NAMD: A Case Study in Multilingual Parallel Programming*

L. V. Kalé, M. Bhandarkar, R. Brunner, N. Krawetz, J. Phillips, and A. Shinozaki

Dept. of Computer Science, and
Theoretical Biophysics Group, Beckman Institute,
University of Illinois,
Urbana Illinois 61801,
{kale,milind,brunner,nealk,jim,ari}@ks.uiuc.edu

Abstract. Parallel languages are tools for constructing efficient application programs, while reducing the required labor. In this light, using the most appropriate tool for each component of a complex system seems natural, resulting in multi-paradigm multilingual programming. The Converse system developed at Illinois addresses the issues involved in supporting multilingual applications. This paper describes the development of a large parallel application in Computational Biophysics from the point of view of multilingual programming. NAMD, a molecular dynamics program, is implemented using three different "paradigms": Parallel message-driven objects, Message-Passing, and Multithreading. The issues faced in implementing such a system, and the advantages of multilingual approach are discussed. NAMD is already operational on many parallel machines. Some preliminary performance results are presented and the lessons learned from this experience are discussed.

1 Introduction

Parallel programming complexity has fueled development of several parallel languages in recent years. Each new parallel language is designed to improve the productivity of parallel programmers, by providing features and incorporating techniques that simplify the task of developing parallel applications, at least for certain classes of parallel programs. However, new parallel languages have not been widely used by application programmers. Most successful parallel applications use programming paradigms that were available more than a decade ago, such as message passing similar to original tag-based messaging in NXLIB, now with a standardized interface such as MPI. Alternatively, some programs are written using the traditional shared-memory paradigm; more recently, a few

* This work was supported in part by National Institutes of Health (NIH PHS 5 P41 RR05969-04 and NIH HL 16059) and National Science Foundation (NSF/GCAG BIR 93-18159 and NSF BIR 94-23827EQ). In addition, J. Phillips was supported by Computational Science Graduate Fellowship from United States Department of Energy.

programs using HPF or vendor-supplied parallelizing FORTRAN compilers have also been developed.

One of the reasons that the new parallel languages [2] are not being utilized to a larger extent is the perceived difficulty of programming a large application entirely in a new language. Application programmers are concerned about the risks involved in committing large amounts of programming resources into a new, "unproved" language. Also, often they prefer to reuse existing parallel libraries written using an older, traditional language. The new parallel languages often appear idiosyncratic, with unusual syntax or parallel models. Having the programming team reach a consensus on switching to such an unfamiliar new language is difficult. Also, although a new programming language may be effective for programming certain algorithms, it may not be effective for other kinds of algorithms.

Multilingual programming is an approach that addresses the above issues. The basic idea is to provide a framework in which modules written in multiple parallel languages can be linked together in a single application. For certain pairs of languages, such as PVM and HPF, such coexistence is easy. Problems arise when the languages involved have different models of scheduling the processor or different runtime models. If such problems are solved, the resultant framework will act as a catalyst for incorporation of new parallel languages. Libraries written in new languages can be incrementally incorporated or can be used in an application program written largely in traditional languages. One can choose to use a language more suited to implementing algorithms in individual modules. As programmers develop familiarity with new languages in this incremental, non-threatening manner, they can be expected to experiment further in using such languages themselves. From the parallel language designer's point of view, a level playing field is now created, because the initial hurdle of making the entire programming team to switch to a new programming language is removed.

We have been developing such a multilingual framework called *Converse* [6] for the last few years. Converse already supports several parallel languages, such as Charm++, PVM, MPI etc. The first large multilingual application using the Converse framework, a parallel molecular dynamics program named NAMD, was recently completed.

