Collective Asynchronous Remote Invocation (CARI) Schedules

A High-Level and Efficient Communication API for Irregular Applications

By

Muhammad Wakeel Ahmad

2007-NUST-MS-PhD IT-12

Supervisor

Dr. Aamir Shafi

NUST-SEECS

External Collaborator

Dr. Bryan Carpenter

University of Portsmouth - UK

A thesis submitted in partial fulfillment of the requirements for the degree of Masters of Science in Information Technology (MS IT)

In

School of Electrical Engineering and Computer Science,
National University of Sciences and Technology (NUST),
Islamabad, Pakistan.

(November 2010)

Approval

It is certified that the contents and form of the thesis entitled "**Collective Asynchronous Remote Invocation (CARI) Schedules**" submitted by **Muhammad Wakeel Ahmad** have been found satisfactory for the requirement of the degree.

Advisor: **Dr. Aamir Shafi**

Signature: ________________
Date: ____________________

Committee Member 1: **Dr. Raihan-ur-Rasool**

Signature: __________________
Date: __________________

Committee Member 2: **Ms. Samin Khaliq**

Signature: __________________
Date: __________________

Committee Member 3: **Mr. Ali Sajjad**

Signature: __________________
Date: __________________

Abstract

Gadget-2 is a production parallel code for cosmological N-body and hydrodynamic simulations. Versions of this code featured in the Millennium Simulation, which is the largest simulated model of the universe. Gadget-2 has been parallelized for distributed memory platforms using the MPI standard. In this thesis — we analyzed Gadget-2 with a view to understanding what high-level SPMD communication abstractions might be developed to replace the intricate use of MPI in such irregular applications and do so without compromising the efficiency. Our analysis revealed that the use of low-level MPI primitives bundled with the computation code makes Gadget-2 difficult to understand and probably hard to maintain. In addition, we found out that the original Gadget-2 code contains a small handful of complex and recurring-patterns of message passing. We also noted that these complex patterns can be reorganized into a higher level communication library with some modifications to the Gadget-2 code. This thesis presents the implementation and evaluation of one such message passing pattern (or schedule) that we term Collective Asynchronous Remote Invocation (CARI). As the name suggests, CARI is a collective variant of Remote Method Invocation (RMI), which is an attractive, high-level, and established paradigm in distributed systems programming. The CARI API might be implemented in several ways — we implement and evaluate two versions of this API on a compute cluster. The performance evaluation reveals that CARI versions of the Gadget-2 code perform as good as the original Gadget-2 code but the level of abstraction is raised considerably. In fact one of the implementations has geared towards scalability on larger number of cores which performs the best when the problem size reaches its scalability limits.

Certificate of Originality

I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, nor material which to a substantial extent has been accepted for the award of any degree or diploma at NUST SEECS or at any other educational institute, except where due acknowledgement has been made in the thesis. Any contribution made to the research by others, with whom I have worked at NUST SEECS or elsewhere, is explicitly acknowledged in the thesis.

I also declare that the intellectual content of this thesis is the product of my own work, except for the assistance from others in the project's design and conception or in style, presentation and linguistics which has been acknowledged.

Author Name: **Muhammad Wakeel Ahmad**

Signature: ________________

Acknowledgments

First and foremost, I am immensely thankful to Almighty Allah the Most Gracious and the Most Merciful, for letting me pursue and fulfil my dreams. Nothing could have been possible without His blessings.

It is with pleasure that I express my affectionate and deeply felt gratitude for my esteemed thesis supervisor Dr. Aamir Shafi, for his supervision, guidance, and advice from the very early stage of this research. Much of what lies in the following pages can be credited to his kind supervision. His truly scientist hunch and passions in research exceptionally inspire and enrich my growth as a student and a researcher. Above all and the most needed, he provided me persistent encouragement and support in numerous ways. I am indebted to him more than he knows.

I gratefully acknowledge Dr. Bryan Carpenter for his supervision, valuable contributions, and whole-hearted cooperation which made this thesis possible. His involvement with his innovation has initiated and stimulated my intellectual maturity that I will benefit from, for a long time to come. Bryan, I am thankful in every possible way and hope to keep up our collaboration in the future.

I am also extremely thankful to my committee members Dr. Raihan ur Rasool, Ms. Samin Khaliq and Mr. Ali Sajjad for their support, valuable suggestions, and positive criticism. It would be quite unjust if I do not mention the helping hand offered by Mr. Hammad Siddique and Mr. Umar Butt during my stay at HPC Lab.

My parents deserve special mention for all their love, encouragement, inseparable support and prayers. They had more faith in me than could ever be justified by logical argument. My dear sisters and brothers have always supported me in all my pursuits. Thanks for being supportive and caring.

Finally, I would like to record my thanks for my friends, and colleagues specially at HPC, and WiSnet Labs for their enormous encouragement and sympathetic help, which made my life more pleasant and easier during research work. Let me grab this opportunity to thank everybody who was important to the successful realization of thesis, as well as expressing my apology that I could not mention personally one by one.

Muhammad Wakeel Ahmad

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Evolution of High Performance Computing (HPC)

The computer software industry has witnessed a sea change where single power hungry cores are making way for multiple power efficient processing cores [1]. The main reason is that increasing the clock speed exponentially increases power consumption and heat dissipation of the processor. As a consequence major microprocessor vendors like Intel, AMD, Sun, and IBM have shifted their business model to increasing cores instead of increasing clock speed [2].

In the context of High Performance Computing (HPC), the commodity clusters have emerged as an economical alternative to Symmetric Multi-Processors (SMPs) and Massively Parallel Processors (MPPs). A cluster typically consists of multiple PCs (usually called nodes) connected to each other via a high-speed network. In the past, each of the nodes in a cluster contained a single core processor. The cluster computers follow the distributed memory model where each single core processor inside a node communicates with its peers by message passing. Today's compute cluster is built with homogeneous multicore processors and follow a hybrid model where inter-node communication still occurs through message passing but intra-node communication is done by writing and reading (to and from) the main memory. However the future compute clusters will be based on heterogeneous combinations of high and low power cores, graphical processors, cache blocks and on-chip interconnect [3]. These changes in the Parallel Computing landscape promise exponential increase in the processing power in the next few years.

With the sea change in the computing hardware architecture, going from serial computers to parallel multicore processing systems, the software community has also faced a massive swing in its thinking. The heterogeneous many-core processor architectures (could) pose significant programming challenges as developing applications that exploit the full power of many-core systems is difficult and developers have to keep track of various problems ranging from writing safe multithreaded code to verification/testing of the complete software package that has to execute on a variety of processing elements. Some traditional computing applications related to computational physics and engineering, that have dominated Parallel Computing are relatively easier to parallelize as they have plentiful coarse-grained parallelism thus, making them easily developed with some of the current software packages. But unstructured and adaptive grids and complication of multi-physics simulations have taxed existing parallel machines and programming models and demand new ways of exploiting parallelism for contemporary unstructured applications.

1.2 Problem Definition

Message Passing Interface (MPI) [4] standard which continues to dominates the landscape of parallel computing as the de facto standard for writing parallel applications. But the critics of the MPI standard argue it is a low-level Application Programming Interface (API). We argue hthat if parallel computing is to become a mainstream tool to address the multicore challenge, the level of abstraction provided by parallel computing libraries and languages must be “raised”.

The current generations of multicore processors are typically shared memory devices. Naturally the programming models touted to program these processors are also shared memory models. But achieving a speedup in line with Moore’s law by concurrency alone would require a doubling of cores every 18 months. The number of cores in a chip would rapidly exceed widely assumed scalability limits for shared memory multiprocessors. There are already some working examples of multicore processors built using distributed NoC systems, an example is the TeraScale chip with 80 cores [5]. In the light of this, the Single Programming Multiple Data (SPMD) model seems to be an obvious candidate for programming parallel applications on the future multi/many-core processors. Also, it is relatively straight forward to port SPMD applications on shared memory multicore processors.

SPMD programming is perceived as being harder than other shared memory programming approaches. In this thesis, we address this issue of programming productivity by proposing a high-level, easy-to-use, and efficient programming API for multi/many-core processors. For this purpose, we analyze

Gadget-2 [6] with a view to understanding what high level SPMD communication abstractions might be developed to replace the intricate use of MPI in such an irregular application-and do so without compromising the efficiency. The Gadget-2 code-a real-world application chosen to drive our research-is based on a novel N-body simulation algorithm, which is also identified as a key dwarf in an influential Berkeley report [7]. Gadget-2 is a production parallel code for cosmological Nbody and hydrodynamic computations. Versions of Gadget-2 have been used in various astrophysics research papers, and in the Millennium Simulation, which simulated the evolution of the structure of the universe using 512 nodes of an IBM p690. It uses a distributed Barnes-Hut (BH) tree to compute gravitational and computational forces, with an approach to load balancing based on the Peano-Hilbert curve.

Our initial analysis of Gadget-2 revealed that the use of low level MPI primitives bundled with the computation code makes the source code difficult to understand and probably hard to maintain. In addition, the original Gadget-2 code contains a small handful of complex and recurring, a pattern of message passing, which essentially means that too much of the code is boilerplate message passing than physics. But the most interesting observation is that by transforming some loops in a meaning-preserving way, and defining some "callback" functions containing user-defined "physics" code, and absorbing the boilerplate message passing code into a library class, this complex pattern of explicit message passing can be reorganized into a high-level communication library with some modifications to the Gadget-2 code. The actual pattern of data movement and message passing is not changed by this reorganization, it will remains at least as efficient as the original code but the level of abstraction is raised considerably.

1.3 Research Hypothesis

High productivity communication libraries built on top of MPI can significantly reduce the programming complexities without incurring any significant loss in performance but the level of abstraction can be raised considerably.

1.4 Research Motivations

A panel with the title "What the parallel-processing community has (failed) to offer the multi/many-core generation" at the International Parallel and Distributed Symposium (IPDPS) 2008 [8] noted that the software community is still not ready to take on the multicore challenge. To tackle this, the panel agreed to stress the importance of a) applications as a driving factor in research and teaching, b) parallel algorithms, and c) focus on programming productivity (by providing higher abstractions [9] and not performance alone) in education, research, and software development. To repeat our research is guided by principles-focuses on applications and programming productivity-set out by the IPDPS panel.

1.5 Contributions of the Project

Main contributions of this thesis include:

1. A thorough analysis of the Gadget-2 code to find recurring message passing patterns.
2. Development of an efficient, high-level, and easy-to-use programming API called CARI.
3. Two implementations of the CARI API. One of the implementations outperform the MPI code demonstrating that performance can also be improved by providing higher-level abstractions.

1.6 Dissertation Overview

Rest of the thesis document is organized as follows. Chapter 2 presents a background and relevant literature. We review the current HPC hardware and software models, and then we try to cover the high productivity programming libraries and languages related to our work. This is followed by an introduction of the Gadget-2 code which is chosen as a sample application of our research work in the Chapter 3. In this chapter we briefly analyzed the parallelization strategy, and irregular communication patterns of Gadget-2 code. Chapter 4 presents the design and implementation of CARI Schedules where we have discussed the low level programming issue and possible implementations of CARI API. We evaluate and compare performance of the original Gadget-2 code with CARI versions in Chapter 5. We conclude and present future work in Chapter 6.

Chapter 2

Background and Literature Review

This chapter reviews the modern HPC hardware and software. The chapter begins by reviewing HPC hardware, emphasizing the emergence of multicore processor clusters, and parallel computing hardware models. The chapter also presents some of the popular programming models including shared memory and distributed memory models. In addition, we make an effort to cover the emerging parallel programming languages and APIs which have similarities to our work.

2.1 High Performance Computing Hardware

According to Top500 list the fastest computers today have reached the performance of teraflops and petaflops. The world's fastest supercomputer today, Cray XT5 known as Jaguar, an AMD Opteron based supercomputer at the Oak Ridge National Laboratory, operates at a processing rate of 1.75 PFLOPS per second with 224162 processing cores. The Nebulae system build from Dawning TC3600 Blade system with Intel X5650 processors and NVidia Tesla C2050 GPU is declared as No. 2 with performance of 1.271 PFLOPS per second. The Nebulae system from National Supercomputing Centre in Shenzhen (NSCS)China contains 120640 number of core. The IBM Blade-Center QS22/LS21 known as Roadrunner declared as world's third fastest supercomputer, at Los Alamos National Laboratory operates at a processing rate of 1.042 PFLOPS per second employing 122400 processing cores. The IBM Blue Gene named Kraken XT5, a PowerPC based supercomputer at Julich Research Center, operates at a processing power of 825 TFLOPS per second employing 98928 processing cores and is currently ranked as number four in the TOP 500 list as of June 2010 [10].

2.2 Parallel Computing Hardware Models

The Landscape of parallel computers has stabilized and evolved into different types which can classified based upon their memory architecture. The different types of parallel hardware range from machines with thousands of processors sharing nothing, to machines employing thousands of processors sharing memory. The understanding of the underlying hardware architecture is essential for writing efficient parallel software.

2.2.1 Massively Parallel Processors (MPPs)

Massively parallel processing system also known as Distributed Memory System is large computing facility employing thousands of processors each having its own memory leaving aside the concept of global address space.

The processors are connected to each other through a communication network which can be as simple as Ethernet and as fast as Gigabit Ethernet, Infiniband or Myrinet. The concept of memory/cache coherence doesn't apply in MPPs as each processor has independent memory. The task of achieving the desired optimized communication of data between the processors is left to the capability of the programmer in this kind of computing system.

