The Journal of Supercomputing, 29, 157–170, 2004
© 2004 Kluwer Academic Publishers. Manufactured in The Netherlands.

A Divide-and-Conquer Algorithm for Irregular Redistribution in Parallelizing Compilers

HUI WANG hwang@u-aizu.ac.jp
MINYI GUO minyi@u-aizu.ac.jp
DAMING WEI dm-wei@u-aizu.ac.jp
School of Computer Science and Engineering, University of Aizu, Aizu-Wakamatsu, Fukushima 965-8580, Japan

Abstract. In order to achieve higher load balancing, it is necessary to solve irregular block redistribution problems, which are different from regular block-cyclic redistribution. High Performance Fortran version 2 (HPF-2) provides irregular distribution functionalities, such as GEN_BLOCK and INDIRECT. This paper is devoted to develop an efficient algorithm that attempts to obtain near optimal scheduling while satisfying the conditions of minimal message size of total steps and the minimal number of steps for irregular array redistribution. The algorithm intends to decrease the computation costs by dividing the whole block into sub-blocks and solving the sub-problems accordingly, and then merging them together to get final results. Simulation results show that our algorithm has comparable performance with a relocation algorithm developed previously (H. Yook and M. Park. *Proceedings of the IASTED International Conference Parallel and Distributed Computing and Systems*, Nov. 3–6, MIT, Boston, USA, 1999).

1. Introduction

In many scientific and engineering applications, distribution of an array in a parallel program cannot remain fixed throughout the program. It is very often to change the distribution of an array at run-time. At that time, each processor is required to know which local elements of the array should be sent to the specified processor, and which elements of the array should be received from the specified processor. So an efficient redistribution communication algorithm is needed to relocate the elements of the array among different processors and perform the necessary communication among different processors. Clearly, it is much faster to access local data than to access remote data located at other nodes in distributed memory multi-computers, since the remote access incurs expensive inter-processor communication overhead. Therefore, it is important to partition the program and data efficiently among processors so that it minimizes communications while maximizing potential parallelism.

As a data parallel programming language, High Performance Fortran version 2 (HPF-2) [3] is an informal standard for extensions to FORTRAN 95 to assist its implementation on parallel architectures. HPF-2 provides compiler directives for programmers to specify data mapping, such as array distribution. For the generalized block distribution, GEN_BLOCK distribution scheme allows unequal-sized consecutive segments of an array to be mapped onto consecutive processors [3]. It is quite useful since many applications need data redistribution during computation since different phases there require different data mapping in order to execute loops efficiently without any data dependencies [9].

```
      PARAMETER (S = /3, 5, 9, 4, 13, 4, 2/)
!HPF$ PROCESSORS P(7)
      REAL A(40), B(40), new(7)
!HPF$ DISTRIBUTE A( GEN_BLOCK (S) ) ONTO P
!HPF$ DYNAMIC B
      new = /2, 5, 3, 6, 8, 6, 10/
!HPF$ REDISTRIBUTE A( GEN_BLOCK(new) )
```

Above is an example of the GEN_BLOCK scheme and the redistribution in HPF-2. Given the above specifications, array elements $A(1{:}3)$ are mapped onto P_0, $A(4{:}8)$ are mapped onto P_1, so on, and $A(39{:}40)$ are mapped onto P_6. After executing REDISTRIBUTE directive, A is redistributed based on the new array, that is, a new distribution scheme is formed such as $A(1{:}2)$ onto P_0, $A(3{:}7)$ onto P_1, and so on.

How to implement an efficient redistribution routine is quite important in parallel compilers. In this paper, we propose an efficient algorithm—a divide-and-conquer algorithm—that attempts to obtain near optimal scheduling while satisfying the conditions of minimal message size of total steps and the minimal number of steps for irregular array redistribution. The algorithm intends to decrease the computation costs by dividing the whole block into sub-blocks and solving the sub-problems accordingly, and then merging them together to get final results. Finally, simulation results show that our algorithm has comparable performance with other algorithms developed previously.

2. GEN_BLOCK redistribution communication models

In this section, GEN_BLOCK redistribution communication models are implemented. A redistribution *R* is a set of routines that transfer all the elements in a set of source processors *SP* to a set of target processors *TP*. Generally, the sending and receiving phases indicate that the array redistribution problem comprises two sub-problems. First, the array to be redistributed should be efficiently scanned or processed in order to build all the messages that are to be exchanged between processors. Then, all messages must be efficiently scheduled so as to minimize communication overhead. Each processor generates massages to send to other processors. The sizes of the messages are specified by values of a user-defined integer for array mapping from source processor to target processor.

