Evaluating Parallel Languages for Molecular Dynamics

Terry W. Clark Reinhard von Hanxleden
Ken Kennedy Charles Koelbel
L. Ridgway Scott

CRPC-TR92202
February, 1992

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

Evaluating Parallel Languages for Molecular Dynamics Computations*†

Terry W. Clark
Department of Computer Science, University of Houston, Houston, TX 77204

Reinhard v. Hanxleden Ken Kennedy Charles Koelbel
Department of Computer Science, Rice University, Houston, TX 77251

L. Ridgway Scott
Department of Mathematics, University of Houston, Houston, TX 77204

Abstract

Computational molecular dynamics is an important application requiring large amounts of computing time. Parallel processing offers very high performance potential, but irregular problems like molecular dynamics have proven difficult to map onto parallel machines. In this paper, we describe the practicalities of porting a basic molecular dynamics computation to a distributed-memory machine. In the process, we show how program annotations can aid in parallelizing a moderately complex code. We also argue that algorithm replacement may be necessary in parallelization, a task which cannot be performed automatically. We close with some results from a parallel GROMOS implementation.

1 Introduction

The purpose of this paper is to examine the practicalities of parallelizing the basic algorithms of molecular dynamics for distributed-memory multiprocessors using annotations to sequential Fortran programs. This set of algorithms represents an important class of unstructured problems in scientific computation. In general, however, unstructured computations are difficult to map onto parallel systems using either automatic or hand-written techniques. Parallelizing molecular dynamics is therefore a problem of great intellectual and practical importance.

*This research was supported in part by the National Science Foundation through award number DMS-8903548, and by the Center for Research on Parallel Computation, a National Science Foundation Science and Technology Center, under the NSF cooperative agreement CCR-8809615.

†A version of this paper is submitted to IEEE Computer Society Press for Publication in the Proceedings of the Scalable High Performance Computing Conference to be held in Williamsburg, Virginia, April 26–29, 1992.

The greatest potential for parallelism in many scientific codes, including molecular dynamics, is due to *data parallelism*, meaning operations applied to all elements of a large data structure. Exploiting data parallelism on distributed-memory MIMD machines requires careful partitioning of the data and computation for good locality, which can be beneficial for shared-memory machines as well [16]. A number of language extensions have been studied which allow the programmer to provide such distribution information. In this work, we examine the facilities available in two such extended languages. We also consider the tradeoffs of parallelizing existing sequential code ("dusty deck") and writing a parallel program from scratch. Our examples will illustrate the need for modification of algorithms to achieve scalability.

The remainder of this paper is organized as follows. Section 2 describes the GROMOS code [10], a standard molecular dynamics program that we are parallelizing [4]. Section 3 gives a short description of the parallel languages used, IPfortran and Fortran D. Sections 4, 5, and 6 each describe the parallelization of one phase of GROMOS. Section 7 gives some performance results, followed by conclusions in Section 8.

2 Molecular dynamics

First developed for simulating atomic motion in simple liquids, molecular dynamics is used routinely to simulate biomolecular systems [17]. Using the compute-intensive data obtained from a molecular dynamics simulation, various kinetic, thermodynamic, mechanistic, and structural properties can be obtained [18]. In molecular dynamics, the motion of each atom, represented as a point mass, is determined by the forces exerted on it by other atoms.

Molecular dynamics algorithms commonly iterate

over the sequence:

1. Calculate bonded and nonbonded forces on each atom as the analytical gradient of a potential-energy function of the atom positions. The pairwise, nonbonded interactions dominate the computation with $\mathcal{O}(N^2)$ time complexity [5] and therefore are key considerations in both the model and its implementation [4], see Sections 4 and 5.

2. Integrate Newton's equations of motion to determine the new atomic momenta and positions. By removing uninteresting, high-frequency motions, larger timesteps can be taken resulting in a more efficient computer utilization [19, 20]. This usually involves the constraining of some molecular motions, as discussed in Section 6.

3. Save data as appropriate for post analysis.

The molecular dynamics program used in this study is from the GROMOS (GROningen MOlecular Simulation) suite designed for the dynamic modeling of biomolecules [10]. GROMOS provides programs for the simulation of biological molecules (and arbitrary molecules) using *molecular dynamics* or *stochastic dynamics*. In addition, *energy minimization* and *analysis* programs are provided. The approximately 127 files comprising GROMOS consist of about 74,000 lines of FORTRAN77, comments included.

