Optimizing Data Scheduling on Processor-In-Memory Arrays[†]

Yi Tian Edwin H.-M. Sha Chantana Chantrapornchai
Peter M. Kogge
Dept. of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN 46556

Abstract

In the study of PetaFlop project, Processor-In-Memory array was proposed to be a target architecture in achieving 10^{15} floating point operations per second computing performance. However, one of the major obstacles to achieve the fast computing was interprocessor communications, which lengthen the total execution time of an application. A good data scheduling, consisting of finding initial data placement and data movement during the run-time, can give a significant reduction in the total communication cost and the execution time of the application. In this paper, we propose efficient algorithms for the data scheduling problem. Experimental results show the effectiveness of the proposed approaches. Compared with default data distribution methods such as row-wise or column-wise distributions, the average improvement for the tested benchmarks can be up to 30%.

1 Introduction

Super parallel architectures are studied in the PetaFlop project in order to achieve 10^{15} floating point operations per second. As one of the design point teams for the PetaFlop project, we propose the Processor-In-Memory (PIM) architecture to achieve the desired PetaFlop performance. The PIM technology is used to avoid the "traditional" Von Neuman problem so as to enhance the computing capability.PIM, the combination on one chip of both dense DRAM memory and significant amounts of logic, can place computing either right next to, or even inside of, the memory macros, where there are huge amounts of raw memory bandwidth. One of the biggest obstacles encountered during the project development is how to find the best way to place data and move data in the PIM array in order to minimize communications. Our experiments show that a straight-forward row-wise or column-wise data placement may significantly degrade performance compared with an optimal data placement. Hence, a good data scheduling which consists of good data placement initialization as well as good data movement that wisely change the data placement during the runtime, are required. This paper presents efficient data scheduling algorithms as well as experimental results which not only give efficient initial data placement but also data movement that minimizes the total interprocessor communications.

Several researchers have proposed methods to automatically determine the data partitioning for distributed memory systems [1–4]. However, these approaches assume a regular data access pattern in a do-loop. In particular, these methods usually assume the dependency distance between loop indices forms a linear combination or uniform data reference. Unlike these approaches, our methods assume neither the linearity nor the uniformity of the data reference pattern. Rather than considering data dependency pattern directly, we investigate the data reference string of an application and provide an optimal data scheduling.

The rest of this paper is organized as follows: Section 2 introduces the problem and terminologies. In Section 3, we present the algorithms for finding good initial data placement and subsequence data movement after a program begins to execute. Three data scheduling techniques are proposed. An algorithm for grouping execution steps is introduced in Section 4. Section 5 includes the experimental results for several well-known benchmarks and compares the performance obtained by several data scheduling schemes. Finally, Section 6 draws the conclusions of the paper.

2 Problem Definitions and Terminology

In the PIM array, two stages are prepared before the execution of the program: the iteration partition and the data scheduling. The iteration partition maps the iteration space to the PIM processor array, while the data scheduling maps the data to the PIM processor array. After an application is partitioned, its data are distributed to each processor before the execution begins. For the clarity of presentation, we assume that the processor array forms a 2-dimensional grid, where each processor has its own local memory. The "x-y routing" method is used to communicate between processors. The *communication cost between any two processors* is defined as the distance between two processors along the x-axis and y-axis in the 2-D grid, weighted by the data volume transferred. Thus, *the total communication cost of a processor* is the sum of the communication cost for *all* of its required data. The *total communication cost for an application* is, therefore, the summation of the total communication cost of every processor in the processor array. For simplicity, we assume the distance between

[†]This work was partially supported by NSF MIP 95-01006 and NSF ACS 96-12028.

two adjacent processors is one, and each data transfer takes one time unit. Also, one copy of data is allowed in a system.

The execution of an application can be divided into multiple *steps*. A sequence of parallel execution steps are grouped into an *execution window*.

