Hierarchical N-Body Simulations in Haskell

Pablo E. Martínez López [*]

LIFIA, Facultad de Ciencias Exactas, UNLP
CC.11 Correo Central, (1900) La Plata, Argentina
Tel/Fax: + 54 21 228252
E-mail: `fidel@info.unlp.edu.ar`
URL: `http://www-lifia.unlp.info.edu.ar/`

Abstract

Functional Programming was historically considered a 'toy' for researchers, but recent developments in the field show that its area of application is wider.

The Hierarchical N-Body method is an iterative method used in astrophysics to simulate the gravitational evolution of collisionless matter, in order to understand the formation of galaxies.

In this work, functional programming is used as a tool for the description and prototipation of the Hierarchical N-Body method, in an attempt to show that it is a suited tool for expressing problems mantaining a good understanding of them and allowing a great degree of abstraction and generalization, and with a reasonable efficiency for a prototype.

[*]This work was partially funded by PEDECIBA, Uruguay.

Hierarchical N-Body Simulations in Haskell

1 Introduction

The simulation of the gravitational evolution of collisionless matter, thogether with the properties of gas in larger scales, is used in the study of theoretical cosmology, mainly for understanding the formation and clustering of galaxies. Analytic treatment of hydrodynamical effects is usually restricted to systems with high degree of symmetry or other simplifying assumptions. A model of the matter based on the selection of a finite number of points, called particles, is used to remove some of these restrictions. Classical methods to solve the problem using the particle model are direct integration – which involves a number of operations increasing with the square of the number of particles, called N–, and iterative methods based on grids – which impose geometrical assumptions and restrictions to the particles. A new approach developed during the late eigthies is based on a tree-structured subdivision of the space [BH86], and has many advantages, including accuracy, freedom from assumptions and applicability to a wide class of systems. This method is called Hierarchical N-Body, and its prototypical implementation in a functional setting is the subject of this work.

Functional Programming is a discipline of programming based on mathematical background [BW88], and it has many theoretical advantages over its classical relative, imperative programming. Historically it was considered a toy for researchers, and it was further confined to computer science applications: compilers, proof asistants, etc., mainly because of the lack of efficient compilers and tools, and because of a poor understanding of its capabilities by the scientific community. In the last ten years several new developments in the field [JM95, JGF96, Röj95] suggest that functional programming may be used to real applications. Studies of this fact were done [Wad95], and the main conclusion is that functional programming is, at least, a very useful tool for prototipation, with efficiency comparable with classical paradigms.

In this work functional programming is used to solve a real application of great significance in astrophysics – the Hierarchical N-Body problem – and it is showed that a functional setting is best suited for the understanding and generalization of the solution.

The rest of the article is divided as follows. In Sect. 2 the Hierarchical N-Body method [BH86] is presented. In Sect. 3 an implementation of the method using the C language [KR78] is discussed. In Sect. 4 the functional implementation of the method used the functional language Haskell [PH$^+$96] is covered in detail. In Sect. 5, a comparison of the running time of both implementations is done. And finally, Sect. 6 presents the conclusions of the work.

2 Hierarchical N-Body

The hierarchical N-body method is used in the simulation of the gravitational evolution of a system composed by N particles floating in a vacuum. Each particle has its own mass, initial position and initial velocity. As time goes by, the position and velocity of each particle are modified because of the graviational effect of all the other particles, according with the gravitational laws.

In order to simulate the evolution in time, a method known as *leap-frog integration* is used. This method consider a small slice of time, `dt`, and calculates the new position and velocities by interpolation using the known values. This step is repeated until the final desired time is reached.

An exact calculation of the interpolation in any step can be done by direct integration, considering all the $N(N+1)/2$ interactions, but when the number of particles is large, the computational complexity grows rapidly.

In real situations, the contribution made by distant particles over a fixed one is much less than that of nearby particles. Moreover, the contribution of a group of distant particles depends more on the total mass and mean velocity of the group than in the particular interactions inside the group. This observation leads to the idea of grouping the particles according with their positions, in a hierarchical structure (i.e. a tree), and use this grouping to save computational effort: if a group is "distant enough" consider it as one (pseudo)particle, but if it is not, divide it in its subgroups, and try again. The resulting method needs to calculate only $NlogN$ interactions in the average. Thus in a given amount of time, the number of particles involved in the simulations can be much larger.

There are two problems to consider. The first one is how to construct the grouping. The second one is the decision criteria, "distant enough". In this work it was taken the approach of [BH86] for the solution of these problems.

