Parallelization of Linearized Applications in Fortran D

Lorie M. Liebrock
Ken Kennedy

CRPC TR93342-S
November, 1993

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

This research was supported by: Center for Research on Parallel Computation, a National Science Foundation Science and Technology Center, through Cooperative Agreement No. CCR-9120008, NSF/NASA Agreement No. ASC-9213821, and ONR Agreement No. N00014-93-1-0158. Use of the Intel i860 was provided under a Texas CER Grant No. CISE 8619893.

Abstract

The Fortran D language extends Fortran by permitting the user to specify the distribution of array variables across the processors of a parallel computer system. This information can then be used by the compiler to derive a multidimensional parallelization. For programs in which multidimensional arrays have been linearized for optimal performance on vector processors, this strategy will not produce the best results because in those programs Fortran D is limited to a one-dimensional parallelization, which yields less efficient communication than multidimensional parallelization because of surface-to-volume effects. This paper proposes Fortran D extensions and associated compiler technology to support natural topology parallelization of applications' codes with linearized arrays. Experimental results are presented to illustrate the improvement in performance of this new approach over the performance of currently available methods such as Fortran D and the PARTI inspector/executor strategy.

1 Introduction

When vector processors came to dominate the supercomputer market in the late seventies and early eighties, vector length was the main factor in determining performance on most machines—longer vectors meant better performance, a rule that holds for these machines even today. To increase the length of vector operations, many programmers "linearized" multidimensional array operations in vector applications. This practice improved performance on vector processors at the cost of obscuring the function of the application code and, in many cases, making efficient parallelization more difficult.

To better support applications coded in this linearized style, we propose extensions to Fortran D and compiler technology to handle linearized arrays. Our extensions are based on the notion of problem topology. In particular, we will allow the user to map linearized arrays to parallel processors according to the logical topology of the problem (logical dimensions of the linearized arrays) rather than the single declared dimension of the linearized arrays. The knowledge of the logical dimensions is used during subscript analysis and communication generation to eliminate the need for runtime interpretation to carry out communication.

This work focuses on the so-called "regular applications," for which regular communication stencils exist. In a regular communication stencil, each element is directly dependent only on a fixed subset of neighbors in the problem topology. Hence, the compiler needs to recognize and support a limited set of linearized index computation patterns. Many of these applications use index arrays. To reduce (in many cases eliminate) reliance on run-time interpretation for index array processing, specifications of special-case index arrays are also introduced. These specifications significantly simplify communication analysis for the special-case arrays.

2 Motivation

Many applications' codes have been linearized to improve performance on vector processors. Some examples of these codes are UTCOMP and UTCHEM from the University of Texas, and reservoir simulation codes such as VIP-COMP, MORE and ECLIPSE [11]. Another such linearized code is KIVA, for chemically reacting flow simulation [9].

A global ocean model simulation code at Los Alamos National Laboratories has recently been ported from the Thinking Machines CM-2 to the Cray YMP. This porting was done by linearizing all of the three physical dimensions of the arrays and breaking the long resulting vectors over the eight processors. The researchers found that direct inline computation of linearized addresses was faster than using index arrays [8]. Linearization was a recommended technique for vectorization on the early vector processors such as the CDC Cyber 205 and the Hitachi-S9 with integrated array processor [15]. The Los Alamos porting experience indicates that linearization is still useful for modern vector computers.

Since the standard supercomputer today is a vector processor, the science and engineering community will encounter many linearized arrays when converting these codes to parallel systems. In the long term, the best approach would be to "delinearize" the application before attempting parallelization in Fortran D. This would permit the compiler system to use a natural multidimensional decomposition, with its performance advantages on a scalable parallel system. In addition, the user could maintain a single source image if there existed a Fortran D compiler that linearized multidimensional arrays when compiling to a vector machine (a fairly straightforward technical problem). Unfortunately, as Rame and Kremer learned while attempting to delinearize part of UTCOMP, this is not always an easy task. Often the functionality of the code is obscured during linearization, particularly if there is further development of the linearized code. Such was the case with UTCOMP[11]. This paper addresses the problem by providing a way to get the advantages of

delinearization without requiring that the programmer perform this tedious process by hand.

Users would like porting to parallel processors to be easy but the result must provide efficient execution to be acceptable. Some applications programmers are considering Fortran D or similar languages for this conversion. If all of the linearized address computations are done directly (via inline computation such as A(i)=f(A(i+ni))) then the code *can* be parallelized using a one dimensional data distribution in Fortran D. However, this might not provide the best performance as we shall illustrate with an example. It is well known [4, 12] that the computation to communication ratio is one of the most important numbers there is in determining the efficacy of a parallelization, so let's examine the relationship between parallelization topology and the computation to communication ratio for two parallelization topologies: the natural problem topology and a one dimensional topology.

Assume a standard computation/communication stencil in which any given element is dependent only on the element before it and the element after it in the each dimension (the two dimensional stencil is a five point stencil and the three dimensional stencil is a seven point stencil). Our discussion also applies to more general symmetric stencils.

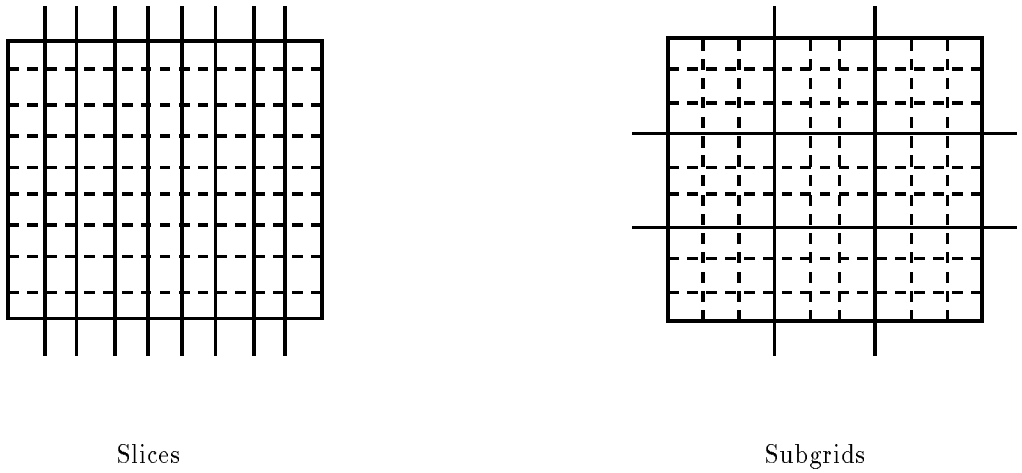


Figure 1: Mesh divided into slices versus subgrids for communication.

Consider a regular application with large linearized meshes that uses a standard computation/communication stencil. When parallelized in a one dimensional fashion, as required for regular support in the original Fortran D, the same size boundary must be exchanged no matter how many processors are used (with fewer processors than columns). If the same application is parallelized in the natural topology of the problem, the size of the boundaries to be exchanged decreases with increasing number of processors. See Figure 1 for a graphical illustration of this well known phenomenon. Since the communication per processor decreases as the computation per processor decreases instead of remaining fixed as it does in the one dimensional parallelization, the multidimensional parallelization is preferable. For a more precise treatment of this discussion of the merits of natural topology parallelization see Appendix A. The experimental results in Section 5 for both examples agree with these theoretical results.