Definition 1 *Processor reference string with respect to a data in one execution window is the sequence of processors requiring this data in this execution window.*

Definition 2 *Data reference string of a processor in one execution window is the sequence of data referred by the processor in this execution window.*

Definition 3 *The center with respect to data D of an execution window is the processor that stores data D.*

Definition 4 *The local optimal center with respect to data D of an execution window is the center of the execution window with the minimum total communication cost with respect to data D.*

In [6], we showed that a good initial data placement without data movement can reduce the total communication cost compared with a straight forward distribution of the data. To achieve these results, a datum should be placed in the processor which is located approximately at the local optimal center.

In this paper, we present algorithms for finding data movement during the run-time to further reduce the total communication time of an application. Instead of finding a center for *all* execution windows, we determine an individual center for *one* or *more* execution window(s). Note that for each data, consecutive execution windows may be grouped to a bigger execution window if the total communication time can be reduced by placing the data in the center of the new larger window.

3 Data Scheduling Algorithms

In this section, we present several approaches for the data scheduling problem. Two kinds of the algorithms are proposed: a single-center method in which the data stays at the same place for the whole execution period and the multiple-center methods which allow data movement during the run-time.

3.1 Single-Center Data Scheduling

The single-center data scheduling does not consider the data movement during the run-time. Once the data are initialized, they remain at the same place during the whole execution steps. By collecting all processor reference strings together, a single center data placement is achieved for each data. In other words, all execution steps for an application are put together in one execution window and the center for each data can be found. The method is shown in Algorithm 1. To be realistic, we assume each processor in the processor array can hold a limited number of data. If a processor is chosen to be the center for a data and the processor'memory is fully occupied, a secondary place to store the data needs to be decided. In Line 7, the process list is constructed for each data, containing a list of processors. It is sorted in the ascending order of the communication cost computed by assuming the data are assigned to each processor. The first processor in the list is the optimal place, which results in the minimum total communication cost, for each data. This kind of data placement is called Single-Center Data Scheduling (SCDS).

Algorithm 1 *Single-Center data placement*
Input : processor reference strings for each data
Output: Single-center initial data placement for each data.

```
1 foreach data i do
2   foreach processor node j do
3     Compute the total communication cost when data i put in node j;
4   od
5   Sort the processors in ascending order of the communication cost;
6   and place them in a processor list;
7   Assign data i to the first available processor in the processor list;
8 od
```

3.2 Multiple-Center Data Scheduling

The multiple-center data scheduling consists of two parts. The first part is the initialization part which gives initial data placement i.e., data placement for the first execution window. The second part provides data movement during the run-time by giving data placement for each data of different execution steps.

In the following, we present two algorithms for data scheduling which allows data to be moved during the run-time. The first approach is called Local-Optimal Multiple-Center Data Scheduling (LOMCDS) and the second approach is called Global-Optimal Multiple-Center Data Scheduling (GOMCDS). LOMCDS gives the local optimal data placement for each data with respect to each individual execution window. On the other hand, GOMCDS considers all the execution windows, and uses the shortest path method to find the global optimal data placement for each data in each execution window.

3.2.1 Local-Optimal Multiple-Center Data Scheduling

Assume that the execution windows are given. Hence, the processor reference string with respect to each data and each execution window can be decided. By applying Algorithm 1 to the first execution window, the center in this window with respect to each data can be obtained. The subsequence data placement for next time steps can be obtained by applying Algorithm 1 to the rest of execution windows. During the run-time, the data are moved to such centers according to these execution windows. Because the center of each execution window gives the minimum cost with respect to an execution window and data, this kind of data scheduling is called the local-optimal multiple-center data scheduling. Again, when the processor memory size is limited, the same technique (using processor list) as discussed in Subsection 3.1 can be applied to find the first available processor for each data for each execution window.

