Available online at www.sciencedirect.com
ScienceDirect
Procedia Computer Science 4 (2011) 26–35
Procedia Computer Science

International Conference on Computational Science, ICCS 2011

Collective Asynchronous Remote Invocation (CARI): A High-Level and Efficient Communication API for Irregular Applications

Wakeel Ahmad[a], Bryan Carpenter[b], Aamir Shafi[a,c]

[a]*School of Electrical Engineering and Computer Science, National University of Sciences and Technology, Islamabad 44000, Pakistan*
[b]*School of Computing, University of Portsmouth PO1 2UP, UK*
[c]*Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge MA 02139, USA*

Abstract

The Message Passing Interface (MPI) standard continues to dominate the landscape of parallel computing as the *de facto* API for writing large-scale scientific applications. But the critics argue that it is a low-level API and harder to practice than shared memory approaches. This paper addresses the issue of programming productivity by proposing a high-level, easy-to-use, and efficient programming API that hides and segregates complex low-level message passing code from the application specific code. Our proposed API is inspired by communication patterns found in Gadget-2, which is an MPI-based parallel production code for cosmological N-body and hydrodynamic simulations. In this paper—we analyze Gadget-2 with a view to understanding what high-level Single Program Multiple Data (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. 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. We present 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 develop 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 well as the original Gadget-2 code but the level of abstraction is raised considerably.

Keywords: SPMD Communication, Programming Productivity, CARI, Asynchronous CARI, Synchronous CARI

Email addresses: `wakeel.ahmad@seecs.edu.pk` (Wakeel Ahmad), `bryan.carpenter@port.ac.uk` (Bryan Carpenter), `ashafi@mit.edu` (Aamir Shafi)

1877–0509 © 2011 Published by Elsevier Ltd. Selection and/or peer-review under responsibility of Prof. Mitsuhisa Sato and Prof. Satoshi Matsuoka
doi:10.1016/j.procs.2011.04.004

1. Introduction

The current generation of multicore processors are typically shared memory devices. 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 [1]. In the light of this, the Single Program 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.

The Message Passing Interface (MPI) standard [2]—that implements the SPMD model—continues to dominate the landscape of parallel computing as the *de facto* standard for writing parallel applications. But the critics argue that it is a low-level API and harder to practice than shared memory programming approaches. We argue that if SPMD is to become a mainstream model to address the multicore challenge, the level of abstraction provided by SPMD libraries and languages must be "raised".

In this paper, we address the issue of programming productivity by proposing a high-level, easy-to-use, and efficient programming API for multi/many-core processors. We use this API to replace the use of MPI in a production parallel N-body code called Gadget-2 [3]. 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.

For the purpose of increasing programming productivity, we begin by analyzing 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 an *irregular* application—and do so without compromising the efficiency. Our 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—patterns 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 remains at least as efficient as the original code. But the level of abstraction is raised considerably.

This paper presents the design, implementation, and evaluation of one such message passing schedule that we term Collective Asynchronous Remote Invocation (CARI). 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. The CARI API might be implemented in several ways—we develop two versions of this API as part of this paper. We later modify the Gadget-2 code to use the CARI library instead of MPI. We found out that the original code can be dramatically simplified. Also the performance—in terms of execution time and memory footprint—can also be improved. We observe that versions of Gadget-2 using CARI perform, at least, as well as the original Gadget-2 version.

Rest of the paper is organized as follows. Section 2 discusses relevant literature. This is followed by an introduction of the CARI API using a simple toy application in Section 3. We introduce the Gadget-2 code in Section 4. This section also presents details on using CARI library in the Gadget-2 code. We evaluate and compare performance of the original Gadget-2 with CARI versions in Section 5. We conclude and present future work in Section 6.

2. Related Work

The two most popular parallel computing platforms include the shared memory and distributed memory hardware. For the shared memory hardware, programming options include using threads/locks or task parallel approaches like OpenMP [4], Intel Cilk++ Software Development Kit (SDK) [5][6] and Threading Building Blocks [7]. For the distributed memory hardware, the preferred programming approach may be the SPMD programming model by using message passing libraries like MPI. More recently, the SPMD programming model is also emerging as a viable option for the future multi/many-core processors.

