An Orchestration Language for Parallel Objects *

Laxmikant V. Kalé
Department of Computer Science
University of Illinois
kale@cs.uiuc.edu

Mark Hills
Department of Computer Science
University of Illinois
mhills@cs.uiuc.edu

Chao Huang
Department of Computer Science
University of Illinois
chuang10@cs.uiuc.edu

ABSTRACT

Charm++, a parallel object language based on the idea of virtual processors, has attained significant success in efficient parallelization of applications. Requiring the user to only decompose the computation into a large number of objects ("virtual processors" or VPs), Charm++ empowers its intelligent adaptive runtime system to assign and reassign the objects to processors at runtime. This facility is used to optimize execution, including via dynamic load balancing. Having multiple *sets* of VPs for distinct parts of a simulation leads to improved modularity and performance. However, it also tends to obscure the global flow of control: One must look at the code of multiple objects to discern how the sets of objects are orchestrated in a given application. In this paper, we present an orchestration notation that allows expression of Charm++ functionality without its fragmented flow of control.

1. INTRODUCTION

We have been developing an approach to parallel programming that seeks an optimal division of labor between the system and the programmer: the system automates what it can, while the programmer does what they can do better. In particular, our approach has been based on the idea of migratable objects: the programmer decomposes the application into large number of parallel parts (called "virtual processors" or VPs), while the runtime system (RTS) assigns those parts to processors. This gives the RTS the flexibility to migrate VPs among processors to effect load balance and communication optimizations. (See Figure 1)[1]

Charm++ is an early system that embodies this idea: Here each VP is an object (aka chare), and VPs communicate via asynchronous method invocations. Many VPs can be organized into an indexed group, called a "chare-array". A single program may contain multiple chare-arrays. The simplest chare arrays are 1-D dense arrays. Charm++ supports multi-dimensional arrays, sparse arrays (where only a subset of indices are "alive"), as well as collections indexed by bit-vectors.

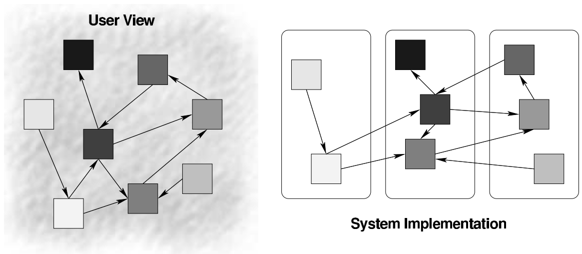


Figure 1: Virtualization in Charm++

Adaptive MPI (also called AMPI) [11] brings the benefits of virtualization to MPI programs. Programs are written using standard MPI, except that each MPI "process" is implemented as a user-level *migratable* thread.

This idea of using indirection in mapping work to processors has been used and developed in the past, including in DRMS [19, 23]. *Olden* [9, 4] did thread migration to move threads to data. Using process migration for load balancing has also been investigated [8]. This broad approach has gained a new momentum recently [1, 12].

A large number of applications have been developed using Charm++. These include NAMD, a production-level Molecular Dynamics program which has demonstrated unprecedented speedups on several thousand processors [2, 17], and CPAIMD[22], a Quantum-Chemistry program. Other examples include Rocket Simulation, Crack propagation, Spacetime meshing with Discontinuous Galerkin solvers, Dentritic Growth in Solidification processes, level-sets methods, computational comsology simulations, parallel visualization of cosmology data, etc.

Although Charm++ has demonstrated its utility in runtime optimizations such as load balancing, and although it is more modular than MPI (see [16]), its expressiveness has a drawback, especially for complex applications that involve multiple sets of virtual processors (i.e. multiple chare-arrays), as seen in the motivational example in next section.

*This work was supported in part by the National Science Foundation (NSF NGS 0103645, NSF ITR 0081307).
[1]Figure taken from [13]

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 the sequential code. This asymmetry often makes it hard to see the parallel structure of an application, which is useful for understanding performance issues.