Thus, we have established that using one-dimensional data distributions on a problem with a natural multidimensional topology is not a good idea. Unfortunately, there is no way in Fortran D to distribute a linearized array according to its natural topology without treating the computation as an irregular one, for which performance is necessarily worse. With the current definition of Fortran D, the computation can either be distributed one dimensionally and treated as a regular computation or distributed according to a user-defined function and processed via an inspector/executor approach, which requires that communication be determined at run time. Hence, you lose efficiency by not using the proper topology for distribution or you lose efficiency by paying the overhead associated with handling irregular problems.

We propose to overcome this problem by extending Fortran D to support parallelization according to the natural problem topology in the presence of linearized arrays. This extension involves two new statements. The LOGICAL DIMENSIONS statement provides information that will allow the Fortran D compiler to generate

a natural topology parallelization without the user having to "delinearize" the arrays in the application. In particular, the LOGICAL DIMENSIONS statement provides the information necessary to determine at compile time what communication is necessary when linearized array index computations are computed inline.

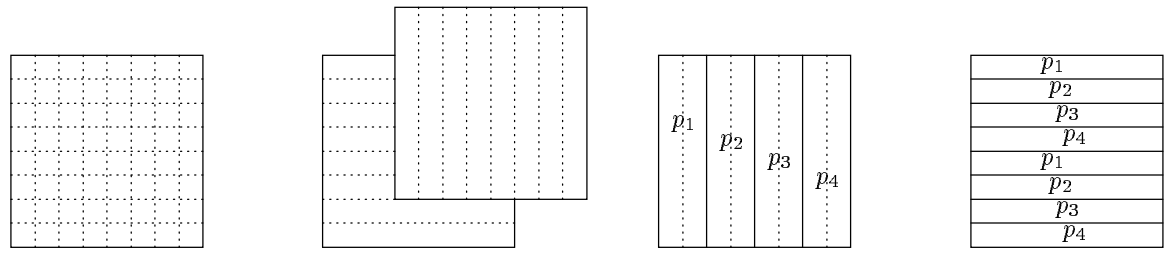
The second extension is proposed to help optimize accesses via the index arrays that are often used in conjunction with linearization. When index arrays are used, the Fortran D compiler cannot handle the code as regular. As the data reference pattern is not analyzable at compile time, the only support for this case is via an inspector/executor strategy [14, 2]. There are a number of problems with this approach. To begin with, current compilers cannot yet generate the inspector/executor automatically. Hence, this conversion must be done by hand. To do this, the user must modify declarations, determine new loop bounds are, figure out how each processor computes the local portion of the index arrays (in global terms) and finally insert calls to localize the index arrays and calls to perform communication. If double indirection is used then the user must be careful to gather the nonlocal value of the appropriate index array(s) before they are localized (including the newly gathered values). If values from index arrays are used in comparisons then the user must save a copy of the global version of the index array and replace all uses of index values in comparisons with uses of values from the global version. For most applications this is a lot work.

The good news is that a new version of the Fortran D compiler will soon be able to generate the inspector and executor automatically. This compiler may or may not be able to handle double indirection and logical comparisons of index values [13]. Even if the complete inspector/executor can be generated automatically, there is still extra execution time associated with the resulting parallelization. Communication requirements must be determined at runtime and communication satisfying those requirements must be generated. The overhead associated with determining the communication pattern is proportional to the number of index arrays present. Since the applications under consideration are actually regular calculations when considered in the natural topology, this is an unnecessary overhead.

To maximize the efficiency of execution of these linearized applications, we propose the INDEX ARRAY specification, which provides an upper bound on the communication necessary for correct execution. Experimental results in Section 5.4 for Example 2 compare the performance of using this new approach to using a block-structured version of the PARTI inspector/executor approach.

3 The Fortran D Language and Extensions

Fortran D supports both *alignment* and *distribution* specifications. A DECOMPOSITION statement specifies an abstract problem or index domain. An ALIGN statement maps array elements onto one or more elements of a decomposition. This provides the minimal information necessary for reducing data movement for the program, given an unlimited number of processors. A DISTRIBUTE statement groups decomposition elements and maps them to the finite resources of the physical machine. Sample data alignment and distributions are shown in Figure 2.




```
DECOMPOSITION          REAL A(N,N)          DISTRIBUTE          DISTRIBUTE
    D(N,N)             ALIGN A(I,J)           D(:,BLOCK)          D(CYCLIC,:)
                        with D(J-2,I+3)
```

Figure 2: Fortran D Data Decomposition Specifications

Fortran D also supports irregular data distributions and dynamic data decomposition, i.e., changing the alignment to or distribution of a decomposition at any point in the program. To permit a modular programming style, the effects of data decomposition specifications are limited to the scope of the enclosing procedure. However, procedures do inherit the decompositions of their callers. The complete language is described in detail elsewhere [3].

Note that the original Fortran D language specification does not support alignment of linearized arrays according to their logical topology and that irregular data distributions require runtime support. This work makes it possible to more efficiently compile the class of linearized regular application codes.

In an attempt to provide better support for linearized applications, we extend Fortran D in two ways. Dimension specification provides the basis for loop bounds updates and communication generation. Index array specifications further simplify communication analysis and in many cases eliminate the need for runtime support.

3.1 Logical Dimension Specification

The logical dimensions of the linearized dimensions of an index array are provided in the LOGICAL DIMENSIONS statement. This allows the two dimensional array z to be aligned as if it were declared to be three dimensional, i.e.,

```
      real z(ni*nj,np)
C     logical dimensions z((ni,nj),np)
C     align z(i,j,*) with d(j,i).
```

In this example (ni, nj) are the logical dimensions for the first declared dimension of the array z. The linearized array is then indexed according to the logical dimensions in the ALIGN statement. The array is still indexed elsewhere according to its declaration. Hence, the program body does not have to be rewritten to parallelize the application according to its natural topology.

For a more complete example, the logical layout and distribution of x from Figure 3 is shown in Figure 4, where $nx = ny = 8$.

```
      function a()
      parameter ($nprocs = 4)
      real x(nx*ny), y(nx*ny)
C     decomposition d(nx,ny)
C     logical dimensions x((nx,ny)), y((ny,nx))
C     align x with d
C     align y(i,j) with d(j,i)
C     distribute d(block,block)
      ...
      return
```

Figure 3: Alignment Statements for Linearized Arrays.

1	9	17	25	33	41	49	57
2	10	18	26	34	42	50	58
3	11	19	27	35	43	51	59
4	12	20	28	36	44	52	60
5	13	21	29	37	45	53	61
6	14	22	30	38	46	54	62
7	15	23	31	39	47	55	63
8	16	24	32	40	48	56	64

Figure 4: Distribution of 64 element (logical 8x8) array x.

