

Data representation strikes a good balance between fast, sequential operations, and potential for independent evaluation of sub-components. Lists are a very common data structure in functional languages, with good language and library support. However, their straightforward representation is inherently sequential. An alternative representation as an append-tree has the advantage of potential parallelism by processing subtrees in parallel (Figure 2.1). Trees
2.10. DATA STRUCTURES AND ALGORITHMS OF INTEREST

(a) Linked list

(b) Append-tree representations

Figure 2.1: Alternative representations of linked list

are hierarchical structure making it easy to apply divide-and-conquer algorithm to it and thus a good match for parallelism. This could possibly introduce ambiguity in data-representation but the underlying representation can be hidden to the user. While we do not ignore the fact that this has the potential of slowing down for some operations, it can give performance improvement for others. So, depending on the application area, a parallel list may be better than using standard list.

2.10.2 Algorithms and structures for the n-body problem

The second stage of the project will look at parallel implementations for the n-body problem which is a popular benchmark application for high-performance computing. The focus is mainly on algorithm but this concrete application also uses specialised data structures.

The n-body problem is a problem of predicting the motion of a system of N bodies that interact with each other gravitationally. N-body algorithms are used in several areas such as molecular dynamics and astrophysics, and thus is the core of a real application rather than a toy example.

The choice of data structure is important in producing an efficient algorithm. One possible implementation of the problem using the Barnes-Hut algorithm [3] involves the use of quadtree for two dimensional space, or octree for three dimensional space.

A quadtree is a tree data structure where each node can have up to four children. Given a number of particles or points in 2D space, a quadtree is constructed by recursively subdividing the space into four quadrants until each quadrant contains exactly one point. The process results in an incomplete tree where the root represents all the points in the system and subtrees represent points in a particular region.

For 3D space, octree is used, in which case the nodes can have up to eight children. Both structures lend themselves easily to parallelisation as subtrees can be processed separately, for example, in search operation.
2.11 Summary

The literature review has covered the main topics relevant to the project. It has helped to get an insight of the current state of research in the area and identify the tools and technologies at our disposal to implement our parallel data structures library and algorithms.

Several approaches to parallelism are covered and the different parallel programming are classified based on the criteria of abstraction level, memory model, data- or task-oriented, explicitness and embedding parallelism constructs. Choosing a high-level functional approach for implementation seems to be the right course of action. Functional language helps to reason about parallelism better than imperative language and prototyping can be done rapidly.

The next two chapters covers the core implementations.
Chapter 3

List data structure

This chapter looks at the standard list data structure and how an alternative tree representation of it can be built in Haskell that lends itself more naturally to parallelisation.

3.1 Standard list

The widely used list data structure in Haskell is simple and quite efficient. The base Prelude package already includes what is needed to start using list in a Haskell program. To create a list of element, simply put them between square brackets separated by commas, e.g:

    myList :: [Int] -- type signature (optional)
    myList = [1,2,3,4,5] -- list of integers

The elements in the list can be of any type, including user-defined ones. The type signature myList::[a] is used where a implies that the elements can be of any type. A list can also be constructed using the cons (: ) function which adds an element to the front of an empty or non-empty list e.g. 1:2:3:4:5:[] is the same as the above example.

Prelude includes basic operations on lists, for example, to get the number of elements in a list, to take or drop the first n elements from the list:

    > length myList
    5
    > take 2 myList

Figure 3.1: Haskell list representation
CHAPTER 3. LIST DATA STRUCTURE

> drop 2 myList
[3,4,5]

These functions are polymorphic as they can operate on list of any type e.g. integers, chars, etc. More operations on lists are found in the Data.List module which has to be imported in the program in order to use them. An example of such an operation is sort which is used later for comparison with the new parallel list.

High-order list functions
A high-order function can take another function as argument, and one important such function in Haskell is map. The map function applies a function to each element of a list and returns the new values.

> map sqrt [4,9,16]
[2,3,4]

This is another interesting function that will be covered in the next sections.

3.1.1 Alternative representation
Lists in Haskell are implemented as singly-linked lists and are inherently sequential. A pointer points to the first item in the list and in order to get to the nth item, the list has to be traversed from the left to right until the element is found. So the access time of the nth element requires a time complexity of $O(n)$.

This can be reduced by using an alternative underlying representation of the list structure. A structure that exposes parallelism more easily has the added advantage to give better performance when used on multi-core machines.

Operations on the standard Haskell list are not parallel. Often, processing a list in parallel involves splitting it into chunks and then processing each chunk in parallel before merging the intermediate results to get what is expected. The following shows how this is usually achieved:

```haskell
myList = [1..10000]
midway = length myList `div` 2
(chunk1,chunk2) = splitAt midway myList
-- process chunk1 and chunk2 in parallel
```

A more natural way of representing the list is by using a binary append tree representation as described in Chapter 2.

Some term definitions:

Binary tree: each node has at most two child nodes (left and right).
3.1. STANDARD LIST

**Binary leaf tree:** elements are stored in the leaves only. Inner nodes can be used to store other information about the tree e.g. subtree size.

**Complete tree:** one that is perfectly balanced.

**Append tree:** is similar to binary leaf tree, with internal nodes storing an operator for e.g. (:) or cons.

Using tree representations, it is easier to view the left and right branches of a tree as subtrees. In the final version of the parallel list implementation, binary leaf tree forms the basis of a more advanced data structure applied in order to get more efficiency out of the structure and also to deal with problem of maintaining a balanced tree.

The implementation progressed over a number of attempts to build the new structure, which is summarised below.

- **Binary tree** A binary tree is well suited for parallelism and seems to be the ideal choice in this case. The first attempt is a very simple binary tree representation and this is achieved by defining a new data type:

  ```
  data Tree a = Leaf a | Node (Tree a) (Tree a)
  ```

  A different version of this involves storing 2 elements instead of a single element on each leaf. This has the effect of decreasing the height of the tree but operations on such a structure can be more tricky and still involve dealing with list. In both cases, operations to convert a Haskell list to the new structure and vice-versa are implemented:

  ```
  parList::[a]-->Tree a and toList::Tree a-->[a].
  ```

  -- version 1 see figure: Append tree
  > parList [1,2,3,4]
  Node (Node (Leaf 1) (Leaf 2)) (Node (Leaf 3) (Leaf 4))

  -- version 2
  > parList [1,2,3,4]
  Node (Leaf [1,2]) (Leaf [3,4])

**Issues** Converting a list of elements to tree in one go is straightforward and we can make sure that the tree is kept balanced in the process. However, when adding elements one-by-one using the `cons` function, this simple representation does not deal with balancing the tree as it grows. It is tempting to think about flattening the tree before adding an
element to it and then convert back to tree, but this obviously is not efficient and actually makes the use of an alternative tree representation pointless. So, the answer is to look into balanced tree.

- **Red-black tree** Red-black tree is the one solution considered to the problem of having to keep a tree representation balanced. Having a balanced tree also ensures that the *lookup* operation does not take more than $O(\log n)$ time. Red-black tree is a binary search tree in which every node is colored either red or black. The colours are used to keep the tree approximately balanced using a *balance* function.

While this seems to be a good solution, this representation goes against the original idea. The red-black tree is essentially not a binary leaf tree, so it stores elements in inner nodes. An attempt to adapt the red-black to fit our purpose by rewriting the *balance* function did not quite work. Re-balance may move leaves inside, it becomes tricky to use this representation.

One more characteristic of the red-black tree is that elements are sorted when they are inserted. So it would require to be changed even more to avoid this as the aim is to only have an alternative representation of list without changing the order of the data.

- **Sets** The *Data.Set* module in Haskell implements a set data structure as a weight balanced tree. However, adapting the code is again tricky given that set does not maintain order and identical elements are not repeated in the structure. The next solution is deemed better alternative which led to abandon looking more into Haskell’s set source.

- **Random-access lists** Okasaki [33] proposed random-access list as an alternative structure that consists of a collection of *completely* balanced binary leaf tree instead of a single *approximately* balanced one.

The structure supports *lookup* operation in $O(\log n)$ time compared to $O(n)$ for the same operation in Haskell’s list. It also provides similar time for *update* operation, while $O(1)$ time for other list operations (*cons*, *head*, *tail*).

Random-access list solves the problem of having to keep a tree always balanced. It is also based on a tree structure albeit with some more improvement. It is a good choice to start implementing an alternative representation for list based on this structure, which we call RAList.

### 3.2 RAList implementation

The RAList is an adaptation of Okasaki’s [33] implementation of random-access list. It is implemented in a new module with the same name, and can be imported in any Haskell program and used instead of the standard list for the operations it implements.
3.2. RALIST IMPLEMENTATION

The online documentation for the RAList module can be found here\(^1\).

### 3.2.1 Data type definition

The new data types for the structure are defined as follows:

```haskell
data RAList a = RAL Int [Digit a]
data Digit a = Zero | One (Tree a)
data Tree a = Leaf a | Node Int (Tree a) (Tree a)
```

The `Tree` data type is the same as defined for the binary leaf tree earlier, except that here it includes an `Int` to store the size of the tree. Similarly, the `Int` in the `RAList` definition refers to the sum of the size of all trees i.e. number of elements in the structure. The above definitions do not include strictness annotations which are added in the final version to improve performance.

This efficient representation guarantees that at all time, we have a list of completely balanced binary leaf tree. For e.g. a structure with 100 elements can be built using 3 completely balanced trees with 4, 32 and 64 elements in each.

### 3.2.2 ParTree

Binary leaf trees form the basis of the RAList structure. RAList depends on Tree as a substructure. It is therefore decided to have the Tree definition in a separate module called ParTree. This gives more modularity in the implementation. ParTree defines functions that exploit parallelism inherent in a tree structure and is part of the parallel library. These functions are exported and used in the RAList.

Parallel operation on ParTree

An example of a parallel operation in ParTree is the `treeMin` function which returns the smallest comparable element from the tree.

Initially: using just `seq` and `par` primitives.

Type signature: `treeMin :: (Ord a) => Tree a -> a`

\(^1\)RAList Documentation http://www.macs.hw.ac.uk/~pt114/msc-project/ralist-doc/
CHAPTER 3. LIST DATA STRUCTURE

### Figure 3.4: ParTree functions

<table>
<thead>
<tr>
<th>Operation</th>
<th>Is parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>No</td>
</tr>
<tr>
<td>link</td>
<td>No</td>
</tr>
<tr>
<td>treeToList</td>
<td>Yes</td>
</tr>
<tr>
<td>treeMap</td>
<td>Yes</td>
</tr>
<tr>
<td>treeFold</td>
<td>Yes</td>
</tr>
<tr>
<td>treeElem</td>
<td>Yes</td>
</tr>
<tr>
<td>treeReverse</td>
<td>Yes</td>
</tr>
<tr>
<td>treeMin</td>
<td>Yes</td>
</tr>
</tbody>
</table>


```haskell
treeMin (Leaf x) = x
treeMin (Node _ l r) = left ‘par’ right ‘seq’ min left right
  where
    left = treeMin l
    right = treeMin r
```

Then: using `runEval` as described in the new strategies paper [26].

```haskell
treeMin (Leaf x) = x
treeMin (Node _ l r) = runEval $ do
    left <- rpar $ treeMin l
    right <- rseq $ treeMin r
    return (min left right)
```

We discuss more about these functions where they are used in RAList.

### 3.2.3 Operations on RAList

The new library provides a set of operations including both sequential and parallel.

**Sequential operations** Some operations are inherently sequential so they can be difficult or even not possible to parallelise. Sometimes, there is also no need for parallelisation, for example, for most $O(1)$ operations.

**Basic**

Basic operations like those found in standard list are implemented for the RAList. These operations are important as they support more advanced functions. Most of the function names are the same as in Haskell’s list, so uses of these function names need to be qualified with an alias for the RAList module. The time complexity for the functions are given.

- **cons** $O(1)$. Prepend an element to the RAList. Same as (:) in standard list.
- **head** $O(1)$. Return the first element in the RAList.
- **tail** $O(1)$. Return the tail of the RAList, that is, a RAList without the first element.
3.2. **RALIST IMPLEMENTATION**

- **empty** $O(1)$. Return an empty RAList: RAL 0 []
- **isEmpty** $O(1)$. Return true if RAList is empty. Same as null in standard list.
- **length** $O(1)$. Return the number of elements in the structure. Note length in standard list is a $O(n)$ operation.

**Conversion Operations**

The following 2 operations help to convert from and to a standard list.

- **fromDataList** Convert from standard list to RAList.
- **toList** Convert from RAList to standard list. (there is a parallel variant of this)

For example, a RAList can be constructed as follows:

```plaintext
> fromDataList [1..7]
RAL 7 [ One (Leaf 1)
  ,One (Node 2 (Leaf 2) (Leaf 3))
  ,One (Node 4 (Node 2 (Leaf 4) (Leaf 5)) (Node 2 (Leaf 6) (Leaf 7)))]
```

The above shows that the tree sub-structures in the RAList are always completely balanced.

**Advanced operations**

The more advanced operations on the structure include lookup (search) by index, update, checking if element is in the RAList, obtaining sub-RAList from take, drop, filter and partition operations.

- **lookup** $O(\log n)$. Lookup the RAList using the given index like (!!) in standard list. Index has to be in range, otherwise an error occurs.
- **update** $O(\log n)$. Update the element at the given index with a new value.
- **elem** Check for any occurrence of the given element in the structure. Sequential implementation similar to elem from standard list.
- **pelem** Par. parallel implementation of elem.
- **pelem'** Par. improved parallel version.
- **take** Take the first n elements.
- **drop** Drop the first n elements.
- **filter** Return a new RAList with only those elements that match the predicate.
• partition Return 2 partitions of the given RAList. The first RAList in the tuple pair satisfies the predicate, while the second does not.

Both lookup and update operations have $O(\log n)$ time. These two functions are essentially sequential but as we will see in the results and evaluation, only a change in representation gives better performance compared to similar functions in standard list.

Parallel strategies

The Control.Parallel.Strategies module contains some useful strategies for standard list e.g. evalList and parList. Similar strategies are defined to apply on RAList.

- evalRAList Strategy combinator that walks over the RAList and applies the argument strategy $s$ to every element. It generalises parRAList.

- parRAList is obtained by composing the element strategy $s$ with rpar:

  \[
  \text{parRAList } s = \text{evalRAList } (\text{rpar } \text{‘dot’ } s) \text{ where rpar sparks its argument (for evaluation in parallel).}
  \]

  seqRAList is not given, but can be easily defined just like parRAList.

  These strategies are used in parallelising some of the benchmark applications and also for parallel map and fold.

Parallel Map

An important high-order function, map in standard list is usually parallelised in 2 ways:

\[
\text{> map } f \text{ list ‘using’ parList rdeepseq, or alternatively}
\]

\[
\text{> parMap rdeepseq } f \text{ list, where parMap is defined in the strategies library.}
\]

Both evaluates the elements in the list in parallel to the evaluation degree (e.g. rdeepseq or rseq) specified.

The RAList provides both sequential and parallel map functions which are important for transformation:

- ralMap Sequential map function which applies a function $f$ to every element of the RAList.

  type signature: \(\text{ralMap}::(a \to b) \to \text{RAList } a \to \text{RAList } b\)

  example:

  ralMap sqrt $ fromDataList [1,4,9]

  RAL 3 [One (Leaf 1.0),One (Node 2 (Leaf 2.0) (Leaf 3.0))]

- pRalMap Par. parallel implementation of map function on RAList, using parRAList.