The construction of the grouping in [BH86] begins with a volume containing the N particles, and recursively subdivides it in eight equal subvolumes, until a subvolume contains at most one particle. The result is a tree with eight subtrees, each being an empty cell, a one-particle cell or a proper subtree. Each node of this *oct-tree* contains a (pseudo)particle with the total mass and center-of-mass of all the particles contained in the volume. The calculation of gravitational force for a given particle p proceeds as a top-down walk of the tree, applying the decision criteria in order to calculate either using the pseudoparticle or adding the calculation of the eigth subtrees. There are two ways for the construction of this tree: top-down and bottom-up. The former begins with an empty tree and adds one particle at a time, growing the tree as needed; it has the advantage of being iterative over the particles, and the disadvantage that pseudoparticles has to be calculated after the tree construction. The latter divides the set of particles in eight subsets and proceeds recursively, constructing at last a node with its eigth subtrees and the pseudoparticle; it is simpler, but recursion can produce a great overhead in some languages.

For the definition of "distant enough", [BH86] considers the size of the volume containing the group, s, the distance between the group's center of mass and the particle position, d, and an accuracy parameter, θ; if $s/d \leq \theta$, then include the interaction with the group as a whole, and if not, subdivide the group. A more involved method includes the displacement of the center of mass from the center of the volume as a correction parameter. The accuracy parameter controls the error and the complexity of the method: a value $\theta = 0$ means always subdivide, and implies that an exact calculation is done; larger values of θ diminishes the complexity but makes the error larger.

A rigorous error analysis is possible because the tree structure is unique for a given set of particles and a given volume. Each cell that is not subdivided introduces a small error due to quadrupole and higher-order moments (the interactions inside the group). A 'worst-case' analysis can be performed, and [BH86] reports that a value of $\theta = 1$ are accurate to $\sim 1\%$, with little dependence on N. As the error from one step to the next is weakly correlated, there is a reasonable probability that the total error after several steps keeps small.

3 The C Implementation

The C implementation of the Hierarchical N-Body method can be obtained from WWW, at the URL `ftp://hubble.ifa.hawaii.edu/pub/barnes/treecode`. This code was developed by Josh Barnes and Piet Hut, in ANSI C, and it is the fourth release of it.

The main characteristics of the code are the construction of the oct-tree, and the calculation of gravitational forces. In order to understand the code, the data structures are described first, trying to recall the programming decisions of the authors.

Each particle is represented as a `struct`, called `body`, having its position, mass, velocity, acceleration and potential. A substructure, called `node`, containing the position and mass, plus structural information, is defined to improve sharing of memory cells with the tree described below. The N particles are stored in an array of bodies. The tree structure is defined as a `struct`, called `cell`, with a `node` for the pseudoparticle information, the critical radius (used for the divsion criteria), an array of subtrees, and information for the quadrupolar moment of the cell. Once constructed, the tree is threaded for a non-recursive walk, and thus the array of subtrees are no longer needed, allowing the sharing of memory with the quadrupolar moment information. The non-recursive walk has the shape of a linked list with two succesors: one to the leftmost child, meaning subdivision of the current cell, and one to the right sibling, meaning calculation with the current cell. The construction of the tree is top-down, and reuse of already allocated pointers is done via a linked list of garbage cells.

The volumes are not represented explicitely. They are represented with two parameters: the position of the center, and the side length of the cube under consideration. There is a global parameter `rsize` that keeps the size of the largest volume used so far, and the position is recorded in the `node` of the cell – this `node` is later updated with the pseudoparticle data. Initially a small volume centered in the origin of coordinates is considered, and every time any particle fall outside of it, its size is doubled until all the particles lie inside again; the volumes never decrease.

There are also some global parameters that are used to customize the algorithm. One of them is the accuracy parameter, θ, but there are some others: a parameter to softening the potential, called ϵ, some parameters to record time, `tnow`, `tstop`, and several others to control input-output and to record statistics.

The set of particles are read from and write to files, and statistics are reported on the standard output. Some option allows to generate the particles at random, instead of reading it from the disk.

One interesting aspect of the code is that the number of dimension can be set to 2 or 3 in compile time by a `#define`, and the precision of numbers can be set to `float` or `double` by the same mechanism, being by this way "general" in both aspects.

The number of code-lines is about one thousand and nine hundred lines, being roughly eight hundred for the main application (method and data structures), six hundred for input-output, and five hundred for base code.