In this paper, we describe the utility of multilingual approach for development of NAMD. The next section describes our earlier PVM-based development efforts for NAMD, and lists its limitations in modifiability and performance enhancements. Section 3 describes Converse, which provides an interoperable framework

[2] It is important to keep in mind what we mean by a parallel language here. Normally, a new parallel language may have its own syntax, a compiler, and runtime system. However, many parallel languages that have evolved are library-based. Yet, they represent different parallel programming paradigms, and for the sake of brevity, are called "languages" in this paper. This is also consistent with the usage in the literature. For example, "Linda" [3] consists of 4 major calls in a runtime library. A Linda program is written in a conventional sequential language such as C or Fortran, and includes calls to the Linda runtime. Yet, such a parallel program represents a paradigm that is dramatically different than programming using MPI, for example.

for the languages and paradigms used to build NAMD. The languages used for NAMD implementation are described in section 4. Section 5 describes the design of key NAMD components in detail and illustrates the use of multilingual modules. Preliminary performance results are given in section 6.

2 Previous Work: PVM-based NAMD

NAMD is a major component of MDSCOPE [13], being developed as a part of the interdisciplinary project on biomolecular modeling led by Prof. K. Schulten, Prof. L.V. Kale, and Prof. R.D. Skeel.

NAMD models the behavior of a molecule by simulating the motions of the constituent atoms in a small region of its environment. Each atom is acted upon by forces exerted by other atoms. The total force on each atom is computed every timestep, and the Newtonian equations of motion produce the resulting motion of the atoms for each simulation timestep. The forces acting on the atoms are divided into two categories: Bonded and Non-bonded. Bonded forces represent the chemical bonds which hold the molecule together. They are modeled as springs connecting an atom with a few nearby atoms. Non-bonded forces represent the Coulomb and van der Waals' forces between each pair of atoms.

Computation of non-bonded forces between all pairs of atoms consumes the majority of the simulation time. Fortunately, two factors reduce the computational costs of calculating these forces. First, the forces are inversely proportional to the square of distance between atoms. Therefore, in many cases it is safe to ignore the negligible forces exerted by atoms outside a specified *cutoff radius*. Second, Newton's third law states that the force exerted on one atom in the pair has the same magnitude and opposite sign as that exerted by the atom. Taking advantage of this often requires additional communication and programming complexity, but it cuts the computation time in half.

NAMD uses spatial decomposition to parallelize the simulation. The space occupied by the molecule is divided into regular cubic regions called *patches*. Each patch is responsible for updating the positions of all atoms contained in its region of space. The patches are typically slightly larger than the cutoff radius. Therefore, all the non-bonded forces for the atoms in the patch can be computed by knowing only the positions of other atoms in the patch, and the atom positions of the 26 neighboring patches. Each processor is responsible for processing an arbitrary number of patches.

Although the cutoff approximation is valid in many cases, some simulations require more accurate computation of the non-bonded forces. This necessitates all-to-all force computations. However, a computationally cheaper alternative exists, and is based on the observation that, even when the forces due to distant atoms (long-range forces) are not negligible, they still vary relatively slowly. Therefore, they can be computed periodically, rather than during every timestep, and the "stale" long-range force values are combined with current short-range forces every timestep. This technique is called dual timestepping.

Even when dual timestepping allows long-range non-bonded force computations every k simulation steps, the long-range forces are still a major component of the computation time. A number of researchers have worked on more efficient N-body solvers for both chemical and astronomical simulations. For NAMD, we chose an efficient implementation of the fast multipole algorithm in DPMTA [14] library developed by researchers at Duke University.

NAMD was originally designed as a message-driven program [4]. The complex dependencies between each patch and its 26 neighbors made the overlapping of communication and computation provided by a message-driven design attractive. Charm++ [9] was a language well-suited to this design, and NAMD was written from the outset as a Charm++ program.

Early in development, it was decided that NAMD would use the DPMTA library when a simulation demanded long-range electrostatic computation. Unfortunately, DPMTA was written in PVM, and, at the time, Charm++ could not coexist with any other parallel languages. As a result, two versions of NAMD were created, one which worked with PVM, and another which used Charm++. The PVM requirement affected program design, since many of the features of Charm++, such as asynchronous reductions, did not exist under PVM. The requirement of maintaining two versions was also a great burden. Since DPMTA made the PVM-based NAMD the more capable program, it received the bulk of the improvement efforts, and the Charm++ version gradually became obsolete. The resulting program was a partially message-driven design built on a tag-based message-passing language, and the underlying algorithms are far less clear than would be exhibited by a purely message-driven program.