To develop a communication schedule, we can use either blocking scheduling algorithms or non-blocking scheduling algorithms. The blocking scheduling algorithms are based on blocking communication primitives, while the non-blocking scheduling algorithms are based on non-blocking communication primitives. In general, because the non-blocking scheduling algorithms avoid excessive synchronization overhead, they are faster than the blocking scheduling algorithms. However, the non-blocking communication primitives need as much buffering as the data being redistributed [5, 13]. It makes non-blocking scheduling algorithm much expensive in some situations. In this paper, all of our discussions are based on blocking communication primitives.

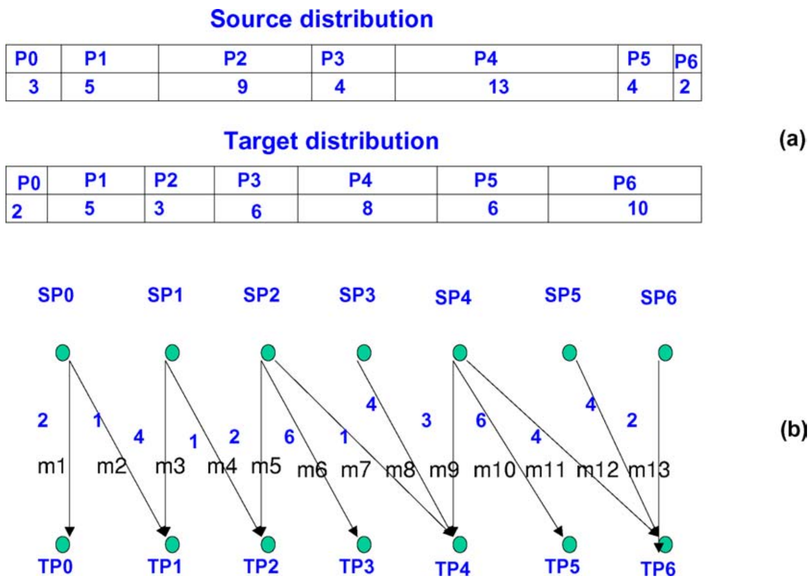
In distributed memory parallel computing, the inter-processor communication overheads can be represented using an analytical model of typical distributed memory machines. To represent the communication time of a message passing operation, the model introduces two parameters: the start-up time T_s and the unit data transmission time T_m. Based on the one-step communication time in Eq. (1), Eq. (2) provides the total communication time of collective message passing delivered in multiple communication steps.

$$T_{\text{step}} = T_s + m \times T_m, \tag{1}$$

$$T_{\text{total}} = \sum T_{\text{step}}. \tag{2}$$

Since node contention considerably influences overhead in communication among source processors and target processors, a processor can only receive at most one message from one other processor in each communication step. Similarly, a processor can only send at most one message to one other processor in each communication step. Unlike regular redistribution, it has not a cyclic message-passing pattern. If SP_i sends messages to both TP_{j-1} and TP_{j+1}, therefore it must certainly send a message to TP_j, vice versa, where SP_i and TP_{j-1}, TP_j, and TP_{j+1} are a source processor node and target processor nodes, respectively.

Figure 1 shows the redistribution communication of the example given above and describes the sending and receiving messages for GEN_BLOCK scheme in detail. In this example, there are 7 source processors and 7 target processors, respectively. Source processor SP_0 has 3 unit data, and target processor TP_0 has 2 unit data. After the messages



Source distribution

P0	P1	P2	P3	P4	P5	P6
3	5	9	4	13	4	2

Target distribution

P0	P1	P2	P3	P4	P5	P6
2	5	3	6	8	6	10

Figure 1. An example of redistribution communication. (a) source processors' and target processors' messages distribution; (b) messages communication between source processors and target processors.

S\T	0	1	2	3	4	5	6
0	2 m1	1 m2					
1		4 m3	1 m4				
2			2 m5	6 m6	1 m7		
3					4 m8		
4					3 m9	6 m10	4 m11
5							4 m12
6							2 m13

Figure 2. The matrix implementation of the above example. The order of the message is labelled in a smaller font. S and T represent source processors and target processors, respectively.

distributed onto source and target processors, it is easy to figure out how many messages each source processor should send out, to which target processor it should send out, and how many unit data it should send to the target processor. Clearly, in this example, SP_0 sends 2 unit data message to TP_0, and 1 unit datum message to TP_1, and so on.

