Minimizing Startup and Transfer Costs During Dynamic Data Redistribution Between Parallel Processor Sets

Stavros I. Souravlas Manos Roumeliotis

Department of Applied Informatics, University of Macedonia
54006 Thessaloniki, Greece

Abstract. This paper describes an algorithm for reducing the index computational and data transfer costs when distributing messages from one processor to another. The transmission cost is reduced by using a pipeline-based schedule, where messages of different communication costs are forming transmission tasks from certain sending processors to certain receiving processors, in such a manner, that each receiver gets only a message at a time. In this way, conflicts on the receiving ports are avoided, while the need for local memory buffers is minimized. Furthermore, the time needed for the computation of the local memory locations where each message will be stored is significantly reduced. The local indices computations take only O($max(P,Q)$) time.

1 Introduction and Related Work

Many complicated parallel computing applications are composed of several stages. Due to the fact that the distribution of data on the various processors may be improper to complete a certain process, data must be redistributed during runtime or dynamically, to the processors. Since the redistribution must be performed during runtime, it's total cost must be reduced to the minimum. Most of the known techniques so far implement the dynamic redistribution of data using round robin techniques. The main reason is that the round robin method guarantees that the processors will be equally charged during runtime and therefore they will be prevented from staying idle. Most of the dynamic data transfer algorithms found in literature are based on the block cyclic redistribution theory initially developed by HPF [1,2,5,6,8,9,13] (High Performance Fortran).

When arranging a dynamic data transmission, three separate costs are involved: *the index computational cost*, that is, the cost of computing the local memory locations where messages will be stored, *the message preparation cost*, that includes the minimal amount of time at which processors perform internal memory read operations to gather the proper data and form the messages to be sent and *the total transmission cost* which is the time it takes for the messages to be transferred to their destination. In this paper, we consider the message preparation cost to be trivial and we apply a technique for minimizing the computational cost and the total transmission cost.

A lot of effort has been focused on the problem of runtime redistribution between several processors. Some interesting schemes that were provided are mentioned below:

Thakur et al. [14] provided algorithms for runtime redistribution. Their work is divided into two cases: the general case of *Cyclic(x)* to *Cyclic(y)* redistribution, where there is no relation between x and y and a special case where x is a multiple of y or y is a multiple of x. For the special case, they developed the KY-TO-Y algorithm. Each processor p calculates the destination processor p_d of it's first element as p_d= *mod(kp, P)*. Then, the first y elements are sent to p_d, the next y to $p_d + 1$ and so on, until the end of the first block. Then, the other blocks are moved in the same pattern. For the general case, they implemented the GCD (greatest common divisor) and the LCM (least common multiplier) algorithms. The main idea was to redistribute from *cyclic(x)* to *cyclic(m)*, where m was the LCM or GCD of x, y using the KY-TO-Y algorithm which solved the special case.

The problem of *message scheduling*, that is, to organize Communication between processor pairs in such a manner that startup costs are reduced, was firstly considered by Walker and Otto [15] who focused on the problem of data redistribution from *cyclic(x)* on P processors to *cyclic(Kx)* on P processors. They provided *synchronized* and *unsynchronized* schemes that were free of conflicts. In the *synchronized* scheme however, performance was reduced by the fact that some processes had to wait for others before receiving data, while the main problem of the *unsynchronized* algorithm was the necessity for buffering space equal to the data redistributed. Furthermore, the number of steps required for the implementation of those schemes was not minimal.

The problem of data redistribution was successfully dealt with, in a very interesting paper by Desprez et al.[3]. Their effort was focused on solving the general redistribution problem, moving messages from a P-processor grid, to a Q-processor grid for any given message size. The main idea behind their algorithm was to create homogeneous communication patterns which they called *classes*. Processor pairs in a certain class exchanged messages of the same size. Having created the classes, they arranged the message scheduling by mixing elements that belonged to different classes in such a manner that the elements of the most expensive classes are distributed in the fewer steps possible.

In this paper, we will try solve the problem of moving data on a PxQ processor grid, focusing on minimizing index computational and transmission costs but we also consider the problem of avoiding any congestions on destination processor ports. The rest of the paper is organized as follows: In Section 2 we present the preliminaries for block-cyclic redistribution necessary for the better understanding of the following sections. In section 3, we present the algorithm in detail, focusing our attention on reducing the index computation and transmission costs.

2 Redistribution Preliminaries

Consider the data that are going to be redistributed as a two-dimensional table MxN. This table is going to be redistributed in a block-cyclic fashion from cyclic(r) on a P-processor grid, to cyclic(s) on a Q-processor grid, where r, s are the number of rows and columns each block contains, respectively. If each block contains r rows and s columns then, provided that M divides r and N divides s, the data array will be divided into an $M_b X N_b$ parts where: $M_b = \frac{M}{r}$,$N_b = \frac{N}{s}$ If M does not divide r then $M_b = \frac{M}{r} + 1$. If N

does not divide s then $N_b = \frac{N}{s} + 1$. Fig.1 shows an array of 15x20 elements partitioned into blocks of size 4x3.

