D NEWSLETTER #9

Handling Irregular Problems with Fortran D — A Preliminary Report

Reinhard v. Hanxleden

CRPC-TR93339-S
October, 1993

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

From the *Proceedings of the Fourth Workshop on Compilers for Parallel Computers*, Delft, The Netherlands, December 1993.

Handling Irregular Problems with Fortran D — A Preliminary Report[*†]

Reinhard von Hanxleden
reinhard@rice.edu

Center for Research on Parallel Computation
Department of Computer Science, Rice University
P.O. Box 1892, Houston, TX 77251

Abstract

Compiling irregular applications written in a data parallel, HPF-like language presents a challenging problem of growing importance. A project addressing this problem is the extension of the FORTRAN D compiler at Rice University to handle such codes. Generality and robustness have been major design objectives throughout this extension, allowing for arbitrary control flow and irregular accesses to multidimensional arrays. Even though this project is still in progress, it can already handle real-world codes fairly well, such as the non-bonded force calculation routine which is critical to molecular dynamics.

This paper is a first report on the experiences gained from extending the FORTRAN D compiler for irregular problems. Since the theoretical background underlying this project has already been described to some degree in previous publications, this paper focuses on the practical aspects of the implementation.

1 Introduction

Several research projects have aimed at providing a "machine independent parallel programming style," in which the applications programmer uses a dialect of sequential Fortran and annotates it with high level distribution information. Examples include High Performance Fortran (HPF) [Hig93], Vienna Fortran [CMZ92], and FORTRAN D [FHK+90]. A prototype FORTRAN D compiler targeting MIMD distributed memory machines has been under development at Rice University as part of the PARASCOPE programming environment [CCH+88]. For regular problems, *i.e.*, applications with relatively simple array subscript functions and fixed communication requirements, this compiler has had considerable successes [HKK+91, Tse93]. However, irregular problems, such as molecular dynamics or unstructured mesh codes, have proven to be significantly harder to parallelize [SH91]. Some of the obstacles encountered are:

- Lack of compile time knowledge about where and which data have to be communicated.
- Limited access locality.
- Large communication requirements.
- Poor load balance.

A number of strategies have been developed to address at least a subset of the difficulties mentioned above. For example, the *inspector-executor* paradigm allows to message-vectorize low-locality data accesses, even in the absence of compile time knowledge. First, an "inspection phase" determines what data have to be communicated within a certain loop and generates a communication schedule. Then, an "execution phase" (repeatedly) performs the actual computation and uses the communication schedule to exchange data [MSS+88, KMV90]. The PARTI communication library was designed to simplify schedule generation and schedule based communication [BS90]. PARTI has also been used to implement user-defined irregular distributions [MSS+88] and to provide a hashed cache for non-local values [MSMB90]. The feasibility of automatically generating inspectors/executors for simple loop nests has been demonstrated by the ARF [WSBH91] and KALI [KM91] compilers. *Run time iteration graphs*

[*]From the *Proceedings of the Fourth Workshop on Compilers for Parallel Computers*, Delft, The Netherlands, December 1993. D Newsletter #9, also available via anonymous ftp from softlib.rice.edu as pub/CRPC-TRs/reports/CRPC-TR93339-S.

[†]This work is supported by an IBM fellowship, and by the National Aeronautics and Space Administration/the National Science Foundation under grant #ASC-9349459.

can assist in improving load balance and access locality when distributing data across processors [PSC93]. *Slicing analysis* is a technique for generating inspectors and pre-inspectors for multiple levels of indirection [DSvH93]. The GIVE-N-TAKE framework is a generalization of partial redundancy elimination techniques that can be used for analyzing data access requirements and for efficient communication placement in the presence of arbitrary control flow [HK93]. Examples of SIMD specific work are the *Communication Compiler* [Dah90] for the Connection Machine and the *loop flattening* transformation to allow efficient execution of loop nests with varying inner loop bounds [HK92].

These projects have laid important ground work towards compiling irregular problems under a data parallel, machine independent programming paradigm. However, they tended to be limited either in their generality or in the degree to which they were implemented and practical difficulties were addressed. What appears to have been missing so far is a general recipe for transforming an HPF-like program that exhibits complicated array access and control flow patterns into an executable parallel program.

To overcome this deficiency, the FORTRAN D compiler prototype has been extended to efficiently handle the programming constructs typical to irregular applications. This is a joint project with the University of Maryland that is still under way; for example, slicing analysis is currently being implemented at UM, and support for irregular distributions is still being developed at Rice. However, major parts of it have been completed and valuable insights have been gained. In particular, communication generation, management of inspectors and executors, name space transformations, and the handling of variable-sized arrays have been implemented and proven their effectiveness with real world codes, including a non-bonded force routine taken directly from the GROMOS Molecular Dynamics program.

This paper is the first detailed account of the techniques and algorithms used in this prototype compiler. Most of the theory underlying the current implementation, like for example the GIVE-N-TAKE data flow framework used for communication generation, has already been described elsewhere. Therefore, this paper emphasizes the practical implementation issues of this project, pointing out technical aspects that may seem minor in theory but may have a significant impact on the overall design of a robust compiler. The rest of this paper is organized as follows. Section 2 introduces Molecular Dynamics as an example application that will be used to illustrate the general compilation process. Section 3 describes the analysis phase of the compiler, and Section 4 covers code generation. Section 5 concludes with a brief summary.

2 An example application: Molecular dynamics

Molecular dynamics (MD) simulations are computational approaches for studying various kinetic, thermodynamic, mechanistic, and structural properties of molecules [CM90]. There exist several MD packages (such as GROMOS, CHARMM, or ARGOS) that are heavily used to simulate biomolecular systems over a broad range of problem sizes and time scales. However, despite the relative maturity and growing importance of MD, it still presents computational challenges. These stem mainly from the complexity and irregularity of the underlying physical problem, and from various optimizations that speed up the simulations but further complicate the computation [CHK+92]. Consequently, parallelizing MD codes by hand is still an active field of research, and automatic parallelization within a data parallel framework presents an even bigger challenge.

