The Journal of Supercomputing, 2, 151-169 (1988)
© 1988 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Compiling Programs for Distributed-Memory Multiprocessors*

DAVID CALLAHAN
KEN KENNEDY
Department of Computer Science, P.O. Box 1892, Rice University, Houston, TX 77251

Abstract. We describe a new approach to programming distributed-memory computers. Rather than having each node in the system explicitly programmed, we derive an efficient message-passing program from a sequential shared-memory program annotated with directions on how elements of shared arrays are distributed to processors. This article describes one possible input language for describing distributions and then details the compilation process and the optimization necessary to generate an efficient program.

1. Introduction

The use of shared-memory is a fundamental paradigm for parallel programming, especially numerical and scientific programming. Unfortunately, large shared-memory multiprocessors are expensive to build. Distributed-memory multiprocessors, where each processor can directly address only a portion of the total memory of the system, are significantly less expensive and hence should have a significant cost performance advantage over shared-memory machines. There are, however, at least two difficulties associated with using distributed-memory computers. First, many problems may require so much interprocessor communication that machine efficiency is very low. However, Geoffrey Fox [Fox and Otto 1986, Fox 1987] and others [Heath 1987] have demonstrated effective utilization of distributed-memory computers for a wide range of scientific applications. Second, distributed-memory computers are harder to program than shared-memory machines [Mandell 1987]. We believe that much of this difficulty is due to the lack of shared memory in the *virtual machine* provided at the programming language level. Furthermore, we believe that shared memory can be provided at the language level even when it is not supported by the hardware. We are investigating the use of advanced compilation technology to provide this support.

In this article, we describe a general approach to compiling programs for distributed-memory multiprocessor computers. A "program" in this context is a sequential program annotated with directives to control the mapping of "shared" arrays to local memories. These directives will be presented as extensions to Fortran 77, though in principle they could be used with any sequential language as a base. The target virtual machine supports an arbitrary number of virtual processors with asynchronous message-passing between them. The compiler is responsible for generating a correct

* Research supported by Intel.

program for this virtual machine; in particular, all data transfers between processors are managed solely by the compiler. The "parallel" program can be roughly described as SPMD (single-program, multiple-data [Karp 1987]): Every processor executes the same program but performs computation on distinct data items. Parallelism is implicitly exploited but speedup will depend on good compiler optimization.

The advantages of this approach to distributed-memory programming are:

- **Well-defined semantics** The behavior of a program is completely defined by the standard sequential semantics of the base language.
- **Portability** Porting a program to a machine with different communication costs is only a matter of changing the data distribution directives. In particular, programs can be easily and efficiently run on sequential computers simply by ignoring the distribution directives; programs can be partially parallelized, allowing programs to be parallelized one step at a time; and programmers are free to experiment with different distribution patterns without the cost of extensive recoding.
- **No message passing** The programmer does not program any of the message passing and so does not spend any time debugging the message passing. All accesses to program data elements are coded the same way.

The disadvantages of this approach are:

- **Complex source/executable relationship** Since the executable program will look very different from the source program, debugging the executable with low-level tools (i.e., break points and single-stepping) will be difficult.
- **Requires sophisticated compilers** Small deficiencies in the compiler can result in large inefficiencies in the executed program.
- **Requires performance debugging tools** Performance problems must be identified and mapped back to the original program before they can be corrected.

These disadvantages are shared by almost all high-level, explicitly parallel languages. Other disadvantages which are particular to the example language described here but not necessarily to the approach in general are:

- **Unrestricted pointers** Base language features which rely on the concept of an *address* or a *linear address space* are difficult to implement efficiently in this system. For Fortran, the primary difficulty is call-by-reference parameter passing, especially the passing of array subsections, and equivalencing.
- **Complex distributions** Complex or irregular distributions are difficult to express and their properties may be difficult for the compiler to exploit.

Despite these disadvantages, this approach provides a powerful vehicle for programming distributed-memory computers.

The remainder of this article details a particular example of this approach, focusing on the programmer-supplied annotations for data distribution, the target machine, the basic compilation mechanism, the types of optimizations necessary to achieve speedup, and the tools available for these optimizations.

2. The Input Language

The extensions to Fortran 77 we assume are statements that specify on which processor each element of a shared variable is stored. The general distribution statement is:

```
    DISTRIBUTE a(il,...,ik) = f(il,...,ik)
```

where `A` is any variable name with at least k dimensions. The expression to the right of the equals sign is an expression involving the dummy names `il, . . . , ik` and other program variables. The `DISTRIBUTE` statement defines a locally known function which identifies the processor on which element `A(il, . . . , ik)` is stored. Processors are assumed here to be identified by integer values but could be represented by a set with more structure. The `DISTRIBUTE` statement is executable (rather than simply declarative); a second `DISTRIBUTE` statement for the same variable changes the distribution function associated with that variable. This redistribution does not imply movement of data between processors; such a movement must be explicitly programmed using two distributed arrays. It is assumed that such a redistribution would accompany some phase change in the algorithm. An example would be to change from a column-oriented distribution for use during an LU decomposition of an array to a row-oriented distribution for use during the backsolve phase.