Distributed memory computer systems can also be built from scratch using from off-the-shelf processor components. These kinds of systems are called Clusters instead of MPPs. Cluster computing has proved to be a cost effective option for many small-middle size organizations as their needs are easily fulfilled without having to purchase a large scale MPP.

2.2.2 Symmetric Multi-Processors (SMPs)

SMP systems contain thousands of processor components sharing the main memory and the memory bus [11]. Each processor may have its own cache on a dedicated bus but all the processors are connected to a common memory bus and memory bank sharing the same memory resources. Changes in a memory location effected by one processor are visible to all other processors. Shared memory machines can be divided into two main classes based upon the memory access times: UMA and NUMA.

Uniform Memory Access (UMA)

The machines employing Uniform Memory Access typically have identical processors having equal access and access times to the memory. Sometimes, it is often called Cache Coherent UMA which means that if one processor updates a location in shared memory, all the other processors know about the update [11].

Non Uniform Memory Access (NUMA)

The machines employing Non Uniform Memory Access are often lined physically by two or more SMPs having UMA where one SMP can directly access memory of another SMP. Unlike UMA, not all processors have equal access time to all memories. A machine with NUMA is explained in Figure 2.3.

2.2.3 Hybrid Distributed Shared Memory

Most of the parallel computers today employ both shared and distributed memory architectures. The shared memory component is usually a cache coherent SMP machine and the distributed memory component is the net-

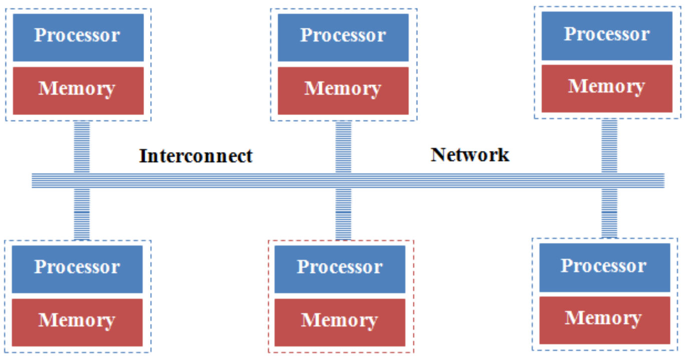


Figure 2.1: Massively Parallel Processors (MPP) Model.

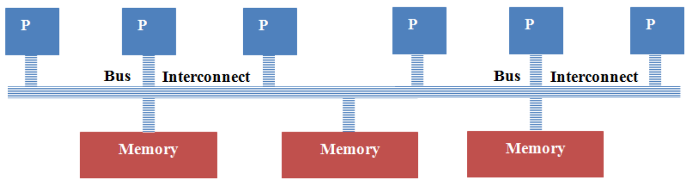


Figure 2.2: Uniform Memory Access (UMA) Model

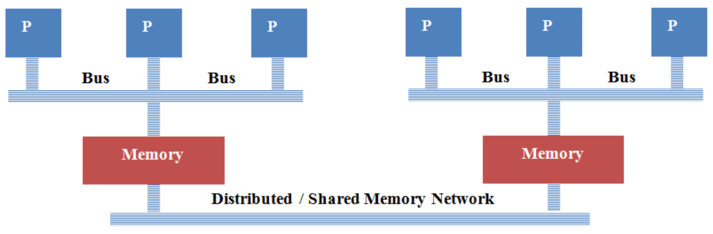


Figure 2.3: Non Uniform Memory Access (NUMA) Model.

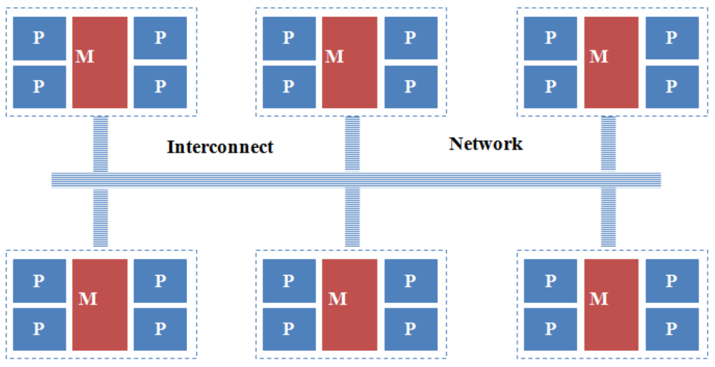


Figure 2.4: Hybrid Distributed Shared Memory Model.

working of multiple SMPs. Current trends seem to indicate that this type of memory architecture will continue to prevail and increase at the high end of computing for foreseeable future.

2.3 Parallel Computing Software Models

There are various software models available for programming the different types of parallel computers. Some of them are, shared memory model, message passing model, data parallel model, threaded model etc. The point worth noting is that these models are not specific to any particular type of hardware architecture i.e. it is possible to use a shared memory programming model on a MPP system. In the following sections, we discuss shared memory programming model and message passing model in detail.

2.3.1 Shared Memory Model

In Shared Memory programming model, a common address space in the memory is shared by all the processes where they can read and write asynchronously. In order to implement control access strategy among different processes performing different tasks, various mechanisms such as locks and semaphores are used.

The simplicity of Shared Memory model brings some advantages and some disadvantages. An important plus, while using Shared Memory model, is the facility of not specifying the communication of data between the processes as every process will read its share of data from the shared memory. This makes developer's life much easier. A critical minus is that, when multiple processors are using the same data, updating the data among all the processors conserves memory accesses which may result in performance issues.

OpenMP

OpenMP, short for Open specifications for Multi-Processing, is an open standard jointly defined by a group of major computer hardware and software vendors for providing parallelization mechanisms on shared-memory processors in 1997. The specification supports C/C++, FORTRAN and recently Java. The specification is composed of three main components:

- Compiler Directives
- Runtime Library Routines
- Environment Variables

OpenMP is based upon a thread paradigm. It creates multiple threads to counter the parallel regions of the code which are synchronized and terminated when the individual threads executing the parallel code terminate. The model, known as, Fork-Join Model, ensures parallel execution of the program [12].

The thread paradigm is the logical choice for Shared Memory multiprocessor. When the master thread encounters a directive to fork off new threads, it creates new threads to execute parallel regions of the program. When the team threads complete the statements, they synchronize and terminate leaving only the master thread.

OpenMP, being the standard for Shared Memory parallelism, is not applicable on distributed memory parallel systems i.e. MPPs. The runtime routines do not guarantee the most efficient use of shared memory; it is up to the ability of the programmer. The runtime also doesn't check for data dependencies, data conflicts, race conditions and deadlocks.

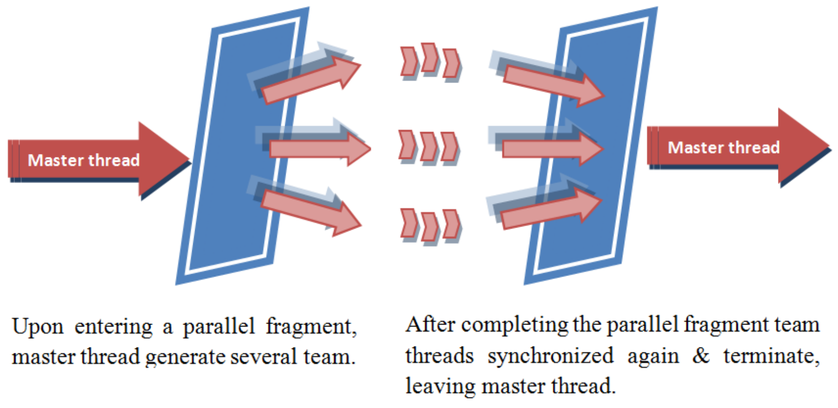

Master thread
Master thread
Upon entering a parallel fragment, master thread generate several team.
After completing the parallel fragment team threads synchronized again & terminate, leaving master thread.

Figure 2.5: OpenMP Thread Mechanism.

2.3.2 Message Passing

Message Passing model is mostly used for programming distributed memory systems. Message Passing model consists of processes or tasks that use their local memory during computation. Since the data is not shared through the memory, the processes exchange the data through communications by sending and receiving messages. Data transfer usually requires cooperative operations to be performed by each process. For example, a send operation must have a matching receive.

Message Passing implementations comprise a library of subroutines that are imbedded in source code. The programmer is responsible for parallelizing the code by calling the appropriate routines.

Parallel Virtual Machine (PVM)

The PVM is a message passing system that views distributed memory machines as a single virtual machine. It is designed to allow a network of heterogeneous UNIX and/or Windows machines to be used as a single distributed parallel processer. Large computational problems can be solved more cost effectively since it permits the heterogeneous collection of the platforms.

The "virtual machine" is the central component of PVM i.e. a set of heterogeneous hosts connected by a network that appears logically to the user as a single large parallel computer. Portability was considered much more important than performance for two reasons: communication across the internet was slow; and, the research was focused on the problems with scaling, fault tolerance, and heterogeneity of the virtual machine [13].

PVM consists of a runtime environment and library for message passing, task and resource management, and fault notification. PVM provides a powerful set of functions for manually parallelizing either an existing program or a new parallel program. Before the advent of MPI, PVM was the de facto standard for message passing on distributed systems and is still used in programs that demand heterogeneity of the platforms.

Message Passing Interface (MPI)

MPI is a message passing library standard specified by a commission of high performance computing experts (from research and industry) in a series of meetings in 1993-1994. The motivation for developing MPI was that each MPP vendor was creating its own branded message-passing API. In this situation, it was becoming impossible to write a portable parallel application. MPI is envisioned to be a standard message-passing specification that each MPP vendor would implement on their system. The MPP vendors need to be able to provide high performance and this became the focus of MPI. Given this design focus, MPI is likely to always be faster than PVM on MPP hosts. MPI-1 contains the following main features:

- Set of point-to-point communication routines
- The ability to specify communication topologies
- The ability to create derived data types that describe messages of non-contiguous data
- Set of collective communication routines for communication among group of processes
- Communication context that provides support for the design of safe parallel software libraries

MPI-1 users soon discovered that their applications were not portable across a network of workstations because there was no standard method to start MPI tasks on separate hosts. In 1995, the MPI committee began meeting to design MPI-2 specification to correct this problem and to add several additional functions to MPI. The message passing model is applicable to virtually any distributed memory parallel programming model. MPI is also used to implement shared memory models such as Data Parallel model on distributed memory architectures [4].

As discussed earlier the two most popular parallel computing platforms include the shared memory and distributed memory hardware. For the shared memory hardware, the preferred approach may be direct programming with threads and locks, which is relatively hard. To overcome this difficulty, higher-level task parallel approaches have been widely advocated. In the task parallel approach, work is divided into independent or loosely coupled tasks. Systems that fall into essentially this camp include OpenMP [14], Intel Cilk++ Software Development Kit (SDK) [15] and Threading Building Blocks [16]. For the distributed memory hardware, the preferred approach may be the SPMD programming model. An enduringly popular approach is to simply write node programmes in standard C or Fortran, using a message passing library-today typically MPI-for communication. More recently, the SPMD programming model is also emerging as a viable option for the future multi/many-core processors.

In HPspmd [17] programming model (for distributed memory machines) a base language is extended with syntax for defining distributed (partitioned) data, but the extended language is agnostic about communication that is subsumed into high-level collective library calls. For a wide class of regular problems the HPspmd model and the original Adlib [18] library provide elegant programming solutions. Of course many computational problems have more irregular structure, and it was always clear that the original Adlib library would not provide convenient or efficient solutions for all of those. In this effort we try to fill this gap by providing communication schedules for irregular applications that could naturally be integrated into the HPspmd programming model. In addition, the current schedule is available in the form of an independent library that relies only on a C++ compiler and an MPI library.

2.4 High Productivity Programming Models

In this section we have discussed modern approaches used for providing higher-level abstractions including the Global Arrays Toolkit [19] and CHARM++ [20]. Our work has similarities to both of these system. There is a superficial analogy of CARI with Active Messages (AM) as well.

2.4.1 CHARM++

A language that attempts to increase programming productivity without compromising performance. CHARM++ system is a set of extensions to the C++ language without changing its syntax and semantics, except the global and static data of the class. It is a comprehensive system for parallel Object Oriented Programming (OOP). CHARM++ extends its mother language characteristics like *multiple inheritance, dynamic binding, overloading*, and *strong typing* for its parallel objects.

The central component of the CHARM++ language is the *Chare* object, which is globally defined and can be asynchronously invoked using the so-called entry methods. Asynchronous method invocation means that the caller (method) does not wait for the method to be actually executed, and for the method's return value. Therefore, CHARM++ entry methods do not have a return value. In CHARM++ an object can resides at remote processor thats why CHARM++ use "proxy" to refer it, instead of C++ style via pointer. The "proxy" class methods act like "forwarders" which corresponds to the remote methods of the actual class.

CHARM++ Execution Model

Instead of the traditional blocking-receive-based communication model, CHARM++ execution model is based on message driven strategy. By message driven means computations are triggered on the arrival of an associated message. On the arrival of a message at a processor, it is put into a queue where the system schedules the target object to process it depending upon its priority. Message driven strategy helps CHARM++ to tolerate communication latency.

In a CHARM++ program, a *Chare* can be created at run time, where the number of *Chare* objects varies during the execution time of the program. CHARM++ system maintains a work-pool, to manage messages of existing *Chare(s)* and the seeds for new *Chare*. The CHARM++ kernel non-deterministically picks item from this work-pool and execute them.