- pRalMap’ Par. another version that uses treeMap. Recall that treeMap is a parallel map function on trees from the ParTree module.
3.3. BENCHMARK APPLICATIONS

Parallel Fold

The standard list provides 2 fold functions: foldr and foldl. The first one folds elements from
the right while the other folds from the left, so the operator (first argument to the functions
e.g. (+) ) gets applied in a different order.

foldr :: (a -> b -> b) -> b -> [a] -> b

e.g. foldr (+) 0 [1,2,3] = 1+(2+(3+0))

foldl :: (a -> b -> a) -> a -> [b] -> a

e.g. foldl (+) 0 [1,2,3] = ((0+1)+2)+3

Both functions are generally not parallelisable unless their type signatures are changed and
limited to commutative and associative operations only e.g. sum (+) and product (*).

So, for parallel fold on RAList, the type signature is:

parfold :: (a -> a -> a) -> a -> RAList a -> a

• parfold Par. parallel implementation of fold function. The collection of trees in the
RAList is folded in parallel and at tree level, more parallelism is gained through the use
of treeFold. A top level fold generates the final result. The function f has to be both
associative and commutative.

• parfold' Par. an alternative parallel version of fold on RAList where the collection of
trees are processed in order and parallelism is at sub-structure (tree level) only.

RAList also provides a fold function that is sequential and folds from the left.

type signature: fold :: (a -> b -> a) -> a -> RAList b -> a

e.g. fold (+) 0 $ fromDataList [1..5]

3.3 Benchmark applications

In addition to the operations covered, the following benchmark applications are implemented
(both sequential and parallel). These are compared with similar implementation in standard
list which is found in the StdList module. The following briefly describes each applications.
Note in some cases, two alternative parallel implementations are given. Their performances are
evaluated in the next section.

• Sum

  – sum Sequential sum function that uses fold.

  – psum Par. parallel sum function that uses parfold.

  – psum' Par. parallel sum function that uses parfold’.

• Factorial
- **facto** Sequential factorial function that uses `fold`.
- **pfacto** Par. parallel factorial function that uses `parfold`.

- **Sort**
  - **quicksort** Sort a **RAList** of \( n \) random numbers using the **quicksort** algorithm.

- **Histogram**
  - **histo** The histogram function counts the occurrences of each integer in a list of \( 5n \) integers chosen randomly from \( 0..n-1 \).

- **Minimum**
  - **pRAlMin** Par. Return the smallest integer in a **RAList**.
  - **pRAlMin’** Par. improved parallel version. \( \min \) is commutative i.e. \( \min x y = \min y x \), so it can be used in **parfold**.

- **Nub**
  - **nub** Par. The nub function removes duplicate elements from the **RAList**. Naive version.
  - **nub1** Sequential version using circular data structure.
  - **nub2** More advanced circular data structure. (Does not work).

### 3.4 Results and evaluation

The **RAList** is evaluated by comparing its performance against standard list. The **StdList** module exports functions used as benchmark applications for standard list. The evaluation consists of looking at the effect of the change in representation and also the performance of parallel operations on the new structure.

#### Structure construction cost

There is an overhead in building the **RAList** (Figure 3.5). Compared to standard list, it takes more time and space to build a **RAList**. This is obvious because of the representation of **RAList**. The runtimes taken for the evaluation do not include the construction time. Only the operation time is measured.

#### Representation performance

The alternative representation of a list itself brings in some good performance for sequential operations as shown in Figure 3.6. This is due to the way the **RAList** is processed compared to a sequential list.
3.4. RESULTS AND EVALUATION

We obtain better performance for all benchmark applications, except for quicksort. The new structure seems to be particularly costly for this operation as it involves walking through the entire RAList and then constructing it back again. The results are consistent with what Okasaki [33] observes in his paper. It shows the efficiency of random-access list in achieving \(O(\log n)\) time for lookup and update operations while maintaining \(O(1)\) for basic operations. The histogram function is nine times faster for \(N=10000\) and takes too long to complete for \(N=100000\) using standard list.

**RAList parallel operations performance**

The operations implemented in parallel are measured on 1 to 8 cores. The runtime and speedup are compared with similar parallel operations implemented in standard list (**StdList**).

**Parallel map**

The performance of the parallel map implementation in **RAList** (**pRalMap'**) against **parMap** from **Control.Parallel.Strategies** on standard list is first compared.

From the measurements and graphs in Figures 3.7a and 3.8, **RAList** parallel map outperforms that of standard list both in terms of runtime and speedup. This is an interesting outcome as **parMap** is used very often instead of sequential **map** function in Haskell to easily specify algorithm in existing code.

**Other parallel operations**

Figure 3.7b shows side-by-side comparison of the other parallel operations on both structures.
### CHAPTER 3. LIST DATA STRUCTURE

#### (a) Parallel map

<table>
<thead>
<tr>
<th>#PE</th>
<th>Runtimes</th>
<th>Speedups</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.535846</td>
<td>0.88</td>
</tr>
<tr>
<td>2</td>
<td>0.523254</td>
<td>0.91</td>
</tr>
<tr>
<td>3</td>
<td>0.466211</td>
<td>1.02</td>
</tr>
<tr>
<td>4</td>
<td>0.357341</td>
<td>1.33</td>
</tr>
<tr>
<td>5</td>
<td>0.372790</td>
<td>1.28</td>
</tr>
<tr>
<td>6</td>
<td>0.345771</td>
<td>1.38</td>
</tr>
<tr>
<td>7</td>
<td>0.312819</td>
<td>1.52</td>
</tr>
<tr>
<td>8</td>
<td>0.321247</td>
<td>1.48</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>#PE</th>
<th>Runtimes</th>
<th>Speedups</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5.959945</td>
<td>0.87</td>
</tr>
<tr>
<td>2</td>
<td>6.100406</td>
<td>0.85</td>
</tr>
<tr>
<td>3</td>
<td>7.090542</td>
<td>0.73</td>
</tr>
<tr>
<td>4</td>
<td>7.194050</td>
<td>0.72</td>
</tr>
<tr>
<td>5</td>
<td>7.172442</td>
<td>0.73</td>
</tr>
<tr>
<td>6</td>
<td>7.084419</td>
<td>0.73</td>
</tr>
<tr>
<td>7</td>
<td>7.248549</td>
<td>0.72</td>
</tr>
<tr>
<td>8</td>
<td>7.440132</td>
<td>0.70</td>
</tr>
</tbody>
</table>

#### (b) Parallel operations

<table>
<thead>
<tr>
<th>#PE</th>
<th>Runtimes</th>
<th>Speedups</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.048656</td>
<td>0.95</td>
</tr>
<tr>
<td>2</td>
<td>0.026891</td>
<td>1.72</td>
</tr>
<tr>
<td>3</td>
<td>0.020585</td>
<td>2.25</td>
</tr>
<tr>
<td>4</td>
<td>0.016751</td>
<td>2.76</td>
</tr>
<tr>
<td>5</td>
<td>0.016348</td>
<td>2.83</td>
</tr>
<tr>
<td>6</td>
<td>0.012912</td>
<td>3.58</td>
</tr>
<tr>
<td>7</td>
<td>0.013629</td>
<td>3.40</td>
</tr>
<tr>
<td>8</td>
<td>0.011776</td>
<td>3.93</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>#PE</th>
<th>Runtimes</th>
<th>Speedups</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.674607</td>
<td>0.92</td>
</tr>
<tr>
<td>2</td>
<td>0.447564</td>
<td>1.38</td>
</tr>
<tr>
<td>3</td>
<td>0.356811</td>
<td>1.73</td>
</tr>
<tr>
<td>4</td>
<td>0.397828</td>
<td>1.56</td>
</tr>
<tr>
<td>5</td>
<td>0.397138</td>
<td>1.56</td>
</tr>
<tr>
<td>6</td>
<td>0.337649</td>
<td>1.83</td>
</tr>
<tr>
<td>7</td>
<td>0.326625</td>
<td>1.89</td>
</tr>
<tr>
<td>8</td>
<td>0.329022</td>
<td>1.88</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>#PE</th>
<th>Runtimes</th>
<th>Speedups</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.061929</td>
<td>0.81</td>
</tr>
<tr>
<td>2</td>
<td>0.05941</td>
<td>1.39</td>
</tr>
<tr>
<td>3</td>
<td>0.028945</td>
<td>1.72</td>
</tr>
<tr>
<td>4</td>
<td>0.023021</td>
<td>2.17</td>
</tr>
<tr>
<td>5</td>
<td>0.021696</td>
<td>2.30</td>
</tr>
<tr>
<td>6</td>
<td>0.020853</td>
<td>2.39</td>
</tr>
<tr>
<td>7</td>
<td>0.019677</td>
<td>2.54</td>
</tr>
<tr>
<td>8</td>
<td>0.021482</td>
<td>2.32</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>#PE</th>
<th>Runtimes</th>
<th>Speedups</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.762265</td>
<td>0.93</td>
</tr>
<tr>
<td>2</td>
<td>0.480656</td>
<td>1.47</td>
</tr>
<tr>
<td>3</td>
<td>0.439548</td>
<td>1.61</td>
</tr>
<tr>
<td>4</td>
<td>0.368033</td>
<td>1.92</td>
</tr>
<tr>
<td>5</td>
<td>0.338942</td>
<td>2.08</td>
</tr>
<tr>
<td>6</td>
<td>0.326859</td>
<td>2.16</td>
</tr>
<tr>
<td>7</td>
<td>0.338334</td>
<td>2.09</td>
</tr>
<tr>
<td>8</td>
<td>0.353767</td>
<td>2.00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>#PE</th>
<th>Runtimes</th>
<th>Speedups</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.071081</td>
<td>0.82</td>
</tr>
<tr>
<td>2</td>
<td>0.039010</td>
<td>1.47</td>
</tr>
<tr>
<td>3</td>
<td>0.030416</td>
<td>1.93</td>
</tr>
<tr>
<td>4</td>
<td>0.025430</td>
<td>2.30</td>
</tr>
<tr>
<td>5</td>
<td>0.025450</td>
<td>2.30</td>
</tr>
<tr>
<td>6</td>
<td>0.023285</td>
<td>2.52</td>
</tr>
<tr>
<td>7</td>
<td>0.024682</td>
<td>2.37</td>
</tr>
<tr>
<td>8</td>
<td>0.024490</td>
<td>2.39</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>#PE</th>
<th>Runtimes</th>
<th>Speedups</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.744979</td>
<td>0.96</td>
</tr>
<tr>
<td>2</td>
<td>0.461518</td>
<td>1.55</td>
</tr>
<tr>
<td>3</td>
<td>0.352895</td>
<td>2.02</td>
</tr>
<tr>
<td>4</td>
<td>0.367259</td>
<td>1.94</td>
</tr>
<tr>
<td>5</td>
<td>0.337482</td>
<td>2.11</td>
</tr>
<tr>
<td>6</td>
<td>0.321410</td>
<td>2.22</td>
</tr>
<tr>
<td>7</td>
<td>0.335741</td>
<td>2.13</td>
</tr>
<tr>
<td>8</td>
<td>0.335451</td>
<td>2.13</td>
</tr>
</tbody>
</table>

Figure 3.7: Parallel map and operations measurements
Figure 3.8: RAList and standard list parallel map runtime & speedup: where \( p\text{RalMap}' \) is RAList parallel map implementation and \( \text{parMap} \) is from Control.Parallel.Strategies for standard list.
Figure 3.9: RALIST and standard list parallel operations runtime: note the Time axis to see the difference in runtime between the operations from the two structures (x10). The decrease in the sum function runtime is more constant for RALIST than standard list over increasing number of processors.
Figure 3.10: RAList and standard list parallel operations speedup: the sum function which uses RAList’s parfold notes a particularly better speedup than the others.
While Okasaki could not get better performance for the sum operation by only using an alternative representation, we manage to make use of parallel fold implemented for \texttt{RAList} (and \texttt{ParTree} sub-structure) in the implementation of \texttt{psum}. This gives better runtime (Figure 3.9) of almost 10 times compared to similar operation on standard list. It also achieves good speedup (Figure 3.10) compared to \texttt{parsum} from \texttt{StdList}.

The other 2 parallel operations (\texttt{elem} and \texttt{min}) compared also gives better performance in \texttt{RAList} against corresponding applications implemented in standard list.

The conclusion from the results and evaluation is that the \texttt{RAList} proves to be efficient for some basic functions including high-order functions like map and fold. The construction cost of it is expected as it maintains admin information such as the subtree size whereas standard list does not store these information. However, this helps to speedup the operations.
Chapter 4

Algorithms and structures for the n-body problem

This chapter focuses on implementations of the n-body problem as described in the background (section 2.10.2). It covers different algorithms and data structures used in solving the problem, and comparison between related implementations from other researchers including those produced for the SICSA Multicore Challenge.

4.1 The problem description

The n-body problem is about simulating the motion of N particles that are mutually affected by gravitational forces. The computation basically proceeds over time steps. The net force on every body has to be computed and then updating its positions and other attributes in each time step. If all pair wise forces are computed directly, this require $O(n^2)$ operations at each time step. This method is conceptually simple, but it is inefficient for large-scale simulations.

4.1.1 Solving methods

There are a number of methods or algorithms to solve the problem. We look at particularly three of them which provides enough ground for comparison and evaluation.

We consider particles in a 2D space in the following methods description. Each particle has a position $x$ and $y$, velocity $vx$ and $vy$, and mass $m$. The initial positions, velocities and masses can be either read from a text file or generated in the code as floating point numbers.

The main part of the n-body problem is to calculate the accelerations and update the velocities and positions at each time step.
Full all-pairs

This method refers to the all pair wise force calculations. It is straightforward but brute method in which the force calculations can be made within a double iteration loop in an imperative setting. The following code extract in Java illustrates the steps in the computation where bodies is an array of body objects with positions (x, y), velocities (vx, vy) and mass:

```java
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        if (i != j) {
            double dx = bodies[i].x - bodies[j].x;
            double dy = bodies[i].y - bodies[j].y;

            double dSquared = dx * dx + dy * dy;
            double distance = Math.sqrt(dSquared);
            double mag = timeStep / (dSquared * distance);

            bodies[i].vx -= dx * bodies[j].mass * mag;
            bodies[i].vy -= dy * bodies[j].mass * mag;
        }
    }
}
// recalculate position
for (int i = 0; i < N; i++) {
    bodies[i].x += dt * bodies[i].vx;
    bodies[i].y += dt * bodies[i].vy;
}
```

Within the nested loop, the force on each body is calculated and velocity updated. The positions of the bodies are then recalculated for each time step.