Typical MD programs have a broad range of functionality, and each simulation consists of several different phases. However, most MD runs perform the bulk of the work (around 90%) in one routine, namely the non-bonded force (NBF) routine. Figure 1 shows an abstracted version of a sequential NBF calculation. This kernel presents the following problems:

Varying inner loop bounds: The NBF intensities decay very rapidly with increasing distance between the atoms involved. This locality is exploited by using a *cutoff radius*, R_{cut}, beyond which the NBF interactions are ignored. This reduces the number of atom pairs for which the NBF has to be computed, but due to the varying density of the simulated molecule, the number of neighbors within R_{cut} may vary from atom to atom.

Irregular off-processor accesses: One common mechanism for exploiting data parallelism is to attempt loop bounds reduction based on the owners of the data computed in each loop. In the NBF kernel, a compiler may try to reduce the bounds of the outermost loop based on the owners of j, *force*, $f(i)$, and $f(j)$. Even though scalars are replicated by default, j and *force* can be recognized as private and therefore will not prevent loop bounds reduction in any way. However, for a single iteration (i, p), a compiler using a strict owner computes rule would consider both the owner of $f(i)$ and the owner of $f(j)$ as participants. If these are different processors, then this iteration would be executed on both of them.

```
do $i = 1, N_{atoms}$
    do $p = 1, inb(i)$
        $j = partners(i, p)$
        $force = \mathrm{nbforce}(x(i), x(j))$
        $f(i) = f(i) + force$
        $f(j) = f(j) - force$
    enddo
enddo
```

Figure 1: Sequential version of the NBF kernel. N_{atoms} is the total number of atoms, $inb(i)$ is the number of atom *partners* that are close enough to atom i to be considered for the NBF calculation. The coordinate and force arrays, x and f, are shown only one dimensional.

In some cases this behavior might be desired, in particular if computation is cheaper than communication. In this case, however, an extra call to nbforce() is typically more expensive than communicating an extra datum. An alternative would be to first perform the computation on the owner of $f(i)$, and then, in case this processor does not own $f(j)$ as well, communicate *force* to that processor. This, however, would result in the communication of many small messages, which is typically more expensive then sending few, large messages. Therefore, if we are willing to assume that floating point addition is associative and commutative (an assumption that we are already at least partially relying on by parallelizing a code), we would rather first perform the assignment to each $f(j)$ locally, and then merge all non-owned $f(j)$ back to their owners at the end of the computation.

Making this kind of decision might be beyond the reach of current compiler technology. However, we would still like to have at least the option to guide the compiler with a language construct (such as the `on_home` directive) to express how the iteration space should be distributed.

Indirect array accesses: The subscript j is itself an array lookup. Consequently, $x(j)$ and $f(j)$ represent indirect array accesses. Since j is not known at compile time, this constitutes another reason to not apply the owner computes rule for the assignment to $f(j)$. Furthermore, it complicates message vectorization, *i.e.*, the combination of communicated data into few, large messages. There exist communication libraries to handle these cases effectively, as for example the PARTI communication routines. However, the compiler still has to first recognize the need for calling such libraries, then it has to generate all necessary parameters and call a suitable routine.

To illustrate how the compiler handles such a code in practice, we will use the Fortran D program `nbf` in Figure 2 as a running example. This program is a condensed and abstracted version of the 340-line GROMOS subroutine `nonbal.f` (without the Coulombic interactions), enhanced with some data initialization. Although this program is very simplified, it still presents similar difficulties as the original code with respect to the compiler. FORTRAN D directives (regularly) distribute pairlist, positions, and forces, and they override strict owner computes in the NBF calculation itself via an `on_home` directive. The program generated by the FORTRAN D compiler is shown in Figure 3. This code, linked together with the PARTI communication library and a memory allocation library, can be run on a message passing architecture after compiling with the machine's native compiler. The `nbf` program, annotated with output statements not shown here, has been compiled and run successfully on both a Sun workstation and, after feeding it through the FORTRAN D compiler, on an iPSC/860 hypercube.

3 Analysis phase

The FORTRAN D compiler can be divided into two phases: the analysis phase and the code generation phase. The *analysis phase* parses the program, builds internal data structures, and analyzes the program, but does not modify it yet. The *code generation phase* performs the actual code transformation by modifying the Abstract Syntax Tree (AST) and generates the final program by unparsing the AST. This implies that the same AST can be used for both source and target language. On the one hand, this strict separation might in some cases slightly increase intermediate storage and run time requirements. On the other hand, not touching the source code until achieving full knowledge about the transformation to be done lengthens the validity of intermediate analyses,

```
      program nbf

      integer i, j, p, natoms, maxp, timesteps
      parameter (natoms = 8000, maxp = 200, timesteps = 10)
      integer inb(natoms), partners(natoms, maxp)
      real force
      real x(natoms), f(natoms)

C     decomposition atomD(natoms), partnersD(natoms, maxp)
C     align inb, x, f with atomD
C     align partners with partnersD
C     distribute atomD(block)
C     distribute partnersD(block, :)

C     Generate some test data
      do i = 1, natoms
        x(i) = i
        inb(i) = maxp
        do p = 1, inb(i)
          partners(i, p) = mod(i + p - 1, natoms) + 1
        enddo
      enddo

C     Initialize forces
      do i = 1, natoms
        f(i) = 0
      enddo

C     Do the actual computation
      do t = 1, timesteps
C       execute (i) on_home f(i)
        do i = 1, natoms
          do p = 1, inb(i)
            j = partners(i, p)
            force = (x(i) - x(j))**(-6)
            f(i) = f(i) + force
            f(j) = f(j) - force
          enddo
        enddo
      enddo

      end
```

