IMPLEMENTATION OF A FULLY-BALANCED PERIODIC TRIDIAGONAL SOLVER ON A PARALLEL DISTRIBUTED MEMORY ARCHITECTURE

T. M. Eidson
High Technology Corporation
Hampton, VA 23681-0001

G. Erlebacher
Institute for Computer Applications in Science and Engineering
NASA Langley Research Center, Hampton, VA 23681-0001

Abstract

While parallel computers offer significant computational performance, it is generally necessary to evaluate several programming strategies. Two programming strategies for a fairly common problem—a periodic tridiagonal solver—are developed and evaluated. Simple model calculations as well as timing results are presented to evaluate these strategies.

The particular tridiagonal solver evaluated is used in many computational fluid dynamic simulation codes. The feature that makes this algorithm unique is that these simulation codes usually require simultaneous solutions for multiple right-hand-sides (RHS) of the system of equations. Each RHS solutions is independent and thus can be computed in parallel. Thus, a Gaussian-elimination-type algorithm can be used in a parallel computation and more complicated approaches such as cyclic reduction are not required.

The two strategies are a transpose strategy and a distributed solver strategy. For the transpose strategy, the data is moved so that a subset of all the RHS problems is solved on each of the several processors. This usually requires significant data movement between processor memories across a network. The second strategy attempts to have the algorithm follow the data across processor boundaries in a chained manner. This usually requires significantly less data movement. An approach to accomplish this second strategy in a near-perfect load-balanced manner is developed. In addition, an algorithm will be shown to directly transform a sequential Gaussian-elimination-type algorithm into the parallel, chained, load-balanced algorithm.

*Research supported by the National Aeronautics and Space Administration under contract No. NAS1-18605 while resident at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA 23665. The first author was supported by contract ...

1 Characteristics of parallel, distributed memory computing related to CFD simulation codes

Parallel computing offers significantly increased performance for solving numerical problems [3, 4]. On distributed memory architectures a data parallel programming strategy is frequently used as a relatively straightforward approach to attain this performance potential. The data parallel approach can be applied conveniently to computational problems which involve computations on several large arrays, particularly if a significant number of these array computations are local. By local, it is meant that the computations involve data elements with identical or neighboring indices from one or more arrays. With an appropriate data distribution strategy, the communication costs can be negligible to moderate for these situations. However, many problems of interest also contain significant global computations, i.e., computations of an array element which depend on other array elements (of the same array or similar shaped arrays) with non-local indices. Usually high communication costs result. Dealing with these global computations is usually the major programming problem when developing code for parallel computers. Because of the higher cost of internode over intranode communication the computational savings for a good programming strategy are significant. The objective of this paper is to discuss several programming strategies for a common algorithm, a tridiagonal system solver, which includes a significant amount of global computations. Performance results from testing two specific strategies will also be presented.

The tridiagonal solver is usually only one component in a realistic application code. For this study the targeted application code is a Navier-Stokes simulation code called CDNS (described below). Simulation codes of this type are very compatible with the data parallel programming model. They involve computations on several (5 to 20), large, multi-dimensional arrays which have similar shapes. Typically, each array stores the value of a physical variable for each node of a 3-dimensional computational grid defined on some control volume of interest. Many of the computations are local in nature as each array element is computed as a function of the value of other variables at the same control volume location (and thus the same computational grid-point and array location). However due to non-local physics or the need for a more accurate algorithm, these simulation codes contain some global computations.

2 Description of CDNS

CDNS (Compressible Direct Navier-Stokes Simulation code) is an explicit, finite-difference code developed at NASA Langley Research Center by the second author to study 3-D compressible turbulence [2]. This code solves the full Navier-Stokes equations using constant viscosity and Prandtl number. Spatial derivatives are calculated using a sixth-order, compact scheme which requires the solution of a tridiagonal matrix system of equations [7]. Time discretization is based on a low-storage, third-order, Runga-Kutta scheme. The code uses 14 three-dimensional arrays. For 432 grid points in each dimension, each array contains 81 megawords and the 14 arrays require 1130 megawords. This constitutes roughly 90% of the total memory requirement of the code. These 14 arrays require 2.6 megawords per node when distributed on 432 nodes. For comparison, a 128 grid-point problem would require 2.1 megawords per array and a total of 29 megawords.

CDNS computes the primitive variables—velocity vector, density and pressure—(stored in 5 of the 14 arrays) for a 3-D control volume. A computational grid with the same number of grid points in each dimension and equal spacing between grid points is used to identify the fluid volume. Periodic boundary conditions are assumed at the edges of the fluid volume. After initializing the primitive variable arrays with a zero-mean, steady-state but turbulent flow field, the code marches forward in time with small time steps to simulate the decay of the turbulent motion. Updating of the flow variables is accomplished by computing the residuals of the Navier-Stokes equations which are then used to estimate the time derivatives. The intermediate calculations are all local-type computations except for the computation of the spatial derivatives. This exception is important since the spatial derivative calculations depend on the solution of a tridiagonal system and account for 70% of the entire operation count.

While some of the derivative calculations are local in nature, most are of the partial global type. By partial it is meant that only one row or column of a 3-dimensional array is involved in a global calculation, not all the elements of the array. This row or column is the input and solution vector(r and s) for the tridiagonal system (Figure 1 and Table 1) and will be referred to as the dependent dimension. The other two dimensions of the 3-D arrays are merged to form a set of multiple solution vectors which can be solved simultaneously and will be referred to as the independent dimensions. This multiple set of problems is the source of the parallelism used to speed up the tridiagonal solver.

For CDNS an equal number of derivatives are needed in each direction. While the operation count is the same for computing the derivatives in each direction, the cost (in terms of time) can vary depending on the choice of data distribution and algorithm. Specifically, if all the elements of one dimension of a multi-dimensional array reside on the same node, the standard Gaussian elimination algorithm can be used in that direction and the resulting load-balanced, no communication code gives maximum parallel performance. However if that dimension is distributed, internode communications increase the cost for computing derivatives in that direction.

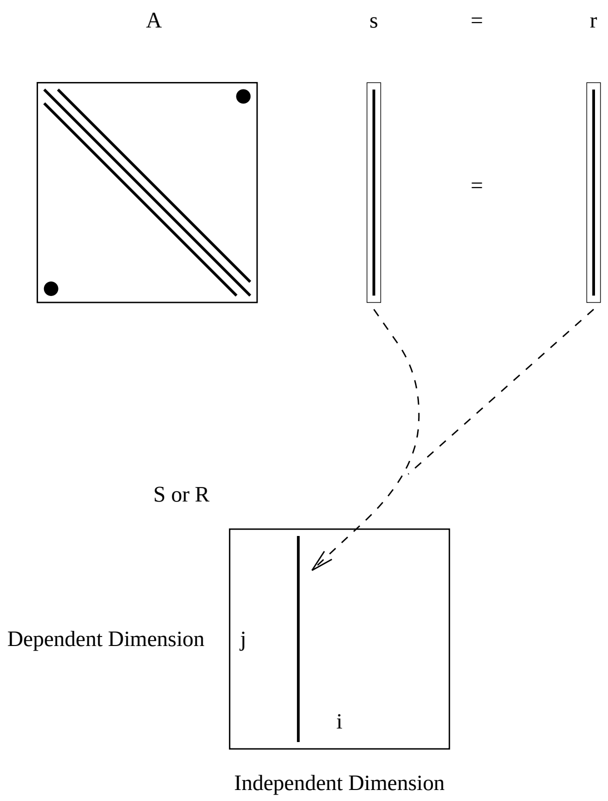

Figure 1: Tridiagonal system of equations

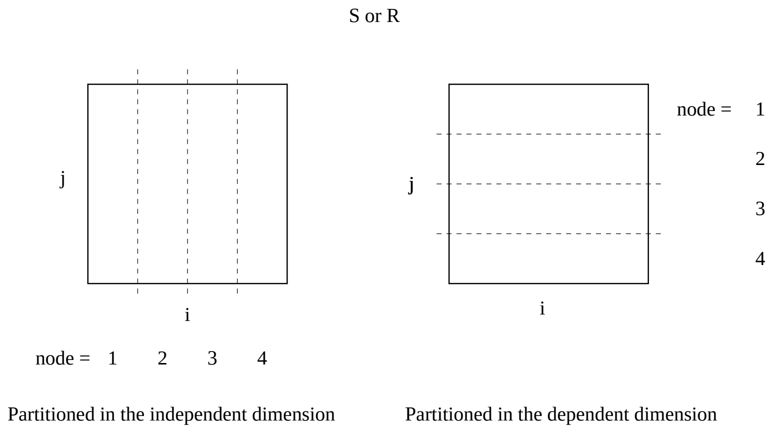

Figure 2: Tridiagonal system of equations partitioned in the independent and dependent dimension

```
Equation(s) to solve:

   A s = r (single system of equations)
   A S = R (multiple systems of equations)

Variable definitions:

   A - periodic tridiagonal matrix
   s - solution vector for single system
   S - set of solution vectors (each column holds one s-vector)
   r - input vector for single system
   R - set of input vectors (each column holds one r-vector)
```

Table 1: Definitions for tridiagonal system of equations

3 Tridiagonal Solver

3.1 Base problem

A periodic tridiagonal linear system of equations is pictured in Figure 1. Matrix A contains zeros in all elements except the 3 diagonal rows marked by solid lines. The uppermost, right-most element and the lowermost, left-most element are also nonzero. The most common algorithm to solve a linear system for one right-hand side or one r-vector, Gaussian elimination, contains little potential for parallelism and more sophisticated (and usually more expensive) algorithms are needed to solve a single right-hand side system in parallel. However, the need to solve multiple columns of an R-array for the same A-matrix results in a problem which is more amenable to an efficient parallel implementation. Recalling the discussion in the section describing CDNS, the "other two dimensions of the 3-D arrays" will map to the independent or i-dimension (Figure 1) to form a problem with an R-array as the right-hand side rather than just a single solution r-vector. The dependent dimension (the direction of the desired derivative) maps to the j-dimension of the R-array. Whether this mapping of the 3-D array to the R-array actually requires data motion depends on the actual computer architecture, compiler, and algorithm. To conserve memory the same storage area is used typically for both the solution (S) and the input (R) of the system.

If the i-dimension of S or R is distributed and the j-dimension is not (Figure 2), then the solution for the piece of S on each node is completely self-contained and a standard sequential algorithm can be used. If the j-dimension is distributed, then a tridiagonal algorithm is necessary which includes internode communication. While both situations generally occur in a code that solves a tridiagonal system in all three directions of a 3-D data set, the thrust of the algorithm development in this section is for the case where the dependent dimension is distributed.

The pseudo-code shown in Figure 3 will be used to demonstrate the concepts needed to implement a tridiagonal solver in parallel. In this figure the b-array corresponds to any one of the diagonals of matrix A. $S(i,j)$ is used to store both S and R as previously discussed. This code segment, which is typical of the loops in a tridiagonal solver on a vector based architecture, computes each step of a recursive problem (j do-loop) for all the multiple problems (i do-loop) before preceding to the next j-step. This recursion is what makes the computations global since the values computed in the j-th step depend on all the previous steps.

In the algorithms discussed in this section, it is assumed that only one dimension is distributed on NP nodes. The extension to the case of multiple distributed dimensions is straightforward since the additional distributed dimensions are always associated with the independent dimension and can be treated as multiple sets of problems of either type shown in Figure 2.

3.2 Simple algorithms

If the independent dimension is targeted for distribution, then the generation of parallel code is straightforward. The sample code in Figure 3 is transformed in Figure 4 . The i-loop is stripmined into the same number of strips as nodes (NP) creating an is-loop and an ip-loop. A dependency analysis shows that the is-loop can be moved outside the j-loop. The code for each pass of the is-loop operates on independent subsets of the $S(i,j)$. The work in each pass of the is-loop and the appropriate subset of the $S(i,j)$ can be distributed to the NP nodes and load-balanced parallelism with no communication results. Note that the b-array must be replicated on all nodes.

Targeting the dependent dimension for distribution is more complicated and is the focus of the remainder of this section. An analogous approach to the independent dimension strategy is shown in Figures 5, 6 and 7 . Here the j-loop is stripmined and the appropriate code and data distributed. Since data needed to compute on node n must first be computed on node $n-1$, only one node at a time can compute and no parallelism is achieved. In developing the parallel code, the $jp = 1$ pass must be peeled off and explicitly recoded to initialize $S(i,0) = S0(i)$ via a message from the preceding node. The message to receive the new data has to wait on the completion of the work on the preceding node before the data is sent. A straightforward insertion of message passing thus enforces the correct data dependency without any explicit synchronization. Clearly, it is naive to expect efficient parallelization of a dependent dimension to proceed so simply.

```
Definition of sizes:

      parameter (JD = "dependent dimension")
      parameter (ID = "independent dimension")
      parameter (NP = "number of processors")
c.............................................|
      dimension b(JD), S(ID,JD)

      do 20 j=2,JD
         do 10 i=1,ID
            S(i,j) = S(i,j) + b(j)*S(i,j-1)
   10    continue
   20 continue
c.............................................|
```

Figure 3: Base loop representing typical code in the tridiagonal solver

3.3 Chained algorithm

To develop a load-balanced code for a distributed j-dimension, one can try various stripmining strategies. Figure 8 gives an overall picture of a load-balanced strategy. The independent dimension is split into NP sets of problems and each set is started on a separate node. The algorithm is designed so that the solution of each problem set follows the other sets across all the nodes in a chained fashion. Thus each problem set has access to its data which is distributed across all the nodes, yet complete load-balancing is achieved. The generation of such an algorithm can be messy however. In the following development a series of transformation steps are outlined which convert a sequential algorithm into a balanced, parallel algorithm. Each step is very simple—most are common transformations used in compiler pre-processors. These transformations could be implemented in such a pre-processor, albeit under user control via directives, providing a reasonably straightforward way of generating a parallel code.

