Syracuse University
SURFACE

College of Engineering and Computer Science - Former Departments, Centers, Institutes and Projects

College of Engineering and Computer Science

1996

Array Decompositions for Nonuniform Computational Environments

Maher Kaddoura
Syracuse University

Sanjay Ranka
Syracuse University

Albert Wang
Syracuse University

Follow this and additional works at: http://surface.syr.edu/lcsmith_other

 Part of the Computer Sciences Commons

Recommended Citation

Kaddoura, Maher; Ranka, Sanjay; and Wang, Albert, "Array Decompositions for Nonuniform Computational Environments" (1996). *College of Engineering and Computer Science - Former Departments, Centers, Institutes and Projects.* Paper 5.
http://surface.syr.edu/lcsmith_other/5

This Article is brought to you for free and open access by the College of Engineering and Computer Science at SURFACE. It has been accepted for inclusion in College of Engineering and Computer Science - Former Departments, Centers, Institutes and Projects by an authorized administrator of SURFACE. For more information, please contact surface@syr.edu.

Array Decompositions for Nonuniform Computational Environments

Maher Kaddoura, Sanjay Ranka, and Albert Wang
4-116 Center for Science and Technology
School of Computer and Information Science
Syracuse University
Syracuse, NY 13244-4100
kaddoura@top.cis.syr.edu, ranka@top.cis.syr.edu
atwang@top.cis.syr.edu

June 1993 (revised February 1995)

submitted to
Journal of Parallel and Distributed Computing

Abstract

Two-dimensional arrays are useful in a large variety of scientific and engineering applications. Parallelization of these applications requires the decomposition of array elements among different machines. Several data-decomposition techniques have been studied in the literature for machines with uniform computational power. In this paper we develop new methods for decomposing arrays into a cluster of machines with nonuniform computational power. Simulation results show that our methods provide superior decomposition over naive schemes.

1 Introduction

Data-parallel applications requires the partitioning of data among processors in a way that the computation load on each node is proportional to its computational power, while minimizing communication. Two-dimensional arrays are widely used in scientific and engineering problems such as weather prediction and image processing. In this paper we discuss the decomposition of two-dimensional arrays for a nonuniform computational environment (NUCE). This environment is made up of machines with different computational power, and the arrays must therefore be divided unequally among processors.

Most previous work has been done on parallel machines with uniform computational power. For these machines, decomposition of arrays is relatively simple. The array is partitioned equally along one or more dimensions. This is reflected in the BLOCK directive of High-Performance Fortran [1]. Belkhale and Banerjee [5] have proposed methods for partitioning a nonuniform grid. Grid problems with varying resource demands have been studied by Nicol and Saltz [13]. Nicol [12] has described the decomposition of unstructured grids into perfectly rectilinear partitions.

There has been very little work done on decomposition of arrays for nonuniform machines. Crandall and Quinn [10] have presented an algorithm for decomposing a two-dimensional data array for a heterogeneous cluster of workstations. This algorithm uses an orthogonal recursive bisection to perform the decomposition. Snyder and Socha [15] have developed a polynomial time algorithm for allocating an $I \times J$ array of array points to a $K \times K$ array of processors. However, their algorithm is restricted to uniform processors. Berger and Bokhari [6] have addressed the problem of decomposing a domain into multiprocessors where subparts of the domain require different computational loads. The domain is divided into rectangles by recursively dividing the partitions into subpartitions of equal computational load until each processor has been allocated its share of the domain.

Although our work is specifically targeted towards a cluster of workstations, it can be generalized to a heterogeneous computing environment consisting of clusters of workstations, clusters of supercomputers, or a hybrid of both. An example of such an environment is shown in Figure 1. For such cases a multilevel decomposition may be required. An array may be decomposed among several higher-level machines, followed by decomposition within each machine.

The algorithms developed in this paper assume that all the processors are connected by an interconnection network in which the cost of unit communication is the same between all the processors (e.g., a bus). We show that the number of possible decompositions is at least $O(p!2^p)$, where p is the number of processors. Thus, trying all possible cases is not possible beyond a small value of p. Our simulation results show that the total amount of communication produced by our algorithms is smaller as compared to naive approaches.

The rest of this paper is organized as follows. Section 2 provides a formal description of the problem. In Section 3 we discuss the decomposition of a two-dimensional $m \times n$ array into k ($2 \leq k \leq 4$) partitions. Section 4 introduces a special class of decompositions. In Section 5 we discuss the effect of latency on the algorithms described in Section 4. In Section 6, several decomposition algorithms are presented and performance comparison of the algorithms is presented.

2 Problem definition

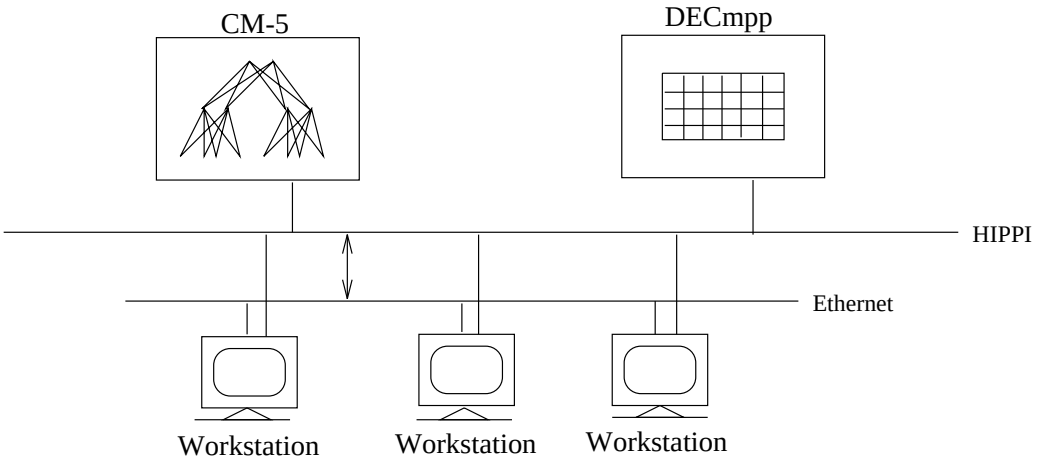

Figure 1: A nonuniform computing environment.

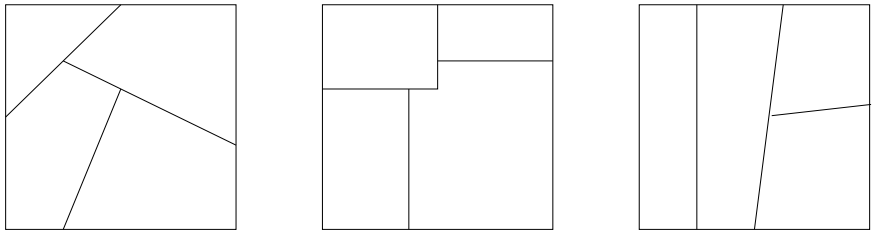

Figure 2: Examples of some decompositions that are not allowed.

NUCE's are made up of processors with different computational power connected by a network. A special case of a NUCE is a cluster of heterogeneous workstations that are connected by Ethernet or Token Ring. An efficient decomposition would minimize interprocessor communication while simultaneously balancing the computational load among the processors in NUCE. We consider only decompositions that assign to each processor a rectangular (or square) subarray, thus decompositions such as are shown in Figure 2 those not allowed. This restriction is necessary for two reasons:

1. The local data assigned to each node is an array, making storing and referencing of data on a local processor efficient.

2. The boundaries of the local subarray are parallel to the length and the width, which makes finding the owner of a particular element relatively simple.