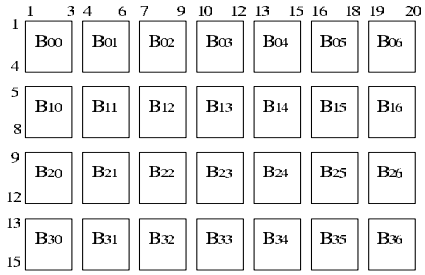

Fig. 1. Data table divided into blocks

Block cyclic redistribution is the mapping of a data array element with index i to a process index p, a block index l and a **local memory position index** x, where [3]:

$$l = \frac{\lfloor i/r \rfloor}{P}, \;\; p = \lfloor i/r \rfloor \; mod \, P, \;\; x = i \;\; mod \;\; r (1) \tag{1}$$

From (1), we derive:

$$i = (lP + p)r + x \tag{2}$$

Consider the redistribution from cyclic(r) on a P-processor grid, to cyclic(s) on a Q-processor grid. If the redistributed data array element is indexed j, the process index is q, the block index is m and the **local memory position index** y, then in a similar way, we derive:

$$j = (mQ + q)s + y \tag{3}$$

If we consider the transfer of a block indexed i during a redistribution stage from a known number of processors P to a known number of processors Q, then according to (2) and (3), this transfer can be described by the following equation:

$$(lP + p)r + x = (mQ + q)s + y \tag{4}$$

We consider as the total redistribution cost of a processor pair all the information about the block that each data element will move to, and the set of all possible local memory positions to which it may be located. Therefore the communication cost is represented by the number of solutions for the redistribution equation (4), given the value of the process coordinates (p, q) and the total number of senders P and receivers Q, and with unknowns l, m, y, x [3] which define the coordinates of the block number and the local memory position. The redistribution equation, can be solved using Euclid's theorem for solving linear Diophantine equations. It must be mentioned, that if for a given pair of sender-receiver (p, q), there is no quadruple (l, m, y, x) to satisfy (4), then there is no communication between p and q.

3 Reducing the Startup and Transmission Cost

This section describes a method for organizing the dynamic data transmission in such a manner that the computational cost of local memory indices and the transmission cost are reduced to the minimum. Let g be the greatest common divisor of Pr and Qs. If we organize the transmission in such a manner that g is divided by the data block size parameters r and s, we can create a communication grid composed of $\frac{PQ}{rs}$ sub-tables in total (see fig. 2).

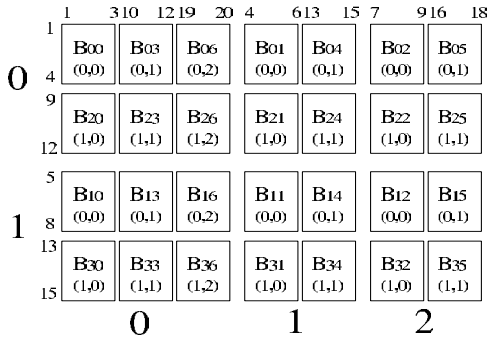

Fig. 2. A communication grid composed of $\frac{PQ}{rs}$ sub-tables

Each of the sub-tables is of size $r \times s$. Proposition 1 is the basis for the implementation of our method.

Proposition 1

Elements that lie in corresponding positions inside each of the $r \times s$ sub-tables represent a communication between processors p_i and q_j such that the quadruples (x, y, l, m) that satisfy (4) are exactly the same.

Using proposition 1, we can organize the communication in such a manner that messages of size $r \times s$, where r and s divide g, are transmitted from P to Q processors. Proposition 2 gives us the cost of such a schedule.

Proposition 2

The index computation cost of transmitting messages of size $r \times s$, where r and s divide g, is only $O(max(P, Q))$

Proof

Consider figure 3(a). To find the maximum number of iterations needed to compute the block and local memory indices, that is to find all quadruples (l, m, x, y) that satisfy

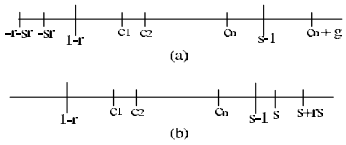

Fig. 3. Finding the solutions of the redistribution equation

(4) for a given pair of processors (p, q), we need to find the maximum length of the interval in which the algorithm will search for solutions.