The transformation of the sequential algorithm proceeds as follows. The pseudocode in Figure 3 is transformed to demonstrate the concepts.

- First, both the i and the j index are stripmined (Figure 9). Stripmining in j is necessary if the algorithm is to be used for data which is distributed in j. The stripmining in i is necessary because this is the source of any potential load-balancing.
- Once stripmined, the next step in the algorithm development is to replace the

```
c... Sequential code transformation ...............|
c
c  ISN = number of processors
c  ISD = number of i-elements per processor
c
      parameter (ISN = NP; ISD = ID/NP)
      dimension b(JD), S(ID,JD)

      do 30 is=1,ISN
         do 20 j=2,JD
            do 10 ip=1,ISD
               i=ISD*(is-1) + ip
               S(i,j) = S(i,j) + b(j)*S(i,j-1)
   10       continue
   20    continue
   30 continue

c... Parallel code ................................|

      parameter (ISD = ID/NP)
      dimension b(JD), S(ISD,JD)

         do 20 j=2,JD
            do 10 ip=1,ISD
               S(ip,j) = S(ip,j) + b(j)*S(ip,j-1)
   10       continue
   20    continue

c..................................................|
```

Figure 4: Sequential code transformation and parallelization of code distributed in the independent dimension

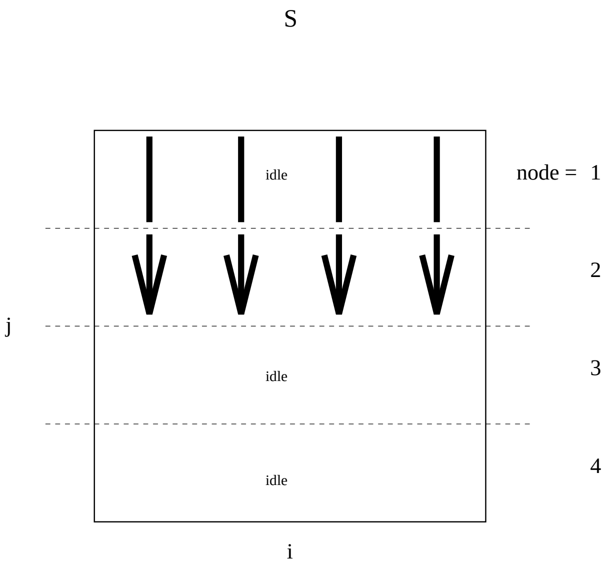

Figure 5: Sketch of naive algorithm

```
Stripmining:

 npi = number of elements per strip
  is = index of strip
  ip = index of element within a strip

   i = is * npi + ip

Base indices:

   i = data & algorithm index - "independent" direction
   j = data index             - "dependent" direction
   k = algorithm index        - "dependent" direction

Strip indices:

  is = data & algorithm strip index - "independent" direction
  js = data strip index             - "dependent" direction
  ks = algorithm strip index        - "dependent" direction
```

Table 2: Definition of strip indices

```
c... Sequential code transformation ................|
     c
     c  JSN = number of processors
     c  JSD = number of j-elements per processor
     c
      parameter (JSN = NP; JSD = JD/NP)
      dimension b(JD), S(ID,JD)

      js=1
      do 2 jp=2,JSD
         j=JSD*(js-1) + jp
         do 1 i=1,ID
            S(i,j) = S(i,j) + b(j)*S(i,j-1)
    1    continue
    2 continue

      do 30 js=2,JSN
         do 20 jp=1,JSD
            j=JSD*(js-1) + jp
            do 10 i=1,ID
               S(i,j) = S(i,j) + b(j)*S(i,j-1)
   10       continue
   20    continue
   30 continue
c...................................................|
```

Figure 6: Sequential code transformation of code distributed in the dependent dimension—naive algorithm

```
c... Parallel code ................................|
c
c  jn = node identifier
c
      parameter (JSD = JD/NP)
      dimension b(JSD), S(ID,JSD), S0(ID)

      js = jn
      if (js .ne. 1) then
         jp=1
         get [ from:  S(i,JSD) on processor n-1
                      after loop 20
                 to:  S0(i) on jn (this processor) ]
         do 1 i=1,ID
            S(i,jp) = S(i,jp) + b(jp)*S0(i)
    1    continue
      end if

      do 20 jp=2,JSD
         do 10 i=1,ID
            S(i,jp) =S(i,jp) + b(jp)*S(i,jp-1)
   10    continue
   20 continue
c..................................................|
```

Figure 7: Parallelization of code distributed in the dependent dimension—naive algorithm

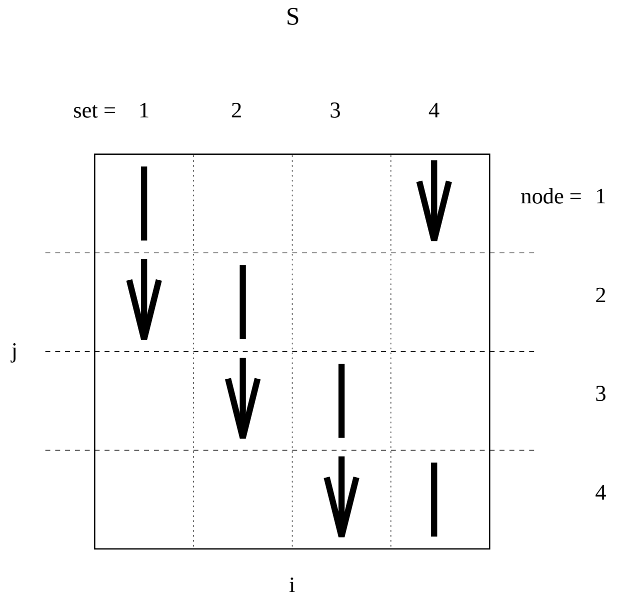

Figure 8: Sketch of chained algorithm

js-index with 2 indices (Figure 10 and Table 2). The original js-index serves 2 functions:

1. indexing the progress of the algorithm (new index is ks)
2. indexing the array to select the correct data at each step (new index is js).

The inner part of the stripmined j-index, the jp-index, is not required to be split. Parallelization is achieved by rearranging the strips (indexed by js) rather than the elements of a strip (index by jp). Initially, this is only a change in name of the indices, so the statement, $js = ks$, must be added to the code.

- For each set of columns with the same is-index (which defines an independent subproblem), the algorithm does not have to start on the node $js = 1$. The top group of rows (first js-strip) of the system could be moved to the bottom as shown in Figure 11 . After re-ordering the columns, the resulting system of equations has the same periodic, tridiagonal form as the original system. However, the first row of the S or R-array for this modified system becomes is a member of the $js = 2$ strip. The $is = 1$ subset of problems can be solved with the original system and the $is = 2$ subset solved with this modified system. The modified A matrix may result in a different factorization, but the modified system is still solving the original problem and will give the same answers within numerical accuracy limits. Other modified systems can be defined for the other is-strips. The important point is that the data in the S or R-array does not have to be moved to solve one of the modified problems. The solution of a modified problem begins at an interior value of the js-index which is the desired goal.

The above algorithm changes along with several other system re-orderings can be achieved by replacing the $js = ks$ statement in Figure 10 with a more general

```
c... Sequential code transformation ...............|

c    stripmine code

      parameter (ISN = NP; ISD = ID/NP)
      parameter (JSN = NP; JSD = JD/NP)
      dimension b(JD), S(ID,JD)

      js=1
      do 2 jp=2,JSD
         j=JSD*(js-1) + jp
         do 1 is=1,ISN
         do 1 ip=1,ISD
            i=ISD*(is-1) + ip
            S(i,j) = S(i,j) + b(j)*S(i,j-1)
    1    continue
    2 continue

      do 20 js=2,JSN
      do 20 jp=1,JSD
         j=JSD*(js-1) + jp
         do 10 is=1,ISN
         do 10 ip=1,ISD
            i=ISD*(is-1) + ip
            S(i,j) = S(i,j) + b(j)*S(i,j-1)
   10    continue
   20 continue
c..................................................|
```

Figure 9: Chained algorithm code development - sequential transformation step 1

```
c... Sequential code transformation ................|

c    separate data(js) & algorithm(ks) index:

      parameter (ISN = NP; ISD = ID/NP)
      parameter (JSN = NP; JSD = JD/NP)
      dimension b(JD), S(ID,JD)

      ks=1
      do 2 jp=2,JSD
         js = ks
         j=JSD*(js-1) + jp
         do 1 is=1,ISN
         do 1 ip=1,ISD
            i=ISD*(is-1) + ip
            S(i,j) = S(i,j) + b(j)*S(i,j-1)
    1    continue
    2 continue

      do 20 ks=2,JSN
      do 20 jp=1,JSD
         js = ks
         j=JSD*(js-1) + jp
         do 10 is=1,ISN
         do 10 ip=1,ISD
            i=ISD*(is-1) + ip
            S(i,j) = S(i,j) + b(j)*S(i,j-1)
   10    continue
   20 continue
c...................................................|
```

Figure 10: Chained algorithm code development - sequential transformation step 2

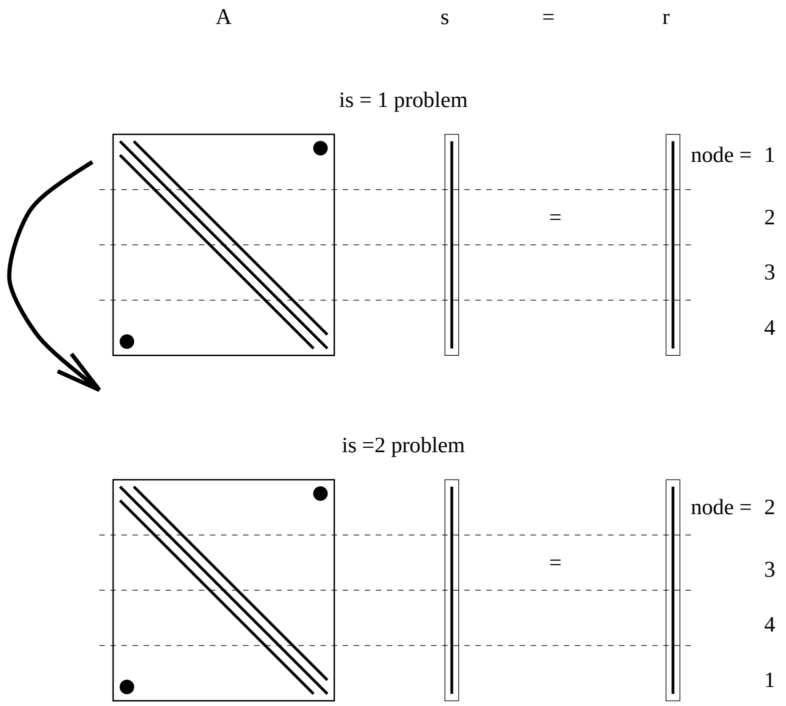

Figure 11: Chained algorithm code development - reorder rows of A for is=2 strip

function, $js = P(ks, is)$, as shown in Figure 12. $P(ks, is)$ specifies the various permutations of the js-index with respect to the ks-index for each is-strip. To recover the original code, Equation 1 is used to define $P(ks, is)$. One particular permutation which load-balances the algorithm using the approach outlined in the preceding paragraph is given by Equation 2.

$$P(ks, is) = ks \tag{1}$$

$$P(ks, is) = mod(ks + is - 2, NP) + 1 \tag{2}$$

Replacing Equation 1 with Equation 2 can be viewed as a code transformation that rotates the algorithm index with respect to the data index (Figure 13). After such a transformation the index relationships between

- is, the index defining the subsets or strips of independent problems,
- js, the index defining which strip of $S(i, j)$ is being used, and
- ks, the index which marches the algorithm in the dependent dimension

would be as shown in Figure 14.

- The problem with the code in Figure 13 is that there is no easy way to distribute the code and obtain parallel execution. If the code in Figure 13 is replicated on each node, the computation of $S(i, j)$ inside the js-loop should only be done when $js = jn$, where jn is a unique identifier assigned to each node and has the same range as js. In other words, the code executing on each node can only use data that resides on that node (except for starting values for the recursion). This strategy would not be an efficient solution, since extra loop passes would be executed with only one of these passes doing any "real" computations.

Alternatively the is-index in the do-loop could be replaced by the js-index and $js = P(is, ks)$ replaced with $is = P_i(ks, js)$ where P_i is defined by Equation 3.

$$P_i(ks, js) = mod(js - ks, NP) + 1 \tag{3}$$

This function maintains the same relationship between is, js and ks as Equation 2. This index swapping transformation is shown in Figure 15. From Figure 14, it can be seen that for a fixed value of ks, js is just a permutation of is, therefore index swapping maintains the load-balancing potential of the algorithm.