Implementation in Haskell is different. Being a pure functional language, it does not allow destructive update. Also, loops are implemented using recursive function calls or alternatively the following list comprehension can be used to generate all the pairs, and then using a map function to perform the calculations as in the nested loop.

```haskell
ns = [0..n] -- n is number of bodies
[(i, j) | i <- ns, j <- ns]
```

Pros and Cons of method

+ simple, precise and vectorizes well
+ easily adapted in Haskell
4.1. THE PROBLEM DESCRIPTION

- number of computations is 2 times greater when compared to triangular version
- not practical when N is large

Triangular

A little less naive algorithm is the triangular method in which less calculations are performed and has time complexity $O(n^2-n)/2$. This is achieved by using a triangular nested loop, and updating the velocity of body $j$ at the same time. The Java code can be changed slightly to the following in this method:

```java
for (int i = 0; i < N; i++) {
    // (1) j=0 changed to j=(i+1)
    for (int j = (i + 1); j < N; j++) {
        ...
        // (2) add the 2 lines to update velocity for body j
        bodies[j].vx += dx * bodies[i].mass * mag;
        bodies[j].vy += dy * bodies[i].mass * mag;
    }
}
```

This algorithm performs better in Java which uses mutable arrays, but in Haskell, we cannot just update the velocities directly but have to accumulate the accelerations. The issue relating to this is discussed later in the chapter. To create the body pairs, the following list comprehension can be used:

```java
[(i, j)|i<-ns, j<-[((i + 1)..(n - 1))] |i<-ns] --or
[(i, [j|j<-[((i + 1)..(n - 1))] |i<-ns]
```

Pros and Cons of method

+ less iterations thus less calculations
- harder to code in Haskell

Barnes-Hut Algorithm

The BH algorithm [3] is an approximation method. It speeds up the conventional pair wise method by grouping nearby bodies and approximating them as a single body. If the group is sufficiently far away, the gravitational effects can be approximated by using the center of mass. The center of mass of a group of bodies is the average position of a body in that group, weighted by mass. If two bodies have positions $(x_1, y_1)$ and $(x_2, y_2)$, and masses $m1$ and $m2$, then their total mass and center of mass $(x, y)$ are given by:

$$m = m1 + m2$$
\[ x = \frac{x_1\times m_1 + x_2\times m_2}{m} \]
\[ y = \frac{y_1\times m_1 + y_2\times m_2}{m} \]

The BH algorithm recursively divides the set of bodies into groups by storing them in a quadtree (octree for 3D space). The algorithm demonstrates considerable amount of parallelism.

**Pros and Cons of method**

- faster than conventional methods
- rather complex algorithm
- result depends on threshold value used

**Other methods**

There are other methods than those presented here. An alternative tree method instead of the BH algorithm is the fast multipole method (FMM). Another possibility is the particle mesh method which uses a different approach. However, for comparison purposes within the scope of this project, a traditional method and an approximation method are sufficient.

### 4.2 Implementation

#### 4.2.1 Approach

A number of sequential implementations in different languages using the full all-pairs method exist as starting points for parallel implementations from the Computer Language Benchmarks.\(^1\) The implementation in Java was useful to understand the algorithm in an imperative context, which later served as a comparison with the functional implementations. However, the Haskell implementation from the CLB uses mutable data structure to do in-place update. We keep our implementations pure by not using IO structures.

Several other implementations from researchers in the area of parallel programming using the Barnes-Hut algorithm were produced for the SICSA MultiCore Challenge: n-body computation.\(^2\) At the end of this chapter, a comparison between these implementations are done.

Implementations are provided for the three methods detailed in the previous section. For each method, we

- start with an initial sequential algorithm.
- perform profiling to optimised the algorithm and to point out which parts of the program are the "big eaters" in terms of time.
- implement parallel versions targeted for shared-memory (via GHC-SMP) basing on the time and heap profiles.

---

4.2. IMPLEMENTATION

- perform further performance tuning to improve runtime.
- measure and compare the results.

4.2.2 Sequential algorithm

**Full all-pairs**

In the all-pairs implementation, the main computation to calculate all forces induced by each of the bodies occurs in the `updateVel` function which uses two `map` functions to simulate the double iteration loop used in an imperative implementation. The `foldl'` function takes the initial velocity and deduces the changes to get the updated velocity. The position of the body is then updated for each time step.

```haskell
updateVel ps vs ms = map (\i -> f (vs!i) i) where
  f v i = foldl' (deductChange) v (map (\j -> g i j) ms)
  g i j
  | i == j = V 0 0 0
  | otherwise = V (dx * mass_j * mag) (dy * mass_j * mag) (dz * mass_j * 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
    P ix iy iz = ps!i
    P jx jy jz = ps!j
    M mass_j = ms!j
```

Full listing in Appendix A.2.1.

**Triangular**

The triangular version is an attempt to have the number of computations decreased by trying to calculate the acceleration for both body i and j at the same time. But in trying to achieve so, updating the velocities for both bodies is made difficult here. So the accelerations are accumulated and then used to update the velocities.

```haskell
calcAccels ps ms i = calc i (i+1) []
```
where
calc i j acc
| j == n = acc
| otherwise = calc i (j+1) new_acc
where
new_acc = accel_i:accel_j:acc -- accumulate accels
accel_i = (i,V (-dx * mass_j * mag) (-dy * mass_j * mag ) (-dz * mass_j * mag))
accel_j = (j,V (dx * mass_i * mag) (dy * mass_i * mag) (dz * mass_i * mag))
...

Full listing in Appendix A.2.2.

This implementation works slower compared to the full all-pairs and even runs into stack-overflow for larger number of bodies.

Barnes-Hut algorithm

The Barnes-Hut algorithm relies on partitioning the bodies of the system into separate cubes in 3D (or squares in 2D). The partitioning is done using an octree as the underlying data structure.

Our implementation is an adaptation of the 2D version of the algorithm presented in several papers focusing on nested-data parallelism [8, 34, 22]. The algorithm is changed to work with 3D space and instead of using parallel arrays, a different approach to parallelism using strategies is used.

The algorithm has 2 phases:
(1) the tree is constructed from a body set (buildTree).

The tree representation is as follows, where PM is used to store the center of mass (P) and total mass (M):

\[
\text{data PM = PM \{\# UNPACK \#\} \!P \{\# UNPACK \#\} \!M}
\]

\[
\text{data BHTree = BHT \{\# UNPACK \#\} \!Double \{\# UNPACK \#\} \!PM ![ BHTree]}
\]

The above definition allows a node to have any number of children, in our case, this will not be more than 8 children per node. Refer to Appendix A.2.3 (relevant function names are put between brackets).

The center of mass and total mass of all bodies are first calculated (calcCentroid) and stored at the root node of the tree to represent the whole space. The coordinates of the bounding box that contain all bodies are found (findBounds, illustrated in 2D in figure 4.1). The box is used to subdivide the bodies into 8 cubes (splitPoints) and then the center of
4.2. IMPLEMENTATION

4.2. IMPLEMENTATION

Figure 4.1: 2D bounding box: Given a number of 2D points on a XY graph, the lower left X, upper right X, lower left Y and upper right Y coordinates create a bounding box which is used as a starting point to recursively subdivide the box into squares.

mass and total mass of the bodies contained in each cube are computed in the same way and stored as the children nodes. The process continues until a region or cube has no body in it. The actual bodies do not need to be stored in external (leaf) nodes as in some implementation as the center and total mass are already calculated when constructing the tree.

(2) the acceleration for each body is computed by traversing the tree (calcAccels).

The traversal along any path is stopped as soon as an internal node that is far enough away is encountered (isFar). This is determined by a threshold value between 0 and 1 coded in the algorithm.

4.2.3 Optimisation

The previous section describes how the sequential algorithms work. However, before we arrived to the final optimised sequential versions for each algorithm, a number of optimisation techniques is applied. The following is taken from the version history of the different implementations and describes the optimisation performed as we evolved from the different versions.

The very first implementation is a direct adaptation of the Java implementation of the problem in Haskell, making sure that the solution remains purely functional. So no mutable structure e.g. mutable IO array is used.

In this simple adaptation, the positions, velocities and masses of the bodies are hardcoded in the source file. The idea is to get an initial working version in Haskell and compare if the results match with that of the Java implementation. At this point no performance consideration is taken at all. List is used to store the bodies and this is very inefficient as the velocities and positions need to be changed during the computation.

There is no new data type defined but instead we use type synonym to represent a body:
type Position = (Double,Double,Double) -- (x,y,z)
type Velocity = (Double,Double,Double) -- (ux,vy,vz)
type Mass = Double -- m
type Body = (Position,Velocity,Mass)

Tail recursion and Strictness

Version 1.1 addresses the stackoverflow error that occurs if the number of bodies is increased. The problem is fixed by making all functions tail recursive and using accumulating parameters. Strictness is also introduced to avoid unnecessary thunking of computations.

Array vs List

Version 2 works differently by accumulating the change in velocity for each body and then updating the initial velocity using the \texttt{accum} function from Haskell’s immutable array module. This gives better performance as the function is already optimised.

Version 3 sees a complete move from using lists to arrays for storing positions, velocities and masses. Access time is greatly reduced when the number of bodies is increased. And with the \texttt{accum} function, it is even better to update the velocities. There is no need to convert from list to array and vice versa.

Data types vs type synonym

Version 4 and 4.1 get more real by reading input from a text file instead of hardcoded bodies. We also moved from using type synonym to type constructor:

\begin{verbatim}
data M = M {-# UNPACK #-} !Double
data P = P {-# UNPACK #-} !Double {-# UNPACK #-} !Double {-# UNPACK #-} !Double

data V = V {-# UNPACK #-} !Double {-# UNPACK #-} !Double {-# UNPACK #-} !Double
\end{verbatim}

Two important changes to note here:

\textbf{Strict fields}

The strictness annotation (\texttt{!}) on constructor fields is used mainly to avoid space leaks. We want the doubles to be evaluated eagerly, not lazily as we will need all the values during the computation and delaying evaluation may be inefficient in our program at each step of the simulation.

\textbf{Unpacking}

The \texttt{UNPACK} pragma indicates to the compiler that it should unpack the contents of a constructor field into the constructor itself, removing a level of indirection.
Other optimisation techniques

- **Double instead of Float** Double is preferred over float because the GHC compiler is optimised to handle double better than float.

- **BangPatterns** GHC extension is used to make argument strict explicitly for some functions. Without this pattern we would have to manually use `seq` when we do not want an accumulator in a function to be lazily evaluated.

- **Using fold\(^l\) instead of fold\(^l\)** To be sure to use a tail recursive pattern, we use appropriate high-order functions from the standard libraries. For e.g. the standard `sum` function on lists is not used. Instead, we define our own `sum\(^l\)` function as follows:

  ```haskell
  sum\(^l\) = foldl\(^l\) (+) 0
  ```

  This prevents stack overflow as `foldl\(^l\)` is more efficient way to arrive at the result because it does not build a huge thunk.

- **Use `map` function instead of list comprehension.** This ensures that we follow certain pattern and also allows adding parallelism easily later.

  ```haskell
  -- list comprehension
  [someFunction x y z | (x,y,z) <- positions]
  -- map function
  map (\(x,y,z) \rightarrow someFunction x y z) positions
  ```

- **Compilation is always done with the optimisation flag on (-O2).**

These optimisations are relevant and applied to all three algorithms.

### 4.2.4 Profiling

Sequential profiling gives information about the time allocation and memory consumption of each function. By analysing this information, it helps to come up with a more optimised sequential algorithm, for e.g. space leak can be found through profiling. Profiling is also used to identify potential functions that can be parallelised.

In order to get the profiling information, the algorithm needs to be compiled for seq. execution and run it with the `-pT` and `-hC` flags on. This produces the output below (trimmed version):

```
<table>
<thead>
<tr>
<th>COST CENTRE</th>
<th>MODULE</th>
<th>%time</th>
<th>%alloc</th>
</tr>
</thead>
<tbody>
<tr>
<td>- All-pairs</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
The output shows the time spent in each function. Some observations:

**All-pairs:** the core computation happens in the `updateVel` function, so it takes the majority of the time to run as expected. The `deductChange` function is called from within the `updateVel` function, so the later accumulates 99.8% of the overall time.

**Triangular:** the triangular version proceeds by computing the accelerations on all bodies, which takes three quarter of the time, and then the `addAccel` function updates the velocities using the accumulated accelerations. This process takes about 10% of the overall time.

**Barnes-Hut:** the `splitPoints`, `inBox` and `calcCentroid` functions are called within the `main` function which accounts for an inherited 33% of the time. The most time consuming function however is `calcAccel`. Depending on the threshold used to determine when to consider a body far enough, the time can varies. For example, if the threshold is high (closer to 1), the traversal is very fast, and this can actually produce no parallelism when a small number of bodies is used. A small threshold e.g. 0.1 runs slower. Ideally, 0.5 is used.

**Heap Profile**

Also important is to look at the heap consumption of each algorithm. This allows to find out if there is any space leak which can lead to stackoverflow. Most of the techniques used earlier to fix stackoverflow issues were done after viewing the heap profile of each algorithm.

Figure 4.2 shows the distribution of memory among the functions for each algorithm. Some points to note are:

**All-pairs:** this algorithm manages to get a pretty constant memory allocation due to the fact of using `foldl'` in updating the velocities. `foldl'` does not create a whole lot of thunks and so it helps to keep the memory usage constant.
4.2. IMPLEMENTATION

(a) Allpairs

(b) Triangular

(c) Barnes-Hut

Figure 4.2: Seq. Profiling
Triangular: the memory use is not good for this algorithm and this is attributed because the accelerations are accumulated and used in the end to update the velocities. This can be seen by the light gray shade on the graph.

Barnes-Hut: an entire tree hierarchy is built which explains the space consumption by the buildTree function. The remaining memory consumption happens during acceleration calculation.

4.2.5 Parallel implementation

With optimised sequential versions of our 3 algorithms, the time now is to add parallelism to the programs. And our approach is summed up by: Algorithm + Strategies = Parallelism [38].

The sequential profiling already helped in identifying which parts of the programs are the "big eaters". By parallelising these parts only, we achieve moderate to good performance already. The approach is to introduce annotations in the sequential code to achieve reasonable parallelism when the algorithm is run on more than one core.

Sequential implementation proceeded in a way to easily expose parallelism. For example, map or other high-order functions that are easily parallelised are used to make the process easy.

All-pairs: the simplest way to parallelise all-pairs algorithm is to use the parList strategy on the map function so that the velocity change for each body is computed in parallel.

\[
\text{updateVel ps vs ms = map (\i -> f (vs!i) i) ns \ 'using' \ parList}
\]

While the updatePos function performs a map operation too and each computation is independent from the others, it can be annotated as well. But profiling shows that this function hardly accounts for any time of the program. So it may be left out as annotating it does not cause any noticeable effect.

Triangular: the same technique is used for the triangular algorithm: we apply the parList strategy to the calcAccels function which is the most time consuming.

Barnes-Hut: the data parallelism in this algorithm arises primarily from the independent force calculations. So, the first focus is on parallelising the top level i.e. the calcAccels function. But the tree construction time itself is quite significant at 33% and therefore is also parallelised such that subtrees are processed in parallel during construction (buildTree). Note that the buildTree happens before calcAccels. So, the quicker the tree is constructed, the quicker we move to traversing it, where we add more parallelism there as well. The two functions can be seen as pipelined processes.

To run the parallel algorithm, it has to be compiled with the -threaded flag on. When running the program, the runtime option -Nx is used to specify the number of cores to be used, e.g. -N4
4.2. IMPLEMENTATION

runs on 4 cores.

Example:

Compile: `ghc --make -O2 -threaded allpairs-final.hs`

Run: `allpairs-final +RTS -N4`

In all 3 cases, the parallel implementations give performance gain and this is experienced by a decrease in runtime. Looking more into details about what happen when the algorithm is run on multiple cores, we add `-sstderr` to the run command. This produces the following output for the all-pairs algorithm, 1024 bodies and 20 time steps on 4 cores:

Listing 4.2: Par. run on 4 cores