Initially, we rewrite (4) as $lPr - mQs + (pr - qs) = y - x$. We have already set g as the greatest common divisor of Pr an Qs. Thus, there must be an integer λ such as: $lPr - mQs = \lambda g$. Also, we set $z = y - x$. According to [3], z lies in the interval $[1 - r...s - 1]$. The relationship $lPr - mQs + (pr - qs) = y - x$ will be rewritten as: $\lambda g + (pr - qs) = z$. This equation has a solution for a given processor pair (p, q) if we can find a multiple of g to add to the constant $(pr - qs)$ and get a value in the interval $[1 - r...s - 1]$. In the relationship $\lambda g + (pr - qs) = z$, if we set $P = P - 1$ and $Q = 0$, our starting point from which we may start adding multiples of g to the constant $pr - qs$ is $-r$ and the ending point beyond which the successive additions stop is $s - 1$. If we set $P = P - 1 - s$, our starting point will be $-r - sr$. The ending point remains $s - 1$. Obviously, the lowest point from which we may start searching for solutions (or adding multiples of g to $pr - qs$) is $-r - Pr$, while the ending point is always $s - 1$. In this case our interval is of length $s - 1 - (-r - Pr) = s + r - 1 + Pr$. We set $V_1 = s + r - 1 + Pr$. After each iteration the interval is reduced by $V_1 - g$, thus, after k iterations it's length would be $V_1 - kg$. The iterations finish when $V_1 - kg \leq 0$ or $kg \leq V_1$. The maximum number of iterations is the minimum k for which $kg \leq s + r - 1 + Pr$. Once the message size variables r and s are defined, the solution of this inequality depend on the value of P. The larger the value of P, the higher the number of iterations the algorithm will perform.

Consider figure 3(b). Similarly as in the previous paragraph, the highest point from which we may start subtracting multiples of g is $s + Qs$. The lowest point beyond which subtractions stop is $1 - r$. In this case, our interval is of size $V_2 = s + Qs - (1 - r) = Qs + s + r - 1$. Again, the maximum number of iterations required would be the minimum k for which $kg \leq V_2$. This number depends on the value of Q. Therefore, the maximum number of loops our algorithm will perform is $O(max(V_1, V_2))$ or $O(max(P, Q))$. For comparison, the index computation cost of other known algorithms are listed:

- Bipartite graphs based scheme: $O(P + Q)^{3.5}$ [3]
- Circulant matrix formalism based scheme: $O(log(K))$, but it solves only an instance of the problem, that is, moving data blocks increased by a factor K on a P-processor grid [7]. Proposition 1 leads us straight to the following corollary.

Corollary 1

Processor pairs that lie in corresponding sub-table positions have the same cost of communication.

We will use corollary 1 to create a communication pattern that reduces the total transfer cost by allowing carefully selected processor pairs to communicate at a time. The basic idea behind our algorithm is to decompose the redistribution problem into a set of pipeline operations. Each pipeline includes a specified number of tasks which are responsible for the message exchanging between carefully selected processor pairs. The main identities of all the tasks in each pipeline operation are:

- All tasks are scheduled in such a way that all sending processors can initialize send requests to multiple destinations at time intervals which are infinitesimally small, thus reducing the time that processors remain idle.
- All tasks are scheduled in such a way that receiving processors get one message at a time, thus congestions are avoided.

The pipeline-based implementation is composed of three stages: a. The creation of the pipeline tasks stage, b. The message preparation stage, and c. the sending stage. These three stages are described below.

3.1 Stage 1: Creating the Pipeline Tasks

As mentioned above, the pipeline tasks must be arranged in such a way that receiving processors get one message at a time. To satisfy this requirement, we schedule each task in such a way that it includes a specified number of message transmissions where the destination processors differ but the cost is the same. Therefore, our first job is to sort out all processor pairs of the communication grid with respect to their communication cost. This is very easy since we know that processor pairs (p, q) which lie at corresponding positions of different sub-tables have the same cost of communication.

3.2 Stage 2: Message Preparation

The message preparation phase includes the minimal amount of time at which processors perform internal memory read operations to gather the proper data and form the messages to be sent. When all the messages of a specific task are read, the task passes to a pipeline segment. When all the memory read operations finish, the pipeline is full and the sending phase can start. This is the main difference of this strategy compared to other strategies in literature. *Each sender does not have to wait for the completion of a transfer to prepare a new message for transmission.* Instead, in each pipeline task, *every processor initializes subsequent, carefully selected message transfers at each clock cycle.* This strategy eliminates the time senders remain idle and thus reduces the total communication cost.

3.3 Stage 3: Sending the Messages

From the moment the pipeline is full, successive message transfer tasks are completed at successive clock pulses. This scheme prevents congestion on the destination ports, because each task cannot contain more than one message to a given destination.

