Charisma: Orchestrating Migratable Parallel Objects

Chao Huang   Laxmikant Kalé
University of Illinois at Urbana-Champaign
201 N Goodwin Ave
Urbana, IL 61801, USA
{chuang10,kale}@cs.uiuc.edu

Abstract

The parallel programming paradigm based on migratable objects, as embodied in Charm++, improves programmer productivity by automating resource management. The programmer decomposes an application into a large number of parallel objects, while an intelligent run-time system assigns those objects to processors. It migrates objects among processors to effect dynamic load balance and communication optimizations. In addition, having multiple sets of objects representing distinct computations leads to improved modularity and performance. However, for complex applications involving many sets of objects, Charm++'s programming model tends to obscure the global flow of control in a parallel program: One must look at the code of multiple objects to discern how the multiple sets of objects are orchestrated in a given application. In this paper, we present Charisma, *an orchestration notation that allows expression of Charm++ functionality without fragmenting the expression of control flow. Charisma separates expression of parallelism, including control flow and macro dataflow, from sequential components of the program. The sequential components only consume and publish data. Charisma expression of multiple patterns of communication among message-driven objects. A compiler generates Charm++ communication and synchronization code via static dependence analysis. As Charisma outputs standard Charm++ code, the functionality and performance benefits of the adaptive run-time system, such as automatic load balancing, are retained. In the paper, we show that Charisma programs scale up to 1024 processors without introducing undue overhead.*

Categories and Subject Descriptors

D.1.3 [**Concurrent Programming**]: Parallel programming; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Design, Languages

Keywords

Adaptivity, Parallel Programming Productivity, Migratable Objects, Orchestration

1. Introduction

Our approach to parallel programming seeks an optimal division of labor between the run-time system and the programmer. In particular, it is based on the idea of migratable objects. The programmer decomposes the application into a large number of parallel computations executed on parallel objects, while the run-time system assigns those objects to processors (Figure 1). This approach gives the run-time system the flexibility to migrate objects among processors to effect load balance and communication optimizations.

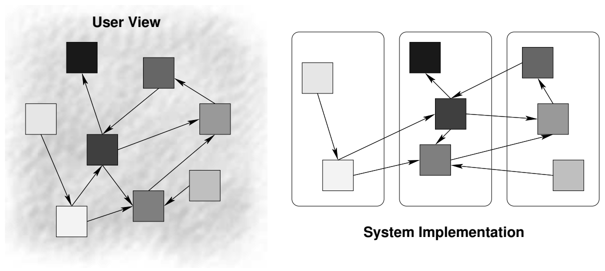


Figure 1. With Charm++, User Programs with Objects and System Maps Objects to Processors

Charm++ is a framework that embodies this concept. Charm++ objects, or *Chares*, execute parallel subtasks in a program and communicate via asynchronous method invocations. A method

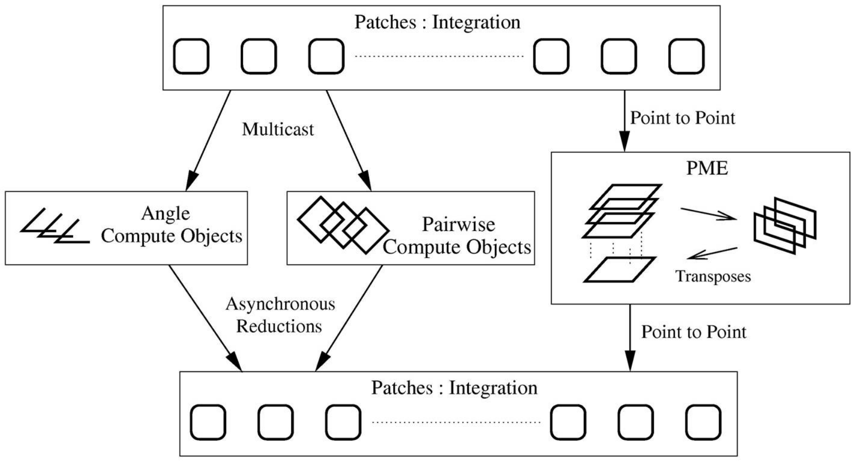


Figure 2. Structure of a Molecular Dynamics Simulation Application: NAMD

on a remote object can be invoked by a message, and the caller does not wait for the method to return. Many chares can be organized into an indexed collection called a *chare array*, and a single program may contain multiple chare arrays for different sets of subtasks. Applications developed with Charm++ enjoy several performance benefits including adaptive overlap of communication and computation, automatic load balancing, system-level fault tolerance support, and communication optimizations. The idea of using over-decomposition and indirection in mapping work to processors has been studied in the past, including in DRMS [18]. Using process migration for load balancing has also been investigated [5], and this broad approach has gained momentum recently [1, 10].

A large number of applications have been developed using the Charm++ framework, such as NAMD [12], a production-level molecular dynamics program which has demonstrated unprecedented speedups on several thousand processors, and LeanCP [17], a Quantum-Chemistry simulation application. Other examples include rocket simulation, crack propagation, space-time meshing with discontinuous Galerkin solvers, dendritic growth in solidification processes, level-set methods, computational cosmology simulations, and parallel visualization of cosmology data.

Although Charm++ has demonstrated its utility in runtime optimizations such as load balancing, and although it is more modular than MPI (see [11]), it can be challenging to clearly express the flow of control due to its local view of control, especially for complex applications that involve multiple sets of chare-arrays, as seen in the motivating example in the next section.

Also, in Charm++, methods clearly distinguish the places where data is *received*, but the places where data is *sent* (invocations) can be buried deep inside functions of the object code. This asymmetry often makes it hard to see the parallel structure of an application, which is useful for understanding performance issues.

We present **Charisma**, a higher-level language notation that retains the benefits of Charm++ while allowing for easy expression of global flow of control as well as symmetric expression of communication. Charisma separates sequential code fragments (methods) from the parallel constructs, and allows the programmer to describe the global control flow with a script language. Since the script language controls the behavior of a *collection* of migratable objects, we also call it an orchestration language.

