A Hybrid Imperative and Functional Molecular Mechanics Application

THOMAS DEBONI, JOHN FEO, AND DOUG PETERS
Lawrence Livermore National Laboratory, Livermore, CA 94550

ABSTRACT

Molecular mechanics applications model the interactions among large ensembles of discrete particles. They are used where probabilistic methods are inadequate, such as drug chemistry. This methodology is difficult to parallelize with good performance, due to its poor locality, uneven partitions, and dynamic behavior. Imperative programs have been written that attempt this on shared and distributed memory machines. Given such a program, the computational kernel can be rewritten in Sisal, a functional programming language, and integrated with the rest of the imperative program under the Sisal Foreign Language Interface. This allows minimal effort and maximal return from parallelization work, and leaves the work appropriate to imperative implementation in its original form. We describe such an effort, focusing on the parts of the application that are appropriate for Sisal implementation, the specifics of mixed-language programming, and the complex performance behavior of the resulting hybrid code.

1 INTRODUCTION

In this article we describe our efforts at parallelizing a legacy scientific application code suite through the use of the Sisal language [1, 2] and its mixed-language capability. Sisal is a general-purpose functional language that hides the complexity of parallel processing, expedites parallel program development, and guarantees determinacy. We parallelized a molecular mechanics (MM) application that performs force minimization.

Parallel programming traditionally involves the management of concurrent tasks and machine resources, in addition to the specification of the computation, thereby greatly increasing the programmer's burden. MM [3] applications model interactions among large ensembles of discrete particles, and are used where probabilistic methods are inadequate, such as the simulation of organic systems and drug molecules. This methodology is difficult to parallelize with good performance, due to its poor locality, unbalanced task workload, and dynamic behavior [4, 5].

Sequential imperative implementations of MM exist, and, given such a program, it is possible to recode the computational kernel in Sisal, a functional programming language, and integrate it with the rest of the imperative program under the Sisal Foreign Language Interface (FLI) [6, 7]. This mixed-language programming capability offers an efficient developmental path to a parallel production code.

In this article, we describe such an effort, beginning with a description of the scientific application in Section 2 and the gross structure of the code in Section 3. Then in Section 4, we describe in detail the Sisal Foreign Language Interface and the way

Received April 1995
Revised June 1995

Scientific Programming, Vol. 5, pp. 97–109 (1996)
CCC 1058-9244/96/020097-13

we used it to parallelize the code, as well as some of the performance behaviors we encountered. In Section 5, we discuss reduction operations and their implementation. We present performance results in Section 6, and our conclusions in Section 7.

2 MOLECULAR MECHANICS

With the continuing advance of the speed of central processing units (CPUs) and the size of accessible memory, it has become practicable to calculate the properties of large ensembles of atoms and molecules. This opens the door to rational drug design, where the properties of large systems of interacting organic molecules are of interest. It is possible to save considerably in both the time and cost required to bring a new drug to market by using these computational techniques. At this time, the computations help reduce the number of potential drugs which need to be tested in laboratory trials, and allow one to focus on the most promising ones. For instance, one might calculate the binding properties of a drug molecule with a strand of human DNA. The computation replaces the technique of simple geometry matching based on crystallographic structure data, and results in a vastly more reliable and powerful tool.

F_MIN is one such application; it is an optimization code used to find the minimum energy configuration of one or more interacting molecules, possibly in solution. The code uses paramaterized potentials for the interactions, with the parameters obtained by a variety of empirical means. The total interaction energy is broken into three parts:

$$E_{tot} = E_{bonded} + E_{non-bonded} + E_{H-bond}.$$

The first term, the bond interaction potential, is

$$E_{bonded} = \sum_{bonds} K_R(R - R_o)^2 + \sum_{angles} K_\theta(\theta - \theta_o)^2 + \sum_{dihedrals} V_n[1 + \cos(n\phi - \gamma)],$$

where R_o is the equilibrium bond length and K_R is the force constant for stretching or contracting the bond. The second term accounts for angular bending forces when the angular separation between two bonds is displaced from its equilibrium, θ_o. Note that these two terms represent a harmonic approximation to the true potentials, and will be most accurate for small displacements from equilibrium. In a simulation, there will be multiple types of bonds, and each one will have its own force constants. The third term is a slightly more complicated form representing the dihedral energy, which is the energy necessary to twist a pair of bonds relative to another pair about their mutual axis.

The nonbonded interaction is

$$E_{non-bonded} = \sum_{i<j} \left[\frac{B_{ij}}{R_{ij}^{12}} - \frac{B_{ij}}{R_{ij}^{6}} + \frac{q_i q_j}{\varepsilon R_{ij}} \right],$$

where the first two terms represent the Lennard-Jones potential, with R_{ij} being the distance between nuclei i and j. The third term is the electrostatic force, screened by a dielectric constant ε, between two ions of charge q_i and q_j.

The final contribution is due to hydrogen bonds,

$$E_{H-bond} = \sum_{H-bonds} \left[\frac{C_{ij}}{R_{ij}^{12}} - \frac{D_{ij}}{R_{ij}^{10}} \right].$$

These potentials are defined over pairs of particles, and can be computed conveniently and relatively quickly. However, N-body effects may be important in some circumstances. There are other techniques being developed, such as density-functional theory that could in principle improve the accuracy of these calculations by using more direct quantum approaches to the electron correlation effects, while still leaving the problem computationally tractable. These advanced methods will not be discussed further here, as they are not implemented in this code.

Once the potential functions for the interactions between the atoms and molecules are defined, the mutual forces can be calculated by taking the gradient of the total potential with respect to the particle coordinates. If the system were at zero temperature in its optimum configuration, there would be no forces on any of the particles, and the system would be at its true minimum of energy. However, for any configuration away from this minimum, there will be net forces on some of the atoms, tending to move them toward the equilibrium position. The exception to this rule would be if the system possessed other local minima in the energy, which would not correspond to the ground state, but rather an excited, metastable state.