3.4 The Total Redistribution Cost

It is obvious that once the pipeline is full, it requires exactly d clock cycles to complete all the tasks (message transmissions). Since a redistribution is completed by a number of pipeline operations the total cost of the transmission phase C_{TR} will equal the sum of the all the maximum communication costs that exist among the tasks of the created pipelines. This is described as:

$$C_{TR} = \sum_{k=1}^{n} maxcost[T_i] \tag{5}$$

where T_i is a pipeline task, $i\epsilon[1..d]$ and n is the number of the pipelines created.

To define the total transmission cost, one more cost needs to be added, the *startup cost*, C_S, which includes procedure call overheads, *memory indexing calculations* and error controls. If a is the startup cost for a pipeline task, then the startup cost of a pipeline operation C_S is: $C_S = da$. The total cost of data redistribution can be found if we add the message preparation cost C_{PR} and the startup cost C_s to Eq. 5:

$$C_{Total} = \sum_{k=1}^{n} maxcost[T_i] + C_{PR} + C_S \tag{6}$$

The total cost of redistribution as calculated in (6) is obviously reduced compared to other known strategies. This improvement is due to the fact that a pipeline operation can implement data transfers in a series of pipeline tasks and it's cost equals the cost of the longest task, name it C_L. During this C_L time, a pipeline operation can also satisfy a good number of sending requests that have lower costs in comparison to C_L.

4 Conclusions

In this paper, we use an algorithm to solve the problem of moving data from P to Q processors during runtime. The main idea behind the algorithm was to arrange the size of the data to be transferred in such a way that we created homogeneous data transmissions between several processor pairs, in terms of both local memory index computation and transfer cost. Then, we created pipeline transmission tasks by mixing elements from diagonal blocks of our communication grid. Each of the task was responsible for the transmission of messages of a specific transfer cost. The big advantage of our strategy was that the memory index computation cost was reduced to only $O(max(P, Q))$, which is significantly small compared to the index computation cost of other algorithms in literature. In addition, the pipeline-based transmission schedule guaranteed of reduced data transfer time.

References

1. B. Chapman, P.Mehrotra, H. Moritsch, H. Zima: Dynamic Data Distribution in Vienna Fortran. *Proceedings Supercomputing Conference*, (1993) 284-293

2. Y.-C. Chung, C.-H. Hsu, S.-W. Bai: A Basic-Cycle Calculation Technique for Efficient Dynamic Data Redistribution. *IEEE Transactions on Parallel and Distributed Systems.* 9(4):359-377 (1998)
3. F. Desprez, J. Dongarra, A. Petitet, C. Randriamaro, Y. Robert: Scheduling Block-Cyclic Array Redistribution. *IEEE Transactions on Parallel and Distributed Systems.* 9(2):192-205 (1998)
4. E.T Kalns and L.Ni: Processor Mapping Techniques toward Efficient Data Redistribution. *IEEE Transactions on Parallel and Distributed Systems*, 6(12):1234-1247 (1995)
5. S.D. Kaushik, C.H. Huang, Sadayappan, J. Ramanujam, P. Sadayappan: Multi-Phase Redistribution: a Communication-Efficient Approach to Array Redistribution. Tech Report OSU-CISRC-9/94-52, Ohio State Univ. (1994)
6. K. Kennedy, N. Nedeljkovic, A. Sethi: Efficient Address Generation for Block-Cyclic Distributions. *Proceedings ACM/IEEE Supercomputing Conference*, (1995)
7. Y.W. Lim, P.B. Bhat, V.K Prasanna: Efficient Algorithms for Block Cyclic Redistribution of Arrays. *Algorithmica*, 24:298-330 (1998)
8. E.M. Miller: Beginner's Guide to HPF. Joint Institute for Computational Science, http://www.-jics.cs.utk.edu/HPF/HPFguide.html (1998)
9. N. Park, V.K Prassana, C.S. Raghavendra: Efficient Algorithms for Block Cyclic Array Redistribution Between Processor Sets. *IEEE Transactions on Parallel and Distributed Systems*, 10(12):1217-1240 (1999)
10. A. Petitet: Algorithmic Redistribution Methods for Block Cyclic Decompositions. Ph.D Thesis, Univ. of Tennessee, Knoxville, (1996)
11. A. Petitet, J. Dongarra: Algorithmic Redistribution Methods for Block Cyclic Decompositions. *IEEE Transactions on Parallel and Distributed Systems*, 10(12):1201-1216 (1999)
12. R. Thakur, A. Choudhary, G. Fox: Runtime Array Redistribution in HPF Programs. SHPCC 94, Northeast Parallel Architectures Center, (1994)
13. R. Thakur, A. Choudhary, J. Ramanujam: Efficient Algorithms for Array Redistribution. *IEEE Transactions on Parallel and Distributed Systems*, 7(6):587-594 (1996)
14. D.W Walker and S.W Otto: Redistribution of Block-Cyclic Data Distributions Using MPI. *Concurrency: Practice and Experience*, 8(9):707-728 (1996)