2. Motivation

Many scientific and engineering applications have complex structures. Some may involve a large number of components with complicated interactions between them. Others may contain multiple modules, each with complex structures. Unfortunately, for these applications, conventional parallel programming models do not handle the balance between high performance and programming productivity well. OpenMP [2] programs have a shared view of data and control. The programmer writes code for all the components of the program, with only independent loop iterations executed in parallel. This model may be easy to program for a subset of applications, but it is often incapable of taking advantage of large scale parallelism among modules and concurrent control flows, and consequently suffers poor scalability. MPI [15], which represents the message passing model, provides a processor centric programming model. A parallel job will be divided into subtasks according to the number of available processors, and data needed for each subtask is localized onto that processor. Then the user expresses an algorithm in the context of local MPI processes, inserting message passing calls to exchange data with other processes. Basically, it provides a local view of data and a local view of control, although for SPMD programs, the global flow of control is often similar to the local flow of control. Performance wise, MPI programs can achieve high scalability, especially if the program has “regular” patterns, typically with systolic computation-communication super-steps. Some algorithms are simply too difficult to be written in such a fashion. In terms of productivity, this model is fairly easy to program when the application does not involve many modules. Otherwise the programmer will have to first partition the processors between modules, losing the potential performance opportunity of overlapping communication and computation across modules, as well as doing resource management across modules. Some programmers may choose to assign multiple roles to the same group of processors for the sake of performance. This results in complexity in writing the message passing procedures, and compromises productivity.

For a concrete example, consider a 3D molecular dynamics

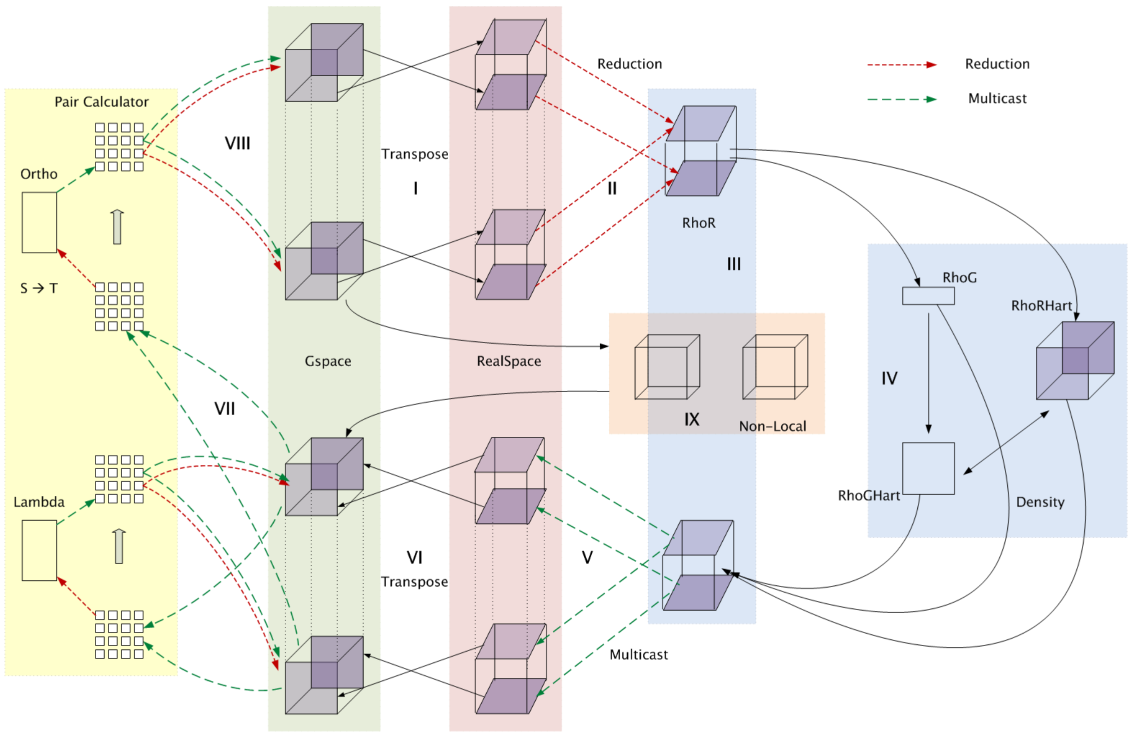


Figure 3. Structure of a Quantum Chemistry Simulation Application: LeanCP

simulation application NAMD [16] illustrated in Figure 2 (taken from [16]). This simplified version of NAMD contains 3 types of components. The spatially decomposed cubes, shown by squares with rounded corners, are called *patches*. A patch, which holds the coordinate data for all the atoms in the cube of space corresponding to that patch, is responsible for distributing the coordinates, retrieving forces, and integrating the equations of motion. The forces used by the patches are computed by a variety of *compute* objects, with Angle Compute and Pairwise Compute shown in the figure as examples. There are several varieties of compute objects, responsible for computing different types of forces (bond, electrostatic, constraint, etc.). Some compute objects require data from one patch and only calculate interaction between atoms within that single patch. Others are responsible for interactions between atoms distributed among neighboring patches. *PME* objects implement the Particle Mesh Ewald method [3], which is used to compute the long-range electrostatic forces between atoms. PME requires two 3D Fast-Fourier-Transform (FFT) operations. The 3D FFT operations are parallelized through a plane decomposition, where first a 2D FFT is computed on a plane of the grid, followed by a global transpose and a 1D FFT along the third dimension. The simulation in NAMD is iterative. At each time step, the patches send out coordinate data to compute objects and PME objects as necessary, and the compute objects and PME objects perform the force calculations in parallel. When force information is available, it is then communicated back to the patches, where integration is performed.

When we consider the various programming models for this relatively simple molecular dynamics application, we find it often difficult to reach a graceful balance between productivity and performance.