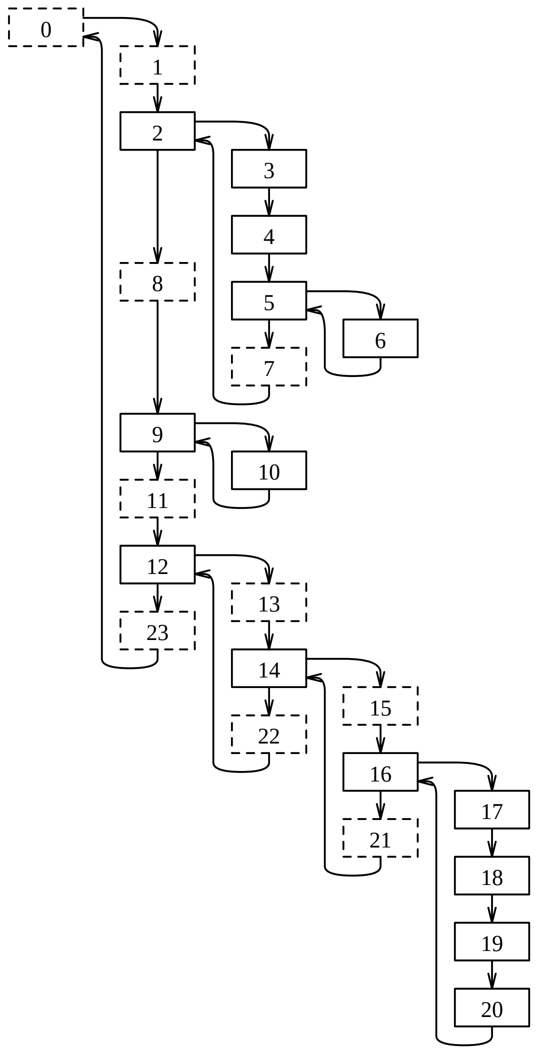

Figure 2: Fortran D version of **nbf** program and corresponding flow graph G. The loop nesting level in G increases from left to right. Node 0 is the root of G. Header nodes have their children attached to the right. Synthetic nodes, which do not directly correspond to statements in **nbf**, are dashed.

such as the control flow graph (CFG), static single assignment information (SSA), value numbers, and their links into the AST. This potentially decreases the need for reanalysis after intermittent code transformations, which in turn can speed up the overall compilation process. Most importantly, however, the clean separation enhances modularity of the compiler itself. This benefits the development and debugging process, and it allows easy integration into an interactive environment where the user wishes to make queries about a program without actually modifying it. The rest of this section discusses the analysis phase in more detail, followed by a description of the code generation phase in the next section.

3.1 Symbolic analysis

Even though the Fortran D compiler can be used in batch mode as a stand alone tool, it is part of the ParaScope programming environment. Therefore the compiler not only can be used within ParaScope, but it also takes advantage of the information and utilities provided by this environment. Besides basic facilities such as file I/O, lexing, parsing, and symbol table management, the environment also provides more advanced features, such as a framework for deriving interprocedural symbolic analysis. One simple example of the use of symbolic analysis in the **nbf** program is the proper handling of the symbolic constants **natoms** and **maxp** in array declarations, Fortran D directives, and loop bounds. Of particular importance with respect to the algorithms presented in this paper is the following information derived from a Fortran D program, P, that is handed to the compiler.

The control flow graph, $G := (N, E)$, with nodes N and edges E. Among other functions, G serves as a basis for the GIVE-N-TAKE data flow analysis framework and therefore has to meet certain criteria, such as reducibility and lack of critical edges (*i.e.*, edges from a node with multiple successors to a node with multiple predecessors). Furthermore, G has to be annotated with Tarjan interval information to guide the data flow phase. Figure 2 shows G for `nbf`; note that the current implementation generates one node for each statement instead of summarizing information for basic blocks.

Value numbers, $VN := \{\ vn \mid e$ an expression in P, $vn = val(e)$, the value number of $e.\}$. Value numbers are computed for both constant and non-constant expressions and are closely related to the SSA information [Hav93]. They provide for example information about whether an expression is an immediate or auxiliary induction variable or a linear combination thereof. Each array reference's value number is a pair $\langle state, (sub_1, \ldots, sub_{rank})\rangle$, where $state$ represents the internal state of the array that gets altered with every modification of the array, and sub_i is the value number of the i-th subscript. This means that the array is viewed as a scalar with respect to modifications, but the actual subscripts are taken into account when comparing array expressions.

3.2 The regular part of Fortran D compiler

Just as there are distinct analysis and code generation phases, there is also a fairly clean separation between those parts of the compiler that apply to both regular and irregular features of an application, and those parts that are specific to either kind. For brevity, the phases specific to irregular applications will be referred to as the "irregular compiler," whereas the rest of the FORTRAN D compiler that applies to all or just regular programs will be called the "regular compiler."

The irregular compiler performs its analysis after the regular compiler has performed interprocedural analysis, but before the regular compiler's intraprocedural analysis. An important interprocedural information provided by the regular compiler is reaching decomposition analysis, which propagates FORTRAN D specific decomposition information from callers to callees [HHKT92].

3.3 The data flow universe for communication analysis

In preparation for analyzing the communication requirements of P, the irregular compiler first determines IREFS, the set of both regular and irregular references to arrays that are accessed irregularly somewhere, and then computes KEYS, the data flow universe. Crucial to this phase is symbolic analysis, which can for example determine the equality of expressions even if they are syntactically different, across loops and other control flow constructs. Symbolic analysis is also used to determine whether a subscript is irregular or not; the current heuristic classifies all subscripts that are not linear combinations of induction variables as irregular.

More formally, the following information is computed.

$$
\begin{array}{rcl}
\text{IARRS} & := & \{x \mid x \text{ is referenced irregularly in } P.\} \\
\text{IREFS} & := & \{x(subs) \mid x \in \text{IARRS}, subs = (s_1, \ldots, s_r) \text{ a subscript list.}\} \\
\text{SUBS} & := & \{s \mid \exists x(\ldots, s, \ldots) \in \text{IREFS.}\} \\
\text{SUB\_VALS} & := & \{vn \mid vn \text{ is value number of some subscript } s \in \text{SUBS.}\} \\
\text{KEYS} & := & \{\langle st, vns, dist, iset\rangle \mid \text{ for some reference } x(subs) \in \text{IREFS, it is} \\
 & & \qquad st := \text{symbol table index of } x, \\
 & & \qquad vns := \text{value number of subscript list } subs, \\
 & & \qquad dist := \text{distribution of } x \text{ at point of reference}, \\
 & & \qquad iset := \text{computational distribution of reference } (e.g., \text{ an iteration set}). \\
 & & \} \\
\text{REFS\_KEY}(key) & := & \text{set of references that match } key \in \text{KEYS.} \\
\text{REFS\_KEYS} & := & \text{set of REFS\_KEYs.}
\end{array}
$$

Note that to allow irregular references to multidimensional arrays, SUB_VALS contains value numbers of individual subscripts, whereas KEYS considers value numbers of whole subscript lists. Furthermore, in order to determine when messages can be combined, or when buffered data become stale because of non-local assignments, etc., the data flow universe must be based on the comparison of actual memory locations, not values. Therefore, KEYS does not use value numbers for classifying data arrays, but instead their symbol table index. This is equivalent to syntactic comparisons plus aliasing and formal/actual parameter resolutions.