```bash
./allpairs-final +RTS -N4 -sstderr
1.76512512e7
1947541.6497521228
time taken: 1.01s
3,242,755,560 bytes allocated in the heap
20,281,720 bytes copied during GC
1,901,640 bytes maximum residency (10 sample(s))
309,440 bytes maximum slop
8 MB total memory in use (0 MB lost due to fragmentation)

Generation 0: 1576 collections, 1575 parallel, 0.41s, 0.11s elapsed
Generation 1: 10 collections, 10 parallel, 0.09s, 0.03s elapsed

Parallel GC work balance: 1.48 (2487998 / 1681035, ideal 4)

<table>
<thead>
<tr>
<th>Task</th>
<th>MUT time (elapsed)</th>
<th>GC time (elapsed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.83s ( 0.87s)</td>
<td>0.15s ( 0.05s)</td>
</tr>
<tr>
<td>1</td>
<td>0.84s ( 0.87s)</td>
<td>0.15s ( 0.04s)</td>
</tr>
<tr>
<td>2</td>
<td>0.00s ( 0.87s)</td>
<td>0.00s ( 0.00s)</td>
</tr>
<tr>
<td>3</td>
<td>0.00s ( 0.87s)</td>
<td>0.00s ( 0.00s)</td>
</tr>
<tr>
<td>4</td>
<td>0.89s ( 0.87s)</td>
<td>0.10s ( 0.02s)</td>
</tr>
<tr>
<td>5</td>
<td>0.90s ( 0.87s)</td>
<td>0.09s ( 0.03s)</td>
</tr>
</tbody>
</table>

SPARKS: 345 (317 converted, 23 pruned)

INIT time 0.00s ( 0.01s elapsed)
MUT time 3.41s ( 0.87s elapsed)
GC time 0.50s ( 0.14s elapsed)
EXIT time 0.00s ( 0.00s elapsed)
Total time 3.91s ( 1.02s elapsed)

\%GC time 12.8\% (13.4\% elapsed)

Alloc rate 952,215,487 bytes per MUT second
Productivity 87.2\% of total user, 332.6\% of total elapsed

The above output shows that 345 sparks are created and 317 converted i.e. taken up for execution in a separate thread. The workload is evenly distributed to the 4 thread workers which take approximately 0.85 seconds to complete. The garbage collection (GC) time accounts for 12.8\% of the runtime. Note that since a small number of bodies is used and runtime is too small (1.01sec), the work balance is 1.48 while ideally it should be 4. Larger number of bodies shows better work balance as we will see in the results and evaluation.

4.2.6 Performance tuning

The parallel implementations are tuned in order to improve performance.

Chunking

Adding a new thread to the spark pool for each element of a list can be useless especially for small computation on the elements. Chunk size is used in order to avoid sparking a parallel thread for each element in a list. This can be too fine-grained and instead the list is broken into chunks which can then be picked up by parallel threads.

\[
\text{chunksize} = (\text{fromInteger } n) \div (\text{numCapabilities} \times 2)
\]

\[
\text{updateVel ps vs ms} = \text{map } (\lambda i \rightarrow f (\text{vs!i}) i) \text{ ns `using'}
\]

\[
\text{parListChunk chunksize rdeepseq}
\]

The difference between the previous annotation is that we use \text{parListChunk} instead of \text{parList}. It sequentially applies a strategy to chunks (sub-sequences) of a list in parallel and there useful to increase grain size.

The chunk size is obtained by dividing the number of bodies by twice the number of cores, \text{numCapabilities}, specified by the \text{-N} while running the code.

Threadscope

Threadscope gives a picture of the distribution of work among the available cores when the parallel program runs. It allows to spot performance issues relating to garbage collection or poor load balancing. For example, initially the parallelism for the BH algorithm was not well understood. Through threadscope, we were able to see what was really happening.

Threadscope is done for:

1) 1024 bodies, 1 time step: to see the work distribution for one iteration. Figure 4.3 shows that single step in the all-pairs algorithm produces good parallelism (green) with very little garbage collection (orange) at regular intervals. The triangular version, however, requires a lot of GC as it uses up more memory space. In Barnes-Hut, the tree construction phase is carried
4.2. IMPLEMENTATION

(a) Allpairs

(b) Triangular

(c) Barnes-Hut

Figure 4.3: Threadscope (1024 bodies, 1 step)
CHAPTER 4. ALGORITHMS AND STRUCTURES FOR THE N-BODY PROBLEM

(a) Allpairs

(b) Triangular

(c) Barnes-Hut

Figure 4.4: Threadscope (1024 bodies, 20 steps)
out mainly by processor #4 and then the tree traversal to calculate accelerations are distributed rather equally.

2) 1024 bodies, 20 time steps: to see the overall work distribution. The main thing to note here is that after each iteration (time step) in all 3 algorithms, GC is performed. The all-pairs achieves good productivity (87.2%) and less GC (12.8%) compared to the other two which spend over 50% of the time garbage collecting. This is also shown in Figure 4.5 which summarises the output of standard error (with -stderr flag on) and complements the information in Figure 4.4.

Another point worth noting is the large number of sparks created for the Barnes-Hut algorithm. This is due to the introduction of parallelism at tree construction level. The number of sparks generated at the tree construction varies depending on the tree hierarchy. Each spark consists of small computation and would most of the time by pruned and not taken for parallel execution. But since it does bring in some performance gain in runtime, and at nearly no memory cost, the large number of sparks is preferred here. It also shows that the cost of sparking computation is very low.

### 4.3 Related implementations

The implementations presented so far uses GpH and for parallelism on shared-memory via GHC-SMP. We look at related implementations undertaken by researchers using different approach in Haskell.

#### 4.3.1 Eden

The Eden Group at Philipp University of Marburg presented all-pairs and Barnes-Hut implementations of the nbody problem at the SICSA Multicore Challenge. Eden is based on Haskell and allows explicit parallelism structures using processes, channels and remote data; unlike GHC-SMP, it is for distributed memory architecture. The original version presented uses PVM and MPI for the all-pairs and Barnes-Hut algorithm respectively.

We obtained executable of the Eden code which was ran on the department’s multi-core cluster and measurements were taken for comparison.
4.3.2 Eval and Par monads

Marlow provides an all-pairs implementation using the Eval monad [26] and explicitly using \texttt{fork} and \texttt{join} in order to have more control of specifying the parallelism (unlike using annotations in our approach).

Marlow also provides all-pairs implementation using the Par monad [27]. Par monad presents a new programming model for deterministic parallel computation in Haskell. Unlike Control.Parallel, in Control.Monad.Par parallelism is not combined with laziness, so sharing and granularity are completely under the control of the programmer. New units of parallel work are only created by \texttt{fork}, \texttt{par}, and a few other combinators. The implementation is based on a work-stealing scheduler that divides the work as evenly as possible between the available processors at runtime.

Measurements for his two implementations are also taken on our cluster for our comparison.

4.4 Results and evaluation

Measurement

It is important that the comparison is made on measurements obtained from running the different implementations on machines with similar configurations. The system used is the multi-core machine \texttt{lxpara} in the department with the following setup:

\begin{verbatim}
Intel Xeon CPU E5410 @ 2.33 GHz
8 cores
8GB RAM (Linux)
\end{verbatim}

The initial algorithms performed 20 iterations and used 1024 bodies and the time included offsetting momentum at start of simulation, calculation of initial energy and energy after \( n \) iterations. To compare with the other implementations, only the main part of the program i.e. the iterations are measured and all other parts e.g. reading input, computing energy before and after are not included.

Comparison is made for similar algorithm e.g. all-pairs implementations using GHC-SMP, Eden, Eval/Par monads are compared, while BH implementations using GHC-SMP and Eden are compared. There is no similar comparison for the triangular algorithm but we do compare it with our all-pairs to point out the differences.

We take measurements for 16k bodies for both the all-pairs and BH algorithm. But the time step is only 1 for all-pairs and 20 for BH, the main reason for this being that all-pairs is slower in general as it is not an approximation method as BH.

The Eden measurements were initially for 1024 bodies/20 iterations while the Eval/Par monads are specifically for 1 iteration and the number of bodies can be varied. All measurements
4.4. RESULTS AND EVALUATION

Figure 4.6: Allpairs measurements

<table>
<thead>
<tr>
<th></th>
<th>GHC-SMP</th>
<th>Eden</th>
<th>Eval monad</th>
<th>Par monad</th>
</tr>
</thead>
<tbody>
<tr>
<td>#PE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Seq</td>
<td>47.71</td>
<td>1.00</td>
<td>248.34</td>
<td>1.00</td>
</tr>
<tr>
<td>1</td>
<td>50.75</td>
<td>0.94</td>
<td>259.40</td>
<td>0.96</td>
</tr>
<tr>
<td>2</td>
<td>26.06</td>
<td>1.83</td>
<td>136.83</td>
<td>1.81</td>
</tr>
<tr>
<td>3</td>
<td>17.07</td>
<td>2.79</td>
<td>93.74</td>
<td>2.65</td>
</tr>
<tr>
<td>4</td>
<td>12.30</td>
<td>3.88</td>
<td>70.90</td>
<td>3.50</td>
</tr>
<tr>
<td>5</td>
<td>9.70</td>
<td>4.92</td>
<td>78.38</td>
<td>3.17</td>
</tr>
<tr>
<td>6</td>
<td>8.70</td>
<td>5.48</td>
<td>60.03</td>
<td>3.65</td>
</tr>
<tr>
<td>7</td>
<td>7.70</td>
<td>6.20</td>
<td>70.49</td>
<td>3.52</td>
</tr>
<tr>
<td>8</td>
<td>6.89</td>
<td>6.92</td>
<td>66.93</td>
<td>3.71</td>
</tr>
</tbody>
</table>

are taken again on the machine mentioned above.

All-pairs

Figure 4.6 shows the runtime and speedup for the all-pairs algorithm using GHC-SMP and the other approaches. Our implementation gives better performance compared to that of Eden and Eval monad. However, Par monad is faster and this is attributed mainly to the explicit manner of specifying parallelism as opposed to our approach.

The runtime and speedup graphs (Figure 4.7) shows GHC-SMP runtime is closer to Par monad compared to the other two implementations. Eden does not scale well on increasing number of processors (on a shared-memory system). This is expected but had we increased the number of cores to more than 8 i.e. using more than one computing node, it would probably perform better on distributed memory. Also, increasing the number of bodies would also probably improve its performance.

Triangular

A comparison between the all-pairs and triangular versions is made using a small set of bodies (1024) as larger set will take too long using the triangular algorithm. The results from the runtime and speedup graphs (Figure 4.8) shows the inefficiency of triangular algorithm in Haskell. In an imperative language, it would give better performance due to in-place update, but the need to accumulate the accelerations does not help to produce better performance than all-pairs.

Barnes-Hut

Figure 4.9 shows the measurements of BH for GHC-SMP and Eden. The parallel version of all-pairs achieved almost linear speedups. In the case of the parallel version of Barnes-Hut’s algorithm, the speedup (Figure 4.10) is not comparable to all-pairs. This is mainly attributed to the small computation of parallel components in the algorithm compared to larger computation in the all-pairs implementation. However, the results are better than that achieved for Eden.
Figure 4.7: Allpairs runtime and speedup graphs - 16k bodies, 1 time step
Figure 4.8: Triangular vs Allpairs runtime and speedup graphs - 1024 bodies, 20 time steps
Concluding in view of the results and evaluation of our algorithms, we note good performance compared to the other similar implementations. BH is a far more efficient algorithm, with less computation being done compared to all-pairs. This explains the low runtime but not so good speedup.

<table>
<thead>
<tr>
<th>#PE</th>
<th>GHC SMP Runtimes</th>
<th>Speedups</th>
<th>Eden Runtimes</th>
<th>Speedups</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq</td>
<td>7.53</td>
<td>1.00</td>
<td>5.01</td>
<td>1.00</td>
</tr>
<tr>
<td>1</td>
<td>7.99</td>
<td>0.94</td>
<td>12.34</td>
<td>0.41</td>
</tr>
<tr>
<td>2</td>
<td>5.38</td>
<td>1.40</td>
<td>8.84</td>
<td>0.57</td>
</tr>
<tr>
<td>3</td>
<td>4.35</td>
<td>1.73</td>
<td>7.85</td>
<td>0.64</td>
</tr>
<tr>
<td>4</td>
<td>3.85</td>
<td>1.96</td>
<td>7.21</td>
<td>0.70</td>
</tr>
<tr>
<td>5</td>
<td>3.64</td>
<td>2.07</td>
<td>8.64</td>
<td>0.58</td>
</tr>
<tr>
<td>6</td>
<td>3.41</td>
<td>2.21</td>
<td>9.74</td>
<td>0.52</td>
</tr>
<tr>
<td>7</td>
<td>3.32</td>
<td>2.27</td>
<td>9.69</td>
<td>0.52</td>
</tr>
<tr>
<td>8</td>
<td>3.16</td>
<td>2.38</td>
<td>9.70</td>
<td>0.52</td>
</tr>
</tbody>
</table>
Figure 4.10: Barnes-Hut runtime and speedup graphs - 16k bodies, 20 time steps
Chapter 5

Conclusion

5.1 Summary

The dissertation gives the motivations behind the need for parallel data structures and algorithms at the outset.

Chapter 3 covers the analysis of traditional list representation and attempts at coming up with alternative representations that could facilitate parallelising operations on the structure. A new parallel list library is implemented which has a number of basic, advanced and other operations. We also provide benchmark applications for the new structure and comparative implementations in standard list. The performance of these applications are measured and results discussed.

Chapter 4 looks at the n-body problem implementation using three different algorithms. Sequential implementations are given for all algorithms which are then profiled and parallelised using strategies. We look at the various optimisations performed at the algorithm level and the use of an advanced tree structure: octree for 3D space simulation.

5.2 Evaluation

The project represented some challenges from the outset as it involved looking at structures and algorithms in a functional context. On top of that, we looked at parallel computing which is generally more difficult to program than sequential code. But we have seen that functional programming helped a lot in hiding many details of parallel computation that we would have otherwise to specify had we chosen an imperative language. Implementations of both the data structures and algorithms parts of the project have seen a high-level functional approach of specifying parallelism through parallel strategies. This has proved to be a quick and efficient way of writing parallel code. This also turned out to be the right choice we made at the outset as the results obtained in evaluating the two main chapters are very promising.
Looking back at the checklist we set early on in the introduction, all tasks have been covered to a very good level and the comparison at the end of Chapter 3 and 4 indicates the positive results from evaluation. So, we met all requirements set forth. We even managed to overcome one of the major risks we identified in the research report: the possibility of not getting any performance from the parallel library. We have seen that linear problem decomposition can be hard to parallelise and the alternative representation of lists proved to be efficient even though with an initial construction cost but this is the price we pay, but still gain in actual operation runtime.