Once a linearized array has been aligned to a DECOMPOSITION of the correct dimensionality, the DECOMPOSITION can be distributed as is most appropriate for the problem. This allows greater flexibility in

distribution. For example, in the code fragment of Figure 3, the (block,block) distribution of array x and the transposed (block,block) distribution of array y cannot be accomplished in the currently proposed version of High Performance Fortran or as a regular computation in Fortran D [3].

3.2 Index Array Specification

Linearized arrays are typically indexed in one of two ways. The most straightforward indexing method is direct, inline computation of linearized indices according to Fortran's standard array storage allocation. The most often used indexing approach is to use an index array to store the index of the neighbor, e.g.,

$$\text{west}(12) = 4$$

in array x of Figure 4.

Index array specifications are added to Fortran D as part of the support mechanism for linearized arrays.

```
      parameter (ni = 10, nj = 10)
      real A(ni*nj)
      integer ind(ni*nj,6)
C     decomposition d(ni,nj)
C     logical dimensions A((ni,nj)), ind ((ni,nj),6)
C     index array 2D ind [(i,1),1,-1], [(i,2),1,0],
C                        [(i,3),1,+1], [(i,4),2,-1],
C                        [(i,5),2,+1], [(i,5),2,0],
C                        [(i,6),1,+1,2,+1]
C     align A with d
C     align ind(i,j,k) with d(i,j,*)
C     distribute d(block,block)
      ...
      return
```

Figure 5: Logical Topology Specification.

The code in Figure 5 specifies that in the logical topology: $A(ind(i,1))$ is the element of A north of element $A(i)$, $A(ind(i,2))$ is the element of A at position i, $A(ind(i,3))$ is the element of A south of element $A(i)$, and $A(ind(i,4))$ is the element of A west of element $A(i)$, $A(ind(i,5))$ is either the element of A at position i or the element of A east of element $A(i)$. This multiple use of $ind(i,5)$ is allowed for applications such as UTCOMP which has an index array where the element pointed to depends on the value in another array, but it is always one of a small set of elements. With this type of specification such conditional indirections are supported. $A(ind(i,6))$ is the element of A south and east of the element of A at position i.

In general, for a specification $[(i,c), d_1, o_1, d_2, o_2, ...]$ the i indicates the index location to consider where d_i has an offset of o_i. Since $ind(i,4)$ is one entry left of center in the second dimension, $A(ind(i,4))$ refers to $A(i-ni)$ in the linearized array where ni is the number of elements in the first dimension. The syntax of index array specifications may be modified as applications with more general communication stencils are studied.

4 Compilation

In this section we will begin with an outline of the strategy used to compile linearized applications. After that we illustrate the compilation strategy with an example. Note that the constants are actual sizes, not just declared sizes.

In this paper we assume that all of the arrays in a given loop are similarly linearized, aligned (mapped), and distributed. The first example deals with the use of directly computed indices for linearized arrays. The second example deals with index arrays. Finally, only unit (+1 or -1) loop step sizes and step sizes that are multiples of the number of elements in the first dimension are allowed. Relaxation of these restrictions will be discussed after the basic compilation strategy is presented.

The examples used for illustration in this paper come from UTCOMP. UTCOMP is a University of Texas code that simulates three dimensional miscible gas displacements.

4.1 Strategy

As in the example of Figure 6, all logical dimension specifications for linearized arrays should use variables that are compile time constants or input runtime constants. These annotations could then be used for index analysis to support communication generation.

```
      program main
      parameter (nip=1, njp=1, n$proc=nip*njp)
      parameter (nim = 1024, njm = 244, nijm = nim * njm)
      real x(nijm), y(nijm), c(nijm), n(nijm), s(nijm), e(nijm), w(nijm)
C     decomposition d(ni,nj)
C     logical dimensions x((ni,nj)),y((ni,nj)),c((ni,nj)),n((ni,nj)),s((ni,nj)),e((ni,nj)),w((ni,nj))
C     align x, y, n, s, e, w with d
C     distribute d(block,block)
      read(*,*)ni,nj
      nij=ni*nj
      call compute(ni, nj, nij, x, y, c, n, e, w)
      return

      subroutine compute(ni, nj, nij, x, y, c, n, s, e, w)
      parameter (nim = 1024, njm = 244, nijm = nim * njm)
      real x(nijm), y(nijm), c(nijm), n(nijm), s(nijm), e(nijm), w(nijm)
C     align x, y, n, s, e, w with d
C     distribute d(block,block)
      do i = ni+1,nij-ni
         y(i) = x(i)*c(i) + x(i-1)*n(i) + x(i+1)*s(i) + x(i-ni)*e(i) + x(i+ni)*w(i)
      enddo
      return
```

Figure 6: Example 1: Linearized Array Computation.

To support direct computation of linearized indices, such as via a function call, the compiler would need to be capable of evaluating the index computation to determine ownership of data for communication. The logical dimension specifications support this analysis. This would eliminate the need to execute the inspector. For more details on how such index computations may be evaluated by the compiler see [6]. Once such computations are recognized, the compiler can determine whether the indirection implies a communication from the distribution in effect. Communication can be generated at this point because the indices of the source and sink of the dependence are known, as are the processors that store the values. In the cases where communication is necessary, the compiler can generate and insert communication for getting the correct element(s) of the linearized array.

To support the use of index arrays, the index array specification supplies the information necessary for communication generation.

To support other forms of indexing for linearized arrays, a runtime inspector could be used to ensure the correctness of communication.

Other considerations for compilation include resizing arrays in declarations and updating loop bounds based on the processor id. Loop bound computations are more complicated than for non-linearized computations as each processor has a set of linearized array indices. This corresponds to one contiguous block of the logical array but not a contiguous block of the linearized array.

4.2 Illustration of Strategy

We illustrate the compilation strategy with an example. Figure 6 shows the Fortran D source code for a linearized computation similar to some of the direct linearized address computation coding used in UTCOMP. This example is a two-dimensional code similar to a part of one of the three-dimensional routines from UTCOMP.

For good performance in an application implementation, the user or an automatic alignment and distribution system [7, 1] would align each array to minimize communication. In the example of Figure 6, all of the arrays are perfectly aligned with decomposition D.

Code generation for this simple code is outlined in Figure 7. This outline serves as a roadmap for the code, shown in Figure 8, that was generated via hand compilation. A setup routine was called in the main

```
modify declarations
compute loop bounds for each processor
exchange standard pattern phantom boundaries
exchange specialized phantom boundaries
modify loop
    use new loop bounds
    put correct north/south neighbors in temporaries
    modify computation to use temporaries
```