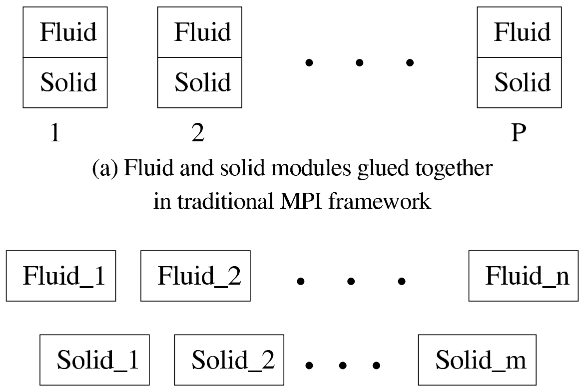
We present a higher-level language notation that retains the benefits of Charm++ while allowing for easy expression of global flow of control as well as symmetric expression of communication. The language separates sequential code fragments (methods) from the parallel constructs. It can be considered a generalization of scripts for the parallel environment. Since it controls the behavior of a collection of virtual processors, we call it an *orchestration language*.

2. MOTIVATION

The language we propose in this paper can be motivated from bottom-up – as an evolution of the concepts developed in Charm++ family of programming abstractions, or from top-down, a from-scratch motivation for a higher level parallel language. The top-down motivation of this project is the need for a language that allows the programmer to express locality of data access and explicit communication, along with global flow of control (a la HPF) and migratable objects. In the following part of the section, we take the bottom-up approach to motivate our language.

Consider as an example, a simplified version of the rocket simulation application we are developing at Illinois: The interior gasses of a burning solid rocket motor is modeled using fluid dynamics whereas the solid fuel (imagine a hollow cylinder) is modeled using structural dynamics. In traditional MPI style, *fluid* and *solid* modules are partitioned across P processors. Since the partitioning program for the solids' unstructured mesh are different than those for the structured grids of fluids, partition `i` of solids is not geometrically or topologically connected with partition `i` of the fluids. Yet, they are glued together in processor i by virtue of them being identically numbered partition! If solid and fluid computations alternate with each other (rather than being run in parallel), you are also required to decompose both domains into equal number (P) of partitions (See Figure 2(a)). In contrast, with Charm++ or AMPI, the fluids and solids code get their own set of virtual processors. Fluids may be implemented by an array of 1000 virtual processors, while solids may need 1,700 VPs (See Figure 2(b)). These numbers are independent of the physical processors being used. (E.g. the above VPs can be distributed to 170 physical processors by the RTS). The RTS's load balancer may decide to assign the i'th VP of Fluids on the same processor as the j'th VP of Solids if they happen to communicate more.

Such independent mapping can lead to significant performance benefit, as the communication volume is reduced. The benefits of measurement-based dynamic load balancing in Charm++ have been demonstrated elsewhere [3, 20]. But even from the point of view of modularity, the separate sets of VPs are beneficial, in general. If the modules can run concurrently, this technique allows idle time in one module being overlapped with useful computation in another, without breaking abstraction boundaries between them [16].



(a) Fluid and solid modules glued together in traditional MPI framework

(b) Fluid and solid modules have their own sets of VPs in Charm++/AMPI framework

Figure 2: Rocket simulation example under traditional MPI vs. Charm++/AMPI framework

However, along with this advantage comes a seemingly small cost. The overall control loop has now disppeared from the program: solid partitions compute when triggered by the data they need, sending messages to other solid and fluid modules, which trigger them to execute. This message driven style requires us to look at implicit transfer of control buried inside all the modules to understand the behavior of the program as a whole. In case of two modules, whether they execute one after the other or concurrently, this expressiveness problem is small. However, as the application becomes complex, the penalty may increase.

Consider the Car-Parrinello algorithm for ab initio molecular dynamics. One major data structure in a particular form of the algorithm[21] is the representation of N electronic states: each state is represented in both real-space and frequency-space, often called G-space, by MxMxM 3D arrays of complex numbers, Ψ_R and Ψ_G. We wanted to parallelize this problem to a number of processors significantly larger than N.

The parallelization of this problem using Charm++ is shown in Figure 3. (It is not important to understand this algorithm completely to see the broad point being made here). Each state's representation is decomposed into M planes. For N=128, and M=100, this decomposition leads to 12,800 VPs in real-space and a somewhat smaller number in G-space (due to its sparsity). The probability density function (ρ_{Real}) and its reciprocal space are each represented by 100 VPs. There are 1,600 VPs in the S-calculator phase for calculating the correlation matrix for orthonormalization. In addition, several VPs are needed for calculations related to the Nuclei.