During the simulation, the patches have a normal cycle of (1) sending positions to neighbor patches, (2) receiving neighbor positions, (3) computing the forces between the local atoms and the neighbor atoms, (4) returning the forces to the neighbor patches (to take advantage of Newton's third law), and (5) integrating the equations of motion to produce updated positions. Meanwhile, as the normal patch cycle proceeds, reductions are being done to collect energy values, positions, and/or velocities for output and for certain integration methods. VMD [5], the visualization component of MDSCOPE uses this data to render the molecule in every timestep. The main patch loop may or may not need to wait for the results of these reductions after each step, depending on the parameters of the simulation.

The main patch loop was originally written in a message-driven style on top of PVM. The reductions were added later in the development, and did not follow the message-driven paradigm. This was mainly due to the difficulties in modifying the original patch loop to accommodate additional message tags, with different types of destination objects. What emerged was a nested loop design, where patch messages are processed in an inner loop, and when no patch-related messages are found, the outer loop processes any outstanding reduction messages. When we added integration schemes requiring results from the reductions, some of the patch activities had to be moved out of the message-driven loop, adding unnecessary synchronization to the simulation.

The resulting implementation had a number of problems. First, the PVM based implementation could not take advantage of the Charm++ based performance analysis tools developed over the last several years [10]. Secondly, it was very difficult to modify, since any features requiring communication require careful considerations of the implicit dependencies in the message loops. Also, the order of tag-based receives enforced a priority on the messages, rather than a more versatile scheme where the priorities of the message and the type of the messages are unrelated. Finally, the loops produced unintended and unnecessary data dependencies just by the location of the message receive calls in the loop. This resulted in an unnecessary loss in performance, since the dependencies were artifacts of the implementation, and not of the algorithms.

A multilingual approach was used to solve these problems. The Converse framework that facilitated this approach and the specific parallel languages used are described in the next sections.

3 Converse

Converse is designed with a two main objectives:

1. Provide a framework that supports multilingual parallel programming
2. Provide an infrastructure that simplifies implementation of new parallel languages.

Converse achieves these goals by using (a) a common scheduler's queue for entities across languages, (b) by providing a customizable scheduler that is exposed to language runtimes, and (c) by providing a component-based architecture with support for many common building blocks of parallel runtimes.

A layered approach to parallel runtime often requires all the languages to pay the overhead of the most general design. In contrast, Converse uses a component-based architecture. A specific language implementation in Converse framework incorporates only the necessary components, customizing them as needed thus causing the minimum overhead. The components provided by Converse include a machine interface (that supports communication, timers and other operating system calls), scheduler queues, a threads package, a message manager, and a load-balancing package.

Converse machine interface (CMI) provides calls to send and broadcast messages both synchronously and asynchronously. There is no explicit receive call. However, the scheduler module provides calls to process messages until a message directed at a particular handler arrives. This has been successfully used to implement SPMD-style languages such as MPI and PVM.

Converse threads package modularly separates the threads abstraction from other aspects of the common threads packages, such as thread scheduling and synchronization. It allows asynchronous creation of threads, and ability to attach different schedulers to threads, thus facilitating hierarchical scheduling.

A message manager acts as a container for incoming messages. Language runtimes making use of this module (such as SM, a simple messaging language), can use tagged messages and can block on specific tagged messages while allowing other modules to proceed.

Converse also provides a general model for dynamic load-balancing. A piece of work represented by a message, to be dynamically placed on (possibly) the least loaded processor, is handed over to the runtime system as a *seed*. The runtime system then moves this seed from processor to processor until the least loaded processor accepts it. Language implementors can easily provide their own load measuring and load-balancing strategies using the calls provided in this component of Converse.

Converse supports a machine model that consists of multiple nodes, where each node may include one or more processes. The processes on a node share an address space. This machine model ideally fits a cluster of SMPs, which seems to be emerging as a dominant architecture. Also, the machine model subsumes a pure shared memory machine such as an SMP PC, or a pure distributed memory machine such as IBM SP2. To provide portable implementation across a wide variety of parallel machines, Converse provides support for declaring and using Node-level and processor-level variables. Converse macros translate these primitives to the declarations required on each individual machines.