We admit that everything did not go completely as planned. But the main tasks as originally planned were carried out in the order e.g. data structures implementations followed by algorithms for n-body problem. Towards the end, techniques learnt from the first part were useful in the second and same applied in the other order. Chapter 4 provides more details in the steps of parallelisation e.g. profiling, optimisation and tuning. These were also carried out for the data structures even though the focus was mainly on representation. With lack of experience at the beginning, many mistakes were done and in many cases we wrote our own functions to do a particular operation in Haskell. It was a good starting point but after time, we start to identify common patterns in the program. And we start making use of advanced inbuilt high-order and generic functions in Haskell (which are already very optimised). As a comparison, the initial source for the all-pairs implementation contained more lines of code compared to the final version.

The issues we faced were mainly workload on the department’s multi-core cluster to take measurements. We also received the Eden executable during the last week which resulted in taking last minute measurements. However, the scripts we wrote to automate many of the tasks e.g. running the program on 1 to 8 cores and then collecting the results and plot the graph helped enormously in reducing the time in doing these tedious job.

5.3 Contribution

The contribution of the project work can be summed up as:

- the implementation of a new parallel list library in a function language. The library offers parallel operations and internally uses an alternative representation than the usual sequential list. The online documentation of the library is available at http://www.macs.hw.ac.uk/~pt114/msc-project/ralist-doc/.
- achieving good performance on commonly used applications e.g. sum and high-order functions through the new library. The use of the new library for certain applications can be beneficial.
- benchmark applications for both the implemented data structure and for standard list.
• parallel implementations of the n-body problem which is a fundamental problem in many area of science using direct body-to-body calculations (all-pairs) and an advanced algorithm (BH).

• apply various optimisations to the algorithms to come up with very efficient implementations.

• compare the results with implementation from other researchers including the Eden Group and Microsoft Research (Eval/Par monads).

• provide a whole set of measurements for both the data structures and algorithms implementation.

• come up with a number of scripts to automate running sequential and parallel programs, and collecting runtimes and calculate speedups, and finally plot the graphs. All these are carried out in 3 scripts execution. All these are available through the project webpage.

5.4 Further work

There is more work to be carried out which would be a good starting point for PhD level research. The following summarises these work:

• for the parallel list library, we hope to cover a wider range of functions in the future. The library requires further polishing and can be uploaded to Hackage, the Haskell libraries repository.

• investigate other possible representation that could be even more efficient than random-access list. This may require building a completely new structure that is specifically aimed for parallel processing.

• we have seen the nub function among our benchmark applications but we did not use it in the evaluation. The reason is because it is still work-in-progress as we attempted to use circular data structure in the implementation. A circular data structure makes some kind of dependency on the structure itself. An initial version works with the RAList but a more advanced (and efficient) version does not work. The reason appears to be because of the use of strictness annotation in the data type definition but circular data structure depends on laziness. This is one thing that we need to look at.

• for the n-body problem, we want to do more comparison against other implementations presented in the SICSA Multicore Challenge, for e.g. implementations using Fortran, Vector Pascal, Intel Threading Building Blocks, Google Go and C# (by Hans-Wolfgang). The results can be put together and publish in a paper.
one more thing we want to look at is parallelising transformation from one data structure to another e.g. list to array. We use a function called `toArr` to convert a list to array which is sequential. A parallel variant can reduce the time in this process.

and finally, updating the MSc webpage with new results and measurements that we will be taking after the project.
Bibliography


Appendix A

Source listings

The project webpage at http://www.macs.hw.ac.uk/~pt114/msc-project/ contains a complete listing of all source including old versions, measurements, scripts and link to the RAList and ParTree modules online documentation.

A.1 List data structure

A.1.1 RAList module source

module RAList (  
    -- * Types
    RAList ( .. ) ,  
    -- * Basic operations
A number of basic operations on the new structure are implemented in order to support other main functions.

- RAList.\texttt{head}, RAList.\texttt{tail}, \texttt{empty}, \texttt{isEmpty}, RAList.\texttt{length},
- \texttt{* Conversion operations}
- \texttt{fromDataList}, \texttt{toDataList}, \texttt{toDataList}',
- \texttt{* More advanced operations}
- \texttt{RAList.lookup}, \texttt{update}, RAList.\texttt{elem}, \texttt{pelem}, \texttt{pelem}',
- \texttt{RAList.take}, RAList.\texttt{drop}, RAList.\texttt{filter}, RAList.\texttt{partition},
- \texttt{* Map}
- \texttt{ralMap}, \texttt{pRalMap}, \texttt{pRalMap}',
- \texttt{* Fold}
- RAList.\texttt{fold}, RAList.\texttt{parfold}, RAList.\texttt{parfold}',
- \texttt{* Parallel strategies}
- \texttt{Strategies are defined for Haskell's list in the module 'Control.Parallel.Strategies'.}
- \texttt{Following are probably the 2 mostly used strategies.}
- evalRAList, parRAList,
- \texttt{* Benchmark applications}
- \texttt{The following are implemented so that a direct comparison can be made with similar implementation in standard list found in the StdList module.}

- ** \texttt{Sum}
- RAList.\texttt{sum}, RAList.\texttt{psum}, RAList.\texttt{psum}',
- ** \texttt{Factorial}
- RAList.\texttt{facto}, RAList.\texttt{pfacto},
- ** \texttt{Sort}
- RAList.\texttt{quicksort},
- ** \texttt{Histogram}
- RAList.\texttt{histo},
- ** \texttt{Minimum}
- pRalMin, pRalMin',
- ** \texttt{Nub}
- \texttt{nub, nub1, nub2}

where
import ParTree
import Prelude hiding (\texttt{head}, \texttt{tail}, \texttt{lookup}, \texttt{length}, \texttt{filter}, \texttt{take}, \texttt{drop}, \texttt{elem})
import Prelude as Pre (\texttt{head}, \texttt{tail}, \texttt{lookup}, \texttt{length}, \texttt{filter}, \texttt{take}, \texttt{drop})
import System
import System.IO
import Data.List as List (\texttt{foldr, foldl, foldl'}, \texttt{sum, filter, reverse, map, or})
import Data.Foldable hiding (minimum)
import Control.DeepSeq
import Control.Parallel
import Control.Parallel.Strategies
import GHC.Conc (numCapabilities)

RAList is the new data type to be used instead of list. It is defined as a list of complete binary trees (implemented in the module @ParTree@).

> data RAList a = RAL ! Int ![Digit a]
> data Digit a = Zero | One !(Tree a)
> data Tree a = Leaf a | Node ! Int !(Tree a) !(Tree a) --- Defined in ParTree module.

data RAList a = RAL ! Int ![Digit a] deriving Show
data Digit a = Zero | One !(Tree a) deriving Show

Providing NFData instances for new data types defined above.

instance (NFData a) ⇒ NFData (RAList a) where
  rnf (RAL s ds) = deepseq ds ()

instance (NFData a) ⇒ NFData (Digit a) where
  rnf (Zero) = ()
  rnf (One t) = deepseq t ()

These functions are not exported and used in this module only.

Construct tree.
First argument is the new element to add to the structure.

consTree t [] = [One t]
consTree t (Zero : ts) = One t : ts
consTree t1 (One t2 : ts) = Zero : consTree (link t1 t2) ts

Unconstruct tree.
unconsTree [] = error "RAList: empty list"
unconsTree [One t] = (t, [])
unconsTree (One t : ts) = (t, Zero : ts)
unconsTree (Zero : ts) = (t1, One t2 : ts')
  where (Node t1 t2, ts') = unconsTree ts

Take a list of 'Digit' and return all the trees in a new list.

filterTrees :: [Digit a] -> [Tree a]
filterTrees ts = List.reverse $ checkT [] ts
  where
    checkT ys [] = ys
    checkT ys (Zero:xs) = checkT ys xs
    checkT ys ((One t):xs) = checkT (t:ys) xs
--- |\(O(1)\) Similar to `cons` in list \@(_:@). It prepends an element to the RAList.

\[
\text{cons} :: a \to \text{RAList} a \to \text{RAList} a \\
\text{cons} x (\text{RAList} s ts) = \text{RAList} (s+1) (\text{consTree} (\text{Leaf} x) t s)
\]

--- |\(O(1)\) Return the first element from the RAList.

\[
\text{head} :: \text{RAList} a \to a \\
\text{head} (\text{RAList} s ts) = \text{let} \ (\text{Leaf} x, _) = \text{unconsTree} t s \ \text{in} \ x
\]

--- |\(O(1)\) Return all elements, except the first.

\[
\text{tail} :: \text{RAList} a \to \text{RAList} a \\
\text{tail} (\text{RAList} s ts) = \text{let} \ (_, ts') = \text{unconsTree} t s \ \text{in} \ \text{RAL} (s-1) ts'
\]

--- An empty RAList where the size is 0 hence indicating that
--- the structure is empty.

--- |\(O(1)\) Check if RAList is empty, that is, size @s@ equals to 0 in:

\[
\text{isEmpty} :: \text{RAList} a \to \text{Bool} \\
\text{isEmpty} (\text{RAL} s ts) = s == 0 ---or, we could do: \text{null} ts
\]

--- |\(O(1)\) Return the number of elements in the RAList. The corresponding
--- function \@\text{length}\@ in standard list has a time complexity of /\(O(n)\)/.

\[
\text{length} :: \text{RAList} a \to \text{Int} \\
\text{length} (\text{RAL} s ts) = s
\]

--- Take the first \@n\@ elements.

\[
\text{take} :: \text{Int} \to \text{RAList} a \to \text{RAList} a \\
\text{take} n \ _ \ | \ n <= 0 = \text{empty} \\
\text{take} \ _ \ (\text{RAL} 0 \ []) = \text{empty} \\
\text{take} n \text{ list} = x \ \text{`cons` \ RAList. take} (n-1) \text{ xs} \\
\text{where} \\
x = \text{RAList. head} \text{ list} \\
xs = \text{RAList. tail list}
\]

--- Drop the first \@n\@ elements.

\[
\text{drop} :: \text{Int} \to \text{RAList} a \to \text{RAList} a \\
\text{drop} n \text{ list} \ | \ n <= 0 = \text{list} \\
\text{drop} \ _ \ (\text{RAL} 0 \ []) = \text{empty} \\
\text{drop} n \text{ list} = \text{RAList. drop} (n-1) \text{ xs} \\
\text{where} \\
x = \text{RAList. tail list}
A.1. LIST DATA STRUCTURE

--- | **Build a RAList from a list.**
fromDataList :: [a] -> RAList a
fromDataList xs = List.foldl' (flip cons) empty (List.reverse xs)

--- | **Convert a RAList to a list.**
toDataList :: RAList a -> [a]
toDataList list@(RAList s _) =
  if s == 0 then []
else (RAList.head list) : toDataList (RAList.tail list)

--- | **/Par./ An alternative (parallel) implementation of ‘toDataList’.**
toDataList’ :: NFData a => RAList a -> [a]
toDataList’ (RAList ts) = List.foldl (++) [] lists
  where
trees = filterTrees ts
lists = map treeToList trees ‘using’ parList rdeepseq

--- | **/O(log n)/ Lookup the RAList using the given index like @(!!)@ in standard list. Index has to be in range, otherwise an error occurs.**
lookup :: Int -> RAList a -> a
lookup i (RAList ts) = look i ts
  where
    fail = error "RAList.lookup:_bad_subscript"
    look i [] = fail
    look i (Zero : ts) = look i ts
    look i (One t : ts) =
      if i < size t then lookTree i t else look (i - size t) ts
    lookTree 0 (Leaf x) = x
    lookTree i (Leaf x) = fail
    lookTree i (Node w t1 t2) =
      if i < w ‘div’ 2 then lookTree i t1
      else lookTree (i - w ‘div’ 2) t2

--- | **/O(log n)/ Update the element at the given index with a new value.**
update :: Int -> a -> RAList a -> RAList a
update i y (RAList ts) = RAList s (upd i ts)
  where
    fail = error "RAList.update:_bad_subscript"
    upd i [] = fail
    upd i (Zero : ts) = Zero : upd i ts
    upd i (One t : ts) =
      if i < size t then One (updTree i t) : ts
      else One t : upd (i - size t) ts
updTree 0 (Leaf x) = Leaf y
updTree i (Leaf x) = fail
updTree i (Node w t1 t2) =
  if i < w `div` 2 then Node w (updTree i t1) t2
  else Node w t1 (updTree (i - w `div` 2) t2)

— Check for any occurrence of the given element in the structure. Seq implementation
— similar to ‘elem’ from standard list.
elem :: Eq a => a -> RAList a -> Bool
elem x ts =
  if RAList.length ts == 0 then False
  else if x == RAList.head ts then True
       else RAList.elem x $ RAList.tail ts

— /Par./ parallel implementation of ‘elem’.
pelem :: Eq a => a -> RAList a -> Bool
pelem x (RAL [] ) = False
pelem x (RAL s (Zero : ts)) = pelem x (RAL s ts )
pelem x (RAL s (One t : ts)) = l `par` r `seq` (l || r)
  where
  l = treeElem x t
  r = pelem x (RAL s ts )

— /Par./ improved parallel version.
pelem’ :: (Eq a, NFData a) => a -> RAList a -> Bool
pelem’ x (RAL _ ts) = List.or lists
  where
  trees = filterTrees ts
  lists = map (treeElem x) trees `using` parList rdeepseq

— Return a new RAList with only those elements that match the predicate.
filter :: (a -> Bool) -> RAList a -> RAList a
filter f ts =
  if RAList.length ts == 0 then empty
  else if f x then x `cons` RAList.filter f xs
       else RAList.filter f xs
  where
  x = RAList.head ts
  xs = RAList.tail ts

— Return 2 partitions of the given RAList. The first RAList in the tuple pair
— satisfies the predicate, while the second does not.
A.1. LIST DATA STRUCTURE

\textbf{partition} :: (a \to \text{Bool}) \to \text{RAList} a \to (\text{RAList} a, \text{RAList} a)
\text{partition} ~ p \; (\text{RAList} s \; []) = (\text{empty}, \text{empty})
\text{partition} ~ p \; \text{list} =
\text{if} \; p \; \text{head} \; \text{list} \; \text{then} \; (\text{cons} \; \text{head} \; \text{list}, \text{empty})
\text{else} \; (\text{empty}, \text{cons} \; \text{head} \; \text{list} \; \text{tail} \; \text{list})

--- HIGH-ORDER FUNCTIONS – Map and Fold

--- | \textit{Sequential} high-order map function which applies a function \textit{‘f’} to every element of the \text{RAList}.
---

--- > \text{ralMap} \; \text{sqrt} \; $\text{fromDataList} \; \{1,4,9\}$
--- > \text{RAL} 3 \; \{\text{One (Leaf 1.0), One (Node 2 (Leaf 2.0) (Leaf 3.0))}\}$
\text{ralMap} :: (a \to b) \to \text{RAList} a \to \text{RAList} b
\text{ralMap} \; f \; \text{ts} =
\text{if} \; \text{RAList} \; \text{length} \; \text{ts} \; == \; 0 \; \text{then} \; \text{empty}
\text{else} \; f \; (\text{RAList} \; \text{head} \; \text{ts}) \; \text{‘cons’} \; (\text{ralMap} \; f \; (\text{RAList} \; \text{tail} \; \text{ts}))

--- | \textit{/Par./ parallel implementation of map function on RAList, using ‘parRAList’}.

\text{pRalMap} :: \text{Strategy} b \to (a \to b) \to \text{RAList} a \to \text{RAList} b
\text{pRalMap} \; \text{strat} \; f = (\text{‘using’} \; \text{parRAList} \; \text{strat}) \; . \; \text{ralMap} \; f

--- | \textit{/Par./ another version that uses ‘treeMap’}.