3.2.2 Global-Optimal Multiple-Center Data Scheduling

Since LOMCDS schedules the data into the local optimal center of each execution window without considering the cost of moving

data between these centers, these centers may not result in the minimum communication cost for all the execution windows. By considering both communication cost for each data reference in each execution window and the cost of moving the data between the consecutive execution windows, we can achieve a *global-optimal* multiple-center data scheduling for a given data. The placement in each execution window with respect to each data resulting from GOMCDS is called the *global-optimal center*. In order to find such centers of each execution window with respect to each data, we construct an edge-weighted directed acyclic graph (DAG), namely *cost-graph* and apply the shortest path algorithm to the cost-graph to obtain the global-optimal centers of each execution window for each data.

For each data, a cost-graph $G = (V, E, w)$ is constructed. Let V be the set of vertices including a pseudo source node s, a pseudo destination node d, and each $v_{i,j} \in V$, corresponds to the jth processor of execution window i, for $0 \leq i \leq n-1$, $0 \leq j \leq m-1$, where n is the number of the total execution windows and m is the number of the total processors. Let E be the set of edges, connecting node $v_{i,j}$ to node $v_{i+1,k}$ for all $0 \leq i \leq n-2$ and $0 \leq j, k \leq m-1$, including edges from source s to node $v_{0,j}$ and edges from node $v_{n-1,j}$ to node d for $0 \leq j \leq m-1$. The weight function w which maps from E to integer is defined as follows: for each edge $e = (s, v_{0,j})$, $w(e)$ is the total communication cost assuming the data is stored at processor j in execution window 0. For each edge $e = (v_{i,j}, v_{i+1,k})$, $w(e)$ is the summation of the communication cost between processors j and k, and the total communication cost assuming the data is stored at processor k in execution window $i+1$. For $e = (v_{n-1,j}, d)$, $w(e)$ is zero. Let v_{i,j_i} be the centers being selected by the algorithm. The path $s \rightarrow v_{0,j_0} \rightarrow v_{1,j_1} \rightarrow \ldots v_{n-1,j_{n-1}} \rightarrow d$ represents the sequence of the data movement among these centers. The summation of the w along the path is the total communication cost including the cost of moving data. Therefore, the shortest path from node s to d gives the global-optimal centers for each execution window.

The algorithm for finding the global-optimal centers with respect to each data is described as follows.

Algorithm 2 *Global-optimal multiple-center data scheduling*
Input : processor reference strings of all the execution windows with respect to each data
Output: the global-optimal multiple-center data scheduling.

```
1 foreach data i do
2   foreach processor node j do
3     Compute the total communication cost when data i put in node j;
4   od
5   Construct the cost-graph for data i;
6   Obtain the shortest path from source s;
7   Assign data i to the processor
8    lying on the shortest path in each execution window;
9 od
```

When there is no processor collision of data in each execution window, Algorithm 2 gives global-optimal centers resulting in the minimum communication cost for an application. Again, the idea of using the processor list can be applied when the memory constraint is violated.

3.3 Example

We demonstrate how these three data scheduling algorithms work using a simple example. Assume that the processor array size is 4×4, the number of the total execution window is 4, and the processor references of data D in each execution window are shown in Figure 1. The figure in each processor is the number of referenced by all processors which require data D.

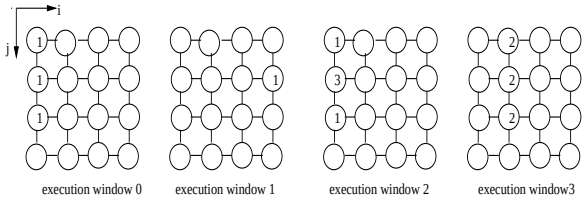

Figure 1. The processor references for data D

3.3.1 Single-Center Data Scheduling

In order to get SCDS for data D, all the references of data D are collected. In other words, all execution windows are merged in order to obtain the single center. Processor$(1,0)$ is the center where data D is placed with the minimum communication cost. Therefore, SCDS for data D is processor$(1,0)$. The total communication cost of D under SCDS is 17.