In earlier work we have defined a programming model for distributed memory machines that we called the HPspmd model [8]. In this approach 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 [9] library provides 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 paper we extend this approach by providing communication schedules for irregular applications.

Our approach has similarities to libraries that provide higher-level abstractions including the Global Arrays Toolkit [10]. This toolkit provides an abstract representation of distributed arrays, coupled with a family of `get` and `put` operations for accessing the elements of these arrays. Our approach is distinguished by the clear factorization 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.

There is a superficial analogy of CARI to Active Messages (though Active Messages is not normally considered a high-level API). 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. We are not aware of existing high-level APIs that exactly provide these features.

Some other library-based approaches are based explicitly on the BSP model [11] 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 [12] and Unified Parallel C (UPC) [13] 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.

Another language that attempts to increase programming productivity without compromising performance is the Charm++ [14] system—a set of extensions to the C++ language. It is a comprehensive system for parallel Object Oriented (OO) programming. Charm++ has some features in common with the CARI API—notably the asynchronous communication calls. But the CHARM++ system does not support the concept of collective completions, which is an important requirement in our pipelined system of invocations. Moreover, it is a much more complicated system that includes advanced features (like object marshalling), which are not part of our simpler and less ambitious API.

3. Collective Asynchronous Remote Invocation

We begin this section by presenting a simple example to motivate the need for programmer friendly API like CARI. The presentation of this example is followed by an introduction to the CARI API.

3.1. A Toy Example

Imagine a graph $G(V, E)$ where V is the set of all vertices, and E is the set of all edges between these vertices. The problem is to find the sum of weights of neighbours for all vertices. The sum of neighbours for each vertex is $\sum_{i=1}^{n} weight(i)$, where n is the total number of neighbours for a particular vertex. A sequential algorithm to solve this simple problem is:

```
foreach vertex in vertices {
        foreach neighbour in vertex.neighbours {
                vertex.neighbours_sum += vertices[neighbour].weight ;
        }
}
```

```
for( ... each vertex in local vertices ... ) {
  for( ... each neighbour in vertex.neighbours ... ) {
    if( ... neighbour.owner_process == me... ) {
      vertex.neighbours_sum +=
              vertices[neighbour.index].weight;
    } else {
         ... Add vertex index to send buffer ...
    }
  }
}
 ...Sort send buffer by destination...
for( ... each peer in peers ... ) {
  ...Send indexes to peer; and
           recv indexes for local computation...
}
... Compute local contribution to received indexes ...
for( ... each peer in peers ... ) {
 ...Recv weights from peer...
 vertex.neighbours_sum += received_weight;
}
```

(a) MPI version

```
CARISchedule <int, float>sched;
for( ... each vertex in vertices ... ) {
  for( ... each neighbour in vertex.neighbours ... ) {
    if( neighbour.owner_process == me ) {
      vertex.neighbours_sum +=
              vertices[neighbour.index].weight;
    } else {
        sched.invoke(vertex.index,
            neighbour.owner_process, neighbour.index);
    }
  }
}
sched.complete() ;

void handleRequest(int *index, float *request){
      *result = vertices[*index].weight;
}
void handleResponse(float *result, int index){
      vertices[index].neighbours_sum += *result ;
}
```

(b) CARI version

Figure 1: Two Parallel Pseudocodes for the Toy Example

In this pseudocode, the `vertices` array stores all vertices in the graph. Each `vertex` element has a `neighbours` array that stores indices of the neighbouring vertex.

Now we turn our attention towards a parallel implementation. Assuming a simple domain decomposition strategy, vertices of the graph can be equally divided amongst all parallel processes. Each process can now compute the sum of weights of locally owned neighbours in parallel. But it is possible that neighbour(s) of a vertex *i*, on a particular process, might be owned by a remote process. In such a scenario this process must engage in communication of weights of remotely owned vertices and then perform the sum computation. This kind of irregular communication frequently occurs in the Gadget-2 that is discussed in Section 4. The main purpose of the CARI library is to abstract out the low-level message passing logic from the application code. As an evidence of this argument, we now discuss and compare parallel versions—presented in Figure 1—implemented using MPI and CARI APIs.