\text{pRalMap}' :: (\text{NFData} b) \Rightarrow (a \to b) \to \text{RAList} a \to \text{RAList} b
\text{pRalMap}' \; f \; (\text{RAL} 0 \; []) = \text{empty}
\text{pRalMap}' \; f \; (\text{RAL} n \; \text{ts}) = \text{RAL} n \; \text{ts’}
\text{where}
\text{ts’} = \text{map} \; f \; \text{ts’} \; \text{‘using’} \; \text{parList} \; \text{rdeepseq}
\text{f’} (\text{Zero}) = \text{Zero}
\text{f’} (\text{One} \; t) = \text{One} \; $ \text{treeMap} \; f \; t

--- | \textit{Sequential fold on the RAList structure, same type signature as ‘foldl’ (folds element from the left)}.
---

--- > \text{fold} \; (+) \; 0 \; $\text{fromDataList} \; \{1..5\}$
\text{fold} :: (a \to b \to a) \to a \to \text{RAList} b \to a
\text{fold} \; f \; z \; \text{ts} =
\text{if} \; \text{isEmpty} \; \text{ts} \; \text{then} \; z
\text{else} \; \text{RAList} \; \text{fold} \; f \; (f \; z \; \text{RAList} \; \text{head} \; \text{ts}) \; (\text{RAList} \; \text{tail} \; \text{ts})

--- | \textit{/Par./ parallel implementation of fold function. The collection of trees in the RAList is folded in parallel and at tree level, more parallelism is
APPENDIX A. SOURCE LISTINGS

−− gained through the use of ‘treeFold’. A top level fold generates the final
−− result. The function ‘f’ has to be both associative and commutative.
parfold :: (NFData a) ⇒ (a → a → a) → a → RAList a → a
parfold f z (RAL ts) = res
  where
trees = filterTrees ts
res = Data.Foldable.fold f z (map (treeFold f z) trees 'using' parList
rdeepseq)

−− | /Par./ an alternative parallel version of fold on RAList where the
collection
−− of trees are processed in order and parallelism is at sub-structure (tree
level) only.
parfold’ :: (NFData a) ⇒ (a → a → a) → a → RAList a → a
parfold’ f z (RAL ts) = pfold z ts
  where
pfold z’ [] = z’
pfold z’ (One t : xs) = pfold (f z’ (treeFold f z t)) xs
pfold z’ (Zero : xs) = pfold z’ xs

−− STRATEGIES

−− | Strategy combinator that walks over the RAList and applies the
−− argument strategy ‘s’ to every element. It generalises ‘parRAList’.
−−
−− ‘seqRAList’ is not given, but can be easily defined by
−−
−− > seqRAList s = evalRAList (rseq ‘dot’ s)
evalRAList :: Strategy a → Strategy (RAList a)
evalRAList s (RAL 0 []) = return empty
evalRAList s ts = do x’ ← s x
  xs’ ← evalRAList s xs
  return (x’ ‘cons’ xs’)
  where
  x = RAList.head ts
  xs = RAList.tail ts

−− | ‘parRAList’ is obtained by composing the element strategy ‘s’ with ‘rpar
−− ‘.
−−
−− > parRAList s = evalRAList (rpar ‘dot’ s)
−−
−− Example
−−
−− > ralist = fromDataList [1..5]
−− > ralMap (+1) ralist ‘using’ parRAList rdeepseq
A.1. **LIST DATA STRUCTURE**

---

Or

---

> pRalMap rdeepseq (+1) ralist
parRAList :: Strategy a -> Strategy (RAList a)
parRAList s = evalRAList (rpar 'dot' s)

---

Benchmark Applications

---

| Sequential ‘sum’ function that uses ‘fold’.

    sum :: (Num a) => RAList a -> a
    sum ts = RAList.fold (+) 0 ts

---

| Parallel ‘sum’ function that uses ‘parfold’.

    psun :: (NFData a, Num a) => RAList a -> a
    psun ts = parfold (+) 0 ts

---

| Sequential ‘factorial’ function that uses ‘fold’.

    facto :: Integer -> Integer
    facto x | x < 0 = error "facto: Negative number!"
                | x == 0 = 1
                | otherwise = RAList.fold (*) 1 $ fromDataList [1..x]

---

| Parallel ‘factorial’ function that uses ‘parfold’.

    p facto :: Integer -> Integer
    p facto x | x < 0 = error "facto: Negative number!"
                | x == 0 = 1
                | otherwise = parfold (*) 1 $ fromDataList [1..x]

---

| Sort a RAList of n random numbers using the quicksort algorithm.

    quicksort :: (Ord a) => RAList a -> RAList a
    quicksort list = qsort list empty
    where
    qsort list rest =
        if isEmpty list then rest
        else
            let x = RAList.head list
            xs = RAList.tail list
            (smalls, bigs) = partition (<x) xs
            part (x, xs, empty, empty)
            in qsort smalls (cons x (qsort bigs rest))

---

| The ‘histogram’ function counts the occurrences of each integer in a list
of n integers chosen randomly from 0..n-1.

    histo :: Int -> [Int] -> RAList Int
his n randlist = Prelude.foldl inc (init n) randlist
  where
  init 0 = empty
  init n = cons 0 (init (n-1))

  inc ts i = update i ((RAList.lookup i ts)+1) ts

-- | /Par./ Return the smallest integer from the RAList.
 pRalMin :: (Num a, Ord a) => RAList a -> a
 pRalMin xs = findMin h t
  where
    h = RAList.head xs
    RAL t = RAList.tail xs

findMin :: (Num a, Ord a) => a -> [Digit a] -> a
findMin x [] = x
findMin x (Zero : ts) = min x (findMin x ts)
findMin x (One t : ts) = min (treeMin t) (findMin x ts)

-- | /Par./ improved parallel version. 'min' is commutative i.e. min x y = min y x,
-- so it can be used in 'fold'.
 pRalMin' :: (Num a, Ord a, NFData a) => RAList a -> a
 pRalMin' ralise0 (RAL ts) = List.foldl' min x mins
  where
    x = RAList.head ralise
    trees = filterTrees ts
    mins = map treeMin trees 'using' parList rdeepseq

-- | /Par./ The 'nub' function removes duplicate elements from the RAList.

Naive version.
 nub :: (Eq a) => RAList a -> [Digit a] -> RAList a
 nub acc [] = acc
 nub acc (Zero : ts) = nub acc ts
 nub acc (One t : ts) = nub (newacc t acc) ts
  where
    newacc (Leaf n) ac@(RAL s xx) = if pelem n (RAL 0 xx) then ac else cons n ac
    newacc (Node l r) ac = newacc r (newacc l ac)

-- | nub-circular.
 nub1 :: (Eq a) => RAList a -> RAList a
 nub1 ts =
  if RAList.length ts == 0 then empty
  else x 'cons' (nub1 (RAList.filter (/= x) xs))
  where
    x = RAList.head ts
    xs = RAList.tail ts
A.1. LIST DATA STRUCTURE

more advanced nub-circular. (Does not work).
nub2 :: (Eq a) => RAList a -> RAList a
nub2 ts = res
    where
        build (RAL 0 _) _ = empty
        build ts' n | mem x res n = build xs n
                      | otherwise = x 'cons' (build xs (n+1))
    where
        x = RAList . head ts'
        xs = RAList . tail ts'

mem _ _ 0 = False
mem x ts' ' n | x == y = True
               | otherwise = mem x ys (n-1)
    where
        y = RAList . head ts''
        ys = RAList . tail ts''

A.1.2 ParTree module source

--|--{-# LANGUAGE DeriveTraversable, DeriveFoldable, DeriveFunctor, CPP,
    BangPatterns #-}--

module ParTree (
    -- * Type Definition
    Tree (...),
    -- * Basics
    size, link, 
    -- * Parallel operations
    -- | All operations that follow are parallel implementations.
    treeToList, treeMap, treeFold, treeElem,
```haskell
import Data.Foldable
import Data.Traversable
import Control.DeepSeq
import Control.Parallel
import Control.Parallel.Strategies

−− | New data type for Tree with strictness annotations in the definition.
−−
−− > data Tree a = Leaf a | Node !Int !(Tree a) !(Tree a)
−−
−− | For example, a tree representation for the list [1,2,3,4]
−−
−− > myTree = (Node 4 (Node 2 (Leaf 1) (Leaf 2)) (Node 2 (Leaf 3) (Leaf 4)))
−−
data Tree a = Leaf a | Node !Int !(Tree a) !(Tree a) deriving Show
  (Traversable, Foldable, Functor, Show, Eq)

−− | This allows for Tree to be evaluated to normal form.
instance (NFData a) ⇒ NFData (Tree a) where
  rnf (Leaf x) = deepseq x ()
  rnf (Node s t1 t2) = deepseq (s,t1,t2) ()

−− | Find the size of a tree.
size :: Tree a −→ Int
size (Leaf _) = 1
size (Node s _) = s

−− | Link two trees, one to the left and one to the right, making a node.
link :: Tree a −→ Tree a −→ Tree a
link t1 t2 = Node (size t1 + size t2) t1 t2

−− | Convert a tree to list.
treeToList :: Tree a −→ [a]
treeToList (Leaf x) = [x]
treeToList (Node l r) = runEval $ do
  left ← rpar $ treeToList l
  right ← rseq $ treeToList r
  return (left ++ right)

−− | Map a function ‘f’ to each element of the tree in parallel, producing a
−− new tree. Similar to ‘map’ function in standard and ‘parMap’ for its
  parallel
−− equivalent.
```
A.1. LIST DATA STRUCTURE

\[\text{treeMap} :: (a \to b) \to \text{Tree } a \to \text{Tree } b\]
\[\text{treeMap } f \ (\text{Leaf } x) = \text{Leaf } (f\ x)\]
\[\text{treeMap } f \ (\text{Node } l\ r) = \text{runEval } \begin{array}{l}
\text{left }\leftarrow \text{rpar }\text{treeMap } f\ l \\
\text{right }\leftarrow \text{rseq }\text{treeMap } f\ r
\end{array}
\text{return } (\text{link }\text{left }\text{right})\]

— | \text{Parallel} ‘fold’ function on tree. The first function passed as argument needs to be both associative and commutative, e.g. sum and product operators.
— Both have type def @Num a \to a \to a
\[\text{treeFold} :: (\text{NFData } a) \Rightarrow (a \to a \to a) \to a \to a\]
\[\text{treeFold } f\ z\ (\text{Leaf } x) = f\ z\ x\]
— using ‘par’ and ‘seq’ primitives
— \text{treeFold } f\ z\ (\text{Node } s\ l\ r) = \text{if } s > 10000 \text{ then } l \text{ ‘par’ } r \text{ ‘seq’ } f\ l\ r\]
— else \text{f l r}
— using ‘runEval’ from new strategies (it fixes issue with space leak)
\[\text{treeFold } f\ z\ (\text{Node } s\ l\ r) = \text{if } s > 10000 \text{ then runEval } \begin{array}{l}
\text{left }\leftarrow \text{rpar }\text{left} \\
\text{right }\leftarrow \text{rseq }\text{right} \\
\text{return } (\text{f }\text{left }\text{right})
\end{array}\]
where
\[\text{left }= \text{treeFold } f\ z\ l\]
\[\text{right }= \text{treeFold } f\ z\ r\]

— | Check if there is any occurrence of the element in the tree.
— Search proceeds by inspecting subtrees e.g. left and right branches in parallel.
\[\text{treeElem} :: (\text{Eq } a) \Rightarrow a \to a \to \text{Tree } a \to \text{Bool}\]
\[\text{treeElem } e\ (\text{Leaf } x) = \text{if } x == e \text{ then True else False}\]
\[\text{treeElem } e\ (\text{Node } s\ l\ r) = \text{runEval } \begin{array}{l}
\text{left }\leftarrow \text{rpar }\text{treeElem } e\ l \\
\text{right }\leftarrow \text{rseq }\text{treeElem } e\ r
\end{array}
\text{return } (\text{left }||\text{right})\]

— | Reverse all the elements in the tree.
\[\text{treeReverse} :: \text{Tree } a \to \text{Tree } a\]
\[\text{treeReverse } (\text{Leaf } x) = (\text{Leaf } x)\]
\[\text{treeReverse } (\text{Node } s\ l\ r) = \text{runEval } \begin{array}{l}
\text{left }\leftarrow \text{rpar }\text{treeReverse } r \\
\text{right }\leftarrow \text{rseq }\text{treeReverse } l
\end{array}
\text{return } (\text{link }\text{left }\text{right})\]

— | Return the smallest element from the tree.
\[\text{treeMin} :: (\text{Ord } a) \Rightarrow a \to \text{Tree } a \to a\]
\[\text{treeMin } (\text{Leaf } x) = x\]

81
APPENDIX A. SOURCE LISTINGS

treeMin (Node l r) = runEval $ do
  left <- rpar $ treeMin l
  right <- rseq $ treeMin r
  return (min left right)

A.1.3 StdList module source

module StdList {
  -- * Seq. functions
  -- | Sequential operations on standard list.
  histo, histo', qsort, update,
  -- * Par. functions
  -- | Parallel operations on standard list.
  parsum, parmin, parelem
  where
  import Data.List
  import Control.DeepSeq
  import Control.Parallel
  import Control.Parallel.Strategies

  -- Seq. functions
  histo :: Int -> [Int] -> [Int]
  histo n randlist = map (\i -> length $ filter (== i) randlist) [0..n-1]

  histo' :: Int -> [Int] -> [Int]
  histo' n randlist = foldl inc (replicate n 0) randlist
    where
      inc xs i = a ++ [y] ++ tail b
      where
        y = head b + 1
        (a,b) = splitAt i xs

  qsort :: Ord a => [a] -> [a]
  qsort [] = []
  qsort [x] = [x]
  qsort (x:xs) = losort ++ (x:hisort)
    where
      losort = qsort [y|y<-xs, y<x]
      hisort = qsort [y|y<-xs, y>=x]
A.2 N-body problem algorithms

A.2.1 Full all-pairs algorithm

{-# LANGUAGE CPP, BangPatterns #-}

-- File: allpairs-final.hs
-- Description: nbody computation: full all-pairs version
-- Author: Prabhat Totoo

-- The algorithm is different from allpairs.hs as it does not
-- offset momentum and calculate energy before and after simulation.
-- It is intended to measure the time for N steps of the simulation.
-- Bodies are generated in the code but this is not included in the time.
APPENDIX A. SOURCE LISTINGS

--- Compile | Run
--- Seq: > ghc --make -O2 allpairs-final.hs | allpairs-final
--- Par: > ghc --make -O2 -threaded -rtsopts allpairs-final.hs | allpairs-final
+RTS -Nz