Several ways of decomposing a 2-D array for a four-processor NUCE such that each partition satisfies the above property are shown in Figure 3. Each decomposition can be looked at as a planar graph with a number of nodes and edges. Edges can be classified as internal edges or external edges. Internal edges are shown by dotted lines in Figure 3. All other edges are external edges. We would like to divide the array in such a fashion that the total communication cost is

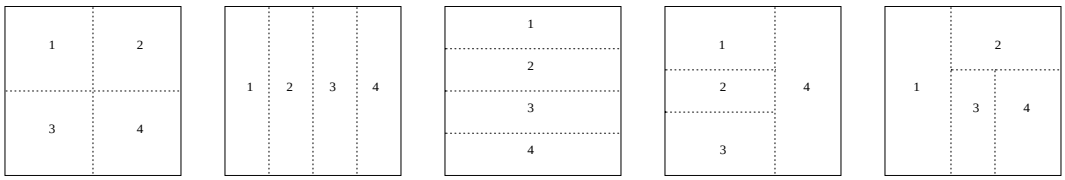

Figure 3: Different decompositions for four processors in a NUCE.

minimized. Clearly, the total communication cost depends on the way data is accessed for a given application. However, for most scientific applications the cost is minimized if the sum of the length of internal edges (representing the communication) is minimized. We will use the term *Acost* to represent this sum. It is worth noting that the minimization of *Acost* reduces the sum of the perimeter of all the rectangles, thus a decomposition that provides the best partitioning would, on the average, have each partition close to a square. In the above discussion we have assumed that the cost of communication for a unit of data is the same across all machines.

In some cases the sum of internal edges, along with the external edges, is a better representation of communication cost. This is required for cases in which interactions are wraparound along rows and/or columns. If two parallel external edges belong to the same partition, then they are not counted in the cost since they do not result in any communication. We will use the term *Bcost* to represent this sum. In Figure 4 *Acost* is $m + n'$ and *Bcost* is $2m + 2n'$.

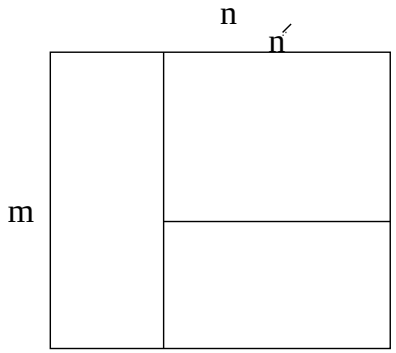

Figure 4: $Acost = m + n'$, $Bcost = 2m + 2n'$.

Let $a_1, a_2, a_3 \ldots, a_p$ be the relative computational power of different processors such that $a_1 + a_2 + a_3 \ldots a_p = 1$ and $a_1 \leq a_2 \leq a_3 \leq a_4 \ldots . \leq a_p$. The problem can then be stated formally as follows:

> Decompose the $m \times n$ array into p partitions such that the size of each partition is in the ratio $a_1 : a_2 : a_3 : a_4 \ldots a_p$, and *Acost* (or *Bcost*) is minimized.

In practice the cost of communication on a local area network can be modeled as $O(\tau + \mu B)$, where τ is the software overhead of sending a message (also known as the startup latency), μ is the transfer rate and B is the size of the message. The startup latency for communication is typically large and can have an impact on the total cost of the communication. The cost measures studied so far do not consider this setup time. We will discuss the effect of startup latency in Section 5.

3 Decomposition for a small number of partitions

In this section we present several ways of decomposing a two-dimensional $m \times n$ array into k $(2 \leq k \leq 4)$ partitions such that the size of each partition is $a_i mn, 1 \leq i \leq k$. Our basic intention is to demonstrate some key ideas that could be used for the development of algorithms for large values of k.

Two partitions

There are four possible ways to decompose an array into two partitions (Figure 5). The decomposition in Figure 5 (a) has *Acost* equal to m and *Bcost* equal to $2m$. For decomposition in Figure 5 (b) *Acost* is n and *Bcost* is $2n$. Since $m \leq n$, decomposition in Figure 5 (a) is always better than decomposition in Figure 5 (b) using either measure. The cost of decomposition in Figure 5 (c) and Figure 5 (d) is equal to that of Figure 5 (a) and Figure 5 (b), respectively.

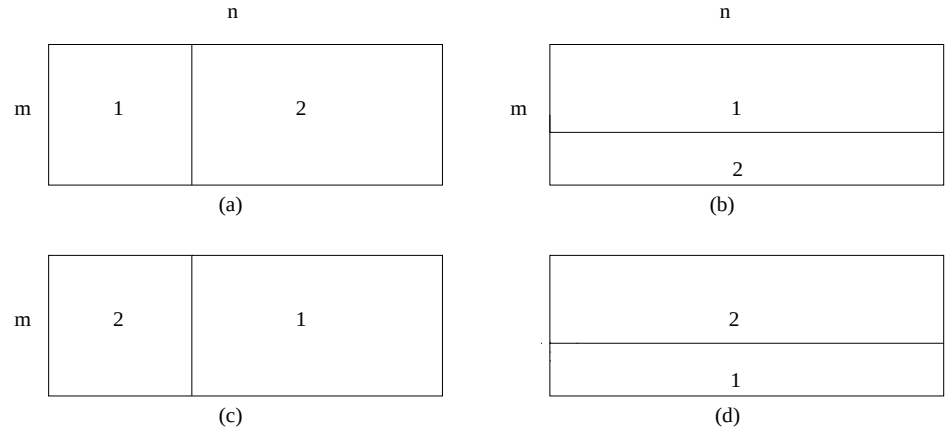

Figure 5: Decomposing an array into two partitions (a) $Acost = m$, $Bcost = 2m$; (b) $Acost = n$, $Bcost = 2n$.; (c) $Acost = m$, $Bcost = 2m$. (d) $Acost = n$, $Bcost = 2n$.

Three partitions

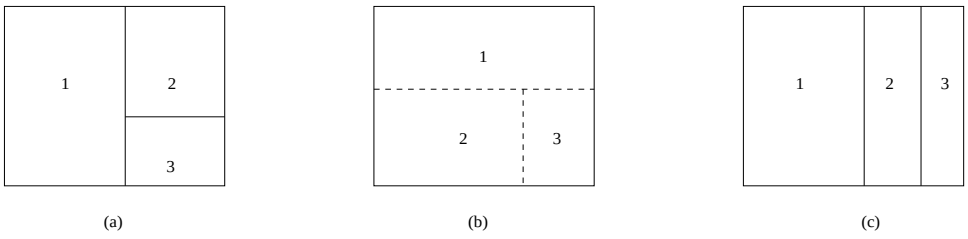

Figure 6: Partitioning an array into three partitions (a) $Acost = m + n(a_2 + a_3)$, $Bcost = 2m + 2n(a_2 + a_3)$; (b) $Acost = n + m(a_2 + a_3)$, $Bcost = 2n + 2m(a_2 + a_3)$; (c) $Acost = 2m$, $Bcost = 3m$.

There are 36 ways to decompose an array into three partitions. Three representative decompositions are shown in Figure 6. The decomposition in Figure 6 (a) has *Acost* equal to $m + n(a_2 + a_3)$ and *Bcost* equal to $2m + 2n(a_2 + a_3)$. Decomposition derived by interchanging a_1 with a_2 or a_3 will

increase the total cost. For example, if we interchange a_1 with a_2, then *Acost* is $m + n(a_1 + a_3)$, $m + n(a_1 + a_3) \geq m + n(a_2 + a_3)$, because $a_1 \geq a_2$. Also, the decomposition achieved by rotating (interchanging the orientation of internal edges by 90 degrees) the decomposition in Figure 6 (a) will result in an increase of communication cost (Figure 6 (b)). For example, the decomposition in Figure (b) has *Acost* equal to $n + m(a_2 + a_3)$, $n + m(a_2 + a_3) \geq m + n(a_2 + a_3)$, because $n \geq m$. Similarly, rotating the decomposition in Figure 6 (c) will lead to a larger communication cost. *Acost* of decomposition in Figure 6 (a) is better than *Acost* of Figure 6 (c) if $m \geq n(a_2 + a_3)$.