It is an important, but unsolved, problem in general to ensure that the minimum found is a global, rather than a local minimum. In the simple techniques employed here, which use local gradient information, one relies on the fact that the

starting configurations of the molecules are derived from crystallographic data, which are assumed to be "close" to the true minimum. Other more advanced methods for global optimization, such as annealing techniques, may someday provide a robust solution to this problem.

The minimization is done either by steepest descent followed with conjugate gradient treatment, or by the conjugate gradient method alone. The method of steepest descents corresponds to the simple physical picture described above. The force on the atoms is the gradient of the potential energy,

$$\vec{F}(\vec{R}_i) = -\vec{\nabla} E(\vec{R}_i) \equiv \vec{g}_i,$$

where the subscript i denotes the values of these functions at iteration i. Once the gradient is found, the atoms are moved in the "downhill" direction until the minimum in that direction is reached. The gradient is recomputed, and a new set of moves is made to find a new minimum in the new direction. Note that the force and potential energies are functions of the 3*N Cartesian coordinates of the N particles, or some more generalized coordinate system chosen to take advantage of the various symmetries of the problem. Also, there are additional forces due to constraints which the user may specify. The process continues iteratively until the forces on the atoms are smaller than some previously determined threshold or until the maximum number of steps have been taken. Thus, we have equivalent descriptions of this technique as one of force minimization (the pairwise interatomic forces are zero at equilibrium) or of energy minimization (the potential energy is a minimum at equilibrium). It is also referred to as line minimization, because the minimization takes place along a series of lines (or vectors) in the multidimensional configuration space. This method is straightforward, but not always very efficient. For some potential surfaces, like a long, narrow valley, the approach to the minimum will consist of a large number of very small steps. What happens is that each successive move "spoils" the minimization of the previous moves because, e.g., after the move in direction of $\vec{g}_{i+1}$, the gradient of the potential in the direction of $\vec{g}_i$ will no longer be zero. The conjugate gradient technique was developed to overcome this problem.

The conjugate gradient method ensures that each step is in a sense conjugate to all of the others, improving the efficiency of the search. Two sequences of vectors are defined; the first is the gradient at each step, $\vec{g}_i$, and the second, which is the actual step direction, is

$$\vec{h}_{i+1} = \vec{g}_{i+1} + \gamma_i \vec{h}_i,$$

where

$$\gamma_i = \frac{(\vec{g}_{i+1} - \vec{g}_i) \cdot \vec{g}_{i+1}}{\vec{g}_i \cdot \vec{g}_i}.$$

Using this method, the successive steps leave the minimization along the previous steps essentially intact. The proof of why this technique works is beyond the scope of this short review, and those who wish further details should consult [8].

F_MIN may be used in, with, or as part of a full molecular dynamics code, allowing one to calculate the dynamics of a system, reconfiguring the molecules for optimum geometry at each step. Alternatively, F_MIN can be used to find the optimum geometry of some molecule or ensemble which is then held fixed as the molecular dynamics simulation proceeds.

3 STRUCTURE OF THE MM CODE

The MM methodology is one of two major aspects of full molecular dynamics. In it, full particle motions are simulated in solving Newton's laws of motion for the system. In MM, particle motion is not simulated. Rather, the positions of the particles are changed with an eye toward discovering a configuration at which potential energy or force is minimized [4, 9, 10]. F_MIN performs only the minimization work. In the following paragraphs we describe the gross structure of the code.

At the highest level, F_MIN is a series of minimization steps or iterations. At each one, the program computes interatomic energies arising from the interaction formulae described in Section 2. Figure 1 depicts the call tree of the sequential Fortran program. The module RUN_PROB includes the iterative minimization loop. It calls the module FORCE_MIN that calculates one or more force functions depending on problem specification. The force functions maintain a running sum of the force on each atom, in an array f. On exit from FORCE_MIN, f contains the total force on each atom.

F_MIN is typical of large scientific applications in three respects: (1) Most of the code performs utility functions, e.g., it reads inputs, initializes

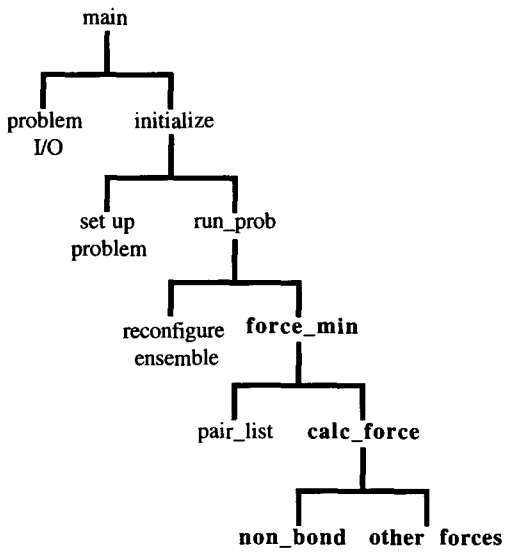

FIGURE 1 Call tree of the sequential Fortran version of the F_MIN program (kernel routines in bold face).

data structures, defines the initial configuration, collects statistics, checks for errors, and writes outputs; (2) the application includes a well-defined computational kernel that performs the desired simulation and comprises a small part (approximately 30% in this case) of the whole code; and (3) more than 90% of the execution time of the program is spent in the computational kernel. One module of the kernel, NON_BOND, dominates the entire application. This module calculates the nonbonded forces modeled by the first term in the expression for total interaction energy by calculating the forces for the interacting atom pairs among the N atoms. To compute a simple nonbonded pairing takes 40 floating-point operations (flops), whereas, to compute a hydrogen nonbonded pairing requires 53 flops.