All the entry methods are non-preemptive; it means that CHARM++ will never interrupt an executing method to start any other work. CHARM++ provides seed based load balancing, it means while creating the remote *Chare*, destination processor (where newly created Chare will resides) number is not required to be specified. Afterwards CHARM++ kernel assigns the newly created *Chare* object to least loaded processor. CHARM++ language endow with dynamic load balancing in such a way that a *Chare* can migrate from one processor to another. However only never-run *Chare* can migrate, once a task is running it cannot be migrated.

Entities in a CHARM++ programs

Given below are the few important entities in a typical CHARM++ program.

- **Sequential Objects** are just like an ordinary sequential C++ code and objects. Such entities are only accessible locally; it means that CHARM++ runtime system is not aware of these entities.

- **Message** objects are used to supply data arguments while asynchronous remote method invocation.
- **Chares** which are normally consider the most important entities in CHARM++ program. Infect it is the basic unit of parallel computation in CHARM++ programs, like a process in MPI programs. *Chares* are the instance of CHARM++ class just like class objects in sequential C++ programming but the difference is that they can be created and invoked asynchronously on remote processors, using public methods. The public methods are called entry methods, which do not have any return type.
- **Chare Arrays** are collections of *Chares*.
- **Chare Groups** are special type of coexisting objects. Each Chare-group is a set of *Chares*, with one group member on each processor. An entire Chare-group could be addressed using globally unique name, and an individual member of a Chare-group can be accessed using the global handle, and a processor number.
- **Chare Node Groups** have similar concept to Chare-groups with the exception of having one group member on each processor. Instead node-group has one member on each shared memory multiprocessor node.

Some other rich characteristics of CHARM++ include *parameters marshalling, dynamic load balancing, callbacks, quiescence detection, reductions*, and many more. Because of these features it stands as an appropriate programming language for a broad range of applications based on either regular or irregular computations. However the implementation of CHARM++ is somehow (if not at all) similar to MPI, hence we have to deal with the same complications as we did in MPI. In other words CHARM++ programming effort is nearly same as with old programming models. The difference is that now instead of MPI sends and receives, we have to use entry methods to request for data or send data to remote processors. The major difference as compare to MPI is that now we can have a better virtualization which enhances the overlap of communication and computation.

CHARM++ have some features in common with the CARI API, notably the asynchronous communication calls and the message driven execution model. But the CARI API is distinguishable due to many aspects. For example in CARI, an incoming message (which is known as request and abbreviated as REQ) is not put into the local waiting-queue like CHARM++.

In CARI API when a REQ arrived at a remote processor it would be served immediately and the computed results are sent back via message (we called it acknowledgement abbreviated as ACK). Instead of runtime system CARI owned a server thread running all the time to respond the client's request which switch its states to REQ and ACK mode alternatively. The other notable difference is that the CHARM++ system does not support the concept of collective completions (briefly explained in Chapter 4), which is an important requirement in our pipelined system of invocations. Moreover, CHARM++ is a much more complicated system that includes advanced features (like object marshalling), which are not part of our simpler and less ambitious API. Programmers usually prefer a light weight API instead of a complicated gigantic system.

2.4.2 Active Messages (AM)

The AM encompasses the vital functionality of message driven models with simpler hardware mechanisms. It provides an efficient mechanism to reduce the software overhead in message passing by overlapping communication to computations to exploit the full capacity of the hardware.

Execution Model

Using AM, messages are transferred in a pipeline that operates at a rate which can be determined by the communication overhead, network depth, and with the latency related to the message length. The sender node kicks off the message into the network and carries on computing. On the other hand receiver is interrupted on message arrival and runs the handler. The header of each message contains the address of a user level handler which will be executed on message arrival with the message body as an argument. In AM, message-packets are not buffered (apart from when required for network transport) to keep the communication overhead to a minimum limit.

There is a superficial analogy of CARI API to Active Messages (though Active Messages is not normally considered a high level API, since it provides low-level messaging primitives). The distinctive characteristic of the CARI schedule is the way it combines asynchronous result processing through callbacks with collective completions, and the specific MPI implementations with bounded communication buffers.

2.4.3 Global Arrays (GAs) Toolkit

Global Arrays toolkit blends the best features of *Shared Memory* programming style with *Message Passing* programming model, notably simplified & efficient coding with high portability respectively. In other words we can say that it provides a portable shared memory programming interface for distributed memory architecture machines.

In GA programming environment each process in a MIMD parallel program can asynchronously access logical blocks of physically dispersed data (stored in multidimensional distributed arrays also known as Global-Arrays) without explicit assistance from other remote processes. GA objects are used to encapsulate data distribution and addressing details, such that whenever data locality information is required it can be taken easily and efficiently. Shared data structure is logically divided into two portions “local” and “remote”, where the local portion of data stored in shared memory which is supposed to be faster assessable as compare to the remote portion. But this difference do not cause any impede in ease of use since GA library provides uniform access for shared data regardless where it is located local or remote portion.

GA also supports few one sided shared memory operations like *get, put, gather, read-and-increment, reduction,* and *lock*. These operations can only be performed on the Global-Arrays data structure rather than random memory locations. These one sided operations can complete regardless of action taken by remote process (that own the referenced data). Also, while accessing remote data in GA, one does not required to specify the target process (where does the actual data reside) unlike other similar system. The reason is that GA provides global-view of the data structure through its index-based-transfer interface, such that wherever required, GA library internally performs the global array index-to-address translation and transfer the data accordingly. The architecture design of GA is more appropriate to those applications which requires block-wise physical distribution of data, dynamic load balancing, or have a fairly large ratio of computation to data movement.

Our approach is distinguished by the clear factorization of the system into a common framework for representing distributed arrays on the one hand, and an extensible family of high level communication libraries on the other. One library might specialize in regular data remapping; another may provide some flavor of collective get/put functionality; yet others provide computational functions over distributed data sets.

Some other library-based approaches are based explicitly on the BSP model [21] of Valiant. These usually provide less structured primitives for message exchange and remote memory access, integrated with barrier synchronization. Other authors have elected to extend the programming language to include threads, and affinity of data to those threads. The PGAS family of languages that includes Co-Array Fortran [22] and Unified Parallel C (UPC) [23] is representative. These languages typically add syntax for defining arrays that are partitioned over threads, together with some primitives for accessing elements owned by other threads. A criticism of this approach is that it typically "hard wires" both the available partitioning strategies and the communication primitives, into the definition of the programming language. These selected communication primitives often have semantics similar to ordinary assignment statements. A runtime system then implements something like a distributed shared memory model.

Chapter 3

Gadget-2: A Code for Cosmological Simulations of Structure Formation

3.1 Overview

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 (for example cosmological-scale) systems under the influence of gravitational and hydrodynamic forces. The system is modeled by a sufficiently large number of test particles, which may represent ordinary matter or dark matter. The main simulation loop increments time steps and drifts particles to the next time step.

To understand the scale of interesting problems, consider the "Millennium Simulation" [24]. This simulation tracks the expansion of 10^{10} dark matter particles from the early universe to the current day. It was executed on 512 processors with 1 Terabyte of distributed memory. The simulation used 350,000 CPU hours over 28 days of elapsed time. It used an upgraded version of Gadget-2.

We are particularly interested in the parallelization strategy of Gadget-2, which is based on irregular and dynamically adjusted domain decomposition, with copious communication between processors.

3.2 Computing Gravitational Forces

One of the main tasks of a structure formation code is to calculate gravitational forces exerted on a particle. Since gravity is a long-range force, every particle in the system exerts gravitational force on every other particle. A naive summation approach costs $O(N^2)$, which is not feasible for the scale of problems that Gadget-2 aims to solve. To deal with this, Gadget-2 can use either of two efficient algorithms. The first is the well-known Barnes-Hut (BH) oct-tree-based algorithm [25]; the second is a hybrid of BH and a Particle Mesh (PM) method called TreePM.

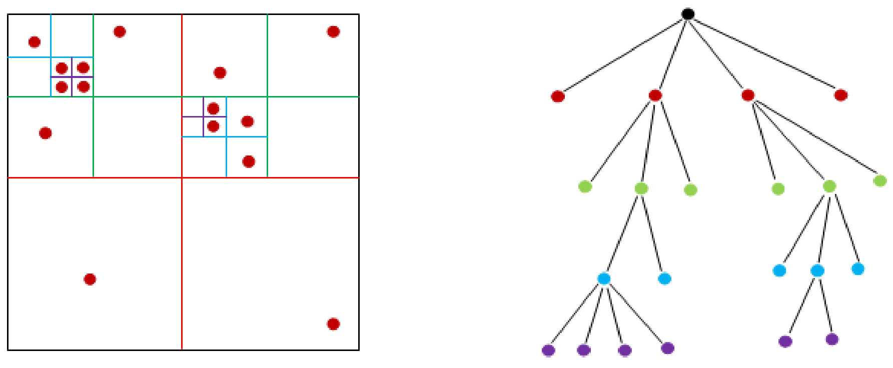


Figure 3.1: Barnes-Hut Tree Construction in 2D Space.

The Barnes-Hut algorithm is a smart scheme for grouping together bodies (nodes) that are sufficiently close. In the BH tree algorithm, the cubical region of 3D space under consideration 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 tree is traversed from root to compute the force on each particle i. The daughter nodes of node n are visited recursively if n is too close to i to be treated as a single mass. For a node n that is sufficiently distant from particle i, a contribution to the force on i is added from the centre of mass of n. Using this approach, it is possible to estimate the gravitational force for each particle in $O(log(N))$ steps. Figure 3.1 illustrates construction of quad Barnes-Hut tree in a 2D space.

Here each node represents a region of the two dimensional space. The topmost node represents the whole space, where its four children represent the four quadrants of the space. The space is recursively divided into quadrants until each subdivision contains maximum one body or empty at all. Hence each external node represents a single body where each internal node represents the group of bodies below it, and stores the center-of-mass and the total mass of all its children.

In the TreePM algorithm the gravitational potential is artificially split into a short range and a long range part:

$$\frac{Gm}{r} = \phi^{\text{short}}(r) + \phi^{\text{long}}(r) \tag{3.1}$$

where:

$$\phi^{\text{short}}(r) = \frac{Gm}{r}\text{erfc}\Big(\frac{r}{2r_s}\Big) \tag{3.2}$$

$$\phi^{\text{long}}(r) = \frac{Gm}{r}\left[1 - \text{erfc}\Big(\frac{r}{2r_s}\Big)\right] \tag{3.3}$$

where erfc is the complementary error function and r_s is some threshold radius. ϕ^{short} is calculated using the BH tree as above, but now ϕ^{long} is calculated by projecting particle mass distribution onto a mesh, then working in Fourier space to calculate the potential rapidly.

As a massively parallel code, Gadget-2 needs to divide space or the particle set into "domains", where each domain is assigned to a single processor. Generally speaking, dividing space evenly would result in poor load balancing, because some regions have more particles than others. Conversely, it is not desirable to divide particles evenly *in a fixed way*, because one wishes to keep physically close particles on the same processor, and the proximity relation changes as the system evolves.

Gadget-2 uses a decomposition based on the space-filling *Peano-Hilbert* curve. Particles are sorted according to their position on Peano-Hilbert curve, then divided evenly into P domains.

3.3 Communications Analysis

The Gadget-2 code is parallelized using MPI, and it makes extensive use of MPI point-to-point and collective communication functions. Some parts of the code are parallelized straightforwardly using essentially regular patterns of communication that can easily be captured in MPI collectives. In such code sections we consider there is a clean factorization between the application code and the (MPI) library code that abstracts the non-trivial aspects of inter-process communication.

But we have identified several code patterns in Gadget-2 that we consider "non-trivial" where communication is not "cleanly abstracted". In these sections there is a relatively adhoc interleaving of application-specific code and MPI *point-to-point* and *collective calls.* The patterns we identified were

1. A partial distributed sort of particles according to Peano-Hilbert key: this implements domain decomposition.

2. Projection of particle density to regular grid for calculation of ϕ^{long}; scatter results back to irregularly distributed particles.

3. Export of particles to other nodes, for calculation of remote contribution to force, density, etc, and retrieval of results.

We might have added a fourth pattern: a distributed Fast Fourier Transform, used in the TreePM algorithm. Gadget-2 uses the FFTW library [26] to implement this transform. The parallel FFT routines in FFTW assume input and output data arrays are distributed block-wise in one array dimension. There is scope for more systematic representation of these arguments as distributed arrays (see for example [27]). But actually this "fourth pattern" is already quite effectively abstracted by the use of FFTW. The question is really whether the other three patterns are amenable to a similar "factorization".

The first of the patterns enumerated above is essentially a distributed sort based on the Peano-Hilbert key. Throughout most of the Gadget-2 code, particles are distributed "irregularly" in a Barnes-Hut tree according to the Peano-Hilbert key (the exact distribution typically changes from time-step to time-step). In practise Gadget-2 does not perform an exact sort to establish the distribution. Instead it performs an approximate sort that exploits features of the application. But it seems likely that if an appropriate distributed sort had been readily available as a library function, the authors of Gadget-2 could have used this instead (on a similar footing to the use of the parallel FFT). This should lead to a clean factorization between communication code (mostly abstracted in the distributed sort)and application code. Based on this assumption, we defer further consideration of this pattern.

The second of the patterns enumerated above is part of the TreePM algorithm. The computation of ϕ^{long} is performed on a regular mesh by Fourier analysis. This mesh is distributed blockwise over processors. The communication pattern of interest here emerges when particle density is transferred from the irregularly distributed tree to the regular mesh, and when results for ϕ^{long} are transferred back. In Gadget-2 the MPI code that does this is interleaved in a complex way with application specific code. But again there is reason to believe that the communication operations required here could be "factored out" of the application code quite effectively. The operations required correspond closely to the `nga_acc` and `nga_get` functions of the Global Arrays Toolkit [27], for example. One may legitimately ask whether an approach based on one-sided communication approach would achieve the same efficiency as the "collective" implementation in the existing Gadget-2 code. Again we defer discussion to later.