- The js-loop can now be moved to the outside of each loop nest based on a dependency analysis. (Figure 16)
- The actual parallelization of the code now entails replacing the js-loop with $js = jn$ and assigning the code inside the js-loop for execution on each node. (Figure 17)

The resulting parallel algorithm is shown in Figure 17. Note that it is completely load balanced, contains the same total operation count as the original algorithm and can easily be mapped to a node network layout to obtain nearest neighbor communication. The communication costs are relatively small, since only two groups of data are actually moved every time the algorithm crosses to the next node for each is-strip. Each group is roughly the size of one "surface" of the 3-D array section located on each node. The communication cost will be discussed in more detail later in the paper. For the periodic system discussed here, the data is mapped to the nodes for the rotated algorithm in a straightforward, data-parallel manner; i.e., no complex data mapping is needed. Therefore, the derivative computations in the non-distributed dimensions are not affected. Also, the sample code used to demonstrate the transformations makes only one pass through the data. The extensions needed for the backward pass in an actual solver are straightforward.

3.4 Transpose algorithm

The algorithm discussed in the previous section, while having many positive characteristics, still has the drawback of complexity. A simple strategy is:

- to rearrange the data globally using internode communications so that the data for each global calculation is all on one node,

```
c... Sequential code transformation ...............|

c    make data index (js) an explicit mapping function

      parameter (ISN = NP; ISD = ID/NP)
      parameter (JSN = NP; JSD = JD/NP)
      dimension b(JD), S(ID,JD), P(ISN,JSN)

      ks=1
      do 2 jp=2,JSD
         do 1 is=1,ISN
         js=P(is,ks)
         j=JSD*(js-1) + jp
         do 1 ip=1,ISD
            i=ISD*(is-1) + ip
            S(i,j) = S(i,j) + b(j)*S(i,j-1)
    1    continue
    2 continue

      do 20 ks=2,JSN
      do 20 jp=1,JSD
         do 10 is=1,ISN
         js=P(is,ks)
         j=JSD*(js-1) + jp
         do 10 ip=1,ISD
            i=ISD*(is-1) + ip
            S(i,j) = S(i,j) + b(j)*S(i,j-1)
   10    continue
   20 continue
c..............................................|
```

Figure 12: Chained algorithm code development - sequential transformation step 3

```
c... Sequential code transformation ...............|

c    rotate map function in recursive dimension

      parameter (ISN = NP; ISD = ID/NP)
      parameter (JSN = NP; JSD = JD/NP)
      dimension b(JD), S(ID,JD), P(ISN,JSN)

      do 100 ks=1,JSN
      do 100 is=1,ISN
         js = ks + (is-1)
         if (js .gt. NP) js = js - NP
         P(is,ks) = js
  100 continue

      ks=1
      do 2 jp=2,JSD
         do 1 is=1,ISN
         js=P(is,ks)
         j=JSD*(js-1) + jp
         do 1 ip=1,ISD
            i=ISD*(is-1) + ip
            S(i,j) = S(i,j) + b(j)*S(i,j-1)
    1    continue
    2 continue

      do 20 ks=2,JSN
      do 20 jp=1,JSD
         do 10 is=1,ISN
         js=P(is,ks)
         j=JSD*(js-1) + jp
         do 10 ip=1,ISD
            i=ISD*(is-1) + ip
            S(i,j) = S(i,j) + b(j)*S(i,j-1)
   10    continue
   20 continue
c..................................................|
```

Figure 13: Chained algorithm code development - sequential transformation step 4

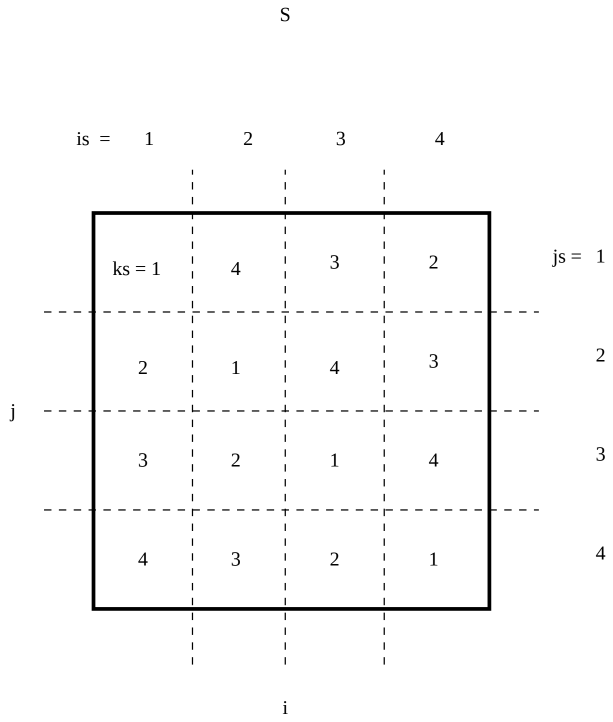

Figure 14: Chained algorithm code development - index relationships after rotation transformation

- to use a sequential algorithm to make those calculations, and
- to return the data to its original distribution.

A global transpose can be used to accomplish the first and last items to implement such a strategy for a tridiagonal solver.

The implementation proceeds as follows. Each problem (one column of an R-array) has data distributed on each node in (js, is)-blocks with the same is-index. All the blocks with the same is-index can be moved to the same node via a global transpose as pictured in Figure 18. With all the data for each problem now on the same node, a sequential algorithm can be used. If a complete transpose, such as implied by Figure 19, is desired, an on-node or local transpose of the data within each (js, is)-block is also needed. This on-node data motion sometimes can be hidden behind the global data motion. The complete transpose frequently will give better on-node performance than just a global transpose.

Bohkari[1] has discussed several algorithms to implement a global transpose. On a hypercube node architecture, the choice of algorithms is very important since algorithms that avoid link contention can be written. On a mesh architecture, link contention problems make the determination of an improved algorithm tedious; moreover, any performance improvements are usually small. To date the XOR algorithm, which is generally best for a hypercube architecture, is also one of the better choices

```
c... Sequential code transformation ...............|

c    invert map function

      parameter (ISN = NP; ISD = ID/NP)
      parameter (JSN = NP; JSD = JD/NP)
      dimension b(JD), S(ID,JD), PI(JSN,JSN)

      do 100 js=1,JSN
      do 100 ks=1,JSN
         is = js - ks +1
         if (is .le. 0) is = is + NP
         PI(ks,js) = is
  100 continue

      ks=1
      do 2 jp=2,JSD
         do 1 js=1,JSN
         is=PI(ks,js)
         j=JSD*(js-1) + jp
         do 1 ip=1,ISD
            i=ISD*(is-1) + ip
            S(i,j) = S(i,j) + b(j)*S(i,j-1)
    1    continue
    2 continue

      do 20 ks=2,JSN
      do 20 jp=1,JSD
         do 10 js=1,JSN
         is=PI(ks,js)
         j=JSD*(js-1) + jp
         do 10 ip=1,ISD
            i=ISD*(is-1) + ip
            S(i,j) = S(i,j) + b(j)*S(i,j-1)
   10    continue
   20 continue
c.................................................|
```

Figure 15: Chained algorithm code development - sequential transformation step 5

```
c... Sequential code transformation ...............|

c    make js-loop the outer loop

      parameter (ISN = NP; ISD = ID/NP)
      parameter (JSN = NP; JSD = JD/NP)
      dimension b(JD), S(ID,JD), PI(JSN,JSN)

      do 100 js=1,JSN
      do 100 ks=1,JSN
         is = js - ks +1
         if (is .lt. 0) is = is + NP
         PI(ks,js) = is
  100 continue

      ks=1
      do 1 js=1,JSN
         do 2 jp=2,JSD
            is=PI(ks,js)
            j=JSD*(js-1) + jp
            do 1 ip=1,ISD
               i=ISD*(is-1) + ip
               S(i,j) = S(i,j) + b(j)*S(i,j-1)
    1       continue
    2    continue
    3 continue

      do 30 js=1,JSN
         do 20 ks=2,JSN
         do 20 jp=1,JSD
            is=PI(ks,js)
            j=JSD*(js-1) + jp
            do 10 ip=1,ISD
               i=ISD*(is-1) + ip
               S(i,j) = S(i,j) + b(j)*S(i,j-1)
   10       continue
   20    continue
   30 continue
c..................................................|
```

Figure 16: Chained algorithm code development - sequential transformation step 6

```
c... Parallel code (processors 1 to NP) ...........|

      parameter (ISN = NP; ISD = ID/NP; JSN = NP; JSD = JD/NP)
      dimension b(JSD), S(ID,JSD), S0(ID), PI(JSN)

      js = jn ( = node identifier )
      do 100 ks=1,JSN
         is = js - ks +1
         if (is .lt. 0) is = is + NP
         PI(ks) = is
  100 continue

      ks=1
      do 1 js=1,JSN
         do 1 jp=2,JSD
            is=PI(ks)
            j=JSD*(js-1) + jp
            do 1 ip=1,ISD
               i=ISD*(is-1) + ip
               S(i,j) = S(i,j) + b(j)*S(i,j-1)
    1 continue

         do 30 ks=2,JSN
            jp=1
            send [ from:  S(i,JSD) on processor n-1 after loop 20
                     to:  S0(i) on processor n (local)]
            do 21 ip=1,ISD
               is=PI(ks)
               i=ISD*(is-1) + ip
               S(i,jp) = S(i,jp) + b(jp)*S0(i)
   21       continue

            do 20 jp=2,JSD
               is=PI(ks)
               do 20 ip=1,ISD
                  i=ISD*(is-1) + ip
                  S(i,jp) = S(i,jp) + b(jp)*S(i,jp-1)
   20       continue
   30    continue
c...........................................|
```

Figure 17: Chained algorithm code development - parallel transformation step 7

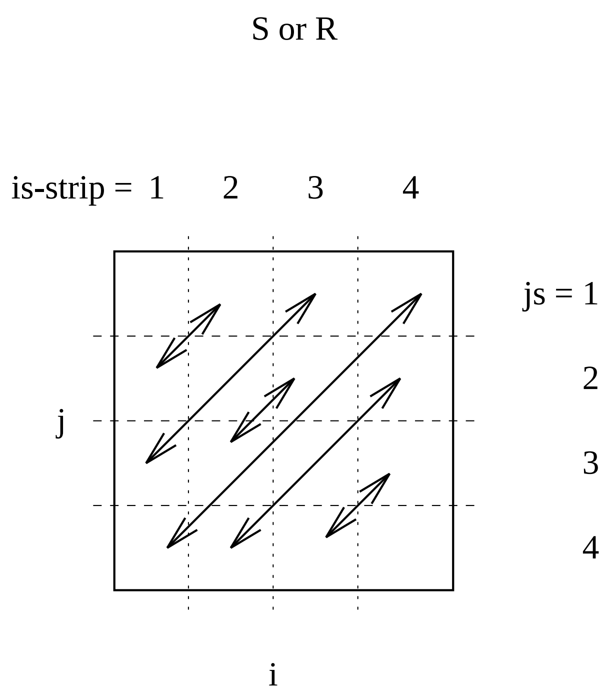

Figure 18: Transpose algorithm code development - global transpose sketch

for a mesh. This algorithm was used for the timing test reported herein. The communication costs are further discussed in the next section.

4 Target data distributions

In developing a strategy to adapt a code like CDNS to a distributed memory computer a major decision is how to subdivide the large arrays. It is natural to equally partition either one, two or three dimensions of each array and then distribute each array identically. The choice of how many dimensions to partition is then the main issue. The discussion in this paper assumes that the number of array elements in each dimension is evenly divisible by the number of nodes allocated to that dimension. The performance results reported herein are all for this evenly divisible case. The conclusions of this study regarding algorithm strategy should also apply to the more general case.

Simple models to estimate the computation and data communication times are more difficult on parallel/distributed memory computers than on other typically used architectures. Link contention can be very difficult to predict even when a code does not share the node network with other codes. Overlapping communications with on-node computations further complicates the picture. Nevertheless, some insight can be gained from such models.

