Experiences in Data-Parallel Programming

TERRY W. CLARK,[1,*] REINHARD v. HANXLEDEN,[2,†] AND KEN KENNEDY[2]

[1]*Texas Center for Advanced Molecular Computation, University of Houston, Houston, TX 77204, USA*
[2]*Center for Research on Parallel Computation, Rice University, Houston, TX 77251, USA; e-mail: ken@rice.edu*

ABSTRACT

To efficiently parallelize a scientific application with a data-parallel compiler requires certain structural properties in the source program, and conversely, the absence of others. A recent parallelization effort of ours reinforced this observation and motivated this correspondence. Specifically, we have transformed a Fortran 77 version of GROMOS, a popular dusty-deck program for molecular dynamics, into Fortran D, a data-parallel dialect of Fortran. During this transformation we have encountered a number of difficulties that probably are neither limited to this particular application nor do they seem likely to be addressed by improved compiler technology in the near future. Our experience with GROMOS suggests a number of points to keep in mind when developing software that may at some time in its life cycle be parallelized with a data-parallel compiler. This note presents some guidelines for engineering data-parallel applications that are compatible with Fortran D or High Performance Fortran compilers.

1 INTRODUCTION

One concern often not foremost in a scientific programmer's mind at the outset of software development is parallelization. Yet, even for scientific applications developed for sequential execution, it is not unlikely that someone at sometime will parallelize the software. As it turns out, some programming styles are easier to parallelize than others. Moreover, for programs to yield to the data-parallel approach of compilers for Fortran D [1] or High Performance Fortran (HPF) [2], certain structural properties must be present in the software. Elements of a program style congruous with an HPF compiler include, for example, consistent distribution and use of data arrays and structured flow of control. It appears that writing such programs from the outset largely embraces good software engineering techniques (such as the structured-programming approach advocated by Dijkstra [3]). In this article, we discuss some of these requirements and provide guidelines for engineering data-parallel applications to be compatible with compilers for data-parallel languages. Alternatively, the observations presented here could also serve as guidelines for making an existing program suitable for data-parallel compilation. In the following, we will first outline some general guidelines, then illustrate them with short code examples, and finally give some additional suggestions to facilitate effective data parallelism.

One underlying idea of data-parallel languages, such as Fortran D or HPF, is that the user does not explicitly specify the parallelism inherent in a program, but instead annotates the program with directives on how to distribute the data, and lets the com-

Received February 1995
Revised January 1996

*Present address: Pacific Northwest National Laboratory, Richland, WA 99352, USA; e-mail: twclark@emsl.pnl.gov
†Present address: Daimler-Benz AG, Responsive Systems, Berlin, Germany; e-mail: vhanx@DBresearch-berlin.de

Scientific Programming, Vol. 6, pp. 153–158 (1997)
CCC 1058-9244/97/010153-06

piler work from there. Performance and investment-preserving independence from environment-specific details are two key objectives. The art of data-parallel programming might be defined as achieving the former without compromising the latter. For this correspondence, this also means that while the programmers should understand certain characteristics of data parallelism, they should not have to develop a style which will have an adverse effect on code readability, maintainability, or efficiency in nondata-parallel environments.

Performance depends on the degree of parallelism and on overheads, such as synchronization and communication costs.* Compilers for distributed memory architectures typically try to achieve parallelism by distributing loop iterations across processors. The first guideline for designing data-parallel programs should therefore be:

The flow of control should be structured; e.g., DO-*loops are preferable to* GOTOs.

Distributing loop iterations is commonly driven by some heuristic for minimizing communication, such as the owner-computes rule [4] or variations thereof. This heuristic may fail even for "embarrassingly parallel" algorithms if the program expressing the algorithm obscures the parallelism by some (perhaps apparently unrelated) means. Even though there are many other issues crucial for achieving successful parallelization, realizing these points and designing program and data structures accordingly should in themselves go a long way toward good performance for many applications.

The perhaps most important guideline that should drive design decisions is:

Loops and arrays should match.

That is, in computationally intensive code regions, array subscripts and loop indices should be related to each other in a simple manner, allowing the compiler to derive a loop parallelization directly from an array distribution. To justify this guideline, let us briefly digress into the workings of a data-parallel compiler for distributed memory machines, such as the Fortran D compiler prototype at Rice University.