Figure 7: Example 1: Linearized Array Computation Roadmap.

```
      subroutine compute(x, y, c, n, s, e, w)
C     modify declarations
      parameter (nip=1, njp=1, n$proc=nip*njp)
      parameter (ni = 1024, nj = 244, nij = ni * nj)
      parameter (n$dip=ni/nip+1, n$djp=nj/njp+1)
      parameter (n$dij=n$dip*n$djp)
      parameter (n$sp=-(n$dip+n$djp))
      real x(n$sp:n$dij-n$sp), y(n$dij), c(n$dij),
     *   n(n$dij), s(n$dij), e(n$dij), w(n$dij)
      integer lb$1, ub$1
      common /buf$s/ buf$1(-n$sp), buf$2(-n$sp),
     *   buf$3(-n$sp),buf$4(-n$sp)
      common /proc$c/ my$proc,my$col,my$row,
     *   in$p,is$p,ie$p,iw$p,ine$p,inw$p,ise$p,isw$p,
     *   n$ip,n$jp,n$di,n$dj,n$i,n$j,in$st,in$nd,
     *   is$st,is$nd,ie$st,ie$nd,iw$st,iw$nd,n$s,n$f
C     compute loop bounds for each processor
      lb$1 = i$bnd2(ni+1,1,1)
      ub$1 = i$bnd2(nij-ni,-1,-1)
C     exchange standard pattern phantom boundaries
      call exc$b1(x,n$sp,n$dij-n$sp,101)
C     exchange specialized phantom boundaries
      if (n$jp .gt. 1) then
          if (my$row .eq. 1) then
              call b$f_dt2(x(1),2,n$di,1,1,buf$1)
              call csend(105,buf$1,8*(n$dj-1),
     *                   nip-1+nip*(my$col-1),my$pid)
              if (my$col .gt. 1) then
                  call csend(106,x,8,n$ip-1+n$ip
     *                       *(my$col-1),my$pid)
              endif
              call crecv(107,x(in$st+1),8*(n$dj-1))
              if (my$col .gt. 1) then
                  call crecv(108,x(in$st),8)
              endif
          elseif (my$row .eq. n$ip) then
              call b$f_dt2(x(1),n$di,n$di,
     *            1,n$dj-1,buf$1)
              call csend(107,buf$1,8*(n$dj-1),
     *                   1-1+n$ip*(my$col-1),my$pid)
              if (my$col .lt. n$jp) then
                  call csend(108,x(n$di*n$dj),8,1-1
     *                       +n$ip*(n$jp+1-1),my$pid)
              endif

              call crecv(105,x(is$st),8*(n$dj-1))
              if (my$col .lt. n$jp) then
                  call crecv(106,x(is$nd),8)
              endif
          endif
      endif
C     modify loop
C       use new loop bounds
      do i = lb$1, ub$1
C       put correct north/south neighbors in temporaries
          if (1 .eq. mod(i,n$di)) then
              xim$1 = x(in$st+i$n-1)
          else
              xim$1 = x(i-1)
          endif
          if (0 .eq. mod(i,n$di)) then
              xip$1 = x(is$st+i$n-1)
          else
              xip$1 = x(i+1)
          endif
C       modify computation to use temporaries
          y(i) = x(i) * c(i) + xi$1 * n(i) + xip1 * s(i) +
     *           x(i - n$di) * e(i) + x(i + n$di) * w(i)
      enddo
      return
      end
```

Figure 8: Example 1: Linearized Array Computation.

program to initialize the variables in the common block *proc$c*.

The function *b$f_dt2*:

call b$f_dt2(x(1),il,ih,jl,jh,buf)

copies from the *n$di* by *n$dj* array x the subarray $x(il : ih, jl : jh)$ into buf.

A single function is written to compute lower and upper bounds of loops based on linearized two-dimensional arrays distributed in two dimensions. First, the bounds computation function determines the location, in the logical two-dimensional array, of the input bound. Then the first location that the calling processor owns in the direction that the loop iterates is determined via the known mapping of elements to

processors. This same basic procedure can be followed for linearized n-dimensional arrays distributed in n dimensions.

One of the disadvantages of linearization of arrays is that communication overlap arrays are more complicated. For efficiency, starting and ending indices of overlaps are computed in the setup routine then used throughout the program to access boundary values. One advantage to this approach is that, with careful packaging during buffering for sends, most communications do not require unpackaging for receives. Boundary messages can be received directly into their linearized storage.

4.3 Details of Compilation

There are three stages in the compilation process: analysis, interprocedural propagation, and code generation. We examine each of these stages in detail. We will discuss each stage in terms of the needed modifications/additions to the Rice Fortran D compiler. For a discussion of the original Fortran D compiler see reference [5].

An outline of the compilation process is shown in Figure 9.

Analysis
- collect index information
- note potential index/indirection arrays
- note potential multiple indirections
- store logical dimensions of linearized arrays
- store index array specifications

Propagation
- propagate specifications down the call chain

Code Generation
- modify declarations
- insert call to setup routine
- modify initialization of index arrays
- update loop step sizes and loop bounds
- modify direct linearized address computations
- modify logical statements using values of index arrays
- generate communication

Figure 9: Compilation Process Outline

4.4 Analysis

Index information is collected on a per loop basis for later communication generation. We must also collect information about the use of potential index/indirection arrays. Multiple indirection candidates must be noted in the symbol table so that boundary indices can be computed if necessary.

Specifications for logical dimensions of arrays and specification of index arrays are collected along with the alignment and distribution information collected in Rice Fortran D compiler. Actual dimension size names are collected for each logically distributed dimension of the linearized arrays. This information is available directly from the LOGICAL DIMENSIONS statements in symbolic form.

Logical tests involving values from potential index arrays are marked for later processing.

4.5 Propagation

All of the specification information collected during analysis (e.g., logical dimensions of arrays, index arrays, and dimension sizes) is propagated down the call chain.

4.6 Code Generation

In each routine, array declarations for all distributed linearized arrays are modified to be the largest size needed by any processor plus the size needed for storing the phantom boundary values.

Immediately after the initialization of the actual dimension sizes (assignment or input), a call to a setup routine with the sizes as parameters is inserted. The setup routine identifies the processor, its location in the logical processor array, the processor identifiers of all logical neighbors in the logical processor array and the offsets for storage of boundary values.

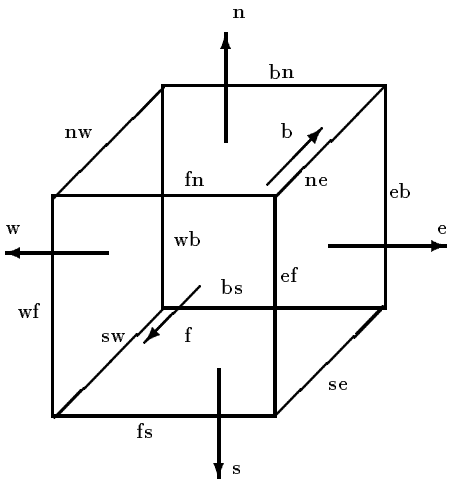


Figure 10: Three Dimensional Processor Layout

In Figure 10, we indicate the neighboring processors in a three dimensional logical processor array for a seven point stencil application. The diagram also indicates what boundary allocation is necessary. For each of the boundaries, starting and ending allocation addresses are computed in the setup routine. These addresses are then used to simplify and improve the efficiency of communication.

If index arrays are used, then their computation must be modified so that boundary addresses are used. This is straightforward when there is at most one indirection in any data access and there are only references to indices of the form i-1, i, or i+1 in any dimension. Otherwise multiple layers of phantom boundaries must be supplied. When double indirection is used, the boundary elements of the index array must be computed and filled in. Fortunately, the more complicated cases having many compound indirections and/or distant elements as neighbors are not the norm for these regular computations. Our second example (the routine called "disper" from UTCOMP) does use one form of double indirection. Note that the double indirection discussion implies we have relaxed the "owner computes rule" for index arrays.