With this aggressive parallelization, we are able to scale the application to 1500 processors with only 128 states (with a speedup of about 550)[18]. This unprecedented scalability for this application attests to the success of processor virtualization. Expressing such an algorithm with MPI would have been difficult, especially since it involves multiple (overlapping) collective operations in the transpose phases (I and

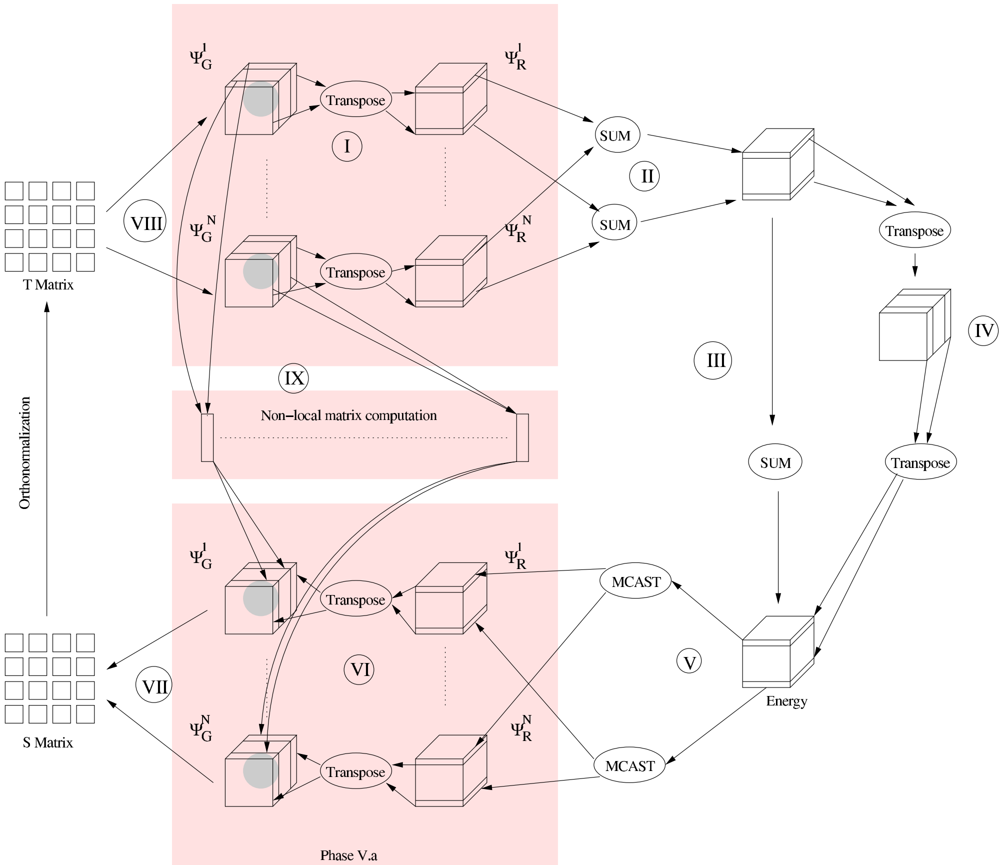


Figure 3: Parallel structure of CPMD implementation using virtual processors

VI). Charm++ separates the issue of resource management, and in that respect, it simplifies programming.

However, the Charm++ code for this application is spread across several C++ classes. The real-space component Ψ_R, for example, only specifies that when it gets all the rows in its plane, it computes a 2D FFT (Ph. I) and sends the result into a non-blocking reduction (II); and *when* it gets a new plane back (in Ph. V via a multicast) it does an inverse 2D FFT and sends rows to the corresponding G-space rows. *The overall global flow of control is not represented directly, and must be inferred by readers.* This makes the code difficult to understand and modify.

Does this disdvantage of inability to clearly express global flow of control so serious that we should go back to programming with plain MPI? Clearly not. The advantage of virtual processors (or migratable objects) methodology are too numerous and too signficant for that [13]. They include dynamic load balancing[15], adaptive overlap of comunication and computation[13], easier automatic checkpoint and restart[10], faster fault tolerance on extremly large parallel machines[24, 5], responding to predicted faults on individual processors by migrating objects away, ability to shrink and expand the sets of processors assigned to a job at runtime, running tightly coupled applications across geographically separated clusters (by taking advantage of the latency tolerance created by a large number of VPs), and so on. Instead, we aim at raising the level of abstraction within the Charm++ framework of processor virtualization via something akin to a scripting language that specifies the global flow of control explicitly without giving up the idea of multiple sets of virtual processors. Since the script is orchestrating the behavior of a large number of objects of different types, we call this an orchestration language. Although the focus of this paper is on global flow of control, we also incorporate our recent work on providing a limited and disciplined use of global arrays in the form of multiphase shared arrays [7], as described in Section 3.