3.3.2 Local-Optimal Multiple-Center Data Scheduling

Since LOMCDS assigns the data placement for each data to the center of each execution window, from Figure 1, the center of each window is the processor which requires data D and gives the minimum total communication cost. For example, in execution window 0, processor $(1,0)$ is the center of this window for data D. The data placement of data D given by LOMCDS for data D are processors $(1,0)$, $(1,3)$, $(1,0)$ and $(1,1)$ for execution windows 0 to 3 respectively. The total communication cost of D under LOMCDS is 15.

3.3.3 Global-Optimal Multiple-Center Data Scheduling

Using the same method as we discussed in Subsection 3.2.2, after the cost-graph for data D is constructed, the shortest path from node s to d gives GOMCDS for data D. GOMCDS for data D are processors $(1,0)$, $(1,0)$, $(1,0)$ and $(1,1)$ for execution windows 0 to 3 respectively. The total communication cost of Data D under GOMCDS is 12.

4 Execution Window Optimization

As we show in previous sections, in order to reduce the total communication cost of an application, each data need not be placed at the same position for all execution windows. Changing the data position during the run-time for different execution windows can give a significant reduction of the communication cost. The size of the execution window plays a role in the total reduction

in the communication cost. Depending on the number of instructions executed per window, execution window may be small or large. If the execution window is too small, the cost of moving data between centers of the windows may be large. In this section, we study the possibility of grouping consecutive execution windows with respect to each data into a bigger one. By putting the data to the center of the new execution window, we can reduce the cost of moving data between centers as well as the total communication cost. Before we present the execution window optimization algorithm, we first introduce some intuition and properties related to the optimization method.

4.1 Execution Window Properties

If merging consecutive execution windows together and putting the data to the center of the new window can reduce the total communication cost, we group these execution windows. Nevertheless, it is difficult to decide how many groups we should have and how many execution windows each group should contain for a data of a given application so that the total communication cost will be minimum.

Consider a basic group consisting of two execution windows, we observe that when we are given two centers, grouping two execution windows cannot reduce the total communication cost. Let $cost(D, T_i, p)$ be the cost of data D in execution window T_i when data D is placed at processor p. If the centers of the two execution windows are the two closest pair of the local optimal centers, grouping these two execution windows cannot reduce the total communication cost. Lemma 1 states that with only two execution windows the communication cost increases strictly monotonically along the path between the two centers in the one-dimensional processor array. Theorem 2 generalizes Lemma 1 into the two-dimensional processor array platform. Based on Lemma 1 and Theorem 2, Theorem 3 states that grouping two execution windows together cannot reduce the total communication cost. Due to the space limitation, we omit the proofs of these theorems (see [5] for details of the proofs).

Lemma 1 (1-dimensional processor array) *Let p_{i1} and p_{i2} be the closest pair of the local optimal centers with respect data D for execution window T_0 and T_1. Hence, $cost(D, T_0, p_i)$ along the direction from p_{i1} to p_{i2} increases strictly monotonically.*

Theorem 2 (2-dimensional processor array) *Let $p_{i1,j1}$ and $p_{i2,j2}$ be the closest pair of the local optimal centers with respect to data D for execution window T_0 and T_1. $cost(D, T_0, p_{i,j})$ along any path which gives the shortest distance between processor $p_{i1,j1}$ and $p_{i2,j2}$ increases strictly monotonically.*

Theorem 3 *If p_1 and p_2 are the closest pair of the local optimal centers with respect to data D for two consecutive execution windows, T_0 and T_1, grouping T_0 and T_1 does not reduce the total communication cost with respect to data D.*

From the properties of grouping, we conclude that the number of execution windows considered for each group can affect the reduction in the total communication cost. However, to exhaustively finding all possible choices of grouping may be costly. In the following subsection, we introduce our greedy heuristic that efficiently finds the number of execution windows in a group and gives an effective new set of execution windows which can reduce the total communication cost.