Assume the compiler is given a simple code segment

```
PROGRAM Good1FortD                    PROGRAM Good1Node
REAL x(100)                           REAL x(25)
DISTRIBUTE x(BLOCK)
                                      do i = 1, 25
do [ i ] = 1, 100                        x(i) = ...
   x([ i ]) = ...                     ENDDO
ENDDO
```

FIGURE 1 Simple example loop with matching array access: the Fortran D program (left) can be compiled into a node program (right) with reduced loop bounds (assuming $N_{proc} = 4$). Distributed array and loop indices are framed.

as shown in Figure 1 on the left, $\texttt{Good1}_{\texttt{FortD}}$.† The Fortran D compiler will generate a node program $\texttt{Good1}_{\texttt{Node}}$, shown in Figure 1 on the right, which in turn will be compiled by the native compiler of the target machine.‡ This program will be written in local name space, as opposed to the single, global name space of the Fortran D program. It will contain the instructions for an individual processor, and it may contain communication statements. Here we will focus on the distribution of computation; for a discussion of communication generation and other compilation issues refer to other publications [5, 6]. The compiler tries to parallelize the `i`-loop in $\texttt{Good1}_{\texttt{FortD}}$ by applying the owner-computes rule to the distributed array reference, `x(i)`. Assuming no sequentializing dependences on the rhs of the assignment, the owner-computes rule works fine here since induction variable `i` and array subscript `i` are in a simple relationship — they are identical. The loop and the array *match*. The compiler can fully apply the owner-computes rule at compile-time and perform loop-bounds reduction: Assuming that there are $N_{proc} = 4$ processors, each processor will perform only one fourth of the total number of iterations.

Now consider the program $\texttt{BAD1}_{\texttt{FortD}}$ in Figure 2. Similar to $\texttt{Good1}_{\texttt{FortD}}$, the array and the loop match in size and we can parallelize the loop, assuming again no dependences. However, the loop index `i` and the array subscript `j` do not match; the compiler cannot apply loop-bounds reduction, but instead it has to apply the owner-computes rule at run-time with a guard. The core of the computation, which we assume

* Dynamic data distribution (redistribution at run-time) can result in considerable overhead. In this article, however, we address only those data distributions that may be interpreted statically.

† We use pseudo code in all code fragments in this correspondence. The Fortran D syntax to distribute an array requires the array to be aligned with a decomposition which in turn is distributed [4]; with HPF, the *align* directive is optional [2].

‡ We have also assumed that the loop bounds and array shapes are known to the compiler. This turns out not to be true in many cases of practical interest, further complicating efficient generation of node code.

```
PROGRAM BAD1FortD
REAL x(100)
DISTRIBUTE x(BLOCK)

do i = 1, 100
   j = ...
   x(j) = ...
ENDDO
```

```
PROGRAM BAD1Node
REAL x(25)

do i = 1, 100
   j = ...
   IF ((j-1)/25 .EQ. my$p) THEN
      x(mod(j-1,25)+1) = ...
   ENDIF
ENDDO
```

FIGURE 2 Example loop not matching the array access; the Fortran D program (left) will be compiled into a node program (right) with full loops and a guard (assuming $N_{proc} = 4$). The id of an individual processor is denoted by `my$p`, which ranges from 0 to $N_{proc} - 1$.

to be the assignment to `x(j)`, will still be executed in parallel, but scalability is likely to be limited due to the fully replicated loop iteration set.

In some cases, it will be difficult to establish a simple relationship between loops and subscripts, e.g., in irregular access patterns of distributed arrays. However, subscript-analysis complications are often avoidable artifacts of programming styles that obscure compiler analysis in general, not only in the data-parallel context. For example, consider the Fortran D fragments in Figure 3 (from now on we will omit the `FortD` subscript in the examples). `Bad2a` and `Bad2b` illustrate the popular practice of linearizing arrays, e.g., by storing a set of coordinate triplets into a one-dimensional (1-D) array. Such linearizations are used to eliminate a loop nest, facilitating vectorization of the resulting single loop [7]. For data-parallel programming, linearization often results in blurring the distinction between distributed and replicated subscript components. For example, in `Bad2a` or `Bad2b`, one would typically want to distribute `x` along the triplets, but keep each individual triplet on a single processor; i.e., the `i` loop should be parallelized, while the `d` loop should be replicated. This, however, is obscured to the compiler by the way the array is indexed, and, in `Bad2b`, by the artificial self-dependence in incrementing the counter `j`.

