MPJ Express Meets Gadget: Towards a Java Code for Cosmological Simulations

Mark Baker[1], Bryan Carpenter[2] and Aamir Shafi[3]

[1] ACET, University of Reading
[2] OMII, University of Southampton
[3] DSG, University of Portsmouth

Abstract. Gadget-2 is a massively parallel structure formation code for cosmological simulations. In this paper, we present a Java version of Gadget-2. We evaluated the performance of the Java version by running a colliding galaxy simulation and found that it can achieve around 70% of C Gadget-2's performance.

1 Introduction

Various computer scientists have argued that Java could make an excellent language for developing scientific codes. To date this argument has not convinced too many practising computational scientists. The scarcity of high-profile number-crunching codes implemented in Java does not help the case.

We have recently released MPJ Express [1], a thread-safe, production quality Java messaging system for high performance computing. To help establish the practicality of real scientific computing using message passing Java we have ported the parallel cosmological simulation code, Gadget-2, from C to Java, using MPJ Express. Gadget-2 [7] is a massively parallel structure formation code developed by Volker Springel at the Max Planck Institute of Astrophysics. Versions of Gadget-2 have been used in various research papers in astrophysics literature, including the noteworthy "Millennium Simulation" [8]—the largest ever model of the Universe.

Producing a Java version of Gadget is an experiment that helps us to understand where Java stands in comparison to C—an already established HPC language. Concerns about Java's performance have stopped many computational scientists from seriously considering it. But constant improvements in JIT (Just In Time) compilers, which translate bytecode into the native machine code at runtime, have improved the compuational performance.

Exploitation of Java for simulation projects has been ongoing for some years. JWarp [3] is a Java library for discrete-event parallel simulations, which builds its own communication infrastructure based on Remote Method Invocation (RMI). MONARC [4] is a simulation framework for large scale computing resources. It has been deployed on an inter-continental testbed to verify simulation results with success. CartaBlanca [5], from Los Alamos National Lab, is a general purpose non-linear solver environment for physics computations on non-linear grids. It employs an object-oriented component design, and is pure Java. These projects suggest that Java has already made its mark on a range of projects involved in parallel simulations, or scientific computing in general.

Section 2 of this paper presents an overview of Gadget-2. We discuss our experiences in porting Gadget-2 to Java in Section 3. We evaluate the performance of the Java version in section 4 and also compare it with the original C version. We conclude and discuss future work in Section 5.

2 Overview of Gadget-2

Gadget-2 is a free production code for cosmological N-body and hydrodynamic simulations. The code is written in the C language and parallelized using MPI. It simulates the evolution of very large, cosmological-scale systems under the influence of gravitational and hydrodynamic forces. The universe is modelled by a sufficiently large number of test particles, which may represent ordinary matter or dark matter.

We are particularly interested in the parallelization strategy, which is based on an irregular and dynamically adjusted domain decomposition, with copious communication between processors.

To give some feeling for the scale of interesting problems, consider the so-called "Millenium Simulation" [8]. This simulation follows the evolution of 10^{10} dark matter particles from the early Universe to the current day. It was performed on 512 processors and used 1 Terabytes of distributed memory. The simulation used 350,000 CPU hours over 28 days of elapsed time.

2.1 Computing Gravitational Forces

One of the main tasks of a structure formation code is to calculate gravitational forces exerted on a particle.

In a N-body cosmological simulation, every particle exerts gravitational force on every other particle. The reason is that gravity is a long range force. Thus, calculating gravitational force in such simulations can be computationally intensive—the total cost is $O(N^2)$ for the naive summation approach. This is not feasible for the scale of problems that Gadget-2 aims to solve.

Thus, Gadget-2 can use either of two efficient algorithms to calculate gravitational forces. The first is Barnes-Hut (BH) [2] oct tree, and the second is a hybrid of BH tree and Particle-Mesh (PM) method called *TreePM*. In this paper, we restrict our attention to the pure BH tree algorithm.

Barnes-Hut Tree Algorithm The cubical region of 3D space is divided into eight sub-regions by halving each dimension. Every sub-region that contains any particles is recursively divided until each region has at most one particle. The root of the Barnes-Hut tree corresponds directly to the whole 3D space. The first division of space results in eight sub-regions that become the daughter nodes of the root. This process continues until each node of the tree contains one particle.

The reason for arranging the particles in a tree data-structure is that it allows efficient calculation of gravitational forces. The tree is traversed from root to compute the force, for example on a particle i. If a node n is *distant from* particle i, the contribution of node n is added to force on i from the center of mass of n. In this case, there is no need to to visit the daughter nodes of n. The daughter nodes of node n are visited recursively if it is *close to* i.

The definition of *distant from* or *close to* depends on an opening criterion. The basic idea is that a node representing some region in space is *distant from* a particle i if the angle it subtends is smaller than a threshold opening angle. Otherwise, a node is considered *close to* particle i.

