Compiling for Distributed-Memory Systems

HANS P. ZIMA, MEMBER, IEEE, AND BARBARA MARY CHAPMAN

Invited Paper

Distributed-memory systems are potentially scalable to a very large number of processors and promise to be powerful tools for solving large-scale scientific and engineering problems. However, these machines are currently difficult to program, since the user has to distribute the data across the processors and explicitly formulate the communication required by the program under the selected distribution. During the past years, language extensions of standard programming languages such as Fortran were developed that allow a concise formulation of data distribution, and new compilation methods were designed and implemented that allow the programming of such machines at this relatively high level. In this paper, we describe the current state of the art in compiling procedural languages (in particular, Fortran) for distributed-memory machines, analyze the limitations of these approaches, and outline future research.

I. INTRODUCTION

In recent years, distributed-memory multiprocessor systems (DMMP’s) (Intel’s hypercubes and the Paragon, the NCUBE, Meiko Computing Surface, etc.) have rapidly gained user acceptance. Other systems have been announced in recent months. These architectures are relatively inexpensive to build, and are potentially scalable to very large numbers of processors. Hence, their share of the market is likely to increase in the near future.

On a computer architecture where the memory is physically distributed among the processors, the time required to access a nonlocal datum may be an order of magnitude higher than the time taken to access locally stored data. This has important consequences for program performance. In particular, the management of data, with the twin goals of making data references, as on a shared memory machine, but require him or her to specify the distribution of the program’s data. This data distribution is then used to guide the process of restructuring the code into a single program multiple data (SPMD) program for execution on the target DMMP. The compiler analyzes the source code, translating global data references, as SUPERB [25],[73], Kali [41], and the MIMDizer [50]. In contrast to the current programming paradigm, these systems enable the user to write code using global data references, as on a shared memory machine, but require him or her to specify the distribution of the program’s data. This data distribution is then used to guide the process of restructuring the code into a single program multiple data (SPMD) program for execution on the target DMMP. The compiler analyzes the source code, translating global data references into local and nonlocal data references based on the distributions specified by the user. The nonlocal references are satisfied by inserting appropriate message-passing statements in the generated code. Finally, the communication is optimized where possible, in particular by combining messages and by sending data at the earliest possible point in time. In algorithms where some data references are made through a level of indirection (such as for unstructured mesh codes and sparse matrix solvers), some of the analysis has to be performed at run-time and the task of the compiler is to generate code to perform this analysis and set up the required communication.

This paper is devoted to compilation techniques for the source-to-source translation of programs in an ex-
REAL UNEW(1:N,1:N), U(1:N,1:N), F(1:N,1:N)
CALL INIT (U, F, N)
...
DO  J = 2, N-1
   DO  I = 2, N-1
      UNEW(I,J) = 0.25 * (F(I,J) + U(I-1, J) + U(I+1, J) + U(I, J-1) + U(I, J+1))
   ENDDO
ENDDO

Fig. 1. Sequential Jacobi relaxation code.

When this code is parallelized by hand, the programmer must distribute the program's work and data to the processors that will execute it. One of the common approaches to do so makes use of the regularity of most numerical computations. This is the so-called single-program multiple-data (SPMD) or data parallel model of computation. With this method, the data arrays in the original program are each partitioned and mapped to the processors. This is known as distributing the arrays. A processor is then thought of as owning the data assigned to it; these data elements are stored in its local memory. Now the work is distributed according to the data distribution: computations that define the data elements owned by a processor are performed by it—this is known as the owner-computes paradigm. The processors then execute essentially the same code in parallel, each on the data stored locally.

It is, however, unlikely that the code on one processor will run entirely without requiring data stored on another processor. Accesses to nonlocal data must be explicitly handled by the programmer, who has to insert communication constructs to send and receive data at the appropriate positions in the code. This is called message passing. The details of message passing can become surprisingly complex: buffers must be set up; and the programmer must take care to send data as early as possible, and in economical sizes. Several issues arise that do not have their counterpart in sequential programming: New types of errors, such as deadlock and livelock, must be avoided. The programmer must decide when it is advantageous to replicate computations across processors, rather than send data. Moreover, for code that is explicitly parallel, debugging is a serious problem.

A major characteristic of this style of programming is that the performance of the resulting code depends to a very large extent on the data distribution selected. It determines not only where computation will take place, but is also the main factor in deciding what communication is necessary. The total cost incurred when nonlocal data is accessed involves not only the actual time taken to send and receive data, but also the time delay when a processor must wait for nonlocal data, or for other processors to reach a certain position in the code. Note that the performance of a program can no longer be estimated solely by the amount of computation it comprises: extra computation is not necessarily costly, and the communication delay inherent in a particular data distribution could be prohibitive.
/* PROCESSOR STRUCTURE PROC(M,M) IS ASSUMED */

/* CODE FOR PROCESSOR (P1,P2) */

PARAMETER( M = ..., N = ... )
PARAMETER( LEN = (N+M-1)/M )

/* DECLARE LOCAL ARRAYS TOGETHER WITH OVERLAP AREA */
/* DATA OWNED LOCALLY IS U(1:LEN,1:LEN) */
/* AND SIMILARLY FOR UNEW AND F */

REAL U(0:LEN+1,0:LEN+1), UNEW(1:LEN,1:LEN), F(1:LEN,1:LEN)

CALL LOCALINIT(U,F,LEN)

/* SEND DATA TO OTHER PROCESSORS */

IF (P1.GT.P) SEND (U(1,1:LEN)) TO PROC(P1-1,P2)
IF (P1.LT.P) SEND (U(LEN,1:LEN)) TO PROC(P1+1,P2)
IF (P2.GT.P) SEND (U(1,LEN)) TO PROC(P1,P2-1)
IF (P2.LT.P) SEND (U(1,LEN)) TO PROC(P1,P2+1)

/* RECEIVE DATA FROM OTHER PROCESSORS, ASSIGN TO */
/* OVERLAP AREAS IN ARRAY U */

IF (P1.GT.P) RECEIVE U(0,1:LEN) FROM PROC(P1-1,P2)
IF (P1.LT.P) RECEIVE U(LEN+1,1:LEN) FROM PROC(P1+1,P2)
IF (P2.GT.P) RECEIVE U(1,LEN) FROM PROC(P1,P2-1)
IF (P2.LT.P) RECEIVE U(1,LEN) FROM PROC(P1,P2+1)

/* COMPUTE NEW VALUES ON LOCAL DATA */

DO I = 1, LEN
  DO J = 1, LEN
    UNEW(I,J) = 0.25 * (F(I,J) + U(I-1,J) + U(I+1,J) + U(I,J-1) + U(I,J+1))
  ENDDO
ENDDO

Fig. 2. Jacobi relaxation code parallelised manually.

The message-passing programming style requires that the communication statements be explicitly hard-coded into the program. But these statements are based upon the chosen data distribution, and as a result, the data distribution is also implicitly hard-coded. It will generally require a great deal of reprogramming if the user wants to try out different data distributions.

To illustrate this, we reproduce in Fig. 2 the above section of code, rewritten to run on a set of $M^2$ processors using message passing code of the kind described. We have simplified matters by assuming that the processors have been organized into a two-dimensional array PROC(M,M) and that the processor array elements may be addressed for the purpose of exchanging data items: normally, a structure of this kind would have to be set up by the user first, and references would have to be converted to those provided by the environment. Further, we assume that the array sizes are multiples of $M$. Optimization of communication has been performed insomuch as messages have been extracted from the loops and organized into vectors for sending and receiving. When communication and computation are overlapped, as could be done here by carefully arranging the order in which local data is updated, the resulting code is considerably longer.

In this version of the Jacobi relaxation, each processor has been assigned a square subblock of the original arrays. The programmer has declared local space of the appropriate size for each array on every processor. Array $U$ has been declared in such a way that space is reserved not only for the local array elements, but also for those that are used in local computations, but are actually owned by other processors. This extra space surrounding the local elements is known as the overlap area. The nonlocal elements must be received, and values from local boundaries must be sent to the processors that need them. Care is taken that the processors whose segments of $U$ are on the original grid boundaries do not attempt to read from or send to nonexistent processors.

III. BASIC MODEL AND TERMINOLOGY

The core of this paper discusses methods for the source-to-source translation from a machine-independent Fortran
language extension for DMMP’s, which we call Data Parallel Fortran (DPF), to Message Passing Fortran (MPF), a language that extends Fortran by a set of statements for explicit message passing. These two languages are introduced for the purpose of this paper only, and we will informally describe the components actually needed, based on Vienna Fortran [74],[14] in the case of DPF.

The central concepts of DPF are processors and distributions:

- DPF allows the explicit specification of processor arrays to define the set of (abstract) processors used to execute a program. This set will in the following be denoted by P. Any two processors in P communicate by exchanging messages. The model abstracts from the machine topology, such as grid, torus, or hypercube and does not reflect different processor “distances.”
- Distributions map arrays to nonempty sets of processors; they can be specified by declaration annotations.

DPF assumes that a program will be executed by a DMMP according to the SPMD programming model. This model requires that each participating processor execute the same program; parallelism is obtained by applying the constraints of the model to different parts of the data domain simultaneously. The generated code will store the local parts of arrays and the overlap areas locally and use message passing, optimized where possible, to exchange data. It will also map logical processor structures declared by the user to the physical processors that execute the program.

The basic message passing operations in MPF are SEND and RECEIVE. Assume below that the expi are expressions with respective values ai, the vi are variables, and p1, p2 are processors. If SEND exp1, exp2, expm TO p2 is executed in processor p1, then a message (a1, ..., am) is sent to p2; if RECEIVE v1, ..., vm FROM p1 is executed in processor p2, then the two statements are said to match and the transfer of the message is completed by performing the assignments v1 := a1, ..., vm := am.

Message passing accomplishes synchronization, since a message can be received only after it has been sent. If a SEND or RECEIVE statement is delayed until a matching RECEIVE or SEND is executed, we speak of blocking or synchronous, otherwise of nonblocking or asynchronous communication statements. In all of the following we assume an asynchronous SEND and a synchronous RECEIVE.

Furthermore, we assume that a sequence of messages sent from a processor p1 to a processor p2 arrives in the same order in which it was sent.

Since this paper will deal mainly with Fortran-based transformation systems, our model will be oriented toward a suitably restricted Fortran. In particular, we will not discuss COMMON and EQUIVALENCE, mainly because the unrestricted use of these features makes automatic parallelization virtually impossible. However, appropriate constraints can be defined under which these primitives can be handled successfully [73],[50],[14],[54]. In examples we will slightly modify the Fortran syntax to obtain a more structured notation.