This leaves the third pattern "particle export". This pattern occurs repeatedly in Gadget-2. Specifically, in the freely available source code, it recurs in the functions `compute_potential`, `gravity_tree`, `gravity_forcetest`, `density`, and `hydro_force`. Some of these functions are responsible for calculation of gravitational forces and potentials, and others are responsible for computation of hydrodynamic forces. The next subsection explores this pattern in more detail.

```
for( ... all local particles ... ) {
        ... Compute local contribution to potential, and flag exports ...
        ... Add exports to send buffer ...
}
. . . Sort export buffer by destination ...
for( ... all peers... ) {
        ... Send exports to peer; recv particles for local computation ...
}
... Compute local contribution to received particles ...
for( ... all peers ... ) {
        ... Recv our results back from peer; send locally computed ...
        ... Add the results to the particles in the P array ...
}
```

Figure 3.2: Simplified Sketch of `compute_potential`.

3.4 Particle Export

Figure 3.2 is a simplified schematic of the `compute_potential` function, one of several functions in Gadget-2 that follow the "particle export" pattern. The three lines (in bold non-italicized font) highlighted in red in Figure 3.2 contain all code that is specific to the physics problem being solved. As the local particles are processed in the first for loop, those whose potentials require a contribution from particles held remotely are flagged. All peer processors that contribute to the local particle's potential are recorded. If the local particle needs remote contributions, it is added to a send buffer. In fact it is added once for each contributing peer processor. Figure 3.3 describes the usage of communication buffer by Gadget-2 code at various stages of the code.

The send buffer is then sorted according to peer processor id. Sections of the send buffer are sent to each peer in the list. This is done in using `MPI_Sendrecv` operations, which concurrently receive particles exported to this processor by peers into a receive buffer. The local contribution to the received particles is computed. The results are sent back to the peer needing them, again the same `MPI_Sendrecv` operations that receive back results for particles we exported. The returning results are accumulated into the main particle array (the P array) held locally. Figure 3.4 depicts the execution patterns for Gadget-2 code at different interval of time when four processes are communicating with each other.

The actual code of `compute_potential` is complicated by the fact that Gadget-2 allocates fixed size buffers for communication. Figure 3.5 is more representative of the real structure. The more complex looping structure here is shaped largely by the need to manage communication buffers. The first for loop now terminates when all local particles have been processed or the send buffer is full, and a new outer loop is needed in case the send

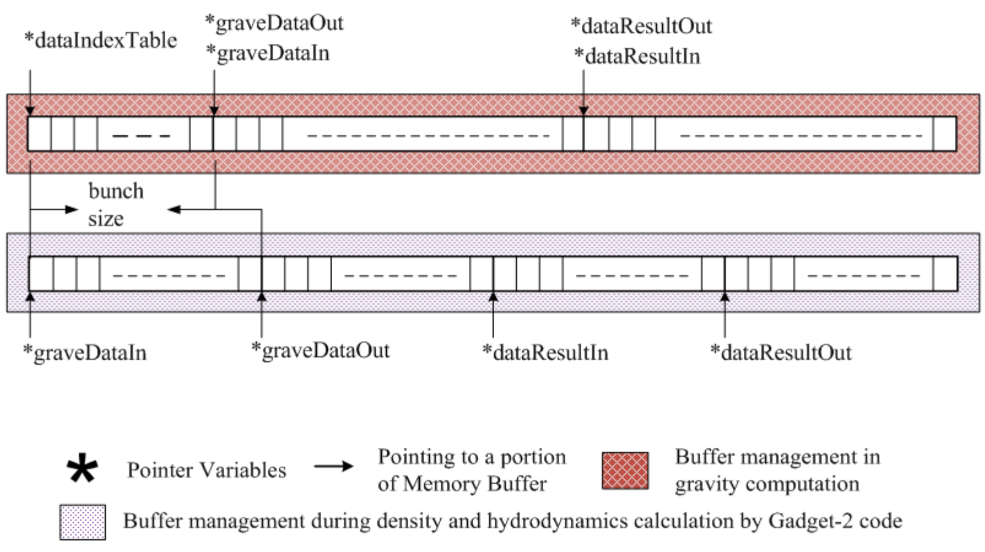

*dataIndexTable
*graveDataOut
*graveDataIn
*dataResultOut
*dataResultIn
bunch size
*graveDataIn
*graveDataOut
*dataResultIn
*dataResultOut
Pointer Variables
Pointing to a portion of Memory Buffer
Buffer management in gravity computation
Buffer management during density and hydrodynamics calculation by Gadget-2 code

Figure 3.3: Gadget-2 Communication Buffer.

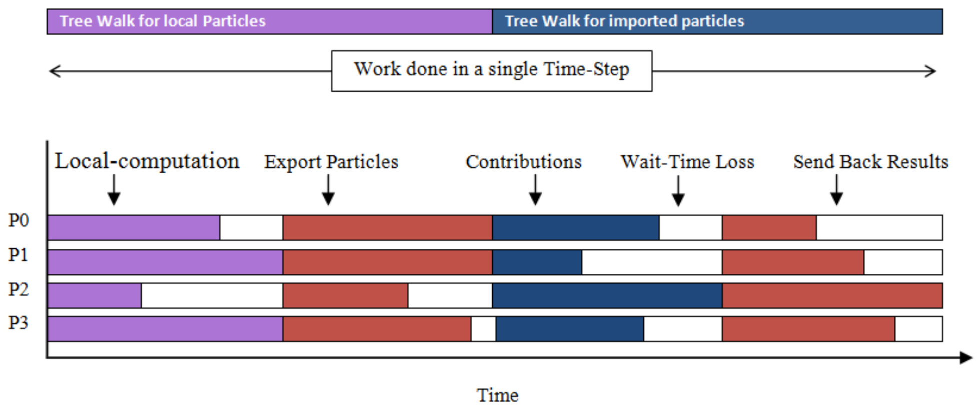

Tree Walk for local Particles
Tree Walk for imported particles
Work done in a single Time-Step
Local-computation
Export Particles
Contributions
Wait-Time Loss
Send Back Results
P0
P1
P2
P3
Time

Figure 3.4: Execution Patterns for Gadget-2 Code.

buffer was exhausted. Additional complications are needed to manage receive buffers. It is necessary to ensure that the export operations converging on any particular processor do not exhaust the receive buffer on that processor.

```
while( ... some particles not yet processed ...) {
  for( ...remaining local particles, while send buffer not full...) {
    ...Compute local contribution to potential, and flag exports ...
    ... Add exports to send buffer...
  }
  ... Sort export buffer by destination ...
  while( ... some peers not yet processed ...) {
    for( ...all remaining peers, while no recv buffer exhausted...) {
      ... Send exports to peer; recv particles for local computation ...
    }
    ...Compute local contribution for received particles . . .
    for( ...all peers communicated with above ...) {
      ...Recv our results back from peer; send locally computed ...
      ...Add the results to the P array ...
    }
  }
}
```

Figure 3.5: More Realistic Structure of `compute_potential`.

The Gadget-2 code broadcasts a matrix containing size of *all* sends from all processors before entering the communication loops. So every process can determine how much data converges on every processor, and collectively group sends and receives so that no receive buffer is ever exhausted.

The real implementation of `compute_potential` is about 300 lines of *C*, with most of the application specific code relegated to separate functions. Figures 3.2 and 3.5 only reproduce its general structure. The point to observe is that application specific code is scattered through the main "skeleton" of communications-related code. The question that concerns us how the code can be re-factored in such a way that it does not compromise the efficiency of the existing production code, but such that there is clean separation on of application specific code and communication code.

3.5 Code Options

A list of compile time switches are provided in the Makefile, while the other custom configuration parameters are provided in the parameter-file available with the code. Parameter file has to be specified at command line while running the code. Particularly the parameter file is used to specify the initial condition files, the start and finish time of the simulation, and the output directory. Some interesting compile time switches from Makefile are shown in Table 3.1.

A small change in the Makefile options, Gadget-2 code requires a full recompilation. Table 3.2 list a few options from parameter-file. Some other interesting parameters include `Omega0, OmegaLambda, NumFilesWrittenInParallel, TreeAllocFactor, PartAllocFactor, TimeLimitCPU`, etc.

1	PMGRID	This switch enables the TreePM method, i.e. the long-range force is computed with a PM-algorithm, where the short range-force with the tree. The parameter has to be set to the size of the mesh that should be used, e.g. 64, 96, 128, etc.
2	DOUBLEPRECISION	Enabling this makes the code compute internal particle data in doubleprecision.
3	DOUBLEPRECISION FFTW	If this parameter is enabled, the code will use the double-precision version of FTTW provided that FTTW is configures with double precision.
4	OUTPUTPOTENTIAL	If this is enabled, it will force the code to compute and store gravitational potentials for all particles each time a snapshot file is generated.
5	OUTPUTACCELERATION	Physical acceleration of each particle will be stored in the snapshot files if this flag is enabled.
6	FORCETEST	This flag can be set to check the force accuracy of the code, as debugging option. Code will generate "forcetest.txt" file for inspection.
7	PEANOHILBERT	If this flag is set, it will bring the code in Peano-Hilbert order after domain decomposition.
8	PERIODIC	Used to turn on the periodic boundary conditions.

Table 3.1: A List Of Selected Compile-Time Flags For Gadget-2 Code

1	OutputDir	Pathname of the directory that will hold all the output generated by the simulation eg snapshot files, restart files, diagnostic files, etc.
2	SnapFormat	This flag specifies the file-format to be used for writing the snapshot files.
3	InitCondFile	This sets the filename of the initial conditions to be read in at start-up.
4	NumFilesPerSnapshot	Each snapshot file can be distributed onto several files. This parameter used to indicate the number files per snapshot.
5	TimeMax	This marks the end of the simulation.
6	BoxSize	The BoxSize can be specified in the parameter file to specify the simulation box, when the periodic boundary conditions are turned on.
7	BufferSize	This flag specifies the size of communication buffer used by the code in various parts of the code.

Table 3.2: A List Of Selected Run-Time Parameters For Gadget-2 Code

Chapter 4

CARI Schedules: Design and Implementation

4.1 Collective Asynchronous Remote Invocation (CARI) Schedules

We use the term *schedule* for the object implementing message passing pattern, the term was popularized by CHAOS/PARTI libraries. As the name CARI suggests, this schedule is a collective variant of *Remote Method Invocation* (RMI), which is an attractive, high-level, and established paradigm in distributed systems programming. The surprise is to find a collective variant of RMI hiding in a production, massively parallel, message passing code.

A reorganization of the code that achieves our objective is given in Figure 4.1. Here CARI is a library class that abstracts all communication. The word `sched` is short for "schedule"; we regard the instantiated object as a kind of communication schedule. The main loop essentially follows the same structure as the first loop in Figure 3.2, but instead of manually adding exports to the send buffer, the invoke method is called on the `sched` object. This loop contains the first section of application specific code from Figures 3.2 and 3.5. The other two sections of application code are in the callbacks `requestHandler` and `responseHandler`.

If the implementation of CARI corresponded to Figure 3.2, the `invoke` method would simply add the particle (of struct type `S`) to the send buffer. The complete method would execute the code in Figure 3.2 following the "main loop", calling the user-defined callbacks at appropriate points. If the implementation of CARI corresponds to Figure 3.5, the `invoke` method adds the particle to the send buffer and checks if the buffer is full. If it is, `invoke` sorts the buffer and runs the "peer processing" loop, wherein MPI communications and user callbacks are called.

```
CARI sched(requestHandler, responseHandler) ;

for( ... all local particles ... ) {
      ... Compute local contribution to potential, and flag exports ...
      for( ... all peers particle should be exported to ... ) {
            sched.invoke(peer, particle) ;
      }
}
sched.complete() ;

void requestHandler(S* request, T* response) {
      ... Compute local contribution to received particle ...
}
void responseHandler(T* response) {
      ... Add the result to the P array ...
}
```

Figure 4.1: Refactored Code of `compute_potential`.

In this implementation, `complete` will likewise sort the send buffer, then run the “peer processing” loop as many times as necessary until all exports by all processors have been dealt with, globally (detecting termination involves a reduction operation). Because of the stylized way in which we have presented the original Gadget-2 implementation, the advantage of Figure 4.1 over Figure 3.5 may not be immediately apparent. But in reality the communication code in Figure 3.5 that is abstracted away in the CARI class is one to two hundred lines of rather dense MPI; in Figure 4.1 this is replaced by a few method calls and definitions. But it is important to re-state the fact that we can go from Figure 3.5 to Figure 4.1 with essentially no changes in the underlying communication pattern, or efficiency of the programme.

The pattern in Figure 4.1 may be recognized as a basic kind of remote method invocation. The user-defined callback `requestHandler` is the implementation code for the remote invocation (on the peer that is acting as “server”). The invoked method takes exactly one struct-type parameter of type `S`, and produces exactly one struct-type result of type `T`.

4.2 CARI Schedules are Asynchronous

The invocation of methods are asynchronous in the sense that the `invoke` method does not in general wait for the invocation to complete and results to come back before returning. Instead it may return immediately, and a user-defined callback function `responseHandler` processes the result locally when it does eventually return to the “client”.

4.3 CARI Schedules are Collective

The pattern is "collective" in a couple of senses.