4.2 Grouping Algorithm

In order to explore more chances of the reduction in the total communication cost, we group the consecutive execution windows to a new window as long as the total communication cost obtained using the new window is not worse than that of the original set of windows.

We describe the grouping algorithm as follows:

Algorithm 3 *Optimizing Execution Windows*
Input : a set of the execution windows $T = \{t_i\}$ for data D
Output: an optimized data scheduling

```
1  calculate center(T);/* compute the center of each execution window */
2  start = 0;
3  j = 1;   /* the total number of the execution windows is MAX */
4  while (j ≤ MAX) do
5        group consecutive windows from start to j into a new window t_new;
6        T_NEW = t_new ∪ (T − ∪_{i=start}^{j} t_i);
7        compute center(T_NEW);
8        if COST(T_NEW) ≤ COST(T)
9          then  confirm this grouping;
10           else start = j;          /* separate into a new group at j */
11                j++;
12         od
```

In Algorithm 3, T is a set of execution windows t_i. Functions $COST(T)$ and $center(T)$ compute the total communication cost and centers for all the execution windows in T respectively. Intuitively, $COST(T)$ is the total communication cost of all the execution windows plus the total communication occurring from moving the data among centers. They can be obtained by either SCDS, LOMCDS or GOMCDS. The algorithm takes the execution windows as an input and computes the centers of these windows as well as their communication cost. Beginning with the first execution window, the algorithm tries to group the consecutive windows into a new window, as long as this grouping does not increase the total communication cost. Otherwise, the algorithm constructs a new group starting from this window and redo the process. Finally, the algorithm gives a set of new windows and the new centers of these windows are obtained.

5 Experimental Results

We compare the total communication cost of our approaches with the straight-forward method which assigns each data element to the corresponding processor in a row-wise fashion. In the experiments, the data reference strings are generated from the following benchmarks: LU factorization, matrix multiplication and the code shown in [5]. The experiments in each table are done using the same processor array size 4×4. In these tables, we assume that the memory size of processor is twice more than the minimum memory size it requires. For example, with the data size of 8×8 and the processor array size of 4×4, the memory size of each processor is eight.

B.	Size	S.F.	SCDS		LOMCDS		GOMCDS	
			Comm.	%	Comm.	%	Comm.	%
1	8x8	1418	776	45	603	57	568	**60**
	16x16	11656	6168	47	5842	50	5190	**55**
	32x32	99296	49952	50	49050	51	45171	**55**
2	8x8	3716	3160	15	2652	29	2446	**34**
	16x16	51520	49152	5	39812	23	38400	**25**
	32x32	791168	786628	.01	623242	21	598992	**24**
3	8x8	8156	7526	8	6773	17	5949	**27**
	16x16	112106	108136	4	89788	20	87144	**22**
	32x32	1676464	1645380	2	1363630	19	1335080	**20**
4	8x8	9298	8064	13	6773	27	5949	**36**
	16x16	120930	114602	5	96776	20	91483	**24**
	32x32	1741008	1697620	2	1419064	18	1365370	**22**
5	8x8	7432	6320	15	5304	29	4903	**34**
	16x16	103040	98304	5	79624	23	76624	**26**
	32x32	1673256	1582336	5	1246484	26	1199476	**28**

Table 1. The total communication cost before grouping: Processor array size = 4×4

B.	Size	S.F.	SCDS		LOMCDS		GOMCDS	
			Comm.	%	Comm.	%	Comm.	%
1	8x8	1418	584	59	583	59	568	**60**
	16x16	11656	5437	53	5259	55	5190	**55**
	32x32	99296	47154	53	45363	54	45171	**55**
2	8x8	3716	3034	18	2587	30	2446	**34**
	16x16	51520	44978	13	39602	24	38400	**25**
	32x32	791168	733298	7	622771	21	598992	**24**
3	8x8	8156	7244	11	5938	27	5886	**28**
	16x16	112106	107566	4	88056	21	87144	**22**
	32x32	1676464	1643082	2	1356649	19	1335080	**20**
4	8x8	9298	8064	13	6657	17	5949	**36**
	16x16	120930	114032	6	92933	23	91483	**24**
	32x32	1741008	1695322	3	1388368	20	13653700	**22**
5	8x8	7432	5606	25	4981	33	4720	**36**
	16x16	103040	90684	12	74006	28	73008	**29**
	32x32	1673256	1408414	16	1153769	27	1145542	**32**