3. ORCHESTRATION LANGUAGE

The orchestration language enables the programmer to have a global flow of control. A typical orchestration program has two components: the script-like orchestration code specifying the global control flow, and the sequential code provided by the programmer. The orchestration code will then be translated into target language, and the user code will be integrated.

The skeleton of the orchestration code is the orchestration statements. The programmer put together the structure of the control flow and corresponding code will be generated by our translator. In this section we give detailed description of the orchestration language design, including important orchestration statements and communication patterns.

3.1 Chare Array Creation

Our language is based on the idea of orchestration of parallel virtual processors. When programming in the orchestration language, the programmer first creates one or more arrays of virtual processors (called *chare arrays*) and describes the behavior of and interaction among the elements of the arrays. A "chare" in Charm++ is an object which includes methods that can be remotely invoked. Although they are called "arrays", the chare-arrays are a collection of chares, indexed by very general mechanisms. In particular, the chares (objects) can be organized into 1 or multi-dimensional arrays that can be sparse, and also into collections indexed by arbitrary bit-patterns or strings. One can also dynamically delete and insert elements in a chare array. The Charm++ run-time system is responsible for adaptively mapping the chare array elements onto available physical processors efficiently.

A chare array is constructed using elements that are declared as C++ classes inheriting from an appropriate chare-array-element type (e.g. ChareArray1D for 1-dimensional chare arrays). The class methods (defined in separate files) specify the behavior of individual chares; however, the size of the array is specified in the orchestration language. Sample code for creating a chare array `myWorkers` follows.

The **classes** section declares one or more *types* of chare-arrays. The private variables and methods of this class are declared in a separate file. The **vars** section specifies instances of chare-arrays. There can be mutlple chare arrays created with the same chare-array class. (E.g. one could have myWokers : MyArrayTpe[100]; in addition to the one shown below).

```
classes
    MyArrayType : ChareArray1D;
    Pairs : ChareArray2D;
end-classes
vars
    myWorkers : MyArrayType[1000];
    myPairs : Pairs[8][8];
    otherPairs: Pairs[2][2];
end-vars
```

3.2 Orchestration Statements

The orchestration language consists of a forall statement that specifies parallelism across all chares of chareArray (or its subset), and an overlap statement that specifies concurrency among multiple sections of foralls. Communication between objects is specified using two main mechanisms: Multiphase shared array accesses and parameters prodcued and consumed by chare methods.

3.2.1 Forall

The programs covered by the orchestration language operate in a loosely data-parallel manner. This is specified by a simple `forall` statement. As an example, the following code segment invokes the entry method `doWork` on all the elements of array `myWorkers`.

```
forall i in myWorkers
    myWorkers[i].doWork(1,1000);
end-forall
```

One can specify a subset of elements in chareArray as:

```
forall i:0:10:2 in myWorkers
```

When we span all elements of an chareArray as in the first statement above, we can ellide the description as:

```
forall myWorkers
    doWork(1,1000);
end-forall
```

The `forall` statement may look somewhat like the `FORALL` statement in other parallel programming languages such as HPF, but the distinction between our language and HPF, especially between the `forall` in orchestration language and `FORALL` in HPF is very important in order to understand our language. In a language like HPF, global data arrays are created and partitioned by user-specified directives, and used by a fixed set of processors executing identical codes. In contrast, in our language, we can have multiple sets of migratable objects (chare arrays), and both local view of data within the object's scope and limited global view of data in the form of imput/output parameters of class methods. In HPF, `FORALL` provides a parallel mechanism to assign values to the elements of a data array, whereas the `forall` statement in orchestration language not only specifies the parallelism among parallel objects, but also allows for object-level distributed control. The code in the `forall` block is more than parallel data access; the distributed control makes powerful optimization possible.

3.2.2 Overlap

`Overlap` statements specifies concurrent flow of control. For example, in the following code segment, the execution of the two `forall` statements can be overlapped at run time.

```
overlap
  forall i in workers1
    workers1[i].doSomeWork(1,1000);
  end-forall
  forall i in workers2
    workers2[i].doOtherWork(1,100);
  end-forall
end-overlap
```

The distributed object-level control and the `overlap` statement exposes the opportunity for more efficient run-time scheduling of the execution of entry methods on parallel objects. With the help of dependence analysis between the statements and appropriate use of communication patterns, we can achieve high parallel efficiency by eliminating unnecessary synchronization without losing the productivity of programming.

