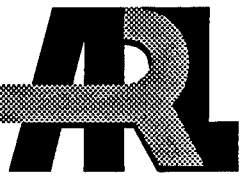
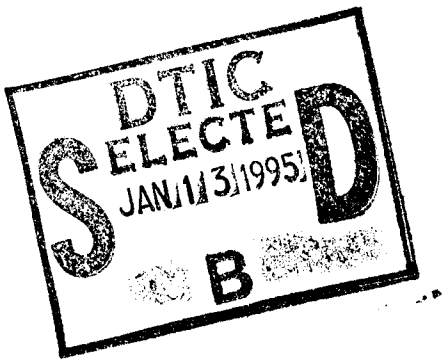
ARMY RESEARCH LABORATORY



Optimization Strategies for Molecular Dynamics Programs on Cray Computers and Scalar Work Stations

Michael J. Unekis
Betsy M. Rice

ARL-TR-661 December 1994



NOTICES

Destroy this report when it is no longer needed. DO NOT return it to the originator.

Additional copies of this report may be obtained from the National Technical Information Service, U.S. Department of Commerce, 5285 Port Royal Road, Springfield, VA 22161.

The findings of this report are not to be construed as an official Department of the Army position, unless so designated by other authorized documents.

The use of trade names or manufacturers' names in this report does not constitute endorsement of any commercial product.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1994	Final, Aug 93–Dec 93

4. TITLE AND SUBTITLE

Optimization Strategies for Molecular Dynamics Programs on Cray Computers and Scalar Work Stations

5. FUNDING NUMBERS

PR: 1L161102AH43

6. AUTHOR(S)

Michael J. Unekis and Betsy M. Rice

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

U.S. Army Research Laboratory
ATTN: AMSRL-WT-PC
Aberdeen Proving Ground, MD 21005-5066

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

U.S. Army Research Laboratory
ATTN: AMSRL-OP-AP-L
Aberdeen Proving Ground, MD 21005-5066

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

ARL-TR-661

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

We present results of timing runs and different optimization strategies for a prototype molecular dynamics program that simulates shock waves in a two-dimensional (2-D) model of a reactive energetic solid. The performance of the program may be improved substantially by simple changes to the FORTRAN or by employing various vendor-supplied compiler optimizations. The optimum strategy varies among the machines used and will vary depending upon the details of the program. The effect of various compiler options and vendor-supplied subroutine calls is demonstrated.

Comparison is made between two scalar workstations (IBM RS/6000 Model 370 and Model 550) and several Cray supercomputers (X-MP/48, Y-MP8/128, and C-90/16256). We find that for a scientific application program dominated by sequential, scalar statements, a relatively inexpensive high-end work station such as the IBM RS/6000 RISC series will outperform single processor performance of the Cray X-MP/48 and perform competitively with single processor performance of the Y-MP8/128 and C-90/16256.

14. SUBJECT TERMS	15. NUMBER OF PAGES
molecular dynamics, simulation, optimization	34
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

INTENTIONALLY LEFT BLANK.

ACKNOWLEDGMENTS

M. J. Unekis gratefully acknowledges the receipt of a National Research Council postdoctoral fellowship.

INTENTIONALLY LEFT BLANK.

TABLE OF CONTENTS

INTENTIONALLY LEFT BLANK.

1. INTRODUCTION

The field of molecular dynamics is of considerable importance to the molecular modeling community. The general approach of molecular dynamics is to treat the motion of particles classically, rather than by the more rigorous rules of quantum mechanics. In addition, it is common to express the interaction between the molecules or particles concerned in terms of simple model functions, (for example, the Lennard-Jones or Morse potentials between pairs of particles), instead of calculating and calibrating a full potential energy surface based upon ab initio methods.

These approximations are performed primarily for two reasons, both of which are related to the size of the system to be treated. The first is for computational simplicity: the largest chemically reactive systems treated to date by rigorous quantum mechanical methods consist of only three atoms, for example the reactions

$$H + H_2 \text{ --------> } H_2 + H$$

or

$$F + H_2 \text{ --------> } HF + H \ ,$$

each of which required hundreds of hours of Cray supercomputer time despite their apparent simplicity. Although the accuracy of the method is limited by the classical approximation, molecular dynamics calculations can reasonably simulate systems containing hundreds or even thousands of atoms. Rigorous quantum mechanical treatment of systems of this size is currently beyond the state of the art.

The second reason is paradoxically an advantage related to the size of the system. In most molecular dynamics simulations, what is sought is not detailed information concerning the internal state of each molecule involved, but bulk properties, such as temperature or diffusion coefficients. These properties are best calculated by the treatment of a large number of molecules simultaneously, where averaged behavior is of interest. This averaging is usually performed by making the ergodic assumption, i.e., that the average value of a quantity over time is the same as the average value of that quantity over a large number of particles. Despite the simplifying approximations described previously, molecular dynamics calculations remain formidable computational tasks, due to the large number of particles involved, and the amount of time for which the molecular motions must be followed. This paper presents the results of several optimization steps for a prototype molecular dynamics program on two IBM RS/6000 scalar work stations (a Model 370 and a Model 550) and several Cray vector supercomputers (a Cray X-MP/48,

a Cray Y-MP8/128, and a Cray C-90/16256). The model system is first described, including the periodic boundary conditions, the interaction potential, and the method of tracking each particle. The FORTRAN implementation of this model is described, and various optimization schemes are applied to the program on each machine. The efficiency of each optimization scheme is compared, and the application of these schemes to more general programs is considered.

2. THEORY AND MODEL

2.1 System. The system modeled by the present program is a two-dimensional (2-D) herringbone lattice of diatomic molecules which are to be perturbed by a shock wave passing from one end of the lattice to the other. The time history of the system is followed by integration of Hamilton's equations of motion using a modified variable step-size Adams-Moulton fourth-order predictor-corrector integrator (Miller and George 1972), with relative error tolerance set at 10^{-7}. The simulation is terminated by a user-defined cutoff on the number of integration steps allowed.

The system used for the purposes of this study consists of 192 atoms, all of which are allowed to move in the simulations. The atoms are laid out in a rectangular grid of 12 rows and 18 columns, as shown in Figure 1.

2.2 Periodic Boundary Conditions. The periodic boundary conditions used in this model will be described first, since they are intrinsic to both the tracking and identification of particles (see 2.3) and the interaction potential (see 2.4). The program constructs a 2-D rectangular lattice of dimension M × N, where M and N are user-supplied parameters read in by the program. The physical system ($\sim 10^{23}$ atoms) to be modeled is much larger than the number of particles that may be simulated ($\sim 10^{4}$). We are interested in the properties of a bulk crystal undergoing shock, rather than a tiny crystal with large surface area. For a system of 10^4 atoms there tends to be a substantial number of atoms in the simulation box which are in close proximity to the edges of the box. These atoms behave as "surface" atoms rather than as atoms in the bulk. Of course, this would dramatically affect the computed final results, and would not give a physically meaningful description of bulk properties.