--- Runtimes/Output
--- Machine: Intel Xeon CPU E5410 @ 2.33GHz, 8 cores, 8GB RAM (Linux)
--- 16k bodies, 1 iteration
--- #PE  Runtimes  Speedups
--- Seq  47.71  1
---  1   50.75  0.94
---  2   26.06  1.83
---  3   17.07  2.79
---  4   12.30  3.88
---  5   9.70   4.92
---  6   8.70   5.48
---  7   7.70   6.20
---  8   6.89   6.92

import IO
import System
import System.Time
import Text.Printf
import Array
import Data.List
import Control.Parallel
import Control.Parallel.Strategies
import Control.DeepSeq
import GHC.Exts
import GHC.Conc (numCapabilities)

data M = M {-# UNPACK #-} ! Double
data P = P {-# UNPACK #-} ! Double {-# UNPACK #-} ! Double {-# UNPACK #-} ! Double
data V = V {-# UNPACK #-} ! Double {-# UNPACK #-} ! Double {-# UNPACK #-} ! Double

instance NFData (M) where
  rnf (M m) = deepseq m ()
instance NFData (P) where
  rnf (P x y z) = deepseq (x, y, z) ()
instance NFData (V) where
  rnf (V vx vy vz) = deepseq (vx, vy, vz) ()

chunksize = (fromInteger n) `quot` (numCapabilities * 2)

n = 16000
ns = [0..(n-1)]
arrbounds = (0,(n-1))
toArr xs = listArray arrbounds xs


\[ \text{timeStep} = 0.001 \]

\[ \text{-- epsilon} \]
\[ \epsilon = 0.01 \]

\[
\text{genPos tag} = P \left( \text{tag}' \times 1.0 \right) \left( \text{tag}' \times 1.2 \right) \left( \text{tag}' \times 0.8 \right)
\]
\[ \text{where tag}' = \text{fromIntegral tag} \]

\[
\text{genVel tag} = V \left( \text{tag}' \times 0.1 \right) \left( \text{tag}' \times 0.2 \right) \left( \text{tag}' \times 0.4 \right)
\]
\[ \text{where tag}' = \text{fromIntegral tag} \]

\[
\text{genMass tag} = M \left( \text{tag}' \times 30.0 \right)
\]
\[ \text{where tag}' = \text{fromIntegral tag} \]

\[
\text{main} = \\
\text{do}
\text{let}
\ms = \text{toArr} \left[ \left( \text{genMass i} \right) \mid i \leftarrow [0..n-1] \right] \\
\pos = \text{toArr} \left[ \left( \text{genPos i} \right) \mid i \leftarrow [0..n-1] \right] \\
\vel = \text{toArr} \left[ \left( \text{genVel i} \right) \mid i \leftarrow [0..n-1] \right] \\
\text{let res1} = \left( \text{foldl} \ \left( \sum \right) 0 \ 
s \right) \\
\text{-- do meaningless sum...}
\text{print (res1)} \\
\text{t1 } \leftarrow \text{getClockTime} \text{ -- so that, at this point, the positions, velocities}
\text{\quad and masses have been read from file}
\text{let (pos',vel')} = \text{nSteps} 1 \left( \text{pos}, \text{vel}, \ms \right) \\
\text{let res2} = \left( \text{foldl} \ \left( \sum \right) 0 \ 
s \right) \\
\text{-- do a meaningless sum to generate a small output}
\text{print (res2)} \\
\text{t2 } \leftarrow \text{getClockTime}
\text{putStrLn} \$ \text{printf "time taken: %.2fs" } \$ \text{secDiff t1 t2}
\]

\[
g \left( P \ x \ y \ z \right) \left( V \ vx \ vy \ vz \right) \left( M \ m \right) = x+y+z+vx+vy+vz+mn
\]

\[
f \left( P \ x \ y \ z \right) \left( V \ vx \ vy \ vz \right) = x+y+z+vx+vy+vz
\]

\[
n \text{Steps} 0 \left( \ps, \vs, \ms \right) = \left( \ps, \vs \right)
\]

\[
n \text{Steps} n \left( \ps, \vs, \ms \right) = n \text{Steps} \left( n-1 \right) \text{ step}
\]
\[ \text{where} \]
\[ \text{step} = \text{oneStep} \left( \ps, \vs, \ms \right) \]

\[
\text{oneStep \left( \ps, \vs, \ms \right)} = \left( \text{new}_\ps, \text{new}_\vs, \ms \right)
\]
\[ \text{where} \]
\[ \text{new}_\ps = \text{updatePos} \ps \ \text{new}_\vs \]
\[ \text{new}_\vs = \text{updateVel} \ps \ \vs \ \ms \]
updateVel ps vs ms = toArr (\i -> f (vs!i) i) ns 'using' parListChunk
  chunksize rdeepseq)
where
  f v i = foldl' (deductChange) v (\j -> g i j) ns
  g i j
    | i == j = V 0 0 0
    | otherwise = V (dx * mass_j * mag) (dy * mass_j * mag) (dz * mass_j * 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
  P ix iy iz = ps!i
  P jx jy jz = ps!j
  M mass_j = ms!j

updatePos ps vs = toArr (\i -> updateP (ps!i) (vs!i)) ns 'using'
  parListChunk chunksize rdeepseq)
where
  updateP (P x y z) (V vx vy vz) = P (x + timeStep * vx) (y + timeStep * vy) (z + timeStep * vz)

deductChange (V vx1 vy1 vz1) (V vx2 vy2 vz2) = V (vx1-vx2) (vy1-vy2) (vz1-vz2)

-- func to calc time taken between t0 and t1 in sec
secDiff :: ClockTime -> ClockTime -> Float
secDiff (TOD secs1 psecs1) (TOD secs2 psecs2) = fromInteger (psecs2 - psecs1) /
  1e12 + fromInteger (secs2 - secs1)

A.2.2 Triangular algorithm

{-# LANGUAGE CPP, BangPatterns #-}

-- File: triangular.hs
-- Description: nbody computation: triangular all-pairs version
-- Author: Prabhat Totoo

-- The algorithm performs n iterations and does a meaningless sum
-- to force evaluate the updated positions and velocities at the
-- end of simulation.
-- Bodies are generated in the code but this is not included in the time.

-- Compile | Run
-- Seq: > ghc --make -O2 triangular-final.hs | > triangular-final
A.2. N-BODY PROBLEM ALGORITHMS

--- Par: > ghc --make -O2 -threaded -rtsopts triangular-final.hs | > triangular-
final +RTS -Nz

--- Runtimes/Output
--- Machine: Intel Xeon CPU E5410 @ 2.33GHz, 8 cores, 8GB RAM (Linux)
--- 1024 bodies, 20 iterations
--- #PE Run times Speedups
--- Seq 18.01 1
--- 1 18.61 0.97
--- 2 18.04 1.00
--- 3 16.02 1.12
--- 4 14.85 1.21
--- 5 13.94 1.29
--- 6 13.18 1.37
--- 7 12.86 1.40
--- 8 12.60 1.43

import IO
import System
import System
import System.Time
import Text.Printf
import Array
import Data.List
import Control.Parallel
import Control.Parallel.Strategies
import Control.DeepSeq
import GHC.Exts
import GHC.Conc (numCapabilities)

data M = M {−# UNPACK #−} !Double
data P = P {−# UNPACK #−} !Double {−# UNPACK #−} !Double {−# UNPACK #−} !Double
data V = V {−# UNPACK #−} !Double {−# UNPACK #−} !Double {−# UNPACK #−} !Double

instance NFData (M) where
  rnf (M m) = deepseq m ()
instance NFData (P) where
  rnf (P x y z) = deepseq (x, y, z) ()
instance NFData (V) where
  rnf (V vx vy vz) = deepseq (vx, vy, vz) ()
chunksize = n ‘quot’ (numCapabilities * 2)

n = 16000
ns = [0..(n-1)]
arrbounds = (0, (n-1))
toArr xs = listArray arrbounds xs
zeroAccels = listArray arrbounds (replicate n (V 0 0 0))

timeStep = 0.001
--- epsilon
eps = 0.01

genPos tag = P (tag' * 1.0) (tag' * 1.2) (tag' * 0.8)
  where tag' = fromIntegral tag

genVel tag = V (tag' * 0.1) (tag' * 0.2) (tag' * 0.4)
  where tag' = fromIntegral tag

genMass tag = M (tag' * 30.0)
  where tag' = fromIntegral tag

main =
do

  let
    ms = toArr $ [ (genMass i) | i <- [0..n-1] ]
    pos = toArr $ [ (genPos i) | i <- [0..n-1] ]
    vel = toArr $ [ (genVel i) | i <- [0..n-1] ]
  let res1 = (foldl (\sum i -> sum + g (pos!i) (vel!i) (ms!i)) 0 ns) -- do meaningless sum...
    print (res1)
    t1 <- getClockTime -- ...so that, at this point, the positions, velocities
      and masses have been read from file
    let (pos',vel') = nSteps 1 (pos,vel,ms)
    let res2 = (foldl (\sum i -> sum + f (pos!i) (vel!i)) 0 ns) -- do a
      meaningless sum to generate a small output
    print (res2)
    t2 <- getClockTime
    putStrLn $ printf "time taken: %2fs" $ secDiff t1 t2

  g (P x y z) (V vx vy vz) (M m) = x+y+z+vx+vy+vz+m
  f (P x y z) (V vx vy vz) = x+y+z+vx+vy+vz

nSteps 0 (ps,vs,ms) = (ps,vs)
nSteps n (ps,vs,ms) = nSteps (n-1) step
  where
    step = oneStep (ps,vs,ms)

oneStep (ps,vs,ms) = (new_ps,new_vs,ms)
  where
    new_ps = updatePos ps new_vs
    new_vs = updateVel vs accels
    accels = accum addAccel zeroAccels x
    x = concat $ y
    y = map (calcAccels ps ms) ns 'using' parListChunk chunksize rdeepseq
A.2. N-BODY PROBLEM ALGORITHMS

\[
calcAccels \, ps \, ms \, i = \text{calc} \, \{i+1\} \, [\]
\]
\[
\text{where}
\]
\[
calc \, i \, j \, \text{acc}
\]
\[
| j === n = \text{acc} \\
| \text{otherwise} = \text{calc} \, \{j+1\} \, \text{new} \, \text{acc}
\]
\[
\text{where}
\]
\[
\text{new} \, \text{acc} = \text{accel}_i : \text{accel}_j : \text{acc}
\]
\[
\text{accel}_i = (i, V (dx * \text{mass}_j * \text{mag}) (dy * \text{mass}_j * \text{mag}) (dz * \text{mass}_j * \text{mag}))
\]
\[
\text{accel}_j = (j, V (dx * \text{mass}_i * \text{mag}) (dy * \text{mass}_i * \text{mag}) (dz * \text{mass}_i * \text{mag}))
\]
\[
\text{mag} = \text{timeStep} / (dSquared * \text{distance})
\]
\[
\text{distance} = \text{sqrt} \, (\text{dSquared})
\]
\[
\text{dSquared} = dx \ast dx + dy \ast dy + dz \ast dz + \epsilon
\]
\[
dx = ix - jx
\]
\[
dy = iy - jy
\]
\[
dz = iz - jz
\]
\[
P = i\_x \, i\_y \, i\_z = ps\!i
\]
\[
P = j\_x \, j\_y \, j\_z = ps\!j
\]
\[
M = \text{mass}_i = ms\!i
\]
\[
M = \text{mass}_j = ms\!j
\]

\[
\text{addAccel} \, (V \, vx1 \, vy1 \, vz1) \, (V \, vx2 \, vy2 \, vz2) = V \, (vx1+vx2) \, (vy1+vy2) \, (vz1+vz2)
\]

\[
\text{updateVel} \, vs \, \text{accels} = \text{toArr} \, (\text{map} \, (\lambda \, i \rightarrow \text{updateV} \, (vs\!i) \, (\text{accels}\!i))) \, \text{ns} \, \text{‘using’}
\]
\[
\text{parListChunk} \, \text{chunks} \, \text{size} \, \text{rdeepseq}
\]
\[
\text{where}
\]
\[
\text{updateV} \, (V \, vx \, vy \, vz) = V \, (vx+x) \, (vy+y) \, (vz+z)
\]

\[
\text{updatePos} \, ps \, vs = \text{toArr} \, (\text{map} \, (\lambda \, i \rightarrow \text{updateP} \, (ps\!i) \, (vs\!i))) \, \text{ns} \, \text{‘using’}
\]
\[
\text{parListChunk} \, \text{chunks} \, \text{size} \, \text{rdeepseq}
\]
\[
\text{where}
\]
\[
\text{updateP} \, (P \, x \, y \, z) = P \, (x + \text{timeStep} \ast vx) \, (y + \text{timeStep} \ast vy) \, (z + \text{timeStep} \ast vz)
\]

\[
\text{-- func to calc time taken between t0 and t1 in sec}
\]
\[
\text{secDiff} : \text{ClockTime} \rightarrow \text{ClockTime} \rightarrow \text{Float}
\]
\[
\text{secDiff} \, (\text{TOD} \, \text{secs}1 \, \text{psecs}1) \, (\text{TOD} \, \text{secs}2 \, \text{psecs}2) = \text{fromInteger} \, \left(\frac{\text{psecs}2 - \text{psecs}1}{1e12} + \text{fromInteger} \, (\text{secs}2 - \text{secs}1)\right)
\]

A.2.3 Barnes-Hut algorithm

\{

\text{-- File: bh.hs}
\}

\text{-- Description: nbody computation: barnes–hut algorithms}

\text{-- Author: Prabhat Totoo}
This implementation of nbody problem uses the barnes-hut method for the simulation. In this version, we first offsets momentum, calculates energy before the N steps simulation, performs simulation and recalculates energy at the end (similar to allpairs.hs).

Reads bodies from text file.