### A. The Data Space of a Program

**Definition 1:** An index domain of rank (dimension) n is a set I that can be represented as a cross product $I^n = \prod_{i=1}^{n} D_i$, where $n \geq 1$ and for all $i$, $1 \leq i \leq n$, $D_i$ is of the form $D_i = [l_i : u_i]$, where $l_i \leq u_i$ and $[l_i : u_i]$ denotes the sequence of numbers $l_i$, $l_i + 1$, ..., $u_i$. $l_i$ and $u_i$ are respectively called the lower and upper bound of dimension i.

Let A denote a declared array. Then A is associated with an index domain $I^A$, and a set of elements, $E^A$. There is a one-to-one correspondence between $E^A$ and $I^A$; if $i \in I^A$ is associated with $e \in E^A$, then i is called the index of e. The elements of an array represent the memory locations associated with the array during program execution. Similarly, if R is a declared processor array, then $I^R$ denotes its index domain.

In the following, we consider the data space $A$ of all declared arrays that are accessible in a given scope. $A$ does not include formal parameters. The set of all elements associated with arrays in the data space is denoted by $E$.

Each $e \in E$ is associated with exactly one element of $A$.

The scalar objects declared in a program are, for simplicity, considered as specially declared arrays: we consider the set of scalar objects to be a subset, $E \subseteq A$, of the data space.

There is a one-to-one correspondence between the set of program variables and $E$, where a program variable is either a scalar variable or an array element name. Note that arrays are not considered to be variables, but are associated with a set of variables identifying their element names. For example, B, as declared in REAL B(4), is associated with the four variables B(1), B(2), B(3) and B(4).

We will need an additional class of objects, which are called private variables. Private variables are strictly local to a processor, used only in that processor, and can never cause communication. An example for a private variable is a processor-local buffer. Private variables of a processor p are allocated in the local address space of p, together with the variables that are owned by the processor.

During program execution, variables—and therefore elements of $E$—are bound to values as an effect of the execution of assignment and input statements. A state $\omega$ of $E$ is a (partial) function $\omega : E \rightarrow \Omega$, where $\Omega$ is a domain of values (such as integers, reals, and logicals). This concept of state—which is natural for sequential programs—can be easily generalized to parallel programs governed by the SPMD model, since the coherency of global data is always guaranteed by the constraints of the model.

Let $S$ denote a statement of the program. The execution of $S$ in a given state $\omega$ is called the instance of $S$ in state $\omega$, and denoted by $S(\omega)$. Whenever a definition involves $S$, we implicitly assume a corresponding definition for instances of $S$, if appropriate.

### B. Distributions

A distribution of an array maps each array element to one or more processors, which become the owners of the
element, and, in this capacity, store the element in their local memory. While the selection of a distribution does not affect the correctness of a program, it determines its load balance and the communication requirements. The amount of communication can be reduced if data objects are replicated.\textsuperscript{2} Hence in our model, distributions map an array index domain into the powerset of the processor index domain, rather than simply into the processor index domain. We model distributions by functions between the associated index domains.

Definition 2: Let $\mathcal{I}, \mathcal{J}$ denote two index domains. An index mapping from $\mathcal{I}$ to $\mathcal{J}$ is a total function $\xi : \mathcal{I} \rightarrow \mathcal{P}(\mathcal{J}) - \{\phi\}$, where $\mathcal{P}(\mathcal{J})$ denotes the power set of $\mathcal{J}$.

Definition 3: Let $A$ denote an array, and assume that $\mathcal{R}$ is a processor array. An index mapping $\xi_A^R$ from $\mathcal{I}^A$ to $\mathcal{I}^\mathcal{R}$ is called a distribution for $A$ with respect to $\mathcal{R}$.

A distribution $\xi_A^R$—which is defined on the index domains—induces an associated element-based distribution that maps elements of $A$ to processors in $\mathcal{R}$. Since there is no danger of ambiguity, we will use $\xi_A^R$ for both purposes.$^3$

Definition 4: Let $A \in \mathcal{A}$, and $\xi_A^R$ a distribution. $A$ is (totally) replicated if and only if (iff) $\xi_A^R(i) = \mathcal{R}$ for all $i \in \mathcal{I}^A$.

Definition 5: Local variables

1) Let the total function $\lambda : \mathcal{P} \rightarrow \mathcal{P}(\mathcal{E})$ be defined as follows: for each processor $p \in \mathcal{P}$, $\lambda(p) = \{ e \in \mathcal{E} \mid p \in \delta(e) \}$. $\lambda(p)$ is the set of local variables of $p$; these variables are said to be owned by $p$.

2) For each $A \in \mathcal{A}$, $\lambda_A^R : \mathcal{P} \rightarrow \mathcal{P}(\mathcal{E}^A)$ determines for each processor the set of elements of $A$ owned by the processor: for each $p \in \mathcal{P}$, $\lambda_A^R(p) := \lambda(p) \cap \mathcal{E}^A$.

C. Basic Distribution Functions

In this section, we illustrate some aspects of the annotation syntax, and explain block and cyclic distributions by a program fragment (Fig. 3).

- The PROCESSORS declaration introduces $R_1$ as a one-dimensional processor array with four elements.
- The annotations attached to the declarations specify a block distribution for $A$ and a cyclic distribution for $C$; both arrays are mapped to $R_1$. Array $B$ is replicated by default. BLOCK distributes an array dimension to a processor dimension in evenly sized portions. CYCLIC maps array elements in a round-robin fashion to the corresponding dimension of the processor array. The elision symbol "::" is provided to indicate that an array dimension is not distributed. For example, the annotation in the declaration of $D$ specifies a blockwise distribution of its columns.

- Array $A$ is partitioned into 4 blocks of length 3. For all $i.1 \leq i \leq 12, \delta_{R_1}^A(i) = \{i/4\}$. For example, processor $R_1(1)$ owns $A(1), A(2),$ and $A(3)$.
- Array $B$ is replicated: $\delta_{R_1}^B(i) = R_1$ for all $i$.
- For all $i.1 \leq i \leq 12, \delta_{R_1}^C(i) = \{MODULO(i - 1.4) + 1\}$. For example, processor $R_1(2)$ owns $C(2), C(6),$ and $C(10)$.
- The first dimension of $D$ is not distributed, while the second dimension is distributed by BLOCK, with a block length of 25. More precisely: For all $i$ and $j.1 \leq i \leq 100, \delta_{R_1}^D(i,j) = \{i/25\}$. For example, processor $R_1(3)$ owns the columns 51–75 of $D$.

One method for distributing the data in the Jacobi code, in which UNEW, $U$, and $F$ are all partitioned, is shown in Fig. 4.
D. General Block Distributions

An array \( A \) has a **general block distribution** if the set of elements of each processor \( p \) in the corresponding processor array is a segment, or rectilinear section [21], of \( A \) and, further, the segments associated with different processors are either disjoint or identical. More precisely:

**Definition 6 General Block Distributions** Let \( A \in \mathcal{A} \), with \( I^A = X_{n+1}^i D_i \) and \( D_i = \{ l_i : u_i \} \) for all \( i \). Then a distribution of \( A \) is called a **general block distribution** iff the following conditions both hold:

1. For each \( p \in P \), \( \lambda(p) = \lambda_i (l'_i : u'_i, \ldots, l'_n : u'_n) \) where \( l'_i \leq u'_i \leq u_i \) for \( i = 1, \ldots, n \), and \( \lambda_i(l' : u'_i, \ldots, l'_n : u'_n) \) specifies the set of all elements \( A(j_1, \ldots, j_n) \) with \( l'_i \leq j_i \leq u'_i \) for all \( i \).

2. \( \lambda(p) \cap \lambda(p') \neq \emptyset \) for \( p, p' \in P \Rightarrow \lambda(p) = \lambda(p') \).

If \( A \) has a general block distribution, then each segment \( A' = \lambda(p) \) for some \( p \) is called the **distribution segment** associated with \( A \) and \( p \).

Clearly, block distributions as discussed in Section III-C represent a special case of general block distributions. For each array \( A \) with a general block distribution, \( \{ \lambda(p) : p \in P \} \) is a partition (in the mathematical sense) of \( \mathcal{A} \). If all arrays in the data space are distributed in this way, then \( A \) can be represented as the union of two disjoint sets:

\[
A := A^+ \cup A^-
\]

where \( A^- := \{ A \in \mathcal{A} | \lambda(p) = \mathcal{E}^A \text{ for each } p \in P \} \), and \( A^+ := A - A^- \). \( A^- \) and \( A^+ \) are, respectively, called the set of **unpartitioned**, and the set of **partitioned** arrays. Each unpartitioned array is replicated. Clearly, \( Z \subseteq A^- \).

IV. **Basic Parallelization Strategy**

A. **Overview**

In this section we describe the basic features of automatic parallelization by specifying a source-to-source translation from DPF to MPF (see Section III). This presentation relies mainly upon the methods implemented in SUPERB and subsequently adopted by several other systems (cf. Section VII); some of the features were also proposed in [11]. We simplify the presentation by considering a proper subset of DPF here, which restricts distributions to **general block distributions** and **total replication**, and assumes all arrays to be distributed **statically**, in the sense that once an array has been bound to a distribution (in a declaration), this association remains invariant throughout the entire program. Formal arrays in procedures assume the distribution of the actual argument. Explicitly parallel constructs are not considered.

The compilation strategy enforces the **owner computes paradigm**. That is, we consider the **work distribution** to be derived automatically from the data distribution specified in the DPF program.

Some of these restrictions will be reconsidered in the later sections of this paper.

The translation from DPF to MPF will be described as a sequence of **phases**, each of which specifies a translation between two source language levels (cf. Fig. 4). We use the Jacobi code shown in the previous sections to illustrate some of the transitions involved.

Assume that initially a DPF source program \( Q^0 \) is given. We transform \( Q^0 \) into an MPF program \( Q^4 \) in four conceptually distinct consecutive phases:

1) **Phase 1 (Front End): DPF Source Program** \( Q^0 \rightarrow \)

**DPF Normalized Program** \( Q^1 \): The source program \( Q^0 \) is processed by a **Front End**, which performs the following three tasks: it transforms the source code into an **internal representation** suitable for further compiler manipulation; it performs the **initial analysis** of the program, and it **normalizes** the code. This analysis may have to be updated or recomputed during subsequent transformation phases.

Thus phase 1 includes syntactic and semantic analysis, control flow analysis, data flow analysis, data dependence analysis and the construction of the call graph. The call graph is annotated with information needed in phase 2. Further, some standard transformations such as constant propagation and dead code elimination may be applied at this stage.