Converse has been implemented on several parallel machines including workstation clusters, IBM SP2, Convex Exemplar, SGI Origin 2000, Cray T3E, Intel Paragon, etc. Porting Converse to a new machine is relatively easy because only a small machine dependent component needs to be rewritten for each machine.

A number of languages based on different paradigms have been implemented on top of Converse. Charm and Charm++, based on message-driven objects, have been retargeted on top of Converse. Prototype implementations of SPMD languages such as SM, PVM and MPI [1] have been done using Converse framework. A data-parallel language DP [11] (subset of HPF) and a data-parallel object-oriented language pC++ [2] have been implemented on top of Converse. A parallel simulation language, Import [12], and a functional language Agents [15] are under development. Converse has also been used to enrich scripting languages such as Perl to support message-driven objects. Message-driven extensions to Java [7] are also implemented.

4 Languages and Paradigms used

In this section, we will describe the four languages that were used in implementing NAMD: the Charm++ parallel object language, PVM implemented under Converse, a simple messaging language (SM), and Converse threads.

4.1 Charm++

Charm++[9] is a parallel object-oriented programming language, based on C++, that follows the message-driven execution model of its predecessor, Charm[8]. Charm++ has been ported to a variety of distributed and shared memory parallel computers including Workstation Networks. Programs written in Charm++ are source-compatible across all the architectures.

Charm++ modules consist of medium-grained objects called *chare objects* and replicated *branch-office* objects. A chare object consists of private and public state variables and methods similar to the C++ objects. In addition, they support *entry methods*, which can be invoked from objects on remote processors. Remote method invocation on an object is done by sending a message to that object using *chare handle*, which is a first class object. The message sent to the remote object is passed as argument to the corresponding entry method. Messages could have priorities associated with them, which are used by the prioritized scheduling algorithm. Chare objects are created dynamically with programmer-supplied load balancing strategies.

Branch office chare objects (BOCs) differ from ordinary chare objects in that they have a representative on each processor. All the branch offices of a BOC share the same name (a BOC identifier.) Therefore, a chare object can invoke entry methods in the local BOC without knowing where it resides. Charm++ also supports various information sharing abstractions such as distributed tables, read-only variables, accumulators, and reductions.

Charm++ has been implemented as a source-level translator to C++ and a runtime library. The Charm++ runtime system uses Converse scheduler to repeatedly select messages from the scheduler's queue according to the programmer-specified scheduling method and initiates execution of the code specified by the message. The Charm++ translator is provided with the interface of the chare or BOC objects consisting of entry method and message declarations. The translator generates C++ code wrappers around entry methods to facilitate packing and unpacking of remote messages.

4.2 PVM

PVM is a popular portable parallel language that runs on most parallel machines. In addition to the traditional tagged sends and receives, like those available on early parallel computers, it includes support for message-buffer management, and dynamic process creation. The former is used to build a message incrementally, and to copy data back from a message into user's variables, and handles heterogeneous data representation on different workstations. The process management primitives allow users to add and remove workstations from a running program. In the Converse implementation, we chose not to support process creation primitives, in part to be consistent with the rest of the languages implemented using Converse.

Our implementation of PVM supports all the communication calls in PVM specification. Using Converse components facilitated implementation of PVM considerably. A message manager was used to buffer message, and provide tag based retrieval. Converse scheduler call to buffer all incoming messages while waiting for a specific message was used for implementing blocking receives. This ensures the correct semantics that allows no other computation (even belonging to other modules) to take place until the blocking receive is satisfied. When such a strict semantics are not needed, a different variant of the call can be used to allow other language modules to carry out their work in the background until the PVM module gets the awaited message.

4.3 Simple Messaging

The Simple Messaging language (SM) was one of the first SPMD-style languages implemented on top of Converse. It demonstrated the ability of Converse to support blocking-receive communication without affecting concurrent message-driven computations. It was later used as a core for our prototype implementation of MPI [1].

SM utilizes the message manager component of Converse to support message-tags and enforce message order guaranteed by most SPMD languages. It is a minimal SPMD language and provides only three calls: send, receive, and probe. When a program makes a blocking receive call, SM runtime explicitly invokes Converse scheduler, repeatedly delivering messages intended for other modules, so that it does not hold up the entire processor.