Programming with OpenMP, one will write code that, in effect, serializes the coordinate distribution, angle force calculation, pairwise force calculation, PME calculation, force reduction, and patch integration. The flow of the program looks clear, but it is unable to parallelize concurrent subtasks, such as angle force calculation and pairwise force calculation, unless wildcard receives with awkward cross-module flow of control are used. Performance and scalability are sacrificed for the ease of programming.

MPI allows the programmer to partition the job into groups of subtasks and assign the subtasks onto partitions of available MPI processes. The programmer can choose to overlap several subtasks onto same set of processes to keep the CPUs busy. For example, if we have some patch objects and some compute objects residing on the same processor, the patches may use the CPU for the coordinates multicast, and subsequently yield the CPU to the compute objects for force calculation. Since MPI message passing is based on processors, when the programmer wants to express the intention to "send message to subtask S", he/she needs to make the MPI call to send the message explicitly to processor rank K instead of subtask S's ID. Therefore, the programmer has to maintain a mapping between the subtask IDs to the process ranks.

To achieve higher CPU utilization, we want to be able to process the messages as soon as they are received. When the message passing model does in-order message processing with tag matching, the time wasted on waiting for the next in-order message to arrive is unavoidable. The programmer can take advantage of wildcard source and tag matching, accepting any incoming message, and process them accordingly. While it is possible to achieve high efficiency, this approach has a major productivity drawback. When there are multiple subtasks from multiple components on one processor, it is difficult to maintain a definite mapping from an arbitrary incoming message to its destination and handler function. The message passing calls will look confusing, and the flow of control cannot be expressed clearly.

Charm++, like MPI, provides a local view of control, but unlike MPI, it takes an object-based approach. The programmer writes code for various classes for different subtasks, then instantiates object arrays of arbitrary size from such classes. These objects are assigned onto physical processors by the run-time system automatically, and therefore the programmer does not have to be restricted by the concept of processor. In Charm++'s asynchronous method invocation model, each object's code specifies, in a reactive manner, what the object will do when presented with a particular message. When a message is delivered, its destination object and the method to invoke on that object are stated. Because the message contains information on what to do with it at the receiver side, this can be called an *active message* [19]. Such active messages ensure the prompt processing of data as they become available, and the Adaptive Run-Time System (ARTS) offers further opportunities for performance optimization. However, for complex programs with a large number of object arrays, this comes at a cost of obscuring the overall flow of control: The transfer of control is fragmented by the message sending between objects. To follow the flow of control, one often needs to dig deep into the objects' class code and hop from one to another, and in the meanwhile, to understand parallel operations, such as broadcast, multicast and reduction, among the objects. This poses some difficulty for the expression of the parallel program for both the programmer and its readers.

The above example is not an extremely complicated parallel program. Indeed, it has only 3 types of components and a few short-running concurrent control flows. A quantum chemistry simulation [17] under development using Charm++ involves 11 different parallel structures, together with complex concurrent flows (See Figure 3). Clearly, understanding the global control flow is difficult by looking at individual object's codes.

The language we propose, Charisma, aims at achieving high programming productivity without losing the performance benefits from the ARTS. It describes the global view of control in a parallel application or module.

3. Language Design

Charisma employs a macro dataflow approach for productive parallel programming. The programmer writes a script-like orchestration program containing statements that produce and consume collections of values. From analyzing such producing and consuming statements, the control flows can be organized, and messages and method invocations can be generated. This idea is similar to the macro dataflow model [8] and the hybrid dataflow architecture model [7]. In [7], the data-driven distributed control model is combines with the traditional von Neumann sequential control model to exploit fine-grain parallelism without sacrificing the performance benefits of the existing optimizations such as pipelining. In contrast to the instruction level dataflow, Charisma's object-level macro dataflow mechanism takes advantage of the message-driven execution model in Charm++'s and enables dynamic resource management such as automatic load balancing.

A Charisma program consists of two components: the *orchestration code* (in `.or` file) that describes the global view of control, and the *sequential code* (in `.h` and `.C` files) that specifies the local behavior of individual objects. Charisma compiler generates parallel code from the orchestration statements and integrate the sequential methods to produce the target Charm++ program. By separating parallel code from sequential code, the programmer can focus better on the local actions on the objects, such as physics computation.

3.1 Parallel Object Array

In Charisma, a program is composed of parallel objects. A collection of such objects can be organized into an array to perform a subtask, such as the patches and the force calculators in the previous NAMD example. Although they are called "arrays", these are really a collection of objects indexed by a very general indexing mechanism. In particular, the objects can be organized into 1-D or multi-dimensional arrays that can be sparse, or into collections indexed by arbitrary bit-patterns or bit-strings. One can also dynamically insert and delete elements in an object array. Charm++'s ARTS is responsible for adaptively mapping the object array elements onto available physical processors efficiently.

Moreover, these objects are migratable with support from the ARTS. Once created, these parallel objects report the workload at run-time to the system load balancer, and the load balancer will automatically migrate the objects as necessary to achieve higher overall utilization.

```
class Cell : ChareArray2D;
class CellPair : ChareArray4D;

obj cells :  Cell[N,N];
obj cellpairs :  CellPair[N,N,N,N];
```

Above is an example of object array declaration in orchestration code for a 2-D Molecular Dynamics (MD) application. The first part is class declaration for class `Cell` and `CellPair`. The second part is the instantiation of two object arrays `cells` and `cellpairs` from these classes. The array `cells` is responsible for holding the atom information in the 2-D partition that corresponds to its index, and the array `cellpairs` does the pair-wise force calculation for a pair of `cells` objects.

The programmer also provides sequential code that specifies the behavior of individual objects. There will be a `.h` file for each class, with class member variables and methods that are needed for sequential user code. Note that this header file does not have complete class declaration. It just has the variables and methods declaration used in the sequential code. The definition of those sequential functions is provided in the `.C` files.