To reduce the $O(N^2)$ computational complexity of this problem [2, 12], we employ a commonly used approximation. Rather than consider all $N * (N\text{-}1)/2$ possible pairings of atoms, we consider for each atom i only those "neighbor" atoms j within a certain "cutoff radius" in three dimensions. The smaller the radius, the fewer the number of nonbonded pairings. For most of the simulations used in this research, cutoffs greater than 8–10 Å are not warranted, and this typically gives interaction counts ranging from a few tens to a few hundred neighbors per atom. In a typical simulation of a system of 1000 atoms, modeling a large molecule in water, we have from 110000 to 150000 interacting pairs at various times in the minimization, significantly fewer than would result from a complete set of pairwise interactions.

The cutoff apprximation requires the construction of a "pairlist" of atoms. This list is the governing data structure for the nonbond force calculations. It stores for each atom i, $1 \leq i \leq N$, the indices of the atoms that are within the cutoff distance. Since particles move, we must periodically rebuild the pairlist, an $O(N^2)$ computation. Fortunately, the pairlist changes slowly so we do not have to update it every iteration. An additional complication regarding the pairlist is that atoms have a somewhat clumped distribution in 3-space, causing the atoms to have varying numbers of neighbors. Thus, the pairwise computations exhibit poor data locality and load balancing, an important performance consideration for parallel execution.

Despite the cutoff approximation, NON_BOND typically accounts for more than 95% of total CPU time. (Constructing the pairlist involves relatively few flops.) Since the nonbonded pairings are computationally independent, we may compute them concurrently. A large problem of 35000 atoms and 10000 steps can take many hours to solve sequentially on a vector supercomputer, but speedups are potentially achievable under parallel execution. In general, parallel processing has proven effective in dealing with this class of simulated systems [4, 5]. Since NON_BOND is such a large percentage of the total work, we get the largest return on our investment of effort if we parallelize this one routine; in effect, we will have, for all practical purposes, parallelized the entire application.

Several termination criteria are possible in force minimization. Because a true minimum may not be practically achievable, there may be more than one acceptable final configuration. Thus, we can use absolute energy statistics or relative energy change statistics, as well as iteration ceilings, to stop execution. We tended to use iteration ceilings for most of our experiments, because we were more interested in how the code ran than the ultimate values of the system states it produced. Note, however, that we did intend to produce a working production code, so result verification was important and was extensively performed.

4 MIXED-LANGUAGE PROGRAMMING WITH THE SISAL FLI

Most computer programs are not originally written for parallel execution. More often, they begin as

existing sequential programs, written in an imperative language. When parallel execution becomes desirable, the sequential program may be augmented with parallel constructs from an enhanced, perhaps vendor-specific, imperative language. The application we consider in this study is such a legacy code. It was written in Fortran 77 for uniprocessor execution, makes substantial use of common blocks and other Fortran tricks, and was structured for execution on a vector supercomputer.

The programmer who is assigned the task of parallelizing this code has four major concerns. First, the gross behavior of the code must be preserved—the answers must remain the same. Second, the parallel code must execute efficiently on the parallel machine of choice and must exhibit some scalability. Third, the code should port easily to other parallel machines, in particular, new generations of the target machine. Fourth, development costs should be kept as small as possible. As these goals are contradictory, there may be no best solution. Since minimizing programming costs is an important objective, the imperative programmer usually identifies the most computationally intensive parts of the code and parallelizes only the parts that will provide the most gain from parallel execution. By considering only these parallelizable sections, the imperative programmer maximizes performance and minimizes development costs.

The Sisal language provides a notation more closely related to mathematics than to any machine architecture. This allows parallelism to be expressed in a natural form and automatically exploited (or not, as appropriate) for the chosen execution platform. It removes concerns about sequential or parallel execution from the programming task. The Sisal compiler transforms the source code through a number of forms and a suite of optimizers, and eventually emits C. The C compiler for the target execution platform is then invoked either automatically or manually to produce an executable memory image.

The Sisal language is implemented on many uniprocessors and shared memory multiprocessor systems, and allows easy porting among them with determinate execution behavior. It also supports mixed language programming through its FLI. The FLI allows Sisal programs to call or be called from Fortran or C, and to invoke existing libraries or solvers. This allows recoding the computational kernels of existing codes for parallelism, and thus addresses all four of the programmer's concerns.

4.1 Using the FLI and the Run-time System (RTS)

The Sisal FLI allowed us to rewrite only the core of the F_MIN program, leaving unchanged the work best specified in the original sequential Fortran. Here we describe the hybrid Sisal/Fortran code (hereafter called the Sisal version). We rewrote the kernel module CALC_FORCES and its descendants, which includes the module NON_BOND as shown in Figure 1. A Fortran programmer might reasonably concentrate on this routine alone, but we rewrote the other force calculation routines as well. Doing so added only slightly to our development costs, allowed us to exploit a bit more parallelism, simplified the task of integrating the Sisal and Fortran portions of the code, and made our use of the FLI more efficient. In addition, we added calls within the module RUN_PROB to start, configure, and stop the Sisal RTS.

In a purely Sisal program, all data items are dynamic. Identifiers refer to values, not memory locations, and a data value exists only between the computations that create and consume it. The RTS manages the actual storage for such values, allocating space for them on creation and scavenging it when they are consumed. It uses an internal heap to perform these functions, which makes it work faster than standard system calls for similar activities. In addition, it caches units of storage by size, in the expectation that once an array has been consumed and its storage deallocated, another array of the same size may soon be required. This further improves memory management performance. Working with the Optimizing Sisal Compiler (OSC) [6, 7], the RTS is able to minimize overhead operations, such as data copying, and allow aggregate objects to be built and modified in place, without violating Sisal's functional semantics. The optimizations performed by the compiler and the management performed by the RTS are especially effective in managing arrays efficiently.

