AIIM

Association for Information and Image Management

1100 Wayne Avenue, Suite 1100
Silver Spring, Maryland 20910
301/587-8202

Centimeter

Inches

MANUFACTURED TO AIIM STANDARDS
BY APPLIED IMAGE, INC.

Developing Molecular Dynamics Simulation Codes Using Mixed Language Programming

Thomas DeBoni, John Feo, Hugh Caffey, and Frederick Hausheer

ABSTRACT. We describe our experiences parallelizing a large-scale scientific application to model systems of discrete particles. We describe the approach and tasks undertaken to parallelize this application using two different programming paradigms: imperative and functional. The objectives of both exercises were to maximize performance, parallelism and portability, and to minimize program development costs. We believe this study reveals an important relationship between conventional and novel parallel programming paradigms, and identifies important attributes that novel paradigms must have to gain wide acceptance.

1. Introduction

In this paper we describe our experiences parallelizing a large-scale scientific application to model systems of discrete particles. The program is a portion of a code suite used to design anti-cancer drugs and is one step in a complex and computationally expensive process. The use of multiprocessor systems to perform these computations in parallel is of great interest from the points of view of both the practical biomedical application, and parallel programming languages and tools.

The program we work with is called Born. It performs the force minimization steps of models of complex protein and DNA molecules. These steps find minimal energy shapes for complex molecules with many degrees of rotational freedom, and permit us to simulate Newtonian dynamics. In turn, we can study the chemical interactions between the protein molecules, drug molecules, and solvents.

We describe the approach and the tasks undertaken to parallelize this application using two very different parallel programming paradigms: imperative and functional. Imperative programming requires the programmer to specify both the program's computations and implementation details, such as spawning worker processes, scheduling tasks, partitioning data, sending and receiving data, and synchronizing concurrent accesses to shared data. The imperative paradigm is the basis of most conventional programming languages such as Fortran and C. Consequently, it is the paradigm most familiar to application programmers and in which most sequential and parallel applications have been developed. To study functional programming, we use the functional language Sisal (3,4,5,9) developed by Lawrence Livermore National Laboratory and Colorado State University.

MASTER

yr

Sisal allows the specification of parallel algorithms in a notation similar to mathematics, and in which the interactions among the parallel tasks of a program are implicit. The programmer is responsible only for specifying the program's computations. The code consists of a single monolithic source program from which parallelism is extracted and exploited automatically. Functional languages simplify the parallel programming problem and can significantly reduce development costs.

In Section 2, we examine the application and the program in its original sequential Fortran form. In section 3, we discuss the approach taken in the imperative study and how it influenced the approach taken in the Sisal study. We believe these facts have far reaching consequences for researchers developing new parallel programming languages and paradigms. In sections 4 and 5, we present the details of the ports focusing on the expressability, portability, and development costs. In Section 6, we summarize our experiences and present our conclusions.

2. The application

Molecular dynamics is a methodology for modeling the interactions among ensembles of discrete atoms or multi-atom aggregates. The simulated particle behaviors typically are based on pairwise interactions, although larger interacting groups may also be used. The interactions are modeled by algebraic expressions for the electrical potentials among interacting particles. These expressions may take several possible forms based, among other concerns, on whether the particles are bonded—chemically interacting—or not bonded. The formulae for the potentials are given as functions of separation distance, or radius, and other material parameters used to adapt the shapes of the potentials. The important state characteristics of each atom or molecule are: mass, charge, chemical composition, spatial coordinates, and velocities. The first three characteristics are usually static, whereas, the last three characteristics vary during the simulation. For aggregates, we use center of mass values and consider rotations as well as linear motions.

In the case of bonded interactions, the potential expression typically has a minimum at the ideal equilibrium separation. The non-bonded interactions are usually modeled by monotonically decreasing functions of separation. Non-bonded interactions are sometimes used to model the effects of molecules on each other when separated by distances too great to allow individual chemical interactions. Both of these models may be easily parameterized to allow simulation of differing states of temperature, pressure, presence of solvents or catalysts, states of ionization, or other characteristics of real matter.