We can consider the length of a message sent from source processor SP_i to target processor TP_j as an element $m_{i,j}$ of a redistribution table. Figure 2 shows the redistribution table of the above example, where the row and column represent the indexes of source processors and target processors, respectively. In Figure 2, SP_0 sends 2 unit data to TP_0, and sends one unit data to TP_1. SP_1 sends 4 unit data to TP_1, and sends one unit datum to TP_2, and so on. Otherwise, if there is no communication between the source and target processors, the corresponding entry is zero. The zero-value elements are ignored.

Since a neighbor message set (NMS) can be defined as a set of the messages (including more than one message) that to be sent from same processor or to be received by same processor, an NMS is a set of elements in the same row or same column in a redistribution table. An NMS can be labelled accordingly as NMS_k, ($k = 1, 2, 3, \ldots$), from left-up side to right-down side in a redistribution table. Obviously, it is possible that a message belongs to two different NMSs, or $\text{NMS}_k \cap \text{NMS}_{k'} (k \neq k')$, which is defined as a *link* message. Therefore, the GEN_BLOCK redistribution problem has the following properties:

(1) The total number of the neighbors of an element $m_{i,j}$, $N_{\text{Total}} = B_{\text{left}} + B_{\text{right}} + B_{\text{up}} + B_{\text{down}} \leq 2$, where B_{left} (or, B_{right}, B_{up}, B_{down}) is 1 if $m_{i,j}$ has a neighbor in that direction, or is 0 if it has not. It indicates that one message can belong to at most two different NMSs.

(2) In a redistribution table, the total number of non-zero elements N_m satisfies with $\max(l, n) \leq N_m \leq l + n - 1$, where l and n are the sizes of row and column, respectively. (It is reasonable to eliminate a column or a row if all elements in a column or a row are zero. Assume that there is at least one element in a column or a row. Thus

the total number of messages N_m satisfies $N_m \geq \max(l, n)$. If there are total $l + n$ messages or more, at least a cyclic message passing pattern, $\{m_{i,j}, m_{j,i}\}$, or a conflict, $\{m_{i,j}, m_{i,j'}, m_{i',j}, m_{i',j'}\}$ can be found.)

(3) Elements in same row or same column cannot be scheduled in same time step. Clearly, elements in same row or same column represents messages sent or received by the same processor.

To efficiently design redistribution scheduling algorithms, it is better to concentrate on how to organize the order of each message so that the total costs for GEN_BLOCK redistribution can be minimized. The aims for the GEN_BLOCK redistribution algorithm are to find:

(1) Minimal communication step. The minimum number of communication steps is the maximum number of fan-out (fan-in) arrows of the processor nodes in Figure 1(a), or the maximum number of elements in NMSs. Since the messages in the same neighbor message set cannot be scheduled in same communication step, size of total communication steps should be equal to or larger than the maximum number of elements in NMSs. This is called *minimal step condition* in the following context.
(2) Minimal the message size of total steps. The total communication time is a linear function of the message size of total steps. In order to reduce total communication time, one efficient way is to minimize the size of total steps. This is called *minimal size condition* in the following context.

3. Divide-and-conquer algorithm

In the following discussion, for the sake of simplicity, we concentrate on two-dimensional array redistribution. Our divide-and-conquer algorithm can also be generalized to that of higher dimensions.

To develop a redistribution algorithm which satisfies both (1) and (2) goals is a quite difficult task. Therefore, we propose a divide-and-conquer redistribution algorithm which contains three parts: (1) breaking a problem into a set of sub-problems that are similar to the original problems with smaller size, (2) solving these sub-problems recursively, and (3) creating a solution to original problem by combining these solutions to sub-problems together. This algorithm can be viewed as a divide-and-conquer algorithm, which has been widely used in binary searching, Fast Fourier Transformation, matrix inversion, etc.