One restriction we place on the functions defined by `DISTRIBUTE` statements is that they must be one to one into the space of processors. It is our intent that when clustering multiple elements of an array onto a single processor is desired, it will be done either by using the decomposition statement described later or at the operating system level by timesharing one real processor among multiple virtual processors. We believe that load balancing also can be done automatically at the operating system level [Fox 1985].

The compilation system will provide some special functions for common distribution patterns. For example, the statement

```
    DISTRIBUTE A(il, i2) = GRID(il, i2)
```

would be equivalent to the distribution

```
    DISTRIBUTE A(il, i2) = N*(il - 1) + i2
```

where `N` is the row stride of `A`. There are two advantages to these additional functions. First, they hide the arbitrary numbering of processors from the programmer and, second, the compiling system can have greater knowledge about the distribution function than if it had to infer facts from general expressions. The `GRID` function may be particularly valuable, since it retains the independence of subscripts information that might be lost in the expression `N*(i1 - 1) + i2`.

We believe common distribution patterns will involve blocking arrays and mapping all elements in a given block to the same processor. To facilitate such a distribution, the input language also includes a statement for specifying data decompositions. The `DECOMPOSE` statement

```
DECOMPOSE A(l1:u1,...,lk:uk) INTO VA(u1b,...,ukb)
```

dynamically defines a virtual array, the elements of which are blocks of elements of some underlying real array. In this example, `A` plays the role of the real array and `VA` the role of the virtual array.

An element of `A` specified by indices `i1, . . . , ik` is mapped into the element `(b1, . . . , bk)` of the virtual array `VA` where

$$(\mathtt{b1}, \ldots, \mathtt{bk}) = \left(\left\lfloor \frac{\mathtt{i1 - l1}}{\mathtt{BS1}} \right\rfloor + 1, \ldots, \left\lfloor \frac{\mathtt{ik - lk}}{\mathtt{BSk}} \right\rfloor + 1\right) \qquad (1)$$

and the block width in dimension *i* is given by

$$\mathtt{BSi} = \left\lceil \frac{\mathtt{ui - li + 1}}{\mathtt{uib}} \right\rceil \qquad (2)$$

This formula implies that with respect to each dimension, all blocks except the last one are equal in size. An alternative would make all blocks equal in size within 1 in each dimension. This choice is motivated in part so that strip mining at the block size is straightforward. For large numbers of blocks, the inequality should not matter. For small numbers of blocks, the inequality should be small. The defined virtual array can only be used in `DISTRIBUTE` statements and other `DECOMPOSE` statements.

The number of elements in each block must be within the declared bounds for the associated dimensions of the real array. For instance, if `A` has declared range specifiers `dl1:du1, . . . , dlk:duk`, the requirement implies

$$\mathtt{BSi} \leqslant \mathtt{dui - dli + 1} \qquad (3)$$

for each dimension *i*. This requirement means the "real" array's bounds no longer exactly fit the programmer's view. However, it does allow the compiler to allocate the appropriate amount of memory per node and allows programs to be written so that, as problem size grows, one simply increases the number of processors, leaving the program the same.

An example of a fully specified program for multiplication of matrices is shown in Figure 1. Observe that the program first determines the array sizes, then dynamically decomposes and distributes the arrays. Note that the data decomposition implicitly assumes 16 processors and that the 4-by-4 structure is "hard-wired" into the program. The value 4 could also be data-dependent or machine-dependent if there were a mechanism for determining the number of real processors. The `cubedim()` function defined for the Intel iPSC is an example of such a mechanism. The maximum value of `N` for this program is 1024 (4 × 256) but if the decomposition were made dependent on machine size, then running larger problems would entail finding machines with more processors, but not recompiling the program.

3. The Target Machine

The target machine is a multiprocessor without global shared memory. Each processor has some amount of local memory which is the only memory that is directly addressable by that processor. The operating system provides a virtual processor abstraction to allow multiprocessing each node. We will use the term *processor* to refer to a process provided by the operating system rather than a hardware resource.

```
      PROGRAM MULT
      REAL A(256, 256), B(256, 256), C(256, 256)
      READ *,N
      DECOMPOSE A(N,N) INTO VA(4,4)
      DECOMPOSE B(N,N) INTO VB(4,4)
      DECOMPOSE C(N,N) INTO VC(4,4)
      DISTRIBUTE VA(i, j) = GRID(i, j)
      DISTRIBUTE VB(i, j) = GRID(i, j)
      DISTRIBUTE VC(i, j) = GRID(i, j)
      DO 10 I = 1, N
10       READ *,(A(I, J), J = 1, N)
      DO 20 I = 1, N
20       READ *,(B(I, J), J = 1, N)
      DO 30 I = 1, N
         DO 30 J = 1, N
            C(I, J) = 0.0
            DO 30 K = 1, N
30             C(I, J) = C(I, J) + A(I, K)*B(K, J)
      DO 40 I = 1, N
40       PRINT *,(C(I, J), J = 1,N)
      STOP
      END
```

Figure 1. Matrix multiplication.