3.2 `Foreach` Statement

In the main body of orchestration code, the programmer describes the behavior and interaction of the elements of the object arrays using orchestration statements. The most common kind of parallelism is the invocation of a method across all elements in an object array. Charisma provides a *foreach* statement for specifying such parallelism. The keywords `foreach` and `end-foreach` forms an enclosure within which the parallel invocation is performed. The following code segment invokes the entry method `doWork` on all the elements of array `myWorkers`.

```
foreach i in myWorkers
    myWorkers[i].doWork();
end-foreach
```

The foreach statement looks very much like the FORALL statement in HPF [13]. Indeed, they both express the global flow of control. In HPF, FORALL provides a parallel mechanism for value assignment of elements of a distributed data array, whereas the foreach statement in Charisma specifies the parallelism among the entry method invocation of parallel objects.

The programmer can have multiple statements in one foreach enclosure, if those statements are invoked on the same object array with the same indexing. This is really a shorthand notation for having one foreach enclosure for each of these statements. Note also that the implementation does not need to broadcast a control message to all objects to implement this. Global control can be compiled into local control, and modulated by data dependencies described below.

3.3 Producer-Consumer Model

In MPI model, message passing is via specifying the destination processor's rank and communicator, with a tag to be matched. As explained earlier, this mechanism does not always work well in achieving both performance and clear algorithm expression in presence of complex parallel programs. Charm++'s message delivery specifies the destination object and the function handler. With this information, the destination object knows which function to invoke to process the incoming message. While Charm++ offers a more intuitive way to deal with communications between subtasks, the programmer still needs to worry about sending and receiving messages while writing sequential part of the code. To further separate the task of writing communication code for parallelism and composing the sequential computation blocks in a parallel program, Charisma supports producer-consumer communication directly.

In the orchestration code, there is no function call for explicitly sending or receiving message between objects. Instead, each object method invocation can have input and output parameters. Here is an orchestration statement that exemplifies the input and output of this object method workers.foo.

```
foreach i in workers
    <q[i]> := workers[i].foo(p[i+1]);
end-foreach
```

Here, the entry method workers[i].foo produces (or *publishes* in Charisma terminology) a value q, enclosed in a pair of angular brackets before the publishing sign ":=". Meanwhile, p is the value consumed by the entry method. An entry method can have an arbitrary number of published (produced and reduced) values and consumed values. In addition to basic data types, each of these values can also be an object of arbitrary type. The values published by A[i] must have the index i, whereas values consumed can have the index e(i), which is an index expression in the form of $i \pm c$ where c is a constant. Although we have used different symbols (p and q) for the input and the output variables, they are allowed to overlap.

The variables that can be used as input and output values are declared in the *parameter space* in Charisma. The variables in the parameter space correspond to global data items or data arrays of a restricted shared-memory abstraction. The programmer uses them solely in the orchestration code to facilitate the producer-consumer model, and has no knowledge of them in the local-view sequential code. A parameter variable can be of an intrinsic or user-defined data type, or a data array.

```
param error : double;
param atoms : AtomBucket;
param p : double [256];
```

When composing the sequential code in Charisma, the programmer does not need the knowledge of the sources of the input data or the destinations of the output data. The input data is seen as parameters passed in, and the output data is published via a local function call. Specifically, for producing, a reserved keyword *outport* is used to mark the parameter name to be produced as appears in the orchestration code, and a *produce* call associates the outport parameter name with an actual local variable whose value is to be sent out. For instance, in the sequential code for WorkerClass::foo, the programmer makes a local function call produce with *outport* variable q to *publish* the value of a local variable local_q (assuming p and q are double precision type).

```
WorkerClass::foo(double p[], outport q) {
    local_q = ...;
    ...
    produce(q, local_q);
}
```

Fortran-M [6] is similar to Charisma because they both use the concept of *port*. In Fortran-M, ports are connected to create *channels* from which point-to-point communications are generated. It is useful in facilitating data exchange between dissimilar subtasks. Charisma analyzes the *inports* and *outports* of data and generate communications for both point-to-point and collective operations among object arrays, by analyzing data dependencies among parameters in the orchestration code. The goal of Charisma is to provide a way of clearly expressing global flow of control in complicated parallel programs. In addition, Charisma is built on top of a powerful adaptive run-time system which offers the generated program performance benefits at no additional cost of programming complexity.

3.4 Organizing Parallel Control Flows

The control transfer in a Charisma program is clearly expressed in the orchestration code. After the initial statements in the control chain, which typically do not consume any value, the control flow progresses in a data-driven fashion. If a statement consumes some values, then as soon as the values are available, the statement can be executed, without any barrier across the object array or global synchronization. Charisma extends the message driven model of Charm++, taking advantage of its high efficiency and offering clear expression of the control flow and programming productivity.

In the producer-consumer model, Charisma respects the program order in connecting producing and consuming ports. In other words, a consuming statement will look for the value produced by the latest producing statement in the program order. In a legal orchestration program, each consuming statement and tagged input value has its corresponding unique producing statement. Of course, a single produced value may be consumed by multiple consuming statements. If a producing statement does not have a later consuming statement, the produced value will not have any effect on the program behavior.

Beyond the program order restriction of the data flow, Charisma is consistent with Charm++'s asynchronous invocation model, in which explicit barrier or other synchronization operation is not supported. If the programmer does need to enforce a barrier operation, a dummy reduction can be used (see Section 3.5).

This also means there is no further implicit barrier between `foreach` statements. For instance, during any iteration in the following code, `workers[2].bar` does not have to wait till `workers[2].foo` has completed. As soon as `p[1]` is published by `workers[1].foo`, even if `workers[2].foo` has not started yet, `workers[2].bar` can start executing before `workers[2].foo`.

```
for iter = 1 to MAX_ITER
    foreach i in workers
        <p[i]> := workers[i].foo();
    end-foreach
    foreach i in workers
        workers[i].bar(p[i-1]);
    end-foreach
end-for
```

Loops are supported with `for` statement and `while` statement. The first consuming statement will look for values produced by the last producing statement before the loop for the first iteration, and the last producing statement within the loop body for the following iterations. At the last iteration, the last produced values will be disseminated to the code segment following the loop body. Take the code segment in Figure 5 as an example, the `coords` produced in the first `foreach` statement is consumed by the first consuming statement in the for-loop. Thereafter, each iteration produces a fresh `coords` from the `integrate` function at the end to be consumed at the next iteration. The produced parameter of `coords` is available after the for-loop, although it is not used here in this example.