When breaking the problem into smaller units, we first separate the problem into a set of sub-problems where each sub-problem can be scheduled independently. After obtaining the solutions to sub-problems, we combine them together to form a solution to the original problem. Each neighbor message set, NMS_i, is a unit of a sub-problem. After dividing a redistribution table into NMSs, we group two neighboring NMSs as a pair, such as $\{\text{NMS}_1, \text{NMS}_2\}, \ldots, \{\text{NMS}_{2i+1}, \text{NMS}_{2i+2}\}$, and so on. Figure 3 shows the separating and grouping processes for the example shown in Figure 2. In this example, there are totally 8 NMSs, and they are grouped into 4 pairs: $\{\text{NMS}_1, \text{NMS}_2\}$, $\{\text{NMS}_3, \text{NMS}_4\}$, $\{\text{NMS}_5, \text{NMS}_6\}$,

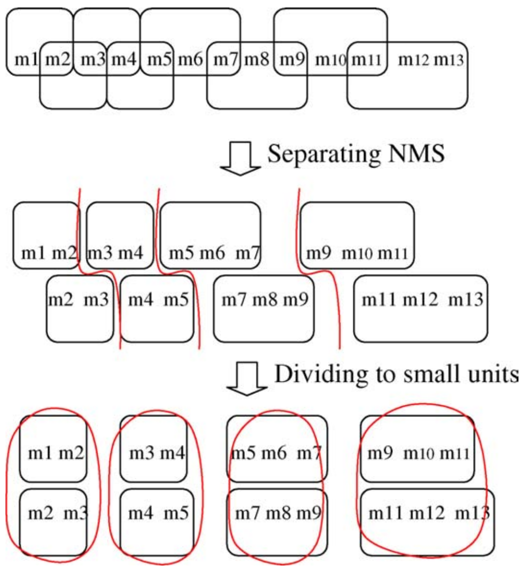


Figure 3. Breaking the problem by separating NMSs and dividing into the smaller units for the example in Figure 1.

{NMS_7, NMS_8}. The example in Figure 3 indicates that all separations are taken place at the link messages.

After obtaining the pairs of NMSs, we begin the merging processes to produce the scheduling. First, recursively merge neighboring NMSs pairs. In phase 0, we generate the scheduling table for each NMSs pairs. There are total k pairs, while each pair forms a group. In phase 1, scheduling tables for the group of {NMS_1, NMS_2, NMS_3, NMS_4}, ..., the i-th group of {NMS_{4i+1}, NMS_{4i+2}, NMS_{4i+3}, NMS_{4i+4}}, ..., will be merged together. Recursively, in phase j, scheduling tables for the i-th group of {$\text{NMS}_{j\times(i+1)}$, $\text{NMS}_{j\times(i+1)+1}$, ..., $\text{NMS}_{j\times(i+j)}$} will be merged together.

Then, generate the scheduling tables for each group. The generating processes are quite same for each group in every phase. First, we detect the conflict information, and pick up the link message and those messages scheduled with it. Then sort the remaining messages to make an optimal solution. If a message has more possible position in the scheduling table, it stands with its closest ceiling. Finally, the link message together with those scheduled messages, will be added into the scheduling table.

Figure 4 shows the merging processes from the smallest units to generate the scheduling tables recursively for the above example. In Figure 4, we assume that the length of total messages in step S_i, L_{Si}, satisfies $L_{Si} \geq L_{S(i+1)}$. In phase 0, $m2$, $m4$, $m7$, and $m11$ are link messages. According to the length of messages, the algorithm generates scheduling tables {$m3$, $m1$, $m2$}, {$m3$, $m5$, $m4$}, {($m6$, $m8$), ($m9$, $m5$), $m7$}, and {($m10$, $m12$), $m11$, ($m9$, $m13$)} for each pair, respectively. In a scheduling table, all data are in descending order. For example, here $m3 \geq m1 \geq m2$, and $m6 \geq m8$. However, it is difficult to determine which is larger, $m8$ or $m5$, just from the expression {($m6$, $m8$), ($m9$, $m5$), $m7$}.