3.3 Communication Patterns

Before we describe the various communication patterns in orchestration language, we explain the input and output of an object method invocation, and the communication patterns are expressed via the input and output of method invocations of different objects.

3.3.1 Input/Output of Object Method Invocations

Here is a statement to exemplify the input and output of a method.

```
forall i in workers
  <..,q[i],..> := workers[i].f(..,p[e(i)],..);
end-forall
```

Here, q is one of the "values" produced by the method f, while p is one of the values consumed by it. The values produced by A[i] must have the index i, whereas e(i) is an affine index expression (such as 3*i+4). Although we have used different symbols (p and q) for the input and output parameters, they are allowed to overlap. The parameters are global data items or data arrays supported with a restricted shared-memory abstraction in the framework.

3.3.2 Point-to-point Communication

We now introduce a mechanism to allow point-to-point communication among objects. First we give a statement to exemplify the input and output of a method.

Now we can specify point-to-point communication via the parameters. For example, in the code fragment below, p[i] is communicated (sent as a message, or asynchronous-method-invocation) between elements of chare-array A and B: A[i] produces p[i] and sends it to another element of B, at the method g.

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

How is the value `p[i]` produced by `A[i].f` used by `B[i].g`? We provide a primitive called `publish` for this purpose. Inside `f`, one can call `publish_f(0,p)` to signify creation of the 0'th output value, assuming the type of a matches that of p[i]. By this mechanism, we avoid using any global data and reduce potential synchronization overhead with data dependence analysis. For example, in the code segment above, B[2].g does not have to wait on all A[i].f to complete to start its execution; as soon as A[2].f is completed and the value p[2] is filled, B[2].g can be invoked.

3.3.3 Multicast

A value produced by a single statement may be consumed by multiple chare-array elements. For example in the following code, p[i] is consumed by 3 elements of B: B[i-1], B[i] and B[i+1], ignoring the boundary case wrap-around for simplicity. This mechanism thus implicitly specifies a multicast of produced values (admittedly of a small degree of 3 here).

```
<p[i]> := A[i].f(...);
<...>  := B[i].g(p[i-1], p[i], p[i+1]);
```

3.3.4 Reduction

A reduction is specified by adding the symbol "+" before an output parameter:

```
< ..  , +e, ..  > := A[i].f(..);
```

The dimensionality of the reduced output parameter must be a subset of that of the array element producing it. Thus

```
< r[i] ..  > := B[i,j].g(..);
```

is allowed because i is a subset of the set of indices of A used (namely [i,j]).

3.3.5 All-To-All

Sometimes, a single method invocation produces a number of output values: you can think of them as an array of output values. This can be specified by the syntax exemplified by:

```
<p[i,j:0:5]> := A[i].f(..);
```

Here, 6 different values are produced by each A[i]. This syntax is especially useful for specifying all-to-all communication and transposes, as illustrated by the 3D FFT example below.

Here a 3-dimensional array of numbers has been decomposed into a 1-dimensional array of chares, each holiding a plane (2-D array of numbers). The `A` array elements perform several 1-D line FFTs along one dimension, and then transpose the data into the `B` array (as shown in Figure 3). The `2Dforward` method of `B` completes the 3D FFT by carrying out a 2D FFT on all of its data. (Only forward part of the FFT is shown: the inverse FFT is similar.)

```
begin
  forall i in A
      <rows[i,j:0:N-1]> := A[i].1Dforward();
  end-forall
  forall k in B
      ... := B[k].2Dforward(rows[l:0:N-1, k]);
  end-forall
end
```

3.4 Example Code

In this section we give example code of three typical parallel programs: Jacobi (stencil computation), parallel prefix, and a simple molecular dynamics computation. Through these examples, we illustrate the use of various statements and parallel computation patterns in orchestration language. Shown here is the orchestration language code, and the user completes the program with function definition in separate file(s).

3.4.1 Jacobi

In the following example of a Jacobi computation, a reduction is used to compute the error value. This error value is then used in the loop control to determine how long to continue the computation. In the definition of `compute` function, the programmer will specify the reduction operation on `e` and the publish destination of the function output.

```
begin
  forall i in J
    <lb[i],rb[i]> := J.init();
  end-forall
  while (e > threshold)
    forall i in J
      <+e, lb[i], rb[i]> :=
        J[i].compute(rb[i-1],lb[i+1]);
    end-forall
  end-while
end
```

3.4.2 Parallel Prefix

The following program of parallel prefix introduces a few additional features of the language.

```
#define N 10
P : PrefixElement[N];
k : int; k = 1;
begin
  forall i in P
      <s[i]> := P[i].init();
  end-forall
  while ( k<N )
      forall i in P: (i>=k)
          <s[i]> := P[i].step(s[i-k]);
          k = 2*k;
      end-forall
  end-while
end
```

The language has its own scalars (k), and constants (N), and normal C++/Java style assignment statements (`k = 2*k;`). Further, the forall clause may be qualified by a guard (here; `i>=k`), since for P[i] with `i<=k`, there are no incoming s values).