Loop modification begins with step and bounds updates. Consider the simplest case, where the step is 1. To find a processor's lower bound we must find its first index (in the original array indexing) greater than or equal to the original lower bound and translate that index into the (smaller) processor's array index space. Since the mapping of elements to processors is known and fixed this not difficult. Let δr and δc be the number of rows and columns that a processor owns. Basically, in the two dimensional case, to find the lower bound on any processor, assuming the step size is positive, you perform the following:

- Determine the row and column of the bound in the original array.
- If the bound's column is greater than the last column the processor owns a portion of, then set the *lbr* to $\delta r + 1$ and the *lbc* to $\delta c + 1$.
- Elseif the bound's column is less than the first column the processor owns a portion of, then set the *lbr* to 1 and the *lbc* to 1.
- Else
  - Set the *lbc* to the bound's column minus the index of the first column the processor owns a portion of plus one.
  - If the bound's row is greater than the last row that the processor owns a portion of then add 1 to *lbc* and set the *lbr* to 1.
  - Elseif the bound's row is less than the first row that the processor owns a portion of then set the *lbr* to 1.

- Else set the *lbr* to the bound's row minus the index of the first row that the processor own's a portion of plus one.

- Compute the new lower bound as $(lbc - 1) * \delta c + lbr$.

The process for the upper bound is similar. Indeed we write one routine and pass $-1 * sign(step)$ for the step to compute the upper bound. Non unit steps complicate the computation. Fortunately, typical non-unit steps are fairly straightforward. In particular, probably the most common non-unit step is the size of the first dimension. In this case, we find the bounds as above but the step size is modified to be the number of elements in the first dimension of the subarray owned by the processor. In the completely general case, a different starting position could be needed for each logical column of the submatrix owned by each processor. After bounds have been computed we must consider the body of the loop. If linearized addresses are computed directly in the loop, as were seen in our first example, then a case structure must be built that moves a neighbor value from either normal neighbor storage or phantom boundary storage in the processor's subarray to a temporary. The temporary is then used in place of the variable with its direct linearized address computation. If index arrays are used, then we need not make this modification because we have already updated the values in the index array to handle the boundaries. This may give an execution time advantage to parallelization of linearized application that use index arrays for indirection rather than direct computation of linearized addresses.

Logical statments that involve tests of actual index array values require modification. An example of such a complication is:

```
iwst = ind(i,4)
.
.
.
if (iwst .ne. 0)  A(i) = f(A(iwst))
```

where 0 is out of bounds for A. Because A(iwst) may now be an internal boundary value, this computation must be modified to something like

```
iwst = ind(i,4)
.
.
.
if (iwst .ne. A$0)  A(i) = f(A(iwst))
```

where A\$0 is a special (invalid) index value which is also used during initialization in place of zeros. If such uses can not be deciphered at compile time, runtime checks must be added for correctness or the user must supply further information.

The final code modification involves communication generation. With direct linearized address computation, communication generation involves a simple modification to the Rice Fortran D compiler. The only difference is in recognizing and using the higher dimensional nature of the arrays and their indexing. For example, recognition of a reference to A(i-ni) as a reference to the element to the west of element i in the logically two (or higher) dimensional array A implies communication is only needed in a west to east pattern along the boundary faces in the processor array.

With the use of index arrays, communication generation is even simpler as a reference to A(ind(i,4)) is specified to be a reference to the element to the west of A(i). Hence the index array specifications provide an upper bound on the communication needed for an indirection through an index array. This is an upper bound as, recall that, we allow entries such as

```
(ind2(i),.,ind2(i))
```

which indicates that ind2(i) refers to either element $i - 1$ or element $i + 1$ in the appropriate dimension. In the cases where such multiple entries occur, communication must be generated at compile time to satisfy all entries or runtime support must be provided to determine which communication is necessary and perform it. In most regular applications there will only be one communication direction associated with an index array usage. Communication optimization is essentially the same as that needed in any good Fortran D compiler.

4.7 Relaxation of Restrictions

When we started, we placed certain restrictions on the codes we would consider compiling. Here we discuss how to handle codes which do not adhere to those restrictions.

4.7.1 Using Arrays with Different Linearizations

If different arrays have been linearized in different ways we must accommodate this multiplicity. One way to handle this is to call a setup routine for each type of linearization and allocate common blocks for each. Then in subsequent bounds computations and communication we select information from the appropriate common block. For communication between arrays with different linearizations, we must perform essentially the same analysis as is done in the Rice Fortran D compiler for arrays with different distributions.

4.7.2 Arbitrary Step Sizes

To support arbitrary step sizes we must first compute the step size for the subblocks. Then a starting and ending position must be computed for each subcolumn on each processor. Finally, the loop must be modified to iterate over the columns with a nested loop that iterates from that subcolumn's starting to ending positions by the subblock step size.

4.8 Optimization for Linearized Array Computations

Sometimes linearization for vectorization introduces extraneous computations. This is usually due to computing boundary values in the same manner (as part of the vector computation) as interior points. Then the boundary conditions are recomputed correctly after the vector-based computation. Typically, this also implies non-productive communication if the code is parallelized based on the linearized code. A good compiler should recognize that the boundary values are stored twice but not used between the stores. Once this has been recognized the compiler can eliminate the useless first computation. The compiler should then recognize that the communication associated with the first computation is also superfluous and may be eliminated as well. Even if the extraneous computations are not eliminated, the communication may still be eliminated and maintain correctness (with proper initialization). A good Fortran D compiler should be capable of performing such analysis/optimization.

In the example of Figure 6 only the interior points loop has been shown. In the complete implementation there would be boundary updates following the loop shown. The computations for elements 9, 16, 17, etc. would all be recomputed as boundary values without ever having been used. Hence their computation in this loop is superfluous.

5 Experimental Performance Results

In order to be able to say something about the actual performance achievable when using the techniques described in this paper, two example applications were hand translated. The codes were then compiled with release 3.3 of the Intel iPSC/860 Fortran compiler.

Timing experiments were run on the Intel iPSC/860 at Rice University. The machine has 32 processors, each with 8Mbytes of memory and a vector processor. Here we present the actual run times for both of the example application codes. All of the times presented are the minimum of at least three executions. Timings are in seconds and are rounded to five significant digits.

5.1 2D Direct Linearized Address Computation Application

The original (sequential linearized) compute routine for this example is shown in Figure 6. This application was parallelized in three ways. First, the approach described in this paper was used to achieve the natural two dimensional parallelization. Next, the communication of this two dimensional parallelization was optimized, as discussed in Section 4.8. Finally, the Fortran D compiler was used to generate a linear (one dimensional) parallelization, the only type of parallelization possible with the original definition of Fortran D. In all cases, the meshes (each of which was 1024x244) was divided evenly across the processors.