In modeling large and complex molecules, their shapes will affect how they interact with each other, through the interactions among their constituent atoms and radicals. A large molecule such as a protein typically has many sites where rotation of portions of the molecule can occur, affecting the three-dimensional shape of the whole. The shape of a molecule may be a function of its temperature, a solvent medium, and the presence of other molecules, such as enzymes. In general, it is not possible to specify an exact shape for such a molecule, even in isolation, and in any realistic situation it will likely be constantly changing shape, possibly oscillating about a quiescent or equilibrium configuration, at both large and small scales of size and distance. An important part of molecular dynamics modeling is to find approximate minimum energy configurations to use as starting points for further modeling. It is this minimization which Born calculates.

Figure 1 – A graph of the potential and force expressions. The vertical axis units are artificial and the horizontal axis units are angstroms

2.1. Molecular dynamics. An example formula for the potential for particle interactions as a function of separation is

$$U(r) = \left(\frac{1}{2ab} \right) e \left(1 - e^{-a(r-re)} \right)$$

and the first derivative of the potential function is

$$F(r) = U'(r) = \left(-\frac{1}{b} \right) e^{-2a(r-re)} + \left(\frac{1}{b} \right) e^{-a(r-re)}$$

where r is the separation, re is the equilibrium separation, and a and b are material parameters. The second equation gives the force generated between a pair of particles as a function of their separation. Abstract plots of the above two expressions are shown in Figure 1. The minimum of the potential function defines the separation distance where inter-particle forces are zero.

To use this in dynamical modeling, we must solve the pair of coupled first-order differential equations that describe Newton's laws of motion. We do so by calculating the acceleration of each atom as the force incident on the atom divided by its mass. However, in the present application, we do not consider the free motion of the particles, but instead consider only the forces among the particles at various configurations. Therefore, we calculate forces for several configurations of the molecule differing in rotation of molecular segments, and define the shape of the molecule as the configuration of minimal total force.

Figure 2 – Call tree of the sequential Fortran program

2.2. Force minimization. Born is the force minimization module of BNPI's (BioNumerik Pharmaceuticals, Inc.) UltiMol package—a procedure for the iterative minimization of forces acting among simulated atoms. Force minimization is the main computational kernel in molecular dynamics simulation, in calculation of Gibbs free energy, and other computational chemistry applications. (The terms “force minimization” and “energy minimization” are often used interchangeably to describe this application.) At the highest level, this program is a series of minimization steps. In each iteration, we must compute inter-atomic energies from several sources. Figure 2 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 stores the total force on each atom.

Born is typical of large scientific applications in four respects: 1) most of the code reads inputs, initializes data structures, defines the initial configuration, collects statistics, and writes outputs; 2) the application includes a well defined computational kernel that performs the desired simulation; 3) the computational kernel is approximately 30% of the whole code; and 4) more than 90% of the execution time of the program is spent in the computational kernel. Born is somewhat atypical of scientific codes in that one module of the computational kernel, NON_BOND, dominates the entire application. This module calculates the electrostatic forces acting among non-covalently bonded pairs of atoms. It

computes the $N * (N-1) / 2$ pairwise electrostatic forces acting among the N atoms. It is one of several variations of the N -body problem (7). To compute a simple non-bonded pairing takes 40 floating-point operations (flops), whereas, to compute a hydrogen non-bonded pairing requires 53 flops.

To reduce the $O(N^2)$ computational complexity of this problem (2,11), we employ a commonly used approximation. Rather than consider all $N * (N-1) / 2$ possible pairings of atoms, we consider for each atom i only those atoms j within a certain "cutoff radius" in three dimensions. The smaller the radius, the fewer the number of non-bonded pairings. For most of the simulations used in our research, cutoffs greater than 8–10Å are not warranted.