Note that value consumed by P[i] must be that produced in the immediately preceding iteration. In other words, there is an implicit barrier at the end of each `forall`. Although the semantics is defined by assuming such a barrier, the implementation does not *need* to use a barrier for the purpose of ensuring the semantics: it is free to use other methods (such as tagging the messages with iteration-number and buffering them as needed, for this example). In cases when the programmer knows there should be no barrier, they can force that by calling `noSynch()` after a `forall`.

3.4.3 Molecular Dynamics

As a final example, we show a NAMD-style parallelization of molecular dynamics (MD). The atoms are partitioned into 3-D chare-array of cells. For each adjacent pair of cells, there is a `cellPairs` object for computing electrostatic forces between atoms. (Thus `cellPairs` is a 6-D sparse array). The coordinates are multicast to cell-pairs, while the forces are added up via a reduction.

```
begin
  forall i,j,k in cells
    <coords[i,j,k]> := cells[i,j,k].init();
  end-forall
  for timestep = 0 to 100000
    forall i,j,k,m,n,p in cellPairs
      <+forces[i,j,k], +forces[l,m,n]> :=
        cellPairs[i,j,k,m,n,p].computeForces(
          coords[i,j,k],
          coords[m,n,p]]);
    end-forall
    forall i,j,k in cells
      <coords[i,j,k]> :=
        cells[i,j,k].integrate(forces[i,j,k]);
    end-forall
  end-for
end
```

4. IMPLEMENTATION

Implementing the orchestration language is made easier by the "back-end" provided by Charm++, which supports migratable objects and asynchronous method invocations. A full-scale implementation requires a translator capable of static analysis to identify the targets of each producer-consumer communication indicated by our syntax. For a first implementation, which allows us to experiment with the language, we have taken a simpler approach: we have developed a simple translator that can parse the language and generate code, but does not perform significant static analysis to identify and generate point-to-point communication, multicasts, or reductions. Instead, it use a slightly inefficient but highly flexible mechanism provided by our implementation of limited-access global arrays, called multiphase shared arrays (MSA). Further, instead of generating Charm++ code, our current implementation of the orchestration language targets Jade[6], Charm++-like parallel language with a Java-like sequential syntax. We first introduce Jade and MSA, then discuss the currently implementation and future plans.

4.1 Introduction to Jade and MSA

Jade is a Java-like language that supports parallel classes with properties and methods. The parallel classes in Jade are parallel objects called *Chares* and parallel arrays of objects called *ChareArrays*. Communication with a parallel object occurs through asynchronous method invocation. In the message-driven paradigm, when a method of a Chare or ChareArray is invoked, it continues to completion before any other method of the same parallel object can run.

In contrast to Java's run-time compilation, the Jade source code is translated to Charm++ source code, which is then compiled and executed on the target machine. The resulting code supports object migration and load-balancing and scales well to large number of processors. Java's standard libraries are not supported.

The reason we chose Jade as the target language for the first version is its simplicity. For example, there is no pointers or pointer arithmetic in Java.

Our initial implementation uses multi-phase shared arrays (MSA) [7]. MSA provides a restricted shared-memory abstraction which can be used to share arrays of arbitrary type. Shared data is stored into pages which have their home processors, Local copies is allowed for the consumer of any particular page, and a synchronization step is required at the end of each phase. An MSA is accessed in either a read, write, or accumulate mode. Access is divided into phases, with all chares in the same phase accessing an array in the same mode; phases are demarcated using sync operations on the array. The phases are used to improve performance and allow the solution to be more scalable. For instance, data only needs to be fetched before being read, and data only needs to be made consistent at the end of a write or accumulate phase (instead of throughout the phase) since it cannot be read in that phase. The accessing pattern of an MSA may look like:

```
/* in parallel code */
MSA1D<...> arr;  // initialization
arr[i] = x;      // write-by-one mode
arr.sync();      // change phase
compute(arr[i-1], arr[i+1]); // read mode
arr.sync();      // change phase
```

. . .