SM was the language of choice for the initialization module of NAMD because specific parts of the initialization computations have to block for the root processor to broadcast molecular structure information, while the Charm++-related initialization proceeds normally. Also, most of the initialization code is taken from PVM-based NAMD, which used blocking semantics for these messages.

4.4 Converse Threads

The threads component provided in Converse is meant to be used by language implementors. However, in combination with other components, it can also be used directly by the application programmer. In contrast to traditional threads packages, which combine the thread abstraction, thread scheduler and synchronization primitives into a single package, the Converse threads modularly separate these functionalities. The user can attach any scheduler of their choice to each thread at its creation time. Typically, in a Converse program, the user chooses the Converse scheduler, which allows the threads to interleave with other Converse entities. However, by choosing other schedulers, it is also possible to implement hierarchical scheduling schemes.

Thread private variables are supported in addition to node private and processor private variables. Each thread may obtain its own thread ID via a system call. In addition to the traditional "yield" call, Converse threads may also support suspend and awaken calls. suspend stops a running thread without putting it back into the scheduler's queue, while awaken puts a given thread in the scheduler's queue again. These primitives permit many flexible ways of synchronizing the threads.

5 NAMD

The constraints on the design of NAMD can be summarized as follows:

– It must be able to utilize the DPMTA module written in PVM, without having to pay the overhead of interacting with separate PVM processes.
– The performance-critical core of parallel computation must be written in Charm++, to take advantage of its message-driven execution for adaptive overlap of communication with computation, and the ability to prioritize messages.
– The coordination logic of the program must be modularly separated from the parts of the program that implement the parallel mechanics needed to execute it.
– Nearly half of the code involved parallel initialization, using synchronous message passing. Reusing code from older version of NAMD without extensive modification was desirable.

Using different languages for implementing different modules, we were able to address all of the above (see figure 1.) The initialization code involved reading

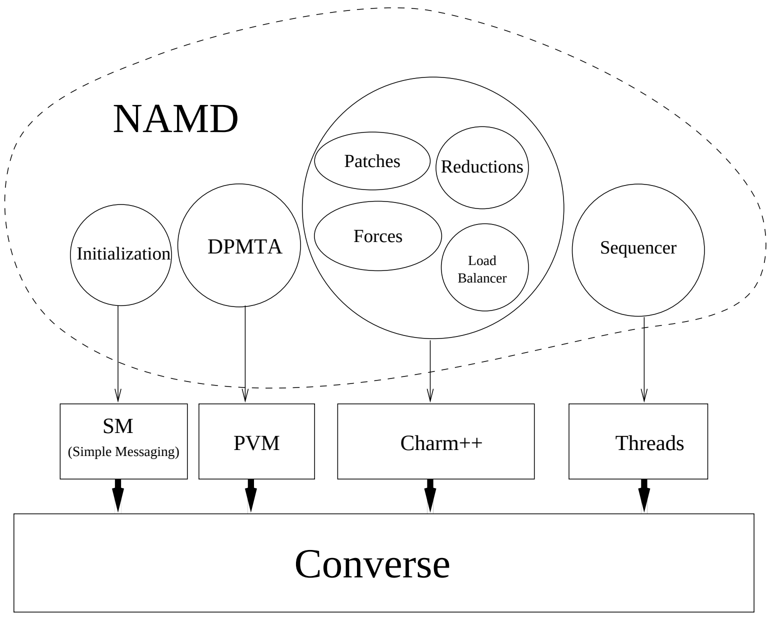

Fig. 1. NAMD Architecture

in the data from the disk, and installing individual data structures on the parallel processors. To avoid race conditions, it used synchronous message passing, with tagged messages and blocking receives. SM language provides such message passing with lower overhead than PVM or MPI.

We were able to reuse the DPMTA module by simply linking it with the implementation of PVM on Converse (see Section 4.2).