4 The Haskell Implementation

Haskell [PH$^+$96] is a lazy functional language that becomes the standard in the field. It was designed in the late eighties and early nineties by a comitee of the most renowned members of the functional community. Haskell more important features are lazy evaluation, modules, higher order functions, functions as data types and a static type system with parametric polymorphism, algebraic types and a class system for overloading.

Lazy evaluation is a mechanism of evaluation that follows the rule "compute only when something is needed, and then, only once". Lazy evaluation allows modularization, because functions can be programmed independently of each other without thinking in the order of evaluation, but efficiency is still achieved.

Modules provide the definition of abstract data types, with the ability of importing the functions needed from other modules, and of exporting the functions visible from outside the module. They also provides the posibility of separate compilation.

Higher order functions are functions that have other functions as arguments or that return functions as their result. Partial evaluation and abstract version of control structures defined by the user are some of the advantages provided.

In functional languages, functions can be stored in data types, and even used as them. This characteristic allows to store functions to perform some computation later.

Haskell is a strongly typed language. Its type system is static – which means that the type of every expression is determined in compile-time – and provides type inference – which means that the types can be determined even when the programmer doesn't provide type information. Some characteristics of the type system are parametric polymorphism, algebraic data types and classes. Parametric polymorphism is the ability of a function to work without knoledge about the type of some parameter. This allows, for example, to have code for lists that don't care about what things are stored in the lists. Algebraic types allows the construction of user defined types, even recursive, without using pointers or explicit representation at all. The class system provides overloading of functions. A class is a collection of types that share the name of some functions; an instance is one of these types. This mechanism allows the construction of a common interface for several types, and by this way, do not rely on a particular implementation. The instantiation mechanism decides, in compile time, wich particular instance to use.

The Haskell language comes with a rich predefined system of classes. In particular, there are several classes for numbers, providing classical operations and standard convertions with the same name for all number types. An interesting feature of Haskell is the way it understands the numerical constants: a numerical constant is overloaded, and its exact type is determined by context inference.

The Haskell code for the hierarchical N-body method has several modules, and it can be obtained from the author by e-mail. Each module contains the definition of a type or class of types for some abstract data structure. There are modules defining Vectors, Volumes, Particles, Bodies, Trees and Systems. Each module will be described independently.

The number of code-lines is approximately one thousand. This number is not meaningful, because many lines were dedicated to type signatures and comments that enhance code readability.

4.1 Vectors

Module `Vector` defines the class of vectors, providing functions to create, access and operate with them. In particular, elementwise and scalar operations were defined as higher order functions, and further instantiated for addition, substraction and multiplication. Also dot vector product, distance between vectors and module are declared here.

Two instance types were defined, for two and three dimensional vectors. The particular implementations are for test purposes, and more efficient instances can be defined without alter the existing code in any way.

Vectors are a parametrized type, allowing the definition of vectors with different types of numbers, and providing by this way independence of the precision.

4.2 Volumes

In contrast to the C code, volumes are defined explicitely in the Haskell version. The module `Volume` contains the definition of the class for volumes, providing functions to create, access and manipulate them. Important functions are `quadrant` that returns one subvolume needed for the subdivision and `changeToFit` that, given a volume and a list of vectors, returns a new volume that contains all the given vectors.

The type of volumes is parametric in both, the type of numbers and the type of vectors used.

4.3 Particles

The module `Particle` contains the definition of the class of particles. A particle is any element that has a position in the space and a mass. Functions to create and access particles are provided. Particles play the role of the `struct node` of the C version, but in an abstract way.

Particles are parametric in the both the type of numbers and the type of vectors used.

This module also provides the code to calculate the center of mass of a given list of particles.

4.4 Bodies

Bodies are particles that have velocity, accelaration and potential. The class of bodies is a subclass of the class of particles, thus extending it with functions to update and access the added data. There are also functions to update the position and velocity according to the physical laws.

Bodies are parametric in the both the type of numbers and the type of vectors used.

4.5 Trees

Trees are defined in two modules: `BodyTree` and `Force`. The module `BodyTree` contains the definition of the type of trees used for the hierarchical calculation of forces, and the module `Force` contains the function that actually calculates them.

Body-trees are defined as:

- an empty tree,
- a single body, or
- a cell containing a particle, a list of body-trees and a function.