3 Parallel languages

Both IPfortran and Fortran D aim at providing a more convenient interface for programming parallel machines. Users rightfully expect compilers to handle the low-level, machine-dependent details of programming. A familiar example of this is register allocation on sequential machines. On parallel architectures, an equivalent challenge is generating synchronization and interprocessor communication operations. Both languages provide support for this, in contrast to other languages which required those operations to be programmed explicitly. The languages differ, however, in that one uses a local memory model and the other uses a global memory model.

IPfortran utilizes a local memory model [1]; the key concept of IPfortran is to provide a better abstraction for interprocessor communication than simple message-passing [15]. IPfortran programs use an SPMD (Single-Program Multiple-Data) style of programming. Variables are implicitly local to each processor; thus, X on processor 1 may have a different value from X on processor 2. Nonlocal accesses are denoted by the @ operator, so $A(i)@j$ means the i-th element of array A on processor j. Note that only the processor *using* a nonlocal value must reference it, and that the reference may be made within a larger expression. This is in contrast to message-passing languages, which require matching but separate "send" and "receive" operations. Global reductions are also supported. For example, $+\{X\}$ denotes the sum of all values of X on all processors. This set of nonlocal access and reduction operations supports programming at a convenient level of abstraction, while still allowing a relatively simple compiler to produce excellent code.

Fortran D utilizes a global memory model, providing a modified shared name-space for array elements [7]. Arrays are declared to be their full, global size and are *aligned* with virtual *decompositions* which are *distributed* across processors. Statements are executed sequentially (except for the **FORALL** loop, for which the iterations conceptually execute simultaneously). The compiler must detect and exploit opportunities for parallel execution, in addition to inserting any necessary communication. This requires sophisticated compiler technology, but the potential gain is an additional level of machine independence.

4 Nonbonded force calculation

The GROMOS code approximates nonbonded forces by calculating them only for atom pairs which are within a certain *cutoff* radius of each other. As described in Section 5, these pairs are stored in a pair list which is updated in regular intervals. In the original code, this pair list is represented by two arrays, INB and JNB. $INB(I)$ gives the number of partners of atom I, and JNB can be thought of as a concatenation of lists of partners, one list for each atom. We also introduce the arrays $firstJ$ and $lastJ$, so that the array section $INB(firstJ(I) : lastJ(I))$ gives the list of partners of atom I. Obviously, $INB(I) = lastJ(I) - firstJ(I) + 1$.

An important optimization is due to the fact that for each force exerted by an atom A on an atom B, atom B exerts an equal but opposite force on atom A. We therefore can cut the number of force calculations in half by storing each atom pair only once, for example by storing only partners with a higher atom index. The resulting sequential version for N atoms is shown in Figure 1, where we assume that the force array F is initialized to 0 (similarly in the following code samples). Note also that in practice the forces F and the positions X are vectors in $\mathbb{R}^3$.

In the IPfortran implementation [4], each processor executes the outer loop of the sequential version for a range $(firstI(me) : lastI(me))$ of atom indices. (Here,

```
DO I = 1, N
 DO J = firstJ(I), lastJ(I)
  force = nbf(X(I) − X(JNB(J)))
  F(I) = F(I) + force
  F(JNB(J)) = F(JNB(J)) − force
 ENDDO
ENDDO
```

Figure 1: Sequential form of the nonbonded force calculation.

as throughout the text and the code samples, *me* stands for the local processor id, and P stands for the total number of processors.) This range is determined in an initial call to a load balancing routine, of which gathering the corresponding portion of JNB, namely $myJNB$, is the most time consuming part. The code finishes with accumulating forces across processors, see Figure 2. An important point to keep in mind is that the

```
Balance(firstI, lastI)
DO I = firstI(me), lastI(me)
 DO J = firstJ(I), lastJ(I)
  force = nbf(X(I) − X(myJNB(J)))
  F(I) = F(I) + force
  F(myJNB(J)) = F(myJNB(J)) − force
 ENDDO
ENDDO
F = +{F}
```

Figure 2: Force calculation, IPfortran version.

atom numbering and the resulting pair list do not inherently have a good locality, *i.e.*, atoms close together in space do not necessarily have similar numbers. This is inherited from sequential GROMOS, another parallel version we are currently developing overcomes this limitation using a *hierarchical decomposition* [6].