The following observations can be made for achieving the decomposition with the least cost:

1. Partitions should be considered in decreasing order of their size.

2. Partitioning should be done along the longer dimension.

To find the best possible decomposition for three partitions, we need to consider only cases (a) and (c) and choose the better of the two. This is a significant reduction from the 36 possible cases.

Four partitions

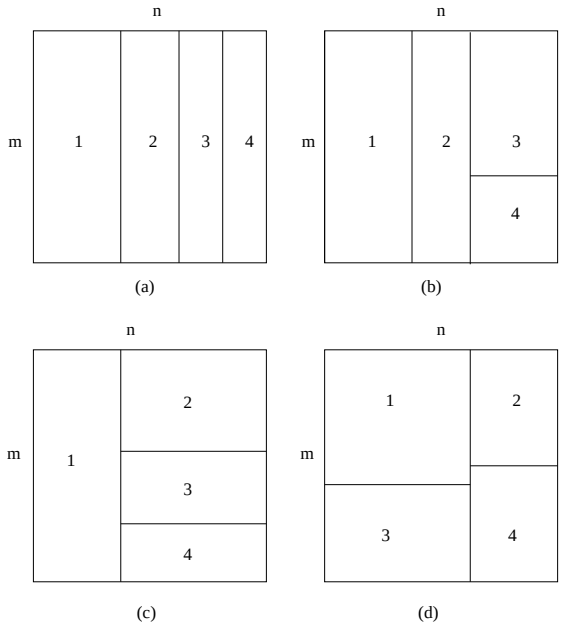

Figure 7: Partitioning an array into four partitions (a) $Acost = 3m$, $Bcost = 4m$; (b) $Acost = 2m + n(a_3 + a_4)$, $Bcost = 3m + 2n(a_3 + a_4)$; (c) $Acost = m + 2n(a_2 + a_3 + a_4)$, $Bcost = 2m + 3n(a_2 + a_3 + a_4)$; (d) $Acost = m + n$, $Bcost = 2m + 2n$.

There are at least 384 possible ways to decompose an array into four partitions (Figure 7 shows four of them). If we rotate these decompositions we will get another four decompositions. For each of the eight decompositions, we can get 4! different decompositions by interchanging a_1, a_2, a_3, a_4 with each other. *Acost* and *Bcost* for these cases are shown in Figure 7. Interchanging a's in any of the decompositions shown will not reduce *Acost* or *Bcost*. For example, if we interchange a_1 with

a_2 in Figure 7 (c), then *Acost* is $m + 3n(a_1 + a_3 + a_4)$, $m + 3n(a_1 + a_3 + a_4) \geq m + 3n(a_2 + a_3 + a_4)$, because $a_1 \geq a_2$.

Another decomposition of an array into four partitions is shown in Figure 8. The structure of the decomposition in Figure 8 is different from the ones described in Figures 5 through 7, because the latter can be derived by partitioning along X-axis (Y-axis), followed by partitioning each of the subarrays along Y-axis (X-axis).

The *Acost* and *Bcost* of the decomposition in Figure 8 are $m + m(a_3 + a_4)/(a_2 + a_3 + a_4) + n(a_2 + a_3 + a_4)$, and $2m + m(a_3 + a_4)/(a_2 + a_3 + a_4) + 2n(a_2 + a_3 + a_4)$, respectively.

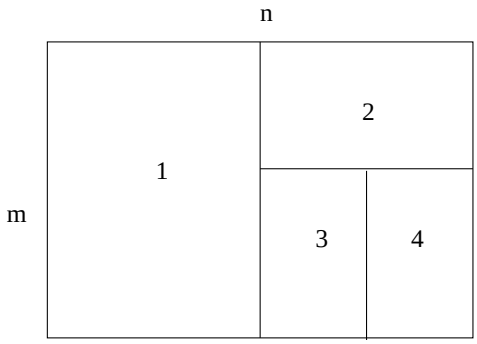

Figure 8: A different decomposition of an array into 4 partitions.

We get the decomposition in Figure 8 by rotating the dotted subpart of Figure 7 (b) (see Figure 9). The *Acost* of decomposition in Figure 8 is better than Figure 7 (b) only when

$$\begin{aligned} m(1 \quad &- \quad (a_3 + a_4)/(a_2 + a_3 + a_4)) - na_2 > 0 \\ &\Leftrightarrow \quad m(a_2/(a_2 + a_3 + a_4)) > na_2 \\ &\Leftrightarrow \quad m > (a_2 + a_3 + a_4)n. \end{aligned}$$

This corresponds to the fact that after the first division (removing a_1), the length of the remaining rectangle is less than the width, and the decomposition for the subpart corresponds to the best decomposition for $k = 3$ with a_2, a_3, and a_4. This does not hold for *Bcost*, because a_2 in Figure 7 (b) has a free communication edge along the width. For *Bcost*, the decomposition in Figure 8 is thus better than Figure 7 (b) only when

$$m(1 - (a_3 + a_4)/(a_3 + a_4 + a_2)) > 2(a_2)n$$

$$\Rightarrow m > 2(a_2 + a_3 + a_4)n.$$

Discussion

From the special cases which we studied in this section we make the following observations:

1. A good method for minimizing *Acost* seems to be to look at all the decompositions of the type given in Figure 7. These decompositions can be achieved by partitioning along the larger dimension into several subarray, followed by partitioning each of the subarrays along the other dimension.

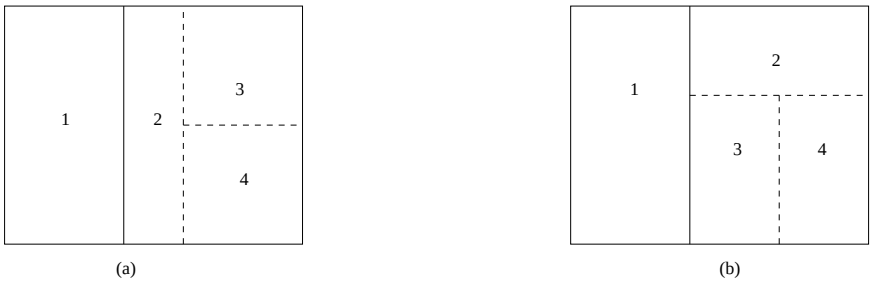

Figure 9: Decomposition in (b) can be achieved by rotating the dotted lines of decomposition in (a).

2. To achieve the decomposition with a good cost, the partitions should be considered in decreasing order of their sizes.

3. The partitioning should be done along the longer dimension. Whenever the remaining length is less than the remaining width, switching directions will generally, produce better decomposition.

For most decompositions, especially for a large number of partitions, the difference between *Acost* and *Bcost* will be equal to the perimeter of the array. Hence a good decomposition for one of the cost measures will also be a good decomposition for the other cost measures. The exceptions to the rule are the cases in which a given two parallel external edges belong to the same partition. In the following we will limit our discussion and analysis to only one of the cost measures, *Acost*. However, most of the discussion can potentially be extended to the other cost measure.