Normalizations have the purpose of simplifying subsequent compiler operations. They include transformations to eliminate certain program constructs (such as statement functions, ENTRY statements or alternate RETURNS), to position specific kinds of statement at a prescribed place in the code (e.g., FORMAT or DATA statements), and to transform other statements so that they always appear in a standard form. The number of different kinds of statements affecting control flow may be considerably reduced by such transformations.

This method of presentation has been chosen in order to present the major steps as clearly as possible; in contrast, a real system will frequently use shortcuts.
Fig. 5. The structure of a parallelization system.

Typically, the normalization transformations performed include loop normalization, if conversion and subscript standardization.

We do not discuss the details of the front-end analysis and transformations in this paper: they are essentially the same as those employed for automatic vectorization and shared-memory parallelization, and as such have been discussed extensively in the literature [75]. The resulting program, $Q'$, is called the normalized program.

2) Phase 2 (Splitting): DPF Normalized Program $Q' \rightarrow$ (Host Program $Q^H$, Node Program $Q^N$): $Q'$ is split into a host program, $Q^H$, and a node program, $Q^N$. The host program will, after compilation, be executed on the host computer or a specially designated host node of the target system, as the host process. It performs global management tasks, such as requesting resources, loading code, and terminating program execution; it also handles all input/output (I/O). Hence the I/O and termination statements of the original program, and all control statements that these depend on, will remain in the resulting host program. Functions and subroutines that cause I/O also appear in the host program in a rudimentary form; both are converted to subroutine calls without arguments.

The node program, $Q^N$, which is still a sequential DPF program, contains the actual computation; input/output is represented by communication with the host program. All I/O statements are removed from the node program; reads require a SEND statement in the host and a RECEIVE in the node program; writes require a RECEIVE in the host and a SEND in the node program.

Program control flow is modeled in both codes: hence the node program may have to communicate with the host to provide values of loop bounds, the result of conditionals, or values required in I/O operations. Error or end-of-file speciﬁcations in I/O statements also inﬂuence control ﬂow, and require communication of values from the host to the node program. Optimizations are necessary to ensure that only those values are transmitted that are actually required by the recipient to perform its tasks in the correct order.

The host and node programs resulting from this phase are loosely coupled; all synchronization necessary between them at run time will be achieved through message passing.

We use a program excerpt that calls a subroutine performing the Jacobi iteration of the previous chapters to illustrate the workings of this code (Fig. 6). The sequential program as shown reads two input values; it then executes a loop $ITER$ times, which calls two subroutines, and writes a line of information. Neither of these subroutines performs I/O; however, subroutine RESIDU calls NORM, which writes two values.

The host program after splitting (Fig. 7) reads the input values and sends them to the node. Since the subsequent condition uses an input value, the value of the conditional does not need to be received from the node here, as is generally the case. If the compiler can be sure that the value of $ITER$ does not change before the do loop is entered, it could eliminate the receive that follows and its

5 Evolving systems will also have to permit individual node I/O to concurrent files.
PROCESSORS R(2,2)
REAL UNEW(1:N,1:N), U(1:N,1:N), F(1:N,1:N) DIST(BLOCK,BLOCK)
REAL RES(1:N,1:N) DIST(BLOCK,BLOCK)
...
READ (5,*) ITER, OMEGA
IF (ITER .EQ. 0) STOP
DO I=1,ITER
CALL JACOBI(OMEGA, F, U, UNEW, N)
CALL RESIDU(I, RNORM, RES, F, U, N)
RKONV = RNORM/RNORM0
WRITE(6,*), ' CONVERGENCE :', RKONV
RNORM0 = RNORM
ENDDO
END

Fig. 6. JACOBI relaxation: excerpt from main program before splitting

PROGRAM HOST
... READ (5,*) ITER, OMEGA
SEND ITER, OMEGA TO NODE
IF (ITER .GT. 0) GOTO 2
... RECEIVE ITER FROM NODE
DO I=1,ITER
CALL HOSTRESIDU
RECEIVE I, RKONV FROM NODE
WRITE(6,*), I, ' CONVERGENCE :', RKONV
ENDDO
2 STOP
END
C************************************************************************
C************************************************************************
SUBROUTINE HOSTRESIDU()
CALL HOSTNORM()
END
C************************************************************************
SUBROUTINE HOSTNORM()
INTEGER ITER
REAL RNORM
RECEIVE ITER, RNORM FROM NODE
WRITE(6,*), ' RESIDUUM ', RNORM
END

C************************************************************************
SUBROUTINE RESIDU(ITER,RNORM,RES,F,U,N)
CALL NORM(ITER, RNORM, RES, N)
... SEND ITER TO HOST
DO I=1,ITER
CALL JACOBI(OMEGA, F, U, UNEW, N)
CALL RESIDU(I, RNORM, RES, F, U, N)
RKONV = RNORM/RNORM0
SEND I, RKONV TO HOST
RNORM0 = RNORM
ENDDO
2 STOP
END
C************************************************************************
SUBROUTINE NORM(ITER, RNORM, RES, N)
RNORM = 0.0
DO J=2,N-1
DO I=2,N-1
RNORM = RNORM + RES(I,J)*RES(I,J)
ENDDO
ENDDO
RNORM = SQRT(RNORM)/FLOAT(N-1)
SEND ITER, RNORM TO HOST
END

Fig. 7. Host and node programs after splitting

The node program (Fig. 7) contains all computation, but is still sequential. It receives data from the host, and in the excerpt given, executes the loop ITER times, sending two values to the host in each iteration. We show only subroutine NORM here, which transfers output to the host.

corresponding send in the node program. The host loop invokes one of the subroutines: the host version has no arguments and performs no computation. It simply calls NORM to receive two values from the node subroutine and write them.

ZIMA AND CHAPMAN: COMPILING FOR DISTRIBUTED-MEMORY SYSTEMS
3) Phase 3 (Initial Parallelization): This phase transforms the node program $Q^2$ into the defining parallel program $Q^3$ by processing the data distribution specified in the original DPF program. It does so in two steps, which are referred to as masking and communication insertion. Masking enforces the owner computes paradigm by associating a boolean guard, called the mask, with each statement, in accordance with the ownership implied by the distributions. For all nonlocal data accesses, communication insertion generates communication statements that copy nonlocal data items to private variables of the processor.

According to the SPMD execution model, the compiler does not generate separate node programs for each processor. Instead, each will execute the same program, receiving its parameters and initial data from the host program.

The performance of code to which only these simple transformations are applied is likely to be very poor. It is the task of all subsequent program manipulations to convert this node program into a form that will run well on the target machine. Both the construction of this version of the program and the subsequent improvements are discussed in more detail below.

4) Phase 4 (Optimization and Target Code generation): Defining Parallel Program $Q^3$ to Optimized Target Program $Q^4$: In this phase, the defining parallel program is transformed into an optimized parallel MPF target program, $Q^4$. Communication and masking are improved: Communication statements are moved out of loops and combined to perform aggregate communication where possible; the strip mining of loops [75] across the processors can be achieved in many cases by propagating the information in masks to the loop bounds. A prerequisite for many optimizations is precise flow and data dependence information, as gathered in phase 1, and overlap analysis, which detects certain simple regular communication patterns and reorganizes communication based upon this information. Overlap analysis also helps determine the minimum amount of storage that must be reserved for each partitioned data array in the memory of a node processor.

B. Initial Parallelization

Initial parallelization is the third of the phases described above. Recall that a program has been analyzed and normalized before this phase begins, and that it has been split into a host and a node program. We study the steps required to convert the node program into a simple parallel program in Sections IV-B1 and IV-B2 below. First, however, we define some notation we will need below. Let $S$ denote a statement, $ref$ an array element, and $p$ a processor. Then

- The occurrences of variable names in the program text are classified as definitions or uses, depending on whether they are assigned to or their value is read. We denote the corresponding sets for a statement $S$ by $DEF(S)$ and $USE(S)$.
- $MY\_PROC$ returns the unique identifier of the executing processor.
- $OWNED(\text{ref}, p)$ is a boolean function that yields true if the array element denoted by ref is owned by $p$. The default for $p$ is $MY\_PROC$.
- $\text{MASTER}(\text{ref})$ returns a uniquely defined processor $p$, for which $OWNED(\text{ref}, p)$ holds. It is called the master of ref. The choice of $p$ is system dependent.

1) Masking: The first step in the initial parallelization of a program enforces the owner computes paradigm: Masking replaces each statement $S$ of the program by the associated masked statement

$$\text{IF} \ mask(S) \ \text{THEN} \ S \ \text{ENDIF}$$

where

- $mask(S)$ is $OWNED(A(x))$ if $S$ is an assignment statement of the form $A(x) = \cdots$, where $A$ is partitioned and $x$ is a list of subscript expressions
- $mask(S) \neq \text{true}$ in all other cases. Clearly, in this case the masked statement associated with $S$ can be immediately replaced by $S$.

Note that $mask(S) \neq \text{true}$ for all statements that are not assignments (in particular, control statements) and for all assignments to replicated variables.

Example 1: The masked version of the assignment statement in the Jacobi relaxation (Fig. 4) is:

$$\text{IF} \ OWNED(\text{NEW}(I,J)) \ \text{THEN} \ \text{UNEW}(I,J) = 0.25 \ast \{F(I,J) + \text{UNEW}(I-1,J) + \text{UNEW}(I+1,J) + \text{UNEW}(I,J-1) + \text{UNEW}(I,J+1)\} \ \text{ENDIF}$$

2) Communication Insertion: The intermediate program version after masking may contain nonlocal references, i.e., a statement instance to be executed on one processor may refer to data owned by another one. Whenever this occurs, the required values must be communicated by message passing. This is the next step of parallelization.

a) Communication primitive EXCHO: Processor communication in the defining parallel program $Q^3$ uses the exchange primitive of level 0, EXCHO, defined in Fig. 8. EXCHO generates exactly those message passing statements that are required to satisfy the nonlocal data references in a processor’s code. Each execution of a SEND ref TO $p'$ in a processor $p$, which is caused by a call to EXCHO($m, \text{ref}, \text{temp}$), corresponds to exactly one execution of RECEIVE temp FROM $p$ in $p'$, caused by a call to the same occurrence of EXCHO($m, \text{ref}, \text{temp}$), in the same state.

b) Transformation 1—Communication insertion: For every statement $S$ with $m = mask(S)$, and every ref $\in USE(S)$ of the form $A(x) = \cdots$, where $A$ is partitioned, insert CALL EXCHO($m, \text{ref}, \text{temp}$) before $S$. In $S$, replace ref by temp. EXCHO($m, \text{ref}, \text{temp}$) is said to be associated with $S$. The order of different EXCHO statements associated with a statement $S$ is irrelevant; however, their private variables must be distinct.