Table 2. The total communication cost after grouping: processor array size = 4×4

Column "B." lists the benchmarks tested. Benchmark 1 computes LU factorization, 2 computes the square of a matrix, 3 combines benchmark 1 and CODE1 in [5], 4 which combines benchmark 2 and CODE1, and 5 combines the CODE1 and the code in the reverse execution order of the CODE1. Column "Size" shows the size of the data array tested. Column "S.F." shows the total communication time using the straight-forward data distribution. Columns SCDS, LOMCDS and GOMCDS in Tables 1–2 show the results of the single-center data scheduling, the local-optimal multiple-center data scheduling and the global-optimal multiple-center data scheduling respectively. Table 1 presents the total communication cost of these benchmarks before we apply the execution window optimization (Algorithm 3). Column "%" shows the percentage of the improvement over the straight forward way. Consider the average improvement, we conclude that the performance of GOMCDS is the best while LOMCDS outperforms SCDS. All of the proposed schemes give significant improvement compared with the straight forward data distribution. GOMCDS and LOMCDS give better performance than SCDS because the data movement was taken into account. Since GOMCDS considers the data movement in a global view, GOMCDS obtains the best performance of these three approaches. From these experiments, we found out that considering the data movement can be more effective especially for the benchmarks with complicate data reference patterns.

Table 2 presents the total communication cost of these benchmarks after we apply the execution window optimization (Algorithm 3 assuming using LOMCDS to compute centers). The experimental results show the performance is further improved by applying the grouping algorithm. In general, the results in these tables demonstrate the effectiveness of our these approaches, SCDS, LOMCDS, GOMCDS as well as the grouping method and the improvement over the straight forward data distribution.

6 Conclusion

In this paper, we propose the data scheduling techniques for Processor-In-Memory arrays which minimize the total interprocessor communication. The algorithms are general enough to handle non-uniform loops, which find the initial data placement as well as data movement during the run-time. As a result, the total interprocessor communication time of an application can be significantly reduced. The effectiveness of our approaches are shown by comparing the total communication cost obtained by using traditional row-wise data distribution and our methods for well-known benchmarks.

References

[1] Young Won Lim, Prahanth B. Bhat, and VIktor K. Prasanna. Efficient algorithms for block-cyclic redistribution of arrays. In *Proceedings of the Eighth IEEE Symposium of Parallel and Distributed Proceeding*, 1996.

[2] Young Won Lim, Neungsoo Park, and Viktor K. Prasanna. Efficient algorithms for multi-dimensional block-cyclic redistribution of arrays. In *Proceedings of International Conf. on Parallel Processing*, 1997.

[3] Angeles G. Navarro, Yunheung Paek, Emiliio L. Zapata, and David Padua. Compiler techniques for effective communication on distributed-memory multiprocessors. In *Proceedings of International Conf. on Parallel Processing*, 1997.

[4] Tajeev Thakur, Alok Choudhary, and J. Ramanujam. Efficient algorithms for array redistribution. *IEEE Transaction on Parallel and Distributed Systems*, 7(6), 1996.

[5] Yi Tian, Edwin Sha, Chantana Chantrapornchai, and Peter Kogge. Data scheduling on processor-in-memory arrays based on data placement and data movement. Technical Report 97-09, CSE Dept., University of Notre Dame, 1997.

[6] Yi Tian, Edwin Sha, Chantana Chantrapornchai, and Peter Kogge. Efficient data placement for processor-in-memory array processors. In *The Ninth ISATED International Conference on Parallel and Distributed Computing and Systems*, 1997.