The remaining constraints presented the major challenge. Writing the code for core computations entirely in Charm++ would lead to good performance, and also reasonable perspicuity, however, we wanted to separate the algorithmic logic from the parallel logic, so that someone wishing to change the basic numerical algorithm (e.g. Verlet-I integration, Langevin dynamics) can do so without being encumbered by the parallel logic. This was accomplished by observing that the algorithmic logic involves the description of the life cycle of a patch: sequencing of the steps of the algorithm, and choosing individual sub-algorithms for each step. In contrast, the parallel logic involves movement of data, triggering of computations based on the availability of data, and interprocessor synchronization. The latter can be thought of as mechanisms under control of the algorithmic logic, encapsulated in a *Sequencer* object.

As there are multiple patches per processor, multiple sequencers must be active on individual processors, with the possibility that they may not be in lock-step. Some sequencers may be lagging behind, while other race ahead, constrained only by the data dependencies. Periodically each sequencer must wait for its data to become available, when one of the other sequencers on the processor can continue. Also, the bulk of parallel computations in Charm++ modules must be allowed to execute interleaved with the sequencers. This suggested the use of threads for implementing sequencers. Normally, it would be difficult to concurrently run multiple threads and message-driven objects in the same program. For one thing, Charm++ would use its own message-driven loop to schedule messages for its objects, which would conflict with the thread scheduler. Running Charm++ in one of the threads is not a good solution, because of the stack space limitations imposed by threads, and the context switching overheads of threads which we wish to avoid for the fine-grained dominant computations. However, the threads implemented within Converse allow applications to mix threaded and non-threaded computations effectively. The common scheduler can switch between threads and objects, while allowing unlimited stack space to the objects.

6 Computational Core

NAMD has four interleaved subtasks occurring during a simulation. They are: (1) force computation, (2) proxy patch update, (3) integration, and (4) reduction. These subtasks occasionally exchange information, but otherwise do not affect each other.

Force computations are performed by force objects. Each force object is assigned a number of patches for which it must compute some portion of the force. The force object accesses the atoms belonging to each patch either directly, when the patches are on the same node, or through proxies in the case of remote patches. Patches and proxies inform the force object that new positions are available. When all required patches have reported in, the force object schedules itself by enqueuing a Charm++ message to itself. This message may have a priority associated with it, so the order of force calculation can be optimized. Upon receiving the message, the force object performs the calculation, and returns the forces to the awaiting patches and proxies.

The second component of the simulation is the proxy patch update system. Each real patch knows which other nodes require its position data to perform force computations. On each of those other nodes is a proxy object, which awaits positions from the home patch. When the proxy receives the new position message, it informs any force objects that new positions are ready. When the force objects are finished, they deposit the computed forces, and the forces are returned to the home patch.

The integration procedure is outlined in figure 2. In earlier versions of NAMD, this functionality was spread over several parallel modules. Using the multilingual approach, it is now modularly separated from the other components. Since this algorithm must interact with all the other components, it runs inside a thread, to allow easy context switching when required data is not available. One sequencer object is created for each patch. In a simple simulation it simply enters a timestep loop:

– Atom positions are updated using previously computed forces.
– New positions are delivered to the patch's proxies on other nodes.
– Local force objects are informed that new forces should be computed.
– The thread suspends until all force objects and proxies have returned the new forces.
– Atom velocities are updated using the new forces.
– Position, velocity and energy values are passed to the reduction component.

In a case where the reduction results are required, the loop can suspend after submitting information to the reduction module and await a response.

After patches have computed each timestep, they submit the positions, velocities and energies to the reductions component. The reduction module on each node awaits data from each patch. When this data is received, it is propagated to a master node, which outputs the data. In some simulations the results of the reduction must be delivered back to the patches. In these cases, the sequencers suspend at the point of requiring the new data, and the reduction module awakens them when the data has returned.

We were successful in our major goals of NAMD, most importantly improving the clarity and modifiability of the integration module. Figure 3 shows a modified sequencer algorithm which implements dual timestepping for full range electrostatics. No additional communication or thread control calls are visible. The only visible modifications are additional calls to use the new forces. We modified the patch code to provide storage for a long-range force component. We also added a call so the long-range force object can determine if long-range forces are to be computed in this timestep, and another so that the sequencer can specify whether or not to include the force in the integration.