In `nbf`, the `j`'s are considered irregular subscripts, resulting in IARRS = $\{\mathtt{x}, \mathtt{f}\}$. Therefore, all references to `x` and `f` are collected to build the data-flow universe.

3.4 Communication analysis

The communication model used by the FORTRAN D compiler is that each datum, d, has an owner, $owner(d)$. If a processor p referencing d owns d, it is assumed to have a valid copy of d at the point of reference. If p does not own d, it might either have to receive d from $owner(d)$, which together with the corresponding send of $owner(d)$ is considered a global READ operation, or p might still have buffered a valid copy of d. An option not considered here is that p might receive d from any processor that has a valid copy of d [GS93].

Since the owner computes rule is not strictly applied, any processor p may define d. If $p \neq owner(d)$ and d will be referenced by some processor q, $q \neq p$, then p must send d to $owner(d)$ after defining d. That, together with the corresponding receive of $owner(d)$, is considered a global WRITE. Under these constraints, the objective of communication placement is to communicate as little and as infrequently as possible. Small messages are combined into larger ones, and data buffered locally (due to proceeding READs or local definitions) are reused as long as they are valid.

The data flow framework uses a binary lattice and represents its variables as bit vectors, one bit for each $key \in$ KEYS. The position within the bit vector constitutes the *id* for each *key*. In the following, the "*id* of *ref*" denotes the *id* assigned to the key matching a reference *ref*. Similarly, the definition, use, etc. of an *id* refers to the definition, use, etc. of a reference with the key corresponding to *id*.

The following information is computed for each node $n \in N$:

$$\begin{aligned}
\mathsf{REF}(n) &:= \{id \mid n \text{ references } id\},\\
\mathsf{DEF}(n) &:= \{id \mid n \text{ defines } id\},\\
\mathsf{ADD}(n) &:= \{id \mid n \text{ adds to } id\},\\
\mathsf{MULT}(n) &:= \{id \mid n \text{ multiplies to } id\},\\
\mathsf{RED}(n) &:= \mathsf{ADD}(n) \cup \mathsf{MULT}(n),\\
\mathsf{IND}(n) &:= \{id \mid n \text{ redefines the subscript dependence set of } id\}.
\end{aligned}$$

Here the *subscript dependence set* refers to the data that a subscript depends on. This can be thought of as the set of data that are referenced when inspecting for a subscript; a more detailed discussion can be found elsewhere [DSvH93]. For example, a subscript that is itself an indirection array lookup depends on the indirection array. In `nbf`, it is $\mathtt{x(j)} \in \mathsf{REF}(18)$[1], $\mathsf{IND}(6)$, and $\mathtt{f(j)} \in \mathsf{REF}(20)$, $\mathsf{DEF}(20)$, $\mathsf{ADD}(20)$, $\mathsf{RED}(20)$, $\mathsf{IND}(6)$; there are additional entries for the `x(i)`'s and `f(i)`'s.

Based on this local information, several instances of GIVE-N-TAKE are solved, one instance for each kind of communication. The communication types considered are global READs (or GATHERs), global WRITEs (SCATTERs), and global ADD and MULT reductions (SCATTER_ADD, SCATTER_MULT). The extension towards additional reduction types is straightforward. Separating SEND and RECV operations for each type of communication can expose opportunities for hiding communication latencies by overlapping them with computation. The GIVE-N-TAKE mechanism accommodates that by providing both an EAGER and a LAZY solution for all communication instances, where one solution indicates where to place the SENDs and the other computes where to place RECVs. However, the PARTI communication library generates SENDs and RECVs internally and presents them as monolithic entity; *i.e.*, a single PARTI call spawns SEND/RECV pairs.

Each GIVE-N-TAKE instance is initialized by assigning bit vector values to the variables TAKE_{init}, STEAL_{init}, and GIVE_{init} for each node $n \in N$. For example, the READ instance is initialized as follows.

$$\begin{aligned}
\text{READ}.\mathsf{TAKE}_{init}(n) &:= \mathsf{REF}(n) \setminus \mathsf{RED}(n), && \text{// READ what is referenced and not reduced}\\
\text{READ}.\mathsf{STEAL}_{init}(n) &:= \mathsf{IND}(n) \cup \mathsf{DEF}(n)^{\circ}, && \text{// Redefining data or indirection arrays blocks READs}\\
\text{READ}.\mathsf{GIVE}_{init}(n) &:= \mathsf{DEF}(n)^{\cap}. && \text{// READs come ``for free'' from local definitions}
\end{aligned}$$

Here $\mathsf{DEF}(n)^{\circ}$ and $\mathsf{DEF}(n)^{\cap}$ are the data that are either partially or fully enclosed by references in $\mathsf{DEF}(n)$ [HKK^{+}92]. For example, since `i` and `j` cannot proven to be disjoint subscript ranges in `nbf`, it is $\{\mathtt{x(i)}\}^{\circ} = \{\mathtt{x(i)}, \mathtt{x(j)}\}$.

[1]Read as: "The key associated with the reference `x(j)` is contained in the REF bitvector at node 18 which corresponds to the assignment to `force` in `nbf`."

After initialization, each instance is solved as either a FORWARD GIVE-N-TAKE problem, which places communication before computation (as needed for READs), or as a BACKWARD problem, which places communication after computation (WRITEs and reductions). The results of the data flow analysis reside for each node $n \in N$ in $\mathsf{RES}_{in}(n)$, which contains all references that have to be communicated on entry to n, and $\mathsf{RES}_{out}(n)$, on exit from n. The actual data flow equations, formal correctness and optimality criteria, etc., are described elsewhere [HK93].

A minor, but in practice interesting point is that in some cases communication might be placed at *synthetic nodes, i.e.*, nodes that correspond to not-yet existing basic blocks. For example, if there is an IF-THEN without a matching ELSE, then breaking critical edges generates a synthetic node corresponding to an empty ELSE branch. This may require creating new basic blocks during code generation. However, this can sometimes be avoided at no extra run time cost by shifting communication from synthetic nodes to other nodes. For example, if a loop is guarded by a condition, then synthetic nodes are introduced for the ELSE branch and between the loop and the merge after the condition; any subsequent READ that is blocked by the loop will be placed on both of these synthetic nodes instead of the merge after the guard. To take advantage of such opportunities for code simplification, an additional data flow phase post-processes the $\mathsf{RES}_{in/out}$ results into new variables, $\mathsf{GEN}_{in/out}$, that try to move results to non-synthetic nodes and to minimize the need for additional basic blocks.