Note that this implementation *replicates* the force array F. A Fortran D version of this kind of algorithm can be written by expanding each array by one dimension (the *processor dimension*), the introduced index being the processor number, and then distributing that dimension blockwise. The resulting code is shown in Figure 3.

However, this approach does not really take advantage of the Fortran D data distributions, and it ultimately limits our scalability. Instead of replicating the force array F, we should distribute it to distribute the workload. To allow load balancing, we distribute the data irregularly using a mapping array Map [7], for which

$$firstI(p) \leq I \leq lastI(p) \Leftrightarrow Map(I) = p.$$

must hold.

```
DECOMPOSITION AtomD(N, P), PairD(MP, P)
DISTRIBUTE AtomD(*, BLOCK), PairD(*, BLOCK)
ALIGN FF,X with AtomD, myJNB with PairD

FORALL me = 0, P−1
 DO I = firstI(me), lastI(me)
  DO J = firstLocJ(I), lastLocJ(I)
   force = nbf(X(I,me) − X(myJNB(J,me)))
   FF(I,me) = FF(I,me) + force
   FF(myJNB(J,me),me) = FF(myJNB(J,me),me)−force
 ENDDO
 ENDDO
ENDFORALL

FORALL me = 0, P−1
 FORALL I = 1, N
  REDUCE(SUM, F(I), FF(I,me))
 ENDFORALL
ENDFORALL
```

Figure 3: Force calculation, Fortran D version 1. MP is the maximal number of pairs per processor.

We should also rethink how to distribute the neighbor list JNB, which represents the largest data structure of the problem. In the IPfortran version this was done by replacing the global pair list JNB with just portions thereof, $myJNB$. We then modeled that with our first Fortran D version by adding an extra processor dimension. In our second Fortran D version, we also use a two dimensional structure, $JNBL$, but now the second dimension does not represent a processor number, but a partner number instead. So, $JNBL(I,:)$ represents all of the partners of atom I whose atom number is greater than I. We have

$$JNBL(I, 1 : INB(I)) = JNB(firstJ(I) : lastJ(I)).$$

We guide the compiler by using a **FORALL** loop to indicate the fact that the operations are independent. For enhanced locality, we combine that with an **ON** clause. The resulting code is shown in Figure 4.

It is worthwhile to aid our intuition about actual costs of computation and communication and about locality in general with some analysis. We start with defining a predicate which indicates whether an atom pair (I, K) is stored in the pair list:

$$isPair(I,K) = \begin{cases} 1 & \text{if } \exists J,\ JNBL(I,J) = K, \\ 0 & \text{otherwise.} \end{cases}$$

We are also interested in how many force calculations involving atom K we have to perform on processor p:

$$Partners(p,K) = \sum_{I=firstI(p)}^{lastI(p)} isPair(I,K).$$

```
DECOMPOSITION atomD(N), PartnD(N, MaxINB)
DISTRIBUTE atomD(Map), PartnD(Map, *)
ALIGN F,X WITH atomD, JNBL WITH PartnD

FORALL I = 1, N ON HOME F(I)
  DO J = 1, INB(I)
    force = nbf(X(I) - X(JNBL(1,J)))
    REDUCE(SUM, F(I), force)
    REDUCE(SUM, F(JNBL(I,J)), -force)     (*)
  ENDDO
ENDFORALL
```

Figure 4: Force Calculation, Fortran D version 2.