1. All peers potentially act symmetrically as clients and as servers.
2. Creation of the CARI schedule is also a collective operation.
3. The logical synchronization that marks completion of all invocations occurs in the strictly collective `complete` method.
4. The `invoke` methods themselves may make use of collective methods for their implementation, as in the Gadget-2 inspired implementation implied above. But this is strictly an implementation issue. From a logical point of view the `invoke` method is not collective. Different peers make different numbers of `invoke` calls. Some may make none. We refer to this pattern as *Collective Asynchronous Remote Invocation* (CARI).

4.4 Implementation of the CARI API

As part of our research presented in this thesis, we have developed two implementations of the CARI API.

- Synchronous CARI (SCARI)
- Asynchronous CARI (ACARI)

The two versions of the CARI API are developed with the aim to increase programming productivity and develop potentially high-performance and efficient implementations. These two implementations require a C++ compiler and an MPI communications library. It is illustrated in a self-explanatory Figure 4.2.

4.4.1 Synchronous CARI (SCARI)

The first implementation is known as Synchronous CARI (SCARI), which is developed with the aim to exactly preserve the performance of the original Gadget-2 code. Mostly this implementation reorganizes the source code, and attempts not to incur any performance overhead. SCARI implements a variant of the `MPI_Alltoall` method with a bounded send and receive communication buffer.

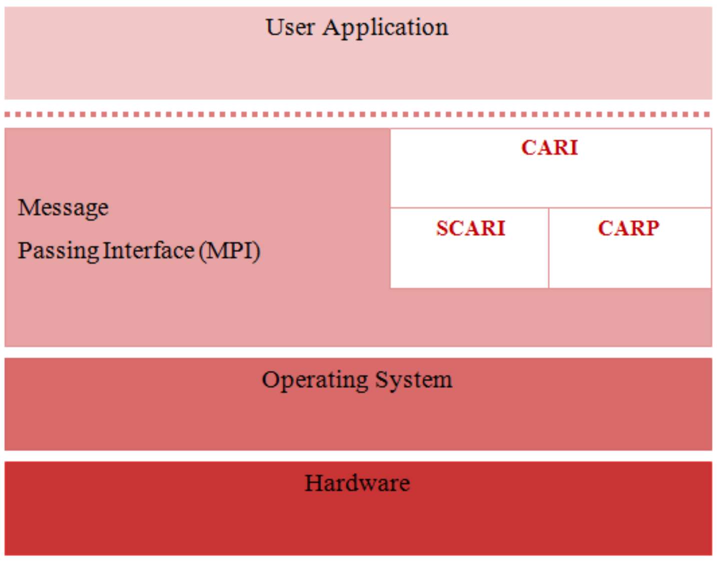


Figure 4.2: Architecture of CARI API.

Design and Architecture of SCARI

Figure 4.1 is somewhat stylized. Without comment, we have assumed an object-oriented C++ style of programming. The original Gadget-2 which is our motivating example and it is written in ANSI C. With few exceptions, C is a subset of C++, and it only needs a few minor adjustments to the Gadget-2 code to allow it to be compiled with a C++ compiler. Modern C++ compilers will compile programmes written in the C subset with no noticeable overhead. So there is no immediate downside to treating Gadget-2 as a C++ programme. We then only need to ensure that any new class libraries introduced into the programme do not negatively impact the overall performance of the system. In an object-oriented language it is most natural for the "handler" functions to be methods on objects. The classes of these objects can extend library-defined interfaces that specify the methods' signatures. Moreover the user-defined subclasses that implement them can conveniently be instantiated to hold references to relevant application state, like the P array.

A more realistic API for CARI is given in Figure 4.3. The constructor for `CARISchedule` is passed an MPI communicator, and an application-specific object whose class extends `CARIHandler`. The constructor also passed a workspace in the form of buffer space allocated in the application code. Internally, `CARISchedule` class will carve this workspace according to its requirements, for example, the implementation follows that of Figure 3.5, this workspace will be carved into a send buffer and a receive buffer, and perhaps extra workspace used in sorting the send buffer.

```
template <class S, class T>

class CARISchedule {

public:

  CARISchedule(MPI_Comm comm, CARIHandler <S, T> *hndlr,
               int buffSize, void *buffer);
  ~CARISchedule();
  void invoke(S *request, int dest, int tag);
  void complete(void);

private:
  void clearBuffer(void);
  ...
  ...
};


template < class S, class  T >

class CARIHandler {

public:
  virtual void handleRequest(S *request, T *response) = 0;
  virtual void handleResponse(T *response, int tag) = 0;
};
```

Figure 4.3: Realistic C++ API for CARI.

The `invoke` method takes a pointer to an instance of type S. This type S will typically be a primitive type, an array type of constant size, or a simple “struct-like” class containing only fields of like types. In C++ parlance, S must be a Plain Old Data (POD) type. No provision is made for arguments that are more general data structures. If the type S includes pointers, CARI will simply copy the address values of these pointers, and in general the copied values cannot be de-referenced in the context of a remote processor. This applies equally to `char*` pointers, where the strings that are to be processed remotely must be passed as char arrays of constant size.

The second argument of `invoke` method is a tag that is simply passed to the local `handleResponse` method when the result of the invocation returns. This tag can be used in any manner that is convenient to the application. Note this tag is deliberately not passed to the `handleResponse` method that handles the remote invocation on the “server” side. This is so that the API admits implementations that do not pass the tag in request or response messages. The result of the remote invocation is of type T, which again should be a POD type, and the same comments apply as for the argument type.

To simplify the task of the programmer, we developed the CARI API with certain guarantees of atomicity. Execution is serial in the sense that a `handleRequest` or `handleResponse` method is never called concurrently with execution of other user code on the local processor (and no two handlers are called concurrently on the local processor). Specifically, handlers are called sequentially (though in general in undefined order) during execution of CARI library code, either during an `invoke()` or `complete()` call. So, while the application writer needs to be aware that in general there is no uniquely defined order in which handlers are invoked, there is no need to synchronize access to variables used, for example, as accumulators. If, say, a handler performs an accumulation like:

$$x+=a$$

On a programme variable x, this is also modified in the main body of the client code or in another handler, so there is no need for explicit mutual exclusion protecting this operation.

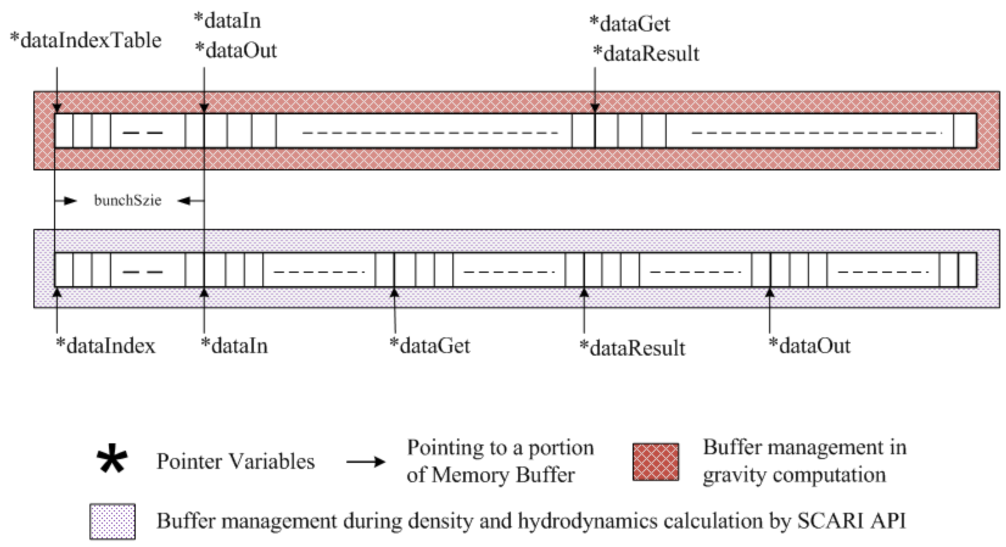

*dataIndexTable
*dataIn
*dataOut
*dataGet
*dataResult
bunchSzie
*dataIndex
*dataIn
*dataGet
*dataResult
*dataOut
* Pointer Variables
Pointing to a portion of Memory Buffer
Buffer management in gravity computation
Buffer management during density and hydrodynamics calculation by SCARI API

Figure 4.4: Communication Buffer used by SCARI

Figure 4.4 represents the usage of communication buffer by SCARI implementation. Given figure shows the communication buffer in two shades; the red shade buffer portrays how it is used by the `gravity, potential,` and `force` computation code. The blue shade buffer depicts how it is carved and used by the `density` and `hydra` computation code.

4.4.2 Asynchronous CARI (ACARI)

The second implementation, called Asynchronous CARI (ACARI), is a proof-of-concept implementation to demonstrate that-beyond simply refactoring the code to simplify it and improve its the maintainability, our higher-level libraries can potentially enhance performance, by admitting alternate implementations. In ACARI this is achieved by reducing wait time using non-blocking MPI primitives.

The motivation for ACARI comes from considering overheads of the the original Gadget-2 code, parts of which are refactored to give the SCARI implementation of CARI. This original implementation maintains a global list of numbers of particles exchanged at each communication stage during a single time-step. The size of this list is $O(P^2)$ where P is the total number of processors involved in the parallel computation. The data structures holding the particles of physical interest on each processor are of size $O(N/P)$ where N represents the total number of particles in the system.

As P increases, the overheads become comparable to the particle data when:

$$P^2 \propto N/P \tag{4.1}$$

or:

$$P^3 \propto N \tag{4.2}$$

or:

$$P \propto \sqrt[3]{N} \tag{4.3}$$

Similar considerations will apply to the associated computation and communication overheads, since they are broadly linear, or log linear, in the data sizes.

The intuition behind ACARI was that if we can eliminate the $O(P^2)$ data structures on each processor that record how much data all processors are exchanging, we may improve scalability. For example if overheads where only $O(P)$, P might be scaled up to $O(\sqrt{N})$ rather than $O(\sqrt[3]{N})$.

Collective Asynchronous Request Protocol (CARP)

CARP is a simple protocol in which each process can asynchronously send a request to any of its peers. The only collective aspect of the protocol is that a CARP “superstep” is terminated by a barrier synchronization. To get the implementation of CARP correct, we found it convenient to give an abstract representation in a CSP-like notation [28]. In terms of equations given in 4.2 , the abstract implementation of CARP on an individual process i is:

$$\text{CLIENT}(R(i), 0) \parallel \text{SERVER} \tag{4.4}$$

Here $R(i)$ is the sequence of requests issued by process i in the "superstep". At the level of abstraction here we don't consider the data content of the message-only the sequencing of events. So $R(i)$ is just a sequence of destination process ids (i.e. a sequence of numbers between 0 and $P-1$). There are four kinds of message exchanged by the protocol:

- req(i, j) is a request message from process i to process j
- ack(i, j) is a response message from process j back to process i
- bar(i, j) is a fan-in message from process i to process j in the first phase of a barrier synchronization
- rab(i, j) is a fan-out message from process j back to process i in the second phase of a barrier synchronization.

Superscripting the message names with O or I yields the CSP events corresponding to output or input of these message. The barrier synchronization is presumed performed on a spanning tree where the parent of process i is process p, and the sequence c contains process ids of the children of process i. Note $\square$ is the CSP choice operator.

Equation 4.4 essentially says that each process has two logical threads, the client thread that is sending out requests and get responses while the server thread is receiving requests and sending out responses. In the CSP model note that the only point of synchronization between these two threads is on the event rab$(i, p)^I$, which is the only event in the alphabet of both processes. For the server the receipt of the fan-out messages causes shut down.

We should add that this model does not actually describe the communication *between* processes, only the local behavior of the processes. With a lot more work it might be possible to model the interaction (and perhaps even formally prove correctness of the protocol). This wasn't our goal, we simply wanted a clean way to describe the local behavior of the processes engaged in the CARP protocol, which was intuitively correct. We believe it is reasonably clear from inspection that there is no scope for (say) deadlock in the model as given.

```
CLIENT$(r, n) =$
        (if $r \neq \langle\rangle$ then
          req$(i, r_0)^O \rightarrow$
            (if $r_0 \neq i$ then CLIENT$(n+1, r')$ else CLIENT$(n, r')$)
         else
          STOP)
         $\Box$
        (if $n > 0$ then
            $\Box_{j=0}^{P-1}$ ack$(i, j)^I \rightarrow$ CLIENT$(n-1, r)$
         else
          STOP)
         $\Box$
        (if $r = \langle\rangle \wedge n = 0$ then
          BARRIER$(\|c\|)$
         else
          STOP)
SERVER $=$
        ( $\Box_{j=0}^{P-1}$ req$(j, i)^I \rightarrow$ ack$(j, i)^O \rightarrow$ SERVER)
         $\Box$
        rab$(i, p)^I \rightarrow$ STOP
BARRIER$(m) =$
        if $m > 0$ then
            $\Box_{j \in c}$ bar$(j, i)^I \rightarrow$ BARRIER$(m-1)$
         else
          bar$(i, p)^O \rightarrow$ rab$(i, p)^I \rightarrow$ BARRIER$'(c)$
BARRIER$'(s) =$
        if $s \neq \langle\rangle$ then
          rab$(s_0, i)^O \rightarrow$ BARRIER$'(s')$
         else
          STOP
```

Figure 4.5: CSP model for CARP API

The next stage was to transcribe the abstract multithreaded model to an MPI implementation, single threaded on every processor. To achieve this we used MPI non-blocking communications with a simple finite state machine to implement the "background" server thread.