4 XY decompositions

Based on our observation in Section 3, we find that partitioning along the dimension with the larger size, followed by partitioning along the smaller size, leads to good decomposition in most cases. We will use the term XY *decompositions* to represent this class. Without loss of generality we will assume that the X-axis is larger of the two array dimensions. An example of such a decomposition for 9 partitions is given in Figure 10. We will use the term *vertical subarrays* to denote the subarrays formed by partitioning along the X-axis, and the term *horizontal subarrays* to denote the subarrays formed by partitioning each *vertical subarray* along the Y-axis. Each horizontal subarray corresponds to one partition and we will use these terms interchangeably. We make the following observations regarding XY decompositions. Let the number of vertical subarrays be v. We define $count_i, 1 \leq i \leq v$ to be the number of horizontal subarrays (partitions) assigned to the vertical subarray i.

Lemma 1 *Vertical subarrays can be interchanged so that they are in an increasing order of their count values without increasing the cost.*

Lemma 2 *The partitions within vertical subarrays can be arranged in a decreasing order of their size without increasing the cost.*

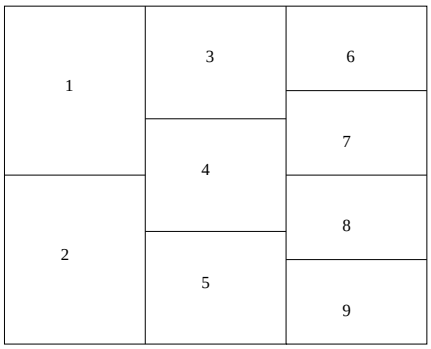

Figure 10: An XY decomposition of an array into 9 partitions

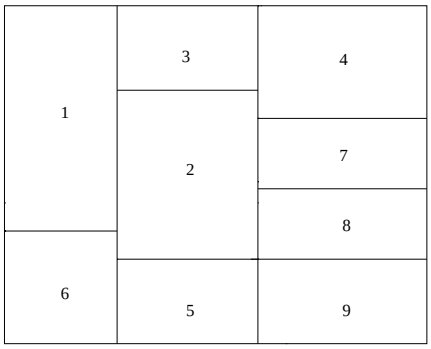

Figure 11: A rearrangement of partitions of the decomposition in Figure 10.

We define *rearranging the decomposition* as an operation in which partitions within a subset are interchanged without changing the number of vertical subarrays and without changing the count of each vertical subarray. For example, the decomposition in Figure 11 is a rearrangement of the decomposition in Figure 10.

Theorem 1 *Consider an XY decomposition of a two-dimensional array. The communication cost cannot be increased by rearranging the decomposition such that vertical subarrays are sorted according to their count values (number of partitions), and the partitions within and between the vertical subarrays are sorted according to size from top to bottom and left to right, respectively.*

Proof 1 Assume the vertical subarrays in the array are sorted according to the number of partitions in each vertical subarray (e.g., $(1,2),(3,4,5),(6,7,8,9)$). This can always be done without increasing the cost (Lemma 1). Then

$$Acost \ = (v-1)m + n\sum_{i=1}^{v}((count_i - 1)\sum_{j=1}^{count_i} a_{K_i+j})$$

where v is the number of vertical subarrays, $count_i$ is the number of partitions in the i-th vertical subarray, and $K_i = \sum_{x=1}^{i-1}(count_i)$.

To prove Theorem 1 we need only to prove the following two general cases:

1. $k_\alpha(a_x + a_y + \cdots) + k_\beta(a_i + a_j + \cdots) + \cdots = k_\alpha(a_y + a_x + \cdots) + k_\beta(a_i + a_j + \cdots) + \cdots$

p	$pt(p)$	$p!2^{p-1}$
4	5	192
5	7	1920
7	15	322560
10	42	1.857945e+09
20	627	1.275541e+24

Table 1: $pt(p)$ and $p!2^{p-1}$ for different p.

2. $k_\alpha(a_x + a_y + \cdots) + k_\beta(a_i + a_j + \cdots) + \cdots \leq k_\alpha(a_i + a_y + \cdots) + k_\beta(a_x + a_j + \cdots) + \cdots$

such that $k_\alpha \leq k_\beta$, $a_x \geq a_i$ and $a_x \geq a_y$.

The first case is always true, since addition is commutative. To prove the second case we need only to prove the following:

$$k_\alpha(a_x + a_y + \cdots) + k_\beta(a_i + a_j + \cdots) - k_\alpha(a_i + a_y + \cdots) - k_\beta(a_x + a_j + \cdots) \leq 0$$

$$\begin{aligned}
&\Leftrightarrow (k_\alpha a_x + k_\beta a_i) - (k_\alpha a_i + k_\beta a_x) \leq 0 \\
&\Leftrightarrow (k_\alpha a_x - k_\alpha a_i) + (k_\beta a_i - k_\beta a_x) \leq 0 \\
&\Leftrightarrow k_\alpha(a_x - a_i) + k_\beta(a_i - a_x) \leq 0 \\
&\Leftrightarrow (k_\alpha - k_\beta)(a_i - a_x) \leq 0,
\end{aligned}$$

which is always true since $k_\alpha \leq k_\beta$ and $a_x \geq a_i$. Thus, one can always move a horizontal subarray with a high value of a_i to the left by interchanging it with a horizontal subarray having a value lower than a_i without increasing the cost. □

The number of potentially different arrangements can be derived by finding the number of ways p can be partitioned into k parts, $1 \leq k \leq p$ such that $x_1 + x_2 + x_3 + \cdots x_k = p$, where x_1, x_2, x_3, ... , x_k are integers. This can be shown to be 2^{p-1}. For each such case there are $p!$ ways of arranging a_1, a_2, ..., a_p. Thus the total number of XY decompositions is $O(p!2^{p-1})$. Let $pt(p)$ represent the number of cases in which $x_1 \leq x_2 \leq x_3 \ldots \leq x_k$. For example, $pt(5)$ is equal to 7, namely, $5 = 4+1 = 3+2 = 3+1+1 = 2+2+1 = 2+1+1+1 = 1+1+1+1+1$. Theorem 1 thus reduces the possible cases to be considered for finding the best XY decomposition from $p!2^{p-1}$ to $pt(p)$, where p is the number of partitions. There is no closed-form expression for $pt(p)$ [9]. Table 1 gives the comparison for small values of p.

Lemma 3 *Let the communications cost of a decomposition produced by partitioning an $m \times n$ array along an X-axis followed by partitioning it along a Y-axis be*

$$Acost = dm + kn.$$

Then the communication cost of the same array with the same decomposition rotated (partitioning along the Y-axis followed by partitioning along the X-axis) will be better only if $k > d$.

Lemma 4 *If the partitions are equal, then Acost is given by:*

$$\min_{1 \le l \le p} \{[(\lfloor p/l \rfloor (\lfloor p/l \rfloor - 1))(l - p \bmod l) + (\lceil p/l \rceil (\lceil p/l \rceil - 1))(p \bmod l)]na_i + lm\}.$$

5 Latency considerations

In the previous section latency was ignored in the communication cost. Latency in current implementations of message-passing software is two to three orders of magnitude larger than the transmission time required for sending a byte of information across the network. This can be significantly reduced by decreasing the number of software layers which the message-passing software is built upon. Several such efforts are currently underway. In this section we discuss the effect of latency on some of the results derived in the previous section.

On a broadcast network like Ethernet every message is received on all the nodes. The lower layers ignore the messages not destined for the node. However, future message-passing software should support a multicast. In such cases the software overhead of sending multiple messages over a broadcast network could be reduced by multicasting one message with the combined data for all the processors together. Each receiver would use the part for which it was the destination, which would reduce the overhead cost without increasing the actual cost of transmission. For the above scenario the latency overhead for each processor would be fixed independently of the number of processors a given processor needs to send a message. Hence, the results discussed in Section 4 would still hold.