The cutoff approximation requires the construction of a "pairlist" of atoms. This list is the governing data structure for the non-bond 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, and each atom i may have a different number of pairings, or neighbor atoms. Thus, the pairwise computations exhibit poor data locality and load balancing, an important performance consideration on parallel, hierarchical memory architectures.

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 non-bonded pairings are computationally independent, we may compute them concurrently. A large problem of 35,000 atoms and 10,000 steps can take many hours to solve on a sequential, vector supercomputer, but it will run significantly faster on a multiple processor system. In general, parallel processing has proven effective in dealing with this class of simulated system (1, 10). Since NON_BOND is such a large percentage of the total work, if we parallelize this one routine, we will have for all practical purposes parallelized the entire application.

3. Parallel implementations

An important objective of our research is to understand how parallel programs are developed, and to compare development procedures using imperative and functional programming languages. Most parallel programs are not new programs. Programmers do not write them from scratch, but instead begin with an existing sequential program written in an imperative language, typically Fortran 77. The Born application we consider in this study is a true legacy code. It is written in Fortran 77, makes substantial use of common blocks and sundry Fortran tricks, is strictly sequential, and structured for a vector supercomputer.

The programmer 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 parallelize only those subparts that benefit the most from parallel processing. By considering only these "parallel" sections, the imperative programmer maximizes performance and minimizes development costs.

Due to old programming styles and years of additions and changes, legacy codes are convoluted. In our experience, code that cannot be read and understood cannot be parallelized. Thus, the first step is to restructure and rewrite the parallel sections. The user may find it necessary to redesign control and data structures to get correct parallel execution and high-performance. Opportunities for more mundane clean-up actions also present themselves, such as removing or regularizing Fortran common blocks, re-engineering or removing packed data structures, and adding modularity where it was missing but advantageous to have. Some of these are necessary for execution purposes and some for human understanding. Moreover, the sequential and/or vector character of the original code must be relaxed. It is not unusual in the course of these activities to uncover logical errors in the original code as well as errors that come to light only under parallel execution.

Once the code is cleaned-up, the program may be parallelized. In sections four and five, we discuss the development of three different parallel versions of Born. The first two versions add support code to the original Fortran code. The third version is a hybrid of the original Fortran code and new Sisal code. The first implementation uses point-to-point, send-receive messages. The second uses a shared memory programming model in which interprocess communication is implicit. Both of these versions use a master-worker model in which a single master process spawns multiple identical worker processes. In each minimization step, workers are used solely to compute concurrently the non-bond electrostatic forces. The bulk of the programming work involved the specification of implementation details, such as the spawning and scheduling of worker processes, the partitioning of tasks, communication of data, and synchronization. For the rest of this discussion, we will refer to the first two parallel implementations as conventional parallel versions.

We wrote the third version using Sisal. Functional languages elevate the level of program abstraction by enforcing mathematical principals and referential transparency. Functions map inputs to outputs without side-effects and do not retain state between invocations. Consequently, programs can be compiled to exact dependency graphs that identify all data and control dependencies, communications, and synchronization. Functional languages simplify the programming problem by eliminating the need to specify implementation and machine architecture details in the source code. The Sisal program encodes the mathematics of the computation and then relies on the compiler to extract parallelism and introduce the necessary code for communication, synchronization, and task scheduling.

When writing a new code from first principles, the Sisal programmer has the luxury of starting with mathematical equations and the computation's logical structure; however, such opportunities are rare. Since the legacy code will frequently include I/O and interface modules that cannot be changed and are not easily expressed in a functional language, redesigning and rewriting the entire code is not an alternative. Consequently, the Sisal programmer must identify the most judicious use of Sisal and proceed accordingly.