When periodic boundary conditions are enforced, a particle exiting the simulation box at one edge is replaced by another particle entering the simulation box at the opposite edge, with identical mass and velocity vector of equal magnitude and direction. (To visualize this better, consider a sheet of paper rolled

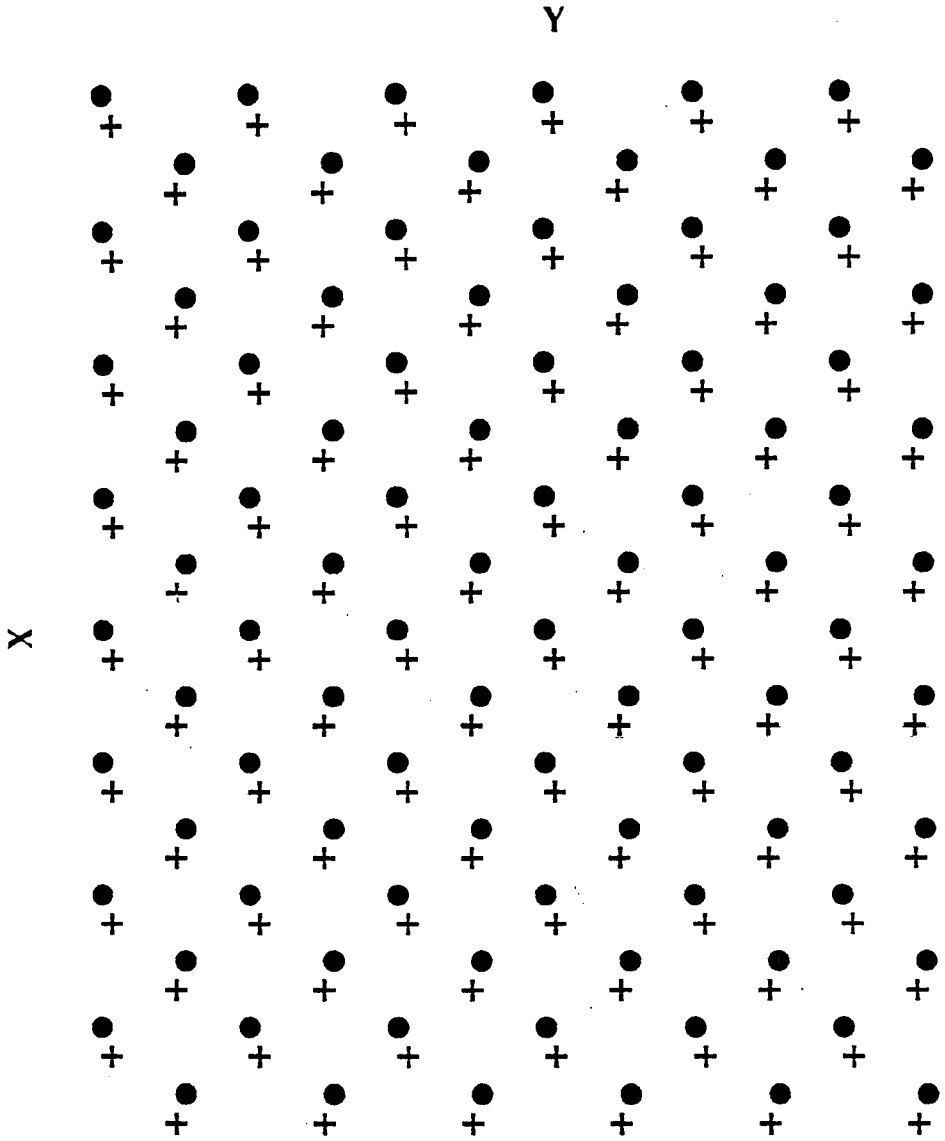


Figure 1. Test system for benchmark studies.

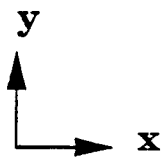
into a cylinder. A pencil line drawn off of the top of the page immediately crosses over to the bottom of the page with no discontinuity. If the periodic boundary conditions are applied in both the horizontal and the vertical directions, the cylinder becomes a torus.) The periodic boundary condition method effectively allows the description of an infinite system by a finite number of particles. The method is described in more detail in Allen and Tildesley (1990). For the current simulation, periodic boundary conditions are enforced in both directions.

2.3. Tracking and Identification of Particles. The technique used for the tracking and identification of particles during the simulation is the method of cells and linked lists, as described in Allen and Tildesley (1990). However, this method has been modified for our purposes, since the current system is only 2-D. The rectangular simulation box is divided into a rectangular lattice of $L_x \times L_y$ cells, as shown in Figure 2, where the origin of the system is the lower left-hand corner of cell 1. The value of L_q (q = x, y) is chosen so that the length of each cell in each dimension, d_q, is larger than the cutoff radius r of the interaction potential (see 2.4). Also, each cell must be a multiple of the unit cell of the crystal. For the system in this study, the cell sizes are twice the size of the unit cell in both the x and y directions, making $L_x = 4$ and $L_y = 3$.

As an example, consider cell 6 in Figure 2. The only particles that may be considered as neighboring particles of those in cell 6 are those in cells 1, 2, 3, 5, 7, 9, 10, and 11. All particles in all other cells are outside the range of the interaction potential. As an example of an edge cell, consider cell 1. The only particles that are considered neighbors of those in cell 1 are those in cells 12, 9, 10, 4, 2, 8, 5, and 6. Since each of these cells has a unique list of the particles it contains, searching through the list of particles in the neighboring cells is a much more rapid process than testing every particle in the entire box.

2.4 Interaction Potential. The interaction potential describing this system was developed by Brenner (1991). It is a pair-additive function, which incorporates a many-body effect to the bonding environment between two atoms. Its functional form is:

$$V = \sum_{i=1}^{N} \sum_{j>i}^{N} \left\{ f_c(r_{ij}) \left[V_R(r_{ij}) - \frac{1}{2}(B_{ij} + B_{ji}) V_A(r_{ij}) \right] + V_{NB}(r_{ij}) \right\} \qquad (1)$$



9	10	11	12
5	6	7	8
1	2	3	4

Figure 2. Division of simulation volume into cells.

where

$$V_R(r) = \frac{D_e}{S-1} \exp\left[-\alpha\sqrt{2S}\,(r - r_e) \right] \tag{2}$$

$$V_A(r) = \frac{S \cdot D_e}{S-1} \exp\left[-\alpha \sqrt{\frac{2}{S}}\,(r - r_e) \right] \tag{3}$$

$$B_{ij} = \left\{ 1 + G \sum_{k \neq i,j}^{N} f_c(r_{ik}) \exp\left[m\,(r_{ij} - r_{ik}) \right] \right\}^{-n} \tag{4}$$

$$f_c(r) = \begin{cases} 1 & r < 2 \\ \frac{1}{2}\left[\, 1 + \cos\left(\pi\,(r-2)\right)\right] & 2 \le r < 3 \\ 0 & 3 \le r \end{cases} \tag{5}$$

and

$$V_{NB}(r) = \begin{cases} 0 & r < 1.70 \\ \sum_{i=3}^{5} C_i\,(r - 1.70) & 1.70 \le r < 1.75 \\ P_0 + r\left(P_1 + r\left(P_2 + rP_3\right)\right) & 1.75 \le r < 2.91 \\ 4\varepsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right] & 2.91 \le r < 7.31 \\ \sum_{i=3}^{5} d_i\,(r - 7.31) & 7.31 \le r < 7.32 \\ 0 & 7.32 \le r \end{cases} \tag{6}$$

Parameters for equations 1–6 are given later in the report in Table 1. We wish to point out that there are several changes in this potential from that given by Brenner (1991). First of all, the equation reproduced in equation 4 here was written incorrectly in Brenner (1991). The correct form is in equation 4. Values of the P's were given in White (1992); however, using these parameters led to a discontinuity in the potential. We have therefore chosen the P's in Table 1 such that this polynomial has continuous first derivatives. Also, there were two discontinuities in V_{NB}, which we have corrected by adding quintic splines. The coefficients of the quintic splines were chosen such that the first and second derivatives of this function were continuous.