The other major responsibility of the RTS is the management of the worker processes that perform the parallel execution. The RTS initiates the requested number of "workers," which then attempt to access work from a queue. The original executing process then executes until work becomes available for the workers, which it then puts on the work queue. Minor responsibilities of the RTS include the gathering of run-time checks and statistics and execution profiling.

The RTS is normally started automatically on execution of any program compiled by the OSC.

In a hybrid program, however, it must be explicitly initiated by the Fortran portion of the code prior to invocation of the outermost Sisal function. The parameters governing the execution of the RTS, such as the number of worker processes, memory allocation, and loop slicing, which would be provided on the execute line of a pure Sisal program, are provided by calling either a configuration routine or a routine that reads a file of execute-line options. Therefore, in either pure or hybrid Sisal contexts, changing the execution environment on a particular hardware platform does not require recompilation.

We start the RTS at a level above that at which the outermost Sisal function is invoked. This allows the Sisal code to be called from, and to return results to, the Fortran code within each iteration of the minimization loop. Starting and terminating the RTS at each iteration would have incurred unnecessary expense, due to repeatedly creating and terminating the worker processes and setting up the work queue. This decoupling of the functionality of the RTS and the Sisal kernel demonstrates the flexibility of the mixed-language programming capability.

The current practicalities of parallel systems dictate that most effectively exploitable parallelism arises from slicing loops, and the Sisal language software excels at this. The Sisal portion of the force minimization program consists of 26 functions containing a number of parallel loops. Each of these loops is an opportunity to exploit parallel and vector execution based on decisions made by OSC and the RTS, and through parameters specified by the user at compile time and run-time. These parameters can control, among other things, how deeply to parallelize nested loops at compile time and how finely to slice parallel loops at run-time. As would be the case with an imperative parallelization, most of the parallelism (and most of the work) is in the kernel routine NON_BOND.

4.2 Storage Management and the FLI

The RTS dynamically allocates memory for aggregate objects from a heap, and deallocates and reuses this memory efficiently. This dramatically reduces the traditional performance problem of functional languages—data copying and inefficient memory usage. The RTS also manages the protection of shared objects so that reads and writes proceed safely and deterministically. All synchronization and communication among parallel workers are done automatically at run-time.

The Sisal FLI passes data to and from the imperative and Sisal data spaces. Scalars are passed by value. Whenever possible, arrays are passed by reference, minimizing copying and the cost of crossing the interface. Array descriptors are built for every Sisal array, which contain the array's starting address in physical memory, its lower bound, its length, and reference count information. This information is automatically generated for Sisal arrays, due to their dynamic nature, but it must be provided for arrays passed in from Fortran, since they will be treated syntactically the same as Sisal arrays on the Sisal side of the FLI. The array descriptors are built on the imperative side of the interface and passed across the FLI as companion structures to the arrays they describe.

There are two problems with the FLI that we wish to address. The first is relatively minor, more of an irritant than a real problem: The programmer must explicitly define the array descriptors, one per array. This task should be automated. Pure Sisal's dynamic nature normally relieves the programmer of the need to declare the static sizes of arrays, but the FLI reimposes it. The second problem is more serious. Fortran and C store all arrays in contiguous memory, but Sisal stores multidimensional arrays as arrays of arrays. In its current implementation, the FLI may copy multidimensional arrays across the interface to compensate for the different storage layouts. Note that one-dimensional arrays are less likely to be copied, as are arrays passed into Sisal from the imperative side (as opposed to those returned from Sisal to the imperative side). The programmer may elect to pass all Fortran and C arrays, regardless of their dimensionality, as one-dimensional arrays, but this may cause the index expressions in the Sisal code to be more complicated to compute.

Because of the number and size of the arrays in F_MIN, we elected to pass all arrays as one-dimensional structures. In a prototype version of our hybrid code, the Sisal code ran essentially within the Fortran data space. The RTS allocated only around 64 KB of memory, and the Fortran provided around 1 MB.

The parallelism in this program was exploited by way of slicing the many large loops in NON_BOND and the other force calculation routines. Sisal performed this slicing automatically. It also fused those loops with the same ranges, improving performance in three ways. First, it provided longer vectors; second, it provided more work between

accesses to the RTS task queue structures; and, third, it reduced the need for intermediate array structures that would otherwise be needed between producer and consumer loops. Loops with different ranges, however, could not be fused. This caused a storage problem, forcing the RTS to allocate and scavenge intermediate structures. Unfortunately, the nonstandard loop ranges tended to vary widely, exacerbating the storage use problem by reducing the effectiveness of the RTS' storage caching mechanisms. We regard this as a characteristic of the MM problem domain, and have not seen it in other work with the Sisal FLI.

The asymmetry between one- and two-dimensional arrays gives rise to unusual programming decisions, best illustrated by describing how we handled the pair list in F_MIN. This structure contains the list of atom pairs for which bond calculations are to be performed. The Fortran code generates this list as a one-dimensional array of atom indices, with auxiliary structures containing the starting index and number of neighbors for each atom. It would seem desirable to generate this list in Sisal, using a ragged two-dimensional array, where row i contains the indices of the neighbors of atom i, and then use the array to drive the nonbond force calculation.