Every processor may communicate with any other processor via asynchronous message passing. The two basic primitives are

```
SEND(DEST = pid) Variable List
```

and

```
RECV(ONLYSRC = pid) Variable List
```

where `Variable List` is a list of Fortran variable references (scalars and array elements). The keyword `DEST` in the `SEND` statement specifies the processor to which the data is to be sent. For the `RECV` statement, an `ONLYSRC = pid` clause specifies that only a message from the processor identified by `pid` should be received. The data transmitted between processors are the concatenation of the values of the variables specified on the variable list.

It is assumed that the `SEND` statement is nonblocking (asynchronous) but that the `RECV` statement will block until a message is received that satisfies the `ONLYSRC` clause when one is specified. Messages from the same source are received by the program in the order in which they were sent. Message passing could be synchronous, but that would reduce parallelism and efficiency [Fox 1987].

In addition to `SEND`, we also assume a `GSEND` which broadcasts a message to all other processors. Such a message is received with a standard `RECV`.

We also assume a special variable `thisproc` which identifies to each processor its unique identifier. This variable has type *integer* and its value can be stored in any integer variable. Processors are numbered sequentially from 1 to some positive upper bound.

The target machine is fully connected: Every virtual processor can communicate directly with every other virtual processor. Real hardware usually simulates fully connected hardware with hypercube or mesh interconnections. This simulation is made transparent by the operating system, but careful selection of data distributions will make more effective use of real hardware. The choice of distributions is beyond the scope of the compilation system proposed here.

4. Basic Compilation

The basic compilation rule followed here is: the processor that can access a data element is the processor that will perform a calculation to update that location. In the most naively compiled program, every processor "executes" every statement. One of the processors will collect data values, perform a calculation, and update its local memory. Some of the remaining processors will send data to the first processor and the rest of the processors simply move on to the next statement. In an ideally compiled program, after a processor contributes, to the execution of a statement, it moves on to the next statement to which it contributes, as ordered by sequential execution. The compiler optimizations described in the next section bridge the gap between the basic compilation described here and the ideal.

In a shared-memory computer, every program data element is assigned a particular address in memory expressed as an integer index into a linear sequence of locations. In the distributed model, every data element is assigned an address which is an ordered pair. The first component identifies a processor and the second component identifies an (integer) address in the local memory space of that processor. To refer to these components individually, we use two functions, δ and α. Both of these functions take as arguments a distributed shared variable and an element selector. The function δ returns the processor identifier of the processor which contains the selected data element. The function α returns the location of a particular data element in the local memory of the processor which holds it. By context, the value returned by α will be used to retrieve the value contained in the data element or store a new value into it. In the transformed (parallel) program, α will be represented by an access to a local array name.

In this section, we will denote references to original program variables using uppercase letters and references in the transformed program using lowercase letters. For example, `A(I, J)` is a reference to the programmer-defined shared array `A` but `a(i, j)` is a reference to a compiler-created local array. The terms `MI, MO, MØ, M1, . . . , Mk` refer to arbitrary references to original program variables, both scalars and indexed array elements. The variables `t, t1, . . . , tk` refer to compiler-introduced variables local to each processor.

The primary task of the compiler is to separate the movement of data from the computation of new results. For this purpose, we introduce two new statements, `LOAD MI, t, pid` and `STORE MO, f(t1, . . . , tk)`. These statements appear only in the compiler's intermediate language, not in the source programming language. The `LOAD` statement moves the value stored in data element `MI` into temporary location `t` on processor `pid`. The definition of data element will be made more concrete later. The statement `LOAD MI, t, pid` can be expressed as

```
if $\delta$(MI) = thisproc then
    if pid = thisproc
        then t ← $\alpha$(MI)
        else SEND(DEST = pid) $\alpha$(MI)
else if pid = thisproc
    then RECV(ONLYSRC = $\delta$(MI)) t
```

In the `STORE MO = f(t1, . . . , tk)` statement, `MO` is a program data element and `f` is an arithmetic expression with input values `t1, . . . , tk`. This statement can be expressed as

```
if $\delta$(MO) = thisproc
    then $\alpha$(MO) ← f(t1, . . . , tk)
```

A standard Fortran assignment statement

```
MO = f(M1, . . . , Mk)
```

where f is any arithmetic expression built from the input data elements M1, . . . , Mk is "compiled" to the sequence

```
LOAD M1, t1, δ(MO)
LOAD M2, t2, δ(MO)
⋮
LOAD Mk, tk, δ(MO)
STORE MO = f(t1, . . . , tk)
```

Observe that this compilation strategy has the processors actively conspiring to simulate shared memory.

The function δ is specified by the programmer for some arrays via the DISTRIBUTE statement. If the programmer does not specify a distribution for a data element, we say the data element has a *floating distribution* If MO is a data element with a floating distribution, a statement

```
MO = f(M1, . . . , Mk)
```

which assigns into that data element, determines its current distribution

```
δ(MO) ← δ(Mi)
LOAD M1, t1, δ(MO)
LOAD M2, t2, δ(MO)
⋮
LOAD Mk, tk, δ(MO)
STORE MO = f(t1, . . . , tk)
```

for some Mi with the arbitrary assignment δ(MO) $\leftarrow$ 0 if the right side of the assignment statement is constant. Where necessary, floating distributions could be stored as replicated tables.