For the following, we require additional notation:

c) Definition 7: nonlocal USE sets: Let $S$ denote a statement, $p$ a processor, and ref $\in USE(S)$. Then $\text{NU}(S, \text{ref})$ denotes the set of all nonlocal uses caused by an access to ref when executing $S$ in any processor, and $\mu(p)$ the set of
/* EXCHO(m,ref,temp) */

Assume that $S$ is a statement, $m = \text{mask}(S)$, and $ref$ is in $USE(S)$ and of the form $A(x) = \ldots$, where $A$ is partitioned. Furthermore, let temp denote a private variable of the executing processor. $MVAL(m, p)$ yields true iff the value of $\text{mask}(S)$ in the current state is true for processor $p$.

The algorithm below specifies the effect of executing a call to $EXCHO(m, ref, temp)$. */

```
IF OWNED(ref)
    THEN temp := ref;
    IF (MY_PROC = MASTER(ref))
        THEN FOR EVERY $p$ SUCH THAT $MVAL(m, p) \land (ref \not\in p)$ SEND temp TO $p$
        ENDFOR
    ELSE
        IF ~OWNED(ref) \land $MVAL(m, MY_PROC)$
            THEN RECEIVE temp FROM MASTER(ref)
        ENDFL
    ENDFL
```

Fig. 8. The EXCHO primitive.

all nonlocal accesses in processor $p$. Furthermore, we define $\mu^A(p)$ as the subset of $\mu(p)$ that contains only elements of $A$.

Example 2: We insert communication statements in the Jacobi loop (Fig. 4). No exchange statement is generated for distributed and the subscript tuples associated with them are also identical in $S$, as the subset of $pA(p)$ is partitioned array are used in a processor. For many regular computations, the precise pattern of nonlocal accesses can be computed; this information can then be used both to determine the storage requirements for the array and to optimize communication.

1) Overlap Analysis: Overlap analysis is performed in the compiler to determine which nonlocal elements of a partitioned array are used in a processor. For many regular computations, the precise pattern of nonlocal accesses can be computed; this information can then be used both to determine the storage requirements for the array and to optimize communication.

For each partitioned array $A$ and processor $p \in P$, $\lambda^A(p)$ is the distribution segment of $A$ in processor $p$; it is allocated in the local memory associated with $p$. In the defining parallel program, communication is inserted each time an element of $\mu^A(p)$—the set of all nonlocal elements of $A$ accessed in $p$—is referenced, and private variables are created to hold copies of the original nonlocal values [cf. Section IV–B2]. Overlap analysis is used to allocate memory space for these nonlocal values in locations adjacent to the local distribution segment. More precisely, the overlap area, $OA(A, p)$, is the smallest rectangular contiguous area around the distribution segment of a processor, containing all nonlocal variables accessed. The union of the distribution segment and the overlap area is called the extension segment, $\Lambda^A(p)$, associated with $A$ and $p$. This description can be used to significantly improve the organization of communication; it facilitates memory allocation and the local addressing of arrays. The relevant analysis is described in detail in [25].

The overlap area for an array $A$ is specified by its overlap description, $OD(A)$, which is determined by the maximum...
PARAMETER (M=2,N=16)
PROCESSORS R(M,M)
REAL UNEW(1:N,1:N), U(1:N,1:N), F(1:N,1:N) DIST(BLOCK,BLOCK)
PRIVATE REAL TEMP1, TEMP2, TEMP3, TEMP4

DO J = 2, N-1
  DO I = 2, N-1
    C1: CALL EXCHO(OWNED(UNEW(I,J)),U(I-1,J),TEMP1)
    C2: CALL EXCHO(OWNED(UNEW(I,J)),U(I+1,J),TEMP2)
    C3: CALL EXCHO(OWNED(UNEW(I,J)),U(I,J-1),TEMP3)
    C4: CALL EXCHO(OWNED(UNEW(I,J)),U(I,J+1),TEMP4)
    IF OWNED(UNEW(1,J)) THEN
      S: UNEW(1,J) = 0.25 * (F(1,J) + TEMP1 + TEMP2 + TEMP3 + TEMP4)
    ENDIF
  ENDDO
ENDDO

Fig. 9. Jacobi relaxation code after initial parallelization.

- $OD(S,F(I,J)) = [0:0:0:0]$ 
- $OD(S,U(I-1,J)) = [1:0:0:0]$ 
- $OD(S,U(I+1,J)) = [0:1:0:0]$ 
- $OD(S,U(I,J-1)) = [0:0:1:0]$ 
- $OD(S,U(I,J+1)) = [0:0:0:1]$

Fig. 10. Overlap area for array $U$ in Jacobi code ($M \geq 1$)

**Example 3:** Consider now the Jacobi code (Fig. 9). The overlap descriptions associated with the different elements in USE(S) are computed as shown in Fig. 10. Thus $OD(F) = [0:0:0:0]$, and $OD(U) = [1:1:1:1]$. The segment of $U$ on $R(1,2)$ is $U(1:8,9:16)$. Its extension segment is given by $\Lambda^E(R(1,2)) = U(0:9,8:17)$, and the overlap area is $U(0,9:16) \cup U(9,8:16) \cup U(1,8:17)$. It consists of an area of depth 1 around the distribution segment.

Note that the overlap area may contain variables which do not correspond to elements of the original arrays. They will never be accessed, as they are not related to nonlocal uses.

We outline the computation of overlap descriptions and areas by considering the one-dimensional case.

Consider an assignment statement $S : B(y) = \cdots A(x) \cdots$ in a loop with loop variable $I$ and assume that $A,B \in A^+$ and that $x$ and $y$ are linear expressions in $I$.

1) Assume that $\delta^A = \delta^B$ and $x = c_1 * I + d_1, y = c_2 * I + d_2$, where $c_1 = c_2$, and $d_1$ and $d_2$ are constants. Let $d := d_1 - d_2$. If $d \geq 0$, then $OD(S,A(x))=[0:d]$; otherwise $OD(S,A(x))=[-d:0]$.

2) Assume now that $\delta^A = \delta^B$, $x$ and $y$ as above, but $c_1 \neq c_2$. Then, if a constant $d$ is obtained from the symbolic subtraction $x - y$, we can proceed as above; otherwise, we apply the test below.

3) The general test is applicable to arbitrary distributions $\delta^A, \delta^B$. It constructs a set of inequalities that
/* EXCH(ref,od)

Assume that \( S \) is a statement, \( ref \in USE(S) \), where \( ref \) is of the form \( A(\ldots) \) with \( A \in A^+ \), \( \od = OD(S,ref) \). The algorithm below specifies the effect of executing a call to \( EXCH(ref,od) \).

```
IF OWNED(ref) \&\& (MY_PROC = MASTER(ref))
THEN FOR EVERY \( p \) SUCH THAT \( ref \in OA(A,p) \) SEND ref TO p ENDFOR
ELSE
IF \( ref \in OA(A,MY_PROC) \)
THEN RECEIVE ref FROM MASTER(ref)
ENDIF
ENDIF
```

Fig. 11. The EXCH primitive.

```
DO J = 2, N-1
DO I = 2, N-1
   C1A: CALL EXCH(U(I-1,J),[O:1,O:O])
   C2A: CALL EXCH(U(I+1,J),[O:O,O:1])
   C3A: CALL EXCH(U(I,J-1),[O:O,1:O])
   C4A: CALL EXCH(U(I,J+1),[O:O,O:1])
   IF OWNED(UNEW(I,J)) THEN
      S: UNEW(I,J) = 0.25 * (F(I,J) + U(I-1,J) + U(I+1,J) + U(I,J-1) + U(I,J+1))
   ENDIF
ENDDO
ENDDO
```

Fig. 12. Program Jacobi with EXCH-based communication.

have to be satisfied by the loop variable for the instances of \( S \) associated with a process \( p \in P \). This is based on \( y \) and the bounds of the distribution segments \( \Lambda_B(p) \). For example, if \( y = c_2 * I + d_2 \) and \( \Lambda_B(p) = B(l_p : r_p) \) then \( l_p \leq c * I + d \leq r_p \) must be satisfied for each \( p \). The corresponding set of constraints for the loop variable can be determined by applying the Fourier-Motzkin method [43],[20] to these inequalities. From these constraints, \( \delta^A(p) \), and \( x \), the overlap description \( OD(S, A(x)) \) can be computed.

4) Whenever there is insufficient static information to determine the value of a component in the overlap description, a conservative estimate has to be made. This may mean that the remainder of an array dimension is included in the overlap area. In the worst case, we obtain an extension segment with \( \Lambda_B(p) = A \).

Hence the overlap concept is not an appropriate approach if there is any form of indirection in the code. In such a case, alternative methods must be employed. For a further discussion of this, see Section VI-B.

2) The Communication Primitive—EXCH: The overlap concept can be used to organize communication: the exchange statement \( EXCH0 \) is replaced by \( EXCH \), which refers to overlap descriptions \( OD(S,ref) \) rather than statement masks. \( EXCH \) is defined in Fig. 11.

3) Optimization of Communication I

\( a) \) Transformation 2—Communication insertion based on overlap: For every statement \( S \) and every \( ref \in USE(S) \) with \( ref \sim A(\ldots) \), \( A \in A^+ \), and \( od = OD(S,ref) \), insert \( CALL EXCH(ref,od) \) before \( S \). The order of different \( EXCH \) statements associated with the same statement \( S \) is irrelevant.

We replace transformation 1 by transformation 2. The statements in the new version of \( Jacobi \) are given in Fig. 12. The local declarations are omitted [see Section IV-CS].

4) Optimization of Communication II: The modified exchange statement forms the basis for a more advanced optimization of communication: we generalize \( EXCH \) to an aggregate communication primitive that moves blocks of data rather than single objects, and then discuss how to move communication out of loops and vectorize communication statements.

\( EXCH \) can be readily extended to move any rectilinear contiguous section \( \hat{A} \) of a segment \( \Lambda_B(p) \) by replacing \( ref \) in Fig. 11 with \( \hat{A} \), and taking care that more than one processor may be the source of a \( RECEIVE \). The communication for a data item can be moved out of a loop if no true dependence is violated. Loop distribution and vectorization [75] can then be applied to generate aggregate communication. Further optimization of communication includes fusion and elimination of redundant communication, as described in the literature [27],[25],[46]. A more general approach—independent of the overlap concept—is discussed in Section V-B.
Fig. 13. Jacobi—Final version: code for processor p.