For each processor p, we sum this up over all atoms:

$$Pairs(p) = \sum_{K=1}^{N} Partners(p, K) =$$

$$\sum_{I=firstI(p)}^{lastI(p)} \sum_{K=1}^{N} isPair(I, K) = \sum_{I=firstI(p)}^{lastI(p)} INB(I).$$

In terms on averages (which are denoted by a subscript ave), we have

$$Pairs_{ave} = \frac{1}{P} \sum_{I=1}^{N} INB(I) = \frac{N \times INB_{ave}}{P}.$$

We notice that $Pairs(p)$ is proportional to the computational load of processor p. Therefore, the overall computational cost is given by

$$T_{comp} \propto \max_{p=1}^{P} Pairs(p) = \max_{p=1}^{P} \sum_{I=firstI(p)}^{lastI(p)} INB(I).$$

This results in a typical load balancing problem [12], where the goal is to lower T_{comp} down to $T_{ideal} \propto Pairs_{ave}$. This means that we have to choose $firstI$ and $lastI$ such that we have for all p:

$$\sum_{I=firstI(p)}^{lastI(p)} INB(I) \approx \frac{N \times INB_{ave}}{P}$$

After considering the computational costs, we now take a closer look at communication costs and, closely related to that, scalability. Most of the overall communication is associated with the potentially nonlocal assignment to the force array (statement (*) in Figure 4). Compiling this naïvely, without message blocking, would result in roughly $Pairs_{ave}$ (short) messages per processor. For systems whose communication time to send m units of data is well modeled by $\lambda + \beta m$ with $\lambda \gg \beta$ [3], this would increase the communication cost by a factor λ/β. This would make it unacceptable to send individual messages for each nonlocal access, instead of combining them at the end of the loop.

Another issue besides raw message blocking is how much we can gain by combining non-local reductions. For example, if processor p has to make several contributions to an $F(I)$ owned by processor q, $p \neq q$, then it would be profitable to combine these contributions on p which then sends only the sum to q. We should estimate how often this may occur. While the assignments may be skewed for various reasons (for example, because a pair (I, K) is stored only if $I < K$), the average number of assignments which involve a particular atom, per processor, is simply $Pairs_{ave}$ divided by N:

$$Partners_{ave} = \frac{INB_{ave}}{P}.$$

Thus for $P \ll INB_{ave}$, we can expect every processor to contribute to most elements of F; for $P \gg INB_{ave}$, each processor contributes to very few elements of F. Therefore, the ratio $Partners_{ave}$ is crucial for how we should distribute F and perform the reduction operation, which we will discuss subsequently.

In the IPfortran version and the corresponding first Fortran D version, F is replicated and all local contributions to an element of F are summed up locally. After computing the global sum $\{F\}$, all forces are everywhere available. Using a simple dimensional exchange, this summation can be done in $\mathcal{O}(N \log P)$, with $\log P$ messages per processor (this works best for hypercubes, but can also be done on other parallel architectures). A more sophisticated divide-and-conquer approach works in $\mathcal{O}(N)$, with $2 \times \log P$ messages per processor [8]. If we use the latter approach and have a balanced workload, then we have communication cost $T_{comm} \propto N$ and computation costs $T_{comp} \propto Pairs_{ave}$. Their ratio then is

$$R_{comp/comm} = Pairs_{ave}/N = INB_{ave}/P,$$

which is again $Partners_{ave}$. Thus for fixed INB_{ave}, the cost of communication will dominate for large P, independent of N. In the experiments with GROMOS parallelized in this way, for a system with $INB_{ave} \approx 80$, the cross-over point occurs near 128 processors on the Intel iPSC/860, see Figure 10. However, INB_{ave} increases as the cube of the cutoff radius, so that the break-even number of processors, for which $P \propto INB_{ave}$, increases cubicly with the cutoff radius as well. In the limit of having an infinite cutoff radius (that is, no cutoff at all), we have $INB_{ave} = N$ and the communication costs are swamped by the computational cost for any $N \gg P$. In that case the algorithm is fully scalable as N increases.

In our second Fortran D version we do not replicate F, but instead distribute it across processors. However, to allow message blocking, the compiler should still provide buffer space for all elements of F which are accessed nonlocally. The global combination would then be performed via a reduction operation like *scatter_add* [2]. A typical implementation of *scatter_add* would again sum all local contributions to a particular array element up before combining them globally with other contributions. However, it would probably not use a dimensional exchange, but instead send point-to-point messages. The size of each of these messages would be $\mathcal{O}(N/P \times \min(1, Partners_{ave}))$, where the limiting 1 stems from combining reductions locally. Whether the overall cost of a *scatter_add* would be less than a global exchange of the full array depends on $Partners_{ave}$.