This multi-phase minimizes fetch and consistency traffic, which is a major difference from other shared memory abstraction like Global Array. For instance, in many-write phase, one page may have different part of data written by different parallel objects, and the data will be merged at the sync toward the end of this phase. In GA this might incur large amount of consistency traffic.

4.2 Current Implementation

A complete orchestration language program consists of two components: a file containing orchestration language code that declares the parallel objects and orchestrates their interactions, and sequential code for the methods of their parallel classes.

The parallel objects like chare arrays declared in orchestration code by the programmer are translated to Jade classes. The creation and initialization code for the chare arrays as well as for the MSAs will also be generated. The orchestration will be translated into distributed control specification for each chare array. For example, the block of code in a `forall` statement in orchestration language will be transformed to threaded method of corresponding chare array classes and distributed down to each individual chare array element, thus allowing for adaptive optimization.

The programmer is supposed to write code for detailed method including computations, input and output, etc. The code is later integrated into the target language code, currently Jade. The invocation of user methods will be controlled by the orchestration program. This way the orchestration code is simplified and the division of labor between the programmer and the run time system is clear too. We believe such optimal division of labor is a key to high productivity in parallel programming.

Current implementation uses MSAs to store and retrieve the produced and consumed parameters. In orchestration code, the programmer first declares MSAs to hold the data to be processed. Methods producing the data write to MSAs and methods consuming the data read from MSAs, in different access phases. In the orchestration language, arrays are defined in the orchestration source file in an arrays section. Code generated by the orchestration compiler creates the arrays in the main chare and initializes them in all other chares that will use them. The user can then access the arrays in the user defined methods.

5. SUMMARY AND FUTURE WORK

We described a higher level notation that allows specification of parallel algorithms without specifying placement of data and control. The data resides in indexed collections of objects called chare-arrays. The control is specified at a higher level by an notation that includes a *forall* statement looping over all objects of a collection, and sequencing and concurrent overlapping of such statements. The body of such statements only invoke methods inside the objects. The code inside the methods provides low-level sequential flow of control. Further, in a pure version of the language, such sequential code is allowed to access only local data — instance variable of their object. (Additionally, global read-only data is supported as in Charm++). The only communication supported by this language is producer-consumer communication: each method may produce values that are consumed by other method invocations.

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 has been implemented via a simple translator, using multi-phase shared arrays and Charm++/Jade as the back-end. However, clearly, MSAs are an inefficient mechanism because they require each data item to travel twice: from the producer to the MSA (home page) and from MSA to the consumer (via an additional request message). Simple optimizations to MSAs can be used to store the page where it is produced; however, our long-term plan is to use compiler analysis to infer the destinations where each produced value is needed, and send it via a single message, implemented as Charm++ asynchronous method invocation. Further compiler optimizations include inferring when and how to omit barriers. MSA will continue to serve as a programming paradigm in our framework, especially when a global view of data is desired.

When the method of an object produces a value, the current implementation waits until the method returns to send the value to its destination. This is not necessary. A more efficient method of exchanging messages is to use a publish-connect style of message exchange, such as that currently available in Charm++ with Structured Dagger [14]. The code generated by the orchestration language inserts callbacks for the produced values before calling a method. When the user code inside a method invokes the appropriate publish methods within their code, the data will be sent to the appropriate recipients using the callback specified. This will ensure that published values are sent to their destinations without waiting for the method that produced them to finish.

The language proposed here does not cover expression of all application patterns, especially the highly asynchronous patterns supported by Charm++. Further, 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 coexist with Charm++ modules and mechanisms thus ensuring completeness and high interoperability. Also, our implementation of MPI, the Adaptive MPI (AMPI) can be used inside an object method with little effort.

We plan to investigate several ways of extending, enhancing and leveraging this language. For example, we plan to support *implicit methods*, so that a forall can include object code in its body directly: If all methods are implicit, then we can support languages other than C++ with this syntax.

```
forall i in W {
consumes(p[i+1], p[i-1])
produces(q[i]);
```

```
.... user code in C/C++/F90 ...
}
```

The language described here supports a global view of control but local view of data, since only object's own variables are accessible. In contrast, MSA (which we described as an initial implementation vehicle) supports a local view of control and global view of data. Integrating the two notations is an interesting future work. An initial approach is suggested by the implicit methods above. The implicit method code can include MSA array-access calls, and array-specific sync() calls, as needed.