In addition to floating distributions, some variables will be *replicated*: Every processor contains an up-to-date copy of this variable's value. If M is a replicated variable, then we define δ(M) to be 0, which is not a valid processor identifier. The GSEND statement is used to update a replicated variable from a non-replicated source. The most general LOAD MI, t, pid is now

```
if δ(MI) = thisproc then
    if pid = thisproc
        then t ← α(MI)
    else if pid ≠ 0
```

```
            then SEND(DEST = pid) α(MI)
        else do
            t ← α(MI)
            GSEND t
    else if δ(MI) ≠ 0 then
        if pid = thisproc or pid = 0
            then RECV(ONLYSRC = δ(MI)) t
    else if pid = thisproc or pid = 0
        then t ← α(MI)
```

and for `STORE MO = f(t1, . . . , tk)`:

```
    if δ(MO) = thisproc or δ(MO) = 0
        then α(MO) ← f(t1, . . . , tk)
```

Enough machinery has been developed to rigorously define the arguments to LOAD and STORE. For `LOAD M, t, pid,` M must be either a scalar variable in the original program or a subscripted array reference, `A(I1, . . . , Ik)` where A is an array in the original program and each index variable `I1, . . . , Ik` is a replicated variable. The compiler will have to break up complex subscripts, such as `IP(I)` in `A(IP(I))`, into separate statements by introducing replicated variables. The statement

```
    B(I) = A(IP(I))
```

would compile into

```
    δ(T) ← 0
    LOAD IP(I), t1, δ(T)
    STORE T = t1
    LOAD A(T), t1, δ(B(I))
    STORE B(I) = t1
```

assuming that I is already replicated.

Control flow decisions can be made by introducing replicated variables to hold the branch choice. For example, the conditional branch

```
    IF (f(M1, . . . , Mk)) GOTO 10
```

is compiled into

```
    δ(L1) ← δ(MI)
    LOAD M1, t1, δ(L1)
    LOAD M2, t2, δ(L1)
```

```
:
LOAD Mk, tk, δ(L1)
STORE L1 = f(t1, . . . , tk)
δ(L) ← 0
LOAD L1, t, 8(L)
STORE L = t
IF (α(L)) GOTO 10
```

We expect that most control decisions inside loops will be made either based on the values of replicated variables or that only one processor will be active for that iteration and so branching will not be as expensive as it appears.

Loop control variables are also replicated everywhere. The loop

```
      DO 10 I = IL, IU, IS
             :
   10 CONTINUE
```

is compiled into

```
      δ(I) ← 0
      LOAD IL, tIL, δ(I)
      LOAD IU, tIU, δ(I)
      LOAD IS, tIS, δ(I)
      DO 10 α(I) = tIL, tIU, tIS
             :
   10 CONTINUE
```

Many distributed-memory machines are attached to some host and depend on that host to perform I/O. This arrangement can be handled by running a process on the host and "assigning" the I/O system to it. When an I/O statement is encountered, all of the variables needed are sent to the host and the host performs the I/O operation and returns any results. This can be accomplished uniformly by assigning the host an otherwise invalid process id.

The local component of an address, $\alpha(M)$, is straightforward to realize. Given the general decomposition shown above, `α(A(I1, . . . , Ik))` would be realized as

```
    A(I1 - BS1*b1 + BS1, . . . , Ik - BSk*bk + BSk)
```

where `(b1, . . . , bk)` is the block containing `A(I, J)` as defined in Equation 1 and each `BSi` is the block width as defined in Equation 2.

5. Optimization

The basic compilation mechanism described in the previous section will correctly map a Fortran program onto a distributed-memory computer but effective utilization of the multiple processors will require aggressive optimization. In this section we identify three basic goals for optimization together with the tools available to achieve these goals, and we outline a strategy for applying these tools.

5.1. Goals

Under the basic compilation mechanism, every processor "executes" every statement. Parallelism is implicit in that no synchronization occurs between processors that do not contribute to the current statement and so they can proceed to the next statement. Speedup, however, requires that the average time spent by all processors on a particular statement is less than the time to execute that statement on a single processor. The primary goal of optimization is to reduce this average time for each statement.

The most effective way to reduce the average time is to have processors that do not contribute to a statement simply not execute that statement at all. One way to do this is to recognize when several statements are all executed by one processor and take data from one processor. Once it is determined that a given processor does not contribute to the first statement, we can infer it does not contribute to the rest and it can "escape" to the next statement where it might contribute. A more powerful example of this idea applies to loops: a given processor may only contribute on a single iteration. If we can determine quickly which iteration, the processor can avoid iterating through the loop "searching" for something to do. This determination will require some compile-time knowledge of distribution functions.