To summarize, it appears that distributing F is advantageous when $P \geq INB_{ave}$ roughly holds, otherwise replicating F is appropriate.

5 Pair list generation

Figure 5 shows the original version of the generation of the pair list used in the nonbonded force calculation. The critical obstacle for parallelizing this algorithm is

```
J0 = 0
JJ = 0
DO I = 1, N
 firstJ(I) = JJ + 1
 DO J = I+1, N
  IF ABS(X(I) - X(J)) < R THEN
   JJ = JJ + 1
   JNB(JJ) = J
  ENDIF
 ENDDO
 INB(I) = JJ - J0
 lastJ(I) = JJ
 J0 = JJ
ENDDO
```

Figure 5: Pairlist computation, original version.

the loop carried dependence on JJ, which causes a dependence for the assignment to *JNB*. However, the dependency does not occur in the *values* of *JNB*, but rather in the *locations* of the values. Thus we may compute each iteration separately and figure out later where to put them. This amounts to computing the *JNBL* array which appeared already in Figure 4. *JNBL* serves again to remove dependences as shown in Figure 6, similarly to scalar privatization. Here *JNBL* can also be viewed as a "scratch space" to hold the values

```
DO I = 1, N
 JL = 0
 DO J = I+1, N
  IF ABS(X(I) - X(J)) < R THEN
   JL = JL + 1
   JNBL(JL,I) = J
  ENDIF
 ENDDO
 INB(I) = JL
ENDDO

DO I = 1, N
 JNB(firstJ(I):lastJ(I)) = JNBL(I, 1:INB(I))
ENDDO
```

Figure 6: Pairlist computation, parallel-vector version.

which are then combined into *JNB*. One difficulty in doing this automatically is that the second dimension of *JNBL* has to be large enough to fit $max_{I=1}^{N} INB(I)$ elements, for which there is no *a priori* bound that could be easily derived by a compiler.

Now we can parallelize the outer loop by letting each processor p compute a range $(firstI(p) : lastI(p))$ of the iterations. In IPfortran we use processor specific loop bounds; the Fortran D version distributes *JNBL* and uses the owner computes rule. The range sizes vary across processors due to the triangular shape of the double loop [4]. Each processor will have a local copy of only *part* of the *JNB* array. However, for performing load balancing we need to know all of *INB*, which can be collected in $\mathcal{O}(N)$ with $\mathcal{O}(\log P)$ communication steps using a dimensional exchange [8], as seen for example in Figure 7.

6 The SHAKE algorithm

SHAKE utilizes a typical form of relaxation to solve a system of constraints regarding the distance (or angles) between particular atoms. Mathematically, there is a system of constraints, $\gamma_i(x_1, \ldots, x_n) = 0$, $1 \leq i \leq k$, on the positions, x_j, of atoms. Typically, this is extremely sparse, with γ_i depending only on a small number of x_j. Although the constraints are not linear (for example, they involve the distance between two points which includes expressions quadratic in the coordinates), any given constraint can be satisfied exactly by moving only one atom, with the others fixed. Thus, the SHAKE algorithm iterates on i and moves a particular atom, j_i, so that $\gamma_i(x_1, \ldots, \hat{x}_{j_i}, \ldots, x_n) = 0$. Since the previously computed values of x_j are used, this can be viewed as a nonlinear Gauss-Seidel method.

To make the analogy precise, consider a system of linear equations,

$$a_{i1}\xi_1 + \cdots + a_{in}\xi_n - f_i = 0, \quad i = 1, \ldots, n$$

(think $\xi_i = x_{j_i}$ and $\gamma_i \approx a_{i1}\xi_1 + \cdots + a_{in}\xi_n - f_i$). Then the i-th step of a Gauss-Seidel iteration is

$$\xi_i = \left(\sum_{j \neq i} a_{ij}\xi_j - f_i \right) \Big/ a_{ii}.$$

It is well known that the Gauss-Seidel method does not parallelize well due to the fact that each step of the iteration depends on previous ones. A simple solution would be to use instead the Jacobi iteration

$$\xi_i^{\text{new}} = \left(\sum_{j \neq i} a_{ij}\xi_j^{\text{old}} - f_i \right) \Big/ a_{ii}, i = 1, \ldots, n,$$

followed by the assignment $\xi^{\text{new}} \leftarrow \xi^{\text{old}}$. This is now perfectly parallelizable, but is (usually) a more slowly convergent algorithm than Gauss-Seidel [9].

