Experiences on Data-Parallel Programming

Terry W. Clark
Reinhard von Hanxleden
Ken Kennedy

CRPC-TR94495-S
December 1994

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

Experiences on Data-Parallel Programming

Terry W. Clark
tclark@kacha.chem.uh.edu
Dept. of Computer Science
University of Houston
Houston, TX 77204

Reinhard v. Hanxleden
reinhard@rice.edu
Dept. of Computer Science
Rice University
Houston, TX 77251

Ken Kennedy
ken@rice.edu
Dept. of Computer Science
Rice University
Houston, TX 77251

December 13, 1994

Abstract *To parallelize a scientific application with a data-parallel compiler requires certain structural properties in the source program, or conversely, the absence of others. We have encountered a number of difficulties in applying* Fortran D *to* Gromos*, a popular dusty-deck program for molecular dynamics, that probably are neither limited to* Gromos *nor do they seem likely to be addressed by improved compiler technology in the near future. This parallelization effort motivated this correspondence where we present some guidelines for engineering data-parallel applications that are compatible with* Fortran D *or* High Performance Fortran *compilers. 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.*

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. 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 [4] or High Performance Fortran (HPF) [7], 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 (e.g., see [3]). In this correspondence, 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 principles, 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 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 compiler 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 programmer should understand certain characteristics of data-parallelism, she or he should not have to develop a style which will have an adverse effect on code maintainability or efficiency in non-data-parallel environments.

Performance depends on the degree of parallelism and on overheads, such as communication and synchronization 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:

```
PROGRAM Good1FortD
REAL x(100)
DISTRIBUTE x(BLOCK)

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

```
PROGRAM Good1Node
REAL x(25)

do i = 1, 25
   x(i) = ...
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.

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

Distributing loop iterations is commonly driven by some heuristic for minimizing communication, such as the owner-computes rule [1] or variations thereof. This heuristic may fail even for incarnations of "embarrassingly parallel" algorithm 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 full success, realizing these points and designing program and data structures accordingly should already go a long way towards good performance for many applications.

The perhaps most important principle that should guide 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 prototype at Rice University.

Assume the compiler is given a simple code segment as shown in Figure 1 on the left, `Good1`$_{\texttt{FortD}}$. The FORTRAN D compiler will generate a node program `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, identified by `my$p`, and it may contain communication statements. Here we will focus on the distribution of computation; for a discussion of communication generation and other compilation issues we refer the reader to other publications [6, 10]. The compiler tries to parallelize the `i`-loop in `Good1`$_{\texttt{FortD}}$ by applying the owner-computes rule to the distributed array reference, `x(i)`. The owner-computes rule works fine here, assuming no sequentializing dependences on the rhs of the assignment, 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, and assuming that there are $N_{proc} = 4$ processors, each processor will perform only a quarter of the total number of iterations.

Now consider the program `BAD1`$_{\texttt{FortD}}$ in Figure 2. Similar to `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 arrays subscript `j` do not match; the compiler cannot apply loop bounds reduction, but instead has to apply the owner-computes rule at run-time with a guard. The core of the computation, which we assume 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.

```
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 $N_{proc} = 4$).

```
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.

In some cases, it will be difficult to establish a simple relationship between loops and subscripts, for example 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 two 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, for example by storing a set of coordinate triplets into a 1-D array. (Such linearizations are used to eliminate a loop nest, facilitating vectorization of the resulting single loop [8].) 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 still 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 $\texttt{Bad1}_{\texttt{Node}}$. It is much more desirable to clearly reflect the programmers intent by splitting the subscript of `x` into the distributed triplet index `i` and the replicated dimension index `d`, as shown in `Good2`. In general, a rule for making the compiler's life easier is:

Arrays should not be linearized.