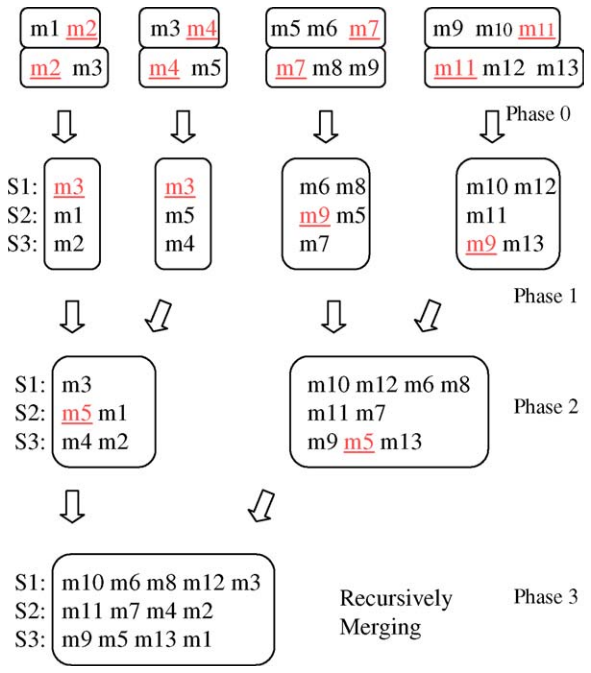


Figure 4. To form the solution to the original problem, it merges from the smallest units recursively for the above example. In each phase, these active link messages in merging processes are emphasized with underscores.

In phase 1 of Figure 4, we detect that $m3$ and $m9$ are link messages. Further merging is straight forwarded for the left pair because both of them have the link message $m3$ in the same scheduling step. When the link message $m9$ is in the different scheduling step, we rearrange $m9$ and its related messages to perform optimal scheduling results. Similar situation happens in phase 2. Finally we obtain the scheduling table for the whole redistribution problem.

Our algorithm can generate sub-optimal communication scheduling table. It satisfies minimal size condition (higher priority) and minimal step condition. The algorithms are described in Figures 5 and 6. The time complexity of our algorithm can be estimated as $O(k(\log k)^2)$, where k is the number of link messages, because message sorting processes need $O(k \log k)$, and the merging processes need $O(\log k)$ steps totally.

4. Performance and discussion

To evaluate the effect of our algorithm, we implement the divide-and-conquer algorithm and compare it with the relocation algorithm [14]. The relocation algorithm consists of two phases. First, it sorts all messages in descending order, and allocates messages one by one. When it cannot allocate a message within the minimal step condition, it goes to the relocation phase. When dividing the schedule into two sets: a left-side set and a right-side

```
Divide-and-conquer Algorithm:
   Input: the distribution scheme M,
          the source, target processor numbers p, q
   Output: scheduling table S(p,q)
  {
      separation $NMS_1$, $NMS_2$, …, $NMS_k$;// k: total number of NMSs
      // First phase: separation and grouping
      for (i = 0; i < k; i += 2) {
         picking left-down link message out, putting it aside;
         tmpMax(i) = Join($NMS_{i+1}$, $NMS_{i+2}$);
         adding left-down link message to tmpMax(i);
         updating S(p,q);
      }
      // Second phase: recursively merging
      j = 2;
      while (j < k) {
         for (i = j; i < k; i += 2 × j) {
             if (Order(i in $NMS_i$) != Order(i in $NMS_{i+1}$)) {
                Merge(sub-matrix from M(i-j) to M(i+j));
             }
         }
         j = 2 × j;
      }
  }
```

Figure 5. Divide-and-conquer algorithm for GEN_BLOCK redistribution.

set in the relocation phase, both of them are kinds of GEN_BLOCK redistributions and can be treated as an input of the relocation algorithm. One of them may be disregarded because one of them does not satisfy the minimal size condition.

The simulation program generates a set of random numbers in a given range as the size of message. Here we suppose that the number of source processors equals to the number of target processors. However, it is possible that some processors do not contain any elements. To keep the balance between source processors and target processor, we also suppose the total size of messages in both source processors and target processors are equal.

The input parameters of the program are two arrays: a message set of source processor and a message set of target processors. Obviously, the scheduling matrix can be generated according to the input arrays. When the program runs, the algorithm presents a communication scheduling table, which figures out the time step for each message.

In our experiments, the up-bound of message size is one of the important input parameters. Normally, a random number is generated according to the up-bound of message size. What we concern here is the effects of message size to the simulation results of redistribution algorithms. In Figure 7, the events percentage is plotted as a function of the message size. The total generated events is about 100000 for each message size. Events percentage shown in Figure 7 means the number of events is normalized. The up-bound message size changes from 10 to 1280. For each different message size, 3 weighted tuples are compared. In Figure 7, "DCA Better" represents the percentage of the number of events that the divide-and-conquer algorithm has lower total communication time than the relocation

```
Merging Algorithm (subroutine):
   Range: sub-matrix from M(i-j) to M(i+j))
   Location: link message i
  {
      tmpOrder =0 ;
      while (1) {
         // tmpNum = number of elements in link message i's
         // row or column with tmpOrder; if any element is larger
         // than link message i, tmpNum++;
         if (tmpOrder == Max(number of messages in i's row,
                            number of messages in i's column) {
            S[p[i]][q[i]] = tmpOrder;
            break;
         } else if (tmpNum == 0) {
            S[p[i]][q[i]] = tmpOrder;
            break;
         } else if (tmpNum == 1) {
            relocation with minimum costs among {
            order(i) == Order(i in NMS$_{i+1}$),
            order(i) == Order(i in NMS$_{i}$),
            order(i) when relocating link message i's row and column};
            updating S(x,y) in sub-matrix from M(i-j) to M(i+j);
            break;
         }
         tmpOrder++;
      }
  }
```