Another interesting problem is that while global READs are shifted up and global WRITEs are shifted down, they should not be moved past each other if they communicate non-disjoint data. This can be satisfied by first solving the READ problem, and then initializing the WRITE problem as follows:

$$\begin{aligned}
\textsc{Write}.\mathsf{TAKE}_{init} &:= \mathsf{DEF} \setminus \mathsf{RED}, \\
\textsc{Write}.\mathsf{STEAL}_{init} &:= (\textsc{Read}.\mathsf{GEN}_{\mathrm{in}} \cup \mathsf{REF} \cup \mathsf{RED})^{\circ}, \\
\textsc{Write}.\mathsf{GIVE}_{init} &:= \emptyset.
\end{aligned}$$

As an example for a reduction initialization, we have

$$\begin{aligned}
\textsc{Add}.\mathsf{TAKE}_{init} &:= \mathsf{ADD}, \\
\textsc{Add}.\mathsf{STEAL}_{init} &:= (\textsc{Read}.\mathsf{GEN}_{\mathrm{in}} \cup (\mathsf{REF} \setminus \mathsf{ADD}))^{\circ}, \\
\textsc{Add}.\mathsf{GIVE}_{init} &:= \emptyset.
\end{aligned}$$

In `nbf`, it is for example `x(j)` $\in$ READ.$\{\mathsf{TAKE}_{init}(18),\ \mathsf{STEAL}_{init}(3),\ \mathsf{STEAL}_{init}(6),\ \mathsf{RES}_{in}^{eager}(8),\ \mathsf{GEN}_{in}^{eager}(9)\}$, and `f(j)` $\in$ ADD.$\{\mathsf{TAKE}_{init}(20),\ \mathsf{STEAL}_{init}(6),\ \mathsf{STEAL}_{init}(10),\ \mathsf{RES}_{in}^{eager}(23),\ \mathsf{GEN}_{in}^{eager}(12)\}$.

3.5 Inspectors

Inspectors have to be generated whenever runtime resolution is required for determining which data have to be communicated. In the current prototype, inspector generation and placement are sufficiently robust to handle simple cases like the NBF kernel, but overall they are still done in an ad hoc fashion. In particular, only one inspector is allowed per subscript value number, and inspectors are hoisted out of loops only as far as the communications are moved out; both assumptions might result in unnecessary re-inspections.

The set of inspectors is computed as follows. For each $vn \in$ SUB_VALS, let target(vn) be the least common ancestor of all nodes $n \in N$ at which communication involving a subscript with value number vn is generated. Let INSP(n) be the set of all subscripts s for which target(val(s)) $= n$. Let INSPS $:= \{(n, \mathrm{INSP}(n)) \mid \mathrm{INSP}(n) \neq \emptyset\}$; in `nbf`, it is INSPS $= \{(9, \{\texttt{j}\})\}$.

3.6 Executors

Executors are slightly modified versions of regions in P that contain irregular references. Part of the modifications is to replace irregular subscripts by references to *trace arrays* [DSvH93]. These arrays contain traces of the subscripts encountered during inspection, localized from global to local name space. Trace arrays are one of several possible options to perform the name space conversion. In the presence of high subscript reuse, for example, hash tables would be a more complicated but also more space efficient alternative.

The use of trace arrays requires insertion of counters for indexing. For generating code to initialize and increment counters and for eliminating duplicate counters, executors are collected as follows. For each $s \in$ SUBS, let limit(s) $= n \in N$ s.t. n corresponds to the beginning of the inspector for s. This node may for example be the header of the outermost loop enclosing s that will be copied into the inspector. Let EXEC(n) be the set of all subscripts s with limit(s) $= n$. Let EXECS $:= \{(n, \mathrm{EXEC}(n)) \mid \mathrm{EXEC}(n) \neq \emptyset\}$; in `nbf`, it is EXECS $= \{(14, \texttt{j}\text{'s in nodes 18 and 20})\}$.

4 Code generation phase

After the compiler has finished the analysis phase, the AST is modified to transform the FORTRAN D program into a message passing node program.

4.1 The regular compiler

An important transformation performed by the regular compiler is the conversion of global name space references to distributed arrays into local name space references to local arrays. In most cases, this can be achieved by *loop bounds reduction*, which is also a convenient and efficient way to exploit data parallelism. In the `nfb` code, this transformation is applied to all `i` loops, whose upper bound is reduced from `natoms` (= `8000`) to `2000`. This corresponds to using four processors, which is the default but can be overridden by initializing `n$proc` in the FORTRAN D input program. Note how the explicit use of induction variables in reduced loops may necessitate the need for adding a processor dependent offset. For example, in the initialization loop `off$0` (= `my$p*2000`) is added to `i` before using it to initialize `x(i)`.

An additional task currently performed by the regular compiler is the communication generation for regular subscripts (none such communication is needed in `nbf`). This process does not make use of the GIVE-N-TAKE analysis yet, one of the reasons being that GIVE-N-TAKE so far does not include dependence analysis. That alone could be changed relatively easily [HK93], but there are also other issues that deserve special attention when generating regular communications from the information provided by GIVE-N-TAKE. For example, message tags have to be generated for matching separate sends and receives, which might require the construction of SEND/RECV equivalence classes in case of complicated control flow. Furthermore, if the analysis indicates that data are sent together but received separately, data have to be grouped accordingly.

4.2 Inspectors

The inspector code has to perform the following tasks:

1. If the size of a subscript trace is not known at compile time, then a *counting slice* has to be generated and the trace array has to be allocated. A significant implication of this scenario is the need for dynamically allocatable memory. For example, in `nbf` the size of the pairlist storing all pairs of atoms within R_{cut} depends on the physical properties of the molecule to be simulated and is not known at compile time. Therefore, a counting slice is generated to compute `j$cnt` ($= \sum_{\mathtt{i=1}}^{2000}$ `inb(i)`) and to allocate trace arrays for both the global (`j$glob`) and local (`j$loc`) name space.

2. Collect subscript traces, in global name space.

3. Generate PARTI calls to `reglocalize()` for computing a communication schedule (`x$sched`), for counting the number of non-local elements (`x$offsize`), and for transforming the subscript trace from global to local name space.