The questions to ask are what does Sisal do well and where can it do the most good. Functional languages simplify the expression of mathematical computations and reduce the cost of writing parallel code. Sisal's high-level syntax and mathematical semantics provide a more natural and efficient medium for expressing computations than Fortran or C. A Sisal program, which is implicitly parallel and portable, is typically 75% as long as the equivalent sequential imperative program and typically takes around half as much time to develop. Given that parallel imperative programs can be 4 or 5 times as expensive to write as sequential programs, Sisal can substantially reduce programming costs if the Sisal and the Fortran programmer write the same amount of code. If the Sisal programmer must rewrite the entire application (for instance, if Sisal and Fortran code cannot co-exist), while the Fortran programmer rewrites only the parallel sections, then developments costs will be comparable. It is paramount, therefore, that the Sisal compiler, runtime system, and programming tools support mixed-language programming. The Sisal programmer must be able to use as much of the imperative code as warranted. We believe this observation is true for all new programming languages and paradigms. No new approach will gain acceptance if it requires that the millions of lines of extant code be thrown away.

4. Two conventional implementations of Born

In this section, we describe two parallel versions of Born using conventional methods. Since NON_BOND is the most important routine, we focus our discussion on it. After cleaning up the Fortran code, we analyzed the scoping and use of every symbol found in the subroutine. This analysis is necessary to avoid race conditions on shared variables. We assigned each symbol to exactly one of the following four categories:

1. local;
2. global – never modified after initialization;
3. global – possibly modified every iteration, but only outside NON_BOND; and
4. global – possibly modified every iteration inside NON_BOND.

Local symbols present no problem for safe parallel execution; however, global symbols must be handled carefully. These and other details are implementation-specific and are discussed below.

In both conventional versions, NON_BOND consists of a doubly nested loop. The outermost loop runs over all N atoms

$$do\ i = 1, N$$

and the inner loop is

$$do\ j = 1, jn$$

where jn is the number of neighbor atoms of i . The j loop reads the index of an atom paired with i from the pairlist, and then calculates the non-bond force between i and that atom. For all practical purposes, the number of non-bonded pairings computed, *total_pairs*, represents the total amount of computational work in the parallel kernel. To

achieve good load balance, we must partition and schedule the *total_pairs* pairs as equitably as possible among the *n_workers*.

Since *jn* varies widely among the *i* atoms, simply dividing the outer loop by the number of workers results in a poor load balance. If we divide *total_pairs* by *n_workers*, two processes will likely share responsibility for the same *i* atom requiring us to store information on *i* at two sites. This duplication of information wastes storage and increases communications. A more equitable, although not perfect, solution is to divide the outer loop by more than the number of workers, and dynamically assign the chunks to the workers.

We partition the *N* atoms into *n_chunks*, or subsets, where *n_chunks* is a multiple of the number of worker processes. The value of the multiple is a runtime parameter and is typically in the range 1 to 8. If the multiple is 1, then each worker gets one chunk. For larger multipliers, the first *n_workers* chunks are scheduled, and then the master assigns the remaining

$$n_chunks - n_workers$$

chunks to worker processes on a first-come, first-served basis. This "pool of tasks" dynamic scheduling scheme has been found to improve load balance and parallel speed-up. Note that if the multiplier is too large, then the partitions become too fine-grained, increasing overhead and communications. The outer loop over *i* in the non-bond force routine for each worker becomes

$$do\ i = l_bound(chunk_no), u_bound(chunk_no)$$

4.1. Message-passing. Distributed memory machines are a class of parallel processing systems built from multiple processing elements interconnected by a communication network. Each element includes at least a CPU and memory, and the memories are disjoint. Communications and synchronization are realized by passing messages among processing elements. PVM (6) is a library of control and communication routines available on most distributed memory systems. PVM gives the programmer a great deal of flexibility and control over the spawning of worker processes. In this application, we implement a master-worker model and permit only one worker per node in the parallel virtual machine. We also provide a user-selectable flag for the run-time specification of whether or not a worker process may be spawned on the same node as the master. Thus, the number of worker processes is either equal to the number of processors or is one less.