5) Optimization of Masking: After initial masking [Sec- 

tion IV-B1], all processors execute the same masked state- 

dment sequence. For each masked statement, each processor 

first evaluates the mask and executes the corresponding 

following transformations optimize the handling of masks 

in loops and, in many cases, lead to the 

strip-mining [75] of the loop across the processors by partitioning the iteration 

space:

1) Iteration elimination: Iteration elimination deletes 

irrelevant statement instances by eliminating an entire 

loop iteration for a process.

2) Mask simplification: If it can be shown that a mask 

is always true for each instance of each process, it 

can be eliminated.

A more general discussion of mask optimization and the 

associated work distribution can be found in [28]. We again 

illustrate these optimizations as well as the target code 

generation with the example program.

Fig. 13 gives the final version of the Jacobi iter- 

ation. The program is parameterized in terms of the 

executing processor, p. It is assumed that \(\lambda^p\) = 

\(U(S_L1 (p) : S_U1 (p) , S_L2(p) : S_U2(p))\). Hence, for 

example, \(S_L2(R(1,2)) = 9\) and \(S_U2(R(1,2)) = 16\). The 

local declarations reserve space for the extension segments 

of UNEW and U. To simplify matters, we assume here that 

the target language can handle expressions in array bound 

declarations, so that global addressing is possible.

The execution of the first exchange statement, \(CALL \ EXCH(U(1:N-2,2:N-1),(I,0,0,J))\), in a processor p has the 

following effect: first, all elements of \(U(1:N-2,2:N-1)\) that 

are owned by p and belong to the corresponding overlap 

area of another processor \(p'\), are sent to \(p'\). Secondly, all 

elements of \(U(1:N-2,2:N-1)\) that are in \(OA(U,p)\) are 

received from the respective owner. For example, \(R(2,1)\) 

sends \(U(9,14,8)\) to \(R(2,2)\), and \(R(1,2)\) receives \(U(1,8,8)\) 

from \(R(1,1)\). These communication operations can be 

executed in parallel.

For each \(p\) the set of associated instances of \(S\) is given by 

\(\{S(I,J) \mid MAX(S_L1(p),2) \leq I \leq MIN(S_U1 (p), N - 1)\} \) 

and \(MAX(S_L2(p),2) \leq J \leq MIN(S_U2 (p))\). For these 

instances, the mask can be eliminated; all other iterations 

can be eliminated for \(p\).

D. Procedures

In this section, we extend our discussion to procedures 

and their formal scalar and array parameters. We apply the 

notation to describe local variables to formal procedure pa- 

rameters without further introduction. The \textit{SPMD paradigm} 

is applied to procedures in an obvious way: a procedure 
call in the node program is executed by all processors 

allocated to the program; masking and communication 

is performed as described above. We assume here that the 

distributions of formal parameters are \textit{inherited} from 

the corresponding argument distributions. The quality of 

the code generated for procedures then depends critically 

on how much information is available at compile time 

about the distributions of formal parameters in different 

incarnations of the procedure, and on how efficiently run- 
time information can be organized.\(^6\) In example 4, we show 

that this does not necessarily imply that formal parameters 

have a block distribution.

Example 4: Consider the program in Fig. 14. The 

processor array, \(R\), and \(U\) are declared in the same way 
as in the Jacobi program, and \(U\) has the same distribution 

(see Fig. 4).

The first subroutine call maps the two-dimensional array 

\(U\) to the one-dimensional array \(X\). The sequence associ- 
aption defined in Fortran means that \(U(i,j)\) is associated with 

\(X(i + (j - 1) * 16)\) for all \(i, j\) with \(1 \leq i, j \leq 16\). Now let 

\(p\) denote the processor \(R(1,1)\). Since \(\lambda^p = U(1 : 8, 1 : 

8)\), we obtain \(\lambda^p = X(1 : 8) \cup X(17 : 24) \cup \ldots \cup X(113 : 

120)\)—thus, \(\lambda^p\) is not a segment, but rather a union of 

disjoint segments. Similar observations can be made 

for the other processors. As a consequence, \(X\) does not 

have a block distribution.

\(^6\)This problem can be avoided if the language does not permit a 

formal parameter to be associated with differently distributed arguments 
in different incarnations, or if distributions of the formal parameters are 

enforced in the procedure.
In contrast to this, the Jth call to the second subroutine maps the Jth column of U, U(1 : 16, I), to Y; that is, for every k, 1 ≤ k ≤ 16, U(k, I) is associated with Y(k). The distribution of Y is thus as follows: if 1 ≤ i ≤ 8, then
- \( \lambda^Y(R(1, 1)) = Y(1 : 8), \lambda^Y(R(2, 1)) = Y(9 : 16), \lambda^Y(R(1, 2)) = \lambda^Y(R(2, 2)) = \phi \)

If 9 ≤ i ≤ 16, then
- \( \lambda^Y(R(1, 1)) = \lambda^Y(R(2, 1)) = \phi, \lambda^Y(R(1, 2)) = Y(1 : 8), \lambda^Y(R(2, 2)) = Y(9 : 16) \)

In both cases Y has a block distribution; note however, that some segments may be empty.

Formal arrays that do not have a block distribution cannot be handled as efficiently as those that do, since masks can no longer be efficiently implemented, and the related optimizations cannot be performed in general. Note, too, that different incarnations of the same procedure may associate the same formal parameter with arrays that have different distribution characteristics. The code implementing the accesses to a formal array parameter has to take into account all possible cases for the corresponding argument.

Gerndt [25] has formulated a sufficient condition (using the concept of a subarray) to guarantee that the formal parameter associated with a block-distributed actual argument is also distributed by block. This is used below to optimize the code generation for procedures.

We describe the main features of an interprocedural optimization strategy for the handling of partitioned arrays in procedures as implemented in SUPERB. More details are given in [26],[25]. We assume that a call graph, G, for the program has been constructed. G contains a node for each procedure of the program, and an edge from node n to node n' if the body of n contains a call to n'. Since we assume all procedures to be nonrecursive, G can always be made acyclic (possibly by applying node splitting transformations). The transformations described below can be performed during a single pass over G in invocation order; thus, whenever a procedure is processed, all procedures that may execute a call to it have already been handled.

The following steps are taken to process a procedure \( T \) in the pass over G:

1) **Attribute Specification**: For each formal parameter, \( f \), of \( T \), and each call site associated with \( T \), the argument, \( a \), with which \( f \) is associated is examined. If \( a \) is a nonformal object, then it is uniquely associated with the attribute "r" or "b," depending on whether \( a \) is replicated or (general) block distributed. If \( a \) is a formal parameter, then it has one or more of the attributes "r," "b," and "n," where "n" stands for non-block distribution, and the presence of an attribute indicates the kind of distribution that may be associated with \( a \). We now describe how the attributes of \( f \) are determined:
   a) If \( a \) has the attribute "r," then \( f \) is marked with "r."
   b) If \( a \) has the attribute "b" and the association of \( f \) with \( a \) is such that block distributions are maintained, then \( f \) is marked with "b," else \( f \) is marked with "n."
   c) If \( a \) has the attribute "n," then \( f \) is marked with "n."

2) **Cloning**: If \( f \) is marked by more than one attribute, then a separate copy of \( T \) is created for each attribute associated with \( f \), and the call graph as well as the call statements to \( T \) are correspondingly modified. This process is called cloning of \( T \). In each clone, the attributes for each formal parameter are unique. Since cloning may lead to an exponential growth of the program, it must be carefully controlled.

3) **Expansion of Scalars**: If \( f \) is scalar and is marked with either "b" or "n," then it is expanded into an array.

4) **Computation of Distribution Vectors**: Assume that \( T \) has formal parameters \( f_1, \ldots, f_m \). Then for each incarnation of \( T \), a distribution vector is computed. For each \( f_k, 1 \leq k \leq m \), if \( f_k \) is replicated or has a block distribution, the distribution vector specifies the distribution to which \( f_k \) is bound in the given incarnation. If it has a nonblock distribution, the distribution of the corresponding actual array is determined.

7Note that more than one case may apply.
V. OPTIMIZATION TECHNIQUES

The efficiency of the code generated according to the strategy described in the preceding section depends critically on the optimizations performed either as part of that strategy or in addition to it. We discuss below two areas in which techniques have been developed to optimize code for distributed memory systems. One of these aims to modify the code in such a way that more efficient parallel code can be generated by applying valid loop transformations; the other is directed toward improving the performance behavior of the transformed code by optimizing the communication. We describe the former of these only briefly below, and then discuss a major contribution to the latter area in Section V-B.

A very useful optimization technique that cannot be discussed in general here is the application of pattern matching; the detection of frequent code patterns, including stencils, reduction operations and linear algebra kernels, may be paired with knowledge of their efficient implementation on the target machine. Together with suitable transformations, which may make use of library routines, (locally) close to optimal target code can be generated. The usefulness of pattern matching techniques paired with library routines for the target machine is exemplified by the work described below in Section V-B.

A. Loop Transformations

A number of loop transformations have been developed to parallelize code for execution on shared memory computers [53]. Many of these transformation can be equally fruitfully applied to the task of parallelizing code for distributed memory machines. Precise conditions for their valid application are known and have been described in the literature [75]. Their application will depend on the manner in which the program data has been distributed. Particularly useful in this context are loop interchange and loop distribution. Loop interchange exchanges two levels of a nested loop. Loop distribution distributes the loop control over the statements in the loop body. These transformations may be used to move the position of communication statements, to generate vector statements or to remove masks.

However, the application of these transformations must be exercised with due caution. If loop distribution is applied to a loop with communication constructs in an attempt to reduce the amount of synchronization involved, by moving the SEND and RECEIVE statements individually so that data is received as early as possible, additional analysis must be performed to ensure that no deadlock can arise. The techniques needed to perform this potentially very useful transformation have been implemented in SUPERB and are described in [30].

Further transformations that may be applied to optimize the generated code include loop fusion, loop unrolling, strip mining of loops, scalar forward substitution, induction variable substitution, reduction operation recognition and, sometimes, in-line expansion.

B. A Pattern Matching Approach to Optimization of Communication

The previous section discussed the optimization of communication under the constraints of block distributions (Sections IV-C3 and IV-C4). Li and Chen [45] have examined communication under more general conditions. Their approach is based on syntactic pattern matching: it identifies communication patterns in the program and matches them to communication routines of the target machine, taking into account cost functions describing the execution time of these routines. We outline the basic features of their work.