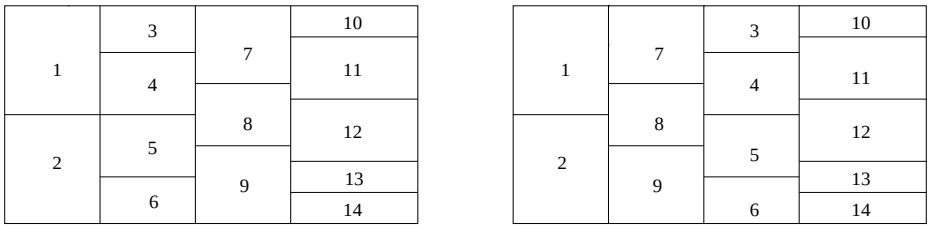

Figure 12: Two different decompositions that are "rearrangements" of each other. The one on the left has a larger number of internal edges.

When point-to-point communication is used there is a latency cost associated whenever two partitions share a common boundary. Thus, the additional cost is equal to the number of internal edges for a given decomposition multiplied by the latency cost. The number of internal edges can be evaluated by using the well-known Euler Theorem for planar graphs,

$$V - E + F = 2,$$

where E is the number of edges, V is the number of vertices and F is the number of regions. Assuming p partitions, $F = p + 1$. Hence,

$$E = V + p + 1 - 2 \Leftrightarrow E = V + p - 1. \tag{1}$$

We define any vertex to be a boundary vertex if it lies on the outside boundary of the array to be decomposed. Otherwise, the vertex is defined to be an internal vertex. Similarly, any edge

is defined to be a boundary edge if both endpoints of that edge are boundary vertices, else it is defined as an internal edge. Let V_b, V_i, E_b, E_i be the number of boundary vertices, internal vertices, boundary edge, and internal edges, respectively.

Since $E_b = V_b$, we can derive the following by using (1):

$$E_i = V_i + p - 1.$$

Thus the number of internal edges for a given decomposition can be derived by finding the unique corner points of partitions of a given decomposition, which can be done either by hashing or sorting. In the latter case the time requirements are $O(p \log p)$ for each decomposition.

Consider the equivalence class derived by the "rearrangement" operation described in Section 4. The number of internal edges for two different decompositions from the same equivalence class need not be equal. Consider the decompositions given in Figure 12 (a) and Figure 12 (b). Although they belong to the same equivalence class, the number of internal edges of the decomposition in Figure 12 (a) is less than in Figure 12 (b). A decomposition has the *sorted order property* if the vertical subarrays are sorted according to their *count* values (number of partitions) and the partitions within and between the vertical subarrays are sorted according to their size from top to bottom and left to right, respectively. A possible improvement is to consider all the rearrangements of a decomposition satisfying the sorted order property by allowing interchanges between any two vertical subarrays and/or interchanges of partitions within a vertical subarray and then choosing the decomposition with the minimum number of edges. Unfortunately, the number of such decompositions, though less than $p!2^p$, would still be quite large.

One would expect the difference between the number of internal edges to be small within a given equivalence class. Making this assumption, an algorithm that considers only $pt(p)$ decompositions as discussed in Section 4, should be able to find a close to optimal solution for all XY decompositions when the cost of decomposition includes the effect of latency corresponding to every internal edge.

6 Algorithms

In this section we describe four algorithms for decomposing an array into nonuniform-sized partitions. These are based on the ideas discussed in Sections 3, 4, and 5.

6.1 XY algorithm

A high-level algorithm that generates all the XY partitions is given in Figure 13. One can improve the quality of partitioning produced by the algorithm by using **Lemma 3**. For each decomposition considered, we compute the constant associated with each dimension separately (i.e., $Cost = d * m + k * n$). If $d < k$, then rotating the decomposition will produce a cheaper communication cost. We will call the new algorithm $XY2$.

6.2 TILE algorithm

In Section 3 we observed that partitioning along the shorter dimension will reduce the communication cost. We can use this fact to improve the algorithm in the previous section. This new

Initial condition:
1. The dimensions of the array are such that $m \leq n$.
2. $WeightList$ is the list of the weights of remaining partitions in nonincreasing order.
3. r is the current size of $WeightList$.
4. Sl (Sr) is equal to the left (right) subarray after partitioning the current array along dimension M.
5. min is the least number of partitions allowed in a vertical subarray. It is initialized to 1.

procedure $XY(WeightList, r, m, n, min)$
  while $(min \leq r)$
    for $(1 \leq i \leq min)$ $WeightListLeft[i] := WeightList[i]$.
    if $(min \neq r)$
      for $(1 \leq i \leq r - min)$ $WeightListRight[i] := WeightList[min + i]$.
    if $(min \leq r - min)$
      Divide current array $[m \times n]$ into two parts Sl $[m_{Sl} \times n_{Sl}]$ and Sr $[m_{Sr} \times n_{Sr}]$ such that:
        $m_{Sl} := m.\ n_{Sl} := n * (\sum WeightListLeft / \sum WeightList)$.
        $m_{Sr} := m.\ n_{Sr} := m - n_{Sl}$.
      if $(min > 1)$
        divide Sl into min partitions along n.
      if $(r - min > 1)$
        $XY(WeightListRight, r\text{-}min, m_{Sr}, n_{Sr}, min)$
    else
      Divide current array $[m \times n]$ into r parts along n.
      Exit while loop.
    $min := min + 1$.
  endwhile.
end.

Figure 13: XY Algorithm.

Initial conditions:
1. The dimensions of the array are such that $m \leq n$.
2. $WeightList$ is the list of the weights of remaining partitions in nonincreasing order.
3. r is the size of $WeightList$.
4. Sl (Sr) is equal to the left (right) subarray after partitioning the current array along dimension M.
5. min is the least number of partitions allowed in a vertical subarray. It is initialized to 1.

procedure TILE$(WeightList, r, m, n, min)$
  while $(min \leq r)$
    for $(1 \leq i \leq min)$ $WeightListLeft[i] := WeightList[i]$.
    if $(min \neq r)$
      for $(1 \leq i \leq r - min)$ $WeightListRight[i] := WeightList[min + i]$.
    if $((min \leq r - min) or (m_{Sr} > n_{Sr}))$
      Divide current array $[m \times n]$ into two parts Sl $[m_{Sl} \times n_{Sl}]$ and Sr $[m_{Sr} \times n_{Sr}]$ such that:
        $m_{Sl} := m.\ n_{Sl} := n * (\sum WeightListLeft / \sum WeightList)$.
        $m_{Sr} := m.\ n_{Sr} := n - n_{Sl}$.
      if $(min > 1)$
        divide Sl into min partitions along N.
      if $((r - min > 1) and (m_{Sr} > n_{Sr}))$
        TILE$(WeightListRight, r\text{-}min, n_{Sr}, m_{Sr}, 1)$
      else if $((r - min > 1) and (min \leq r - min))$
        TILE$(WeightListRight, r\text{-}min, m_{Sr}, n_{Sr}, min)$
    $min := min + 1$.
  endwhile
end.