Ability to support modules written in this language is crucial for productivity via code reuse. We plan to design and implement language features to this end. We plan to integrate user-level libraries such as parallel high-dimensional FFTs in the framework. We expect further exploration of this model to lead to significant improvements in productivity. Other planned extensions include support for sparse arrays, arrays with index of user-defined type, and other useful array organizations such as trees.

6. REFERENCES

[1] K. Barker and N. 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] R. Brunner, J. Phillips, and L.V.Kalé. Scalable molecular dynamics for large biomolecular systems. In *Proceedings of SuperComputing 2000*, 2000.

[3] R. K. Brunner, J. C. Phillips, and L. V. Kale. Scalable Molecular Dynamics for Large Biomolecular Systems. In *Proceedings of Supercomputing (SC) 2000, Dallas, TX, November 2000. Nominated for Gordon Bell Award.*, November 2000.

[4] M. C. Carlisle and A. Rogers. Software caching and computation migration in olden. In *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 29–38. ACM Press, 1995.

[5] S. Chakravorty and L. V. Kale. A fault tolerant protocol for massively parallel machines. In *FTPDS Workshop for IPDPS 2004*. IEEE Press, 2004.

[6] J. DeSouza and L. V. Kalé. Jade: A parallel message-driven Java. In *Proc. Workshop on Java in Computational Science, held in conjunction with the International Conference on Computational Science (ICCS 2003)*, Melbourne, Australia and Saint Petersburg, Russian Federation, June 2003.

[7] J. DeSouza and L. 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.

[8] D. L. Eager, E. D. Lazowska, and J. 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.

[9] W. C. Hsieh, P. Wang, and W. E. Weihl. Computation migration: enhancing locality for distributed-memory parallel systems. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 239–248. ACM Press, 1993.

[10] C. Huang. System support for checkpoint and restart of charm++ and ampi applications. Master's thesis, Dept. of Computer Science, University of Illinois, 2004. http://charm.cs.uiuc.edu/papers/CheckpointThesis.html.

[11] C. Huang, O. Lawlor, and L. V. Kalé. Adaptive MPI. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 03)*, College Station, Texas, October 2003.

[12] 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.

[13] L. V. Kalé. Performance and productivity in parallel programming via processor virtualization. In *Proc. of the First Intl. Workshop on Productivity and Performance in High-End Computing (at HPCA 10)*, Madrid, Spain, February 2004.

[14] L. V. Kale and M. Bhandarkar. Structured Dagger: A Coordination Language for Message-Driven Programming. In *Proceedings of Second International Euro-Par Conference*, volume 1123-1124 of *Lecture Notes in Computer Science*, pages 646–653, September 1996.

[15] L. V. Kale, M. Bhandarkar, and R. Brunner. Run-time Support for Adaptive Load Balancing. In J. Rolim, editor, *Lecture Notes in Computer Science, Proceedings of 4th Workshop on Runtime Systems for Parallel Programming (RTSPP) Cancun - Mexico*, volume 1800, pages 1152–1159, March 2000.

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

[17] L. V. Kalé, S. Kumar, G. Zheng, and C. W. 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.

[18] S. Kumar, Y. Shi, R. V. Vadali, L. V. Kalé, M. E. Tuckerman, and G. J. Martyna. Scalable Parallelization of Ad Initio Molecular Dynamics. *to be published in Journal of Computational Chemistry*, 2004.

[19] J. E. Moreira and V.K.Naik. Dynamic resource management on distributed systems using reconfigurable applications. *IBM Journal of Research and Development*, 41(3):303, 1997.

[20] J. C. Phillips, G. Zheng, S. Kumar, and L. V. Kalé. NAMD: Biomolecular simulation on thousands of processors. In *Proceedings of SC 2002*, Baltimore, MD, September 2002.

[21] M. Tuckerman, D. Yarne, S. Samuelson, A. Hughes, and G. Martyna. *Comp. Phys. Comm.*, 128:333, (2000).

[22] R. Vadali, L. V. Kale, G. Martyna, and M. Tuckerman. Scalable parallelization of ab initio molecular dynamics. Technical report, UIUC, Dept. of Computer Science, 2003.

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

[24] G. Zheng, L. Shi, and L. V. Kalé. Ftc-charm++: An in-memory checkpoint-based fault tolerant runtime for charm++ and mpi. In *2004 IEEE International Conference on Cluster Computing*, San Diego, CA, September 2004.