The model estimates herein include the communication cost for the formation

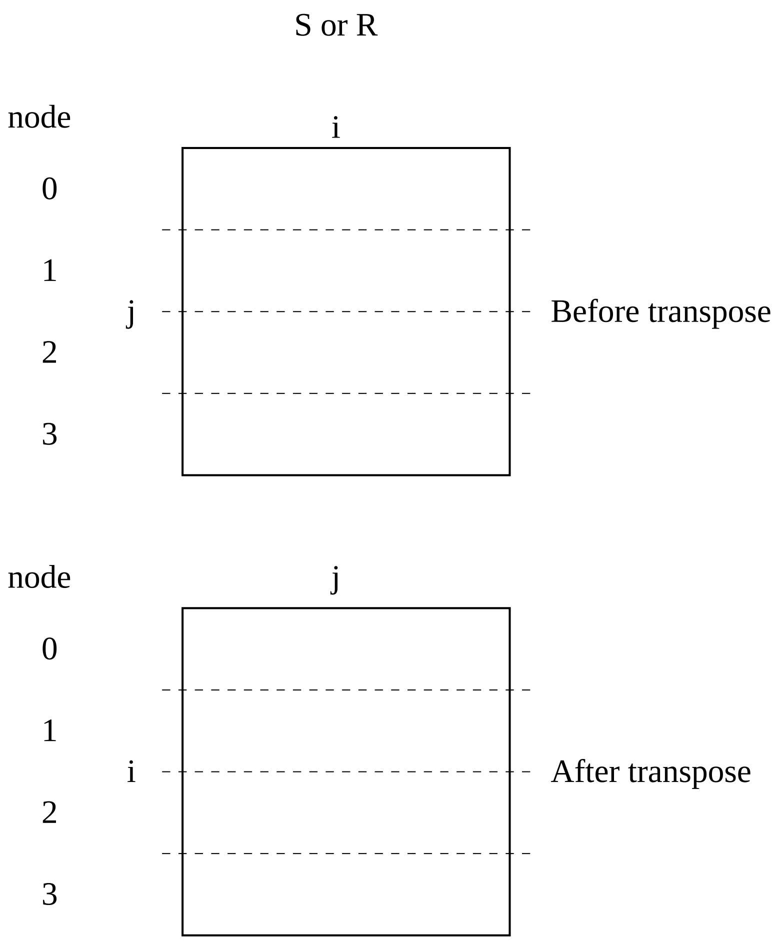

Figure 19: Transpose algorithm code development - complete transpose sketch

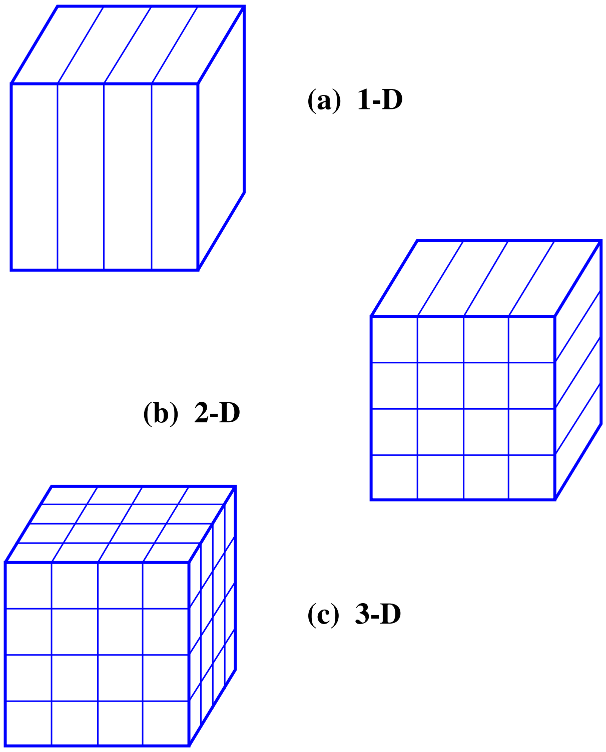

Figure 20: Data distributions

of the input array and for the solution of the tridiagonal system. Figure 20 shows the three basic choices for distributing the 3-dimensional arrays. For any of the distribution methods and for any of the three directions of the derivatives, the overall problem can be formulated into a set of problems of either of the two types shown in Figure 2.

Two approaches based on the above tridiagonal algorithms were considered to solve the problem of computing a derivative in each of the three directions with good overall efficiency. One approach is based on the chained algorithm which is used only in the directions where the data is distributed. For the directions that are not distributed the sequential algorithm is used and the communication costs are zero. Communication costs for the chained algorithm are estimated in Appendix A. The estimates for each of the three possible data distributions are given in Table 3. The estimates are for the sum of the costs of computing a derivative in each of the three directions.

variable	1-D	2-D	3-D
n	N	$N^{1/2}$	$N^{1/3}$
m	$4N^2$	$8N^{3/2}$	$12N^{4/3}$
s	L^2/N	L^2/N	L^2/N
t	$4NL^2$	$8N^{1/2}L^2$	$12N^{1/3}L^2$
f	$4N/L$	$8N^{1/2}/L$	$12N^{1/3}/L$

Table 3: Communication estimates for the chained algorithm

L = number of grid points in each direction
A = size of one 3-D array = L^3
N = total number of nodes
n = number of nodes in each dimension
m = total number of messages
s = size of one message
t = total size of data moved
f = fraction of one array moved = t/A

For the chained algorithm both the number of messages and the amount of data moved decrease as the number of partitioned dimensions increases. Thus, higher dimension partitioning is preferable. However, since the target architecture, the Touchstone Delta [5], for this current work is a 2-D node mesh, the 2-D partitioning should be better because the communication will then be predominantly nearest-neighbor between the physical nodes. For architectures where 3-D partitioning maps without link contention penalties, then the higher dimension partitioning should give better performance.

The second approach is to globally transpose the entire S or R array for those directions were the dependent dimension is distributed. The communication costs for this case are given in Table 4. The development of these estimates is described in Appendix A.

While the number of messages is smaller for the 3-D partitioning, the total amount of data moved is approximately 3 times that for the 1-D partitioning. Since the amount of data is large (note that message sizes are proportional to the array "volume") the total amount of data moved should be more significant than the overhead costs (represented by the number of messages). Therefore, 1-D partitioning appears to be the better choice for an algorithm involving transposes.

This simple analysis does not determine whether the transpose strategy or the distributed tridiagonal solver strategy is better. Thus the 1-D case for the transpose

variable	1-D	2-D	3-D
n	N	$N^{1/2}$	$N^{1/3}$
m	$2N^2$	$4N^{3/2}$	$6N^{4/3}$
s	L^3/N^2	$L^3/(N^{3/2})$	$L^3/(N^{4/3})$
t	$2L^3$	$4L^3$	$6L^3$
f	2	4	6

Table 4: Communication estimates for the transpose algorithm

strategy and the 2-D case for the distributed solver strategy were both implemented and tested. The performance behavior of the two algorithms was determined by experiment and is reported later in this paper.

While the above total communication statistics give some estimate of the communication costs, the effective communication statistics defined by Equations 4 and 5 would give a better measure. The factor e in these equations is a parallel efficiency which is a function of the amount of parallelism available, load-balancing and link contention.

$$m_e = m/e \tag{4}$$

$$t_e = t/e \tag{5}$$

$e = N$ implies that the maximum available parallel communication is achieved, while $e = 1$ implies that the nodes send a set of messages sequentially. Link contention and load-balancing can be difficult to quantify so their effects must usually be discussed qualitatively. Neither the chained nor the transpose algorithm should loose efficiency due to load-balancing problems. While the transpose algorithm has no link-contention problems on a hypercube architecture, it can have significant problems on a mesh architecture. Since the chained algorithm uses nearest neighbor communications, link contention is not a problem. Therefore the communication costs for the chained algorithm should be lower than for the transpose approach.

5 Selection of specific algorithm for testing

From the preceding discussion, it is clear that several tradeoffs exist between the different algorithms. The transpose approach is the most straightforward, since the sequential algorithm can be used with only a few modifications. The difficulties of

variable	1-D	2-D	3-D	2D/1D
m	2	2	2	$4/(N^1/2)$
s	n/L	n/L	n/L	N/L
t	$2n/L$	$2n/L$	$2n/L$	$4(N^1/2)/L$

Table 5: Ratio of communication statistics: balanced algorithm / transpose algorithm

the distributed memory architecture are localized to a global data transpose module which can be made available as a routine or template in a library. This approach can be used for a very wide range of applications. However, the large data communication costs of a global transpose leaves much room for improvement. The ratio of message sizes of the 2 algorithms in Table 5 shows that the chained algorithm requires less total data transferred, although the number of messages is only slightly higher. The parallel efficiency effects also favor the chained algorithm. However, the coding of these parallel algorithms is more difficult. Quantitative performance comparisons of these two approaches as well as code development experience are needed to better evaluate the trade-off between programming time and execution time.

The transpose algorithm was implemented for the case where one dimension of the 3-dimensional arrays is distributed. This is referred to as the SLAB version. The three array dimensions (indexed by i, j, k) correspond to the three physical directions (x, y, z). The chosen dimension for distributing was the second (j or y). For on-node performance reasons (mainly better vectorization) the tridiagonal solver algorithm is written so that the third dimension is the solution vector dimension. By letting the z-direction reside on-node, no data motion is needed for solutions in the z-direction. For solutions in the y-direction, the local data motion to make y the last dimension can be overlapped with some of the global data motion needed to get all the distributed y-strips on the same node. Choosing either x or y (but not z) for the distributed dimensions allows for this overlap. In this case the choice of distributing y made the code development slightly easier due to the style of the original code and to the specifics of some analysis routines which were part of the code but are not discussed herein. This overlap potential cannot be fully realized on the Touchstone Delta, but this capability is part of the design for some newer architectures.

A 2-dimensional distribution strategy was chosen for the chained algorithm since the target computer, the Touchstone Delta, has a 2-dimensional node mesh. x and y were chosen as the distributed dimensions for the reasons mentioned in the discussion of the transpose algorithm. This version is referred to as the TUBE version.

6 Timing results on the Touchstone Delta

6.1 Test description

A series of timing tests were made on the derivative kernel for the CDNS code for both the TUBE and SLAB versions. The derivative kernel computes each of the three spatial derivatives of a scalar function defined on a 3-dimensional computational grid. The derivative in each of the 3 directions was timed. Any necessary data movement to change the data from the base memory layout to a layout specific to a particular derivative is included in the timing of that derivative. The timings only include the forward and backward substitution phase of the tridiagonal solver. Since the tridiagonal matrix does not vary during a run, the factorization is executed once. Both single direction timings and the total time to compute the derivative in all 3 directions are presented. These 3-direction timings give a good measure of how the kernel performs in the CDNS code since the code requires the same number of derivative calculations in each direction. In general, CDNS runs about 5% faster than the kernel. This is because the other operations in the simulation code are mostly of the local type. CDNS also includes some input/output and analysis code which executes slower but less frequently.

Most of the timing results will be presented in Mflops. Assuming the same number of grid points in each direction, operation counts for the derivative kernel are:

$$\begin{aligned} O &= 14L^3 - 12L^2 \\ P &= O/(T10^6) \end{aligned} \tag{6}$$

where,

L = number of grid points in each dimension
O = number of floating point operations using the sequential, periodic Gaussian elimination algorithm
T = measured execution time in seconds
P = performance measure in units of Mflops.

It should also be noted that the entire code is written in highly optimized Fortran code. The Mflop rates for derivatives in the z-direction, which contain no internode communications, are in the 15-20 range which is high for Fortran code which does not take particular advantage of the cache. As an aside, it was found to be difficult to write assembler code that significantly exceeds the Fortran compiler performance for the algorithms under investigation.

6.2 Performance measures

In evaluating an algorithm, one is interested in both its absolute performance as well as its scalability[8, 9]. An algorithm which has good or best performance over a large range of nodes is obviously desirable. Frequently, a performance measure such as Mflops is plotted versus number of nodes for a fixed problem size (a fixed grid size for CDNS and similar codes). The closeness of this curve to the linear extensions of the single node performance, Equation 7, is the basis of many conclusions about scalability.

$$P_g = P_1 N \tag{7}$$

where,

$$\begin{aligned} N &= \text{number of nodes} \\ P_1 &= \text{single node performance measure} \\ P_g &= \text{multi-node performance goal.} \end{aligned}$$

The first problem with this is that one is not usually interested in running a small problem on a large number nodes. One is typically interested in running a larger problem as the number of nodes increases, although there are exceptions. Most codes will have better performance per node when they are run on the minimum number of nodes possible. Maximizing an individual code's performance translates into improved overall system throughput. Therefore, a more useful measure is the performance versus nodes at a fixed memory utilization. In this study, for each set of tests which were run on the same number of nodes, linear interpolation was used to estimate the performance when the 14 major arrays used in CDNS filled 90% of each node's memory. The curves marked "90% mem" are quadratic least squares fits through these 90% estimate points. Since the test kernel for the derivative uses fewer than the 14 arrays needed by CDNS, some of the test data is for CDNS equivalent problem sizes greater than 100%.

In addition, the 90% measure eliminates some on-node effects that may not be desired in a scalability measure. The Touchstone Delta uses an Intel i860 processor for which the Fortran compiler generates a form of vector code [6]. The performance of the resulting vector code is dependent on vector length for vectors up to several hundred in size. As the number of nodes increases for a fixed problem size, the vector lengths can become shorter. Including this effect in the scalability measure biases the result. The 90% measure does not guarantee the elimination of vector performance variations or other on-node effects, but it should reduce these effects.

Another problem relates to the direct comparison of data to Equation 7. The comparison of multi-node timings to single node timings is not particularly relevant. Many cases which are currently run on a large number of nodes cannot be run on a

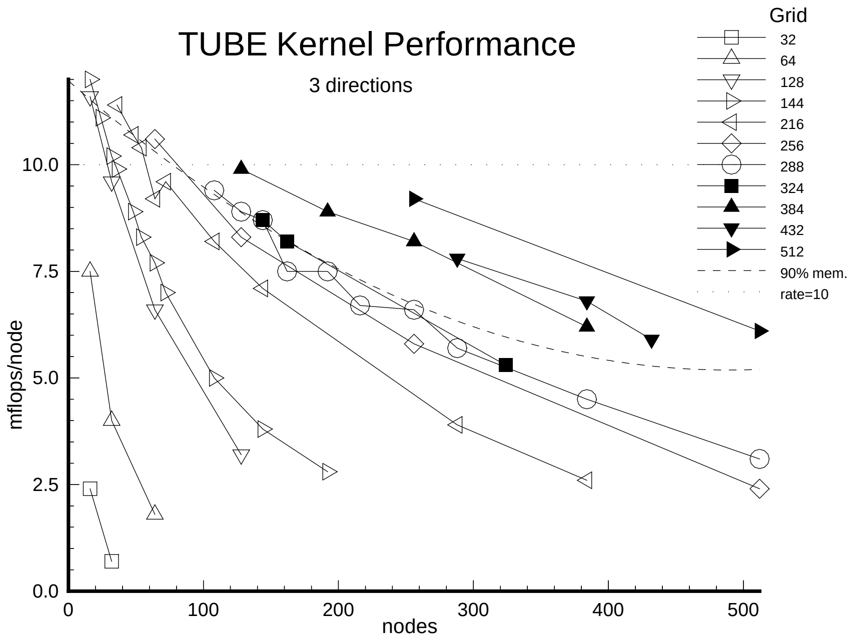

Figure 21: TUBE kernel performance - rate per node for all directions

single node. The choice of what to use for the single node performance is not clear—a single computation for a smaller problem, the extrapolation of the very smallest number of nodes possible to a single node, or the single node performance on another machine with larger memory. The comparison of multi-node timings to any of the above choices can be justified to gain insight into some performance issues, but they do not seem to give the best overall performance picture. A better measure is the performance per node versus number of nodes. A constant value of the performance per node over a range of nodes implies scalability for that range. An algorithm may have several scalability plateaus or constant ranges. In a plot of performance versus nodes, one generally tries to discern a constant slope region in a set of data and compare this slope to a single measure of goodness, i.e. Equation 7. If multiple constant slope regions are observed, a linear curve through that region (Equation 7 with a secondary value of P_1) must extend through zero to imply scalability. All this information is much easier to ascertain in the performance per node format.