3.5 Describing Communication Patterns

The method invocation statement in the orchestration code specifies its consumed and published values. These actions of consuming and publishing are viewed as input and output ports, and Charisma run-time will *connect* these ports by automatically generating efficient message between them. Using the language and the extensions described below, the programmer is able to express various communication patterns.

• Point-to-point communication

We now introduce the mechanism to allow point-to-point communication among objects via the producer-consumer model. For example, in the code segment below, `p[i]` is communicated via a message in asynchronous method invocation between elements of object array `A` and `B`.

```
foreach i in A
    <p[i]> := A[i].f(...);
end-foreach
foreach i in B
    <...> := B[i].g(p[i]);
end-foreach
```

From this code segment, a point-to-point message will be generated from `A[i]`'s publishing port to `B[i]`'s consuming port. When `A[i]` calls the local function `produce()`, the message is created and sent to the destination `B[i]`. By this mechanism, we avoid using any global data and reduce potential synchronization overhead. For example, in the code segment above, `B[2].g()` does not have to wait on all `A[i].f()` is completed to start its execution; as soon as `A[2].f()` is done and the value p[2] is filled, `B[2].g()` can be invoked. In fact, even before `A[i].f()` completes, `p[i]` can be sent as soon as it is produced, using callback in the implementation.

• Reduction

In Charisma, the publishing statement uses a `+` to mark a reduced parameter whose value is to be obtained by a reduction operation across the object array. Following is an example of a reduction of value `err` on a 2-D object array `A`.

```
foreach i,j in workers
    <..., +err> := workers[i,j].bar(..);
end-foreach
...
Main.testError(err);
```

In the sequential code for `WorkerClass::bar`, the programmer calls a local function `reduce` to publish its local value `local_err` and specifies the reduction operation ">" (for MAX). Similar to the `produce` call, an `outport` keyword indicates for which output port parameter this reduce call is publishing data. This call is almost identical to the `produce` primitive, only with an extra parameter for specifying the reduction operation.

```
WorkerClass::bar(..., outport err) {
    local_err = ...;
    ...
    reduce(err, local_err, ">");
}
```

The dimensionality of the reduced output parameter must be a subset of that of the array publishing it. Thus reducing from a 2-D object array onto a 1-D parameter value is allowed, and the dimension(s) on which the reduction will be performed on is inferred from comparison of the dimensions of the object array and the reduced parameter.

• Multicast

A value produced by a single statement may be consumed by multiple object array elements. For example, in the following code segment, `A[i]` is a 1-D object array, `B[j,k]` is a 2-D object array, and `points` is a 1-D parameter variable. Suppose they all have the same dimensional size N.

```
foreach i in A
    <points[i]> := A[i].f(...);
end-foreach
foreach k,j in B
    <...> := B[k,j].g(points[k]);
end-foreach
```

There will be N messages to send each published value to the consuming places. For example, `point[1]` will be multicast to N elements in `B[1,0..N-1]`.

• Scatter, Gather and Permutation Operation

A collection of values produced by one object may be split and consumed by multiple object array elements for a scatter operation. Conversely, a collection of values from different objects can be gathered to be consumed by one object. Combining the two, we have the permutation operation.

```
foreach i,j,k in cells
  <coords[i,j,k]> := cells[i,j,k].produceCoords();
end-foreach
for iter := 1 to MAX_ITER
  foreach i1,j1,k1,i2,j2,k2 in cellpairs
    <+forces[i1,j1,k1],+forces[i2,j2,k2]> := cellpairs[i1,j1,k1,i2,j2,k2].
                                calcForces(coords[i1,j1,k1],coords[i2,j2,k2]);
  end-foreach
  foreach i,j,k in cells
    <coords[i,j,k],+energy> := cells[i,j,k].integrate(forces[i,j,k]);
  end-foreach
  MDMain.updateEnergy(energy);
end-for
```

Figure 4. MD with Charisma: Clear Expression of Global View of Control

```
/* Scatter Example */
foreach i in A
    <points[i,*]> := A[i].f(...);
end-foreach
foreach k,j in B
    <...> := B[k,j].g(points[k,j]);
end-foreach
```

A wildcard dimension “*” in `A[i].f()`’s output `points` specifies that it will publish multiple data items. At the consuming side, each `B[k,j]` consumes only one point in the data, and therefore a scatter communication will be generated from `A` to `B`. For instance, `A[1]` will publish data `points[1,0..N-1]` to be consumed by multiple array objects `B[1,0..N-1]`.

```
/* Gather Example */
foreach i,j in A
    <points[i,j]> := A[i,j].f(...);
end-foreach
foreach k in B
    <...> := B[k].g(points[*,k]);
end-foreach
```