In the PVM version, each worker redundantly computes its own private copy of the pairlist. This duplication of work greatly reduces the infrequent, but large messages, that the master would otherwise have to multicast to the workers. Because of the infrequency of updating and the relatively few flops involved in pairlist generation, the amount of redundant calculation is a small price to pay for the reduction in communication achieved. Either way, every worker must store the full pairlist; the storage can amount to several MB for large problems. (We use an alternative approach in the shared memory implementation discussed later.)

Global symbols initialized early in execution (before iteration begins) and used on a read-only basis by NON_BOND are sent once, as early as possible, in a PVM multicast to

all workers. Global symbols possibly modified outside NON_BOND between minimization steps are multicast to all workers at the beginning of each step. These values include a scalar flag that signals whether or not the pairlist should be updated in this step.

The main output of NON_BOND is the force array, f . After each worker computes the non-bond force between two atoms, it updates f accordingly. Since the updates occur haphazardly and must execute in a critical section, they can introduce a severe bottleneck and increase communications. One solution to the problem is to have each worker maintain a local, fully dimensioned force array and make all updates local. When the worker receives a special "no-more-chunks" signal from the master, it sends its force array to the master, who adds together serially the workers' force arrays as they arrive. On small, fine-grained runs (small problem and/or many workers) with good load balance, this step can become a serial bottleneck as workers contend for the master's attention. After a worker has sent back its results to the master, it re-initializes various local values and then waits on a blocking receive for a signal from the master that tells it whether or not there are more minimization steps in the run. If yes, the worker waits for a multicast from the master sending it new values for the next iteration, such as new particle coordinates. On receiving these values, the worker begins computing its first chunk of the next step. If there is no next step, the worker exits.

We took several measures to minimize the wall clock costs of PVM communications and to hide latency. We used *pvmfinitend(PVMRAW)* to avoid unnecessary data encoding and decoding, and *pvmfadvise(PVMROUTEDIRECT)* to enable direct process-to-process routing via TCP. Additionally, we reorganized the code to overlap computations and communications.

4.2. Shared memory. The second implementation was written for the IBM Power/4 system [8]. This system supports shared memory. The spawning of workers in this environment occurs via a call to the P6K_CALL library routine. The routine's first argument is the name of the parallel subroutine to be executed. All processors initiated for the run, including that of the master, participate in this execution. Thus,

$$n_workers = n_processors$$

In the shared memory version, unlike the message-passing versions, the master process generates and updates the pairlist. Instead of each worker process generating a separate copy of the pairlist, the master generates a single instance in global shared memory where it is accessed by all workers on a read-only basis. In general, all large arrays, for example the coordinate and charge arrays, and sundry scalars that are only read by the NON_BOND workers are stored in shared global memory.

As in the message-passing version, workers in the shared memory version maintain local force arrays. Unlike the message-passing version, however, workers in the shared memory version are fully self-scheduling. At the beginning of each minimization step, a globally shared "chunk pointer" is initialized to zero. On entry to the NON-BOND routine, each worker attempts to enter a small critical section implemented using a shared, named lock. On entering the critical section, the worker increments the value of the shared chunk pointer and assigns this value to a local chunk pointer, *chunk_no*. It then relinquishes the lock and tests the value of *chunk_no* to see if it is greater than the number of chunks. If

not, it begins execution of that chunk of atoms. If all chunks have already been scheduled in the current step, it branches to another small critical section controlled by a second shared, named lock. On successful entry of this section, it adds its local force array to the globally shared force array, releases the lock, calls a barrier synchronization routine to note its imminent exit, and then exits.

5. Functional Parallelism in Sisal

The scheme for porting this code into Sisal for parallelization was slightly different from that used in the two conventional versions. Once we had cleaned-up the Fortran code, we rewrote the computational kernel in Sisal. The rewrite was quick and easy, and we did experiment with several different expressions of the problem. We then integrated the Sisal code into the original Fortran program, compiled the mixed program, and ran it. The Sisal software system supports a mixed language programming capability called the Foreign Language Interface (FLI). The FLI allows Sisal programs to call or be called from Fortran or C. This allows recoding only the computational kernel of an existing code for parallelism, as in the current experiment. Alternatively, Sisal shells can invoke existing libraries or solvers across the FLI.