4. Resize the data arrays subscripted irregularly (`x` and `f`) to accommodate the non-local data, as computed by `reglocalize()`.

Apart from generating the code slices for computing the subscript traces (which in the current prototype is only implemented for relatively simple cases without complicated internal control flow), most of the code generation is relatively straightforward. A few implementation details are given below.

- For each inspector INSP(n) $\in$ INSPS, let INSP_VNS be the set of value numbers of subscripts inspected. To avoid duplicate code and storage, only one subscript trace will be generated for each $vn \in$ INSP_VNS. For example, in `nbf` the subscript `j` has the same value number in `x(j)` and `f(j)`, and therefore only one trace will be collected.

- When merging the code for generating multiple traces (which is not an issue in `nbf`), transformations such as loop fusion are applied if possible.

- As with all variables generated by the compiler (indicated by a `$` as part of the name), the identifiers for schedule, subscript trace, etc. must not collide with earlier declarations. Furthermore, on the one hand they should resemble the variables they are related to in the original program, and on the other hand, they should be reused as much as possible. For example, in `nbf` the communication schedule `x$sched` and off-processor count `x$offsize` are used for both `x` and `f`.

- Another issue related to compiler generated variables is the use of trace indices and counters providing the trace size for localization. A variable holding the size of the trace is needed if the number of iterations of the slice is not known. An auxiliary induction variable for indexing the trace array is required in case the loop is nested or not in normal form. The slice for `j$loc` needs both, since the size of the pairlist is not known, and since the slice is a nested loop; `j$cnt` is used to serve both purposes.
- Comments are generated to delineate the generated inspectors and the subscripts they are inspecting (similarly for counting slices, executors, and buffer initializations). This kind of information is not only useful for making the generated output more readable, but also for later debugging support. Subscripts occurring more than once are annotated with a count in brackets.

4.3 Communication statements.

For each node $n \in N$, the GEN sets of the different GIVE-N-TAKE instances indicate the communication to be generated. Here we have to distinguish between communication to be prepended (indicated by GEN_{in} for a FORWARD problem and GEN_{out} for a BACKWARD problem) and communication to be appended (GEN_{out} and GEN_{in}, respectively). Furthermore, if we wish to generate separate SENDs and RECVs, then we have to consider both GEN^{eager} and GEN^{lazy}. For example, the SENDs to be prepended to n are given by the *communication set*

$$\text{PREPEND}_{send}(n) = \text{READ}.\mathsf{GEN}_{in}^{eager}(n) \cup \text{WRITE}.\mathsf{GEN}_{out}^{lazy}(n) \cup \text{ADD}.\mathsf{GEN}_{out}^{lazy}(n) \cup \text{MULT}.\mathsf{GEN}_{out}^{lazy}(n).$$

For each generated communication set, the data to be actually communicated are indicated by the *st* and *vn* components of the keys contained in the bit vector, where *st* indicates the data array and *vn* gives a range of subscripts to communicate. For irregular communications, *vn* is annotated with the name of the communication schedule to use. Since irregular references are communicated using the PARTI routines, SENDs and RECVs are not separated, and just the EAGER solution is used for placing communication.

In `nbf`, the generated communications are $\text{PREPEND}(9) = \text{READ}(\{\texttt{x(j)}\})$ and $\text{APPEND}(12) = \text{ADD}(\{\texttt{f(j)}\})$. This translates into an `fgather` of `x(j$loc(1:j$cnt))` before the force initialization and an `fscatter_add` of `f(j$loc(1:j$cnt))` after the executor.

An optimization that is not implemented yet but at least conceptionally fairly straightforward is to use *incremental schedules* for pruning messages in case at least some of the data covered by a reference are already locally available [DPSM91, HKK+92]. The information about what data are already available is stored for each node $n \in N$ in $\text{READ}.\mathsf{GIVEN}(n)$.

4.4 Reduction initialization

An issue specific to reduction communications (like ADD and MULT) is the need for initializing buffer space for non-local data (assigning `0` for ADD, `1` for MULT). The heuristic used for placing this initialization code for a reduction instance of GIVE-N-TAKE is as follows. Let the "local reduction" of a node $n \in N$ be the variables that are affected by a reduction operation in n itself or in one of the children of n (*i.e.*, in a statement in a loop headed by n). The local reduction is computed in the GIVE-N-TAKE variable $\mathsf{TAKE}(n)$. Let $\mathsf{TAKE}_{header}(n)$ be the local reduction of the header node of the loop directly enclosing n, if n is in a loop, let it be $\emptyset$ otherwise. The set of data for which initialization code should be prepended to n is then given by $\mathsf{TAKE}(n) \setminus \mathsf{TAKE}_{header}(n)$.

In the `nbf` example, it is $\texttt{f(j)} \in \text{ADD}.\mathsf{TAKE}(20)$. However, since `f(j)` is not "stolen" within the enclosing `p` loop, it is also $\texttt{f(j)} \in \text{ADD}.\mathsf{TAKE}_{header}(20) = \text{ADD}.\mathsf{TAKE}(16)$. Similarly, `f(j)` is in the local reduction sets of the headers of the enclosing `i` and `t` loops. The `t` loop is outermost, therefore `f(2001:2000+x$offsize)` is initialized there.