Using this approach, it is possible to calculate the gravitational force for each particle in $O(\log N)$ steps. For the range of N of practical interest this is clearly a huge win over the summation approach that results in $O(N)$ steps.

2.2 Domain Decomposition

Being a massively parallel code, Gadget-2 needs to divide space or particle set into domains, where each domain is handled by a single processor. It is particularly challenging in Gadget-2 because it is not practical to divide space evenly. This would result in poor load balancing because some regions have more particles than the others. Conversely, it is also not possible to divide particle evenly in a fixed way because they move throughout space and it is desirable to keep physically close particles on the same processor.

To solve this, Gadget-2 uses a space-filling *Peano-Hilbert curve* originally suggested by Warren and Salmon [6]. Gadget-2 applies the standard recursion for constructing the curve 20 times, logically dividing space into up to $2^{20} \times 2^{20} \times 2^{20}$ cells on the Peano-Hilbert curve. Each cell is labelled by its location along the Peano-Hilbert curve—2^{60} possible locations. The information about the location of each cell can be stored in a `long` word called the *Peano-Hilbert key*. These Peano-Hilbert keys play an important role during domain decomposition. Because the total number of cells is far greater than total number of particles, points of the discrete linear Peano-Hilbert curve are sparsely populated with particles. To establish the domain decomposition, one sorts particles by their Peano-Hilbert keys and then divides them evenly into P sections, where P is the total number of processors.

This technique implements an efficient domain decomposition. It provides good load balancing. The domains are simply connected and quite "compact" in real space, because particles that are close along Peano-Hilbert curve are close in real space (the converse is often but not always true). An added advantage is that the Peano-Hilbert curves provide simple mapping to Barnes-Hut tree nodes.

Distributed Representation of Tree The BH tree is implemented as a distributed data structure. Nodes of the tree can be classified according to whether all particles in the node belong to one processor, or the node contains particles from multiple processors. Nodes in the first category are stored locally on the relevant processors. All nodes in the second category—this typically means higher nodes in the tree—are replicated over all processors.

So every processor holds a copy of the root nodes and all daughter nodes down to the point where all particles of a node are held on a single processor. Where this is a remote processor the corresponding node is called a *pseudo-particle*. To compute the force on a single local target particle, the tree is traversed starting from root as usual accumulating force contributions from locally held particles.

2.3 Communication

The original Gadget-2 is parallelized using the standard MPI specifications. As part of the parallel tree-force computation, a processor walks the tree for every locally held particle accumulating force contributions. These contributions may come from local particles or *pseudo-particles*. If the daughters of a node representing *pseudo-particles* need to be traversed, the locally held particles are marked for export to the processor that owns the *pseudo-particle* in question. After the tree-walk, all particles marked for export are communicated to remote hosts. These hosts calculate the force contributions and communicate them back. Also, there is some communication involved during domain decomposition for distributed sorting of the particle list.

3 Porting Gadget-2 to Java

Gadget-2 was manually translated to the Java language. We deliberately kept similar data structures in the translated version so that we could cross reference the original source code for debugging. Currently there are some functional limitations compared with the C code. For example, the Java version only provides the option of using BH oct tree for calculating gravitational forces.

There are three dependencies for Gadget-2; GNU Scientific Library (GSL), parallel version of Fastest Fourier Transforms in the West (FFTW), and of course a MPI library. Gadget-2 only uses a handful of GSL functions—we manually translated these to Java. FFTW would be required for the *TreePM* algorithm, and for this reason we use BH tree algorithm for calculating gravitational forces in the current Java version. For communication, we use MPJ Express, our own thread-safe implementation of MPI-like bindings for the Java language.

The source distribution of the original Gadget-2 contains initial conditions for some small simulations including *Colliding Galaxies* and *Cluster Formation*. The input data is read from the initial conditions file into a `ParticleData` array.

The main simulation loop increments timesteps and drift the particles to the next timestep. This involves calculating gravitational forces for each particle in the simulation and updating their accelerations. The BH tree could either be dynamically updated or redrawn to depict the new state of the system. Calculating the gravitational forces, or in other words, walking the tree is the most compute intensive task in the simulation.

3.1 Test Cases for Java version

The source distribution of the original Gadget-2 code comes with some initial conditions files including *Colliding Galaxies* and *Cluster Formation*. The Gadget-2 code produces snapshot files at regular intervals during the simulation which can be used to plot the state of the system. The distribution also provides some IDL (software for data visualisation and analysis) scripts to view the system. We used these scripts along with the snapshot files to generate visual output, which are essentially indistinguishable for the two versions. This provides us with a very high degree of confidence in correctness of the translated code.