The FLI supports our belief that legacy codes need not be completely rewritten to allow for parallel execution. Using this feature, we rewrote the core of the Born code, leaving unchanged the work best performed in the original sequential Fortran. The Sisal portion of the hybrid Sisal/Fortran code (hereafter called the Sisal version) consists of the module named "calc_forces" in Figure 2. Thus, we rewrote NON_BOND as well as the other force calculation routines. Recoding the other force calculation routines added little 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 two calls within RUN_PROB to start and stop the Sisal Runtime System (RTS).

The current practicalities of parallel systems dictate that most effectively exploitable parallelism arises from slicing loops, and the Sisal language software excels at this. Sisal contains a statement type that implicitly specifies parallelism—the *for expression*. Unlike parallel loop constructs in imperative languages, the *for expression* guarantees correctness and determinacy. The Sisal portion of the force minimization program consists of 26 functions containing a total of 17 parallel loops. Each of these loops is an opportunity to exploit parallel and vector execution based on decisions made by the Optimizing Sisal Compiler (OSC), the runtime system, and compile time and run time parameters specified by user. The latter 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 is the case with the conventional implementations, most of the parallelism (and most of the work) is in NON_BOND. OSC's advanced capabilities allow optimizations across function boundaries that are not routinely possible in conventional languages, such as loop fusion. These optimizations reduce total instructions, eliminate intermediate aggregates, eliminate copy operations, reduce reference counts, and decrease overall memory usage. They also allow us to exploit parallelism in loops where it would not otherwise be economically practical, and increase vectorization in processors where it is an option. In summary, rewriting the

force calculation routines in Sisal took very little programmer time and did not require any "implementation" code.

The Sisal Runtime System consists of a microtasking kernel and memory management routines that automatically manage the system resources during execution. Given the 17 parallel loops in the Sisal version of the code, the RTS can automatically slice and execute them in parallel, thus managing the partitioning and scheduling tasks that we had to do manually in the conventional versions. Since Sisal programs are guaranteed to be determinate and free of deadlock, the results of execution with varying numbers of worker processes do not vary, and the programmer need never debug a race condition in the parallel Sisal code.

The RTS dynamically allocates memory for aggregate objects from a heap, and deallocates and reuses this memory efficiently. Our RTS reduces dramatically one of the traditional performance problems 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. There is no need (and in fact it is not possible) for the programmer to code locks, critical regions, or other synchronization or communications into the source program. All such coordination among parallel workers is done automatically at run time.

Because the original Fortran code expected all of its data items, scalars, vectors, and matrices of any and all types, to be stored in a large array in a common block, we had to do considerable work to set up the data needed by the Sisal portion of the program. The Fortran program accesses individual items through Fortran EQUIVALENCE statements that specify subsections of the large array. Heterogeneous aggregates of this sort are not allowed in Sisal. We passed the array by reference across the interface twice, once as a real and once as an integer. We then extracted the individual data items from the arrays so that they could occupy their proper places in function argument lists. The extraction did not copy any data, but created an array descriptor for each extracted array. Array descriptors are built for every Sisal array, and store the array's start address in physical memory, its lower bound, its length, and reference count information.

This code represents the most extensive use yet made of the Sisal Foreign Language Interface, and was, in addition, an application area we had little previous experience with. While constructing and experimenting with it, we encountered a known but overlooked deficiency in Sisal, namely, the lack of histograms and support for histogram operations. Histogramming is an example of a reduction operation—values generated in either a parallel or sequential loop reduced to a single value. Sisal does support seven reduction operations: sum, product, least, greatest, array, stream, and catenate. Sisal's functional semantics make it difficult to express the parallelism of reduction operations. To eliminate the inefficiency of executing separately the computation and reduction expressions, we developed compile-time analysis to recognize pairs of computation–reduction expressions, and to fuse them into single parallel loops.