6.3 Results of TUBE algorithm

Figure 21 is a plot of the 3-direction TUBE algorithm performance results. For a fixed grid size, the performance drop with increasing number of nodes is large. However, above 250 nodes the scalability suggested by the 90% measure is very good. Clearly, the algorithm and architecture are most compatible for the larger problem/node sizes. The same data is replotted in Figure 22 as Mflops versus nodes. Equation 7 is included on the figure with a single node performance of 10 Mflops assumed. It is nearly

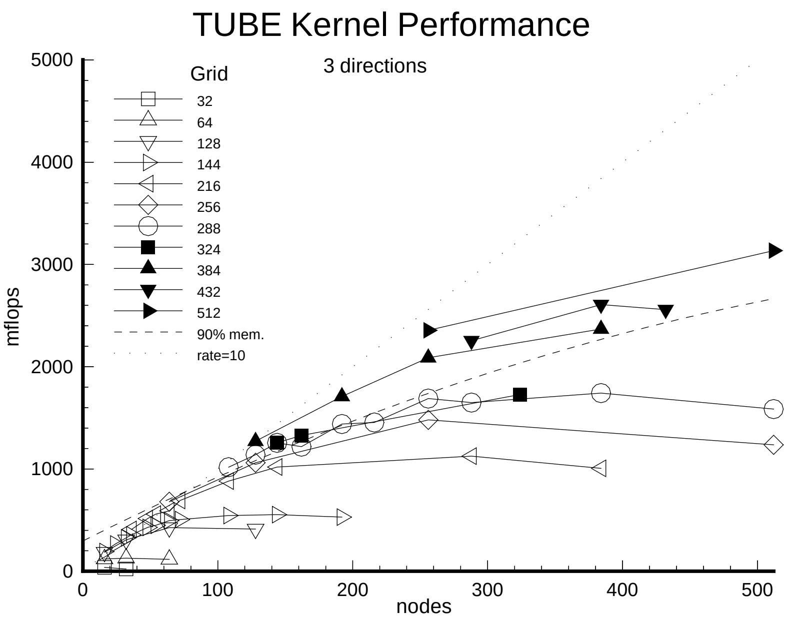

Figure 22: TUBE kernel performance - rate for all directions

impossible to discern in this plot format the scalability plateau between 250 and 500 nodes that was observed in Figure 21.

The data for a derivative in one direction only is also informative. The z-direction data (Figure 23) is almost constant which is expected since there is no internode communication in this part of the algorithm. For a fixed problem size, the performance drops off as the number of nodes increases. This is an example of the on-node vector length effect cited above. The x- and y-direction data (Figures 24 and 25) look similar to the 3-direction data. This is as expected since the slower performance sections of an algorithm will dominate its overall performance character. While the node grid was selected as close to square as possible, the number of nodes in the x-direction was sometimes larger than that in the y-direction. That is why the y-direction performance tends to be slightly greater. Otherwise the derivatives in these two directions are essentially the same—both using the chained parallel algorithm.

6.4 Results of SLAB algorithm

The 3-direction data for the SLAB algorithm is shown in Figure 26. The SLAB algorithm was run on only a few node/grid combinations because of restrictions due to the particular transpose algorithm chosen. Since the performance of this algorithm was generally lower than the TUBE version, a more flexible transpose algorithm was not implemented. With the limited data one is not able to determine how well this algorithm scales for large problems. Since the cost of communication grows like the

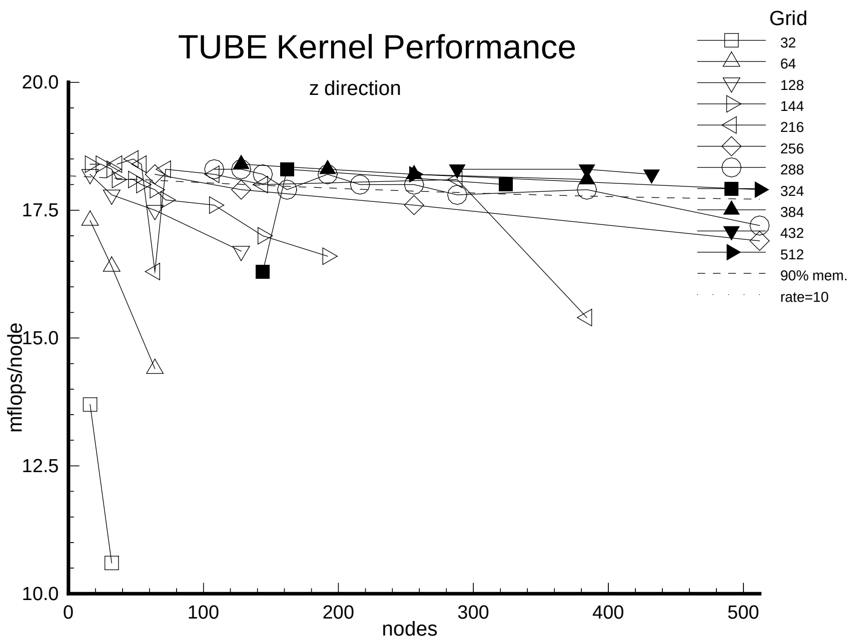

Figure 23: TUBE kernel performance - rate per node for z directions

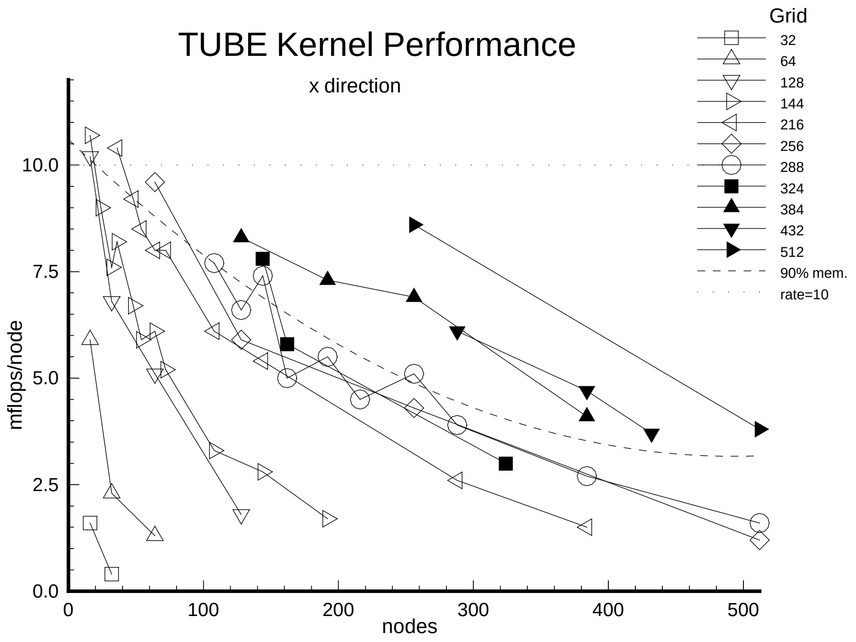

Figure 24: TUBE kernel performance - rate per node for x directions

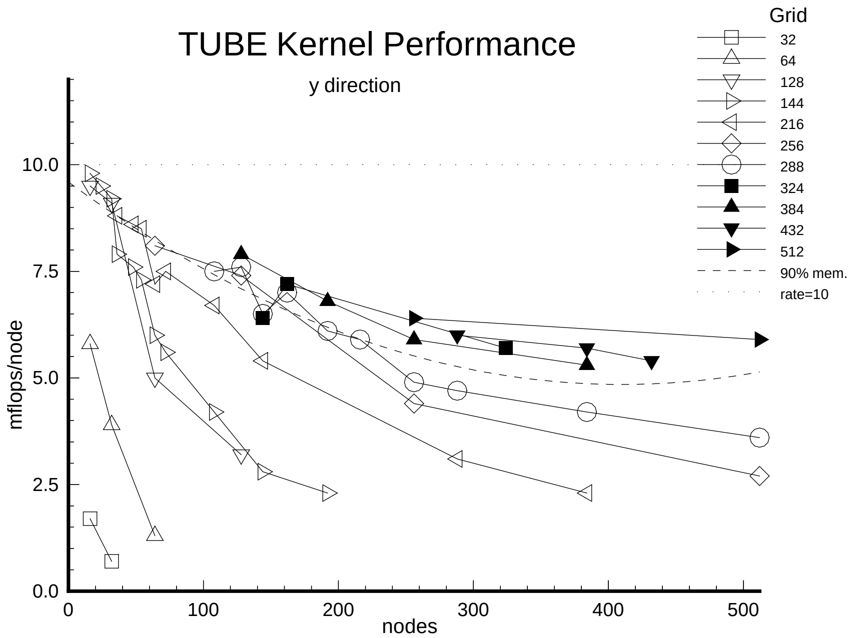

Figure 25: TUBE kernel performance - rate per node for y directions

"volume " of the arrays as the problem size increases, the scaling should be worst than that for the TUBE algorithm. For the smallest grids, the SLAB algorithm actually results in better performance than the TUBE algorithm. This SLAB algorithm also had better performance on a hypercube network as compared to a mesh network. This is because a transpose can be coded more efficiently on a hypercube where fewer link contentions generally result.

7 Conclusion

Parallel, distributed memory computing, as compared to sequential and vector computing, generates a larger variety of algorithms that need to be considered. The different choices also result in a wider range of performance. Understanding implementation details is also more important, since different implementations of the same basic algorithm can have even larger differences in performance. It is especially important when presenting algorithm comparisons to specify sufficient detail so that any performance comparisons can be fully appreciated and duplicated.

The trade-offs between simple, easy-to-implement algorithm strategies (such as the transpose algorithm) and more complicated strategies (the chained algorithm) for this current study seem to favor the chained algorithm. Although the improved performance, including scalability potential, favor this algorithm, the existence of a straightforward sequential to parallel transformation strategy reduces the disadvan-

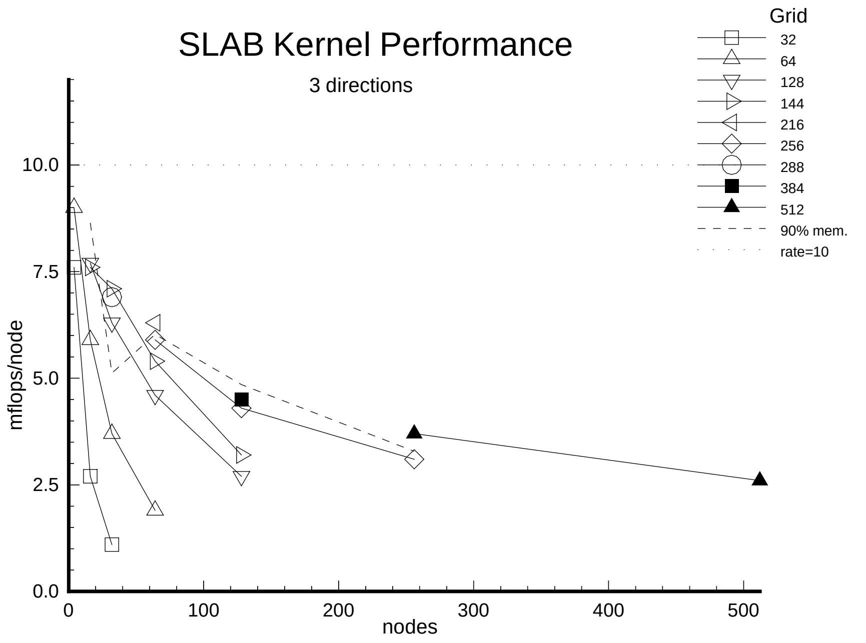

Figure 26: SLAB kernel performance - rate per node for all directions

tage of being more complicated. If code development tools were available to assist the user in implementing these transformations, the complexity of the final code would be less of an issue.

The transformation strategy outlined above should be extendible to many other programming situations. For example, when the tridiagonal matrix to invert is not periodic, it is possible after a reordering of the elements within the array, to derive a balanced algorithm similar to the one described here. The price to pay will be somewhat increased communication costs. If parallel, distributed memory computing is to become practical in a production code environment, the generation of parallel code must be done via systematic programming procedures rather than via "creative" programming. Only when parallel code can be so generated will compilers be able to efficiently convert sequential-computer style code to parallel-distributed-memory style code.