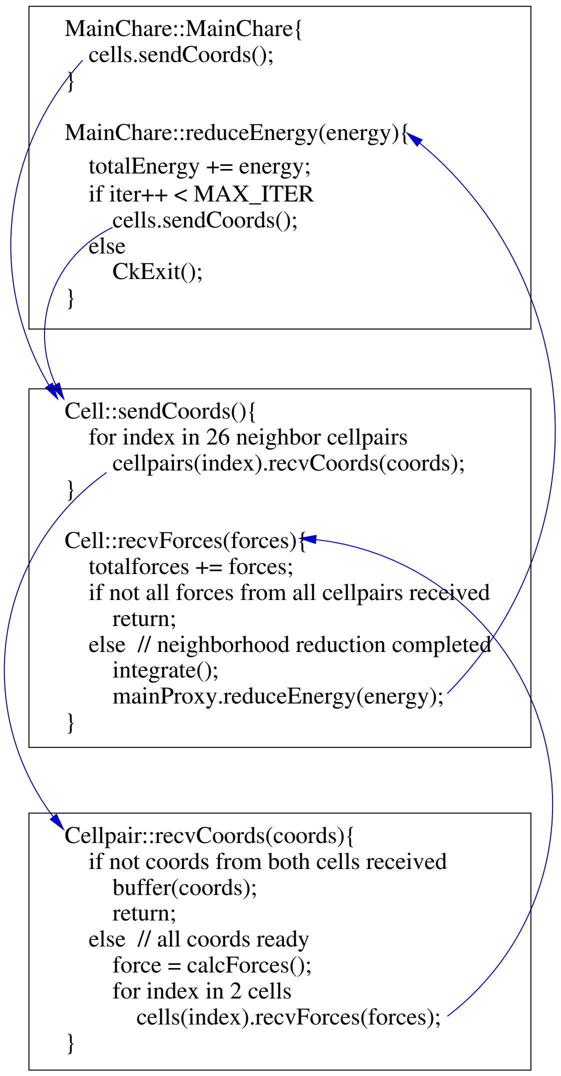
Similar to the scatter example, if a wildcard dimension “*” is in the consumed parameter and the corresponding published parameter does not have a wildcard dimension, there is a gather operation generated from the publishing statement to the consuming statement. In the following code segment, each `A[i,j]` publishes a data point, then data points from `A[0..N-1,j]` are combined together to for the data to be consumed by `B[j]`.

Combining scatter and reduction operations, we get the permutation operation. Please refer to Section 6.2 for an code example.

4. Code Example: MD

In this section, we show how Charisma can overcome some of Charm++’s difficulty of describing global view of control with a concrete example to . This example is a simplified version of the NAMD simulation explained in Section 2, with only the pairwise force calculation included. `Cells` are the objects that hold the coordinates of atoms in patches, and `cellpairs` are the objects calculating pairwise forces between two `cells`. In the following comparison, sequential functions such as `Cell::Integrate` and `CellPair::calcForces` are not listed, since they access only local data and should be the same for both versions.

With Charisma, the MD code is listed in Figure 4. First, elements in object array `cells` *produce* their coordinates, providing the initial data for the first iteration. During each iteration,



```
MainChare::MainChare{
  cells.sendCoords();
}

MainChare::reduceEnergy(energy){
  totalEnergy += energy;
  if iter++ < MAX_ITER
    cells.sendCoords();
  else
    CkExit();
}
```

```
Cell::sendCoords(){
  for index in 26 neighbor cellpairs
    cellpairs(index).recvCoords(coords);
}

Cell::recvForces(forces){
  totalforces += forces;
  if not all forces from all cellpairs received
    return;
  else  // neighborhood reduction completed
    integrate();
    mainProxy.reduceEnergy(energy);
}
```

```
Cellpair::recvCoords(coords){
  if not coords from both cells received
    buffer(coords);
    return;
  else  // all coords ready
    force = calcForces();
    for index in 2 cells
      cells(index).recvForces(forces);
}
```

Figure 5. MD with Charm++: Overall Control Flow Buried in Objects’ Code

cellpairs calculate forces by *consuming* the coordinates provided by two cells elements. In the same statement, cellpairs *produce* forces combined via a reduction within a cell's neighborhood. These values get *consumed* in the integration phase. The integration also *produces* coordinates for the next iteration and total energy via a reduction operation across all cells. In the Charisma code, each orchestration statement specifies which pieces of data it *consumes* and *produces*, without having to know the source and destination of those data items.

Figure 5 lists corresponding Charm++ pseudo code for the same program. In three boxes are method definitions for three classes MainChare, Cell, and CellPair, which are typically separated in different C files. To organize the global control flow, one has to dig into the files and hop among them (represented by the arrows). Thus, the flow is fragmented and buried in the object code. Following control flow in such a parallel program is more complicated than in sequential object-oriented programming code, due to the complexity of the parallel operations among the objects. For instance, collecting force data among a cell's neighboring cellpairs through a neighborhood reduction requires non-trivial code (not shown in the pseudo code here), and this kind of code is automatically generated in the Charisma version.

We are not listing the corresponding MPI code here, because it would be much more complicated than the Charm++ version. In addition to handling the collective operations, the MPI programmer has to write code for explicitly managing various sets of subtasks, maintaining mapping scheme between subtasks' identities and their physical locations (processor number), and auxiliary code such as load balancing. When the programmer wants to achieve higher degree of overlap between computation and communication, more code is needed to handle the wildcard source and tag matching as discussed in Section 2.

5. Implementation

Charisma generates Charm++ code which can be compiled and run in the adaptive run-time system. The code generation starts with parsing the orchestration code in the .or file. Once the input and output parameters are identified for each orchestration statement, static dependency analysis is performed to find the connections between these input and output parameters. By analyzing the indices of the parameters and of the object arrays, a global graph of control flow is created. Next, Charisma generates appropriate method invocations and messages from the graph of parallel control flow, since the program progress in Charm++ is driven by asynchronous remote invocation with messages. During this process, parallel code for expressing a variety of communication patterns, including broadcast, multicast and reduction, is produced. After the parallel flow is set up, the user's sequential code is integrated into the final output of the Charm++ program.

An important goal in the implementation of Charisma is to ensure the high efficiency of the generated code. One technique is immediate outgoing messages. As soon as the data for an outgoing message becomes available (indicated to the ARTS by a *publish* statement), the message is assembled and sent out, without having to wait for the function to complete. This mechanism allows for a larger degree of adaptive overlap between communication and computation. Another optimization improves memory efficiency. When multiple messages are needed to drive the next link in the flow, such as in a gather operation, the naive solution of buffering messages already received into user-allocated memory incurs overhead for a memory copy. Charisma eliminates this unnecessary memory copy by postponing the deleting of the received Charm++ messages until after all the messages have been received.