On current distributed computers, message passing is very expensive compared to basic arithmetic. Furthermore, the start-up cost for a message is large compared to the per-element cost. These characteristics suggest a second optimization goal should be to combine separate messages from the same source to the same sink into a single message and to eliminate redundant messages. A redundant message would occur when a processor uses nonlocal data multiple times. As long as there is available local memory and the nonlocal data does not change, the nonlocal data can be cached locally.

The third optimization goal is to reorder access to local data to avoid bottlenecks to individual processor memories. The basic compilation mechanism preserves the order of memory accesses to the data assigned to a particular processor. This property can cause bottlenecks when one processor performs a computation and then sends unrelated data to other processors. This bottleneck can effectively serialize a loop in much the same way cycles of dependences limit parallelism under the DOACROSS model of parallel loops [Cytron 1986]. Consider the following example:

```
      DISTRIBUTE A(I) = I, B(I) = I
      DO 10 I = 2, 15
10       A(I) = B(I - 1) + B(I + 1)
```

Almost every processor contributes to three iterations and, except for the boundary iterations, each processor will perform a send, two receives and then another send. For example, processor 8 contributes to iterations `I = 7, 8, 9`. For iteration `I = 7`, processor 8 sends data (`B(8)`) to processor 7 then processor 8 must wait for processor 7 to complete iteration 7 and send `B(7)` to processor 8. In turn, processor 9 must wait for processor 8 to complete iteration 8 to send it data. Again, there is a tradeoff between reducing this bottleneck and local memory consumption. For example, we could transform the above loop into

```
      DISTRIBUTE A(I) = I, B(I) = I
      DISTRIBUTE BM1(I) = I
      DO 11 I = 15, 2
11       BM1(I) = B(I - 1)
      DO 10 I = 2, 15
10       A(I) = BM(I) + B(I + 1)
```

Note that the iteration order of the `DO 11` loop has been reversed so that most processors will do a send, a receive, another send, and another receive without any serialization. This eliminates a bottleneck at the expense of a temporary copy of `B`.

5.2. Tools

One great strength of the basic compilation mechanism is that any correct transformation of the Fortran program under traditional compilation mechanisms is still correct under the basic compilation mechanism for distributed memories. For example, constant propagation, common subexpression elimination, loop invariant code motion and dead store elimination [Allen and Cocke 1972] are as applicable to the mechanism described in Section 4 as in any other compiler. Furthermore, exactly the same analysis is needed for correctness. Hence, we are able to adopt essentially all of the "traditional" scalar optimization machinery. All aspects of data dependence analysis [Kuck 1978, Kennedy 1980, Wolfe 1982] and the loop restructuring it allows [Kuck et al. 1983, Allen and Kennedy 1987, Padua and Wolfe 1986] can also be exploited. Only the objectives of the restructuring are tailored to match the goals listed above.

In addition to optimizations at the source level, we can also optimize the intermediate level where `LOAD`s and `STORE`s are present but before `SEND`s and `RECV`s have been introduced. An important correctness issue is that if the order of `SEND`s and `RECV`s is changed on one processor, we must make sure that that change is respected by other processors; otherwise deadlock is possible. This is not an issue at the LOAD/STORE level and so we want to do as much of the data access reordering as possible at this level.

To facilitate combining messages, we introduce a generalized LOAD instruction,

```
    LOAD varlist, pid
```

where varlist is a list of items to be loaded. Each item is a program variable/compiler temporary pair, possibly indexed and included in an implied DO. For example, the instruction

```
    LOAD N, t, (A(L + i), t(i) i = 1, M), pid
```

would load the value of N into t and the values of A(L + 1), . . . , A(L + M) into t(1), . . . , t(M), respectively. Of course, we have the requirement that the distribution function δ have the same value for every program variable referenced on varlist.

Since LOADs and STOREs are essentially assignment statements, they can be moved and reordered under the same correctness considerations as standard Fortran statements. In particular, the tests for loop distribution and the application of scalar expansion can be used to "preload" block-decomposed data before execution of a loop begins. This is how array temporaries can be introduced into LOAD/STORE statements.

Since the distribution functions appear explicitly in LOAD statements and implicitly in STORE statements, at the LOAD/STORE level we can exploit compile-time knowledge about distribution functions. For loops which "sweep" over the arrays in a regular manner, the compiler can infer which iterations a particular processor contributes to and limit the processor to those iterations. If the distribution functions are simple enough that the compiler can recognize their inverses, loops can be removed altogether. If we make the obvious use of the distribution function and the special variable thisproc, we can further transform the loop

```
        DISTRIBUTE A(I) = I
        DISTRIBUTE BM1(I) = I, BP1(I) = I
        DO 10 I = 2, 15
    10      A(I) = BM1(I) + BP1(I)
```

into

```
    if (thisproc .ge. 2) then
       if (thisproc .le. 15) then
          α(A(thisproc)) ← α(BM1(thisproc)) + α(BP1
            (thisproc))
       endif
    endif
```