We first discuss the MPI version shown in the Figure 1(a). In this pseudocode, each process first iterates over its `local_vertices` to compute the sum of weights of locally held neighbours. If a neighbour is remotely owned, the neighbours vertex index is stored in the `send_buffer`. In the first communication loop, each process sends accumulated indexes of foreign neighbours to remote processes. In return, each process receives indexes of its vertices that are connected to remote vertices. This is followed by application specific code where each process computes weights for incoming indexes. After this computation, another communication phase follows in which each process receives weights from peers and also sends locally computed weights. The received remote weights are added back to the partially calculated sums—this is again application specific code. Note that the three lines (in non-italicized bold font) highlighted in red represent the application specific code.

The parallel version can be reorganized—as given in the Figure 1(b)—using the CAPI API. The code first creates an instance of the `CARISchedule` called `sched`. This object implements the recurring messaging pattern discovered in the MPI parallel code. Each process iterates over its `local_vertices` and computes sum of the locally held neighbours. If a neighbour is owned by a remote processor, the CARI version calls the `invoke()` method on the CARI schedule object. The communication loop is followed by a call to the `complete()` method on the `sched` object.

In addition this version also defines two *callbacks*. The `handleRequest` handler implements the application specific code, which is executed when a "server" receives a request from a "client". On the other hand, the `handleResponse` callback is invoked when the "client" receives response back from the "server". Unlike the MPI parallelized version, here the application code is neatly abstracted out of the message passing code.

There are multiple implementation choices for `invoke()` and `complete()` methods. We outline one possible implementation here. If the implementation of `CARI` corresponded to Figure 1(a), the `invoke` method would simply add vertex index to the send buffer assuming that the buffer has sufficient space to store all indexes. The `complete`

```
template <class S, class T>
class CARISchedule {
public:
        CARISchedule(MPI_Comm comm, CARIHandler<S, T> *hndlr, int buffSize, void *buffer);
        ~CARISchedule();
        void invoke(int tag, int dest, S *request);
        void complete(void);
private:
        ...
};

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 2: A realistic C++ API for CARI

method would execute the code in Figure 1(a) following the "main loop", calling the user-defined callbacks at appropriate points. We will discuss other possible implementations of `CARI` in the Section 4.4.

The pattern in Figure 1(b) may be recognized as a basic kind of remote method invocation. The user-defined callback `handleRequest` 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`.

The invocation is 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 `handleResponse` processes the result locally when it does eventually return to the "client". The pattern is "collective" in a couple of senses. All peers potentially act symmetrically as clients and as servers. Creation of the `CARI` schedule is a collective operation. The logical synchronization that marks completion of all invocations occurs in the strictly collective `complete` method. The `invoke` methods themselves *may* make use of collective methods for their implementation. 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).

3.2. The CARI API

Figure 1(b) is somewhat stylized. Without comment, we have assumed an object-oriented C++ style of programming. 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 `vertices` array.

A more realistic API for CARI is given in Figure 2. The constructor for `CARISchedule` is passed an MPI communicator, and an application-specific object whose class extends `CARIHandler`. The constructor may also be passed workspace in the form of buffer space allocated in the application code. Internally, `CARISchedule` will carve this workspace according to its requirements—if, for example, the implementation follows that of Figure 3(b), this workspace will be carved into a send buffer and a receive buffer, and perhaps extra workspace used in sorting the send buffer.

The `invoke` method takes a pointer to an instance of `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 dereferenced in the context of a remote processor. This applies equally to `char*` pointers—strings that are to be processed remotely must be passed as `char` arrays of constant size.

The second argument of `invoke` 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 S.

To simplify the task of the programmer, the CARI API comes 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`—which is also modified in the main body of the client code or in another handler—there is no need for explicit mutual exclusion protecting this operation.

4. Gadget-2

Gadget-2 is a free production code for cosmological N-body and hydrodynamic simulations. The code is written in the C language and parallelized using MPI. It simulates the evolution of very large (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. We are particularly interested in the parallelization strategy of Gadget-2, which is based on an irregular and dynamically adjusted domain decomposition, with copious communication between processors.

4.1. 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 naïve 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 [15]; the second is a hybrid of BH and a Particle Mesh (PM) method called *TreePM*.

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.

4.2. 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 ad hoc 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.