Charisma also offers the user great flexibility to customize the parallel program. The current implementation supports creation of a sparse object array and its collective operations. The user can supply sequential functions to provide hints to Charisma on issues such as which elements to create in the sparse object array.

6. Experiments and Results

In this section, we show the results of a few benchmarks with Charisma. We compare the productivity, in terms of source lines of code (SLOC), as well as the performance and scalability. The benchmarking platforms are PSC's Cray XT3 MPP system with 2068 dual 2.6 GHz AMD Opteron compute nodes linked by a custom-designed interconnect, and NCSA's Tungsten Cluster with 1280 dual 3.2 GHz Intel Xeon nodes and Myrinet network.

6.1 Stencil Calculation

Our first benchmark is a 5-point stencil calculation. This is a multiple timestepping calculation involving a group of regions in 2-D decomposition of a 2-D mesh. At each timestep, every region exchanges its boundary data with its immediate neighbors in 4 directions and performs local computation based on the neighbors' data. This is a simplified model of many applications including fluid dynamics and heat dispersion simulation, and therefore it can serve the purpose of demonstration.

Figure 6 compares the performance of the stencil calculation benchmark written in Charisma vs. Charm++. The total problem size is 16384^2 decomposed onto 4096 objects. The performance overhead introduced by Charisma is 2 - 6%, scaling up to 1024 processors. Because this benchmark is relatively simple, the parallel code in Charm++ forms a significant part of the code. Therefore we see a 45% reduction on SLOC with Charisma.

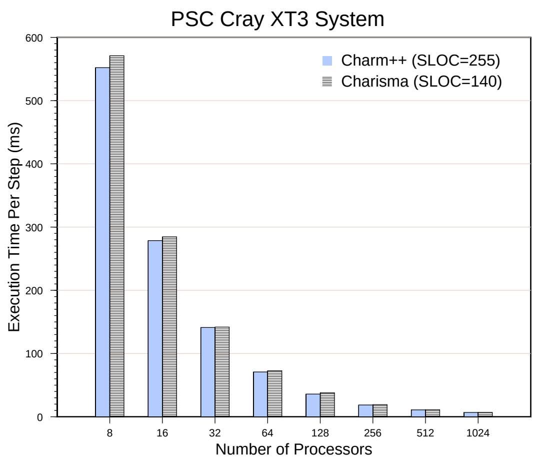
6.2 3D FFT

FFT is frequently used in engineering and scientific computation. Since highly optimized sequential algorithms are available for 1-D FFTs, multi-dimensional FFT containing multiple 1-D FFTs on each dimension can be parallelized with a transpose-based approach [14].

Following is the main body of the orchestration code for the transpose-based algorithm for 3D FFT. From this code segment, Charisma generates the transpose operation between the two planes holding the data. Messages are created and delivered accordingly.

```
foreach x in planes1
  <pencildata[x,*]>:=planes1[x].fft1d();
end-foreach
foreach y in planes2
  planes2[y].fft2d(pencildata[*,y]);
end-foreach
```

Figure 8 compares the performance overhead of runs with problem size of 512^3 on 256 objects, scaling up to 128 processors.



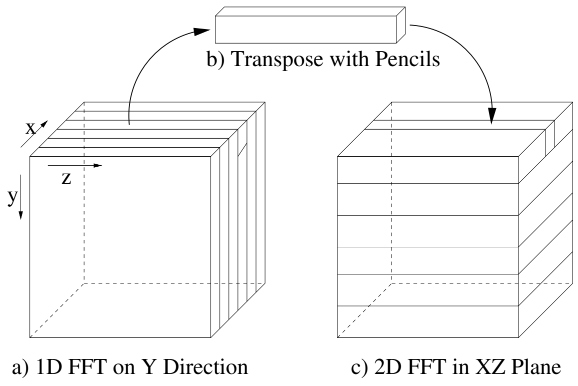


Figure 7. Transpose-based 3D FFT

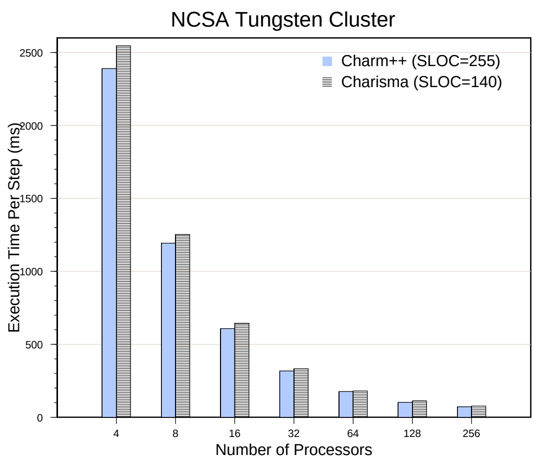


Figure 6. Performance of Stencil Calculation

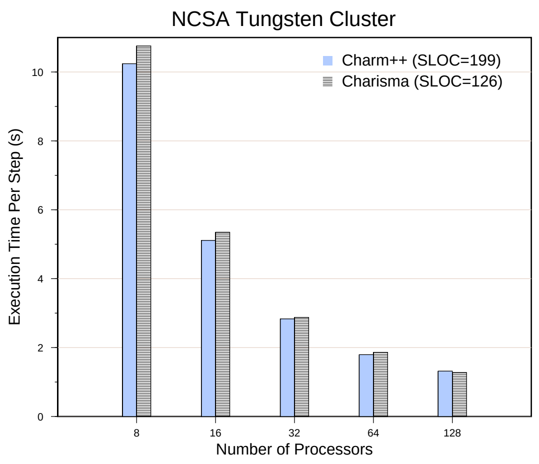


Figure 8. Performance of 3D FFT

From the results, we can see that Charisma in this benchmark incurs up to 5% performance overhead, which can be attributed to additional buffer copy for parameter variables. The reduction on SLOC is only 37%. In this specific benchmark, sequential code dealing with local FFT computation consists of a bigger portion of the program, and therefore the reduction on the SLOC is not as significant as simpler programs. This percentage of SLOC reduction is expected to be even smaller on larger and more complex programs. It must be noted, however, that SLOC alone does not make a good metric of productivity as it does not reflect the actual programming effort. In fact, in more complicated applications, to express parallel flow of control is far more difficult than in simpler cases, and tools such as Charisma can better help programmers code with less effort.