3. COMPUTER SYSTEMS

The computer systems on which the benchmarking studies have been performed are a Cray Research, Inc. Cray X-MP/48, with operating system UNICOS version 6.1.3, a Cray Research, Inc. Cray Y-MP8/128, with operating system UNICOS version 7.0.4.4, a Cray C-90/16256, with operating system 7.C.3 prerelease, and two IBM RS/6000 RISC workstations (a Model 550 and a Model 370, each with operating system AIX version 3.2). The Crays are conventional vector supercomputers. For the Crays, numerical

calculations (such as FORTRAN DO LOOPs) may be partially parallelized by the vector registers, and other FORTRAN constructs are handled sequentially. The IBM work stations are scalar processors with reduced instruction set chip (RISC) architectures that may handle certain FORTRAN tasks very efficiently due to specific low-level commands within the reduced instruction set.

The next section of this report relates the performance of this prototype molecular dynamics code on each of the machines, and compares the results on each machine when analogous compiler optimizations and FORTRAN constructs are included in the program on each machine. Although there are differences between the Cray X-MP architecture and that of the Y-MP and C-90 (e.g., the length of the vector registers on the C-90 is 256 compared to 64 on the X-MP), these machines are similar enough that the discussion of the X-MP vector architecture is applicable to the Y-MP and C-90 as well.

4. OPTIMIZATION

This section details the performance of the computer program on each of the machines listed in section 3, together with a description of accuracy tests on the output, which are discussed in more detail later. Section 4.1 describes the unoptimized version of the program and the checks on accuracy of the final results. Section 4.2 describes changes to important FORTRAN kernels which speed execution on some of the machines. Section 4.3 describes the influence of various compiler options on the scalar and vector performance of the program.

4.1. Unoptimized Program and Accuracy Tests. This section describes the performance of the raw, unoptimized FORTRAN on each of the four platforms of section 3. With only two exceptions, the default compiler options were used in each case. The first exception is common to all of the machines tested, and is the use of a compiler flag to detect and warn of all non-ANSI standard FORTRAN 77 within the program. This was done to ensure that the same program was run on all of the machines. The second exception was for the Cray computers. Most computers represent real floating-point numbers as a 32-bit word, which is referred to as "single precision" accuracy. This representation allows four figures after the decimal, which is not sufficiently accurate for most scientific computations. To overcome this difficulty, most scientific computer programs are written in "double precision" accuracy (64-bit words), which allows eight figures after the decimal point. However, the Cray computer architecture was designed from the start primarily for scientific applications, and so the Cray architecture allows for 64-bit words already within single precision. To ensure compatibility of the same FORTRAN program on all of the machines, the

compiler option "-dp" was used on the Cray. This option tells the Cray to treat all variables labeled as double precision as though they were Cray single precision, so that the same 64-bit word length is used on all machines in the benchmark tests.

There are two tests which are used to ensure the accuracy of the results on all five of the machines. These tests are used to make certain that the program changes and compiler options introduced later do not lead to errors in the program. The first test is that of conservation of energy. That is, the total energy of the system may be described at each step by

$$E = V + T \tag{7}$$

where V is the potential energy described in equation 1, and T is the kinetic energy of the system. The possibility of error in energy conservation arises within the numerical integration step, when the solution to Hamilton's equation of motion for the system is propagated numerically. The error of the Adams-Moulton integration scheme is on the order of h^6 where h is the time step used in the integration. If too large of a time step is used, then numerical errors will appear in the predicted values of the momenta of the particles, and these will intensify as the propagation is continued. Conversely, if too small of a time step is used, then round-off errors in the numerical quantities will accumulate and affect the accuracy of the results. For the simulation described here, a maximum tolerance of 0.1% is allowed for variation of the total energy of the system at each step. The second check on the final results is given in detail in Tsai (1979). Briefly, this check involves two independent methods of computing the average pressure of the system over the course of the simulation. The first of these is to compute the pressure from the internal stress of the system:

$$P = \frac{1}{2}(\sigma_{xx} + \sigma_{yy}) \tag{8}$$

where σ_{xx} and σ_{yy} are the internal normal components of the stress of the system. We note that we calculated σ_{xx} and σ_{yy} at imaginary dividing surfaces located on the edge of every unit cell of this crystal, and used the time-averaged value of these components in equation 8. The second of these is to calculate the pressure using the virial theorem (Tsai 1979). Because the internal pressure is being calculated for a system at thermal equilibrium, it is expected that the virial theorem is valid, and that the two methods

for calculating the pressure of this system should be comparable. Variance of approximately 1% in the average pressure calculated by the two methods may be expected (Tsai 1979).

Initially, all of the atoms are in the equilibrium position; the potential energy of the system is zero. The total energy of each atom is taken to be kT, where k is the Boltzmann constant and T is set to 298.15 K. The energy of each atom is equipartitioned into the X and Y momenta components, with the sign of each component assigned randomly. The equations of motion for the system are integrated for 2,000 steps (~0.08 ps) to "randomize" the energy of the system. From this point through the next 10,000 integration steps, the internal pressure, virial, and energy were calculated and stored for averaging at each tenth step of the simulation. At the end of the total 12,000 integration steps, the simulation was stopped. The total simulation time period was approximately 0.4 ps.

The next section of this report discusses simple changes that may be made to the FORTRAN, using widely supported and publicly available subprograms that help to improve the floating-point performance of the program.

4.2 Code Optimizations. As a first step towards optimization of the programs, it is necessary to understand the differences between the Cray computer and the work stations. The Cray machine achieves its high floating-point performance through two factors: its high clock speed and its architecture. The clock speed of the machine is the maximum rate at which instructions can be sent from the CPU. For the Cray XMP/48 used here the clock speed is 8.5 ns, or approximately 120 MHz. The clock speed of the Y-MP/8128 is 6.0 ns (~165 Mhz), and the clock speed of the C-90/16256 is 4.2 ns (~240 Mhz). By comparison, the clock speed of the IBM Model 550 is approximately 42 MHz, and the clock speed of the IBM Model 370 is approximately 62 MHz). However, due to the fact that several instructions must be issued for each floating-point operation that is performed, the clock speed is greater than the millions of floating-point operations per second (MFLOPs) rate. Furthermore, the exact number of instructions which must be issued for each floating-point operation differs from machine to machine and for different floating-point operations (e.g., computation of a SINE function is much more costly than a floating-point addition) so that an exact relation of clock speed to program execution rate is impractical. Therefore in the discussion that follows we concentrate upon the architecture of the Crays as compared to the IBM work stations, and the influence of these architectural differences between the machines upon CPU performance.

The Cray machines are vector pipeline computers. This means that if there exists a floating-point operation that must be performed identically on some subset of the data within memory, and that the data is contiguous or each element of the data is separated by a constant interval, then the Crays can read the data to be manipulated from memory into the vector registers, perform the pipelined floating-point operations on the data, and write the result back to memory. A typical example of such a situation in FORTRAN is the DO LOOP, for example:

```
      DIMENSION A(1000)
      DO 10 I=1,1000
         A(I) = A(I) * 0.023
10    CONTINUE
```

where A is the name of the array on which the identical operations are to be performed; A is stored contiguously, and each element of A is to be multiplied by a constant. On a Cray machine, the multiplication of elements of A would be performed in parallel (vectorized), where the number of elements of A multiplied at one time are dependent upon the length of the Cray's vector register (this number ranges from 64 for the Cray X-MP/48 to 256 on the Cray C-90/16256). Once the vector register is loaded, floating-point operations are produced at a rate of one every clock cycle. There exists some overhead in setting up the vectorization of a DO LOOP, so that the time saved by the vectorization may not always be sufficient to make up for the time lost in setting up the vectorization. Therefore, vectorization of small DO LOOPs may actually slow down the execution of a program.