```
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 ...
}
```

(a) Simplified sketch of `compute_potential`

```
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 ...
    }
  }
}
```

(b) More realistic structure of `compute_potential`

Figure 3: Two different views of `compute_potential` code

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.

In this paper we focus on 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.

4.3. Particle Export

Figure 3(a) 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(a) 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.

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.

The actual code of `compute_potential` is complicated by the fact that Gadget-2 allocates fixed size buffers for communication. Figure 3(b) 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 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. 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(a) and 3(b) only reproduce its general structure. The point to observe is

```
CARISchedule <S, T>sched;
for( ... all local particles ... ){
  ... Compute local contribution to potential, and flag exports ...
  for( ... all peers particle should be exported to ... ){
    sched.invoke(pindex, processid, particle);
  }
}
sched.complete() ;

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

Figure 4: Refactored `compute_potential` code

that application specific code is scattered through the main ”skeleton” of communications-related code. The question that concerns us how the code can be refactored 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.

4.4. Implementing Gadget-2 using the CARI Library

A reorganization of the code that achieves our objective is given in Figure 4. Here `CARI` is a library class that abstracts all communication. `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 3(a), 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(a) and 3(b). The other two sections of application code are in the *callbacks* `handleRequest` and `handleResponse`.

If the implementation of `CARI` corresponded to Figure 3(a), 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(a) following the “main loop”, calling the user-defined callbacks at appropriate points.

If the implementation of `CARI` corresponds to Figure 3(b), 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. 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 over Figure 3(b) may not be immediately apparent. But in reality the communication code in Figure 3(b) that is abstracted away in the `CARI` class is one to two hundred lines of rather dense MPI; in Figure 4 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(b) to Figure 4 with essentially no changes in the underlying communication pattern, or efficiency of the programme.

5. Performance Evaluation

As part of the work presented in this paper, we have developed two implementations of the CARI API. The first implementation—called Synchronous CARI (*SCARI*)—is developed with the aim to exactly preserve the performance of the original application code. Mostly this implementation reorganizes the original source code, and attempts not to incur any performance overhead. SCARI makes extensive use of MPI collective communications and blocking point-to-point calls. 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. We plan to present details of these two implementations in the future work.

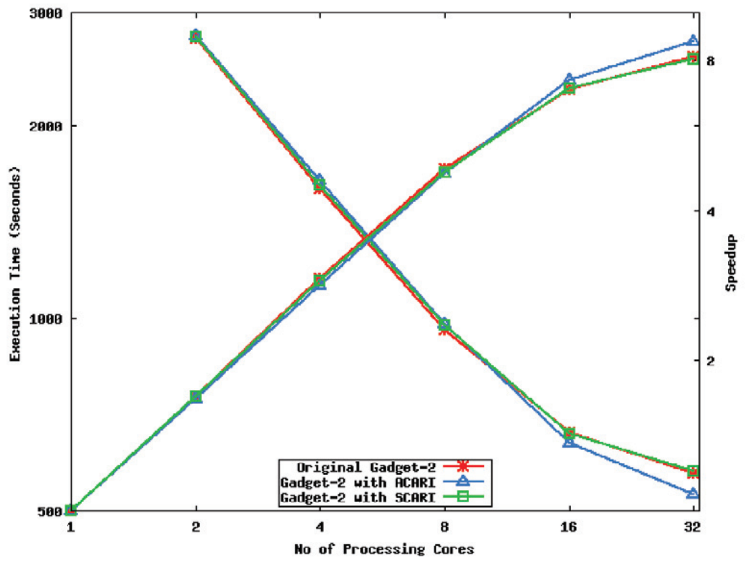

Figure 5: Execution Time Comparison for the Original, SCARI, ACARI versions of Gadget-2 code