Turning to our experience with parallelizing GROMOS using the FORTRAN D compiler, we first give a brief introduction into the underlying application for the examples presented here; for

```
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(Natom,MaxAtomP)
DISTRIBUTE inb(BLOCK)
DISTRIBUTE jnb(BLOCK,*)

DO i = 1, Natom
   DO p = 1, inb(i)
      j = jnb(i,p)
      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(Nchg,MaxChgP)
DISTRIBUTE inb(BLOCK)
DISTRIBUTE jnb(BLOCK,*)

DO ii = 1, Nchg
   DO i = firstAt(ii), lastAt(ii)
      DO p = 1, inb(ii)
         jj = jnb(ii, p)
         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.

more details, we refer the reader to the literature [5, 9]. Molecular dynamics (MD) 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 a molecular dynamics 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 evaluated 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 is typically between 10 and 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 inherently parallel. We base this report on the replicated approach, where we distribute the pairlist data structures, `inb` and `jnb`, while replicating the other principle arrays, which includes the coordinate and velocity arrays, `x` and `v`, and the forces, `f`. For reports on more aggressive distributions we refer the interested reader to the literature [2].

The non-bonded force (NBF) constitutes the main component of the molecular dynamics 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 `NBF_Lin_At()` in Figure 4, a (highly abstracted) subroutine in GROMOS. 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` (Figure 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.*, one has

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

DO ii = 1, Nchg
   DO jj = ii + 1, Nchg
      IF (isPair_func(ii, jj)) THEN
         isPair(jj) = .TRUE.
      ENDIF
   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(i, cnt) = j
            ENDDO
         ENDIF
      ENDDO
      inb(i) = cnt
   ENDDO
ENDDO
```

```
SUBROUTINE Pairs_Delin_Chg()
INTEGER inb(Nchg), jnb(Nchg,MaxChgP)
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(ii, cnt) = 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).

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 pairlist, we only need to reserve enough storage to accommodate the sum of partners. However, the storage savings in distributing `jnb` across processors would be much higher, and to do so requires a delinearization as shown in `NBF_Delin_At()` in Figure 4. This will also allow parallelization, since now the distributed and replicated array dimensions are separated, and they directly correspond to the surrounding parallel and sequential loops.

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 GROMOS routine, with `jnb` delinearized according to `NBF_Delin_At()`. It turns out that the criterion for including pairs of atoms in the pairlist 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.") `Pairs_Delin_At()` implements this by looping over charge groups `ii`, deciding for each charge group which other charge groups `jj` are close to it (and storing this in an array `isPair`), and then looping over the atoms `i` of charge group `ii` and constructing `inb(i)` and `jnb(i, 1:inb(i))` 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 atom index, whereas the enclosing loops iterate over charge groups (`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.[1] To finalize the data-

[1] GROMOS actually already provides two versions of the pair-list construction and corresponding NBF calculation,

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.

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 elaboration, a few points which may be of particular significance to dusty-decks.

> *Do not use distributed arrays as work space for other, non-distributed (or differently distributed) data.*
>
> *Try to 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. That is not to say that a compiler can be of little use for parallelization. What appears to have hampered a breakthrough-success of parallel computation so far is mostly the amount of tedious, error-prone, machine-specific, low-level work that usually comes along with it (name-space translation, communication generation, synchronization, etc.), not the high-level task of extracting exploitable parallelism. While data-parallel compilers promise only limited help in the latter, they should certainly be able to assist in the former. This note intends to help programmers harness this power to its fullest potential.

References

[1] D. Callahan and K. Kennedy. Compiling programs for distributed-memory multiprocessors. *Journal of Supercomputing*, 2:151–169, October 1988.

[2] T. W. Clark, R. v. Hanxleden, J. A. McCammon, and L. R. Scott. Parallelization using spatial decomposition for molecular dynamics. In *Scalable High Performance Computing Conference*, Knoxville, TN, May 1994. Available via anonymous ftp from `softlib.rice.edu` as `pub/CRPC-TRs/reports/CRPC-TR93356-S`.

[3] E. W. Dijkstra. Go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, March 1968.

[4] 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.

[5] 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.

[6] R. v. Hanxleden. *Compiler Support for Machine-Independent Parallelization of Irregular Problems.* PhD thesis, Rice University, December 1994. Available via anonymous ftp from `softlib.rice.edu` as `pub/CRPC-TRs/reports/CRPC-TR94494-S`.

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

[8] L.M. Liebrock and K. Kennedy. Parallelization of linearized application in Fortran D. In *Proceedings of the 8th International Parallel Processing Symposium*, New York, NY, April 1994.

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

[10] C. Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines.* PhD thesis, Rice University, January 1993.

an atom-based version and a charge-group-based one; However, both versions use linearized pair-list representations.