8 Acknowledgments

This research was performed in part using the Intel Touchstone Delta System operated by Caltech on behalf of the Concurrent Supercomputing Consortium. Access to this facility was provided by NASA Langley Research Center.

References

[1] Bokhari, S. H., 1991, Complete Exchange on the iPSC-860. *ICASE Report no. 91-4*, Institute for Computer Applications in Science and Engineering, Hampton, VA.

[2] Erlebacher, G., Hussaini, M.Y., Kreiss, H.O., and Sarkar, S., The analysis and simulation of compressible turbulence. 1990, *Theoret. Comput. Fluid Dynamics*, **2**, 73.

[3] Fox, G., Johnson, M., Lyzenga, G., Otto, S., Salmon, J. and Walker, D. 1988, *Solving problmes on concurrent processors*, Prentice-Hall, Englewood Cliffs, New Jersey.

[4] Hockney, R. W., and Jesshope, C. R. 1981, *Parallel Computing*, Adam Hilger, Bristol, Great Britain.

[5] Lillevik, S. L. 1991, *The Touchstone 30 Gigaflop DELTA Prototype*, **DMCC91**, IEEE Press, 671-677.

[6] Margulis, N. 1990, *i860 Microprocessor Architecture*, Osborne McGraw-Hill, Berkley, California.

[7] Wei, S. K., Compact Finite-difference schemes with spectral-like resolution. 1992, *J. Comp. Physics* **103**.

[8] Sun, X. H. and Gustafson, L.G., 1991, Toward a better parallel performance metric. *Parallel Computing*, **17**, 1093.

[9] Sun, X. H. and Ni, L. M., 1992, Scalable Problems and Memory-Bounded Speedup. *ICASE Report no. 92-59*, Institute for Computer Applications in Science and Engineering, Hampton, VA.

9 Appendix A - Communication Estimates

Communication estimates for two algorithms for the CDNS derivative kernel are discussed in Section 4 of this paper. The development of those estimates is given in this appendix.

The derivative kernel was developed to accept a three dimensional array containing the values of some variable at each node of a three dimensional, uniformly spaced physical grid. The kernel computes the partial derivatives of that variable using a 6th order, compact, finite-differencing scheme[7]. This results in the need to solve a tridiagonal system of equations with multiple right-hand sides. The original data is assumed to have periodic boundary conditions.

The estimates given below are for three calls to the kernel—one requesting a derivative in each of the three physical directions. As discussed in the previous sections, a straightforward partitioning strategy is taken. The various strategies evaluated, each assign equally one or more dimensions of the 3-D array to the available nodes. Estimates are based on the available nodes also being allocated equally to each distributed dimension. The actual algorithms discussed in this paper are not limited by this restriction.

9.1 Definitions

L	$=$	number of grid points in each dimension
A	$=$	size of one 3-D array
N	$=$	total number of nodes
n	$=$	number of nodes allocated to each distributed dimension
m	$=$	total number of messages
s	$=$	size of one message
t	$=$	total size of data moved
f	$=$	fraction of one array moved

$$\begin{aligned} A &= L^3 \\ t &= ms \\ f &= t/A \end{aligned}$$

9.2 Chained algorithm

The chained algorithm works as follows:

- The 3-D array that is passed to the derivative kernel is treated conceptually as a 2-D array. The dimension of the 3-D array that corresponds to the direction for which a derivative is desired is mapped to the dependent dimension of the 2-D conceptual array (R, input or S, output; see Table 1 and Figure 1). The dependent dimension is the direction of the vectors (input,r and solution,s) that are solved by the tridiagonal system. The other two dimensions of the 3-D array map to the second dimension of the 2-D array. It is referred to as the independent dimension since it indexes the multiple input vectors and multiple solutions vectors, which can be solved independently of each other.

- It is useful to view the 2-D array as stripmined in both dimensions. The strips in the independent dimension identify groups of vectors that are lumped together to form an independent sub-problem. The strips in the dependent dimension correspond to the pieces of the array that are distributed to the various nodes. See Figure 27. All sub-problems are computationally equivalent and thus any message estimates can be made for one sub-problem and then multiplied by the number of sub-problems to get overall estimates.

- For the 2-D and 3-D case, the independent dimension is also distributed. Each of these slices (2-D case, see Figure 29) or tubes (3-D case, see Figure 30), formed as a result of the independent dimension partioning, are similar to the 1-D case, where the partitioning is only in the dependent dimensions.

- The algorithm for each sub-problem starts on one of the n sub-strips and proceeds in a chained fashion to each of the other sub-strips. This chain must be completed twice—once for the forward and once for the backward pass of the tridiagonal solver. The algorithm proceeds on each node in a manner similar to the sequential, Gaussian elimination algorithm[4], but for only the data located on that node. During a message passing step, the last row of data is passed to the next node. Because the problem is periodic, a second row of data is also passed.

$n-1$ message passing steps are needed for each pass of the tridiagonal solver portion of the algorithm. Two message passing steps are also needed for some preliminary calculations involving the R-array before the solver is executed. Combining these two stages results in n total message passing steps per pass. Several simplifications are not discussed which only affect lower order terms of the estimates. The chaining algorithm requires that the number of subproblems equals the number of nodes in the dependent dimension for a load balanced algorithm to result (see Figure 8). In the following subsections, estimates for 3 data distributions are made—distributing the original 3-D array in one, two and three of its dimensions.

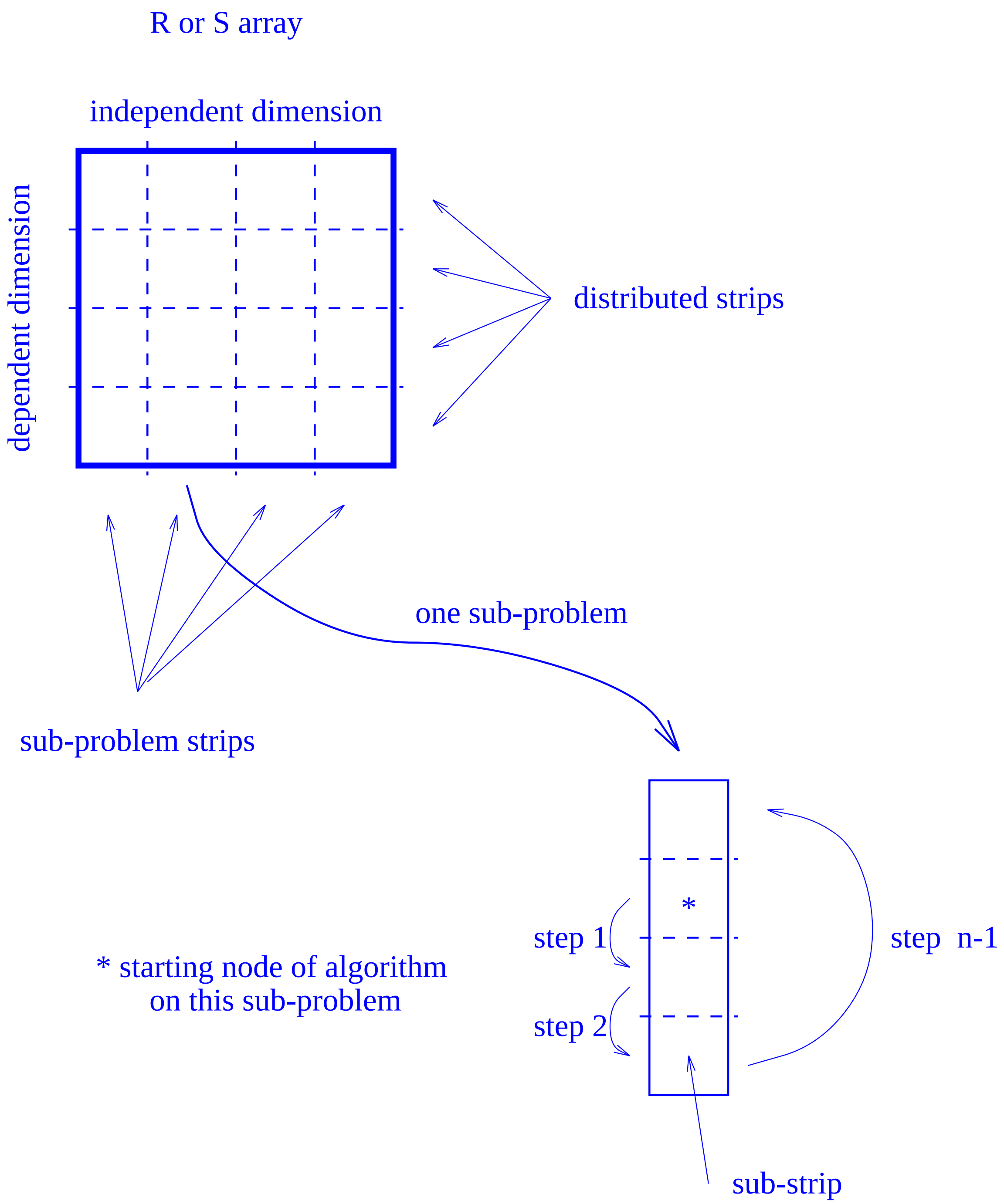

Figure 27: Chained algorithm steps

9.3 Chained algorithm: 1-D partitioning

See Figure 28.

- number of nodes in each distributed dimension

$$n = N$$

- compute number of messages

$$\begin{aligned} m &= 1 \quad \textit{distributed direction} \\ &\quad * \ n \quad \textit{subproblems} \\ &\quad * \ 2 \quad \textit{passes(forward \& backward)} \\ &\quad * \ n \quad \textit{steps/pass} \\ &\quad * \ 2 \quad \textit{messages/step} \\ m &= 4n^2 \quad \textit{messages} \end{aligned}$$

- compute message size

$$\begin{aligned} s &= L^2 \quad \textit{size of independent dimension} \\ &\quad / \ n \quad \textit{subproblem strips} \\ s &= L^2/n \quad \textit{size of each message} \end{aligned}$$

- results

$$\begin{aligned} m &= 4N^2 \\ s &= L^2/N \\ t &= 4NL^2 \\ f &= 4N/L \end{aligned}$$

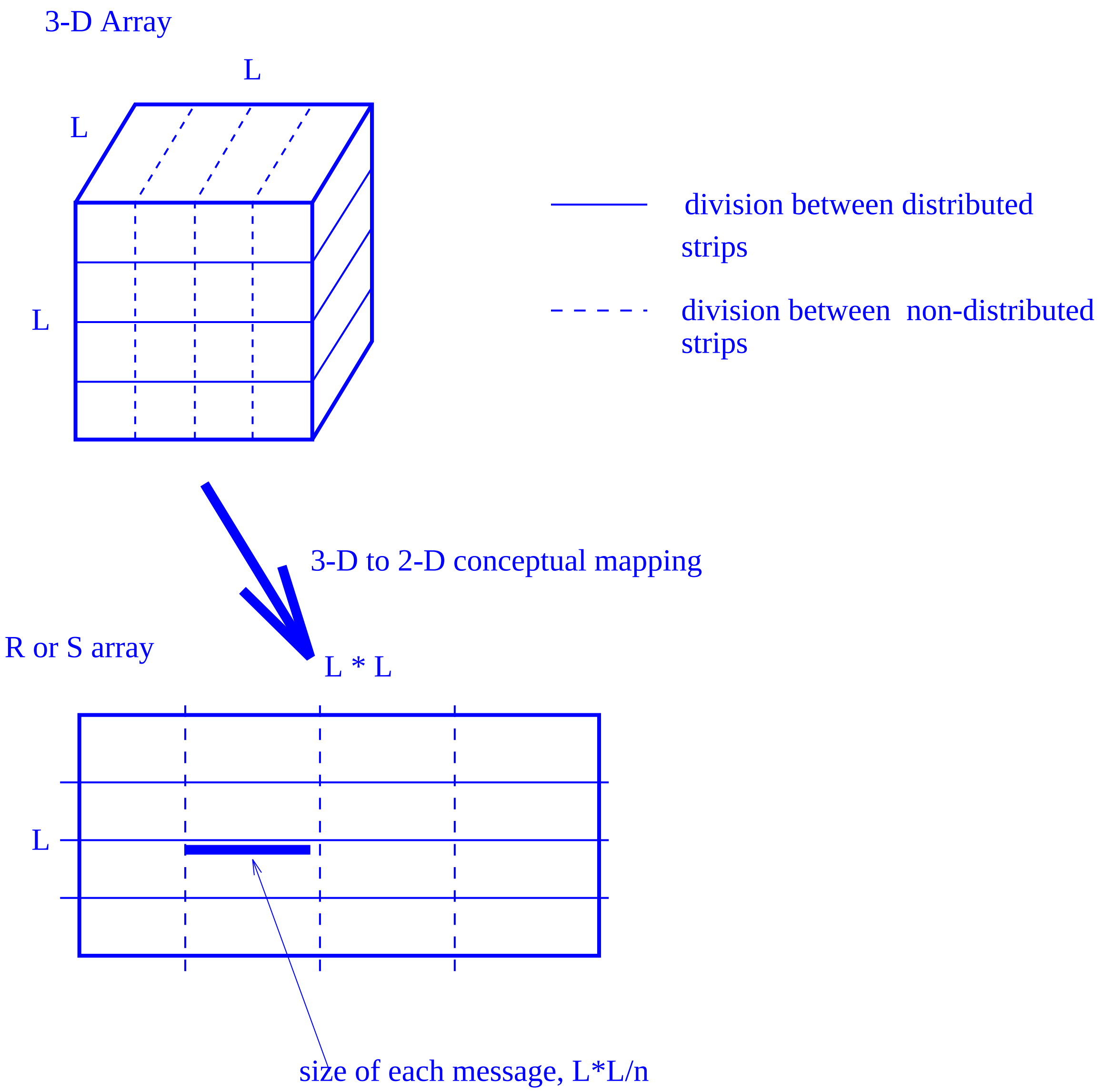

Figure 28: Chained algorithm for a 1-D data distribution

9.4 Chained algorithm: 2-D partitioning

See Figure 29.