Because the pair list is not regenerated on each iteration of the minimization loop, we would have to pass the list out to Fortran so it could be passed back into Sisal on the next iteration. Passing the two-dimensional array back and forth across the FLI would be very expensive due to copying. For this reason, we decided to let Fortran generate the pair list and pass it into Sisal as a one-dimensional structure, along with two companion lists, one list of starting points in the neighbor list for each atom and another list of numbers of neighbors for each atom. Now the index of an element of the pair list is no longer the index of one of the two atoms in the bond, but the index of the bond itself. This lack of information complicates the Sisal code, but this organization also offers some performance advantages, to be explored in a later section.

It is a long-standing goal of the Sisal language project that application programmers be freed from concerns that are secondary to the design of their applications. Memory management, the static sizing and explicit allocation and deallocation of aggregate objects, is one such concern that the Sisal language all but does away with. However, we have learned from this work that in a mixed-language context we are still not completely freed from such responsibilities.

4.3 Transportability of Mixed-Language Code

Performance is not the only concern in application programming; we must also consider source code portability. Sequential imperative languages are generally portable among the many commercially available computer systems. Some impediments remain, such as word size, arithmetic algorithms, naming, and system service calls, but these tend to be relatively minor. The more serious problems arise from the architectural differences among parallel computer systems, and from the varying and nonstandard extensions vendors add to programming languages to support their parallel machines. These extensions tend not to be very portable. Sisal code by itself enjoys very good portability among the platforms it runs on, but it, too, is affected by some of the minor impediments mentioned above, because it is dependent on lower-level system idiosyncrasies.

The first problem, word size, affects Sisal differently on each machine on which it runs. Sisal contains the base scalar types *integer*, *real*, and *double_real*, but the length in bits of a *real* on one machine may be equivalent to that of a *double_real* on another. Both the IBM RS/6000 and SGI 340 have processors capable of 32-bit floating-point arithmetic, which is the default for data of type *real*, but on the Cray C-90, a type *real* operand is 64 bits long. In Sisal's defense, compiler flags exist that force *real* or *double_real* data into one or the other form. However, the differences in usage of these terms still demand more precise type specifications. In Fortran, this is answered by the use of length declarations, such as *real*8* for 8-byte floating-point values. Such a facility is conspicuously absent from the Sisal language. The corresponding problem with integers also exists. Although integers do not contribute to numerical errors and their propagation, they are heavily used to store bond and atom indices, as well as other problem parameters. To save space, integers of appropriate lengths may be carefully chosen by application programmers. The original Fortran version of F_MIN stores the pair list as an array of 2-byte integers because the maximum number of atoms it can handle is limited by design to 32000. Since all integers in Sisal are 4 bytes and the pair list crosses the interface, we were forced to redeclare the pair list in the Fortran code as an array of 4-byte integers, effectively wasting half its space.

The second problem, different arithmetic algo-

rithms on our target machines, affects both Sisal and Fortran equally. Most modern processors support IEEE-standard arithmetic. IBM and SGI systems do too, but Cray systems do not. Therefore, there is an expected difference in the results on the Cray, and this is normally not of great consequence. Full molecular dynamics is an application in which such differences can rapidly grow and propagate, and significantly affect result values, but this tends not to occur in MM. The final converged configuration and its energy statistics are not likely to be greatly affected by arithmetic differences, although the number of iterations taken to arrive at it and the intermediate values may be affected. Merely reordering the operations in a molecular dynamics computation can affect the results, and imperative language compilers on different machines will almost certainly emit different operation sequences for specific source code. A further wrinkle in this matter is that one of our target machines, the IBM, is capable of arithmetic more precise than is standard, due to merged multiply-add instructions. That the use of this instruction can be disabled at compile time, however, still does not stop the compiler from generating different code sequences than might be obtained on another make or generation of computer or compiler. Thus, even though we were not adversely affected by arithmetic differences across our target machines, if our MM code were to be used as part of a full molecular dynamics code, such differences could become large and important.

The naming problem exists because there is no agreement among system implementors about how to generate temporary names in compilation, assembly, and for libraries. OSC uses underscore characters in its temporary names in such a way as to conflict with some, but not all, UNIX C compilers. There is also some disagreement among Fortran compilers as to the maximum length of user data and routine names within programs. Both of these problems were of lesser impact. We also found one arithmetic library function whose name was completely different on various machines, requiring changes in our Sisal source code.

The final impediment to portability was more of an inconvenience than a problem, because it did not affect our ability to run, or the answers we generated, on our target machines. We were simply unable to arrive at a standard method of timing our codes. Some of the machines had high-resolution clocks accessible through system calls, while others had lower-resolution UNIX timing routines. At one point, we were reduced to using the UNIX shell command "time" to time a run.

Notwithstanding the above, we believe the FLI offers two distinct advantages to the parallel programmer. First, it provides a means of rapidly parallelizing existing application codes by concentrating programmer effort where it will provide the best return. Second, it offers a developmental path for codes ranging from experimentation on cheap workstations to production on expensive supercomputers.

Indeed, we used just such a strategy in the work described here. When rewriting the computational core of F_MIN, we used a small microprocessor-based multiprocessor system manufactured by Silicon Graphics, Inc., as well as uniprocessor workstations of various sorts. We ran our hybrid code for verification purposes on an IBM RS/6000 system. And we executed performance runs on a multiprocessor Cray C-90 supercomputer and a large SGI multiprocessor system.

The problems outlined in this subsection were neither different nor worse in their impact on our hybrid code than would have been the case for an ordinary port of an imperative language code between machines. Indeed, such differences between system hardware and software are exactly the reasons why porting is so onerous a task.

5 IMPLEMENTATION OF THE HYBRID CODE

In implementing F_MIN in Sisal, we encountered a known but unaddressed deficiency in the language, namely, the lack of general user-defined reduction operations. This shortcoming became more apparent during this work. We addressed the problem experimentally in our code, but we will describe the problem and our efforts only superficially here. A more formal treatment of it can be found in [11]. We also describe the available schemes for structuring the expensive and parallelizable portion of the code, and the execution behavior of each.