1) Communication Patterns: Let a processor array, $R$, with index domain $I^R$ be given. Let $i$ denote an index vector, $i \in I^R$, an array element, $a$ a predicate over $I^R$, and $p(i)$, $p'(i)$ processor expressions that depend on $i$. Then the quoted symbolic form $I'$

$$\Gamma : a \times p(i) \rightarrow p'(i) \mid I^R$$

is called a communication pattern over $R$. It describes the set of all data movements that send $a$ from processor $p(i)$ to processor $p'(i)$, where $i$ ranges over $\Gamma := \{ i \in I^R \mid a(i) \}$. This set is denoted by $\Gamma$. For each assignment statement in a loop, the associated communication pattern can be readily constructed by taking into account the distributions of the arrays involved. Consider, for example, the Jacobi code with the array distributions as given in Fig. 4; assuming that $\Gamma$ denotes integer division, the access to $U(IJ+1)$ in the assignment to $UNEW(IJ)$ generates the communication pattern

$$U(IJ+1) \times R(IJ+1) \rightarrow R(IJ+1) \mid 2 \leq I, J \leq N - 1$$

2) Communication Routines: Basic communication routines are defined by selecting a set of communication patterns over $R$ that satisfy two criteria: first, they must have an efficient implementation on the target machine; second, the associated communication patterns must be uniquely recognizable by the compiler. Each of these routines is associated with a cost function, parameterized by the message size and the number of processors, that estimates its execution time. Li and Chen distinguish between general routines and simple routines.

For general routines, the data movement may cross multiple dimensions of $R$. Examples for such routines are: one-to-all broadcast, all-to-one reduce, single-send-receive, uniform shift, and an affine transform.

Simple routines are restricted in that data movement must remain within one dimension of $R$. Otherwise, they essentially correspond to the general routines.

For example, a one-to-all broadcast of $a$ (owned by processor $p_0$) can be characterized by the communication pattern "$a \times p_0 \rightarrow i$ | true" and a single-send-receive by "$a \times p_0 \rightarrow p_1$ | true".

The notation can be easily extended such that $a$ denotes a set of array elements rather than a single element. In the current implementation, the predicate is not used.
A. Overview

Further, let $C$ denote a communication routine, and $\Gamma_C$ the associated communication pattern. Then we say that $C$ matches $\Gamma$ for a certain selection of $C$'s parameters iff $\Gamma_{ref} \subseteq \Gamma_C$. $\Gamma$ and $\Gamma_{ref}$ are perfectly matched iff $\Gamma_{ref} = \Gamma_C$. A communication pattern in which the two processor expressions differ only in exactly one dimension is said to be simple, otherwise it is general.

We outline the matching algorithm in Fig. 15. Some simple optimizations are performed to eliminate duplicate communications. If a series of communication primitives are invoked to exchange data, their order in the code is optimized. A detailed description can be found in [46], see also Li's Ph.D. thesis [44].

VI. ADVANCED PARALLELIZATION TECHNIQUES

A. Overview

Most of the compilation techniques discussed so far are particularly useful for a language that permits the specification of general block distributions. However, despite the fact that this will suffice to successfully deal with a range of programs, there are other important distributions, and other language constructs that can help to improve the efficiency of program execution on parallel computers.

This section reconsiders some of the restrictions we have imposed and discusses some further language features and their implementation; in particular, we discuss parallel loops in Section VI-B, and handling arbitrary irregular distributions in Section VI-B4). We combine the discussion of these features with the introduction of an alternative approach to dealing with indirect data accesses that involves run-time analysis.

For a more general discussion of the features implemented in other systems, see Section VII.

• Processor Arrays: In our model we assumed implicitly that processor array declarations have constant bounds. This forces the user to recompile the program whenever it has to be run on a different machine configuration; on the other hand, it allows target code optimization to be tailored to individual processors. A more general approach parameterizes the program with the processor structure that will be associated with the program at the time of its actual execution: the compiler must parameterize all distributions and data access functions. This approach has been implemented in Kali [48],[40].

• Distributions: Cyclic and block-cyclic distributions are sometimes useful for achieving a good load balance. They do not fit well into the optimization patterns related to the overlap concept, however, and may incur a good deal of run-time overhead, since the computations required to determine the owner of an array element are very costly. Such distributions have been implemented in a number of systems, including Kali [48], DINO [60], and the MIMDizer [50]. Some languages propose user-defined distribution functions that allow the specification of arbitrary distributions [48],[23],[74]. In Kali, a restricted set of such functions has been implemented. Frequently, partial rather than total replication is advantageous. This has been implemented in many systems.

• Dynamic Distributions The term dynamic data distribution refers to the possibility of changing the distribution associated with an array at run time. In a few systems, this has been implemented within narrow constraints [4],[3]. Some recent languages propose very general dynamic distribution features [23],[74],[14]; however, these have not yet been fully implemented.

• Owner Computes Paradigm: The owner computes paradigm simplifies code generation considerably. However, it does not provide an adequate solution in situations where the right-hand side of assignment statements contains complex expressions whose components must be communicated to the owner of the left-hand side. Methods of relaxing this constraint, by allowing an explicit specification of where work is to
be performed, have been proposed and implemented in Kali [48]; they will be discussed in Section VI-B below.

- **Formal Procedure Parameters** In some cases, the rule of inheritance with respect to the distributions of formal procedure parameters (Section IV-D) is not appropriate. For example, it may be preferable for a computationally intensive routine to deal with arguments that have a fixed distribution in accordance with the algorithm employed. As a consequence, languages have been defined that allow more generality for formal procedure parameters, and in particular, the explicit enforcement of a distribution. This may lead to additional communication at procedure entry and exit, but can significantly improve the efficiency of data accesses in the procedure body [47],[74].

### B. Explicitly Parallel Loops and Run-Time Analysis

Several different kinds of explicitly parallel loop have hitherto been proposed. Their semantics differ; however, they do share the common purpose of giving the program-tions. We base our treatment of this topic on work by Saltz, Mehrotra, and Koelbel, in particular on Koelbel's Ph.D. thesis.

In the simple variant used here, they can be specified not conform to the owner computes rule. Reference; where

```
FORALL loop-control on-clause
  executable-statement-sequence
END FORALL
```

where `loop-control` has the same syntax as in a Fortran do-loop, and the `on-clause` has the form ON `processor-reference`; it determines for each iteration a unique processor that executes that iteration. The forall loop thus does not conform to the owner computes rule.

By definition, there are no loop-carried dependences in a forall loop; this means that a variable defined in one iteration of the loop is not defined or used in any other iteration; as a consequence, all iterations can be executed in parallel. This concept can be easily extended to allow arbitrary perfectly nested loops.

The `on-clause` in the example shown in Fig. 16 specifies that the `ith` iteration (`1 ≤ i ≤ N`) of the loop is executed on processor `MASTER(C(K(i)))`. The processor may also be specified explicitly, such as in ON `R1(I)`, where `R1` is a processor array. The array `K` serves as a pointer array, whose values are determined at run time. As a consequence, the access pattern associated with the use of `C` and `Y` cannot be statically analyzed. The compiler cannot generate communication statements nor can it determine which processor must execute a specific iteration of the

```
FORALL 1 = 1,N ON MASTER(C(K(I)))
  Y(K(I)) = X(I) + C(K(I))*Y(K(I))
ENDDO
```

---

Note that the parallel loop introduced here, called a **forall loop**, is not the forall loop proposed during the development of Fortran 90.
**Step 1: Loop Analysis**
Perform a dynamic analysis of the loop to determine the local and non-local iteration sets, and, for all \( p' \), \( \text{RECEIVES}(p,p') \).

**Step 2: Computation of SENDS sets**
In a global communication phase, compute all sets \( \text{SENDS}(p,p') \) from the receive sets determined in Step 1.

\[ \text{RECEIVES}(p,p') \] for all \( p' \in P \). The second step uses the relationship \( \text{SENDS}(p,p') = \text{RECEIVES}(p',p) \) to compute, as a result of a global communication phase, the sets of values that must be transmitted prior to execution of the loop.

Although it is superior to individual communications of values, the inspector phase nevertheless represents a significant run time overhead. It is therefore important to recognize situations in which one execution of the inspector suffices to organize the communication for multiple executions of the executor. This may happen when the same \( \text{ON} \) clause is used for several different loops. If arrays are not redistributed, then repeated executions of the same forall loop (at each time step, for example) will use the same iteration sets and the same communication structure; hence, the inspector phase need only be executed once.

3) **The Executor**: The executor is the second phase in the execution of the forall statement; it performs the actual computation. When the executor commences, all communication and iteration sets have been computed in the inspector. The executor consists of four steps (see Fig. 17). The first of these sends all local data that are required by other processors. Then all local iterations are executed. In parallel (if the target machine permits this), nonlocal data required in one processor is received from the others and stored in a temporary buffer. Finally, the executor performs all iterations with at least one nonlocal use, accessing the corresponding buffer elements.

We have not shown the final stage of a forall loop; if values have been assigned to nonlocal data, then there must be a concluding communication phase that transmits these to all processors that own them.

4) **Arbitrarily Distributed Arrays**: The Parti Routines: There are many codes in which the structure of the computation is determined by data that are either input or computed at run time. Among numerical computations, this includes sweeps over irregular meshes and sparse matrix solvers.

The Parti routines [71],[64] represent a concrete system that addresses the specific problems related to the fact that either distributions of data or data dependence patterns are determined by program data that is not known until run time. A library of primitives have been constructed to support the handling of irregularly distributed arrays, whose distributions are specified via mapping arrays, where the value of a mapping array element is the number of the processor storing the array element with the same index. The mapping array is very costly in terms of storage space, since it is as large as the original array, however it permits the specification of an arbitrary distribution.

The implementation has concentrated on the efficient construction of the library routines. Parti routines receive and transmit irregularly distributed nonlocal data, and perform reduction operations on them. The strategy adopted generalizes the inspector-executor paradigm discussed above.

In order to deal with array elements that are arbitrarily assigned to processors, a translation table is constructed from the mapping array, recording the owner of each datum and its local index. It is generally itself distributed in a regular fashion across the executing processors. In step 1 of the inspector, communication has to be performed to find out which processor a data item is stored on, and get the local index. In view of the expense of these operations, a good deal of work has gone into optimizing the handling of these arrays: local copies of nonlocal data are stored using hash tables in a so-called hashed cache. This helps eliminate duplicate fetches. Methods to keep track of the location of nonlocal data and reuse values already received have been developed on the basis of incremental communication schedules, which fetch only those data items that are not already available locally—this often proves useful for sequences of loops where reuse of data items is not uncommon.