Upon computing efficiencies for this example, it was found that the new parallel code ran at better than 100 percent efficiency. Some time was spent tracking down the cause of this anomaly. We found that the actual runtime of the code generated for this application is highly dependent on the precise memory allocation used. The declarations were changed (and nothing else) in the original program to be the same as those in the parallel program. The original sequential code ran in 1.0050 seconds while the "reallocated" code ran in 0.53127 seconds. This illustrates how much difference memory allocation can make on very small simple programs. On large programs with many data structures, changes in memory allocation do not appear to make significant differences in runtime. For all speedup comparisons the timings of the "reallocated" code are used. Hence comparisons are made against the "best" known sequential code.

The timings for the first two parallelizations are shown in Table 1. These timings provide an example of the type of performance improvement possible when the communication optimization procedure in Section 4.8

is performed on a small simple application.

Num. Procs	Unoptimized	Optimized	unopt./opt.
2	2.5932E-01	2.4425E-01	1.06
4	1.2532E-01	1.1585E-01	1.08
8	6.4001E-02	5.9694E-02	1.07
16	4.0701E-02	3.1243E-02	1.30
32	3.7446E-02	1.6700E-02	2.24

Table 1: Communication Optimization Timing Comparison

In the following comparisons with Fortran D, the optimized code is used. Table 2 presents runtimes and speedup factors comparing against the runtime of the reallocated sequential code. These results illustrate that the techniques described here are capable of providing an improvement in performance even for such a simple computation as Example 1.

Let's consider the superlinear speedups in Table 2. The Intel i860 we are using has a 2 way, set associative, 8Kbyte cache with a write-back policy. This cache is the cause of the superlinear speedups. In this trivial example, there are only a few arrays and a very simple memory access pattern. The cache makes a tremendous difference for this application. To illustrate this the sequential program was run on the same size problem that was run on each of the parallel processors. Since the original problem had 1024x244 meshes, size (1024x244)/P meshes for P = 2, 4, 8, 16, and 32 were used. Table 3 illustrates runtime efficiencies with cache effect factored out.

Since it appears that there could be a memory allocation imbalance between the Fortran D code and the logically two dimensional code, it can not be stated categorically that this procedure outperforms Fortran D on this application. It can be said that an old (a preliminary compilation procedure was used to generate it) version of the code, with similar memory allocation performance to the Fortran D code, also outperformed the Fortran D code. Now let's turn to a more realistic application for a better look at performance results.

5.2 3D Index Array Application

Our second example consists of three routines from UTCOMP. Two of the routines are index array computations, one of which must be modified to support double indirection usage. The third routine, *disper*, which computes the dispersion term for UTCOMP, is over one thousand lines long, uses index arrays including double indirection and is representative of the code throughout UTCOMP [10].

This code was not parallelized using standard Fortran D as that would require runtime support. The requirement for runtime support is due to the indirections that use index arrays. This support is not available in the current version of the Rice Fortran D project. Hence we will compare with the results of a hand parallelization using the inspector/executor approach for regular problems supported by the PARTI routines. This parallelization will be discussed first, in Section 5.3, then we will return to a discussion of the performance results in Section 5.4.

	1D Fortran D		Logical 2D	
Num. Procs.	runtime	speedup	runtime	speedup
2	5.5696E-01	0.95	2.4425E-01	2.18
4	2.7976E-01	1.90	1.1585E-01	4.59
8	1.2224E-01	4.35	5.9694E-02	8.90
16	4.7520E-02	11.18	3.1243E-02	17.00
32	2.3309E-02	22.79	1.6700E-02	31.81

Table 2: Example 1 Timing Results

P	Sequential runtime	1D Fortran D runtime	speedup	efficiency	Logical 2D runtime	speedup	efficiency
2	1.7928E-01	5.5696E-01	0.64	32.2	2.4425E-01	1.47	73.4
4	8.5883E-02	2.7976E-01	1.23	30.1	1.1585E-01	2.97	74.1
8	4.3315E-02	1.2224E-01	2.83	35.4	5.9694E-02	5.80	72.6
16	2.1925E-02	4.7520E-02	7.38	46.1	3.12430E-02	11.23	70.2
32	1.0182E-02	2.3309E-02	13.98	43.7	1.6700E-02	19.51	61.0

Table 3: Example 1 Cache Effect Timing Results (Mesh size: (1024x244)/P)

5.3 PARTI Parallelization of 3D Index Array Application

In the original treatment of applications using the PARTI routines, the data element to processor mapping was determined by having the user set up an array on each processor with global indices of the elements the processor owned. The inspector then built a distributed translation table that described the mapping of global indices to processors and the offsets in the array. This approach is still used for irregular problems that do not have standard distributions. For irregular problems with standard distributions, the indices are now "dereferenced" via a function. The elimination of the distributed translation table provides up to a factor of five improvement in runtime of the inspector [13]. Standard distributions (supplied as part of PARTI) are BLOCK and CYCLIC. Note that the BLOCK distribution will provide the linear distribution that has been discussed. The user may also modify the PARTI routines to support other regular distributions. To begin with we will only discuss the application modifications necessary to use the standard distributions.

First, consider the inspector. This is a new addition to the application code that must be written to support the initialization of the PARTI data structures. For simple programs, using only single indirections, it is only necessary to call a routine that localizes index array values (convert the global array indices to indices for the specific processor) and constructs a communication schedule for the index array(s). This is very easy to do, but Example 2 uses double indirection. When double indirection is used, the inspector is more complicated. For a double indirection like

$$\mathrm{A}(\alpha(\beta(i)))$$

(where α and β are index arrays), after β is localized and a schedule for β-induced communication has been computed, the off processor values of α that will be referenced are collected (via a call to a gather routine). Next, localization and communication schedule generation is performed for α including the gathered elements. But our application is still not quite done. Example 2 also uses the values of index arrays in comparisons. To handle this properly we must keep a copy of the index arrays (before they were localized) to use in the comparisons.

Now we can consider what needs to be done in the executor portion of the code. Declarations must be modified so that storage is declared only for the portion of each array that a processors owns. Index array computations must be modified so that each processor computes only their set of index values (in terms of the global index space). The program must be analyzed and calls inserted at the appropriate places to perform communication. Loop bounds must be correctly computed for each processor based on the portion of the arrays that the processor owns. Logical comparisons must be modified to use the values of the global index arrays (recall that they were saved by the inspector).

Finally, the application is ready to run using a PARTI supplied regular distribution. In the comparisons of Section 5.4 we use the standard block distribution and refer to this version of the code as "PARTI-1D".

In order to run the application using its natural topology parallelization in PARTI, the user must modify one of the PARTI routines. In particular, the user must write code that when given a vector of indices in the global index space will generate vectors of owning processor numbers and offsets. The routine was modified to support a three dimensional distribution of the linearized arrays. In the comparisons of Section 5.4 we will refer to this code as "PARTI-3D".

We would like to point out that work is being done on a compiler to automatically generate inspector/executor programs for a restricted type of source program. The version of that compiler currently in progress may or may not be able to handle the double indirection and/or the logical comparisons with global index values [13].

Also note that we are not claiming that the approach we have presented can achieve all of the things that can be done via use of the PARTI routines. We are using the PARTI comparison because it is capable of handling the problems under consideration (as well as much more general, with more user work, problems).