Figure 6. Merging sub-problems methods in divide-and-conquer algorithm for GEN_BLOCK redistribution.

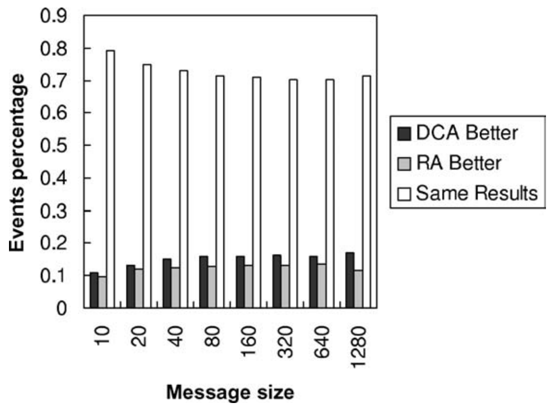


Figure 7. The events percentage of both the divide-and-conquer algorithm and the relocation algorithm is shown as a function of the message size with 8 processors.

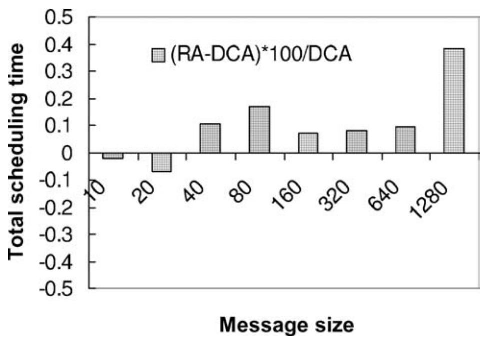


Figure 8. The normalized difference percentages of average communication times of the relocation algorithm and the divide-and-conquer algorithm, $(RA - DCA) \times 100/DCA$, are shown as a function of message size with 8 processors.

algorithm, while "RA Better" gives the reverse situation. If both algorithms have same total communication time, events are collected into "Same Results" cylinder. The simulation results in Figure 7 are generated with 8 processors as an example. From the figure, we can see the message sizes from 40 to 640 give quite similar results, while the results from the message sizes 10, 20, or 1280 are a little bit deviation from the others. Generally, the message size of 100 is selected for generating redistribution events in the rest of this paper. Before that, Figure 8 is plotted to validate whether it is reasonable or not to use the message size of 100. The normalized difference percentages of average communication times of the relocation algorithm and the divide-and-conquer algorithm, $(RA - DCA) \times 100/DCA$, are shown as a function of message size from 10 to 1280. Similar to the conclusion derived from Figure 7, results with message sizes from 40 to 640 are close, while others show the abnormalities.

Figure 9 shows the simulation results of both the divide-and-conquer algorithm and the relocation algorithm with different number of processors. For each different number of processors, events percentages for "DCA Better", "RA Better", and "Same Results" are

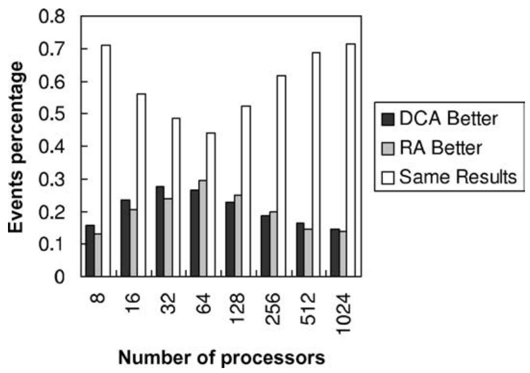


Figure 9. The events percentage of both divide-and-conquer algorithm and the relocation algorithm is plotted with different number of processors.