The Parti routines have been designed so that they can be inserted into code by a compiler or used directly by programmers. They provides a means for handling code that is otherwise very difficult to adapt to parallel machines, and have been used (manually) to successfully transfer several real codes to the Intel hypercube. The Parti routines have been successfully integrated in the Vienna Fortran compiler that automatically inserts them into code with irregular data accesses.

**VII. EXISTING SYSTEMS**
In this section, we discuss a number of systems that have been constructed to compile code for distributed memory multiprocessors. Most of them were built at research institutions. Yet, two are commercial products that are available on current DMMP's; others that started out as research tools have reached a high degree of maturity.

Most of the work described here is based upon Fortran or C, although there are a few systems based upon functional languages. An integral part of many of the following approaches is the development of suitable language con-
with the syntax of such constructs, but will need knowledge of their functionality in order to understand the role of the user in each system. The kinds of language extensions used include

- processor declarations, to define virtual processors or impose a structure on the real processors,
- specifications for distributing (and moving) data for which a variety of mechanisms exist, ranging from one-dimensional block distributions only to completely arbitrary mappings, and
- explicitly parallel loops, of which there are several different kinds.

Further directives may be provided to enable some kind of local code, or to convey other kinds of information to the compiler: this includes a description of overlap areas, a specification of a work distribution and marking of nonlocal accesses.

Some systems require all relevant information at compile time, whereas others can accept information at run time; there are large differences in the amount of compile-time analysis performed. A variety of different approaches are represented by this material; each has made some contribution to the general problem of converting programs (written at a higher level than message passing code) for a DMMP. Where a system makes assumptions that might prevent its use on general purpose machines, we indicate this. Further, almost all of these systems generate SPMD style code and employ the owner computes rule to distribute the program’s computation. Where this is not so, we state that fact explicitly.

Note that the tools are not presented in a chronological order: many of these developments are not only concurrent, they are projects that were improved upon over a longer period of time and thus do not necessarily have a clear-cut completion date.

**SUPERB** is an interactive restructuring tool that was developed at the University of Bonn by H. Zima and coworkers ([73]). It translates Fortran 77 programs into message-passing Fortran for the Intel iPSC, the GENESIS machine, and SUPRENUM ([31]). SUPERB performs coarse-grain parallelization for a DMMP and is also able to vectorize the resulting code for the individual nodes of the machine. The user specifies the distribution of the program’s data via an interactive language. Program flow and dependence analysis information, using both intraprocedural and interprocedural analysis techniques, is computed and made available to the user, who may select individual transformation strategies or request other services via menus.

SUPERB adapts code in accordance with the general strategy described in Section IV. It puts a good deal of effort into optimizing the target program, extracting communication from loops whenever possible, and combining individual communication statements (by vectorization and fusion) to reduce the overall communication cost [27]. Simple reductions are recognized and handled by the system. SUPERB handles full Fortran 77, dealing with common blocks and equivalencing. It permits arbitrary general block distributions, and has implemented the interprocedural analysis described in Section IV-D. SUPERB does not perform run-time analysis, nor does it allow for redistribution of arrays. It was completed in 1989 and thus was the first system that compiled code for DMMP’s from Fortran 77 code and a description of the distribution of data. SUPERB provides special support for handling work arrays, as are commonly used in Fortran codes, for example to store several grids in one array.

A major concern of the Kali project worked upon by Koelbel, Mehrotra, and Van Rosendale [48],[41],[40], was the design of suitable language constructs to facilitate numerical programming on distributed memory systems. A set of extensions to a high-level programming language were developed for the user to provide the necessary information to map data and computation to the target machine. These extensions were first used with the Blaze language, and subsequently with minor syntactic modifications, with Fortran.

Kali allows the specification of a logical processor array and of basic distributions as explained in Section III-C. The actual number of processors may be unknown at compile time; it must be defined at load time and remains invariant throughout program execution. There is provision for user-defined distributions, using linear mappings. Further, explicitly parallel forall loops are provided (see Section VI-B). The owner computes paradigm is applied to code outside parallel loops. Finally, the user marks procedures that have partitioned arrays as actual arguments; the executing processors are passed to them as a special parameter.

The Kali implementation translates Blaze programs annotated with the Kali constructs to C code with communication statements. Optimization of messages is not performed. Formulas are generated to determine the processor number where each array element is stored. Each processor generates a representation of the data it owns. Nonlocal array elements are generally stored in buffers; this arrangement alleviates the difficulty that the storage requirements are unknown for loops whose communication sets cannot be determined at compile time, but at the cost of needing to keep track of the indices associated with the items in the buffer. Where there is enough information available at compile time, send and receive sets are constructed and communication is inserted into the code for the node processes. Where compile time techniques fail, code is inserted according to the inspector-executor paradigm (Section VI-B) and optimized so that the communication sets generated by the inspector can be reused.

The Parti routines and the ARF compiler [71],[64], developed by Saltz and coworkers at ICASE, represent techniques developed to handle the kind of codes written for sparse and unstructured problems in scientific computing. Their methods have been discussed in some detail in Section VI-B4); they are designed to handle the general
case of arbitrary data mappings, and efficient techniques were developed for a number of subproblems.

The source language for the Crystal compiler built by Li and Chen at Yale University [44] is the functional language Crystal, which includes constructs for specifying data parallelism. Thus there is a certain amount of parallelism explicit in the original code. Experimental compilers have been constructed for the iPSC hypercube and the nCUBE. The program is transformed in a number of phases, first to a shared memory form and thence to a distributed memory version. The domains of the data fields accessed in a code block are aligned with a common index domain: in this way, the compiler determines the data layout. The compiler currently requires additional support from the user to perform these tasks, also requiring information such as can be obtained from profiling tools. The Crystal compiler places particular emphasis on an analysis of the communication requirements to generate efficient communication: the strategy adopted has been described in Section V-II.

Dino [59],[60] represents an early attempt to provide higher-level language constructs for the specification of numerical algorithms on DMMP's. It is explicitly parallel, providing a set of C language extensions. The Dino programmer declares a data structure of virtual processors, called environments, and a mapping of program data and procedures to this virtual machine. Arrays are distributed by block or cyclic mappings, or replicated. The mappings include a specification of the nonlocal elements that might be read by an environment. Distributed procedures are invoked and run in SPMD fashion on an environment. The code within such procedures is local code, referring to local parts of partitioned arrays. Nonlocal data may be read and written; thus DINO does not conform to the owner computes paradigm. Remote accesses are marked by the user.

Dino assumes a sequential host environment and a node environment. In the original implementation, the environment structure could not be modified throughout the program and there had to be exactly one environment per node processor. A subsequent version of Dino (1990) permits, among other things, an arbitrary number of environment declarations, distributed procedure calls to a subset of environments, and functional parallelism, specified by making concurrent distributed procedure calls to different environments.

In this approach, the user explicitly controls the mapping of data, organizes the communication, its optimization, and manages the coherency of data. The compiler must organize the details of setting up reads and writes between sets of processors. In the later version, contracting environments to node processes.

The DINO language constructs have been fully specified and implemented. The DINO compiler generates C code with Intel iPSC/2 message-passing constructs.

Dataparallel C ([34]) is a SIMD extension of the C language that is a slightly modified version of the original C* for the connection machine. Like DINO, it is explicitly parallel and requires the user to specify a local view of computations. Sets of virtual processors, called domains, are declared, together with their local data and procedures. Nonlocal data accesses are marked with the name of their owner. The user may map the virtual processors to the physical processors; the mapping is static, and the number is fixed at compile time. In the implementation, sequential code is replicated on all processors. Code associated with a set of virtual processors is executed on them with a SIMD semantics.

Dataparallel C compilers have been constructed for both shared and distributed memory machines: they produce C code with calls to the Dataparallel C runtime library and are full implementations of the defined features.

The language model provides a global name space and assumes an unbounded number of virtual processors. A major part of the implementation relies upon routing libraries: calls to the routing routines are inserted into the source code by the compiler to implement the communication. Their use is optimized. Major tasks performed by the compiler for DMMP’s are to set up the process code for each node, and relax the SIMD synchronization requirements as far as possible whilst preserving program semantics. A good deal of analysis is performed to optimize communication at synchronization points. A further optimization known as scalarization replaces the local copies of a scalar variable by a single copy when it is safe to do so.

There may be drawbacks in writing code in a SIMD fashion. In view of the mismatch between the language model and the target machines used, the compiler must perform rather different kinds of transformations than the ones we have described. Nevertheless, the implementations have reached a high level of maturity.

A commercially available system is the MIMDizer [50], which is an interactive, menu-driven system intended to be a programming environment for constructing parallel programs. The front end performs standard intra- and inter-procedural analysis and creates a data base of information on the program. The system may be used either to create a program based on functional parallelism or to parallelize a program that is initially written in sequential Fortran according to the SPMD model.

As a parallelization system, the MIMDizer takes a similar approach to SuperC. Cyclic distributions are permitted, however only one array dimension may be distributed, and all arrays associated with a particular formal procedure parameter must be identically distributed. Further consistency requirements have to do with aliasing as permitted in Fortran. It deals with a number of specific Fortran issues, including a very flexible handling of common blocks. The MIMDizer also provides facilities for instrumenting code, and replaying the output from an instrumentation run. The directives used to parallelize a program are retained in a separate file of directives.

The Oxygen compiler, developed by Annaratone, Ruhl and colleagues [61], was originally constructed for the experimental K2 DMMP, but has been ported to the Warp and iWarp machines. Its compilation strategy is less general.
than the above since it relies on the assumption that communication between different processors is relatively inexpensive. It uses techniques that permit updates of a data element on different processors and keeps track of the most recent copy; these mechanisms would be prohibitively expensive on many current DMMP's.

It parallelizes Fortran source code annotated by directives as described above; the implementation generates code in C++ for a torus of processors. The Fortran program is decomposed by the user into a series of blocks: these are either local, and execute without communication on each processor, or public, which may require communication. Oxygen performs both compile time and run time analysis in much the same way that Kali does, but performs the latter analysis for any loop involving indirect references. Public blocks are converted along the lines of the inspector-executor paradigm (Section VI-B). In contrast to Kali, a parallel loop may contain inter-iteration dependences, which are handled by performing run time data dependence analysis. Both data and loop iterations may be distributed either in blocks or cyclically in one or two dimensions. Data are declared to be either local, single copy, (owned by one processor throughout execution) or multicopy (each processor has a copy and ownership changes dynamically). This latter requires a good deal of work at run time and increases storage requirements, but permits very flexible handling of arrays with indirect accesses.