5.1 Reduction Operations

Scientific codes often contain paired expressions: A computation that generates a set of values and a reduction operation that reduces them to a single value. Sisal supports seven reduction operations: sum, product, least, greatest, array, stream, and catenate. However, a more general linguistic

mechanism is needed for user-defined reductions. Histogram operations, in particular, tend to arise frequently in Sisal programming, but other sorts of reductions have turned up as well.

Sisal's functional semantics make it difficult to express and exploit the parallelism of general reduction operations. This is partly because a reduction can carry hidden states between its initiation and completion, and partly because the programmer must make certain guarantees about user-defined reductions. A full discussion of these matters is beyond the scope of this article, but we have determined that some reductions (either built-in or user-defined) are inherently associative (i.e., "catenate"), some are inherently commutative (i.e., floating-point addition), some are inherently both associative and commutative (i.e., integer addition), and some may be inherently neither. In the most general case, the operational semantics of loops in which these reductions appear must be adjusted so as to produce determinate results, in either sequential or parallel executions, as appropriate. Doing so may require advice from the programmer.

There are two possible implementation strategies for reductions that require minimal or no changes to the compiler [12]. The first is to execute the computation and reduction expressions sequentially, an undesirable strategy when parallelism is a goal. The second is to execute the computation expression in parallel storing the values in an array, and then execute the reduction expression sequentially, allowing us some parallelism at the expense of increased memory usage. A case where this arises is in the accumulation of the per-atom forces calculated in F_MIN. The forces on a target atom *i* generated by each neighbor atom *j* may each be calculated and stored independently, but all such force values must then be accumulated.

To retain parallelism and yet eliminate the need for temporary storage, we developed compile-time analysis to recognize pairs of computation–reduction expressions, and to fuse them into a single parallel loop [12]. Each iteration of the fused loop computes one value, locks the shared accumulator (the storage for the reduced value), executes the reduction operation, and unlocks the accumulator. Our analysis does not guarantee that the reduction operation is commutative or associative. A variation of this scheme has each worker process generating each accumulated value locally, so that locking is not needed. Then, after all workers have completed their work, the partial results are further reduced, using locks.

5.2 Parallelization Schemes

There are, in general, two ways to parallelize the calculation of forces in a particle dynamics code. We experimented with both of them, in our work on routine NON_BOND, and with two variations of the second implementation. Each scheme affects performance differently on the two sorts of parallel system architectures we used. We were strongly motivated by the desire to have a single version of the code for both types of system architectures. This led to a long series of experiments (which we will not describe here) forcing us into compromises in the structure of our code. We now describe the two general code organizations we dealt with, and the one "best compromise" version we ultimately settled on.

In the first scheme, the forces on all the particles by each of their neighbor particles can be expressed in a nested loop; the outer loop ranges over all the atoms and the inner loop ranges over all the neighbors of the current atom. The code looks like the following:

```
for i in 1, n_atom cross
    j in n_neighbors (i)
     fx, fy, fz :=
       force (i, j, pair_list)
returns array of fx
        array of fy
        array of fz
end for
```

Each body of the inner loop calculates the vector components of the force on a single atom caused by a single neighbor. The output of this loop is a set of two-dimensional arrays of forces, with each row representing all the forces on a single atom, and with one full two-dimensional structure for each vector component of the forces. This scheme is quite parallel, both in the generation and reduction of the force arrays, but has performance problems of three sorts. First, it allocates 3000 row vectors each time the computation executes, and the individual atoms have associated rows of varying lengths. Worse, the rows themselves vary over the life of the program, because the sizes of the neighbor sets vary. This causes the memory management mechanisms within the Sisal RTS to spend a large amount of time allocating and deallocating memory, which cannot easily be cached and reused due to the constantly changing sizes of the structures it must manage. It also causes the program as a whole to need a very large amount of

memory in which to run. The second performance penalty caused by this scheme is its lack of long vectors which are efficiently exploitable by Cray-type vector architectures. The third penalty is the load imbalance due to the varying sizes of neighbor sets.

In the second parallelization scheme, the forces are calculated in a single loop that ranges over all the bonded pairs of atoms. We refer to this scheme as the "pair-loop version." The code looks roughly as follows:

```
for k in 1, num_bonds
    i, j := pair (k, pair_list);
    fx, fy, fz :=
       force(i, j, pair_list)
returns array of i
        array of j
        array of fx
        array of fy
        array of fz
end for
```

Each body of this loop calculates the vector components of the force on a single atom, as before. However, the output of this loop is a set of one-dimensional arrays of forces, with each element representing the force of a single bond pairing. The reduction of this array is complex, since the forces on a single atom are not naturally grouped in the three component arrays, as they were in the rows of the two-dimensional structure generated from the previous scheme. This scheme carries its own performance penalties, caused basically by memory contention. When the atom and neighbor indices, `i` and `j`, are read, each instance of the loop body reads a different `j` but many instances read the same `i`. Further, the arrays of force components, represented by the names `fx`, `fy`, and `fz`, are ordered by the values in the arrays of index values, `i` and `j`. One natural ordering is shown below:

```
i: 1 1 ... 1 2 2 ... 2 ...998 998 999
j: 2 3 ...n1 3 4 ...n2 ...999 1000 1000
```

Here, n_1 is the highest atom index among atom `i`'s neighbors, and similarly for n_2, etc. This ordering arises from the way the pair list is generated, with all the neighbors of a given atom clustered together. Here we see that all the forces for atom 1 will occur first in the force arrays, followed by all the forces for atom 2, and on through atoms 998 and 999. (No bond-pair is duplicated in this scheme.) Shuffling the pair list provided a 10% speed improvement on the Cray system, but caused cache misses on the SGI system.