To overcome this problem, it is common to use a compound algorithm for parallel computation which involves a Jacobi iteration across processors, but a Gauss-Seidel iteration interior to each processor. More precisely, the equations are partitioned into P sets, $\mathcal{I}_p$, $p = 0, \ldots, P-1$, and the iteration in each processor becomes

$$\xi_i = \left(\sum_{j \neq i} a_{ij}\xi_j - f_i \right) \Big/ a_{ii}, \quad i \in \mathcal{I}_p.$$

At the completion of this, all processors exchange values of ξ_j as necessary. Suppose that the sets $\mathcal{I}_p$ were the ranges $1 \leq j - pn/P \leq n/P$ where for simplicity we assume that P divides n. In IPfortran this can be written as in Figure 7. Note the dimensional exchange loop at the end, which has $\mathcal{O}(\log P)$ messages with total volume of $\mathcal{O}(N)$ per processor.

When coding SHAKE in Fortran D, it is tempting to write the code in Figure 8. However, this is just Gauss-Seidel in a more complex form, since the innermost loop references all of J. As it turns out, there is no simple sequential algorithm that can be annotated to yield the hybrid relaxation. In this kind of situation, where we want to use different algorithms depending on whether we operate within processors or across them, it is useful to have an escape mechanism from the default, global level into the processor level.

Here the concept of *local blocks* is very useful [13, Thinking Machines proposal]. The Fortran D code in Figure 9 shows how a **FORALL** loop, which by definition performs all iterations independently of each other, can realize this concept.

```
much = n / P
myslice = me * much
DO i = myslice+1, myslice+much
  XI(i) = 0.0
  DO j = 1,n
    XI(i) = AMATRIX(i,j) * XI(j)
  ENDDO
  XI(i)= (XI(i)−F(i)) / AMATRIX(i,i)
ENDDO

mask = 1
DO d = 1, CUBEDIM
  a = 1 + (XOR(me, mask)/mask) * mask * much
  b = a − 1 + mask * much
  XI(a : b) = XI(a : b) @ XOR(me, mask)
  mask = 2 * mask
ENDDO
```

Figure 7: Hybrid relaxation: Gauss-Seidel locally, then Jacobi update globally. IPfortran version.

```
much = n / P
DO ip = 0, P−1
  myslice = ip * much
  DO i = myslice+1, myslice+much
    XI(i) = 0.0
    DO j = 1, n
      XI(i) = AMATRIX(i,j) * XI(j)
    ENDDO
    XI(i) = (XI(i)−F(i)) / AMATRIX(i,i)
  ENDDO
ENDDO
```

Figure 8: Hybrid relaxation, Fortran D version 1.

7 Performance results

Figure 10 gives performance results for the parallel implementation of GROMOS using IPfortran. The calculation on an iPSC/860 uses a model for the enzyme Superoxide Dismutase [22], with a total of 6968 atoms, for 500 timesteps. Both the overall execution times and the breakdowns into the principal sections of the calculation are given.

The dominating parts of the sequential algorithm, the nonbonded forces and pair list, have been parallelized with nearly perfect speedup. Time-stepping and the bonded force calculation, performed redundantly at every processor, remain constant. These, together with the load balancing and global sum of the nonbonded force, form an asymptote which the total time approaches as the number of processors is increased. As the problem size, N, is increased, the pair list construction (and the long-range force calculation which is computed along with the pair list) grows as $\mathcal{O}(N^2)$;