Figure 14: TILE Algorithm.

```
Initial conditions:
1. The dimensions of the array are such that m ≤ n .
2. WeightList is the list of the computational ratio of remaining partitions in nonincreasing order.
3. r is the current size of WeightList.
4. d refers to the two dimensions of the array. It is initialized to vertical.
procedure RecursiveBisection(WeightList, r, m, n, d)
    x := ⌈r/2⌉.
    for (1 ≤ i ≤ x) WeightListLeft[i] := WeightList[i].
    for (1 ≤ i ≤ r − x) WeightListRight[i] := WeightList[i + x].
    if (d = vertical)
        Let y := (Σ_{i=1}^{x} WeightListLeft_i)/(Σ_{i=1}^{y} WeightList_i).
        Divide the current array into two subarrays along d according to y.
        j := y * n.
        c := n − j.
        d := horizontal.
        if (x > 1)
            RecursiveBisection(WeightListLeft, x, m, j, d).
        if ((r − x) > 1)
            RecursiveBisection(WeightListRight, r − x, m, c, d).
    else
        Let y := (Σ_{i=1}^{x} WeightListLeft_i)/(Σ_{i=1}^{y} WeightList_i).
        Divide the current array into two subarrays along d according to y.
        j := y * m.
        c := m − j.
        d := vertical.
        if (x > 1)
            RecursiveBisection(WeightListLeft, x, j, n, d).
        if ((r − x) > 1)
            RecursiveBisection(WeightListRight, r − x, c, n, d).
end.
```

Figure 15: Recursive Bisection Algorithm.

algorithm (described in Figure 14) is similar to XY, with one modification: whenever the remaining length of the current dimension along which the vertical subarrays are partitioned becomes less than the other dimension, the order of the partitioning is switched, i.e, the XY algorithm is applied with the other dimension as length and the current dimension as width. The lower bound on the computational requirements of this new algorithm is $\Omega(pt(p))$. This happens when partitioning of vertical subarrays continues along one dimension and the algorithms behaves as an XY algorithm. The worst-case complexity of the algorithm is $O(2^{p-1})$.

6.3 Recursive Bisection algorithm

A simple algorithm was proposed by Crandall and Quinn [10] for decomposing a two-dimensional array for a NUCE (Figure 15). We shall refer to it as RecursiveBisection. RecursiveBisection partitions the current array according to the first half of the partitions available. Two simple variations are:

1. RecursiveBisection switches the dimension along which partitioning is performed independently of the difference between the length and the width of the array. If the aspect ratio of an array is large, the communication cost can be reduced by partitioning along the dimension with the larger size. We shall refer to this variation as "RecursiveBisection2." This algorithm,

also partitions the current array according to the y partitions available such that the sum of their weights is approximately half of the total.

2. The way RecursiveBisection2 partitions the array may result in unbalanced total weights being assigned to the two partitions. For example, consider the case of four partitions with sizes 0.4, 0.4, 0.1, and 0.1, respectively. In this case the array would be partitioned into two parts with one partition having 80% of the size. We wanted to study the effect of choosing balanced partitions by using a binpacking algorithm to divide the list into two parts with nearly equal total weight. We call this variation "RecursiveBisection3."

6.4 Simulation results

In this section we present the performance of the algorithms described in the previous sections. The quality of partitioning produced by the different algorithms was measured for the following parameters:

1. *Number of partitions:* We performed our simulation for 4, 5, 7, 10, 15, and 20 partitions. We decided upon a limit of 20 partitions because we believe that the use of such environments for data-parallel computing would typically be limited to this number.

2. *Ratio of Computational power of Maximum/Minimum:* We varied this ratio from 1 to 8 to study the effect of nonuniformity between the computational units.

3. *Size of the arrays:* We considered arrays of different sizes and shapes ranging from 1K $\times 1K$ to 1K $\times$ 20K to study effects based on the aspect ratio of the array to be partitioned.

4. *Latency Costs:* We considered three different latency costs: $0, 100, 1000$. The first corresponds to the case when latency effects are not relevant (e.g., in the case of a multicast primitive as described in Section 5). The latter two cases represent practical scenarios.

Based on our preliminary experiments, we concluded that RecursiveBisection had the worst average communication cost, $XY2$ produced the same as or marginally better results than XY, and RecursiveBisection3 had similar performance as RecursiveBisection2. Therefore, we will not present the performance of RecursiveBisection, RecursiveBisection3 and XY.

We decided to present comparative rather than absolute results due to the large amount of data generated by our simulation. All comparisons were made with RecursiveBisection2 because it produces very good results at a relatively low cost. A comparative study is more useful for determining the utility of a more expensive algorithm. The following summarizes the different results presented.

1. Tables 2 through 4 give the percentage improvement of the Tile algorithm over the RecursiveBisection2 for different values of the parameters.

2. Tables 5 through 7 give the percentage improvement of the $XY2$ algorithm over the RecursiveBisection2 for different values of the parameters.

Each entry shows the percentage improvement for the average communication cost (using the *Acost* measure) over 20 randomly generated samples. Table 8 presents the average time spent by

the decomposition algorithms for different numbers of partitions. The case when latency is nonzero requires more time for each of the algorithms, because the number of internal edges needs to be calculated for each decomposition.

Max/Min Ratio	Array Size = $1K \times 1K$. Processors						Array Size = $1K \times 2K$. Processors						Array Size = $1K \times 3K$. Processors					
	4	5	7	10	15	20	4	5	7	10	15	20	4	5	7	10	15	20
1	0	0	0	4	1	3	0	0	0	0	4	4	0	0	0	5	0	3
2	0	4	0	5	2	3	0	3	1	1	3	5	0	2	2	5	2	4
3	0	2	2	6	4	2	0	5	3	1	3	5	0	0	1	5	4	3
4	0	1	3	6	4	3	0	6	3	2	3	4	0	3	0	3	4	3
8	0	0	2	5	4	3	1	5	3	2	4	4	1	4	1	3	4	3
	Array Size = $1K \times 5K$. Processors						Array Size = $1K \times 10K$. Processors						Array Size = $1K \times 20K$. Processors					
	4	5	7	10	15	20	4	5	7	10	15	20	4	5	7	10	15	20
1	0	0	0	0	0	5	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	1	3	4	0	0	0	0	0	2	0	0	0	0	0	0
3	0	0	0	3	3	2	0	0	0	0	0	2	0	0	0	0	0	0
4	0	0	0	4	3	3	0	0	0	0	1	1	0	0	0	0	0	0
8	0	0	1	3	2	3	0	0	0	0	2	1	0	0	0	0	0	1

Table 2: Percentage improvement of Tile over RecursiveBisection2 (Latency=0).

Max/Min Ratio	Array Size = $1K \times 1K$. Processors						Array Size = $1K \times 2K$. Processors						Array Size = $1K \times 3K$. Processors					
	4	5	7	10	15	20	4	5	7	10	15	20	4	5	7	10	15	20
1	0	0	-2	0	-2	5	0	0	0	8	2	2	0	0	0	8	-1	4
2	0	1	0	2	0	2	0	1	0	3	2	4	0	0	2	3	2	2
3	0	1	1	4	2	2	0	3	0	1	2	4	0	2	1	3	3	2
4	0	-3	1	4	2	2	0	4	0	0	2	3	0	4	0	4	3	2
8	0	-3	1	3	2	2	3	3	1	2	2	2	1	3	1	2	2	3
	Array Size = $1K \times 5K$. Processors						Array Size = $1K \times 10K$. Processors						Array Size = $1K \times 20K$. Processors					
	4	5	7	10	15	20	4	5	7	10	15	20	4	5	7	10	15	20
1	0	0	0	0	0	9	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	1	0	4	0	0	0	0	0	2	0	0	0	0	0	0
3	0	0	0	3	2	2	0	0	0	0	0	2	0	0	0	0	0	0
4	0	0	0	3	2	2	0	0	0	0	1	1	0	0	0	0	0	0
8	0	0	0	2	2	2	0	0	0	0	1	1	0	0	0	0	0	1

Table 3: Percentage improvement of Tile over RecursiveBisection2 (Latency=100).