5.4 3D Index Array Application Results

We now return to the runtime results for Example 2. The runtime for the original sequential code is 109.76 seconds.

	PARTI-1D		PARTI-3D		Logical 3D	
Num. Procs.	runtime	speedup	runtime	speedup	runtime	speedup
2	69.339	1.58	60.271	1.82	57.206	1.92
4	34.868	3.15	30.186	3.64	28.597	3.84
8	17.070	6.43	15.134	7.25	14.268	7.69
16	8.3085	13.21	7.5702	14.5	7.1471	15.36
32	4.2477	25.84	3.8696	28.36	3.6997	29.67

Table 4: Example 2 Timing Results

As expected, the natural topology parallelization using PARTI significantly outperforms the linear parallelization using PARTI. The results in Table 4 illustrate this.

Looking at Table 4, we wondered whether these are really very good speedups after cache effect observations made with the first example. An efficiency experiment was done to see what part of this apparent good performance was due to cache effect.

	Sequential	PARTI-1D		PARTI-3D		Logical 3D	
P	runtime	runtime	efficiency	runtime	efficiency	runtime	efficiency
2	54.106	69.339	78.0	60.271	89.8	57.206	94.6
4	26.654	34.867	76.4	30.186	88.3	28.597	93.2
8	12.935	17.070	75.8	15.134	85.5	14.268	90.7
16	6.2814	8.3085	75.6	7.5702	83.0	7.1471	87.9
32	3.0237	4.2477	71.2	3.8696	78.1	3.6997	81.7

Table 5: Example 2 Cache Effect Timing Results (Mesh size: (10x24x24)/P)

Table 5 compares the timings for the sequential program on the mesh sizes (10x24x24)/P for P = 2, 4, 8 16, and 32 to the timings for the parallel programs. This table shows that the time for running a problem of size (10x24x24)/P is very near 1/P times the time for running a problem of size 10x24x24. It is clear that there is much less cache effect in this application than was present in the first example.

From the actual timing results, we see that this new compilation approach slightly outperforms the results achieved by the best handcoded use of the PARTI routines. The difference in performance combined with the difference in the amount of work that the user must do makes this a preferable approach to parallelization of linearized applications.

6 Conclusions

Neither Fortran D nor High Performance Fortran provide regular support for the natural topology parallelization of linearized applications codes. We have proposed extensions to Fortran D that permit specification of the logical dimensions of linearized arrays and the use of index arrays to specify regular communication in linearized array references, along with an approach to compiling the resulting programs. Hand simulation of the compilation algorithm shows that runtime support is not necessary for most linearized applications even if indirection is used.

Better performance and, in comparison to the inspector/executor approach, reduced manual recoding effort make this new approach preferable to using the original Fortran D, HPF or inspector/executor approaches for regular linearized applications.

A Theoretical Results

Consider applications with standard computation/communication stencils. In each dimension of a mesh, any given element is dependent only on the element before it and the element after it in the given dimension, i.e., the two dimensional stencil is a five point stencil, the three dimensional stencil is a seven point stencil, etc. These results can be extended for other symmetric stencils.

First, let's prove a result for applications with large two dimensional meshes using many processors.

Theorem 1 *Given $p_i * p_j = P$ processors ($p_i, p_j >> 1$) and an application with m-by-n meshes ($m \leq n$)) that uses a standard two dimensional stencil such that $m >> \frac{n}{P}$, P divides m and n, and $\frac{m}{p_i} = \frac{n}{p_j}$, a subblock (two dimensional) partition of the problem onto the processors will yield less communication than a linear (one dimensional) partitioning.*

Proof: Let α be the number of computations per element and β be the number of communications (sends) per element in each direction of each dimension.

The minimum cuts possible with a linear (one dimensional) partitioning is $m(P-1)$ and is achieved by cutting into P slices of shape m-by-$\frac{n}{P}$. Since each cut requires two elements to perform sends in one direction of the dimension, the total communication is $C_{LP} = 2\beta m(P-1)$. The total computation is αmn. Hence the communication to computation ratio for linear (one dimensional) partitioning is:

$$\frac{2\beta m(P-1)}{\alpha mn} = \frac{2\beta(P-1)}{\alpha n}$$

The minimum cuts possible with a subblock (two dimensional) partitioning is $m(p_j - 1) + n(p_i - 1)$ and is achieved by cutting into P subblocks of shape $\frac{m}{p_i}$-by-$\frac{n}{p_j}$. Since each of the vertical cuts requires two elements to perform sends in one direction of the second dimension, the total vertical communication is $2\beta m(p_j - 1)$. Since each of the horizontal cuts requires two elements to perform sends in one direction of the first dimension, the total horizontal communication is $2\beta n(p_i - 1)$. Hence, the total communication is $C_{SP} = 2\beta(m(p_j - 1) + n(p_i - 1))$. The total computation is αmn. Hence the communication to computation ratio for subblock (two dimensional) partitioning is:

$$\frac{2\beta(m(p_j - 1) + n(p_i - 1))}{\alpha mn}$$

We then arrive at a communication ratio of

$$\frac{C_{LP}}{C_{SP}} = \frac{m(P-1)}{m(p_j - 1) + n(p_i - 1)}$$

which can be reduced to

$$\frac{C_{LP}}{C_{SP}} = \frac{p_i - \frac{1}{p_j}}{2 - \frac{1}{p_j} - \frac{1}{p_i}}. \tag{1}$$

Since both p_i and p_j are much larger than one, the communication to computation ratio is worse for the linear (one dimensional) partitioning than for the subblock (two dimensional) partioning. □

Although the result of Theorem 1 is quite general it may not be obvious to the casual reader just how bad the linear partitioning can be relative to the subblock partitioning. In order to make this clear consider Corollary 1.

Corollary 1 *If $m = n$ in Theorem 1 then the communication ratio is:*

$$\frac{C_{LP}}{C_{SP}} = \frac{p_i + 1}{2} = \frac{p_j + 1}{2} = \frac{\sqrt{P} + 1}{2}$$

Proof: Follows from Theorem 1.

Next, let's prove a similar result for the three dimensional case.

Theorem 2 *Given $p_i * p_j * p_k = P$ processors ($p_i, p_j, p_k >> 1$) and an application with l-by-m-by-n meshes ($l \leq m \leq n$)) that uses a standard three dimensional stencil such that P divides l and m and n, and $\frac{l}{p_i} = \frac{m}{p_j} = \frac{n}{p_k}$, a subblock (three dimensional) partition of the problem onto the processors will yield less communication than a linear partition.*

Proof: Let α be the number of computations per element and β be the number of communications (sends) per element in each direction of each dimension.