For arrays with block decompositions, strip mining can be used so that the outer loop "sweeps" over blocks. For example, a modification of the previous loop

```
        DECOMPOSE A(16) INTO VA(4)
        DECOMPOSE BM1(16) INTO VBM1(4)
        DECOMPOSE BP1(16) INTO VBP1(4)
        DISTRIBUTE VA(I) = I
        DISTRIBUTE VBM1(I) = I, VBP1(I) = I
        DO 10 I = 2, 15
10         A(I) = BM1(I) + BP1(I)
```

could be transformed by strip mining into

```
      DO 10 IP = 1, 4
         I1 = 4*(IP - 1) + 1
         IL = MAX(I1, 2)
         IU = MIN(I1 + 3, 15)
         DO 10 I = IL, IU
10          A(I) = BM1(I) + BP1(I)
```

and then into

```
      if (thisproc .le. 4) then
         I1 = 4*(thisproc-1) + 1
         IL = MAX(I1, 2)
         IU = MIN(I1 + 3, 15)
         DO 10 I = IL, IU
10          α(A(I)) ← α(BM1(I)) + α(BP1(I))
      endif
```

We call this process *loop elimination.*

Finally, the LOAD/STORE statements can be devolved into explicit tests and SEND/RECV statements as described in Section 4. Additional scalar optimizations are now possible. Common subexpression elimination and loop invariant code motion can be applied to the code which tests thisproc against the value of distribution functions. Also loop unswitching and strength reduction [Allen and Cocke 1972] will be useful.

5.2. Strategy

Much of the optimization strategy implied in the previous section is made explicit here. Optimization will proceed in these steps:

1. **Constant propagation** Propagate constants and other information about loop bounds forward through the program.

2. **Strip mining** Locate loops which iterate through block-decomposed arrays and strip mine so that distribution functions are invariant over inner loops.
3. `LOAD/STORE` **introduction** As described in Section 4, we also determine distributions for variables with floating distributions. Reductions (e.g., sum reductions) will be recognized and handled in special ways.
4. **Loop distribution** Apply loop distribution to allow block preloading of data. Scalar expansion can be applied to compiler temporaries introduced between `LOAD` and `STORE` instructions. The correctness conditions are the same as developed for distribution for use in vectorizing compilers [Allen and Kennedy 1987].
5. **Loop interchange** Attempt to interchange inner loops after strip mining to innermost positions in loop nest.
6. **Loop peeling** Peel off boundary iterations so that distribution functions are constant over the loop body.
7. **Loop reversal** Apply loop reversal to reorder sends before receives, where appropriate.
8. **Scalar optimizations on** `LOAD`s Use common subexpression elimination and loop invariant code motion to eliminate redundant `LOAD` statements.
9. `LOAD` **combining** Combine multiple `LOAD` statements with the same source and sink processors into single generalized loads. If startup costs are much larger than per-element costs for message passing, it may be profitable to move `LOAD` statements into regions of greater execution probability (e.g., from inside an if-then-else to above it) to combine it with a previous load.
10. `SEND-RECV` **introduction** Complete the basic compilation mechanism by realizing α functions as local variable references and introducing explicit `SEND` and `RECV` statements.
11. **Loop elimination** Use knowledge of particular loop distributions to replace iteration with direct computation of processor-specific loop iterations. This includes changing an explicit loop around a `LOAD` into an implied loop inside the `LOAD`. Loop unswitching [Allen and Cocke 1972] will be a potent transformation at this point. Loop elimination is the most important of the optimization steps. We are currently researching the limitations and effectiveness of loop elimination.
12. **Scalar optimizations on replicated variables** The computation of distribution functions for different variables is likely to have many common subexpressions, loop invariant expressions, and expressions which can be simplified by strength reductions. Remaining `DO`s may be reindexed to simplify subscripts.

While many of these transformations are complex, none are beyond the abilities of advanced program restructuring compilers. The result of applying by hand this sequence of transformations to the matrix multiply code of Figure 1 is shown in Figure 2.

Two-temporary variables, `ta` and `tb`, were introduced and the original computation loop became

```
      DO 30 I = 1, N
        DO 30 J = 1, N
          Store C(I, J) = 0.0
          DO 30 K = 1, N
            Load A(I, K), ta, Grid(I, J)
          Store C(I. J) = 0.0
          DO 30 K = 1, N
            Load A(I, K), ta, Grid(I, J)
            Load B(K, J), tb, Grid(I, J)
   30       Store C(I, J) = C(I, J) + ta*tb
```