The final API for the CARP class is given in 4.6. The constructor is passed the maximum size of request and response messages, a working buffer, and a callback that defines how the server "thread" handles incoming request to transform them into response messages, and how the client "thread" deals with response messages. The Figure 4.7 represents different segments of the working buffer used by the CARP based CARI API. The only other two

```
class CARP{

public:
  CARP(long max_req_size,  long max_res_size,
       char *srBuffer, CARPHandler *handler);
  void request(int nsend, char *csBuffer, int dest);
  void complete();

private:
  void doServer (MPI_Status st);
  ...
};

class CARPHandler{

public:
  virtual int  handleRequest(char *, char *, int)=0;
  virtual void handleResponse(char *, int)=0;
};
```

Figure 4.6: A Sketch of CARP class

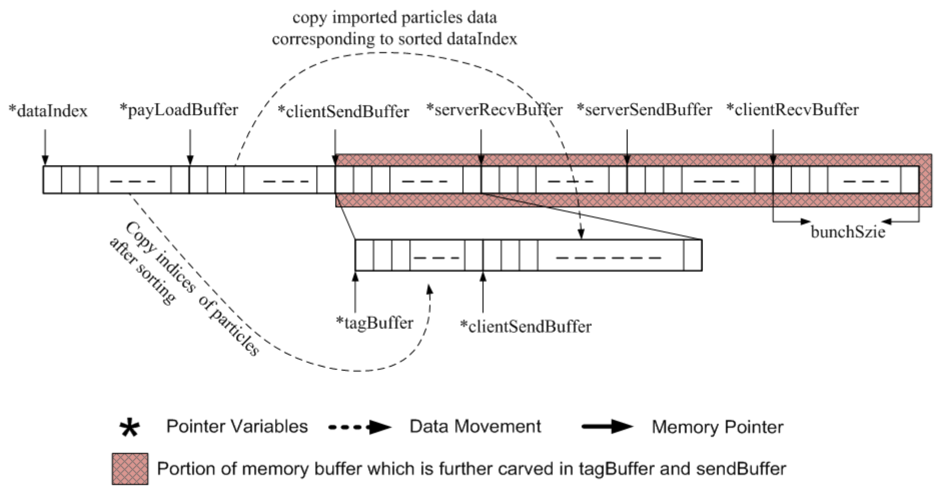

copy imported particles data corresponding to sorted dataIndex
*dataIndex
*payLoadBuffer
*clientSendBuffer
*serverRecvBuffer
*serverSendBuffer
*clientRecvBuffer
bunchSzie
Copy indices of particles after sorting
*tagBuffer
*clientSendBuffer
Pointer Variables
Data Movement
Memory Pointer
Portion of memory buffer which is further carved in tagBuffer and sendBuffer

Figure 4.7: Communication Buffer used by CARP API

methods are `request` and `complete`. The former naturally sends a message and the latter is called when the current "superstep" is over. Basically CARP itself is a newer lower level API that can be used independently from CARI API. We have very clear intentions to use CARP in our future proposed work. The ACARI version has been provided the same invoke and complete public functions (as for SCARI implementation). Figure 4.8 shows a sketch of ACARI implementation.

```
template <class S, class T>

class ACARISchedule {

public:
  ACARISchedule(MPI_Comm comm, int buffer_size,
                CARIHandler<S, T> *cari_handler,
                void *buff_ptr);
  ~ACARISchedule();
  void invoke(S *payload, int tag, int dest);
  void complete(void);

private:
  void clearBuffer(void){
   ...sort export buffer by destination...
   for(j=0; j< NTask; j++){
    if(nsend > 0){
      for(k=0; k< nsend; k++){
        ...copy tags to export buffer...
        ...copy particles to payloadBuffer...
      }
      ...call request method from CARP class,
         peers ready to export data...
      carp_protocol->request(nsend *
                              (sizeof(int)+ sizeof(S)),
                               clientSendBuffer, j);
    }
   }
  }
};
```

Figure 4.8: A Sketch of ACARI class

Chapter 5

Evaluation: Accuracy and Performance

5.1 Testing environment

The CARI API has been evaluated using a 32 processing core Linux cluster at SEECS-NUST. The cluster had eight compute nodes. Each node contained a quad-core Intel Xeon processor. The nodes were connected via Myrinet and Gigabit Ethernet. The compute nodes ran the SuSE Linux Enterprise Server (SLES) 10 operating system and GNU C Compiler (GCC) version 4.1.0. Each compute node had 2 Gigabytes of main memory. We used MPICH2 version 1.2.1p1 as the MPI library used by the original and the CARI Gadget-2.

5.2 CARI Test Suite

CARI's evaluation process comprised of two phases

1. Accuracy Evaluation
2. Performance Evaluation

5.3 Accuracy Evaluation

The public version of Gadget-2 code comes with initial condition files for different simulations including *Lambda Cold Dark Matter (LCDM) gas, Cosmological formation of a cluster of galaxies*, and *Galaxy collision*. Snapshots files are the primary output produced by GADGET-2 during simulation, which are simply dumps of state of the system at certain time intervals. We have successfully incorporated CARI API in Gadget-2 code at five different stages (`compute_potential, gravity_tree, gravity_forcetest, density, and hydro_force`) of the code as discussed earlier in Chapter 4. A comprehensive accuracy evaluation was done by running various simulations (described above) using CARI version Gadget-2 code and the results were compared against the original code's results.

One way to quantify accuracy of CARI version Gadget-2 code was by comparing visual results. `IDL` software was used to create and analyze visualizations from multifaceted numerical data. The Gadget-2 distribution include `IDL` scripts to plot the system from snapshot files. These visual outputs were indistinguishable for the two versions. This provide us a high degree of confidence for correctness of the newly developed API code. We did not rely only on the visual comparison; thus we produced more transparent results by generating human readable output from binary snapshot files for both said versions. Finally, these output files were compared using auto testing modules which we developed specially for the testing purpose.

5.3.1 Runtime environment for Gadget-2 code

Gadget-2 *Makefile* lists a number of compile-time flags to configure a simulation that can be run with a GADGET-2 executable. For example `PERIODIC` flag should be enabled for *Lambda Cold Dark Matter (LCDM) gas* simulation. Similarly there are other flags which enable/disable execution of some specific part of the code. For example if `OUTPUTPOTENTIAL` flag is enabled, it will force the code to compute and store gravitational potential for all particles each time a snapshot file is generated. Other important flags are listed in Table 3.1 in Chapter 3.

5.3.2 Auto Testing

We prefer auto-testing rather than manual because of transparency of results along with efficiency of testing process. Testing is a cyclic process, that's why every time developer changes the source code, we executed the auto-testing modules to verify the correctness of the change.

1. `ReadSnapshot` module was used for producing human readable output from Gadget-2 binary snapshot files.

2. `Testing` module was used to compare these human readable output and to produce a comparison report.

Both of these auto-testing modules require a text file as command line argument, which was named as `testparameters` containing the following information:

- Total number of particles (for how many particles the test should be conducted, 800 particles in our case).
- Threshold value called epsilon (used to ignore insignificant rounding error).
- Randomly selected particle's ID's.

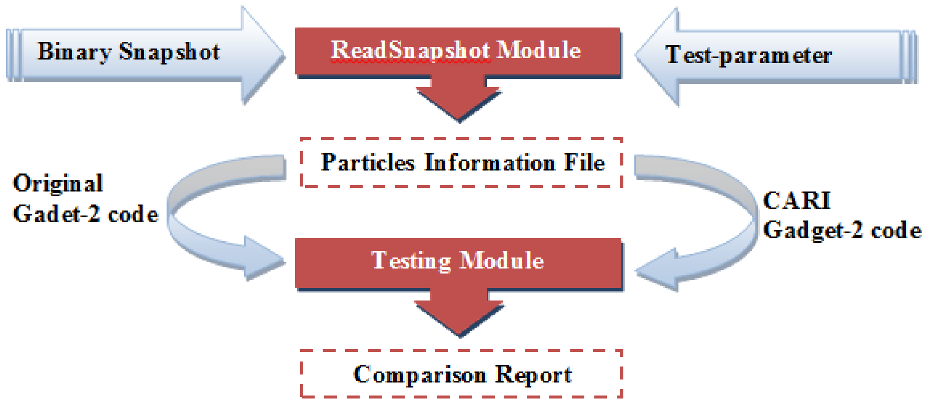


Figure 5.1: Auto Testing Procedure

The importance of insignificant rounding errors, epsilon value, and the details how we have tackled them during the development and evaluation process of CARI API are discussed in the next section.

5.3.3 Intricacies of Floating Point Arithmetic

On a computer, real numbers are always approximated by floating point numbers. Mathematical operations on big real numbers having large fractional part regularly lead out of the space of the represent-able numbers which frequently results in round-off errors. That why, for these mathematical operations the law of associativity for simple additions/subtraction doesn't hold on a computer.

$$(\mathtt{A} + \mathtt{B}) + \mathtt{C} \neq \mathtt{A} + (\mathtt{B} + \mathtt{C})$$

This is what we have observed during the development process of CARI API. We have also observed that even a small change in the order of floating point operations (may) change the expected results; again this is because of the limited precision of computing machines. So comparing the results against some expected value it is highly unlikely that one will get expected results. The solution to minimize the effects of this problem was to use *Relative Error* rather than *Absolute Error*.

Absolute Error

The absolute error can be calculated by taking the magnitude of the difference between the exact value and the approximation.

Relative Error

The relative error is the absolute error divided by the magnitude of the exact value.

CARI-testing process is also based on relative error instead of absolute error. As an example if the exact value is 100 and the approximation is 99.9, then the absolute error is 0.1 and the relative error is 0.1/100 = 0.001. Similarly while developing the auto-testing modules we avoid the use of equality test `(if (A == B) then )` when expectations are based on floating point numbers. These tests are replaced like this:

$$\texttt{If}(\texttt{abs}(\texttt{A} - \texttt{B}) < \texttt{epsilon})\ \texttt{then do} - \texttt{this endif}$$

Here epsilon was sufficiently so small. One can set its values according to the accuracy desired or an application required. For example if an application desired 99% accuracy it means 0.01 is the the tolerance rate. So we need to set epsilon value to 0.01. It means the relative error less than epsilon are insignificant and it can be ignored. CARI API has been tested using various epsilon values by hit and trial method (ranging from 0.000001 maximum up to 0.000020).

5.3.4 Cosmological Simulations used for Testing

This section highlights the various scientific simulations used for CARI API's evaluation process. All the figures given in this section are obtained using CARI version Gadget-2 code. To save space and avoid redundancy, figures obtained using original Gadget-2 code are not pasted here.

Lambda Cold Dark Matter (LCDM) Gas

LCDM Gas simulation often referred to as the typical model of big bang cosmology. It helps to understand the existence and structure of the cosmic microwave background, the large scale structure of galaxy clusters and the distribution of different gases (hydrogen, helium, lithium, oxygen). Initial condition file available with Gadget-2 code contains 65365 number of particles. Table 5.1 list the necessary runtime flags that were enabled while running this simulation along with other default enabled flags.

`OUTPUTPOTENTIAL`	This will force the code to compute gravitational potentials for all particles each time a snapshot file is generated.
`OUTPUTACCELERATION`	Physical acceleration of each particle will be stored in the snapshot files if this flag is enabled.
`FORCETEST=`0.01	This can be set to check the force accuracy of the code, and is only included as a debugging option. The normal tree-forces and the exact direct summation forces are then collected in a file forcetest.txt for later inspection. Note that the simulation itself is unaffected by this option.

Table 5.1: Compile-Time Flags For LCDM Gas Simulation

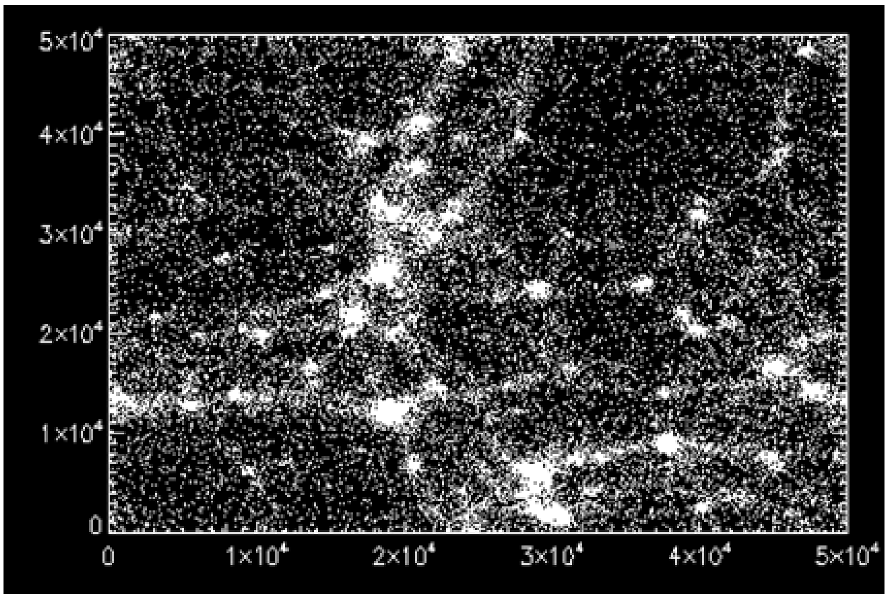


Figure 5.2: Initial State of LCDM Gas Simulation

Colliding Galaxies Simulation

This simulation runs two disk galaxies into each other, leading to a merger between the galaxies. In this example each galaxy consists of 60, 000 particles. After colliding each galaxy is distributed into a stellar disc and an extended dark matter halo. Figure 5.3 shows the initial state of the Colliding Galaxies simulation. Figure 5.4 and Figure 5.5 show evolved states. Each figure shows the Colliding Galaxies from three different angles.