7. Conclusion

We described Charisma, a higher level notation that allows expression of global view of control in parallel programs, while still allowing decomposition into multiple collections of dynamically mapped objects as in Charm++. This approach cleanly separates parallel and sequential code, strongly encourages locality aware programming, allows expression of global flow of control in one place, and still reaps the benefits of runtime optimizations of migratable objects.

The language proposed here does not cover expression of all application patterns, especially the highly asynchronous patterns supported by Charm++. Indeed, it is not even intended to be a complete language. Instead it will be used in conjunction with other paradigms where needed or appropriate. Currently, the orchestration language co-exists with Charm++ modules and mechanisms thus ensuring completeness and high interoperability. Also, our implementation of MPI, the Adaptive MPI (AMPI)[9] also interoperates with Charisma. Beyond these languages, the ability to support modules written in Charisma is crucial for productivity via code reuse. We are designing language features to this end so that we can provide user-level Charisma libraries such as parallel 3D FFT.

Charisma supports a global view of control but a local view of data, since only the object's local variables are accessible in the sequential methods. In contrast, Multi-phase Shared Array

(MSA) [4] supports a global view of data. Integrating the two notations is an interesting future direction to explore.

Last but not least, SLOC is not necessarily a perfect metric for measuring productivity. We plan to conduct classroom experiments among parallel programming students to obtain a more realistic evaluation of Charisma's productivity advantage.

8. References

[1] K. Barker and Nikos Chrisochoides. An Evaluation of a Framework for the Dynamic Load Balancing of Highly Adaptive and Irregular Applications. *Proceedings of IEEE/ACM Supercomputing Conference (SC'03)*, November 2003.

[2] Leonardo Dagum and Ramesh Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science & Engineering*, 5(1), January-March 1998.

[3] T.A. Darden, D.M. York, and L.G. Pedersen. Particle mesh Ewald. An N·log(N) method for Ewald sums in large systems. *JCP*, 98:10089–10092, 1993.

[4] Jayant DeSouza and Laxmikant V. Kalé. MSA: Multiphase specifically shared arrays. In *Proceedings of the 17th International Workshop on Languages and Compilers for Parallel Computing*, West Lafayette, Indiana, USA, September 2004.

[5] Derek L. Eager, Edward D. Lazowska, and John Zahorjan. The Limited Performance Benefits of Migrating Active Processes for Load Sharing. *Conf. on Measurement & Modelling of Comp. Syst., (ACM SIGMETRICS)*, pages 63–72, May 1988.

[6] I. Foster and K.M. Chandy. FORTRAN M: A Language for Modular Parallel Programming. *Journal of Parallel and Distributed Computing*, 25(1), 1995.

[7] G. R. Gao. An efficient hybrid dataflow architecture model. *J. Parallel Distrib. Comput.*, 19(4):293–307, 1993.

[8] Jean-Luc Gaudiot and Liang-Teh Lee. Multiprocessor systems programming in a high-level data-flow language. In *Proceedings of the Parallel Architectures and Languages Europe, Volume I: Parallel Architectures PARLE*, pages 134–151, London, UK, 1987. Springer-Verlag.

[9] Chao Huang, Gengbin Zheng, Sameer Kumar, and Laxmikant V. Kalé. Performance evaluation of adaptive MPI. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming 2006*, March 2006.

[10] H. Jiang and V. Chaudhary. Process/Thread Migration and Checkpointing in Heterogeneous Distributed Systems. In *In Proceedings of the 37th Hawaii International Conference on System Sciences (HiCSS-37)*. IEEE Computer Society, 2004.

[11] L. V. Kale and Attila Gursoy. Modularity, reuse and efficiency with message-driven libraries. In *Proc. 27th Conference on Parallel Processing for Scientific Computing*, pages 738–743, February 1995.

[12] Laxmikant V. Kalé, Sameer Kumar, Gengbin Zheng, and Chee Wai Lee. Scaling molecular dynamics to 3000 processors with projections: A performance analysis case study. In *Terascale Performance Analysis Workshop, International Conference on Computational Science(ICCS)*, Melbourne, Australia, June 2003.

[13] C.H. Koelbel, D.B. Loveman, R.S. Schreiber, G.L. Steele Jr., and M.E. Zosel. *The High Performance Fortran Handbook*. MIT Press, 1994.

[14] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to parallel computing: design and analysis of algorithms*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1994.

[15] Message Passing Interface Forum. MPI: A Message Passing Interface. In *Proceedings of Supercomputing '93*, pages 878–883. IEEE Computer Society Press, 1993.

[16] James C. Phillips, Gengbin Zheng, Sameer Kumar, and Laxmikant V. Kalé. NAMD: Biomolecular simulation on thousands of processors. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–18, Baltimore, MD, September 2002.

[17] Ramkumar V. Vadali, Yan Shi, Sameer Kumar, L. V. Kale, Mark E. Tuckerman, and Glenn J. Martyna. Scalable fine-grained parallelization of plane-wave-based ab initio molecular dynamics for large supercomputers. *Journal of Comptational Chemistry*, 25(16):2006–2022, Oct. 2004.

[18] V.K.Naik, Sanjeev K. Setia, and Mark S. Squillante. Processor allocation in multiprogrammed distributed-memory parallel computer systems. *Journal of Parallel and Distributed Computing*, 1997.

[19] T. von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.