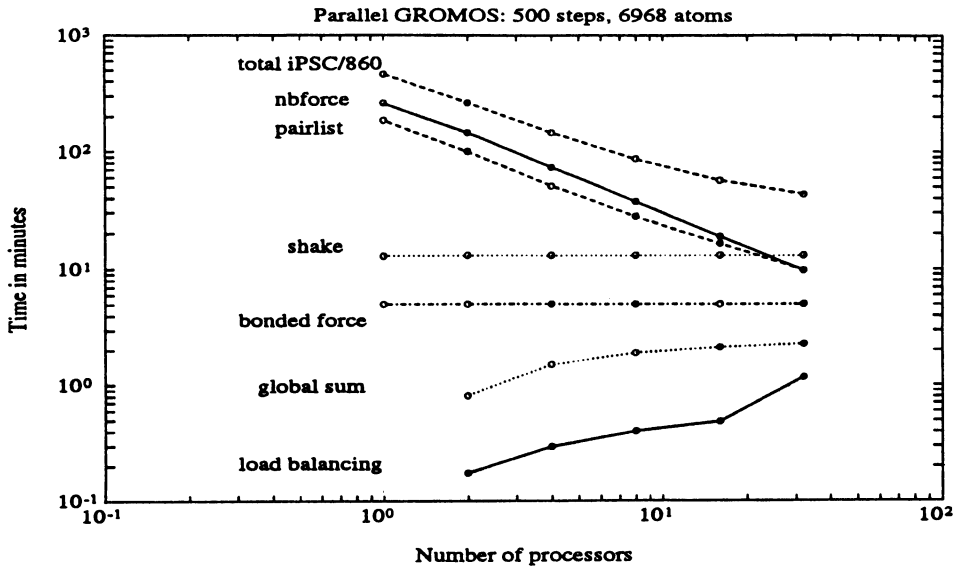

Figure 10: Performance results.

```
DECOMPOSITION CoordD(N)
DISTRIBUTE CoordD(BLOCK)
ALIGN F,X with CoordD

much = n / P
FORALL p = 0, P-1
 myslice = p * much
 DO i = myslice+1, myslice+much
  XI(i) = 0.0
  DO j = 1,n
   XI(i) = AMATRIX(i,j) * XI(j)
  ENDDO
  XI(i)= (XI(i)-F(i)) / AMATRIX(i,i)
 ENDDO
ENDFORALL
```

Figure 9: Hybrid relaxation, Fortran D version 2.

all other parts increase by $\mathcal{O}(N)$, with the exception of load balancing which depends on initial data and, relative to this discussion, is difficult to characterize. Thus we can assert that this simple port of the GROMOS code is an efficient parallelization. Increasing the cutoff radius while keeping N fixed also increases the number of processors that can be used effectively, as discussed in Section 4, thereby shifting the nonbonded force curve up by some δ.

8 Conclusions

Because this study was limited to a single example program and two languages, we cannot make broad generalizations regarding parallel programming. However, we believe that for a range of scientific codes whose complexity is comparable to molecular dynamics the following observations will apply.

- Scalability in molecular dynamics is an achievable goal, but it requires careful algorithm design where the choice of the right algorithm may also depend on the input characteristics.
- Both local and global models are feasible interfaces for programming distributed-memory parallel machines [21]. In addition, both models can provide the user with a higher-level programming interface than current message-passing languages on distributed-memory machines.
- As to be expected, regardless of the implementation model chosen, the key to a good parallel program is the choice of an appropriate algorithm. This implies that programs will have to be rewritten to some extent for parallelization, rather than relying on compiler optimization of "dusty deck" code. Resembling the "vectorizable style" associated with vector architectures [23], an equivalent challenge in the parallel arena is the development of a "machine-independent parallel programming style," a long-term research target of Fortran D [14].
- IPfortran improves on the usual message-passing environment by eliminating the need for explicit "send" operations. This simplifies code significantly, allowing the programmer to concentrate on

higher-level problems such as load balancing. The local model of computation is useful if we want to apply distinct "intra-processor" and "inter-processor" algorithms.

- Fortran D abstracts communication statements out of the program text by providing a modified shared-memory model of computation. The key feature of a Fortran D program is the data distribution, which the compiler uses to generate the low-level communications operations.

We plan to continue this work in several areas. In the computational molecular dynamics area, we will continue to design and implement scalable algorithms, producing high-performance codes [6]. Both the IPfortran and Fortran D implementations will continue to go forward, and the lessons from this study and others like it will affect their development [11, 13].

9 Acknowledgements

We thank Professor McCammon for many helpful discussions and the Institute for Molecular Design (IMD) and Intel for providing computing facilities for this work. McCammon and the IMD are supported in part by the National Science Foundation with additional support from Intel. We also acknowledge the use of computer facilities at Oak Ridge National Laboratory in this research.