The minimum cuts possible with a linear partitioning is $lm(P-1)$ and is achieved by cutting into P slices of shape l-by-m-by-$\frac{n}{P}$. Since each cut requires two elements to perform sends in one direction of the dimension, the total communication is $C_{LP} = 2\beta lm(P-1)$. The total computation is αlmn. Hence the communication to computation ratio for linear partitioning is:

$$\frac{2\beta lm(P-1)}{\alpha lmn} = \frac{2\beta(P-1)}{\alpha n}$$

The minimum cuts possible with a subcube (three dimensional) partitioning is $mn(p_i - 1) + ln(p_j - 1) + lm(p_k - 1)$ and is achieved by cutting into P subcubes of shape $\frac{l}{p_i}$-by-$\frac{m}{p_j}$-by-$\frac{n}{p_k}$. Since each of the horizontal cuts requires two elements to perform sends in one direction of the first dimension, the total horizontal communication is $2\beta mn(p_i - 1)$. Since each of the vertical cuts requires two elements to perform sends in one direction of the second dimension, the total vertical communication is $2\beta ln(p_j - 1)$. Since each of the "level" cuts requires two elements to perform sends in one direction of the third dimension, the total "level" communication is $2\beta lm(p_k - 1)$. Hence, the total communication is $C_{SP} = 2\beta(mn(p_i - 1) + ln(p_j - 1) + lm(p_k - 1))$. The total computation is αlmn. Hence the communication to computation ratio for subcube partitioning is:

$$\frac{2\beta(mn(p_i - 1) + ln(p_j - 1) + lm(p_k - 1))}{\alpha lmn}$$

Then the communication ratio is

$$\frac{C_{LP}}{C_{SP}} = \frac{lm(P-1)}{(mn(p_i - 1) + ln(p_j - 1) + lm(p_k - 1))}$$

which can be reduced to

$$\frac{C_{LP}}{C_{SP}} = \frac{p_i(1 - \frac{1}{P})}{\frac{2 - \frac{1}{p_j} - \frac{1}{p_k}}{p_i} + \frac{p_i - 1}{p_j^2}} \tag{2}$$

Since the denominator is less than one and p_i is much larger than one, the communication to computation ratio is much worse for the linear (one dimensional) partitioning than for the subcube (three dimensional) partioning. □

Although the result of Theorem 2 is quite general it may not be obvious just how bad the linear partitioning can be relative to the subblock partitioning. In order to make this clear consider Corollary 2.

Corollary 2 *If $l = m = n$ in Theorem 2 then the communication ratio is:*

$$\frac{C_{LP}}{C_{SP}} = \frac{1 + P^{\frac{1}{3}} + P^{\frac{2}{3}}}{3}$$

Proof: Follows from Theorem 2.

Finally, a result for n-dimensional hypercube meshes will be proven.‘

Theorem 3 *For an n-dimensional mesh having s elements in each dimension using the standard n-dimensional stencil, the communication ratio is*

$$\frac{1 + P^{\frac{1}{n}} + P^{\frac{2}{n}} + ... + P^{\frac{n-1}{n}}}{n}$$

Proof: Let s be the length of the sides of the hypercube and β be the number of communications (sends) per element in each direction of each dimension.

If the hypercube is partitioned linearly, then the total area of the cuts is $s^{n-1}(P-1)$ yielding a total communication of $C_{LP} = 2\beta s^{n-1}(P-1)$.

If the hypercube is partitioned into $q^n = P$ subhypercubes, then the total area of the cuts is $ns^{n-1}(q-1) = ns^{n-1}(\sqrt[n]{P}-1)$ yielding a total communication of $C_{LP} = 2\beta ns^{n-1}(\sqrt[n]{P}-1)$.

Hence the communication ratio is

$$\begin{aligned}\frac{C_{LP}}{C_{SP}} &= \frac{2\beta s^{n-1}(P-1)}{2\beta ns^{n-1}(\sqrt[n]{P}-1)} \\ &= \frac{s^{n-1}(P-1)}{ns^{n-1}(\sqrt[n]{P}-1)} \\ &= \frac{1+P^{\frac{1}{n}}+P^{\frac{2}{n}}+\ldots+P^{\frac{n-1}{n}}}{n}\end{aligned}$$ □

Theorem 3 shows that as the dimensionality of the problem (and the number of processors) increases the natural topology parallelization has ever greater advantage over the linear parallelization.

For a concrete illustration, consider using 1024 processors in parallelization of our first example with meshes of size 1024x1024. The linear (one dimensional) parallelization would then have 1024 elements per processor (one column of the logically two dimensional meshes) with 2048 boundary values to send and receive on each "internal" processor. Note that the two outside (first and last) processors would only send 1024 boundary values, they would not even be computing any results. The subblock (two dimensional) parallelization would also have 1024 elements per processor, but they would form a 32x32 subblock with 128 boundary values to communicate on "internal" processors. External processors would be computing between 93 and 97 percent as much as internal ones. This example makes it clear that not only is communication per processor reduced by using the natural topology for parallelization but load balancing is also improved. Hence, the computation to communication ratio is improved by parallelizing according to the natural problem topology.

References

[1] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. An interactive environment for data partitioning and distribution. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[2] H. Berryman, J. Saltz, and J. Scroggs. Execution time support for adaptive scientific algorithms on distributed memory machines. ICASE Report 90-41, Institute for Computer Application in Science and Engineering, Hampton, VA, May 1990.

[3] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D language specification. Technical Report TR90-141, Dept. of Computer Science, Rice University, December 1990.

[4] G. Fox and S. Otto. Algorithms for concurrent processors. *Physics Today*, 37:50–59, May 1984.

[5] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler support for machine-independent parallel programming in Fortran D. Technical Report TR90-149, Dept. of Computer Science, Rice University, January 1991. To appear in J. Saltz and P. Mehrotra, editors, *Compilers and Runtime Software for Scalable Multiprocessors*, Elsevier, 1991.

[6] C. Koelbel. *Compiling Programs for Nonshared Memory Machines.* PhD thesis, Purdue University, West Lafayette, IN, August 1990.

[7] L.M. Liebrock, D.L. Hicks, K.W. Kennedy, and J.J. Dongarra. Using problem and algorithm topology in parallelization. Technical Report 91-166, Rice University, Center for Research in Parallel Computation, Houston, TX, August 1991.

[8] Olaf Lubeck. Computer Research and Development, Los Alamos National Laboratories, February 1993. Private communication.

[9] Vince Mousseau. EG&G Idaho, Idaho National Engineering Laboratories, March 1993. Private communication.

[10] Marcelo Rame. Computational and Applied Mathematics Department, Rice University, December 1992. Private communication: This code was supplied as representative of the computations in UTCOMP.

[11] Marcelo Rame. Computational and Applied Mathematics Department, Rice University, February 1993. Private communication.

[12] D. Reed, L. Adams, and M. Patrick. Stencils and problem partitioning: Their influence on performance of multiprocessor systems. *IEEE Transactions on Computers*, C-36(7):845–858, July 1987.

[13] Joel Saltz. Computer Science Department, University of Maryland, February 1993. Private communication.

[14] R. v. Hanxleden, K. Kennedy, C. Koelbel, R. Das, and J. Saltz. Compiler analysis for irregular problems in fortran d. In *Proceedings of the 5th Workshop on Languages and Compilers for Parallel Computing*, New Haven,

CT, August 1992.

[15] G. Wolfgang. *Notes on Numberical Fluid Mechanics, Volumne 8: Vectorization of Computer Programs with Applications to computational Fluid Dynamics*. Friedr. Vieweg and Sohn, Wiesbaden, Germany, 1984.