- number of nodes in each distributed dimension

$$n = N^{1/2}$$

- compute number of messages

$$\begin{aligned}
m &= \quad 2 \quad \textit{distributed directions} \\
&\quad * \ n \quad \textit{subproblems/slice} \\
&\quad * \ n \quad \textit{slices} \\
&\quad * \ 2 \quad \textit{passes/subproblem} \\
&\quad * \ n \quad \textit{steps/pass} \\
&\quad * \ 2 \quad \textit{messages/step} \\
m &= \quad 8n^3 \quad \textit{messages}
\end{aligned}$$

- compute message size

$$\begin{aligned}
s &= \quad L^2/n \quad \textit{size of slices (independent dimension)} \\
&\quad / \ n \quad \textit{subproblem strips} \\
s &= \quad L^2/n^2 \quad \textit{size of each message}
\end{aligned}$$

- results

$$\begin{aligned}
m &= 8N^{3/2} \\
s &= L^2/N \\
t &= 8N^{1/2}L^2 \\
f &= 8N^{1/2}/L
\end{aligned}$$

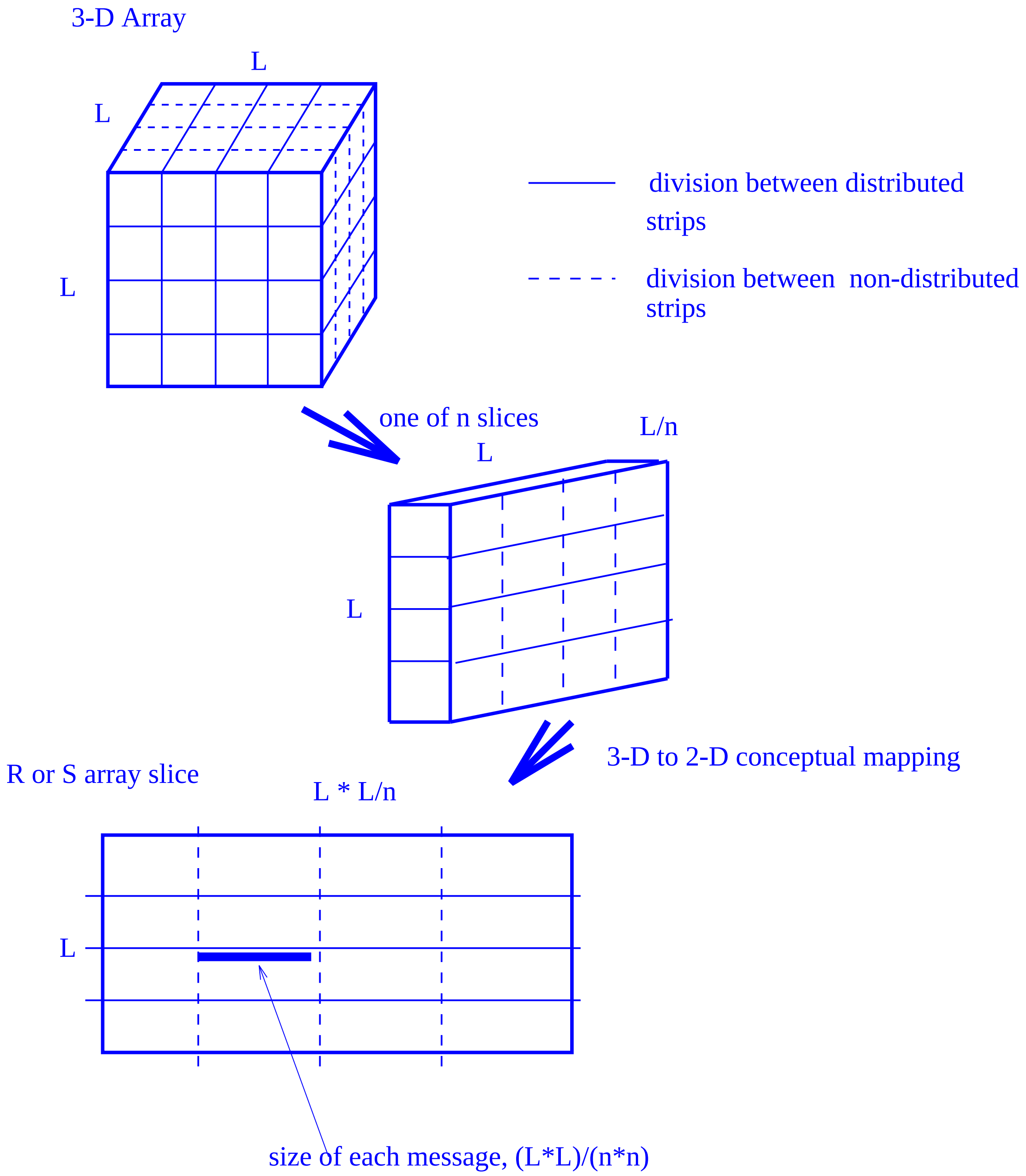

Figure 29: Chained algorithm for a 1-D data distribution

9.5 Chained algorithm: 3-D partitioning

See Figure 30.

- number of nodes in each distributed dimension

$$n = N^{1/3}$$

- compute number of messages

$$\begin{aligned}
m = &\quad 3 \quad \textit{distributed directions} \\
& * \ n \quad \textit{subproblems} \\
& * \ n^2 \quad \textit{tubes} \\
& * \ 2 \quad \textit{passes/subproblem} \\
& * \ n \quad \textit{steps/pass} \\
& * \ 2 \quad \textit{messages/step} \\
m = &\quad 12n^3 \quad \textit{messages}
\end{aligned}$$

- compute message size

$$\begin{aligned}
s = &\quad L^2/n^2 \quad \textit{size of tubes (independent dimension)} \\
& / \ n \quad \textit{subproblem strips} \\
s = &\quad L^2/n^3 \quad \textit{size of each message}
\end{aligned}$$

- results

$$\begin{aligned}
m &= 12N^{4/3} \\
s &= L^2/N \\
t &= 12N^{1/3}L^2 \\
f &= 12N^{1/3}/L
\end{aligned}$$

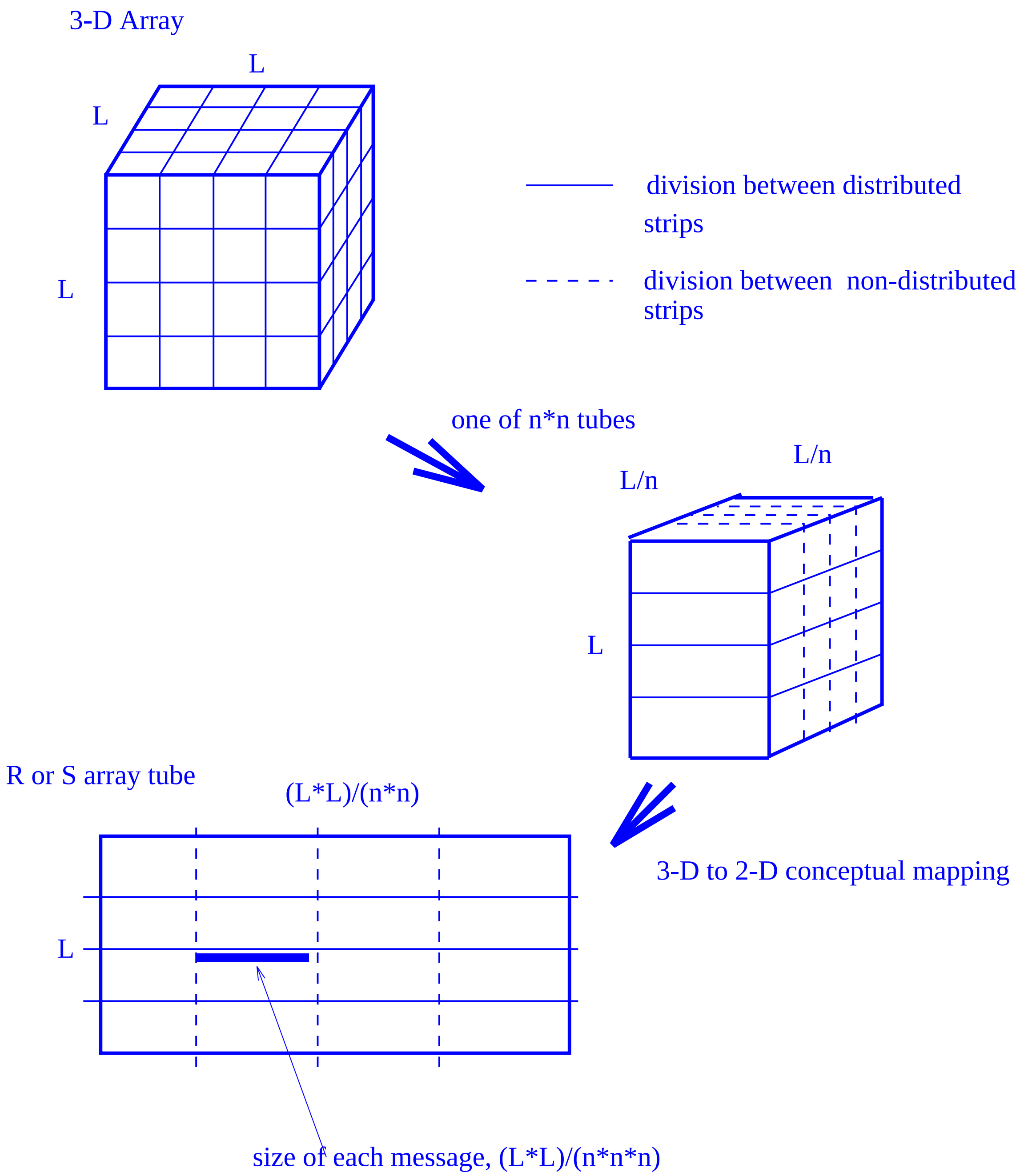

Figure 30: Chained algorithm for a 1-D data distribution

9.6 Transpose algorithm

The transpose algorithm works as follows:

- The conceptual mapping between the 3-D array and the R or S array is similar to the chained algorithm.
- Assuming that the j-dimension in Figure 18 is the dependent dimension, the R array must be transposed so that each j-vector resides on a single node. Principally, this involves the swapping of blocks of data as shown in Figure 18 which is referred to as a global transpose. For on-node performance reasons, each block is also transposed to generate a complete transpose. The on-node data motion is not included in these estimates since it can partially be hidden behind the inter-node communications.
- Once the input (R-array) is transposed, a sequential solver algorithm is used to generate the output (S array). The S-array must then be transposed back to the original data distribution.

In the following subsections, estimates for 3 data distributions are made—distributing the original 3-D array in one, two and three of its dimensions. The transpose algorithm does not divide the R and S arrays into subproblems as is done in the chained algorithm. The use of subproblem in the following estimates refers to the number of slices (2-D case) or tubes (3-D case).

9.7 Transpose algorithm: 1-D partitioning

See Figure 31.