It should also be noted that some FORTRAN constructs will prevent vectorization. For example, the following DO LOOP would not be vectorized:

```
      DIMENSION A(1000)
      DO 10 I=1,1000
       IF(I.LT.200)THEN
         A(I) = A(I) * 0.023
       ELSE
         A(I) = A(I) + 6.152
       END IF
10    CONTINUE
```

because different elements of the vector A would have different arithmetical operations performed on them. The following loop would also be vectorized:

```
      DIMENSION A(1000), B(1000)
      DO 20 I=1, 1000
        A(I) = A(I) + B(I) * 7.2
20    CONTINUE
```

In the latter case, once the vector registers have been loaded,the results of the multiplication are "chained" to the vector register to perform the addition at the same time, so that the net result is two floating-point operations every clock cycle.

On scalar work stations, there are no vector registers, but the rate of execution of a FORTRAN kernel that would vectorize on the Cray is limited by the rate at which data may be transferred from memory to the floating-point processor. On the IBM work stations, this is controlled by a local cache which transfers data from main memory to the processor. If an instruction requires data not in the cache, then a page fault will occur, which delays execution until the proper data can be accessed from memory. Explicitly writing a FORTRAN code which is guaranteed to avoid such delays is very time-consuming and produces complicated DO LOOPs. Fortunately, this difficulty is easily circumvented as described next.

A set of portable routines, the BLAS (Basic Linear Algebra Subprograms), which are available on a wide variety of machines, including the Cray and the IBM work stations, were designed to help circumvent these problems. (The two DO LOOPs in the previous examples are reproducible using the BLAS subprograms DSCAL and DAXPY, respectively.) The BLAS routines provide a standard calling sequence for a number of simple FORTRAN kernels which occur repeatedly in scientific applications, and have been implemented as machine-optimized library routines by a number of hardware vendors. Furthermore, the calling sequences and descriptions of the BLAS are publicly available free of charge via electronic mail (netlib@ornl.gov). Since the BLAS routines are machine optimized, and are widely portable, we decided to modify the existing molecular dynamics prototype program to include them on the scalar work stations. We did not include these routines in the Cray version of the program for two reasons. First, all FORTRAN kernels within the present program which are eligible for replacement by the BLAS are already vectorized by the Cray's compiler and so perform efficiently. Furthermore, the on-line documentation supplied by Cray Research, Inc. on the Cray machines indicates that for many cases (including those situations considered in the present program), the performance of the program created by the Cray vectorizing compiler exceeds that of the program the BLAS (even the Cray Research, Inc. scientific subroutine library version of the BLAS).

We now give a description of those routines within the molecular dynamics code in which the BLAS routines were incorporated, and illustrate the application of the BLAS routines to FORTRAN constructs for which several alternative implementations were possible. This is done progressively in order that the reader may learn to apply the BLAS to his/her own application in the most efficient way possible. The four subroutines in the program in which the BLAS routines were included are as follows:

PROGRAM DRIVER:	the main driver for the program
SUBROUTINE POTEN:	calculates the potential and derivatives
SUBROUTINE VAM:	Adams-Moulton numerical integrator
SUBROUTINE VPRIME:	calculates time derivatives of coordinates and momenta for SUBROUTINE VAM

Not all of these programs are called the same number of times, and so not all of these routines require an equal amount of CPU time for execution. For the purposes of this discussion, we concentrate on the two routines VAM and VPRIME, since these two routines contain all of the different types of FORTRAN DO LOOPS which were replaced by the BLAS in the other two routines.

SUBROUTINE VPRIME consists of the DO LOOP

```
      DO 10 I = 1, NMOVE
        XA(I) = X(I)
        YA(I) = Y(I)
10    CONTINUE
```

followed by a call to POTEN. This DO LOOP is trivially replaced by the simplest of the BLAS routines, DCOPY (the complete calling sequence, including a description of each argument, is given for this and for the other BLAS shown next):

```
CALL DCOPY (NMOVE, X(1), 1, XA(1), 1)
CALL DCOPY (NMOVE, Y(1), 1, YA(1), 1)
```

The call to POTEN is, of course, not changed.

SUBROUTINE VAM contains the following DO LOOPs (in addition to other code):

```
      DO 100 J = 5, 14
        DO 100 I = 1, NEQ
100       T(J,I) = ZERO

      DO 300 I = 1, NEQ
        T(3,I) = Y(I)
        T(9,I) = DY(I)
        IF (NORDER .GT. 0) THEN
         DEN = ZERO
         DO 250 J = 1, NORDER
           DEN = DEN + H(J)
           T(J+9,I) = (T(J+8,I)-T(J+3,I))/DEN
           T(J+3,I) = T(J+8,I)
250     CONTINUE
      END IF
      T(NORDER+4,I) = T(NORDER+9,I)
300   CONTINUE

      DO 500 I = 1, NEQ
        TEMP = 0.00
        IF (NORDER .GT. 0) THEN
          DO 450 J = 1, NORDER
            TEMP = TEMP + T(4+J,I)*AA(J)
450     CONTINUE
      END IF
      T(1,I) = TEMP + T(4,I)*H(1)
      Y(I) = T(1,I) + T(14,I) + T(3,I)
500   CONTINUE

      DO 600 I = 1, NEQ
        T(9,I) = DY(I)
        DEN = ZERO
        IF (NORDER .GT. 0) THEN
          DO 550 J = 1, NORDER
             DEN = DEN + H(J)
             T(J+9,I) = (T(J+8,I) - T(J+3,I))/DEN
550       CONTINUE
        END IF
600   CONTINUE
      DO 700 I = 1, NEQ
        TEMP = 0.00
        IF (NORDER .GT. 0) THEN
          DO 650 J = 1, NORDER
            TEMP = TEMP + T(9+J,I)*BB(J)
650       CONTINUE
      END IF
```

```
      T(2,I) = T(9,I)*H(1) - TEMP
      Y(I) = T(2,I) + T(14,I) + T(3,I)
      T(14,I) = (T(2,I) + T(14,I)) - (Y(I) - T(3,I))
700 CONTINUE
```

While these DO LOOPs may seem complicated, it will be shown that they all may be rewritten in terms of four BLAS routines: DAXPY, DSCAL, DDOT, and DCOPY. For reference, we give the function and a generic argument list for each of these routines.

DAXPY: multiply the elements of one vector X by a constant CON, and add the results to another vector Y.

CALL DAXPY(NN, CON, X(1st), DX, Y(1st), DY)

where

NN is the number of elements of X and the number of elements of Y to be accessed.
CON is the constant by which to multiply the elements of X.
X(1st) is the first element of vector X to be accessed.
DX is the interval between successively accessed elements of X.
Y(1st) is the first element of vector Y to be accessed.
DY is the interval between successively accessed elements of Y.

DSCAL: multiply the elements of a vector by a constant.

CALL DSCAL(NN, CON, X(1st), DX)

where

NN is the number of elements of X to be accessed.
CON is the constant by which to multiply the elements of X.
X(1st) is the first element of vector X to be accessed.
DX is the interval between successively accessed elements of X.
DDOT: take the dot product of two vectors.

CONSTANT = DDOT(NN, X(1st), DX, Y(1st), DY)

where

CONSTANT will contain the final value computed by FUNCTION DDOT.
NN is the number of elements of X and the number of elements of Y to be accessed.
X(1st) is the first element of vector X to be accessed.

DX is the interval between successively accessed elements of X.
Y(1st) is the first element of vector Y to be accessed.
DY is the interval between successively accessed elements of Y.

DCOPY: copy the contents of one vector to another vector.

CALL DCOPY(NN, X(1st), DX, Y(1st), DY)

where

NN is the number of elements of X and the number of elements of Y to be accessed.
X(1st) is the first element of vector X to be accessed.
DX is the interval between successively accessed elements of X.
Y(1st) is the first element of vector Y to be accessed.
DY is the interval between successively accessed elements of Y.