References

[1] Babak Bagheri, Terry W. Clark, and L. Ridgway Scott. IPfortran (a parallel extension of Fortran) reference manual. Research Report UH/MD–119, Dept. of Mathematics, University of Houston, 1991.

[2] H. Berryman, J. Saltz, and J. Scroggs. Execution time support for adaptive scientific algorithms on distributed memory machines. *Concurrency: Practice and Experience*, 3(3):159–178, June 1991.

[3] D. K. Bradley. First and second generation hypercube performance. Technical Report UIUCDCS–R–88–1455, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1988.

[4] Terry W. Clark, J. A. McCammon, and L. Ridgway Scott. Parallel molecular dynamics. In *Proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Computing*, Houston, TX, March 1991.

[5] Terry W. Clark and J. Andrew McCammon. Parallelization of a molecular dynamics non-bonded force algorithm for MIMD architectures. *Computers & Chemistry*, 14(3):219–224, 1990.

[6] Terry W. Clark, Reinhard v. Hanxleden, and L. Ridgway Scott. Scalable algorithms for molecular dynamics computations. Technical report, Dept. of Mathematics, University of Houston, to appear.

[7] G. C. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D language specification. Technical Report TR90-141, Dept. of Computer Science, Rice University, December 1990. Revised April, 1991.

[8] G. C. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Multiprocessors*. Prentice-Hall, 1988.

[9] Girija Ganti and J. Andrew McCammon. Transport properties of macromolecules by Brownian dynamics simulation: Vectorization of Brownian dynamics on the Cyber-205. *Journal of Computational Chemistry*, 7(4):457–463, 1986.

[10] W. F. van Gunsteren and H. J. C. Berendsen. GROMOS: GROningen MOlecular Simulation software. Technical report, Laboratory of Physical Chemistry, University of Groningen, Nijenborgh, The Netherlands, 1988.

[11] Reinhard v. Hanxleden and Ken Kennedy. Relaxing SIMD control flow constraints using loop transformations. In *Proceedings of the ACM SIGPLAN '92 Conference on Program Language Design and Implementation*, San Francisco, CA, 1992.

[12] Reinhard v. Hanxleden and L. Ridgway Scott. Load balancing on message passing architectures. *Journal of Parallel and Distributed Computing*, 13, 1991.

[13] *Proceedings of the High Performance Fortran Forum*, Houston, TX, Jan. 27–28, 1992.

[14] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C. Tseng. An overview of the Fortran D programming system. In *Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing*, Santa Clara, CA, August 1991.

[15] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, NJ, 1985.

[16] C. Lin and L. Snyder. A comparison of programming models for shared memory multiprocessors. In *Proceedings of the 1991 International Conference on Parallel Processing, Vol. 2*, pages 163–170, 1991.

[17] J. Andrew McCammon. Computer-aided molecular design. *Science*, 238:486–491, October 1987.

[18] J. Andrew McCammon and Stephen C. Harvey. *Dynamics of proteins and nucleic acids*. Cambridge University Press, Cambridge, 1987.

[19] Florian Müller-Plathe and David Brown. Multi-colour algorithms in molecular simulation: Vectorisation and parallelisation of internal forces and constraints. *Computer Physics Communications*, 64:7–14, 1991.

[20] Jean-Paul Rycaert, Giovanni Ciccotti, and Herman J.C. Berendsen. Numerical integration of the cartesian equations of motion of a system with constraints: Molecular dynamics of n-alkanes. *Journal of Computational Physics*, 23:327–341, 1977.

[21] L. R. Scott, J. M. Boyle, and B. Bagheri. Distributed data structures for scientific computation. In M. T. Heath, editor, *Proceedings of the 3rd Hypercube Multiprocessors Conference*, pages 55–66, Philadelphia, PA, 1987.

[22] Jian Shen and J. Andrew McCammon. Molecular dynamics simulation of Superoxide interacting with Superoxide Dismutase. *Chemical Physics*, 158:191–198, 1991.

[23] Michael J. Wolfe. Semi-automatic domain decomposition. In *Proceedings of the 4th Conference on Hypercube Concurrent Computers and Applications*, Monterey, CA, March 1989.