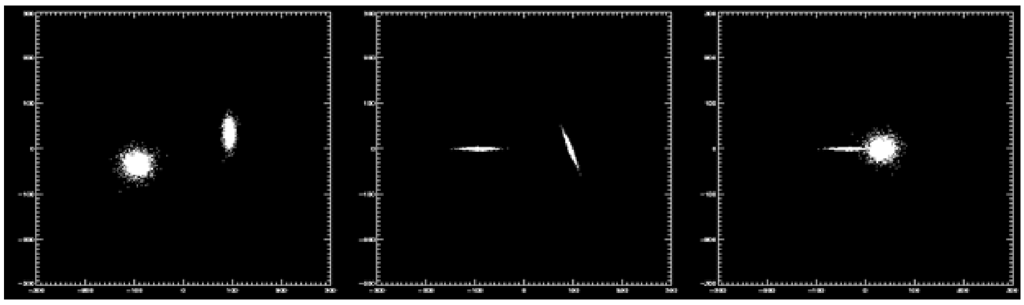


Figure 5.3: Initial State of Galaxy Formation Simulation

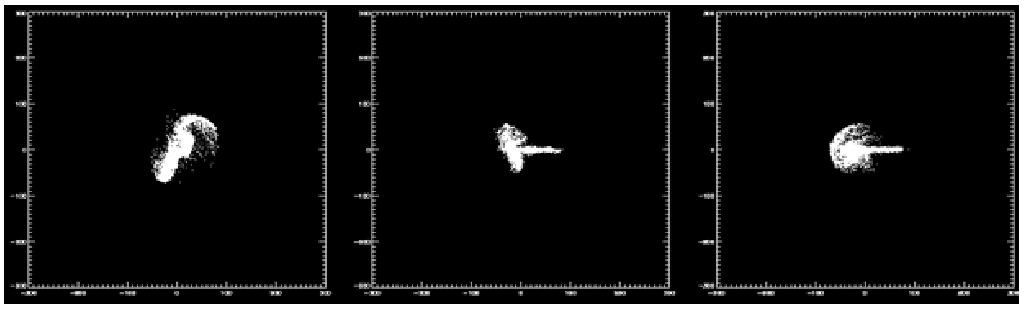


Figure 5.4: Mid State of Galaxy Formation Simulation

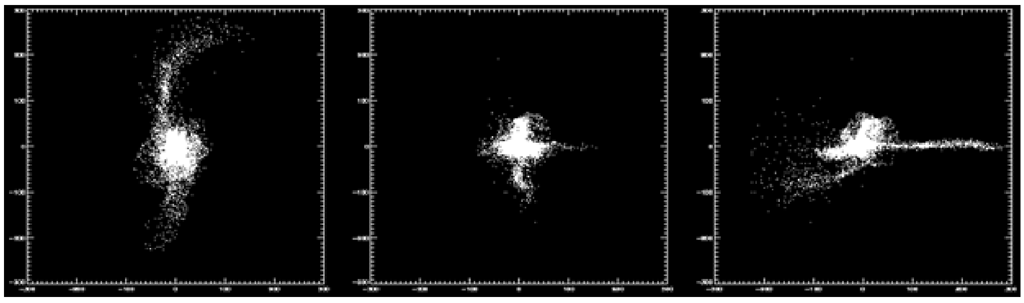


Figure 5.5: Final State of Galaxy Formation Simulation

Cosmological formation of a Cluster of Galaxies

This simulation evolves randomly distributed dark matter 276498 particles to form a cluster of galaxies using collisionless dynamics in an expanding universe.

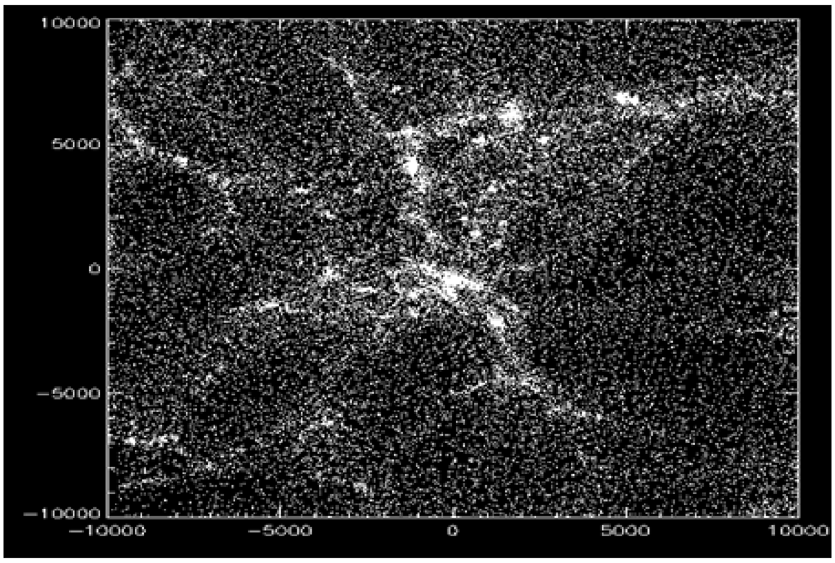


Figure 5.6: Initial State of Cluster Formation Simulation

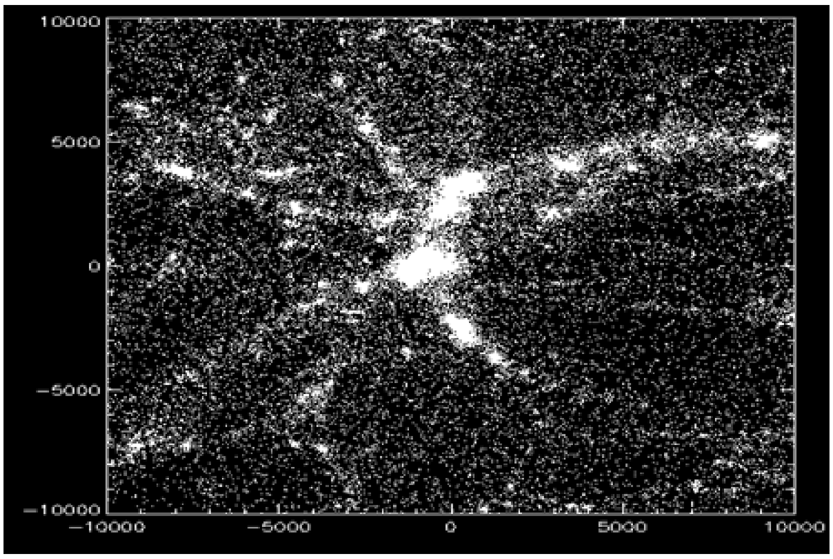


Figure 5.7: Mid State of Cluster Formation Simulation

Cluster simulation's visual results show that the CARI version Gadget-2 code performs as accurate as original version. Figure 5.6 shows the initial state of the Cluster Formation simulation followed by Figure 5.7 which shows an evolved state. Figure 5.8 shows the final state of the Cluster Formation simulation. Figure 5.9 is special in the sense that it is drawn using combined snapshot files (initial, evolved and final state)

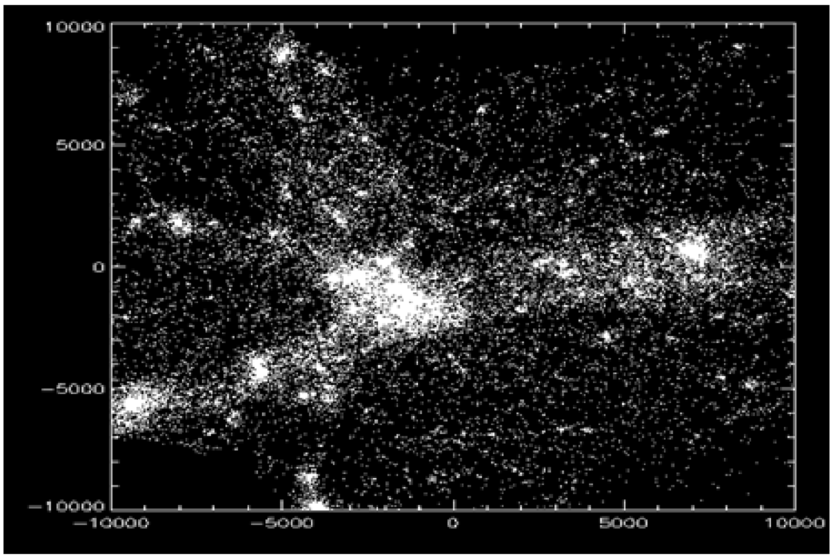


Figure 5.8: Final State of Cluster Formation Simulation

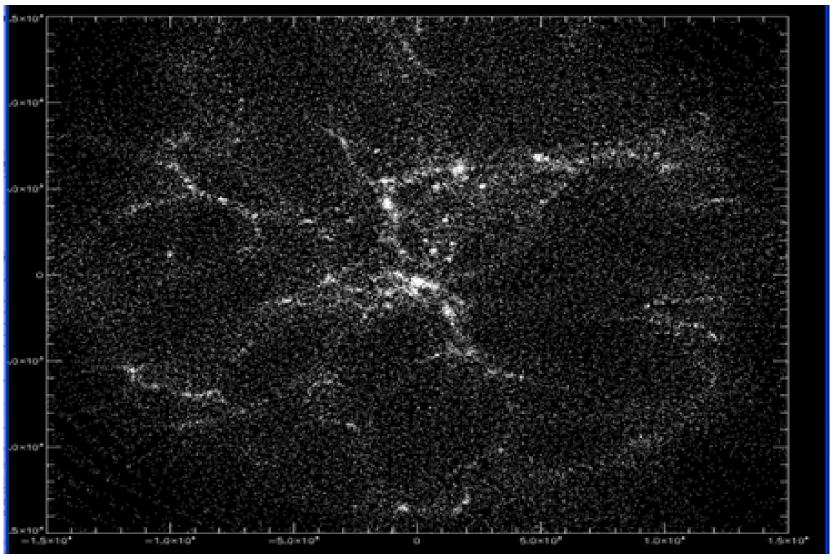


Figure 5.9: Combined Draw for Cluster Formation Simulation

5.4 Performance Evaluation

To test the performance of CARI variants of the Gadget-2 code, we used the Cluster Formation Simulation that comes with the source bundle. This simulation contains a total of 276498 particles and can thus be considered a moderate size simulation.

A relevant issue is the mapping of MPI processes to cluster compute nodes. When the total number of MPI processes is less than or equal to 8 (total number of compute nodes), we run each MPI process on a distinct compute node. As the total number of processes increases the total count of compute nodes, additional MPI processes are distributed evenly on compute nodes in a round-robin fashion. For example in the case of 16 MPI processes, each node runs 2 processes and in the case of 32 processes, each node runs 4 processes.

As part of the performance evaluation effort, we present the execution time and the speedup graph in Figure 5.10 and Figure 5.11 respectively for three variants of the Gadget-2 code. These include the original Gadget-2, Gadget-2 using SCARI, and Gadget-2 using CARP. The Gadget-2 code in the second and third case is exactly the same; the only difference is the implementation of the CARI library. The original Gadget-2 code implements a profiling scheme at the application layer to measure the time spent at various stages including overall execution time, tree construction, tree walk, communication, domain decomposition, I/O, and synchronization. For brevity purposes, we only present the overall execution time, which also is the most relevant here.

Our performance evaluation reveals that both CARI versions of the Gadget-2 code perform, at least, as well as the original Gadget-2 code. This meets one of our main objectives that the performance of CARI-based application must be comparable to the original application code that uses MPI. The performance of the SCARI implementation is exactly similar to the original code as we expected. But the CARP implementation of the CARI API slightly outperforms the SCARI-based and the original application code for all processor counts. In fact the performance of CARP gets even better as the numbers of cores are added to the parallel computation. Although the gain observed by CARP is modest, but it proves an important point that higher-level communication libraries can also provide better performance than MPI alone. The main reason for the better performance of CARP is the asynchronous nature of the communication algorithm. Also the CARP implementation does not rely on the global knowledge of elements transferred at each time step. This global information is maintained using an expensive MPI collective call for several communication stages within a single time step. Due to