4.5 Executors

The major tasks when generating executors are the conversion of irregular subscripts to trace array lookups, and related to that the generation of counters (the *subsubscripts*) to index the trace arrays. The current strategy for generating subsubscripts is as follows. Pick an executor EXEC(n) from EXECS, and a subscript $s \in$ EXEC(n). If n is the header of a loop in normal form and s is enclosed by n directly, then use the induction variable of n as subsubscript. Otherwise, an explicit counter is needed. If there already exists a counter at n for the loop directly enclosing s, then use it as subsubscript, otherwise generate a new one to use as subsubscript. After determining the subsubscript, the value number of s determines which trace array to use, and the subscript can be converted. Note that we might need several counters for one executor, for example for different, imperfectly nested loops, or for references at different loop nesting levels. Note also that in order to allow the use of such indexing variables

```
      program nbf
      integer i, j, p, natoms, maxp, timesteps
      parameter (natoms = 8000, maxp = 200, timesteps = 10)
      integer inb(2000), partners(2000, maxp)
      real force

C     --<< Fortran D variable declarations >>--
      common /FortD/ n$p, my$p, my$pid
      integer n$p, my$p, numnodes, off$0

C     --<< Fortran D/irreg variable declarations >>--
      integer i$wrk(1000000)
      integer x$sched, x$offsize, j$cnt, i$, $init, $newsize,
     . i$type, ialloc, iresize, f$type, falloc, fresize,
     . j$glob$ind, j$glob$size, j$loc$ind, j$loc$size, x$ind,
     . x$size, f$ind, f$size
      parameter (i$type = 1, f$type = 2)
      real f$wrk(1000000)
C     --<< END Fortran D/irreg variable declarations >>--

C     --<< Fortran D initializations >>--
      call iputsize(1000000, i$wrk)
      call fputsize(1000000, f$wrk)
      x$size = 2000
      x$ind = ialloc(f$type, x$size) - 1
      f$size = 2000
      f$ind = ialloc(f$type, f$size) - 1
      n$p = numnodes()
      if (n$p .ne. 4) stop
      my$p = mynode()
      off$0 = my$p * 2000

C     Generate some test data
      do i = 1, 2000
        i$glo = i + off$0
        f$wrk(x$ind + i) = i$glo
        inb(i) = maxp
        do p = 1, inb(i)
          partners(i, p) = mod(i$glo + p - 1, natoms) + 1
        enddo
      enddo

C     --<< Inspector for [3*]j$loc(1:j$cnt) >>--
C     --<< Counting slice for j$cnt >>--
      j$cnt = 1
      do i = 1, 2000
        j$cnt = j$cnt + inb(i)
      enddo
      j$glob$size = j$cnt
      j$glob$ind = ialloc(i$type, j$glob$size) - 1
      j$loc$size = j$cnt
      j$loc$ind = ialloc(i$type, j$loc$size) - 1
C     --<< END Counting slice >>--
      j$cnt = 1
      do i = 1, 2000
        do p = 1, inb(i)
          j = partners(i, p)
          i$wrk(j$glob$ind + j$cnt) = j
          j$cnt = j$cnt + 1
        enddo
      enddo
      call reglocalize(1, 8000, x$sched, i$wrk(j$glob$ind + 1),
     . i$wrk(j$loc$ind + 1), j$cnt, x$offsize, 2000, 1)
      $newsize = 2000 + x$offsize
      x$ind = iresize(f$type, x$ind + 1, x$size, $newsize) - 1
      x$size = $newsize
      $newsize = 2000 + x$offsize
      f$ind = iresize(f$type, f$ind + 1, f$size, $newsize) - 1
      f$size = $newsize
C     --<< END Inspector >>--

      call fgather(f$wrk(x$ind + 2001), f$wrk(x$ind + 1),
     . x$sched)
C     Initialize forces
      do i = 1, 2000
        f$wrk(f$ind + i) = 0
      enddo

C     --<< fscatter_add initialize {f(j$loc(1:j$cnt))} >>--
      do $init = 2001, 2000 + x$offsize
        f$wrk(f$ind + $init) = 0
      enddo
C     Do the actual computation
      do t = 1, timesteps
C       --<< Executor for x(j), [2*]f(j) >>--
        i$ = 1
        do i = 1, 2000
          do p = 1, inb(i)
            j = partners(i, p)
            force = (f$wrk(x$ind + i) - f$wrk(x$ind +
     . i$wrk(j$loc$ind + i$))) ** (-6)
            f$wrk(f$ind + i) = f$wrk(f$ind + i) + force
            f$wrk(f$ind + i$wrk(j$loc$ind + i$)) =
     . f$wrk(f$ind + i$wrk(j$loc$ind + i$)) - force
            i$ = i$ + 1
          enddo
        enddo
      enddo
      call fscatter_add(f$wrk(f$ind + 2001), f$wrk(f$ind + 1),
     . x$sched)

      call free(i$type, j$glob$ind + 1, j$glob$size)
      call free(i$type, j$loc$ind + 1, j$loc$size)
      call free(f$type, x$ind + 1, x$size)
      call free(f$type, f$ind + 1, f$size)
      end
```

Figure 3: NBF kernel, output of FORTRAN D compiler.

within loop headers, they have to be initialized to `1` instead of `0`. This in turn prohibits incrementing them at the beginning of the loop body; instead they have to be incremented at the end, which complicates code generation especially for loops with internal control flow.

In `nbf`, the executor for `x(j)` and `f(j)` needs a counter, `i$`, because the references are nested two levels deep within the executor. The value number of `j` is the same throughout the executor and maps to the trace array `j$loc`. Consequently, `x(j)` and `f(j)` are converted to `x(j$loc(i$))` and `f(j$loc(i$))`, respectively.

4.6 Dynamically allocated arrays

Dynamic array allocation is handled by calls to external library routines, such as `ialloc()`, `iresize()`, and `free()`. These routines manage space provided by some large work arrays and provide offsets into them for each allocated array. In the current prototype, the work arrays are of fixed, predetermined size; this rather crude scheme could be replaced by a more advanced library that does not require fixed size work arrays. Another option not addressed here is to distinguish between a copying and a non-copying resizing operation.

In `nbf`, work arrays are the integer array `i$wrk` and the floating point array `f$wrk`. A reference like `x(j$loc(i$))` becomes `f$wrk(x$ind+i$wrk(j$loc$ind+i$))`; note how this explicit array arithmetic exposes the need for other classical optimizations, like common subexpression elimination or loop invariant code motion. A separate pass, which can also be used as a stand alone tool, converts all dynamic array references into work array references. It also introduces some additional bookkeeping code, for example to store the size of each dynamic array (the variables suffixed by `$size`). Separating explicit dynamic memory handling from the rest of code generation sometimes results in suboptimal code; for example, there are some redundant assignments to size variables if the same size is used for multiple arrays. This, however, is just one of several examples where the compiler relies on later, fairly well understood optimizations, in order to achieve the modularity necessary for efficient use of high level transformations. Another example is the assignment to `j`, which became dead due to subscript conversion.

5 Summary and conclusions

Compiling irregular applications in a data parallel framework presents a challenging problem of growing importance. Previous works have addressed some of the arising problems [WSBH91, KM91, CHK+92, HKK+92, DSvH93, HK93], but very little experience has been gained so far with the compilation of large codes with complicated data access patterns and arbitrary control flow. A major part of the FORTRAN D prototype compiler at Rice is devoted to handling such codes efficiently. This paper has given an overview of the strategies, algorithms, and data structures developed for handling the specific characteristics of compiling data parallel, irregular applications. It turned out that some seemingly simple requirements turned into interesting problems that had a significant impact on the overall design, such as for example the need for handling multidimensional arrays with irregular subscripts.