These loops were then strip mined so that outer loops (using loop control variables `I1, J1`, and `K1`) iterate over blocks of `VC, VB`, and `VA` and inner loops iterate over elements within a block. Scalar expansion was applied to `ta` and `tb` and the two `LOAD` statements separated into distinct loop nests. Figure 2 shows the final versions of the

```
      REAL TA(256,4,256),TB(4,256,256)
      bs = FLOOR((N-1)/4) + 1               !  block size
      n1 = FLOOR((N-1)/bs) + 1
      g1 = FLOOR(THISPROC-1/4)+1            !  thisproc == grid(g1,g2)
      g2 = THISPROC - 4*g1
      i1 = g1
      DO 10 j1 = 1,n1
         IF (j1.EQ. g2) THEN                !  i am a receiver
            DO 11 k1 = 1,n1
               IF (k1.EQ.g2) THEN           !  local copy
                  DO 13 I = 1, MIN(bs*I1,N)-bs*i1+bs
                  DO 13 K = bs*k1-bs+1, MIN(bs*k1,n)
 13                  ta(I,j1,K) = a(I,k-bs*k1+bs)
               ELSE
                  DO 12 I = 1, MIN(bs*i1,N)-bs*i1+bs
                  DO 12 K = bs*k1-bs+1, MIN(bs*k1,N)
 12                  RECV(ONLYSRC=GRID(4,i1,k1)) ta(I,j1,K)
               ENDIF
 11            CONTINUE
         ELSE                               !  i am sender
            k1 = g2
            DO 14 I = 1, MIN(bs*i1,N)-bs*i1+bs
            DO 14 K = 1, MIN(bs*k1,N)-bs*k1+bs
 11            SEND (DEST=GRID(4,i1,j1)) a(I,K)
         ENDIF
 10      CONTINUE
      :                                     !  analagous loop for LOAD tb omitted.
      i1 = g1
      j1 = g2
      DO 30 I = 1, MIN(bs*i1,N)-bs*i1+bs
         DO 30 J = 1, MIN(bs*j1,N)-bs*j1+bs
            c(I,J) = 0.
            DO 30 K = 1,N
 30            c(I,J) = c(I,J) + ta(I,J1,K)*tb(I1,J,K)
```

Figure 2. Transformed program.

transfer loop (`DO 10`) derived from load of `ta` and the computation loop (`DO 30`). A distribution function for the expanded `ta` was chosen so that no data transfers are necessary during the execution of the `DO 30` loop. Note that the variable `ta` does not need to be expanded for all `J`, just for values causing different values of `GRID(I, J)`, which is what is shown in Figure 2.

The following variables have special significance in the final program:

`thisproc` is the processor-specific variable which distinguished each processor.
`bs` is the "width" of a block of `VC`.
`nl` is the last block the `I` and `J` loops iterate over.
`g1, g2` are variables derived from `thisproc` by inverting the distribution function for `VC`. These variables indicate which element of `VC` is "owned" by processor `thisproc`.
`i1, j1, k1` are the loop indices of the outer loops after strip mining. They correspond to traversing elements of `VC, VA` and `VB`.
`GRID(4, i1, j1)`is the distribution function associated with `VC`. The "4" is the row-stride of `VC` and is made explicit so no confusion arises between grids of different sizes.

All scalars in this example are replicated.

The `LOAD` statement for `ta` is now expanded to `SEND/RECV` statements and loop unswitching and loop interchange are applied. Finally the α functions are realized, replacing (for example), `C(I, J)` in `DO 30` with

```
    C(i - bs*i1 + bs, j - bs*j1 + bs)
```

and then the loops are reindexed to simplify subscripts. The resulting program is shown in Figure 2. The lowercase letters follow our earlier convention of distinguishing original shared-memory variables from compiler-generated local temporaries.

6. Related Work

As programming language design, this work is related to other work for distributed-memory computers. In particular, we share goals with C*, Linda, and DINO.

C*, developed for the Connection Machine [Rose and Steele 1987], is an SIMD-oriented language. Programmer's declare replicated variables and the C language is extended to manipulate these variables as aggregates. The language described here allows more general control flow on each node and more general distribution functions. In particular we allow clustering data into blocks. Both languages eliminate explicit message passing from the programmer's virtual machine.

Linda, developed at Yale [Carriero and Gelernter 1985], provides the illusion of shared data structures but forces the programmer to use special primitives to access these data structures and forces the programmer to explicitly program each node. The primary advantage our system has over Linda is that the programmer can specify where data elements are to be stored and the compiler can use this specification to improve the efficiency of the program.

DINO, developed the University of Colorado, Boulder [Rosing and Schnabel 1988], allows the programmer to specify distributed data structures and invoke replicated functions on these data structures. Programmers must mark variable references that indicate updates to the shared data structures and so forces the programmer to program the message passing. The language described here allows more general distribution functions and in particular we allow clustering data into blocks. It is possible that many of the optimizations described here can also be applied to DINO to improve communications efficiency and parallelism.

As a project on compiling for parallel computers, the work described here is similar to work on automatic parallelization [Padua and Wolfe 1986, Allen, Callahan, and Kennedy 1987, Allen et al. 1987] and strongly related to two other projects involving distributed arrays.

Koelbel, Mohrotra, and Rosendale [1987] are investigated translating BLAZE, an explicitly parallel language, into E-BLAZE, a language with low-level tasking exposed. Their work uses many of the same transformations described in Section 5. We have extended their approach with more general distribution and decomposition and mechanisms and identified the problems relevant to a distributed-memory multiprocessor.