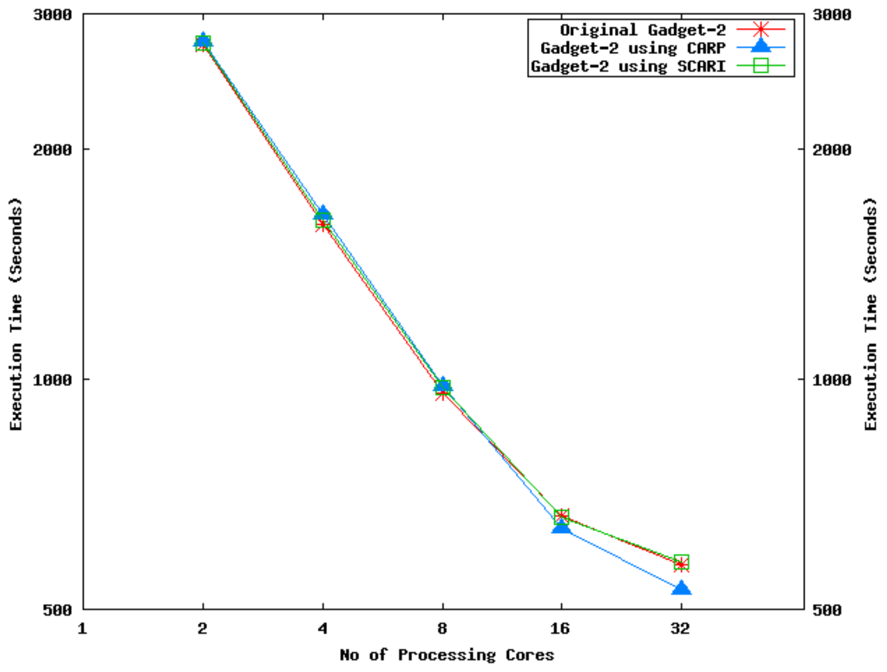


Figure 5.10: Execution-Time for Cluster Formation Simulation

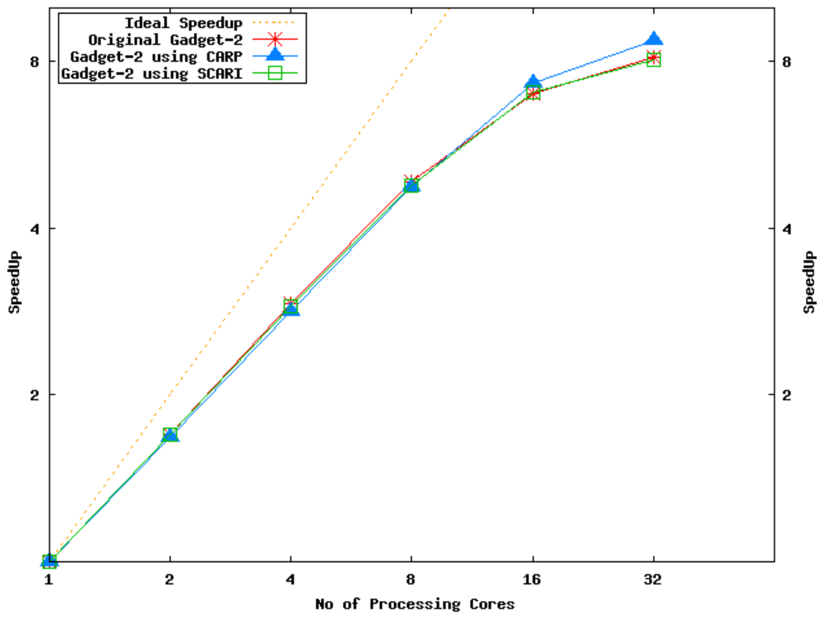


Figure 5.11: Speed-up Measure for Cluster Formation Simulation

its asynchronous implementation using non-blocking MPI calls, CARP removes this bottleneck from the code. In addition, CARP heavily relies on the quality of the implementation of non-blocking communication functions and the associated asynchronous progress engine. To recap, the performance numbers presented in this section clearly demonstrate that SCARI-based implementation of Gadget-2 performs as well as the original code. Our second implementation CARP, which is a proof-of-concept implementation, modestly outperforms the original code.

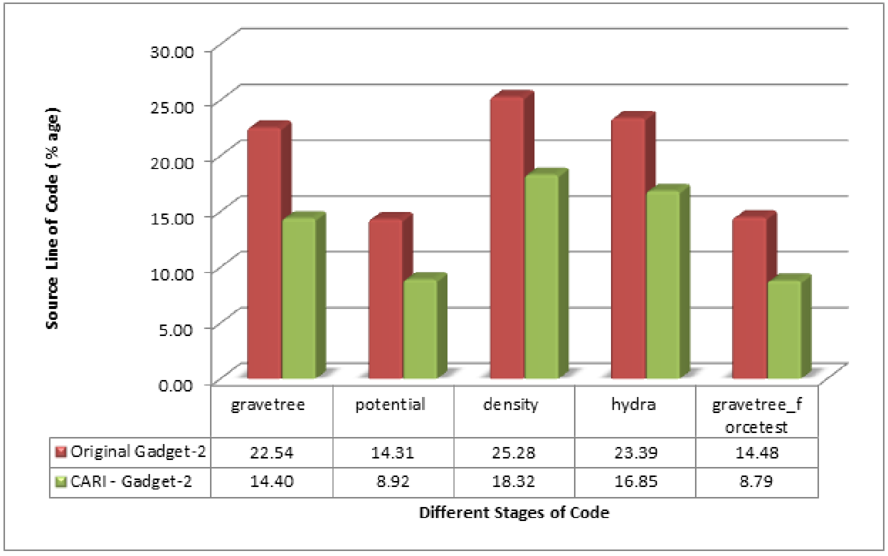
5.5 Productivity - Line of Code (LOC)

By absorbing the message passing code, we have simplified the application (Gadget-2) code. Table 5.2 verifies this by presenting Source Lines of Code (SLOC) comparison between Gadget-2 version using MPI and CARI. There is a notable reduction in SLOC in the case of the gravity calculation physics code. Figure 5.12 describes the usefulness of CARI API at different parts of the Gadget-2 code. For example gravity computation code is almost 22.5% of the overall application when it was using MPI, which involves complex and recurring message passing code along with physics code. But using CARI library this number reduce to 14.5% of the overall, which indeed 8% less than the original. It really means that an application developer will now put 8% less efforts while writing application the code, similarly in the other parts of the code.

	gravetree	potential	density	hydra	gravetree_forcetest
CARI-Gadget-2	14.40	8.92	18.32	16.85	8.79
Original Gadget-2	22.54	14.31	25.28	23.39	14.48

Table 5.2: Productivity In Terms Of LOC Using CARI API

It must be noted that our proposed API (CARI) is applicable to less than 15% of the total code and thus the SLOC reduction is less visible at the application level. But for the relevant computation/communication sections of the code (like `gravity/density` calculations), the CARI API significantly reduces SLOC (mostly dense and complicated MPI code) and improves maintainability of the code.



	gravetree	potential	density	hydra	gravetree_forcetest
🟥 Original Gadget-2	22.54	14.31	25.28	23.39	14.48
🟩 CARI - Gadget-2	14.40	8.92	18.32	16.85	8.79

Figure 5.12: Line of Code Comparison

5.6 Memory Usage - Scalability

As mentioned in subsection 3.4 the Gadget-2 code maintains a global list of all particles exchanged at each communication stage during a single time step. The size of this list is P^2 where P is the total number of processors involved in the parallel computation. One of the CARI implementation (ACARI) avoids keeping and maintaining this global list, which dominates the total memory consumed by the Gadget-2 code as the value of P (number of processors) increases.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

The emergence of multicore hardware has put the burden of performance on the software development community. The only option to increase performance of existing sequential applications is to utilize some form of parallelism. This obviously implies that the software developers must learn parallel programming models and tools to write optimized code for multicore processors.

Gadget-2 is a massively parallel astrophysics code for cosmological N-body and hydrodynamics simulations. Versions of this code featured in the Millennium Simulation, which is the largest simulated model of the Universe. Gadget-2 has been parallelized for distributed memory platforms using the MPI standard, which is considered a low-level and complex API. In this thesis we analyzed Gadget-2 with a view to understanding what high-level SPMD communication abstractions might be developed to replace the intricate use of MPI in Gadget-2. These communication abstractions must enhance programming productivity without compromising performance. Our analysis of the code identified several complex and recurring patterns of message passing cluttered inside the application specific physics code.

We reorganized Gadget-2 to absorb one such message passing schedule into a high level communication library called CARI - a collective variant of RMI. By absorbing the message passing code, we have simplified the application code. We had verifies this by presenting Source Lines of Code (SLOC) comparison between Gadget-2 version using MPI and CARI. There is a notable reduction in SLOC in different phases of the physics code. Right now our newly developed API (CARI) is applicable to less than 15% of the total application (Gadget-2) code and thus the SLOC reduction is less visible at the application level. But for the relevant computation/communication sections of the code, the CARI API significantly reduces SLOC (mostly dense and complicated MPI code) and improves maintainability of the code.

The thesis introduced and evaluated implementations of the CARI schedule. By construction, these schedules have efficient implementations on various architectures including distributed and shared memory machines. We prove this point by presenting two implementations in this thesis.

The first implementation, known as SCARI, retains the original message passing code with the goal of obtaining similar application level performance. This implementation simply reorganizes the code. The second implementation, known as CARP, is developed to show that it is possible to improve performance and programming productivity at the same time. CARP uses an asynchronous request processing protocol that extensively relies on non-blocking MPI communication methods.

The performance evaluation revealed that SCARI-based implementation of the Gadget-2 code achieves comparable performance to the original code. However, the CARI-based Gadget-2 modestly outperforms other implementations. This clearly demonstrates that higher performance and higher productivity and not mutually exclusive.

6.2 Future Work

There are couple of other message passing patterns in Gadget-2 that can also be absorbed into a high-level communication library. The two notable patterns include a) the distributed sort based on the Peano-Hilbert key, and b) parts of the TreePM algorithm. We plan to address these in the future. We also plan to demonstrate usefulness of the CARI library in other irregular applications-the most likely candidate is a Finite Element Method (FEM) code. The CARI library will be released as open-source software towards the end of 2010.

Bibliography

[1] H. Sutter, "The Free Lunch is Over: A Fundamental Turn Towards Concurrency in Software," [Online]. Available: `http://www.gotw.ca/publications/concurrency-ddj.htm`

[2] D. Geer, "Industry Trends: Chip Makers Turn to Multicore Processors," *Computer*, vol. 38, no. 5, pp. 11-13, doi:10.1109/MC.2005.160, May 2005.

[3] "The Manycore Shift: Microsoft Parallel Computing Initiative Ushers Computing into the Next Era," *Microsoft*, http://www.microsoft.com/downloads/.

[4] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard*, University of Tenessee, Knoxville, TN, 1995, www.mcs.anl.gov/mpi.

[5] Y. Hoskote, S. Vangal, A. Singh, N. Borkar, and S. Borkar, "A 5-ghz mesh interconnect for a teraflops processor," *IEEE Micro*, vol. 27, no. 5, pp. 51–61, 2007.

[6] V. Springel, "The cosmological simulation code GADGET-2," *Monthly Notices of the Royal Astronomical Society*, vol. 364, p. 1105, 2005. [Online]. Available: `http://www.citebase.org/cgi-bin/citations?id=oai:arXiv.org:astro-ph/0505010`

[7] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from berkeley," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, Dec 2006. [Online]. Available: `http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html`

[8] J. L. Träff, “What the parallel-processing community has (failed) to offer the multi/many-core generation,” *J. Parallel Distrib. Comput.*, vol. 69, no. 9, pp. 807–812, 2009.

[9] L. Kale, “New parallel programming abstractions and the role of compilers,” *Parallel and Distributed Processing Symposium, International*, vol. 0, pp. 458, 2006.

[10] “Top 500 List Home Page,” www.top500.org/list/2010.

[11] D. Turner, “Introduction to Parallel Computing and Cluster Computers,” *Ames Laboratory*, http://www.scl.ameslab.gov/Projects/

[12] C. Quammen, “Introduction to programming shared-memory and distributed-memory parallel computers ISSN:1528-4972,” *ACM Crossroads*, vol. 8, pp. 16 - 22, 2002.

[13] G. A. Geist, J. A. Kohl, P. M. Papadopoulos, “PVM and MPI: a comparison of features. Tech. Report DE-AC05-96OR22464,” *U.S. Department of Energy, Office of Energy Research*, May 30 1996.

[14] “OpenMP: Simple, Portable, Scalable SMP Programming,” http://www.openmp.org.

[15] “Intel Cilk++ Software Development Kit,” http://software.intel.com/en-us/articles/intel-cilk/.

[16] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multicore Processor Parallelism.* O’Reilly Media, 2007.

[17] G. Z. Bryan, B. Carpenter, G. Fox, X. Li, and Y. Wen, “A high level spmd programming model: Hpspmd and its java language binding,” In International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA’98, Tech. Rep., 1998.

[18] S. Lim, B. Carpenter, G. Fox, and H.-K. Lee, “A device level communication library for the HPJava programming language,” in *IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2003)*, November 2003.

[19] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Aprà, “Advances, applications and performance of the global arrays shared memory programming toolkit,” *Int. J. High Perform. Comput. Appl.*, vol. 20, no. 2, pp. 203–231, 2006.

[20] L. V. Kale and S. Krishnan, "Charm++: a portable concurrent object oriented system based on c++," *SIGPLAN Not.*, vol. 28, no. 10, pp. 91–108, 1993.

[21] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[22] R. W. Numrich and J. Reid, "Co-array fortran for parallel programming," *SIGPLAN Fortran Forum*, vol. 17, no. 2, pp. 1–31, 1998.

[23] T. El-Ghazawi, W. Carlson, T. Sterling, and K. atherine Yelick, *UPC: Distributed Shared Memory Programming.* John Wiley and Sons, May 2005.

[24] V. Springel, S. White, A. Jenkins, C. Frenk, N. Yoshida, L. Gao, J. Navarro, R. Thacker, D. Croton, J. Helly, J. Peacock, S. Cole, P. Thomas, H. Couchman, A. Evrard, J. Colberg, and F. Pearce, " Simulating the joint evolution of quasars, galaxies and their large-scale distribution," *Nature*, pp. 435–629, 2005.

[25] J. Barnes and P. Hut, " A Hierarchical O(N log N) Force-calculation Algorithm," *Nature*, vol. 324, no. 4, pp. 446–449, 1986.

[26] "FFTW Project Home Page," http://www.fftw.org/.

[27] J. Nieplocha, R. Harrison, and R. Littlefield, "The Global Array: Non-uniform-memory-access programming model for high-performance computers," *The Journal of Supercomputing*, vol. 10, pp. 197–220, 1996, http://www.emsl.pnl.gov:2080/docs/global/.

[28] C. A. R. Hoare, *Communicating Sequential Processes*, Prentice Hall, 1985.