--- Compile | Run
--- Seq: > ghc --make -O2 bh.hs | > bh
--- Par: > ghc --make -O2 -threaded -rtsopts bh.hs | > bh +RTS -N2

--- Runtimes/Output
--- Machine: Intel Xeon CPU E5410 @ 2.33GHz, 8 cores, 8GB RAM (Linux)
--- 16k bodies, 20 iteration
--- #PE Runtimes Speedups
--- Seq 7.53 1
--- 1 7.99 0.94
--- 2 5.38 1.40
--- 3 4.35 1.73
--- 4 3.85 1.96
--- 5 3.64 2.07
--- 6 3.41 2.21
--- 7 3.32 2.27
--- 8 3.16 2.38

import IO
import System
import System.Time
import Text.Printf
import Array
import Data.List
#if defined (PAR)
import Control.Parallel
import Control.Parallel.Strategies
import Control.DeepSeq
import GHC.Exts
import GHC.Conc (numCapabilities)
#endif

data P = P {−# UNPACK #−} !Double {−# UNPACK #−} !Double {−# UNPACK #−} !Double
data V = V {−# UNPACK #−} !Double {−# UNPACK #−} !Double {−# UNPACK #−} !Double
data M = M {−# UNPACK #−} !Double
data PM = PM {−# UNPACK #−} !P {−# UNPACK #−} !M

--- acceleration

data Accel = Acc {−# UNPACK #−} !Double {−# UNPACK #−} !Double {−# UNPACK #−} !
  Double

--- bounding box: a region in 3D space containing points
A.2. N-BODY PROBLEM ALGORITHMS

---

**data Bbox** = Bbox ![# UNPACK #] !P ![# UNPACK #] !P

---

*the BH tree*

---

**node consists of size, centroid X, Y, Z, mass, and children**

**data BHTree** = BHT ![# UNPACK #] !Double ![# UNPACK #] !PM ![BHTree]

```haskell

#if defined (PAR)
instance NFData (Accel) where
  rnf (Acc x y z) = deepseq (x, y, z) ()

instance NFData (PM) where
  rnf (PM p m) = deepseq (p, m) ()

instance NFData (P) where
  rnf (P x y z) = deepseq (x, y, z) ()

instance NFData (V) where
  rnf (V x y z) = deepseq (x, y, z) ()

instance NFData (M) where
  rnf (M m) = deepseq m ()

instance NFData (BHTree) where
  rnf (BHT s pm ts) = deepseq (s, pm, ts) ()

```

**chunksize** = (fromInteger n) `quot` (numCapabilities * 1)

```haskell

n = 16000
ns = [0..(n-1)]
arrbounds = (0,(n-1))
toArr xs = listArray arrbounds xs

timeStep = 0.001

---

*If the distance between the points is smaller than this*

**eps** = 0.01  ---  *then ignore the forces between them.*

---

*If s / d < threshold, then the internal node is sufficiently far away.*

**threshold** = 0.8

```haskell

genPms tag = PM (P (tag' * 1.0) (tag' * 1.2) (tag' * 0.8)) (M (tag' * 30.0))
  where tag' = fromIntegral tag

genVel tag = V (tag' * 0.1) (tag' * 0.2) (tag' * 0.4)
  where tag' = fromIntegral tag
```

```haskell

main =
  do
    let
      pms = toArr $ [ (genPms i) | i <- [0..n-1] ]
      vs = toArr $ [ (genVel i) | i <- [0..n-1] ]
```

```
let res1 = (foldl (\sum i -> sum + g (pms!i) (vs!i)) 0 ns) -- do meaningless
sum...

print (res1)
t1 <- getClockTime -- ...so that, at this point, the positions, velocities
and masses have been read from file

let (pms', vs') = nSteps 20 pms vs

let res2 = (foldl (\sum i -> sum + f (pms'!i) (vs'!i)) 0 ns) -- do a
meaningless sum to generate a small output

print (res2)
t2 <- getClockTime

putStrLn $ printf "time taken: %.2fs" $ secDiff t1 t2

\[ g\left(\text{PM}(x, y, z) \text{ (M m)}\right) \langle V v_x, v_y, v_z\rangle = x+y+z+m+v_x+v_y+v_z \]

\[ f\left(\text{PM}(x, y, z) \text{ (M m)}\right) \langle V v_x, v_y, v_z\rangle = x+y+z+v_x+v_y+v_z -- \text{mass does not change, not} \]

\text{need to check}

nSteps 0 pms vs = (pms, vs)
nSteps n pms vs = nSteps (n - 1) pms' vs'

where

(pms', vs') = oneStep pms vs

oneStep pms vs = (new_pms, new_vs)

where

--new_pms = updatePos pms new_vs
--new_vs = updateVel vs accel
--accel = calcAccels pms
new_pms = toArr $ fst unz
new_vs = toArr $ snd unz
unz = unzip new_pms vs

#if defined (PAR)

new_pms_vs = map (\i -> do this tree (pms!i) (vs!i)) ns 'using' parListChunk
chunksz rdeepseq

#else

new_pms_vs = map (\i -> do this tree (pms!i) (vs!i)) ns

#endif

tree = buildTree box (elems pms)
box = findBounds pms

do this tree pm@(PM (x, y, z) m) \langle V v_x, v_y, v_z\rangle = (new_pm, new_v)

where

Acc x' y' z' = calcAccel tree pm
new_v = V (v_x + x') (v_y + y') (v_z + z')
new_pm = PM (P (x + timeStep * v_x) (y + timeStep * v_y) (z + timeStep * v_z)) m

-- calc the accelerations of all points

#if defined (PAR)
A.2. N-BODY PROBLEM ALGORITHMS

calcAccels pms = toArr (\i -> calcAccelTree (pms!i)) ns "using"
   parListChunk chunksize rdeepseq
#else
calcAccels pms = toArr (\i -> calcAccelTree (pms!i)) ns
#endif

where
tree = buildTree box (elems pms)
box = findBounds pms

-- find the coordinates of the bounding box that contains the given points
findBounds pms = foldl' f (bbox (P 0 0 0) (P 0 0 0)) ns
where
  f (bbox (P minx miny minz) (P maxx maxy maxz)) i =
    let (PM (P x y z) _) = pms!i
        new_min = P (min minx x) (min miny y) (min minz z)
        new_max = P (max maxx x) (max maxy y) (max maxz z)
in bbox new_min new_max

-- build the Barnes–Hut tree
buildTree :: Bbox -> [PM] -> BHTree
buildTree bb pms
| length pms <= 1 = BHT s pm []
| otherwise = BHT s pm subTrees
where pm = calcCentroid pms
      boxesAndPts = splitPoints bb pms
#if defined (PAR)
  subTrees = map (\(bb', pms') -> buildTree bb' pms') boxesAndPts 'using'
     parList rdeepseq
#else
  subTrees = map (\(bb', pms') -> buildTree bb' pms') boxesAndPts
#endif

(Bbox (P minx miny minz) (P maxx maxy maxz)) = bb
s = minimum (abs (maxx - minx), abs (maxy - miny), abs (maxz - minz))

-- if two bodies have positions (x1, y1) and (x2, y2), and masses m1 and m2,
-- then their total mass and center of mass (x, y) are given by:
-- m = m1 + m2
-- x = (x1*m1 + x2*m2) / m
-- y = (y1*m1 + y2*m2) / m
-- calculate the centroid of points

calcCentroid :: [PM] -> PM
calcCentroid pms = PM (P (sum xs / mass) (sum ys / mass) (sum zs / mass)) (M mass)
where
  mass = foldl' (+) 0 [ m | (PM _ (M m)) <- pms ]
  (xs, ys, zs) = unzip3 [ (m * x, m * y, m * z) | (PM (P x y z) (M m)) <- pms ]
— split points according to their locations in the box

\[
\text{splitPoints} :: \text{Bbox} \rightarrow \text{PM} \rightarrow \{ (\text{Bbox}, \text{PM}) \}
\]

\[
\text{splitPoints} \ b b \ [] = \{ (\ b b , []) \}
\]

\[
\text{splitPoints} \ bb \ [\text{pm}] = \{ (\ bb , [\text{pm}]) \}
\]

\[
\text{splitPoints} \ bb \ pms = \text{boxesAndPts}
\]

where

\[
\text{Bbox} \ (P \ \text{minx} \ \text{miny} \ \text{minz}) \ (P \ \text{maxx} \ \text{maxy} \ \text{maxz}) = \ bb
\]

\[
p1 = [\text{pm}|\text{pm}<\text{pms}, \text{inBox} \ b1 \ \text{pm}]
\]

\[
p2 = [\text{pm}|\text{pm}<\text{pms}, \text{inBox} \ b2 \ \text{pm}]
\]

\[
p3 = [\text{pm}|\text{pm}<\text{pms}, \text{inBox} \ b3 \ \text{pm}]
\]

\[
p4 = [\text{pm}|\text{pm}<\text{pms}, \text{inBox} \ b4 \ \text{pm}]
\]

\[
p5 = [\text{pm}|\text{pm}<\text{pms}, \text{inBox} \ b5 \ \text{pm}]
\]

\[
p6 = [\text{pm}|\text{pm}<\text{pms}, \text{inBox} \ b6 \ \text{pm}]
\]

\[
p7 = [\text{pm}|\text{pm}<\text{pms}, \text{inBox} \ b7 \ \text{pm}]
\]

\[
p8 = [\text{pm}|\text{pm}<\text{pms}, \text{inBox} \ b8 \ \text{pm}]
\]

\[
b1 = \text{Bbox} \ (P \ \text{minx} \ \text{miny} \ \text{minz}) \ (P \ \text{midx} \ \text{midy} \ \text{midz})
\]

\[
b2 = \text{Bbox} \ (P \ \text{minx} \ \text{midy} \ \text{minz}) \ (P \ \text{midx} \ \text{maxy} \ \text{midz})
\]

\[
b3 = \text{Bbox} \ (P \ \text{midx} \ \text{miny} \ \text{minz}) \ (P \ \text{maxx} \ \text{maxy} \ \text{midz})
\]

\[
b4 = \text{Bbox} \ (P \ \text{midx} \ \text{midy} \ \text{minz}) \ (P \ \text{maxx} \ \text{maxy} \ \text{maxz})
\]

\[
b5 = \text{Bbox} \ (P \ \text{midx} \ \text{miny} \ \text{midz}) \ (P \ \text{midx} \ \text{midy} \ \text{maxz})
\]

\[
b6 = \text{Bbox} \ (P \ \text{midx} \ \text{midy} \ \mid idz) \ (P \ \text{midx} \ \text{maxy} \ \text{maxz})
\]

\[
b7 = \text{Bbox} \ (P \ \text{midx} \ \text{midy} \ \text{midz}) \ (P \ \text{maxx} \ \text{maxy} \ \text{maxz})
\]

\[
b8 = \text{Bbox} \ (P \ \text{midx} \ \text{midy} \ \text{midz}) \ (P \ \text{maxx} \ \text{maxy} \ \text{maxz})
\]

\[
\text{boxes} = b1:b2:b3:b4:b5:b6:b7:b8:[]
\]

\[
\]

\[
\text{boxesAndPts} = \{ (\ \text{box}, \ \text{pts}) | (\ \text{box}, \ \text{pts}) <- \text{zip} \ \text{boxes} \ \text{splitPts}, \ \text{not} (\ \text{null} \ \text{pts}) \}
\]

\[
(\text{midx}, \ \text{midy}, \ \text{midz}) = ((\text{minx} + \text{maxx}) / 2.0 , (\text{miny} + \text{maxy}) / 2.0 , (\text{minz} + \text{maxz}) / 2.0)
\]

— check if point is in box

\[
\text{inBox} :: \text{Bbox} \rightarrow \text{PM} \rightarrow \text{Bool}
\]

\[
\text{inBox} \ (\text{Bbox} \ (P \ x1 \ y1 \ z1) \ (P \ x2 \ y2 \ z2)) \ (\text{PM} \ (P \ x \ y \ z)) \ = \ (x > x1) \ \&\& \ (x <= x2) \ \&\& \ (y > y1) \ \&\& \ (y <= y2) \ \&\& \ (z > z1) \ \&\& \ (z <= z2)
\]

— calculate the acceleration of a point due to the points in the given tree

\[
\text{calcAccel} :: \text{BHTree} \rightarrow \text{PM} \rightarrow \text{Accel}
\]

\[
\text{calcAccel} \ (\text{BHT} \ s \ \text{pm} \ \text{subtrees}) \ \text{pm’}
\]

\[
| \ \text{null} \ \text{subtrees} = \text{accel} \ \text{pm} \ \text{pm’}
\]

\[
| \ \text{isFar} \ s \ \text{pm} \ \text{pm’} = \text{accel} \ \text{pm} \ \text{pm’}
\]

\[
| \ \text{otherwise} = \text{fold1’} \ \text{addAcc} \ (\text{Acc} \ 0 \ 0 \ 0) [ \ \text{calcAccel} \ \text{st} \ \text{pm’} \ \mid \ \text{st} <- \text{subtrees} ]
\]

where

\[
\text{addAcc} \ (\text{Acc} \ a1 \ b1 \ c1) \ (\text{Acc} \ a2 \ b2 \ c2) = \text{Acc} \ (a1+a2) \ (b1+b2) \ (c1+c2)
\]
A.2. N-BODY PROBLEM ALGORITHMS

— calculate the acceleration on a point due to some other point

\[
\text{accel} :: \text{PM} \to \text{PM} \to \text{Accel}
\]

\[
\text{accel} (\text{PM} (P \ x \ y \ z) (M \ m)) (\text{PM} (P' \ x' \ y' \ z') (M \ m')) = \text{Acc} \ (dx \ast m \ast \text{mag}) \ (dy \ast m \ast \text{mag}) \ (dz \ast m \ast \text{mag})
\]

where

\[
\begin{align*}
\text{dsqr} & = (dx \ast dx) + (dy \ast dy) + (dz \ast dz) + \text{eps} \\
\text{d} & = \sqrt{\text{dsqr}} \\
\text{dx} & = x' - x \\
\text{dy} & = y' - y \\
\text{dz} & = z' - z \\
\text{mag} & = \text{timeStep} / (\text{dsqr} \ast \text{d})
\end{align*}
\]

— use centroid as approximation if the point is far from a cell

\[
is\text{Far} :: \text{Double} \to \text{PM} \to \text{PM} \to \text{Bool}
\]

\[
is\text{Far} \ s (\text{PM} (P \ x \ y \ z)) (\text{PM} (P' \ x' \ y' \ z')) = \text{let}
\]

\[
\begin{align*}
\text{dx} & = x - x' \\
\text{dy} & = y - y' \\
\text{dz} & = z - z' \\
\text{dist} & = \sqrt{dx \ast dx + dy \ast dy + dz \ast dz} \\
\text{in} \ (s / \text{dist}) < \text{threshold}
\end{align*}
\]

— update velocity

\[
\text{#if defined (PAR)}
\]

\[
\text{updateV} \ \text{vs} \ \text{acs} = \text{toArr} \ (\text{map} \ (\ \text{i} \to \text{updateV} \ (\text{vs}!\text{i}) \ (\text{acs}!\text{i})) \ \text{ns} \ \text{‘using’}
\]

\[
\text{parListChunk} \ \text{chunksize} \ \text{rdeepseq}
\]

\[
\text{#else}
\]

\[
\text{updateV} \ \text{vs} \ \text{acs} = \text{toArr} \ (\text{map} \ (\ \text{i} \to \text{updateV} \ (\text{vs}!\text{i}) \ (\text{acs}!\text{i})) \ \text{ns})
\]

\[
\text{#endif}
\]

where

\[
\text{updateV} \ (V \ vx \ vy \ vz) \ (\text{Acc} \ x \ y \ z) = V \ (vx+x) \ (vy+y) \ (vz+z)
\]

\[
\text{#if defined (PAR)}
\]

\[
\text{updatePos} \ \text{pms} \ \text{vs} = \text{toArr} \ (\text{map} \ (\ \text{i} \to \text{updatePm} \ (\text{pms}!\text{i}) \ (\text{vs}!\text{i})) \ \text{ns} \ \text{‘using’}
\]

\[
\text{parListChunk} \ \text{chunksize} \ \text{rdeepseq}
\]

\[
\text{#else}
\]

\[
\text{updatePos} \ \text{pms} \ \text{vs} = \text{toArr} \ (\text{map} \ (\ \text{i} \to \text{updatePm} \ (\text{pms}!\text{i}) \ (\text{vs}!\text{i})) \ \text{ns})
\]

\[
\text{#endif}
\]

where

\[
\text{updatePm} \ (\text{PM} (P \ x \ y \ z) \ m) \ (V \ vx \ vy \ vz) = \text{PM} \ (P \ (x + \text{timeStep} \ast vx) \ (y + \text{timeStep} \ast vy) \ (z + \text{timeStep} \ast vz)) \ m
\]

— func to calc time taken between t0 and t1 in sec

\[
\text{secDiff} :: \text{ClockTime} \to \text{ClockTime} \to \text{Float}
\]

\[
\text{secDiff} \ (\text{TOD secs1 psecs1}) \ (\text{TOD secs2 psecs2}) = \text{fromInteger} \ (\text{psecs2} - \text{psecs1}) / \ (\text{1e12} + \text{fromInteger} \ (\text{secs2} - \text{secs1}))
\]