7 Preliminary Performance Data

Preliminary implementation of NAMD has been completed and tested on an SGI Origin 2000 with 32 processors at NCSA. As the bulk of the critical com-

```
begin
    patch→ sendToProxies()                  Send initial positions to proxy
                                            patches
    patch→ triggerForceObjects()            Trigger local force objects to calcu-
                                            late initial forces
    patch→ suspendUntilForcesReturned()     Wait for forces and proxies to return
                                            f(0)
    patch→ submitReductions(0)              Send position data for initial reduc-
                                            tion
    for step = 1 to N
    begin                                   At the start of the loop, we have
                                            v(step-1), x(step-1) and f(step-1)
        patch→ updateHalfstepVelocities()   Use forces(step-1) to find v(step-
                                            1/2)
        patch→ updatePositions()            Find x(step)
        patch→ sendToProxies()
        patch→ triggerForceObjects()
        patch→ suspendUntilForcesReturned()
                                            Wait until f(step) is evaluated for
                                            x(step)
        patch→ updateVelocities()           Use forces(step) to find v(step)
        patch→ submitReductions(step)
    end
end
```

Fig. 2. Sequencer algorithm for a single-timestep Verlet integration simulation

putation was done using Charm++ primitives, it was possible to use Charm++ performance visualization and analysis tools such as Projections.

Table 1 shows times required for one timestep for two different molecules, on different numbers of processors. As the number of atoms in the simulated molecule increase, so does the amount of computation done per processor, resulting in improved parallelization. Thus, NAMD obtains better speedups for APO-A1 simulation than those for ER-GRE. Also, ER-GRE is aperiodic and causes non-uniform density of atoms inside patches, resulting in a more severe load imbalance. We compared the performance of NAMD with its earlier PVM-based version, and also with a popular parallel molecular dynamics program, X-PLOR. The results are shown in Table 2. NAMD clearly performs better than its earlier version, and X-PLOR in terms of execution time per simulation timestep and also scales better. These preliminary performance times are improving as we continue to optimize the parallel code.

```
begin
   patch→ doFullElectrostatics()                 Informs patch to tell force ob-
                                                 jects that this is a full-electrostatics
                                                 timestep
   patch→ sendToProxies()
   patch→ triggerForceObjects()
   patch→ suspendUntilForcesReturned()
   patch→ submitReductions(0)
   for step = 1 to N
   begin
      if (step - 1 mod dtsPeriod) = 0 then       Last step should have evaluated long-
                                                 range forces
      begin                                      Add long and short contributions
         patch→ updateHalfstepVelocitiesFull()
         patch→ updatePositionsFull()
      end
      else begin                                 Add short forces
         patch→ updateHalfstepVelocities()
         patch→ updatePositions()
      end
      if (step mod dtsPeriod) = 0 then
         patch→ doFullElectrostatics()
      patch→ sendToProxies()                     Also requests full electrostatics
      patch→ triggerForceObjects()
      patch→ suspendUntilForcesReturned()
      if (step mod dtsPeriod) = 0 then
         patch→ updateVelocitiesFull()           Adds long and short contributions
      else patch→ updateVelocities()
      patch→ submitReductions(step)
   end
end
```

Fig. 3. Sequencer algorithm for a dual-timestep long-range nonbonded force simulation

8 Summary

Multilingual multi-paradigm parallel programming has the potential to improve the productivity of parallel programmers, while facilitating the introduction of novel parallel languages. We described development of NAMD, a large-scale multilingual application program written using the Converse framework that demonstrates this potential. NAMD was implemented using 4 different "languages" or paradigms: Charm++, PVM, the Converse thread library, and SM, an optimized messaging library. The advantages of multilingual programming were illustrated by comparing our implementation with the previous generation version of NAMD, written entirely in one language. These advantages accrue because of the ability to use powerful languages, and the ability to employ an

Simulation		Processors				
		1	2	4	8	16
ER-GRE (37,000 atoms)	Time	7.2	4.0	2.0	1.2	—
	Speedup	1.0	1.8	3.6	6.0	—
APO-A1 (92,000 atoms)	Time	32	18	9.3	4.6	2.4
	Speedup	1.0	1.8	3.4	7.0	13

Table 1. Time taken per timestep for two molecules on the SGI Origin 2000.