3.2 Initial Java optimizations

The performance evaluation of the initial Java version revealed that the performance was approximately three times slower than the C version on 1, 2, 4, and 8 processors. We now describe the principal optimizations applied to improve performance.

Custom Serialization and Deserialization Initial versions of Java Gadget-2 communicated Java objects, which was made possible by exploiting the JDK default serialization and de-serialization mechanism in MPJ Express. The object serialization and de-serialization is the process of converting Java objects to a byte array and vice versa. It can have detrimental effects on the performance of a parallel application. Thus, we decided to replace Java object communication in Java Gadget-2 with primitive datatypes.

In the original C Gadget-2, initial conditions are read into an array of C `structs` called `ParticleData`. In the Java version, this array of `structs` is replaced by an object array called `ParticleData`. Particles that need to be exported are copied to a contiguous memory region called `CommBuffer` in the original C version. We replaced this with `CommBuffer` object, which contained object arrays. Before the communication operation, the data was copied from `ParticleData` array onto a related object array in `CommBuffer` object and communicated.

In the optimized version of Java Gadget-2, this `CommBuffer` object is replaced by a contiguous memory region, which is an instance of `ByteBuffer` class. Before the actual communication, we copy primitive data from each element of `ParticleData` array to `CommBuffer`. Once all the data has been packed onto this `ByteBuffer`, it is communicated to the receiver process. The receiver process receives the data in `CommBuffer`, and unpacks it onto the `ParticleData` object array. This technique helped us not only to avoid the Java object serialization overhead, but also reduced the memory footprint of the JVM (Java Virtual Machine) by 60%.

Maintaining Memory Locality It is hard to maintain memory locality for Java HPC applications. The reason is that native machine architecture is not aware of Java objects that might be involved in computationally intensive sections of the code. This might result in poor usage of processor cache and page faults. The authors in [9] have identified this problem and proposed an object-aware memory architecture.

In the Java version of Gadget-2, we maintained memory locality by *flattening* sensitive data structures. Using this technique, we replaced Java object arrays with primitive datatype arrays. For example, BH tree nodes are stored in an array of Java objects called `Nodes_base`. Each element of this array has members like an array of `double`s called `center` and a `double` called `len`, that represents the side length of a tree node. In the Java version, these two members `center` and `len` are stored in a `double`s array. This ensures that when a particular tree node is accessed, all the members of particular object element in `Nodes_base` array are in close vicinity in the memory.

We also *flattened* the `ParticleData` array, where each object has attributes like a three element array of `pos` and `vel` representing position and velocities in three dimensions. In addition, we also *flattened* the `TopNodes` array.

4 Performance Evaluation

In this section, we evaluate the performance of the Java version against the C Gadget-2 code. We used the *Colliding Galaxies* simulation for comparison. Note that the C version of Gadget-2 is meant to be a massively parallel code. The *Colliding Galaxies* simulation is too small to utilize its full potential. Nevertheless, it gives us a starting point for evaluating the performance of Java Gadget-2.

We conducted these tests on a cluster called *StarBug* at the DSG. This cluster consists of 8 dual Intel Xeon 2.8 GHz processors. The PCs were equipped with 2 Gigabytes of ECC RAM with 533 MHz Front Side Bus (FSB). The PCs were running the Debian GNU/Linux with the 2.4.32 Linux kernel. The C compiler on this cluster was GNU GCC 3.3.5. There is an option to use Myrinet or Fast Ethernet for communication.

We used MPJ Express (version 0.23) with Sun JDK 1.5 (Update 6) to run the Java version of Gadget-2. The original C Gadget-2 code used MPICH (version 1.2.5.2) on Fast Ethernet and MPICH-MX (version 1.2.6.0.94) using Myrinet.

Figure 1 shows execution time of C and Java Gadget-2 on 1, 2, 4, and 8 processors using Fast Ethernet. A similar comparison of execution time on Myrinet is shown in Figure 2. The Java version is almost 30% slower than the C version.

Figure 3 shows tree-walk time of C and Java Gadget-2. The presented tree-walk is the average of all processors for more than one processor case. The Java version is approximately 30% slower in calculating gravitational force than the C version. This may be acceptable performance given that Java has many extra safety features including mandatory array bounds checking.