Zima, Bast, and Gerndt [1988] are investigated translating Fortran for execution on the SUPRENUM multiprocessor. They implement a somewhat more general decomposition mechanism but not a separate distribution specification—each block corresponds to a processor. Developed independently, their basic compilation strategy is essentially the same as we have described in Section 4 and they have recognized the use of program transformation to eliminate loops and enhance parallelism. We extend their work by having a powerful means of expressing distribution functions and a more complete mechanism for implementing entire programs. Further, the introduction of `LOAD`s into the program is an important intermediate step because it provides a much more flexible framework for optimizing transformations.

7. Summary

We have described a language for programming distributed-memory computers which consists of a sequential, shared-memory base language extended with directives for specifying how program data elements are distributed onto processors. We have described a general mechanism for compiling programs in this language for distributed-memory multiprocessor computers. We have also outlined a sequence of program transformations to improve machine utilization.

We intend to implement the system described here as part of *ParaScope*, our parallel programming environment [Callahan et al. 1987], and target it toward an Intel iPSC hypercube. The resulting system should greatly enhance the usability of distributed-memory computers and provide a powerful environment for the development of parallel programs.

References

Allen, F. E., and Cocke, J. 1972. A catalogue of optimizing transformations. In *Design and Optimization of Compilers*, Prentice-Hall, Englewood Cliffs, N.J., pp. 1–30.

Allen, R., and Kennedy, K. 1987. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4): 491–542.

Allen, J. R., Callahan, D., and Kennedy, K. 1987. Automatic decomposition of scientific programs for parallel execution. In *Conference Record of the Fourteenth ACM Symposium on the Principles of Programming Languages* (Munich, West Germany, Jan.).

Allen, F., Burke, M., Charles, P., Cytron, R., and Ferrante, J. 1987. An overview of the PTRAN analysis system for multiprocessing. In *Proceedings of the First International Conference on Supercomputing*, Springer-Verlag, Athens, Greece.

Callahan, D., Cooper, K., Hood, R., Kennedy, K., Torczon, L., and Warren, S. 1987. Parallel programming support in ParaScope. In *Proceedings of the 1987 DFVLR Conference on Parallel Processing in Science and Engineering* (Koln, West Germany, June). Available as Rice University, Dept. of Computer Science Technical Report TR87-59.

Carriero, N., and Gelernter, D. 1985. *The S/Net's Linda Kernel.* Research Report YALEU/DCS/RR-383, Dept. of Computer Science, Yale University, Cambridge, Mass.

Cytron, R. 1986. Doacross: Beyond vectorization for multiprocessors. In *Proceedings of the 1986 International Conference on Parallel Processing.*

Fox, G. C. 1985. *A Review of Automatic Load Balancing and Decomposition Methods for the Hypercube.* Technical Report C3P-385, California Institute of Technology, Pasadena.

Fox, G. C. 1987. *Domain Decomposition in Distributed and Shared Memory Environments.* Technical Report C3P-392, California Institute of Technology, Pasadena.

Fox, G. C., and Otto, S. W. 1986. *Concurrent Computation and the Theory of Complex Systems.* Technical Report CALT-68-1343, California Institute of Technology, Pasadena.

Heath, M. T., ed. 1987. *Hypercube Multiprocessors 1987.* SIAM, Philadelphia.

Karp, A. H. 1987. Programming for parallelism. *Computer*, 20(5): 43–57.

Kennedy, K. 1980. *Automatic Translation of Fortran Programs to Vector Form.* Technical Report 476–029-4, Dept. of Mathematical Sciences, Rice University, Houston, Texas.

Koelbel, C., Mehrotra, P., and Rosendale, J. V. 1987. Semi-automatic domain decomposition in BLAZE. In *Proceedings of the 1987 International Conference on Parallel Processing*, S.K. Sahni, ed. (Aug.), pp. 521–524.

Kuck, D. J. 1978. *The Structure of Computers and Computation*, Vol. 1. Wiley, New York.

Kuck, D. J., Kuhn, R. H., Leasure, B., Padua, D. A. and Wolfe, M. 1983. Compiler transformation of dependence graphs. In *Conference Record of the Tenth ACM Symposium on the Principles of Programming Languages*, (Williamsburg, Va., Jan.).

Mandell, D. 1987. Experiences and results multitasking a hydrodynamics code on global and local memory machines. In *Proceedings of the 1987 International Conference on Parallel Processing*, S.K. Sahni, ed. (Aug.), pp. 415–420.

Padua, D. A., and Wolfe, M. J. 1986. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12): 1184–1201.

Rose, J., and Steele, G. 1987. *C*: An Extended C Language for Data Parallel Programming.* Technical Report PL87-5, Thinking Machines, Inc.

Rosing, M., and Schnabel, R. B., 1988. *An Overview of DINO—A New Language for Numerical Computation on Distributed Memory Multiprocessors.* Technical Report TR CU-CS-385-88, Dept. of Computer Science, University of Colorado, Boulder.

Wolfe, M. J. 1982. *Optimizing Supercompilers for Supercomputers.* Ph.D. thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign.

Zima, H., Bast, H., and Gerndt, M. 1988. Superb: A tool for semi-automatic MIMD/SIMD parallelization. In *Parallel Computing*, Vol. 6, North-Holland, Amsterdam, pp. 1–18.