Inspection of the DO LOOPS shown previously shows that the sequence DO 250/DO 300 has the same structure as the sequence DO 550/DO 600, so that when one of these computations has been rewritten, then the work for rewriting both of them has been done. Furthermore, the structure of the sequence DO 450/DO 500 has the same structure as the sequence DO 650/DO 700, so that this pair of computations may also be rewritten in tandem. We now consider each of the DO LOOPs in turn.

The first, DO LOOP 100, is a simple double-nested structure:

```
      DO 100 J = 5, 14
        DO 100 I = 1, NEQ
100        T(J,I) = ZERO
```

Since this is a doubly nested DO LOOP, there is no BLAS routine which will perform the work on both indices simultaneously. However, it is possible to replace one of the DO LOOPs with the DSCAL command, while retaining the other. In general, it is more efficient to structure the BLAS so that each call references the greatest number of elements of the arrays concerned. In this case, the innermost DO LOOP is rewritten:

```
      DO 100 J=5,14
      CALL DSCAL(NEQ,ZERO,T(J,1),NDIM)
100   CONTINUE
```

where DSCAL is the BLAS routine which multiplies the selected elements of an array by a constant. Since NDIM is the row dimension of array T, this operation overwrites all the elements of the Jth row of array T with ZERO at each iteration J.

The second DO LOOP is far more complex:

```
      DO 300 I = 1, NEQ
        T(3,I) = Y(I)
        T(9,I) = DY(I)
        IF (NORDER .GT. 0) THEN
         DEN = ZERO
         DO 250 J = 1, NORDER
           DEN = DEN + H(J)
           T(J+9,I) = (T(J+8,I)-T(J+3,I))/DEN
           T(J+3,I) = T(J+8,I)
250      CONTINUE
        END IF
        T(NORDER+4,I) = T(NORDER+9,I)
300   CONTINUE
```

This DO LOOP contains several array copy commands, a partial reduction, and the scaling of an array by a constant value. The structure is further complicated by the presence of an IF...THEN block.

The first thing to do is to see if any of the LOOP can be simplified. Inspection shows that the first, second, and final commands are done irrespective of the IF...THEN block, and furthermore do not access any array values which may be modified within the IF...THEN block. These may therefore be removed immediately from the nested DO LOOP structure and replaced by BLAS DCOPY calls:

```
      CALL DCOPY(NEQ,Y(1),1,T(3,1),NDIM)
      CALL DCOPY(NEQ,DY(1),1,T(9,1),NDIM)
      DO 300 I = 1, NEQ
        IF (NORDER .GT. 0) THEN
          DEN = ZERO
          DO 250 J = 1, NORDER
            DEN = DEN + H(J)
            T(J+9,I) = (T(J+8,I)-T(J+3,I))/DEN
            T(J+3,I) = T(J+8,I)
250       CONTINUE
        END IF
300   CONTINUE
      CALL DCOPY(NEQ,T(NORDER+9,1),NDIM,T(NORDER+4,1),NDIM)
```

(Note also that even though the final DCOPY command involves the array T(NORDER+9,1), which is affected by the IF...THEN block, that array is only used after the IF...THEN block has been completed, and so it is permissible to remove the array from the loop by using the DCOPY command.)

Once this step has been performed, it is easily seen that the IF...THEN block encompasses all of the numerical computation within both DO LOOPs, so that all of the numerical work within the DO LOOPs is determined by the value of NORDER, which is defined *outside* of the DO LOOPs. It is therefore more efficient to move the IF...THEN block outside of the DO LOOPs so that the LOGICAL IF need only be evaluated once for this set of DO LOOPs, instead of NEQ times:

```
      CALL DCOPY(NEQ,Y(1),1,T(3,1),NDIM)
      CALL DCOPY(NEQ,DY(1),1,T(9,1),NDIM)
      IF (NORDER .GT. 0) THEN
        DO 300 I = 1, NEQ
          DEN = ZERO
          DO 250 J = 1, NORDER
            DEN = DEN + H(J)
            T(J+9,I) = (T(J+8,I)-T(J+3,I))/DEN
            T(J+3,I) = T(J+8,I)
250       CONTINUE
300     CONTINUE
      END IF
      CALL DCOPY(NEQ,T(NORDER+9,1),NDIM,T(NORDER+4,1),NDIM)
```

When the DO LOOP structure has been reconfigured in this form, then the parallelism within the inner loop becomes apparent. It is tempting to immediately rewrite the loop over 250, by replacing the recursion over J in terms of a BLAS DCOPY and DAXPY for the first line and a DCOPY in the second line. But this is impossible, because the value DEN in the first line changes with J, and so the first line cannot be written as a DAXPY over J. (Recall that the DAXPY command scales the elements of one matrix by a constant and adds the results to a second matrix.) However, it is still possible to write the work in terms of the BLAS routines, if one first reverses the order in which the DO LOOPs are executed:

```
      CALL DCOPY(NEQ,Y(1),1,T(3,1),NDIM)
      CALL DCOPY(NEQ,DY(1),1,T(9,1),NDIM)
      IF (NORDER .GT. 0) THEN
        DEN = ZERO
        DO 250 J = 1, NORDER
          DEN = DEN + H(J)
          DO 300 I = 1, NEQ
```

```
        T(J+9,I) = (T(J+8,I)-T(J+3,I))/DEN
        T(J+3,I) = T(J+8,I)
300      CONTINUE
250    CONTINUE
    END IF
    CALL DCOPY(NEQ,T(NORDER+9,1),NDIM,T(NORDER+4,1),NDIM)
```

With the logic appearing in this form, the expression of the remaining DO LOOPs in terms of the BLAS is trivial, since DEN is a constant across all values of I for a constant value of J:

```
    CALL DCOPY(NEQ,Y(1),1,T(3,1),NDIM)
    CALL DCOPY(NEQ,DY(1),1,T(9,1),NDIM)
    IF (NORDER .GT. 0) THEN
      DEN = ZERO
      DO 250 J = 1, NORDER
        DEN = DEN + H(J)
        TMP=1.0D0/DEN
        CALL DSCAL(NEQ,ZERO,T(J+9,1),NDIM)
        CALL DAXPY(NEQ,-TMP,T(J+3,1),NDIM,T(J+9,1),NDIM)
        CALL DAXPY(NEQ,TMP,T(J+8,1),NDIM,T(J+9,1),NDIM)
        CALL DCOPY(NEQ,T(J+8,1),NDIM,T(J+3,1),NDIM)
250     CONTINUE
    END IF
    CALL DCOPY(NEQ,T(NORDER+9,1),NDIM,T(NORDER+4,1),NDIM)
```

where the only change to the previous form is the introduction of the temporary TMP to hold the reciprocal of DEN at each step. Remember that in the absence of any data dependency the order in which the operations are performed should not affect the final results, except for possible roundoff error. This is also why the elements of T(J+8,I) and/or T(J+3,I) were not scaled and then copied into T(J+9,I). The final value of T(J+9,I) would have been identical at each step but the values of the arrays T(J+8,I) and T(J+3,I), which are used elsewhere within the program, would have been changed.