One more point should be emphasized with regard to the Sisal version. Sisal currently runs only on shared memory machines. The automatic management of parallelism and machine resources in distributed memory systems is an area of active research. Our efforts in this area, while active, have not yet produced results capable of practical use in this sort of application context.

6. Conclusions

In this paper, we report on the development of three parallel versions of a molecular dynamics simulation code using two different parallel programming methods: imperative and functional. The imperative development first restructured and rewrote the application's computational kernel, and then parallelized only that segment of the code. The latter consisted of writing partitioning and scheduling code to manage worker processes, and inserting communications code to send data between the processes. The functional development started with the clean Fortran code, and rewrote it in Sisal. Once rewritten, the code was called from the Fortran program using Sisal's Foreign Language Interface. Note that both the imperative and functional studies considered the same number of lines of code. Preliminary studies indicate that the performance of the Sisal and Fortran versions are similar.

The significance of this work is that it demonstrates clearly the importance of mixed-language programming models. Functional languages do decrease the cost of writing parallel programs, but their advantages is negated if the functional programmer must rewrite an entire application while the imperative programmer rewrites only the application's computational kernels. To insure a high degree of integration, scientists developing new programming models and languages must consider support for mixed language programming from inception. Without such support, it is unlikely their ideas will gain wide acceptance.

Acknowledgments

This work was performed, in part, under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

References

1. Abraham, F.F., Computational statistical mechanics: methodology, applications and supercomputing, *Advances in Physics* 35 (1986), 1-111.
2. Allen, M.P. & D.J. Tildesley, *Computer Simulation of Liquids*, Oxford Science Publications, Oxford, 1987.
3. Cann, D., J. Feo, T. DeBoni, *SISAL 1.2: High Performance Applicative Computing*, Lawrence Livermore National Laboratory Technical Report UCRL-JC-103980, 1990.
4. Cann, D., R. Oldehoeft, *A Guide to the Optimizing SISAL Compiler*, Lawrence Livermore National Laboratory Technical Report UCRL-MA-108369, 1991.
5. Cann, D., *SISAL 1.2: A Brief Introduction and Tutorial*, Lawrence Livermore National Laboratory Technical Report UCRL-MA-110620, 1992.

6. Geist, A., A. Beguelin, J. Dongarra, W. Jiang, R. Manchek and V. Sunderam, PVM 3 User's Guide and Reference Manual, Oak Ridge National Laboratory ORNL/TM-12187, 1993.

7. Hockney, R.W. and J.W. Eastwood, Computer simulation using particles, McGraw-Hill, New York, 1981

8. IBM Shared Memory System POWER/4 User's Guide and Technical Reference. International Business Machines Corp. publication no. FST-0001-00.

9. McGraw, J., S. Skedzielewski, S. Allan, R. Oldehoeft, J. Glauert, C. Kirkham, B. Noyce, R. Thomas, SISAL - Streams and Iterations in a Single Assignment Language, Language Reference Manual Version 1.2, Revision 1, Lawrence Livermore National Laboratory Technical Report M-0146, 1985.

10. Plimpton, S., Fast parallel algorithms for short-range molecular dynamics, Sandia National Laboratories Technical Report SAND91-1144, 1993.

11. Verlet, L. 1967. Computer 'experiments' on classical fluids, I. Thermodynamical properties of Lennard-Jones molecules, Physical Review 159 (1967), 98-103.

Thomas DeBoni and John Feo
Institute for Scientific Computing Research
Lawrence Livermore National Laboratory
Livermore, CA 94550
deboni@llnl.gov, feo@llnl.gov

Hugh Caffey and Frederick Hausheer
BioNumerik Pharmaceuticals, Inc.
Suite 1250
8122 DataPoint Drive
San Antonio, TX 78229

DATE

FILMED

8/19/94

END