The two basic organizations for the force calculations were combined in our code with experimental reduction optimizations. The first scheme admitted a simpler reduction, but resulted in more memory usage. The second used less memory, but ran slower, due to vectorization failure in the reduction operation. To aid in understanding this, consider the following code fragment from the force reduction function:

```
for initial
   k    := 0;
   f    := f_in;
while k < num_bonds
   k    := old k + 1;
   i, j := pair(k, pair_list);
   fi   := old f[i] + new_f[i];
   fj   := old f[j] - new_f[i];
   f    := old f[ i: fi; j: fj ]
returns value of f
end for
```

Here we see the code responsible for reducing the forces on a single atom. For good performance, this loop must be vectorized, but cannot be automatically judged safe for vectorization due to the indirection. Consequently, we restructured the reduction as a nested loop pair, with the outer loop running across all atoms i and the inner loop running across all neighbors of i. Now the inner loop over the neighbor list can be vectorized since an atom can appear only once in each neighbor list. To force vectorization, a compiler pragma was manually inserted into the C code generated by the OSC.

In the next section, we report the execution time of the pair-loop version for the Cray and the SGI systems. We decided that we had succeeded in our goal of producing a single version of our hybrid code that ran reasonably well on both sorts of architectures. While different organizations would improve performance on either machine, these are not germane to our arguments here. One might argue that the pragma we inserted into the C code to improve vectorization on the Cray violated that goal, but in fact the Sisal and Fortran sources were unchanged, and the pragma itself would be a comment to any C compiler other than the one on the Cray system.

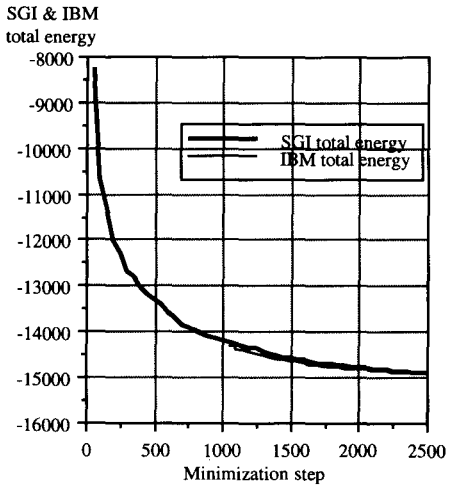

FIGURE 2 BORN total energy statistic from Sisal runs on IBM 540 and SGI 340 showing minimization.

6 PERFORMANCE

We performed numerous tests, with the two basic versions of the F_MIN code, on several computer systems. The first, and most basic, test involved verifying that the answers were acceptable. A direct comparison via a file differencing utility was not practical, since the output files from various executions were written with different precisions, and in fact contained different result values, due to numeric causes outlined earlier. We surmounted this difficulty by observing the minimization process as it evolved over the specified number of iterations. A sample comparison is shown in Figures 2 and 3 that plot the value of the system's total energy.

Figure 2 shows the Sisal answers generated on two different computers under sequential execution. The two curves diverge by a relatively small amount, for a number of steps, and then appear to reconverge. To further check this, however, we calculated the actual differences for this pair of runs and plotted them on the same horizontal axis (Fig. 3) wherein the reconvergence is confirmed. Particle dynamics applications tend naturally to show strong dependencies on initial conditions and operational ordering, so neither the transient divergence nor the eventual reconvergence was surprising to us. However, we also observed the behavior of both language versions on other computer systems, and confirmed that the Fortran and Sisal versions produced qualitatively consistent convergence behavior. Their answers compared as favorably as the two Sisal runs shown on all platforms. The small transient divergences appeared on all comparisons of runs of different versions of the code, comparisons of runs on different platforms, and comparisons of runs on different numbers of processors. However, the aggregate behavior of the codes always adhered to that shown, namely a coarsely monotonic decrease in energy that asymptotically approached a minimum.

We usually did not run to full convergence, since this required very many steps. However, we did perform one very long run, just to make sure the code actually would converge, and the total energy for this run is shown plotted in Figure 4. In practice, the application scientist would determine when a run had produced "good enough" answers, and would run different problems for different numbers of steps for that purpose. In fact, the system we modeled was a "sanity check" used by application specialists to check the results of modifications and ports of the original Fortran code. All of the other runs we cite here were of this standard problem, terminating after either 100 or 1000 steps.

We show performance comparisons in Table 1. This table shows the run-times and calculated speedups for the Sisal version of the code on two computers. Note that the sizes of the runs differ on the two machines by a factor of ten, allowing for similar run-times. Problem size and run-time limits were dictated by machine availability and economics. However, the two machines provide

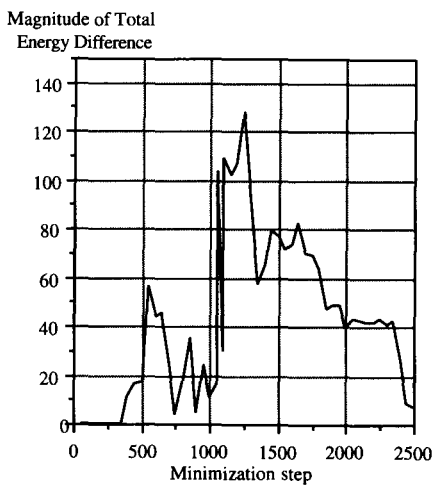

FIGURE 3 Magnitude of differences in total energy statistics from Figure 2 showing reconvergence.