The current prototype is currently able to handle codes such as the molecular dynamics `nonbal.f` routine without major modifications, which can be considered a significant advance over other results reported so far. However, the project is still in progress; in particular, irregular distributions and general slicing are not finished yet. Even though there exist techniques for handling these problems at least to some degree, their implementation and application to general programs will probably require refinements and new strategies.

Acknowledgements

A significant implementation effort, about 15,000 lines of C++, has been involved in the particular project described in this paper. However, it still represents only a relatively small fraction of the PARASCOPE environment, which many people have contributed to. Special thanks to Seema Hiranandani and Chau-Wen Tseng, who developed the regular FORTRAN D compiler, and to Paul Havlak, whose symbolic analysis proved extremely valuable. They have also been helpful partners for discussing higher level problems, as have been other members of the group, in particular Ken Kennedy, Chuck Koelbel, Uli Kremer, and Nat McIntosh; Seema and Uli have also assisted with this paper.

As mentioned in the introduction, the irregular compiler project is joint work between Rice and the University of Maryland. Even though this paper has focussed on the practical aspects of the work done at Rice, the overall design has been heavily influenced by Raja Das and Joel Saltz from UM. Thanks to Jim Humphries, also at UM, for writing the dynamic memory allocator.

Finally, thanks to Terry Clark and Ridgway Scott from the University of Houston, who offered their broad expertise with molecular dynamics and pointed out many practical issues when compiling irregular real-world codes.

References

[BS90] H. Berryman and J. Saltz. A manual for PARTI runtime primitives. ICASE Interim Report 13, Institute for Computer Application in Science and Engineering, Hampton, VA, September 1990.

[CCH+88] D. Callahan, K. Cooper, R. Hood, K. Kennedy, and L. Torczon. ParaScope: A parallel programming environment. *The International Journal of Supercomputer Applications*, 2(4):84–99, Winter 1988.

[CHK+92] T. W. Clark, R. v. Hanxleden, K. Kennedy, C. Koelbel, and L. R. Scott. Evaluating parallel languages for molecular dynamics computations. In *Scalable High Performance Computing Conference*, Williamsburg, VA, April 1992. Available via anonymous ftp from `softlib.rice.edu` as `pub/CRPC-TRs/reports/CRPC-TR992202-S`.

[CM90] T. W. Clark and J. A. McCammon. Parallelization of a molecular dynamics non-bonded force algorithm for MIMD architectures. *Computers & Chemistry*, 14(3):219–224, 1990.

[CMZ92] B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. *Scientific Programming*, 1(1):31–50, Fall 1992.

[Dah90] D. Dahl. Mapping and compiled communication on the connection machine system. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[DPSM91] R. Das, R. Ponnusamy, J. Saltz, and D. Mavriplis. Distributed memory compiler methods for irregular problems — data copy reuse and runtime partitioning. ICASE Report 91-73, Institute for Computer Application in Science and Engineering, Hampton, VA, September 1991.

[DSvH93] R. Das, J. Saltz, and R. v. Hanxleden. Slicing analysis and indirect accesses to distributed arrays. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993. Available via anonymous ftp from `softlib.rice.edu` as `pub/CRPC-TRs/reports/CRPC-TR93319-S`.

[FHK+90] 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.

[GS93] M. Gupta and E. Schonberg. A framework for exploiting data availability to optimize communication. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[Hav93] P. Havlak. Construction of thinned gate single-assignment form. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[HHKT92] M. W. Hall, S. Hiranandani, K. Kennedy, and C. Tseng. Interprocedural compilation of Fortran D for MIMD distributed-memory machines. In *Proceedings of Supercomputing '92*, Minneapolis, MN, November 1992.

[Hig93] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, Center for Research on Parallel Computation, Rice University, Houston, TX, 1992 (revised Jan. 1993). To appear in *Scientific Programming*, July 1993.

[HK92] R. v. Hanxleden and K. 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, June 1992. Available via anonymous ftp from `softlib.rice.edu` as `pub/CRPC-TRs/reports/CRPC-TR92207-S`.

[HK93] R. v. Hanxleden and K. Kennedy. A code placement framework and its application to communication generation. Technical Report CRPC-TR93337-S, Center for Research on Parallel Computation, October 1993. Available via anonymous ftp from `softlib.rice.edu` as `pub/CRPC-TRs/reports/CRPC-TR93337-S`.

[HKK+91] 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.

[HKK+92] R. v. Hanxleden, K. Kennedy, C. Koelbel, R. Das, and J. Saltz. Compiler analysis for irregular problems in Fortran D. In *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, August 1992. Available via anonymous ftp from `softlib.rice.edu` as `pub/CRPC-TRs/reports/CRPC-TR92287-S`.

[KM91] C. Koelbel and P. Mehrotra. Programming data parallel algorithms on distributed memory machines using Kali. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

[KMV90] C. Koelbel, P. Mehrotra, and J. Van Rosendale. Supporting shared data structures on distributed memory machines. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Seattle, WA, March 1990.

[MSMB90] S. Mirchandaney, J. Saltz, P. Mehrotra, and H. Berryman. A scheme for supporting automatic data migration on multicomputers. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[MSS+88] R. Mirchandaney, J. Saltz, R. Smith, D. Nicol, and K. Crowley. Principles of runtime support for parallel processors. In *Proceedings of the Second International Conference on Supercomputing*, pages 140–152, St. Malo, France, July 1988.

[PSC93] R. Ponnusamy, J. Saltz, and A. Choudhary. Runtime compilation techniques for data partitioning and communication schedule reuse. In *Proceedings of Supercomputing '93*, Portland, OR, November 1993.

[SH91] J. P. Singh and J. L. Hennessy. An empirical investigation of the effectiveness and limitations of automatic parallelization. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, Tokyo, Japan, April 1991.

[Tse93] C. Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines.* PhD thesis, Rice University, January 1993.

[WSBH91] J. Wu, J. Saltz, H. Berryman, and S. Hiranandani. Distributed memory compiler design for sparse problems. ICASE Report 91-13, Institute for Computer Application in Science and Engineering, Hampton, VA, January 1991.