The Hypertasking system implemented by Baber [4] converts a sequential C program together with directives into code for the Intel iPSC hypercube. Directives are used to specify the distribution of arrays (by block only), the width of overlap areas, the updating of overlap values, and to indicate that a loop operates on local indices only. This latter may be combined with directives for data motion, e.g., to rotate columns of an array across a ring of processors. The program may be written independently of the number of processing nodes it executes on. The executable redistribute statement is implemented by using and carrying out the appropriate data motion (which is optimized), and updating the distribution specifications maintained by the node processes. A severe limitation of the implementation of this system, however, is that it does not permit procedure calls to be made with partitioned parameters.

Aspar is implemented within the Express environment to support the parallelization of C code ([38]). The "automatic and symbolic parallelization" (Aspar) tool uses the Express run-time support for communication, data partitioning, and I/O. In contrast to the other systems described here, Aspar relies on symbolic analysis to find a partitioning strategy for the arrays in a program. Pattern matching analysis compares loops with reduction operations and a set of standard stencils. In the case of a successful match, the system retrieves a suitable distribution from a data base and determines the required overlap areas and communication. The major drawback of this system is that it performs very little analysis and cannot deal with many codes; moreover, it makes assumptions that could lead to code that is semantically modified in ways that might not be obvious. Nevertheless, the recognition of frequent code patterns and the consultation of a corresponding data base of distributions is an interesting new approach to this task.

Other currently existing systems include AL, which has been implemented on the Warp systolic array processor [67], Pandore, a C-based system [3], and Id Nouveau, a compiler for a functional language [58].

A number of systems, all based upon a set of language extensions to Fortran 77 and/or Fortran 90, are currently being implemented. This includes Adapt, being developed at the University of Southampton [49], the Yale extensions [15], Fortran D [23],[35] (Rice University), a new set of language extensions to Cray Fortran [54], DEC's High Performance Fortran proposal [47], and Vienna Fortran [14]. The full implementation of these languages—for which a common basis is being sought in the High Performance Fortran Forum—will require a significant advance in the state of the art of compilation systems (see Section VIII).

The systems described above are not the only efforts to provide either suitable language constructs for mapping code onto DMMP's or to generate message passing programs from higher level code. A rather different approach that has had some measure of success is to enable functional parallelism by providing a mechanism for coordinating individual threads of a parallel program. Notable examples of this style of program development are Linda [2] and Strand [22]. Special purpose systems have been constructed to compile code for specific architectures and specialized application areas. PDEQSOL [68], Suspense [62], ALPAL [18], PARAGRAM [72], and Parallel ELLPACK [36] translate algorithmic formulations at a very high level into code for a specific architecture. Other systems take input in a special-purpose language and are highly optimized for a specific language and machine. One of these is Apply, a language designed for a specific class of image-processing problems that has been implemented on the iWarp [6], which is in commercial use.

VIII. FUTURE RESEARCH

There are still many challenges facing the builders of compilation systems for DMMP's. Much improvement is needed with regard to functionality, the efficiency of the generated code, and the quality of the interaction with the user.

A. Improved Functionality

Real codes require a combination of compile-time and run-time methods for analysis, data management and communication generation. The handling of dynamic data redistribution, the development and efficient use by the compiler of library routines that may be called with partitioned data, and the more efficient application of source level transformations to facilitate the generation of high-quality code are all topics that require attention. Furthermore, compilers must facilitate the storage and retrieval of arrays in distributed files striped across disks as in modern concurrent file systems [55],[19]. Here, we discuss just a few of the relevant issues.
One research topic receiving attention is the development of support for the user in the crucial task of selecting a data distribution. The techniques employed will depend on the kind of program under consideration and require extensive support from performance analysis tools. Li and Chen have developed a strategy for aligning data in regular codes within the Crystal compiler [45]; several other researchers have made substantial contributions to the problem of determining a distribution of program data. These include both theoretical approaches [57], [37], a constraint-based approach [33] and an implemented system for the CM-2 [39]. A useful extension of this work would be a tool that determines the distribution of data (thus generating DPF programs, at least in part), displays the results and permits graphical user modification of them.

Irregular codes will require a different approach, using general distribution routines provided by the programmer or the system. At run-time, these routines must compute data distributions based on the data access pattern for a particular loop. High level support for distributing the data of both regular and irregular problems within a specific problem domain is provided by the domain decomposer [16] within Parallel Ellpack, which provides tools to assist the user in distributing the nodes of a geometric mesh to processors.

B. New Methodologies

The knowledge of a few useful transformation sequences has enabled an automatic approach to certain tasks in current compiler systems; these include applying transformations to simplify program analysis, and some simple methods for distributing loop iterations. Evolving systems will be expected to perform many tasks that require a global program transformation strategy. Performance tools will provide a means to obtain knowledge about the performance characteristics of the program and the way performance is affected by the application of transformation sequences. There are different, and sometimes conflicting, goals to be attained (e.g., load balancing versus minimization of communication), and nontrivial trade-offs to be considered. The results of analysis, combined with a set of heuristics, may be used to select predetermined transformation strategies. More advanced transformation strategies might include the detection of certain patterns in code and their replacement by highly optimized routines for the target machine.

The hardware and software architectures of parallel computing systems are currently undergoing rapid evolution. These modifications affect the parallelization strategy for a machine: whereas the basic strategy may remain unchanged, the details of optimization could be radically different. The representation and use of explicit knowledge on both architectures and their software systems (including the compilers), and the identification of rules for the application of optimization strategies in terms of this knowledge, may enable parallelizing systems to keep up with the pace of change, and permit them to be retargeted in response to system developments without a major reprogramming effort.

Paulvast et al. [52] have developed a calculus that could be used as a language-independent framework for expressing programs and their data, which enables the representation and reasoning about code transformations in the form of rewrite rules. They have also shown that the language Booster [51] can be converted to their calculus.

There are significant advantages in organizing an advanced programming environment as a collection of knowledge-based subsystems with different levels of expertise: 1) The possibility of rapid prototyping, 2) relatively easy modification of the programming environment if the underlying system changes, and 3) the availability of an explanation facility. So far, only few attempts have been made to develop restructuring systems using these techniques [9], [10], [70], [69]. This method has been explicitly applied to the problem of generating code for distributed-memory computers by Ko-Yang Wang at Purdue University. He has constructed a prototype system that has a description of a target machine, a set of program transformations, and rules for applying the transformations that are written in terms of the machine parameters. This approach permits code to be transformed for a variety of different machines by a single compilation system [69].

C. Integrated Programming Environments

A compiler in the conventional sense will not, of its own, suffice to support the highly complex and challenging task of producing efficient programs for parallel systems. We believe that advanced compilation systems will be integrated into a sophisticated programming environment that includes an extensive set of programming support tools.

These will be needed to provide guidance in a number of forms; they must satisfy the needs of a broad spectrum of users, some of whom will require an integrated system to work as automatically as possible, whereas others will expect to be provided with detailed and precise information and transformation tools, enabling them to make the major strategic decisions for parallelization themselves. Since one of the primary motivations for using parallel systems is to attain high performance, appropriate tools for performance prediction and selective measurement must be integrated. Performance tools will not only identify the most important areas of existing code. They will also support the selection of appropriate algorithms, and help determine appropriate transformation strategies. This may involve invoking a performance prediction tool, running selected areas of the corresponding code with sample input values, or accessing performance information stored within the database of information on the program.

Additional tools include debuggers that will need to relate programming errors to the source program in a comprehensible manner, and visualization tools that will help the user monitor a program’s behavior and utilization of the machine’s resources.

Parallelizing compilers cannot always perform well without assistance from the user. One of the main reasons for this is the undecidability or intractability of many
relevant problems and the lack of adequate heuristics for handling them; furthermore, a static analyzer will have no information at all on variables whose value is input during program execution. The user may play an important role, informing the system via assertions of global relationships (some of which may be due to high-level properties of the algorithm) that an automatic state-of-the-art tool cannot detect.

REFERENCES


[49] J. H. Merlin, "ADAPTing Fortran 90 array programs for dis-


Hans P. Zima (Member, IEEE) was born in Vienna, Austria, in 1941. He studied mathematics, physics, and astronomy at the University of Vienna, where he received his Ph.D. degree in 1964.

After spending more than eight years in the computing industry in Germany and the United States, he accepted a research position at the University of Karlsruhe. He was Professor of Computer Science at the University of Bonn from 1975 to 1989. During this time, he led the development of the SUPERB parallelization system; he performed research in the field of constraint languages at the IBM San Jose Research Laboratory in 1983-84 and initiated a cooperation with Rice University, where he spent a year in 1988-89. He is currently Professor of Computer Science at the University of Vienna, Adjunct Professor of Computer Science at Rice University and Director of the Austrian Center for Parallel Computation (ACPC). He led the development of Vienna Fortran, one of the major inputs for the High Performance Fortran effort. He is the author of several books in the area of compiling and parallel systems, the most recent of which is Supervectors for Parallel and Vector Computers (coauthored with Barbara Chapman). His main research interests are in the field of advanced programming environments for parallel machines, in particular automatic parallelization for distributed-memory machines, performance analysis, and knowledge-based transformation systems.

Dr. Zima is member of the ACM and Gesellschaft fuer Informatik.

Barbara Mary Chapman was born in Dunedin New Zealand, in 1954. She received the B.Sc. (Honors) degree in mathematics from the University of Canterbury in 1975 and pursued further studies in mathematics and computer science at both the Victoria University of Wellington, New Zealand, and the University of Bonn, West Germany.

She joined the staff of the Department of Computer Science at Bonn University in 1985, teaching courses in compiler construction, parallel systems and programming parallel computers, with an emphasis on systems with distributed memory. Ms. Chapman currently holds a research position at the Computer Science Department of the University of Vienna. She is one of the main architects of Vienna Fortran, a Fortran language extension for distributed-memory multiprocessor systems.

Ms. Chapman has received the University Junior Scholarship and the Postgraduate Scholarship in New Zealand, the Cook Memorial Prize from the University of Canterbury, and the D.A.A.D. Scholarship from the Government of the Federal Republic of Germany. She is coauthor of the book Supervectors for Parallel and Vector Computers and of a number of papers in the field of programming support for parallel machines. Her current research interests are in programming languages and environments for distributed memory multiprocessors, in particular advanced language and compiler support for irregular and adaptive problems. Ms. Chapman is a member of the ACM.