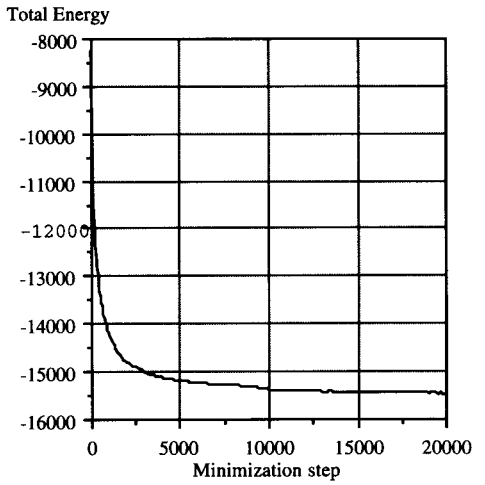

FIGURE 4 BORN total energy statistic from a very long run.

similar parallel speedups. The code is designed to simulate atom ensembles much larger and more complex than the one we had, and we hope to get more experience with it in the future. Running for larger numbers of minimization steps does not seem to change these results on either Cray or SGI systems. For comparison, the Fortran version ran 1000 steps on the Cray C-90 in 86.5 s and 100 steps on the SGI-340 in 169.3 s.

CONCLUSIONS

In this article, we report on the development of multiple parallel versions of an MM simulation code using hybrid imperative and parallel functional methods in a mixed-language compilation and execution environment. The imperative portion of the hybrid code stayed essentially unchanged from its form in the original Fortran program. The computational kernel was reimplemented in the functional language Sisal to exploit the available parallelism. The hybrid code started the Sisal RTS and invoked the Sisal kernel, passing arguments from Fortran's storage space into Sisal, which returned its results to the Fortran calling level. The invocation of, and communication with, the Sisal kernel occurred within the force minimization loop in the Fortran code. Several conclusions are apparent from this work.

First, it demonstrates clearly the practicality of mixed-language programming models. Second, the resulting hybrid code is portable and can be run on various platforms, generating correct results with good performance and reasonable parallel speedups. We exploited parallelism with a single version of a hybrid code on different execution platforms. Third, while we would wish for freedom from architectural concerns in our programming efforts, this is not yet achievable. Tuning remains necessary to achieve acceptable performance. This does not rule out portability, however.

A less positive conclusion must be drawn on the strategy of compiling Sisal into C. While this has been a successful design decision in general, it is also true that producing a hybrid program in the manner described here requires the programmer to deal with three languages, the original Fortran, Sisal, and the C produced by Sisal.

Table 1. Sisal Code Performance on Two Platforms

System Used	No. Steps	No. Workers	Run-time (s)	Speedup
Cray C-90	1000	1	96.47	1.00
Cray C-90	1000	2	63.08	1.53
Cray C-90	1000	3	53.36	1.81
Cray C-90	1000	4	46.86	2.06
Cray C-90	1000	5	39.26	2.46
Cray C-90	1000	6	35.46	2.72
SGI-340	100	1	194.60	1.00
SGI-340	100	2	118.57	1.64
SGI-340	100	3	97.50	2.00

ACKNOWLEDGMENTS

We thank Cray Research Incorporated for use of their C-90 and for providing us dedicated time to run our experiments. We also thank our colleagues Scott Denton, Patrick Miller, and Srdjan Mitrovic for invaluable assistance in understanding the work reported here. This work was supported by Lawrence Livermore National Laboratory under DOE contract W-7405-Eng-48 and the DOE Technology Transfer Initiative.

DISCLAIMER

REFERENCES

1. J. R. McGraw, S. Skedzielewski, R. Oldehoeft, J. Glauert, C. Kirkham, B. Noyce, and R. Thomas. *Sisal: Streams and Iterations in a Single-Assignment Language, Language Reference Manual, Version 1.2. Lawrence Livermore National Laboratory Manual M-146*, Rev. 1: Livermore, CA: Lawrence Livermore National Laboratory, March 1985.
2. J. T. Feo, D. C. Cann, and R. R. Oldehoeft, "A report on the SISAL language project," *J. Parallel Distrib. Comput.*, vol. 12, pp. 349–366, Dec. 1990.
3. R. W. Hockney, and J. W. Eastwood, *Computer Simulation Using Particles*. New York: McGraw-Hill, 1981.
4. M. P. Allen and D. J. Tildesley, *Computer Simulation of Liquids*. Oxford: Oxford Science Publications, 1987.
5. S. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," Sandia National Laboratories, Albuquerque, NM, Tech. Rep. SAND 91-1144, 1993.
6. D. C. Cann, "The optimizing Sisal compiler: Version 12.0," Lawrence Livermore National Laboratory, Ames, IA, Tech. Rep. UCRL-MA-110080, 1992.
7. D. Cann, "SISAL 1.2: "A brief introduction and tutorial," Lawrence Livermore National Laboratory Technical, Ames, IA, Tech. Rep. UCRL-MA-110620, 1992.
8. W. H. Press, et al. *Numerical Recipes*. Cambridge, MA: Cambridge University Press. Chapter 10.
9. F. F. Abraham, "Computational statistical mechanics: Methodology, applications and supercomputing," *Ad. Phys.*, vol. 35, pp. 1–111, 1986.
10. L. Verlet, "Computer experiments" on classical fluids, I. Thermodynamical properties of Lennard-Jones molecules, *Phys. Rev.*, vol 159, pp. 98–103, 1967.
11. S. Denton, "Optimizing parallel reduction operations," Master of Science Thesis, U. C. Davis, June 1995.
12. S. Denton, J. Feo, and P. Miller, "Realizing parallel reduction operations in Sisal 1.2," in *Proceedings of the Working Conference on Parallel Architectures and Compilation Techniques*, Montreal, Canada, August 1994.