In the fairly clean and simple case of `Bad2a` and `Bad2b` one might still be able to teach a compiler to correctly apply loop-bounds reduction to the outer loop; in the general case, however, it is likely that the compiler will resort to replicating both loops and inserting guards similar to `Bad1Node`. It is much more desirable to clearly reflect the programmer's intent by splitting the subscript of `x` into the distributed index `i` and the replicated dimension index `d`, as shown in `Good2`. In general, a guideline for making the compiler's life easier is:

Arrays should not be linearized.

2 GUIDELINES APPLIED TO A MOLECULAR DYNAMICS PROGRAM

Before turning to our experience with parallelizing GROMOS [8] using the Fortran D compiler, we first give a brief introduction to the underlying application for the examples presented here; for more details, refer to the literature [9, 10]. Molecular dynamics (MD)

```
PROGRAM Bad2a
REAL x(300)
DISTRIBUTE x(BLOCK)

do i = 1, 100
   do d = 1, 3
      j = 3*i + d - 2
      x(j) = ...
   ENDDO
ENDDO
```

```
PROGRAM Bad2b
REAL x(300)
DISTRIBUTE X(BLOCK)

j = 0
do i = 1, 100
   do d = 1, 3
      j = j + 1
      x(j) = ...
   ENDDO
ENDDO
```

```
PROGRAM Good2
REAL x(3, 100)
DISTRIBUTE x(*, BLOCK)

do i = 1, 100
   do d = 1, 3
      x(d, i) = ...
   ENDDO
ENDDO
```

FIGURE 3 Fortran D programs with linearized (left) and delinearized (right) array accesses.

```
SUBROUTINE NBF_Lin_At()
INTEGER inb(Natom)
INTEGER jnb(MaxAllP)
DISTRIBUTE inb(BLOCK)
DISTRIBUTE jnb(BLOCK)

cnt = 0
DO i = 1, Natom
   DO p = 1, inb(i)
      cnt = cnt + 1
      j = jnb(cnt)
      ff = nbf_func(i,j)
      f(i) = f(i) + ff
      f(j) = f(j) - ff
   ENDDO
ENDDO
```

```
SUBROUTINE NBF_Delin_At()
INTEGER inb(Natom)
INTEGER jnb(MaxAtomP,Natom)
DISTRIBUTE inb(BLOCK)
DISTRIBUTE jnb(*,BLOCK)

DO i = 1, Natom
   DO p = 1, inb(i)
      j = jnb(p,i)
      ff = nbf_func(i,j)
      f(i) = f(i) + ff
      f(j) = f(j) - ff
   ENDDO
ENDDO
```

```
SUBROUTINE NBF_Delin_Chg()
INTEGER inb(Nchg)
INTEGER jnb(MaxChgP,Nchg)
DISTRIBUTE inb(BLOCK)
DISTRIBUTE jnb(*,BLOCK)

DO ii = 1, Nchg
   DO i = firstAt(ii), lastAt(ii)
      DO p = 1, inb(ii)
         jj = jnb(p,ii)
         DO j = firstAt(jj), lastAt(jj)
            ff = nbf_func(i,j)
            f(i) = f(i) + ff
            f(j) = f(j) - ff
         ENDDO
      ENDDO
   ENDDO
ENDDO
```

FIGURE 4 NBF kernel with linearized (left) and delinearized (middle) atom-based pair list, and with delinearized charge-group-based pair list (right). `Natom` is the number of atoms, `MaxAllP` the maximum total number of partners, and `MaxAtomP` the maximum number of partners per atom. `Nchg` is the number of charge groups and `MaxChgP` the maximum number of partners per charge group. `firstAt` and `lastAt` give the range of atoms for a charge group.

is a classical mechanics approach typically used to determine the motion of large molecular systems. At the core of the simulation, a force is calculated for each atom from the analytic derivative of a potential energy function. This force displaces the atom from its position in the previous time step. The MD program iterates over some number of time steps in the course of calculating an MD trajectory. Since each atom interacts with other atoms in some spatial neighborhood, dependences arise between atoms in so far as the potential energy function for each atom is evalauted with the positions of surrounding atoms from the previous time step. A pair list indicating which atoms interact with each other is computed every t steps, where t usually ranges from 10 to 50. Since there are typically tens to hundreds of interaction partners for each atom, the data structures representing the pair list tend to be the most space consuming in the program. Within a time step, the computation for each atom is independent from the computation for all other atoms and therefore is inherently parallel.* We base this report on the replicated approach [12], where we distribute the pair list data structures `inb` and `jnb` while replicating the other principal arrays, which includes the coordinate and velocity arrays, `x` and `v`, and the forces, `f`. For studies on more aggressive distributions refer to [9].