Max/Min Ratio	Array Size = $1K \times 1K$. Processors						Array Size = $1K \times 2K$. Processors						Array Size = $1K \times 3K$. Processors					
	4	5	7	10	15	20	4	5	7	10	15	20	4	5	7	10	15	20
1	10	16	10	8	15	20	0	17	8	26	6	12	0	0	12	11	11	22
2	10	12	13	11	13	16	0	8	10	14	15	17	0	0	7	16	17	17
3	10	13	13	14	18	17	0	6	7	10	14	16	0	2	5	17	16	17
4	13	11	11	14	17	15	0	20	6	10	15	15	0	3	6	15	15	16
8	11	8	12	14	16	18	0	2	5	11	19	15	2	-2	5	10	15	19
	Array Size = $1K \times 5K$. Processors						Array Size = $1K \times 10K$. Processors						Array Size = $1K \times 20K$. Processors					
	4	5	7	10	15	20	4	5	7	10	15	20	4	5	7	10	15	20
1	0	0	0	0	12	26	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	10	21	22	0	0	0	0	2	13	0	0	0	0	0	0
3	0	0	0	10	19	22	0	0	0	0	7	10	0	0	0	0	0	0
4	0	0	0	6	18	21	0	0	0	-5	4	11	0	0	0	0	0	0
8	0	0	1	7	16	19	0	0	1	-1	5	10	0	0	0	0	0	2

Table 4: Percentage improvement of Tile over RecursiveBisection2 (Latency=1000).

Max/Min	Array Size = $1K \times 1K$.						Array Size = $1K \times 2K$.						Array Size = $1K \times 3K$.					
	Processors						Processors						Processors					
	4	5	7	10	15	20	4	5	7	10	15	20	4	5	7	10	15	20
1	0	0	0	4	1	3	0	0	2	3	5	5	0	0	0	5	1	3
2	0	4	2	6	2	3	0	3	4	3	3	5	0	2	2	5	2	4
3	0	2	3	7	4	2	0	5	5	1	4	6	0	2	1	5	5	3
4	0	1	3	7	4	2	0	6	5	1	4	4	0	4	0	4	5	3
8	0	1	2	5	4	3	0	5	5	2	4	3	1	4	1	4	4	3
	Array Size = $1K \times 5K$.						Array Size = $1K \times 10K$.						Array Size = $1K \times 20K$.					
	Processors						Processors						Processors					
	4	5	7	10	15	20	4	5	7	10	15	20	4	5	7	10	15	20
1	0	0	0	0	1	5	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	1	4	4	0	0	0	0	0	2	0	0	0	0	0	0
3	0	0	0	3	3	3	0	0	0	0	1	2	0	0	0	0	0	0
4	0	0	0	4	3	3	0	0	0	0	1	1	0	0	0	0	0	0
8	1	0	0	3	2	4	0	0	0	1	2	1	0	0	0	0	0	1

Table 5: Percentage improvement of $XY2$ over RecursiveBisection2 (Latency=0).

Max/Min	Array Size = $1K \times 1K$.						Array Size = $1K \times 2K$.						Array Size = $1K \times 3K$.					
	Processors						Processors						Processors					
	4	5	7	10	15	20	4	5	7	10	15	20	4	5	7	10	15	20
1	0	0	0	2	4	10	0	0	1	9	5	4	0	0	0	8	0	6
2	0	3	1	3	1	2	0	2	2	4	2	4	0	1	2	3	3	3
3	0	2	2	5	3	2	0	3	3	1	3	5	0	2	1	4	4	2
4	0	0	3	5	3	2	0	5	3	1	3	4	0	4	0	4	4	2
8	0	0	2	3	3	2	1	4	2	2	4	3	1	3	1	3	3	4
	Array Size = $1K \times 5K$.						Array Size = $1K \times 10K$.						Array Size = $1K \times 20K$.					
	Processors						Processors						Processors					
	4	5	7	10	15	20	4	5	7	10	15	20	4	5	7	10	15	20
1	0	0	0	0	0	9	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	2	1	4	0	0	0	0	0	2	0	0	0	0	0	0
3	0	0	0	3	2	3	0	0	0	0	0	2	0	0	0	0	0	0
4	0	0	0	3	2	3	0	0	0	0	0	1	0	0	0	0	0	0
8	0	0	0	3	2	3	0	0	0	0	2	1	0	0	0	0	0	1

Table 6: Percentage improvement of $XY2$ over RecursiveBisection2 (Latency=100).

Max/Min	Array Size = $1K \times 1K$.						Array Size = $1K \times 2K$.						Array Size = $1K \times 3K$.					
	Processors						Processors						Processors					
	4	5	7	10	15	20	4	5	7	10	15	20	4	5	7	10	15	20
1	0	17	11	8	15	20	0	17	18	26	14	12	0	0	18	26	11	22
2	14	17	16	17	21	23	12	17	21	25	25	27	0	5	21	24	25	25
3	14	16	17	30	22	23	12	17	19	22	25	27	4	11	17	25	25	25
4	14	16	17	30	29	23	11	17	18	23	26	26	4	11	16	26	25	25
8	14	15	17	19	21	23	11	16	18	23	25	26	11	10	16	24	24	27
	Array Size = $1K \times 5K$.						Array Size = $1K \times 10K$.						Array Size = $1K \times 20K$.					
	Processors						Processors						Processors					
	4	5	7	10	15	20	4	5	7	10	15	20	4	5	7	10	15	20
1	0	0	0	0	14	26	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	14	26	27	0	0	0	0	3	15	0	0	0	0	0	0
3	0	0	6	15	24	28	0	0	0	0	5	14	0	0	0	0	0	0
4	0	0	6	13	24	28	0	0	0	1	7	15	0	0	0	0	0	2
8	0	2	7	14	23	26	0	0	0	3	10	15	0	0	0	0	2	4

Table 7: Percentage improvement of $XY2$ over RecursiveBisection2 (Latency=1000).

Processors	Tile		$XY2$		RecursiveBisection2		RecursiveBisection3	
	NonZero Latency	Zero Latency	NonZero Latency	Zero Latency	NonZero Latency	Zero Latency	NonZero Latency	Zero Latency
4	0.00052	0.00014	0.00033	0.00011	0.00008	0.00003	0.00008	0.00004
5	0.00125	0.00030	0.00058	0.00018	0.00019	0.00004	0.00010	0.00005
7	0.00718	0.00112	0.00173	0.00047	0.00015	0.00005	0.00016	0.00008
10	0.07509	0.00664	0.00710	0.00192	0.00022	0.00008	0.00025	0.00012
15	3.83530	0.12115	0.05077	0.00906	0.00039	0.00012	0.00040	0.00020
20	11.81843	1.27512	0.23845	0.03935	0.00049	0.00017	0.00058	0.00028

Table 8: The average execution time of different algorithms (in seconds).

Zero latency

This section compares the different algorithms based on the total amount of communication generated. The following observations can be made from Tables 2 and 5:

1. The performances of Tile and $XY2$ are better than that of RecursiveBisection2. The performance improves with number of partitions when the aspect ratio is less than the number of partitions.

2. The performances of Tile and $XY2$ are comparable. However, the computational cost of $XY2$ is considerably less.

3. When the aspect ratio of the array is larger than the number of partitions, the partitioning is generally along one dimension and hence all algorithms perform equally well.

We conclude that $XY2$ performs better than the other schemes when the number of partitions is large and the aspect ratio is smaller than the number of partitions. In all other cases, RecursiveBisection2 produces reasonably good-quality solutions at a very small cost.

The decompositions produced by different algorithms for partitioning an array of size 1000×3000 into 7 partitions ($a_1 = 0.5$, $a_2 = 0.1$, $a_3 = 0.1$, $a_4 = 0.1$, $a_5 = 0.1$, $a_6 = 0.05$ and $a_7 = 0.05$) are given in Figure 16. The communication costs are 5750, 4500, 4500, 4666, and 4700 by using RecursiveBisection, $XY2$, TILE, RecursiveBisection2, and RecursiveBisection3, respectively.