Given the translation of the DO 250/DO 300 previous sequence, it can easily be shown that the proper translation of the DO 550/DO 600 sequence is as follows:

```
    CALL DCOPY(NEQ,DY(1),1,T(9,1),NDIM)
    IF(NORDER .GT. 0)THEN
      DEN=ZERO
      DO 550 J=1, NORDER
        DEN=DEN+H(J)
```

```
        TMP=ONE/DEN
        CALL DSCAL(NEQ,ZERO,T(J+9,1),NDIM)
        CALL DAXPY(NEQ,TMP,T(J+8,1),NDIM,T(J+9,1),NDIM)
        CALL DAXPY(NEQ,-TMP,T(J+3,1),NDIM,T(J+9,1),NDIM)
550     CONTINUE
      END IF
```

The original FORTRAN version of the DO 450/DO 500 sequence is as follows.

```
      DO 500 I = 1, NEQ
        TEMP = 0.00
        IF (NORDER .GT. 0) THEN
          DO 450 J = 1, NORDER
            TEMP = TEMP + T(4+J,I)*AA(J)
450       CONTINUE
      END IF
      T(1,I) = TEMP + T(4,I)*H(1)
      Y(I) = T(1,I) + T(14,I) + T(3,I)
500   CONTINUE
```

This loop also contains a dot product, a multiplication of an array by a constant, and several matrix additions. There is also an IF...THEN block interspersed with the DO LOOPs. As before, it is tempting to move the array manipulations which are outside of the IF...THEN block to BLAS routines outside of both DO LOOPs. It appears at first that the reference to T(14,I) in the line just before statement label 500 would prevent this, because if NORDER is large enough, this array would be referenced within the DO LOOP; however, the program has limited the value of NORDER to 4, so that this will never happen. The real barrier to the use of the BLAS is that T(1,I) depends upon TEMP, which is conditionally defined by the IF...THEN block. The solution is to break up the computation of T(1,I) into two pieces, one of which is dependent upon the IF...THEN block and one of which is independent of it:

```
      DO 500 I = 1, NEQ
        T(1,I) = T(4,I)*H(1)
        TEMP = 0.00
        IF (NORDER .GT. 0) THEN
          DO 450 J = 1, NORDER
            TEMP = TEMP + T(4+J,I)*AA(J)
450       CONTINUE
      END IF
      T(1,I) = TEMP + T(1,I)
      Y(I) = T(1,I) + T(14,I) + T(3,I)
500   CONTINUE
```

The computation of Y(I) may also be broken into two pieces, one of which is independent of the IF...THEN block, and one which depends implicitly upon the IF...THEN block through the dependence upon T(1,I):

```
      DO 500 I = 1, NEQ
        T(1,I) = T(4,I)*H(1)
        Y(I) = T(1,I) + T(14,I) + T(3,I)
        TEMP = 0.00
        IF (NORDER .GT. 0) THEN
          DO 450 J = 1, NORDER
            TEMP = TEMP + T(4+J,I)*AA(J)
450       CONTINUE
      END IF
      T(1,I) = TEMP + T(1,I)
      Y(I) = T(1,I) + Y(I)
500   CONTINUE
```

Once the separation has been performed, separation of the code into BLAS calls is performed:

```
      CALL DCOPY(NEQ,T(4,1),NDIM,T(1,1),NDIM)
      CALL DSCAL(NEQ,H(1),T(1,1),NDIM)
      CALL DCOPY(NEQ,T(14,1),NDIM,Y(1),1)
      CALL DAXPY(NEQ,ONE,T(3,1),NDIM,Y(1),1)
      DO 500 I = 1, NEQ
        TEMP = 0.00
        IF (NORDER .GT. 0) THEN
          DO 450 J = 1, NORDER
            TEMP = TEMP + T(4+J,I)*AA(J)
450       CONTINUE
      END IF
      T(1,I) = TEMP + T(1,I)
500   CONTINUE
      CALL DAXPY(NEQ,ONE,T(1,1),NDIM,Y(1),1)
```

The structure of the nested DO LOOPs and the IF...THEN statement may be collapsed once it is recognized that the only array whose value is changed is T(1,I); the nested DO LOOPs may be replaced by a single DO LOOP containing a DDOT, and the single DO LOOP enclosed within the IF...THEN statement, completing the optimization:

```
      CALL DCOPY(NEQ,T(4,1),NDIM,T(1,1),NDIM)
      CALL DSCAL(NEQ,H(1),T(1,1),NDIM)
```

```
      CALL DCOPY(NEQ,T(14,1),NDIM,Y(1),1)
      CALL DAXPY(NEQ,ONE,T(3,1),NDIM,Y(1),1)
      IF (NORDER .GT. 0) THEN
        DO 500 I = 1, NEQ
          T(1,I) = T(1,I) + DDOT(NORDER,T(5,I),1,AA(1),1)
500     CONTINUE
      END IF
      CALL DAXPY(NEQ,ONE,T(1,1),NDIM,Y(1),1)
```

There are two caveats to the preceding discussion. Note first that time is saved by calculating the final value of T(1,I) first, including the IF...THEN dependence, before adding the value of T(1,I) to array Y, rather than adding each piece to Y as it is calculated. Second, it is important to notice that the vector T(4,I) was copied to T(1,I) in the first line, and then T(1,I) was scaled in the second line. If the scaling had been performed on T(4,I) before copying, then the value of T(1,I) would have remained correct, but the value of T(4,I) would have been different than the value produced by the original DO LOOP structure, changing the values returned by the subroutine. The DO LOOP sequence in DO 650/DO 700 may be replaced by BLAS calls similar to those just discussed:

```
      CALL DCOPY(NEQ,T(9,1),NDIM,T(2,1),NDIM)
      CALL DSCAL(NEQ,H(1),T(2,1),NDIM)
      IF(NORDER.GT.0)THEN
        DO 650 I=1,NEQ
          T(2,I)=T(2,I)-DDOT(NORDER,T(10,I),1,BB(1),1)
650     CONTINUE
      END IF
      CALL DCOPY(NEQ,T(2,1),NDIM,Y(1),1)
      CALL DAXPY(NEQ,ONE,T(3,1),NDIM,Y(1),1)
      CALL DAXPY(NEQ,ONE,T(14,1),NDIM,Y(1),1)
      CALL DAXPY(NEQ,ONE,T(2,1),NDIM,T(14,1),NDIM)
      CALL DAXPY(NEQ,-ONE,Y(1),1,T(14,1),NDIM)
      CALL DAXPY(NEQ,ONE,T(3,1),NDIM,T(14,1),NDIM)
```

4.3. Compiler-Level Optimizations. This section is concerned with the compiler optimizations used on the different machines. It is important first to realize that the Crays have different compiler options than the IBM machines: not only are there differences in the flags used to signal the use of a given compiler option, but some options are used by default on one machine which must be called explicitly on the other machines, or which do not even exist on the other machines. Since an exhaustive study of all of the possible compiler options is beyond the scope of this report, the following table lists the compiler options which were investigated, the calling sequence on each machine, and the default value of that

compiler option on each machine. If a given compiler option is nonexistent on a particular machine, then the legend under that entry will read "N/A" (not applicable). A brief description of the names and the functions of the various compiler options follows the table.

Table 1. Compiler Options Used on Each Machine

Machine	Optimization	Compiler Options Calling Sequence	Default Value
Cray (X-MP/48) (Y-MP/8128) (C-90/16256)	scalar optimization	-o scalar	disabled
	vector optimization	-o vector	enabled
	subroutine inlining	-o inline or -I *subroutine_name*	disabled
	preprocessor	N/A	N/A
IBM RS/6000 (Model 550 and Model 370)	scalar optimization-	03	disabled
	vector optimization	N/A	N/A
	subroutine inlining	[a]	disabled
	preprocessor	-Pk (+options) or -Pv (+options)	disabled

[a] This option may be invoked directly, using the command "-inline_from *(subroutine_name)*," or called as an option from either of the preprocessors.