* Interatom dependences can arise for the integration step in simulations with constraints [11], but, in any case, the energy calculations are independent.

2.1 Delinearizing the Nonbonded Force Calculation

The nonbonded force (NBF) constitutes the main component of the MD computation performed by GROMOS and other MD programs. Since a force is computed for each atom, one natural implementation of the NBF algorithm loops over atoms and their partners, computes the force between them, and accumulates it according to Newton's third law, as shown in the (highly abstracted) Gromos subroutine `NBF_Lin_At()` in Figure 4. The underlying computation is inherently parallel; however, the compiler will fail to parallelize this loop nest. The reason is the loop-carried dependence on `cnt`, which is similar to the dependence on `j` in `Bad2b` (Fig. 3). Here, however, even advanced compiler analysis cannot identify a simple relationship between the array subscript `cnt` and the loop indices `i` and `p`. The problem is that in order to retrieve the list of partners for some atom `i`, one has to calculate the correct offset into `jnb`, which in turn depends on `inb(i')` for $1 \leq i' < i$; i.e., to determine the range of `j` for some iteration `i`, one has to iterate through all previous `jnb` segments. The advantage of linearizing `jnb` this way is space conservation; instead of having to reserve an equal amount of storage for each atom's pair list, we only need to reserve enough storage to accommodate the sum of partners. However, we can expect the storage savings in distributing `jnb` across processors to more than compensate for this increased space requirement, and to do so requires a delinearization as shown in `NBF_Delin_At()` in Figure 4. This will also allow parallelization, since now

```
SUBROUTINE Pairs_Delin_At()
INTEGER inb(Natom), jnb(MaxAtomP,Natom)
DISTRIBUTE inb(BLOCK), jnb(*,BLOCK)

DO ii = 1, Nchg
   DO jj = ii + 1, Nchg
      isPair(jj) = isPair_func(ii, jj)
   ENDDO

   DO i = firstAt(ii), lastAt(ii)
      cnt = 0
      DO jj = ii + 1, Nchg
         IF (isPair(jj)) THEN
            DO j = firstAt(jj), lastAt(jj)
               cnt = cnt + 1
               jnb(cnt,[i]) = j
            ENDDO
         ENDIF
      ENDDO
      inb([i]) = cnt
   ENDDO
ENDDO
```

```
SUBROUTINE Pairs_Delin_Chg()
INTEGER inb(Nchg), jnb(MaxChgP,Nchg)
DISTRIBUTE inb(BLOCK), jnb(*,BLOCK)

DO [ii] = 1, Nchg
   cnt = 0
   DO jj = ii + 1, Nchg
      IF (isPair_func(ii, jj)) THEN
         cnt = cnt + 1
         jnb(cnt,[ii]) = jj
      ENDIF
   ENDDO
   inb([ii]) = cnt
ENDDO
```

FIGURE 5 Pair list generation with delinearized atom-based pair list (left) and charge group-based pair list (right).

the distributed and replicated array dimensions are separated, and they directly correspond to the surrounding parallel and sequential loops.

2.2 Delinearizing the pair List Construction

The force calculation in `NBF_Delin_At()` now corresponds to the pattern in `Good2`, so, in itself, it can easily be parallelized. However, we must also consider the construction of the pair list; `Pairs_Delin_At()` in Figure 5 shows a simplified version of the corresponding GROMOS routine, with `jnb` delinearized according to `NBF_Delin_At()`. It turns out that the criterion for including pairs of atoms in the pair list actually depends on which charge group each atom belongs to, where a charge group is a collection of atoms treated collectively by the MD model. (Two atoms are considered "close" if their respective charge groups are "close.") `Paris_Delin_At()` determines interacting pairs by looping over charge group `ii`, determining for each charge group which other charge group `jj` is close to it, storing the charge-group partners of `ii` in an array `isPair`. Then looping over the atoms `i` of charge group `ii`, `inb(i)` and `jnb(1:inb(i),i)` are constructed accordingly. Distributed and replicated array dimensions are cleanly separated; however, we again have unmatched loop and data structures. The distributed dimensions of `inb` and `jnb` are both indexed by an atom index, whereas the enclosing loops iterate over charge group `(ii)` and atoms within each charge group `(i)`. The problem is that the granularity of the pair list computation is not the atom, but the charge group. We therefore switch to a charge group-based representation, as in `Pairs_Delin_Chg()`; this not only allows parallelization by loop-bounds reduction, but also preserves memory.* To finalize the data parallelization (at the level presented here), we now have to also modify the NBF calculation to use the charge group-based pair list. The result is `NBF_Delin_Chg()` shown in Figure 4.