Program	Processors			
	1	2	4	8
XPLOR	8.3	4.7	2.9	2.2
PVM-NAMD	9.1	5.3	3.8	2.1
NAMD	6.1	3.5	2.2	1.4

Table 2. Execution time per simulation step for PRO1-WAT (15,000 atoms) on the HP 735/125 cluster.

appropriate language for each module individually.

Future plans for development of NAMD include testing on larger machines, and exploring opportunities for further performance improvements. Based on our success so far, we also plan to explore using Structured Dagger, a coordination layer for Charm++, for specific parts of the program. Structured Dagger allows cleaner specification of the flow of control within a parallel object. We have identified sections of NAMD where it can lend clarity without sacrificing performance.

Several new languages are being implemented using Converse. The ease of developing parallel runtime systems with converse, along with the ability to interoperate with other languages, will hopefully lead other language implementors to use Converse, thus enriching the framework further. We also plan to develop techniques and tools for performance analysis and debugging of multilingual programs. We expect this research to lead to broader acceptance of the multi-paradigm approach and substantial productivity improvement in parallel application development.

References

1. Milind Bhandarkar and L. V. Kale. MICE: A Prototype MPI Implementation in Converse Environment. In *Proceedings of the second MPI Developers Conference*, pages 26–31, South Bend, Indiana, July 1996.
2. F. Bodin, P. Beckman, D. Gannon, S. Narayana, and S. Yang. Distributed pC++: Basic Ideas for an Object Parallel Language. *Scientific Programming*, 2(3), 1993.

3. N. Carriero and D. Gelernter. Linda in Context. *Communications of the ACM*, 32(4), April 1989.
4. A. Gursoy. *Simplified Expression of Message Driven Programs and Quantification of Their Impact on Performance*. PhD thesis, University of Illinois at Urbana-Champaign, June 1994. Also, Technical Report UIUCDCS-R-94-1852.
5. William F. Humphrey, Andrew Dalke, and Klaus Schulten. VMD – Visual molecular dynamics. *Journal of Molecular Graphics*, 14(1):33–38, 1996.
6. L. V. Kale, Milind Bhandarkar, Narain Jagathesan, Sanjeev Krishnan, and Joshua Yelon. Converse: An Interoperable Framework for Parallel Programming. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 212–217, Honolulu, Hawaii, April 1996.
7. L. V. Kalé, Milind Bhandarkar, and Terry Wilmarth. Design and implementation of parallel java with a global object space. Technical report, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, April 1997. Submitted for publication.
8. L.V. Kale. The Chare Kernel parallel programming language and system. In *Proceedings of the International Conference on Parallel Processing*, August 1990.
9. L.V. Kale and S. Krishnan. Charm++: A portable concurrent object oriented system based on C++. In *Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications*, September 1993.
10. L.V. Kale and Amitabh Sinha. Projections : A scalable performance tool. In *Parallel Systems Fair, International Parallel Processing Sympos ium*, pages 108–114, April 1993.
11. E. Kornkven and L. V. Kale. Efficient implementation of high performance fortran via adaptive scheduling – an overview. In *Proceedings of the International Workshop on Parallel Processing*, Bangalore, India, December 1994.
12. Vance P. Morrison. Import/dome language reference manual. Technical report, US. Army Corps of Engineering Research Laboratory, ASSET group., 1995.
13. Mark Nelson, William Humphrey, Attila Gursoy, Andrew Dalke, Laxmikant Kalé, Robert D. Skeel, and Klaus Schulten. NAMD— A parallel, object-oriented molecular dynamics program. *J. Supercomputing App.*, 1996.
14. W. Rankin and J. Board. A portable distributed implementation of the parallel multipole tree algorithm. *IEEE Symposium on High Performance Distributed Computing*, 1995. [Duke University Technical Report 95-002].
15. Joshua Yelon and L. V. Kale. Agents: An undistorted representation of problem structure. In *Lecture Notes in Computer Science*, volume 1033, pages 551–565. Springer-Verlag, August 1995.

This article was processed using the LaTeX macro package with LLNCS style