Empty trees are used only for intermediate calculation, and are not stored explicitely in the final tree. In a cell, the particle stores the data of the pseudoparticle, the list of body-trees containt the different groups of bodies, and the function is used in the calculation phase to determine if subdivision is needed. A function for constructing a tree from a given list of particles is given; this function performs a bottom-up construction, since functional programming is best suited for recursive functions.

The module `Force` provides a function `hackgrav` that given a body and a body-tree, calculates the gravitational influence of all the particles in the body tree over a single body using the hierarchical method. Insted of using the non-recursive walk (that would be impossible because there is no pointers in the language), a stack of pendant groups is mantained, and subdividing a group means to push the subgroups.

Functions and types in this module are parametric in the type of numbers, the type of particles and the type of volumes.

In this version of the functional code the trees were provided as a single type, but a more abstract and general view can be implemented using the class system. A class of trees needs to provide two main functions: one that builds up the tree, and one that performes the force calculation. The previously described type can be an instance of such a class, and other instances may be provided – for example with the top-down method of construction.

4.6 Systems

In functional programming there are no global variables. For that reason, a new abstraction, system, is defined. A system contains the list of particles and all the "global" information required to input, output and evolve the system.

There are three modules for systems: `Params`, `BodySys` and `Interface`. The module `Params` contains the definition and default values for the different parameters used – the accuracy, the softening of potential, the method used for subdivision calculation, etc. The module `BodySys` provides functions to create and manipulate systems, including the evolution in time and the input-output to disk. The module `Interface` provides functions to convert data read from and write to the files in the format used by the C code.

Systems are parametric in the type of volumes, bodies and numbers.

In this version, systems were provided as a single type, but great level of abstraction and generalization can be achieved by the definition of a class for systems. The type described below can be an instance of such class, and also other instances may be defined – for example systems that performs statistics.

5 Comparing the Implementations

The results reported in this section are preliminar, as only a few runs of the codes were performed, and the Haskell code was not optimized in any way.

In order to compare the implementations, they were compiled and runned in a Sun workstation, running the SunOS 5.3 operating system. The C version was compiled using the gcc compiler native to the operating system, and the Haskell version was compiled using the Glasgow Haskell Compiler, version 2.01, which performs optimizations in compilation.

Three test cases were used: one with 36 particles, one with 1024 particles and one with 4096 particles. The first one is a toy example, but the other two are simulation of galaxies. The Haskell code is aproximately 200 times slower than the C version. Some analysis performed showed that much of the time the program is performing I/O. More tests are needed to determine the exact source of slowness.

6 Conclusions

This article presents the implementation of the hierarchical N-body method using the lazy functional language Haskell. The resulting code was compared with a C implementation of the same algorithm.

The functional code is much more easier to read than the C version. With respect to the efficiency, without any enhancement, the functional version is aproximately 200 times worst than its imperative counterpart. Having into account that further enhancements are possible, the result is not so bad.

A great degree of flexibility is achieved by means of abstraction and generalization. Several implementation decisions can be studied, and different methods that follows the same pattern can be implemented, only adding new instances for the classes that conforms the system.

Acknowledges

I want to thank my director, Roberto Di Cosmo, for joining me with this subject, for the carefull reading of the draft of this paper, and for his advises. I also want to thank my counsellor, Juan Echagüe, for his guidance.

References

[BH86] Josh Branes and Piet Hut. A hierarchical O(N log N) force-calculation algorithm. *Nature*, 324(4):446–449, December 1986.

[BW88] Richard S. Bird and Philip Wadler. *Introduction to Functional Programming*. Prentice Hall, 1988.

[JGF96] Simon L. Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *ACM Symposium on the Principles of Programming Languages (PoPL'96)*, St.Petersburg Beach, Florida, January 1996.

[JM95] Johan Jeuring and Erik Meijer, editors. *Advanced Functional Programming, LNCS 925*. Springer-Verlag, May 1995.

[KR78] Brain W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1978.

[PH$^+$96] John Peterson, Kevin Hammond, et al. Report on the programming language Haskell, a non–strict, purely functional language. Version 1.3. Technical report, Yale University, May 1996.

[Röj95] Niklas Röjemo. Efficient parsing combinators. In *Garbage Collection, and Memory Efficiency, in Lazy Functional Languages*, Göteborg, Sweden, May 1995. Chalmers University of Technology, Department of Computer Science. Part of the Ph.D. thesis.

[Wad95] Philip Wadler, editor. *Journal of Functional Programming. Special Issue on State-of-the-art Applications of Pure Functional Programming Languages*, volume 5 (3). Cambridge University Press, July 1995.