The scalar optimization compiler flag instructs the compiler to perform various changes to the code, in the interest of increased computational efficiency, when compiling the original FORTRAN. For example, on many machines, floating-point divisions are considerably more costly than floating-point additions and/or multiplications. (On the IBM RS/6000, for example, a single floating-point division can take up to 19 times as long as a single floating-point addition or multiplication.) One way to help reduce the CPU required by the program, therefore, would be to replace a floating-point division (say division by a constant within a DO LOOP) by a reciprocation of the same constant before the DO LOOP, and to replace the division within the DO LOOP with multiplication by the reciprocal. The risk associated with such a replacement is that, due to the binary representation of floating-point arithmetic on the computer, the two results might not be bitwise identical. This may lead to the accumulation of roundoff errors as execution of the program continues. In most cases, however, the differences in the final computed results are negligible, while the speedup is significant. Therefore, even though the scalar optimization flag is not performed by default on many machines, use of the scalar optimizer is strongly encouraged.

The vector optimization flag only exists on the Cray machines, since only they have vector processors. This optimization helps in the vectorization of certain FORTRAN constructs such as a reduction (i.e., the summation of all elements of a vector into a single scalar value). The deficiency of the vector optimization on the Crays is that its use is mutually exclusive from the scalar optimization.

The subroutine inlining flag allows the compiler to replace a call to a specific subroutine with an inline (hence the name) version of the subroutine itself. For subroutines which are called repeatedly, this will save CPU time by eliminating the overhead associated with transferring control of the program to the subroutine.

The preprocessor flags are similar to the scalar optimization flags, except that they perform much more drastic changes to the FORTRAN source code. Since the changes made to the source code are dependent upon both the specific machine used and upon the program itself, the discussion and/or comparison of the preprocessors available upon the machines used here, and of their effects upon the program considered here, will not necessarily be applicable to programs in general or to work stations in general. (Specific details of the options and the performance of optimizing preprocessors on the various machines are given in proprietary publications supplied by the hardware vendors and will not be included here.) Some illustrative examples of the changes which may be made to the FORTRAN by optimizing preprocessors include the following:

- Generation of calls to the BLAS routines.
- "Unrolling" of DO LOOPs to improve memory access.
- Changing of IF statement to BLOCK IF constructs.
- Loop nest reordering.

For this work, we have compared the performance of two versions of this program on each of the IBM machines to the performance of the "original" version on each of the Cray machines. The two versions are identical, except for the replacement of certain FORTRAN DO LOOPs with BLAS as detailed in section 4.2. The Cray version of the program also differs slightly from the version used on the work stations in two respects. First, as noted earlier, Cray computers have 64-bit words, and so the compiler options used on the Crays include the -dp option in order that the FORTRAN source code used on the other machines will not require extensive modification before porting it to the Crays. Second, the random

number generator used on the other machines is not present on the Crays, so a proprietary Cray Research, Inc. random number generator is used instead. We compare the performance of each version of the program subject to the following initial conditions:

• Compilation using only default optimization (note that this includes vector optimization on the Crays).

• Compilation using scalar optimization.

• Compilation using scalar optimization in conjunction with the preprocessor (on the work stations, use of the preprocessors includes inlining; on the Crays, the inlining must be invoked separately).

Since the correspondence between the compiler options on the different machines is not exact, the conditions under which the runs are compared will not be "identical" from machine to machine, and this should be borne in mind during the discussion of the timings. The timings of the different runs and a discussion of the performance are given in the next section.

5. RESULTS AND DISCUSSION

The runs performed on each version of the program on each machine are as follows.

Machine Name	Version of Program	Compiler Options[a]	CPU Seconds
Cray X-MP/48	original	-Wf,"-dp -ensciv"	6973.8
	original	-Wf,"-dp -ensciv -o scalar -A fast"	6643.1
	original	-Wf,"-dp -ensciv -o inline -o scalar -A fast"	6674.8
Cray Y-MP/8128	original	-Wf,"-dp -ensciv"	5781.0
	original	-Wf,"-dp -ensciv -o scalar -A fast"	5774.1

[a] These options are discussed in section 4.3.

Machine Name	Version of Program	Compiler Options[a]	CPU Seconds
Cray C-90/16256	original	-Wf,"-dp -ensciv"	3977.0
	original	-Wf,"-dp -ensciv	3975.1
		-o scalar -A fast"	
IBM RS/6000	original	-qfips -qlist -Q	15320.4
Model 550	original	-O3 -qfips -qlist -Q	10215.5
	original + BLAS 1	-O3 -qfips -qlist -Q	10126.7
	original	-O3 -qfips -qlist -Q	10950.6
		-Pv,*options*	
	original	-O3 -qfips -qlist -Q	9878.8
		-Pk,*options*	
IBM RS/6000	original	-qfips -qlist -Q	10356.4
Model 370	original	-O3 -qfips -qlist -Q	6714.3
	original + BLAS	-O3 -qfips -qlist -Q	6929.5
	original	-O3 -qfips -qlist -Q	6743.8
		-Pv,*options*	
	original	-O3 -qfips -qlist -Q	6335.2
		-Pk, *options*	

[a] These options are discussed in section 4.3.

The timings differ both by machine and by the compiler options selected. We first discuss the performance on the Cray X-MP/48, then the performance on the IBM work stations. (The runs on the Y-MP/8128 and C-90/16256 are included for comparison.) Finally we relate the execution of the program to the architecture of the machines used and make recommendations for optimization of more general FORTRAN codes.

For the Cray X-MP/48, it is noteworthy that the best performance does *not* result from the use of the default compiler (which emphasizes optimization of vector constructs), but rather from the use of scalar optimization; this will be discussed in more detail later. In addition, the use of the inlining option on the Cray X-MP/48 does not significantly increase the execution rate of the program. (To ensure that the subroutines chosen for inlining were actually those which would be likely to improve the program's performance, the program was first run on the Cray X-MP/48 with the *-flowtrace* option, in order to find

which subroutines were the best candidates for inlining. Neither increasing the depth of subroutines which were inlined, nor explicitly naming the most favorable candidates, significantly increased the execution rate of the program.)

The execution rate of this program was actually enhanced by the use of the scalar optimization on the Cray X-MP/48 as opposed to the default vector optimization. This is contrary to the "usual" behavior of programs on a Cray, where vectorization of FORTRAN structures is recommended to achieve optimum performance. Programs that achieve optimum performance on a Cray usually contain either well-defined numerical subroutine calls (matrix multiply, solution of linear equations, etc.), which are both well vectorized (for example the LINPACK and EISPACK families of linear algebra subroutines present in the Cray scientific library routines) and numerically intensive. Extensive use of these routines within a program virtually guarantees that the majority of the computational work within the program will execute optimally on a Cray. Also, even if the application being run does not explicitly contain calls to vectorized scientific subroutine libraries, many scientific application programs are constructed such that the majority of the numerical work in the program is performed in DO LOOPs of the type described in section 4.2.

The ratio of CPU times for the two IBM RISC machines are approximately proportional to their respective clock speeds, as expected. The performance of different versions of the program increases in the order Raw code < Scalar-optimized raw code < Scalar-optimized raw code +BLAS < Scalar-optimized raw code + Preprocessor.

It is obvious that a significant improvement results from the inclusion of *any* optimization on the IBM machines. It has been our experience that significant increases in throughput on scalar work stations are readily attainable by minor coding changes and/or use of compiler flags. For the program considered here, a 50% improvement in performance was obtained by using simple compiler optimizations. This improvement made it possible for a RISC work station (the IBM 370) to outperform the Cray X-MP supercomputer on a prototype production program. (In previous work (Unekis et al., 1994), Unekis has improved the CPU performance of another computational chemistry code by a factor of *8* on an IBM RS/6000 RISC workstation using the methods discussed here).