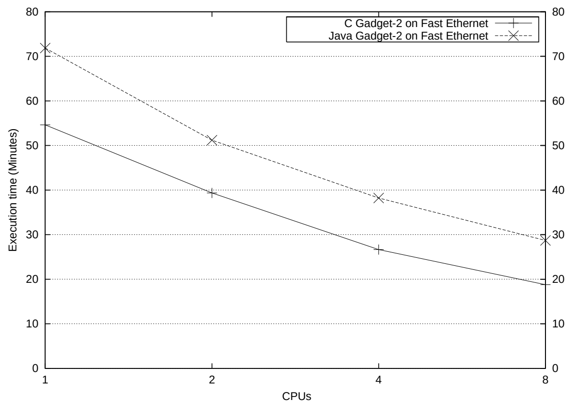

Fig. 1. Execution Time Comparison on Fast Ethernet

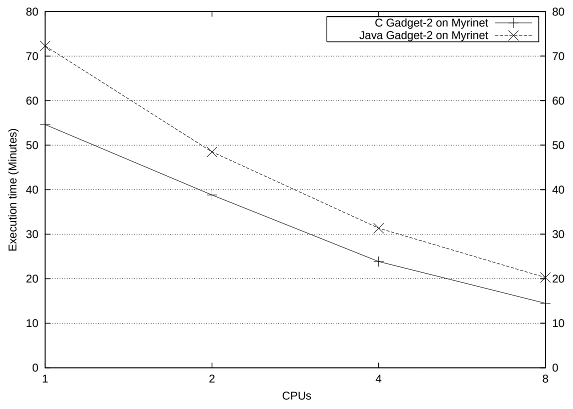

Fig. 2. Execution Time Comparison on Myrinet

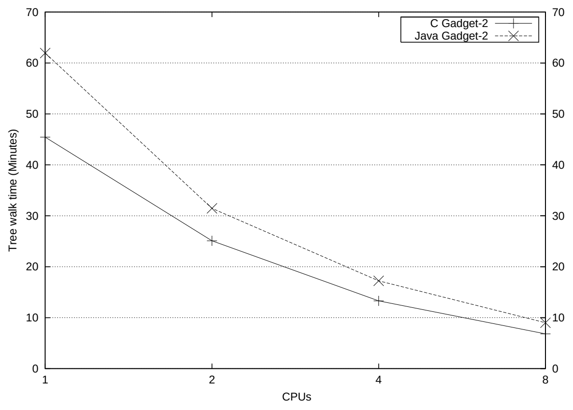

Fig. 3. Tree Walk Time Comparison

5 Conclusions and Future Work

In this paper, we have presented a Java version of Gadget-2. The performance evaluation of the Java version revealed that it can achieve around 70% of C Gadget-2's performance. It should be noted that we are comparing a production quality C code with a Java code that could potentially be optimized further.

The performance of Java Gadget-2 reinforces our belief that Java is a viable option for HPC. With careful programming, it is possible to achieve performance in the same general ballpark as C code.

In general, Java encourages better software engineering by being an object oriented language and is more portable than its precursors. Also, Java has many extra safety features including array bounds checking that could help identify potential bugs in the code. For example, we discovered a scenario in the original C Gadget-2 where seventh element of a six element array was accessed. The Java Gadget-2 helped identify this scenario by throwing a `ArrayOutOfBound` exception. We have informed the developer of C Gadget-2, who has fixed this problem in the distribution.

We plan to continue working on the software and make a public release in the future.

References

1. Mark Baker, Bryan Carpenter, and Aamir Shafi. An Approach to Buffer Management in Java HPC Messaging. In V. Alexandrov, D. van Albada, P. Sloot, and J. Dongarra, editors, *International Conference on Computational Science (ICCS 2006)*, LNCS. Springer, 2006.
2. J. Barnes and P. Hut. A Hierarchical O(N log N) Force-calculation Algorithm . *Nature*, 324(4):446–449, 1986.
3. Pedro Bizarro, Luís Moura Silva, and João Gabriel Silva. JWarp: A Java library for parallel discrete-event simulations. *Concurrency: Practice and Experience*, 10(11-13):999–1005, 1998.
4. The MONARC project. www.cern.ch/MONARC.
5. N. T. Padial-Collins, W. B. VanderHeyden, D. Z. Zhang, E. D. Dendy, and D. Livescu. Parallel operation of CartaBlanca on shared and distributed memory computers. *Concurrency and Computation: Practice and Experience*, 16(1):61–77, 2004.
6. John K. Salmon and Michael S. Warren. Skeletons from the treecode closet. *J. Comput. Phys.*, 111(1):136–155, 1994.
7. Volker Springel. The cosmological simulation code GADGET-2. *MON.NOT.ROY.ASTRON.SOC.*, 364:1105, 2005.
8. Volker Springel, Simon D. M. White, Adrian Jenkins, Carlos S. Frenk, Naoki Yoshida, Liang Gao, Julio Navarro, Robert Thacker, Darren Croton, John Helly, John A. Peacock, Shaun Cole, Peter Thomas, Hugh Couchman, August Evrard, Joerg Colberg, and Frazer Pearce. Simulating the joint evolution of quasars, galaxies and their large-scale distribution. *Nature*, 435:629, 2005.
9. Greg Wright, Matthew L. Seidi, and Mario Wolczko. An object-aware memory architecture. Technical Report TR-2005-143, Sun Microsystems, February 2005. http://research.sun.com/techrep/2005/abstract-143.html.