3 DISCUSSION

We have stressed the importance of matching array and loop structures for data-parallel programming. However, there are many other issues influencing the quality of a compiler's analysis and the performance of the resulting code. We list, without further elabora-

* GROMOS already provides two versions of the pair list construction and corresponding NBF calculation, an atom-based version and a charge group-based one; however, both versions use linearized pair list representations.

tion, other guidelines which may be of particular significance to dusty decks.

Do not use distributed arrays as work space for other, nondistributed (or differently distributed) data.
Keep unrelated computations separate.
Keep array uses consistent across procedure boundaries.

As a general rule of thumb, one may say that programs that are hard to parallelize by hand will be even harder to parallelize for the compiler; data-parallel compilers provide only limited help with the high-level task of extracting exploitable parallelism from an application. However, that is not to say that a compiler can be of little use for parallelization. High-level data-parallel languages such as Fortran D and HPF remove from the programming task the tedious, error-prone, machine-specific, low-level work that has traditionally accompanied parallel computing. This note intends to help programmers harness that power to its fullest potential.

ACKNOWLEDGMENTS

Terry W. Clark was supported in part by the National Science Foundation grants ASC-921734 (including funds from DARPA) and MCB-9202918. The work of Reinhard von Hanxleden was supported by an IBM fellowship and by NASA/National Science Foundation grant ASC-9349459. Ken Kennedy was supported by NASA/National Science Foundation grant ASC-9349459.

REFERENCES

[1] G. C. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu, "Fortran D language specification," Department of Computer Science, Rice University, Tech. Rep. TR90-141, Dec. 1990. Revised April 1991.

[2] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, Jr., and M. Zosel, *The High Performance Fortran Handbook*. Boston, MA: The MIT Press, Cambridge University Press, 1994.

[3] E. W. Dijkstra, "Go to statement considered harmful," *Commun. ACM*, vol. 11, pp. 147–148, March 1968.

[4] D. Callahan and K. Kennedy, "Computing programs for distributed-memory multiprocessors," *J. Supercomput.*, vol. 2, pp. 151–169, Oct. 1988.

[5] R. v. Hanxleden, "Compiler support for machine-independent parallelization of irregular problems," PhD thesis, Rice University, 1994. Available via anonymous ftp from `softlib.rice.edu` as `pub/CRPC-TRs/reports/CRPC-TR94495-S`.

[6] C. Tseng, "An optimizing Fortran D compiler for MIMD distributed-memory machines," PhD thesis, Rice University, 1993.

[7] L. M. Liebrock and K. Kennedy, "Parallelization of linearized application in Fortran D," in *Proc. of the 8th Int. Parallel Processing Symp.*, 1994.

[8] W. F. van Gunsteren and H. J. C. Berendsen, "GROMOS: Groningen molecular simulation software," Laboratory of Physical Chemistry, University of Groningen, Nijenborgh, The Netherlands, Tech. Rep., 1988.

[9] T. W. Clark, R. v. Hanxleden, J. A. McCammon, and L. R. Scott, "Parallelizing molecular dynamics using spatial decomposition," in *Scalable High Performance Computing Conference*. New York: IEEE Computer Society, 1994, pp. 95–102. Available via anonymous ftp from `softlib.rice.edu` as `pub/CRPC-TRs/reports/CRPC-TR93356-S`.

[10] J. A. McCammon, "Computer-aided molecular design," *Science*, vol. 238, pp. 486–491, Oct. 1987.

[11] J.-P. Rycaert, G. Ciccotti, and H. J. C. Berendsen, "Numerical integration of the cartesian equations of motion of a system with constraints: Molecular dynamics of n-alkanes," *J. Computational Phys.*, vol. 23, pp. 327–341, 1977.