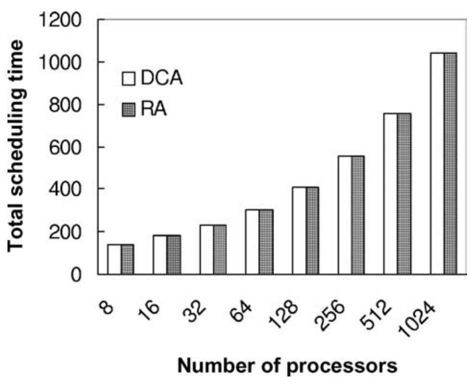


Figure 10. The averaged total communication times of both the relocation algorithm and the divide-and-conquer algorithm as a function of the number of processors.

compared in the figure. The number of processors stride from 8 to 1024. Most of results are filled into "Same Results" cylinder, while both the divide-and-conquer algorithm and the relocation algorithm have chances in the ascendant. The results indicate that both the relocation algorithm and the divide-and-conquer algorithm have good performance.

The average communication times for both the divide-and-conquer algorithm and the relocation algorithm are shown in Figure 10. The average communication time in an event increases from about 100 to about 1000 with the increasing of the number of processors. The simulation results are quite same from both algorithms. Figure 11 shows the normalized difference percentages of average communication times of the relocation algorithm and the divide-and-conquer algorithm. The differences between the divide-and-conquer algorithm and the relocation algorithm is quite small, while each of them dominates one half of different processors.

Both the divide-and-conquer algorithm and the relocation algorithm have advantages and disadvantages. One disadvantage of the relocation algorithm exists in its relocation phase. In the relocation phase, the *bad* link message cannot be allocated to other time

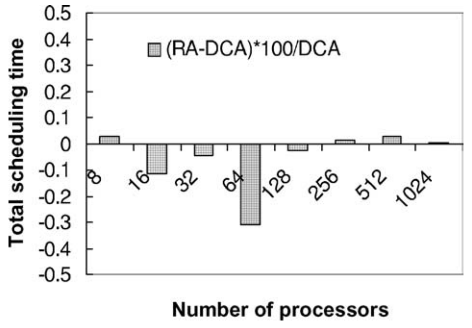


Figure 11. The normalized difference percentages of averaged total communication times of the relocation algorithm and the divide-and-conquer algorithm for different number of processors.

steps but only two time steps, for the left-side relocation and for the right-side relocation, respectively. Obviously, it is possible to include other time steps in order to reduce the total communication time. The divide-and-conquer algorithm requires *n* steps totally to merge *n* link messages. But the disadvantage in the divide-and-conquer algorithm is that many *bad* link messages can be generated during the processing of merging. For example, in phase *i*, the link message can be joined two NMSs together naturally. But in phase *i-1*, it is possible that the time step of the link message is changed, so that the link message becomes one of *bad* link messages. Bad link messages make the scheduling quite inefficient. It indicates the list scheduling, which is the first part of the relocation algorithm, is an efficient algorithm for GEN_BLOCK redistribution.

5. Related work

Recent work on array redistribution can be divided into two categories: regular redistribution and irregular redistribution. Many researches have mainly concentrated on the efficient index computation for generating the communication messages to be exchanged by the processors involved in the redistribution. Efficient index set computation has been used to compile array redistribution by assigning source array A to target array B [5]. A non-blocking communication algorithm presented by Thakur et al. [13] can perform the computation and communication in parallel which makes it more efficient than blocking communication. A redistribution technique based on the descriptors called *pitfalls* has been presented in [12].

The following papers concentrate on the communication optimization in redistribution. A generalized circulant matrix formalism was proposed by Lim et al. [8] in order to reduce the communication overheads for CYCLIC(n)-to-CYCLIC(kn) redistribution. Park et al. [10] proposed an extended algorithm that reduces the overall time for communication by considering the data transfer, communication schedule, and index computation costs. Kalns and Ni [4] presented a mapping technique which can map data to logical processor to minimize the total amount of communication data for BLOCK-to-CYCLIC(n) redistribution.

Guo and Nakata [1] developed techniques to reduce overheads in both index computation and inter-node communication. Guo et al. [2] presented an efficient index computation method and generated a schedule that minimizes the number of communication steps and eliminates node contention in each communication step.

There are relative little papers on irregular array redistribution. However, as we may see, it is an important problem in HPF-2. Leair et al. [6] have implemented the GEN_BLOCK data distribution in PGHPF, a High Performance Fortran compiler [11]. Some simple benchmark results show that the GEN_BLOCK distribution increasing the speed up to twice over regular distribution.