We begin by describing our test environment, which consisted of a 32 processing core Linux cluster at NUST. The cluster consists of eight compute nodes. Each node contains a quad-core Intel Xeon processor and has 2 Gigabytes of main memory. The nodes are connected via Myrinet and Gigabit Ethernet. We used MPICH2 version $1.2.1p1$—as the MPI library—that was compiled with GNU C Compiler (GCC) version 4.1.0. We used the Cluster Formation Simulation for executing the original and CARI versions of the Gadget-2 code.

As part of the evaluation effort, we present the execution time in Figure 5 for three variants of the Gadget-2 code. These include the original Gadget-2, Gadget-2 using SCARI, and Gadget-2 using ACARI. The Gadget-2 code in the second and third case is exactly the same; the only difference is the implementation of the CARI library.

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 ACARI implementation of the CARI API slightly outperforms the SCARI-based and the original application code for all processor counts. In fact the performance of ACARI version gets even better as the number of cores are added to the parallel computation. Although the gain observed by ACARI 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 ACARI is the asynchronous nature of the communication algorithm.

We quantify productivity benefits of the CARI API by presenting Source Lines of Code (SLOC) comparison in Table 1 between Gadget-2 versions using MPI and CARI. There is a notable reduction in SLOC in the case of the gravity calculation physics code. It must be noted that our proposed API (CARI) is applicable to less than 10% 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 calculations), the CARI API significantly reduces SLOC (mostly dense and complicated MPI code) and improves maintainability of the code.

6. Conclusions and Future Work

In this paper 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

Table 1: Comparison of Source Lines of Code (SLOC)

Gadget-2	**Application Code**	**Gravity Code**	**Potential Code**	**Density Code**	**Hydra Code**	**Force Test Code**
Using MPI	16981	551	350	618	572	354
Using CARI	16102	352	218	448	412	215

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.

The paper also introduced and evaluated implementations of the CARI schedule. 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 ACARI, is developed to show that it is possible to improve performance and programming productivity at the same time. 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.

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 an open-source software in the mid of 2011.

References

[1] Y. Hoskote, S. Vangal, A. Singh, N. Borkar, S. Borkar, A 5-ghz mesh interconnect for a teraflops processor, IEEE Micro 27 (5) (2007) pp. 51–61.

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

[3] V. Springel, The cosmological simulation code GADGET-2, Monthly Notices of the Royal Astronomical Society, 364(4) (2005) pp. 1105–1134.

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

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

[6] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, Y. Zhou, citeseer.ist.psu.edu/blumofe95cilk.htmlCilk: An Efficient Multithreaded Runtime System, Journal of Parallel and Distributed Computing 37 (1) (1996) pp. 55–69.

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

[8] G. Z. Bryan, B. Carpenter, G. Fox, X. Li, 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), July 1998.

[9] S. Lim, B. Carpenter, G. Fox, 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), 2003.

[10] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, E. Aprà, Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit, Int. J. High Perform. Comput. Appl. 20 (2) (2006) pp. 203–231.

[11] L. G. Valiant, A Bridging Model for Parallel Computation, Commun. ACM 33 (8) (1990) pp. 103–111.

[12] R. W. Numrich, J. Reid, Co-array Fortran for Parallel Programming, SIGPLAN Fortran Forum 17 (2) (1998) pp. 1–31.

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

[14] L. V. Kale, S. Krishnan, CHARM++: A Portable Concurrent Object Oriented System Based on C++, SIGPLAN Not. 28 (10) (1993) pp. 91–108.

[15] J. Barnes, P. Hut, A Hierarchical O(N log N) Force-calculation Algorithm , Nature 324 (4) (1986) pp. 446–449.