Even though the use of the compiler flags gave markedly superior results to the use of the BLAS for this program on these work stations, in general it is better if the user makes comparisons on a case-by-case basis rather than relying on compiler optimizations alone. There are two main reasons for this. The first,

of course, is portability. Even though the IBM preprocessors and compiler optimizations worked very efficiently for this program, the IBM compilers are only available on IBM machines (and the quality of the compilers and preprocessors may vary from vendor to vendor), whereas the BLAS are portable across a wide variety of platforms and have been optimized separately by each vendor for their particular hardware. The second is that the use of the BLAS encourages the programmer to think both about the efficiency of the program and about other potential users of the program. Overreliance on the compiler may lead to lazy programming habits, and in practice to inefficient usage of the machine if one set of compiler options is inappropriate for a given program. On the other hand, overreliance on hand-tuning of a program may lead to nonportable FORTRAN, or to a program which is incomprehensible to other users. The use of the BLAS is a middle ground between these two extremes.

One of the important lessons to be learned from these benchmarking studies is the interplay of the FORTRAN and the machine architecture in code optimization. For the molecular dynamics program considered here, the majority of the executable statements in the program are concerned with the calculation of the contribution to the interaction potential by each distinguishable pair of particles in the simulation, and the resulting derivatives (forces). Even though the *formal* evaluation of the potential (see equations 1–6) appears to contain many vectorizable statements, both the segmented representation of the interaction potential and the nonuniform stride through memory inhibit vectorization of this portion of the program. This means that the majority of the executable statements within the program are processed in scalar mode, for which the IBM work stations are approximately as fast as the Cray vector machines. The difference in the overall performance of the program on the workstations as opposed to the Crays is dependent upon the remainder of the program (primarily within the numerical integrator) which consists of DO LOOPs as described in section 4.2. These statements are ordinarily performed optimally on the Cray architecture by vectorization, but are bottlenecks on the scalar processors due to the necessity of accessing the proper array elements from the memory. The high proportion of scalar statements within the program, therefore, makes the use of a vector processor relatively inefficient; since the program executes primarily in scalar mode, the use of alternative computer architectures, such as high-end scalar work stations, may become competitive.

Further work in this area will include the modification of the program to run on a massively parallel computer, and a discussion of the changes made to the algorithm and/or the program required by the new computer platform.

6. SUMMARY AND CONCLUSIONS

The findings of this set of benchmarking studies are summarized as follows.

- The program consists primarily of nonvectorizable (scalar) code, with a significant minority of vectorizable statements. These vectorizable statements execute well on the Cray machines but become bottlenecks for the IBM work stations.

- The performance of the IBM RS/6000 work stations can be significantly improved by making simple changes to the program or by using compiler optimization directives.

- When these changes are made, the IBM RS/6000 workstations become competitive with the Cray supercomputers and can even outperform the Cray XMP/48.

7. REFERENCES

Allen, M. P., and D. J. Tildesley. Computer Simulation of Liquids. Oxford: Oxford University Press, 1990 (and references therein).

Brenner, D. W. Shock Compression of Condensed Matter. Edited by S. C. Schmidt, R. D. Dick, J. W. Forbes, and D. G. Tasker. Amsterdam, Netherlands: Elsevier Publishers B. V., p. 115, 1991.

Miller, W. H., and T. F. George. "Analytic Continuation of Classical Mechanics for Classically Forbidden Collision Processes." Journal of Chemical Physics, vol. 56, p. 5668, 1972.

Tsai, D. H. "The Virial Theorem and Stress Calculations in Molecular Dynamics." Journal of Chemical Physics, vol. 70, p. 1375, 1979.

Unekis, M. J., D. W. Schwenke, N. M. Harvey, and D. G. Truhlar. MOTECC93: Modern Techniques in Computational Chemistry. Edited by E. Clementi. Leiden, Germany: ESCOM, 1993.

White, C. T., D. H. Robertson, and D. W. Brenner. "Dissociative Phase Transitions From Hypervelocity Impacts." Physica A, vol. 188, p. 357, 1992.

INTENTIONALLY LEFT BLANK.

NO. OF COPIES	ORGANIZATION
2	ADMINISTRATOR DTIC ATTN DTIC DDA CAMERON STATION ALEXANDRIA VA 22304-6145
1	CDR USAMC ATTN AMCAM 5001 EISENHOWER AVE ALEXANDRIA VA 22333-0001
1	DIR USARL ATTN AMSRL OP SD TA 2800 POWDER MILL RD ADELPHI MD 20783-1145
3	DIR USARL ATTN AMSRL OP SD TL 2800 POWDER MILL RD ADELPHI MD 20783-1145
1	DIR USARL ATTN AMSRL OP SD TP 2800 POWDER MILL RD ADELPHI MD 20783-1145
2	CDR US ARMY ARDEC ATTN SMCAR TDC PCTNY ARSNL NJ 07806-5000
1	DIR BENET LABS ATTN SMCAR CCB TL WATERVLIET NY 12189-4050
1	DIR USA ADVANCED SYSTEMS R&A OFC ATTN AMSAT R NR MS 219 1 AMES RESEARCH CENTER MOFFETT FLD CA 94035-1000
1	CDR US ARMY MICOM ATTN AMSMI RD CS R DOC RDSTN ARSNL AL 35898-5010
1	CDR US ARMY TACOM ATTN AMSTA JSK ARMOR ENG BR WARREN MI 48397-5000

NO. OF COPIES	ORGANIZATION
1	DIR USA TRADOC ANALYSIS CMD ATTN ATRC WSR WSMR NM 88002-5502
1	CMDT US ARMY INFANTRY SCHOOL ATTN ATSH CD SECURITY MGR FT BENNING GA 31905-5660
	ABERDEEN PROVING GROUND
2	DIR USAMSAA ATTN AMXSY D AMXSY MP H COHEN
1	CDR USATECOM ATTN AMSTE TC
1	DIR USAERDEC ATTN SCBRD RT
1	CDR USACBDCOM ATTN AMSCB CII
1	DIR USARL ATTN AMSRL SL I
5	DIR USARL ATTN AMSRL OP AP L

NO. OF COPIES	ORGANIZATION
1	HQDA SARD TR MS K KOMINOS WASHINGTON DC 20310-0103
1	HQDA SARD TR DR R CHAIT WASHINGTON DC 20310-0103

USER EVALUATION SHEET/CHANGE OF ADDRESS

This Laboratory undertakes a continuing effort to improve the quality of the reports it publishes. Your comments/answers to the items/questions below will aid us in our efforts.

1. ARL Report Number ARL-TR-661 Date of Report December 1994

2. Date Report Received _____

3. Does this report satisfy a need? (Comment on purpose, related project, or other area of interest for which the report will be used.) _____

4. Specifically, how is the report being used? (Information source, design data, procedure, source of ideas, etc.) _____

5. Has the information in this report led to any quantitative savings as far as man-hours or dollars saved, operating costs avoided, or efficiencies achieved, etc? If so, please elaborate. _____

6. General Comments. What do you think should be changed to improve future reports? (Indicate changes to organization, technical content, format, etc.) _____

CURRENT ADDRESS

_____
Organization

_____
Name

_____
Street or P.O. Box No.

_____
City, State, Zip Code

7. If indicating a Change of Address or Address Correction, please provide the Current or Correct address above and the Old or Incorrect address below.

OLD ADDRESS

_____
Organization

_____
Name

_____
Street or P.O. Box No.

_____
City, State, Zip Code

(Remove this sheet, fold as indicated, tape closed, and mail.)
(DO NOT STAPLE)

DEPARTMENT OF THE ARMY

OFFICIAL BUSINESS

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO 0001, APG, MD

Postage will be paid by addressee

Director
U.S. Army Research Laboratory
ATTN: AMSRL-OP-AP-L
Aberdeen Proving Ground, MD 21005-5066