- number of nodes in each distributed dimension

$$n = N$$

- compute number of messages

$$\begin{aligned} m &= \quad 1 \quad \textit{distributed direction} \\ &\quad * \; 2 \quad \textit{transposes} \\ &\quad * \; n \quad \textit{blocks(messages) per step} \\ &\quad * \; n-1 \quad \textit{steps} \\ m &= \quad 2n(n-1) \quad \textit{messages} \end{aligned}$$

- compute message size

$$\begin{aligned} s &= \quad L^2 \quad \textit{size of independent dimension} \\ &\quad / \; n \quad \textit{strips} \\ &\quad * \; L \quad \textit{size of dependent dimension} \\ &\quad / \; n \quad \textit{strips} \\ s &= \quad L^3/n^2 \quad \textit{size of each block (message)} \end{aligned}$$

- results (neglecting the lower order terms of m)

$$\begin{aligned} m &= 2N^2 \\ s &= L^3/N^2 \\ t &= 2L^3 \\ f &= 2 \end{aligned}$$

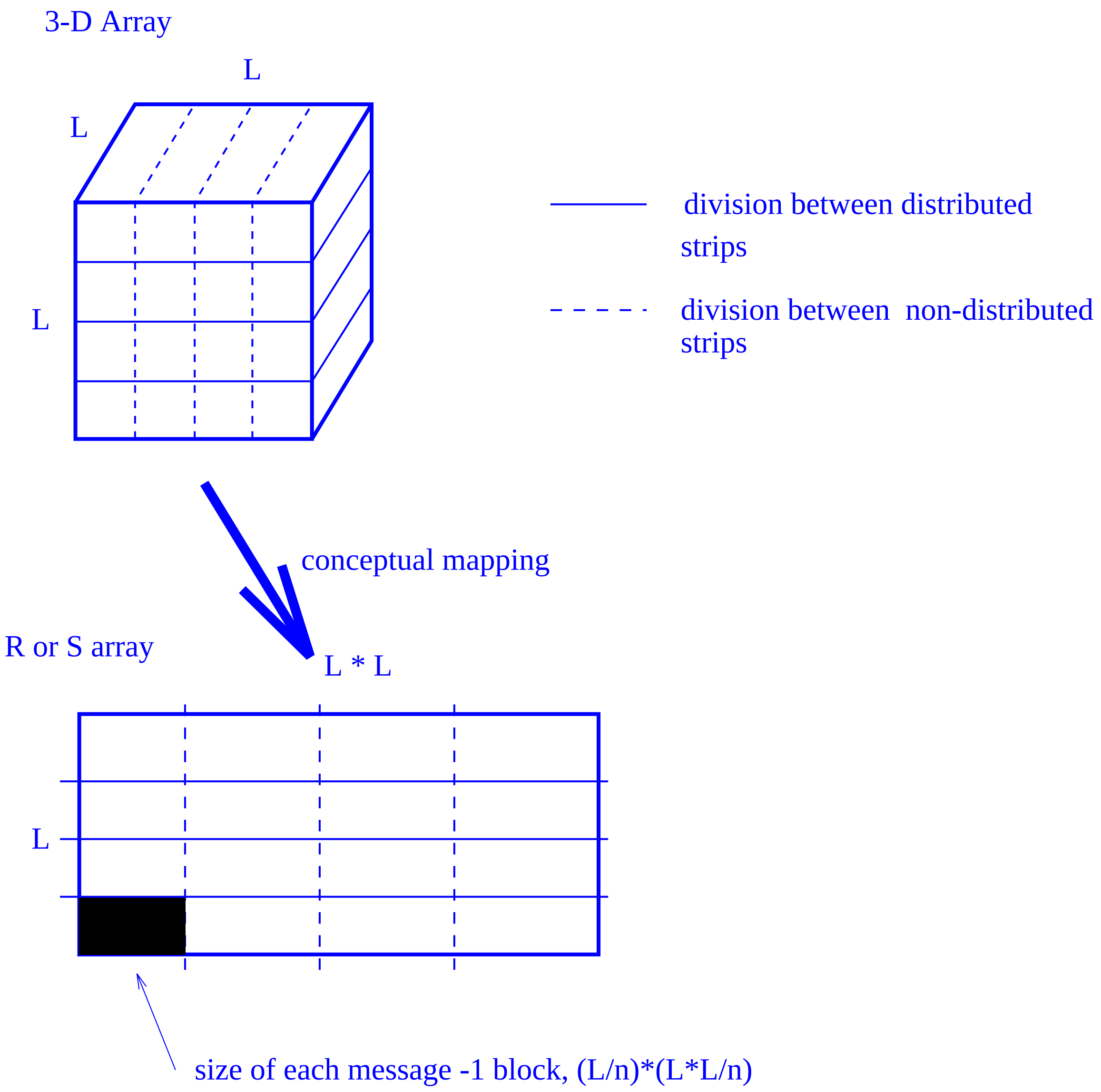

Figure 31: Transpose algorithm for a 1-D data distribution

9.8 Transpose algorithm: 2-D partitioning

See Figure 32.

- number of nodes in each distributed dimension

$$n = N^{1/2}$$

- compute number of messages

$$\begin{aligned}
m = &\quad 2 \quad \textit{distributed directions} \\
&* \; 2 \quad \textit{transposes} \\
&* \; n \quad \textit{blocks(messages) per step} \\
&* \; n-1 \quad \textit{steps} \\
&* \; n \quad \textit{subproblems or slices} \\
m = &\quad 4n^2(n-1) \quad \textit{messages}
\end{aligned}$$

- compute message size

$$\begin{aligned}
s = &\quad L^2 \quad \textit{size of independent dimension of each slice} \\
&/ \; n^2 \quad \textit{strips} \\
&* \; L \quad \textit{size of dependent dimension} \\
&/ \; n \quad \textit{strips} \\
s = &\quad L^3/n^3 \quad \textit{size of each block (message)}
\end{aligned}$$

- results (neglecting the lower order terms of m)

$$\begin{aligned}
m &= 4N^{3/2} \\
s &= L^3/(N^{3/2}) \\
t &= 4L^3 \\
f &= 4
\end{aligned}$$

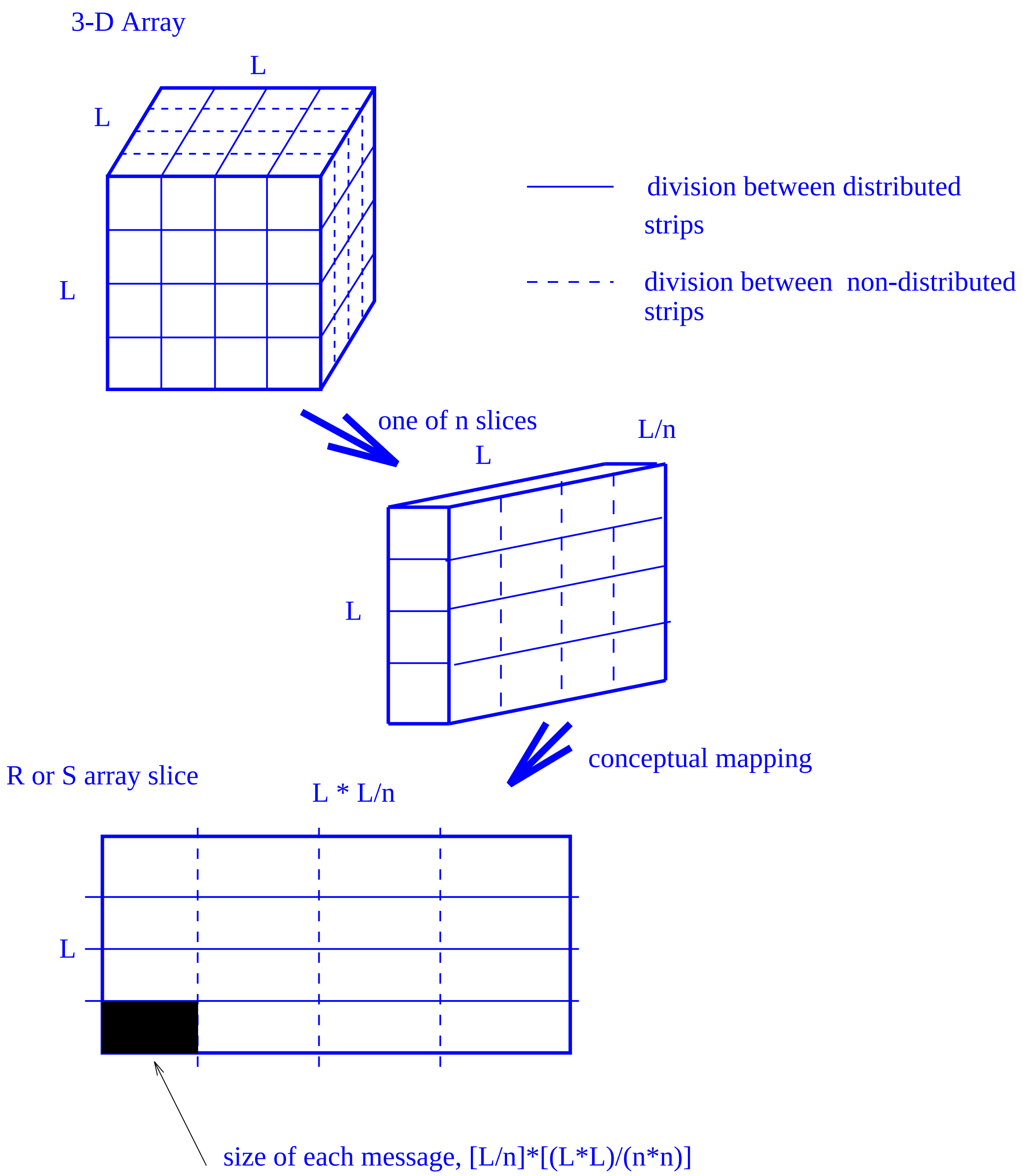

Figure 32: Transpose algorithm for a 1-D data distribution

9.9 Transpose algorithm: 3-D partitioning

See Figure 33.

- number of nodes in each distributed dimension

$$n = N^{1/3}$$

- compute number of messages

$$\begin{aligned}
m &= \quad 3 \quad \textit{distributed directions} \\
& \quad * \; 2 \quad \textit{transposes} \\
& \quad * \; n \quad \textit{blocks(messages) per step} \\
& \quad * \; n-1 \quad \textit{steps} \\
& \quad * \; n^2 \quad \textit{subproblems or tubes} \\
m &= \quad 6n^2(n-1) \quad \textit{messages}
\end{aligned}$$

- compute message size

$$\begin{aligned}
s &= \quad L^2 \quad \textit{size of independent dimension of each tube} \\
& \quad / \; n^3 \quad \textit{strips} \\
& \quad * \; L \quad \textit{size of dependent dimension} \\
& \quad / \; n \quad \textit{strips} \\
s &= \quad L^3/n^4 \quad \textit{size of each block (message)}
\end{aligned}$$

- results (neglecting the lower order terms of m)

$$\begin{aligned}
s &= L^3/(N^{4/3}) \\
m &= 6N^{4/3} \\
t &= 6L^3 \\
f &= 6
\end{aligned}$$

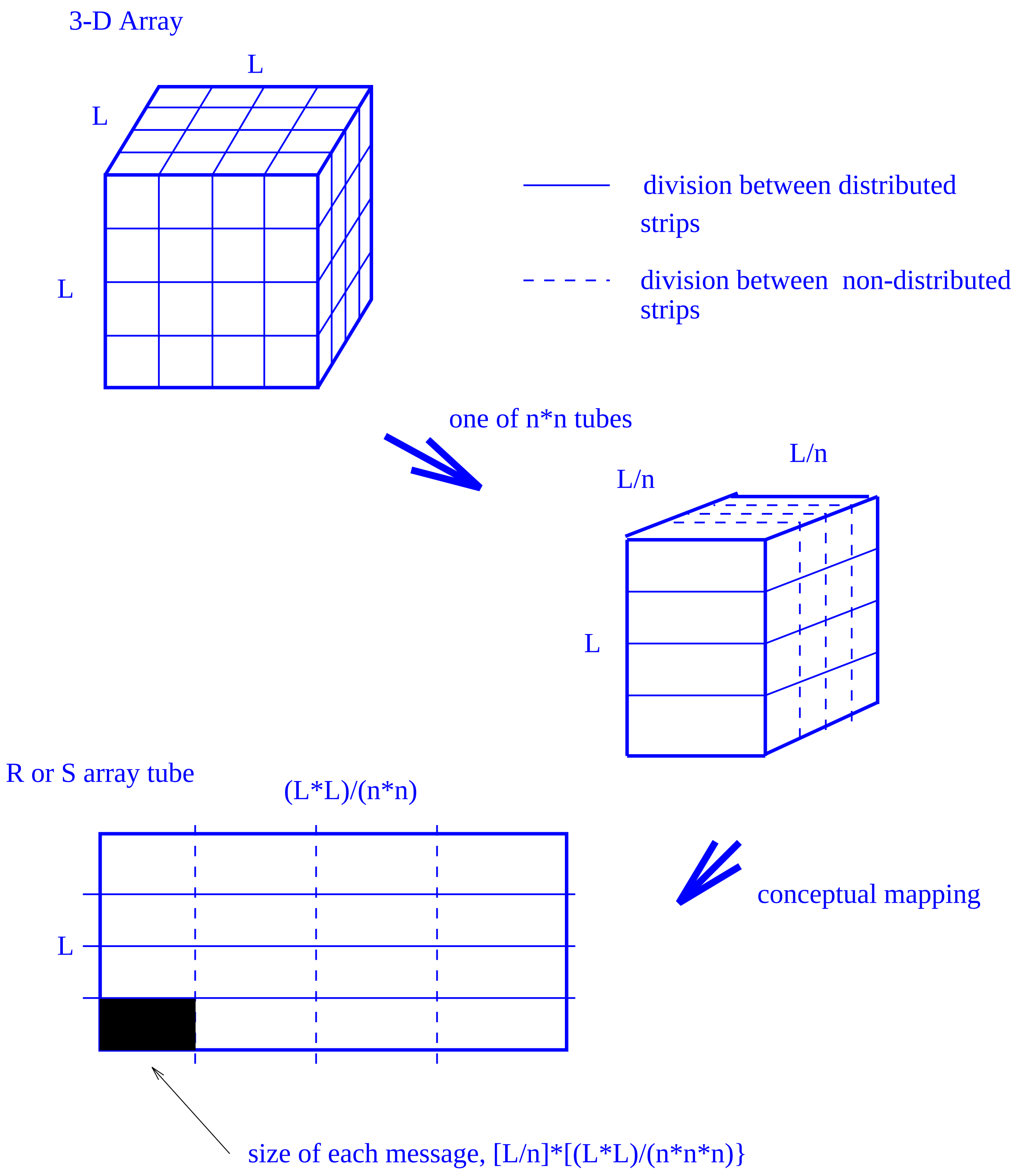

Figure 33: Transpose algorithm for a 1-D data distribution