Yook and Park [14] proposed an algorithm for the redistribution of one-dimensional arrays in GEN_BLOCK. The algorithm exploits a spatial locality in message passing from seemingly irregular array redistribution. The algorithm attempts to obtain near optimal scheduling by trying to minimize the size of communication step and the number of steps. The algorithm shows good performance in typical distributed memory machines.

There exists a performance tradeoff between the expected higher efficiency of a new distribution for subsequent computation and the communication cost of redistributing the data among processor memories. Lee et al. [7] focused on reducing the communication cost in GEN_BLOCK redistribution using a logical processor reordering method. According to their experiments on CRAY T3E, the algorithms show good performance comparing typical GEN_BLOCK redistribution, which does not reorder logical processor numbers.

6. Conclusion

In HPF-2 compilers, GEN_BLOCK array redistribution makes unequal-sized blocks mapping possible. An efficient algorithm for GEN_BLOCK distribution is developed to achieve sub-optimal solutions. The divide-and-conquer redistribution algorithm first breaks the problem into a set of sub-problems, and then solves these sub-problems recursively and creates the solution to original problem by combining these solutions to sub-problems.

Results from our algorithm are compared with these from the relocation algorithm. The results indicate that both of them have good performance on GEN_BLOCK redistribution. We discuss the advantages and disadvantages of both the divide-and-conquer algorithm and the relocation algorithm.

Multi-dimensional GEN_BLOCK array redistribution is extended versions of one-dimensional array redistribution. Thus, the minimal step condition and the minimal size condition are also important in multi-dimensional GEN_BLOCK redistribution scheduling. Multi-dimensional GEN_BLOCK array redistribution scheduling can be performed by extending our algorithm.

References

1. M. Guo and I. Nakata. A framework for efficient data redistribution on distributed memory multicomputers. *The Journal of Supercomputing*, 20(3):243–265, 2001.
2. M. Guo, I. Nakata, and Y. Yamashita. Contention-free communication scheduling for array redistribution, *Proceedings of the International Conference on Parallel and Distributed Systems*, Dec. 1998, pp. 658–667.
3. High performance fortran forum. *High Performance Fortran Language Specification version 2.0*, Rice University, Houston, TX, Jan. 1997.
4. E. T. Kalns and L. M. 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, and P. Sadayappan. Efficient index set generation for compiling HPF array statements on distributed-memory machines. *Journal of Parallel and Distributed Computing*, 38(2):237–247, 1996.
6. M. Leair, D. Miles, V. Schuster, and M. Wolfe, *Euro-Par99 Parallel Processing 5th International Euro-Par Conference*, Toulouse, France, Aug. 31–Sept. 3, 1999, Proceedings, Springer-Verlag LNCS 1999.
7. S. Lee, H. Yook, M. Koo, and M. Park, Processor reordering algorithms toward efficient GEN_BLOCK redistribution. *Proceedings of the 2001 ACM Symposium on Applied Computing*, Las Vegas, Nevada, USA, 2001, pp. 539–543.
8. Y. W. Lim, P. B. Bhat, and V. Prasanna. Efficient algorithms for block-cyclic redistribution of arrays. *IEEE Symposium on Parallel and Distributed Processing*, Oct. 1996.
9. Y. Pan and J. Shang. Efficient and scalable parallelization of time-dependent Maxwell equations solver using high performance Fortran, *The 4th IEEE International Conference on Algorithms & Architectures for Parallel Processing,* Hong Kong, Dec. 11–13, 2000, pp. 520–531.
10. N. Park, V. K. Prasanna, and C. S. Raghavendra. Efficient algorithms for block-cyclic array redistribution between processor sets. *IEEE Transactions on Parallel and Distributed Systems*, 10(12):1217–1239, 1999.

11. PGHPF, a High Performance Fortran compiler, http://www.pgroup.com/products/pghpfindex.htm.
12. S. Ramaswamy, B. Simons, and P. Banerjee. Optimizations for efficient array redistribution on distributed memory multicomputers. *Journal of Parallel and Distributed Computing*, 38:217–228, 1996.
13. R. Thakur, A. Choudhary, and G. Fox. Runtime array redistribution in HPF programs. *Proceedings Scalable High Performance Computing Conference*, May 1994, pp. 309−316.
14. H. Yook and M. Park. Scheduling GEN_BLOCK array redistribution, *Proceedings of the IASTED International Conference Parallel and Distributed Computing and Systems*, Nov. 3–6, 1999, MIT, Boston, USA.