Nonzero latency

This section compares the different algorithms when latency costs are important. From Tables 3, 4, 6 and 7, we observe the following:

1. The performances of Tile and $XY2$ are better than that of RecursiveBisection2. The performance improves with the number of partitions when the aspect ratio is less than the number of partitions.

2. The performance of $XY2$ is better than that of Tile as the latency increases. This suggests that the number of internal edges produced by Tile is larger due to the change in direction when partitioning is performed. The relative performance improvements of $XY2$ are better when the latency is higher, the number of partitions is larger, and the aspect ratio is not larger than the number of partitions.

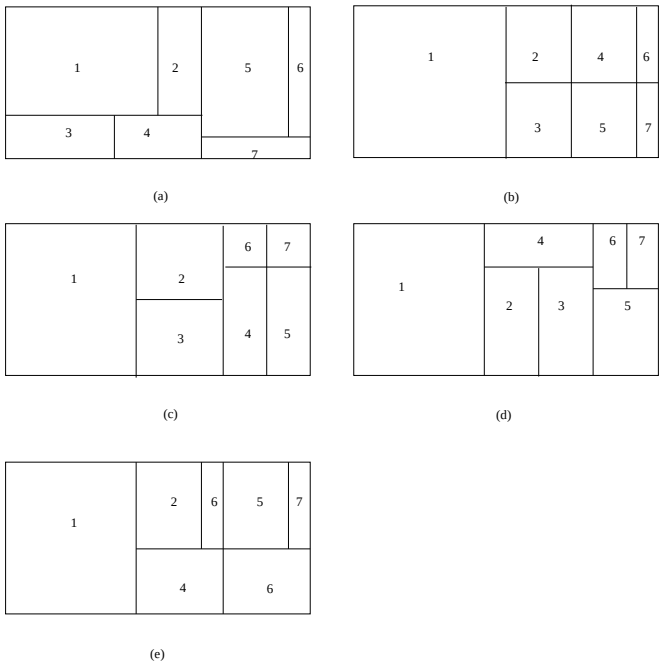

Figure 16: The decompositions produced by different algorithms for partitioning an array of size 1000×3000 into 7 partitions ($a_1 = 0.5$, $a_2 = 0.1$, $a_3 = 0.1$, $a_4 = 0.1$, $a_5 = 0.1$, $a_6 = 0.05$ and $a_7 = 0.05$); (a) RecursiveBisection, (b) $XY2$, (c) TILE, (d) RecursiveBisection2, (e) RecursiveBisection3.

3. When the aspect ratio of the array is larger than the number of partitions, the partitioning is generally along one dimension and hence all algorithms have a similar performance.

Based on the above observations, the $XY2$ is the algorithm of choice when latency cost is an important factor. It produces the best results at a reasonable cost.

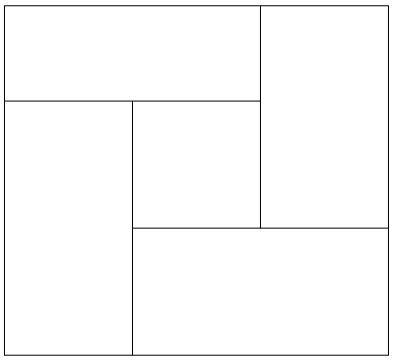

Figure 17: Decomposition not considered by any of our algorithms.

7 Conclusions

In this paper we have presented several decomposition algorithms for decomposing two-dimensional data array for a heterogeneous processors network. One important question that is relevant here is

whether there are any decompositions that none of the algorithms described in this paper consider. One such decomposition is given in Figure 17. When all the partitions are of equal size it is easy to show that the communication generated by decomposition in Figure 17 (using both the communication cost and the latency overhead) is much larger then the decomposition produced by $XY2$ partitioning. For different-sized partitions, it is difficult to evaluate the communication generated by such decompositions. Hence it is difficult to comment on the optimality of the decompositions generated by our algorithms.

Our results indicate that XY-based partitioning produces the best cost decomposition at a relatively small cost. The number of decompositions considered by XY partitioning is $pt(p)$, where p is the number of partitions. The growth of this function is reasonable (Table 1) for practical values of p. Further, the time required for partitioning is not prohibitive for most applications (Table 8). This makes the algorithm very important for practical considerations.

In an adaptive system the resources may change over a period of time and may require remapping of data, thus these algorithms may have to be executed at runtime. If computational cost is at a premium the simple recursive bisection algorithm, which always partitions along the smaller dimension, works very well. One can also use a hybrid algorithm that uses RecursiveBisection for the first few steps, followed by an XY partitioning.

References

[1] High-Performance Fortran Forum. "High-Performance Fortran Language Specification," January 1993.

[2] Mikhail Atallah, Christina Lock, Dan Marinescu, Howard Siegel, and Thomas Casavant. "Models and algorithms for co-scheduling computer-intensive tasks on a network of workstations," *Journal of Parallel and Distributed Computing*, pp. 319-327, December 1992.

[3] Sara Baase. *Computer Algorithms*, Addison-Wesley, 1988.

[4] Adam Beguelin, Jack Dongarra, Al Geist, Robert Manchek, and Vaidy Sunderam. "A user's guide to PVM," Technical Report ORNL/TM-11826, Oak Ridge National Laboratory, 1991.

[5] Krishna P. Belkhale and Prithviraj Banerjee. "Recursive partitions on multiprocessors." *Proceedings of the Fifth Distributed Memory Computing Conference*, pp. 930–938, April 1990.

[6] Marsha J. Berger and Shahid H. Bokhari. "A partitioning strategy for nonuniform problems on multiprocessors," *IEEE Transactions on Computers*, Vol. C-36, No. 5, pp. 570–580, May 1987.

[7] G. Bernard, A. Duda, Y. Haddad, and G. Harrus. "Primitives for Distributed Computing in a Heterogeneous Local Area Environment," *IEEE Transactions on Software Engineering*, Vol. 15, No. 12, pp. 1567–1577, December 1989.

[8] Alex L. Cheung and Anthony P. Reeves. "High-performance computing on a cluster of workstations," *Proceedings of the First International Symposium on High-Performance Distributed Computing*, pp. 152–160, Sept 9–11 1992.

[9] Ronald L. Graham, Donald E. Knuth and Oren Patashnik. *Concrete Mathematics*, Addison-Wesley, 1988.

[10] Phyllis E. Crandall and Michael J. Quinn. "Block Data Decomposition for Data-Parallel Programming on a Heterogeneous Workstation Network," *Proceedings of the Second International Symposium on High-Performance Distributed Ccomputing*, pp. 42–49, July 1993.

[11] R. Freund. "Optimal Selection Theory for Superconcurrency," *Proceedings of Supercomputing'89*, pp. 699–703, November 1989.

[12] David M. Nicol. "Rectilinear partitioning of irregular data parallel computations," Technical report NASA CR-187601, ICASE, 1991.

[13] David M. Nicol and Joel H. Saltz. "Dynamic remapping of parallel computations with varying resource demands." *IEEE Transactions on Computers*, Vol. 37, No. 9, pp. 1073–1087, September 1988.

[14] M. Rosing and R. P. Weaver. "Mapping data to processors in distributed memory computations," *Proceedings of the Fifth Distributed Memory Computing Conference*, pp. 884–892, April 1990.

[15] Lawrence Snyder and David G. Socha. "An algorithm producing balanced partitionings of data arrays," *Proceedings of the Fifth Distributed Memory Computing Conference*, pp. 867–875, April 1990.

[16] U. Warrier and C. Sunshine. "A Platform for Heterogeneous Interconnection Network Management," *IEEE Transactions on Selected Areas in Communications*, Vol. 8, No. 1, pp. 119–126, January 1990.