PAPER *Special Section on Parallel/Distributed Computing and Networking*

Message Scheduling for Irregular Data Redistribution in Parallelizing Compilers

Hui WANG†a), ***Nonmember***, **Minyi GUO**†, ***Member***, ***and*** **Daming WEI**†, ***Nonmember***

SUMMARY In parallelizing compilers on distributed memory systems, distributions of irregular sized array blocks are provided for load balancing and irregular problems. The irregular data redistribution is different from the regular block-cyclic redistribution. This paper is devoted to scheduling message for irregular data redistribution that attempt to obtain suboptimal solutions while satisfying the minimal communication costs condition and the minimal step condition. Based on the list scheduling, an efficient algorithm is developed and its experimental results are compared with previous algorithms. The improved list algorithm provides more chance for conflict messages in its relocation phase, since it allocates conflict messages through methods used in a divide-and-conquer algorithm and a relocation algorithm proposed previously. The method of selecting the smallest relocation cost guarantees that the improved list algorithm is more efficient than the other two in average.

key words: *parallel computing, parallel compiler, GEN_BLOCK, data redistribution, list algorithm*

1. Introduction

Parallelizing compiler, the core of automatic data parallel programming, is a recent approach to overcome the difficulty of programming in using message passing library, and of designing and writing a large number of different interacting tasks to solve a single problem. In parallel computing, many applications need data redistribution during computation since different phases require different data mappings in order to execute loops efficiently without data dependencies [1]. Data distribution in High Performance FORTRAN (HPF) specifies how a data array is to be distributed over a processor set. Data distribution, in general, can be classified into two categories, regular and irregular. The regular array distributions like BLOCK, CYCLIC, and CYCLIC(n) are suitable for applications that show regular data access patterns, whereas irregular (or extensible) distributions such as GEN_BLOCK and INDIRECT are provided for load balancing and irregular problems.

As a generalized block distribution, GEN_ BLOCK allows unequal-sized consecutive segments of an array to be mapped onto consecutive processors [2]. 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. An efficient algorithm is needed to relocate the elements of the array among different processors and to perform the necessary communication among different processors.

Example

```
   PARAMETER (S=/7,30,25,0,13,25/)
!HPF$ PROCESSORS P(6)
      REAL A(100), B(200), new(6)
!HPF$ DISTRIBUTE A(GEN_BLOCK(S)) ONTO P
!HPF$ DYNAMIC B
      new=/9,13,35,27,2,14/
!HPF$ REDISTRIBUTE A(GEN_BLOCK(new))
```

Above is an example of GEN_BLOCK and the data redistribution in HPF version 2. Given the above specification, array elements A(1:7) are mapped on P_1, A(8:37) are mapped on P_2, A(38:62) are mapped on P_3, no elements are mapped on P_4, A(63:75) are mapped on P_5, and A(76:100) are mapped on P_6. Then A is redistributed based on the *new* array.

Data redistribution process takes two steps to complete a communication. First, the array to be redistributed should be efficiently scanned or processed, and all the messages are ready to be exchanged between processors. Then, all messages should be efficiently scheduled so as to minimize communication overhead. Finally, the message is redistributed among processors. Compared with the message redistribution process, message generation process requires less time and also the time is fixed in most cases, so it is quite important to develop efficient and practical communication scheduling algorithms.

The communication between processors can be either blocking or non-blocking. A non-blocking scheduling algorithm avoids excessive synchronization overhead. So in general, it is faster than a blocking scheduling algorithm. However, a non-blocking scheduling algorithm is much expensive since the non-blocking communication primitives need as much buffering as the data being redistributed [3]. Only blocking communication primitives are used throughout this paper.

To obtain the optimal scheduling solution to the irregular data redistribution is an NP complete problem since we have implemented the irregular data redistribution as a weighted interval graph in Ref. [4]. In this paper, an efficient heuristic algorithm for irregular block redistribution is

Manuscript received April 2, 2005.
Manuscript revised August 15, 2005.

†The authors are with the School of Computer Science and Engineering, University of Aizu, Aizu-Wakamatsu-shi, 965–8580 Japan.
a) E-mail: hwang@u-aizu.ac.jp

DOI: 10.1093/ietisy/e89–d.2.418

developed and compared to two previously developed algorithms, in order to obtain suboptimal scheduling while satisfying the minimal size of total steps condition and the minimal step condition [5].

This paper is organized as follows. Section 2 shows the irregular block redistribution model and its implementation. In Sect. 3, the list algorithm is presented in details. The results are compared and discussed in Sect. 4. Section 5 introduces related work in both regular and irregular block redistributions followed by conclusions in the last section.

2. GEN_BLOCK Redistribution Model

Before presenting GEN_BLOCK redistribution model, we give the definition of GEN_BLOCK data redistribution first. A data redistribution **R** is a set of routines that transfer all elements in a set of source processors *SP* to a set of target processors *TP*. The matrix implementation is used in [5] due to its easy to make the inter-processor communication understandable. A message sent from the *i*th source processor SP_i to the *j*th target processor TP_j is regarded as an element $m_{i,j}$ of a redistribution table **R**. The value of $m_{i,j}$ represents the length of the communication message.

Different from a regular data redistribution, a GEN_BLOCK data redistribution does not have a cyclic message passing pattern. Besides this, if one source processor P_i sends messages to two target processors P_{j-1} and P_{j+1}, therefore it must send a message to the target processor P_j (*i* and *j* are the indices of processors), and vice versa. So to design scheduling algorithms, we concentrate on organizing the order of each message and minimizing the cost for GEN_BLOCK data redistribution.

The output of a GEN_BLOCK data redistribution algorithm is the scheduling table **S** used for the redistribution processes. Without loss of generality, the scheduling table **S** can be defined as

$$\begin{pmatrix} s_{00} & s_{01} & s_{02} & \cdots & s_{0m} \\ s_{10} & s_{11} & s_{12} & \cdots & s_{1m} \\ \vdots & \vdots & \vdots & & \\ s_{i0} & s_{i1} & s_{i2} & & \\ \vdots & \vdots & \vdots & \ddots & \\ s_{j0} & s_{j1} & s_{j2} & & \\ \vdots & \vdots & \vdots & & \\ s_{n0} & s_{n1} & s_{n2} & \cdots & s_{nm} \end{pmatrix} \tag{1}$$

Where the rows are the time steps, for example, s_{00}, s_{01}, and s_{0i} are scheduled in the time step S_0, and s_{i0}, s_{i1}, and s_{ij} (*i* and *j* are arbitrary index numbers that satisfy $s_{ij} \neq \emptyset$) are scheduled in the time step S_i, respectively. *n* is the total time step, and *m* is the maximum number of messages scheduled in one time step, respectively. An element s_{ij} in scheduling table **S** presents a message to be scheduled in the time step S_i. Since we assume that $s_{ij} \geq s_{(i+1)j}$ and $s_{ij} \geq s_{i(j+1)}$ in the scheduling table **S**, s_{ij} is the *j*th largest value in time step S_i. For example, $s_{00} \geq s_{01}$ and $s_{00} \geq s_{10}$, and s_{00} is the message with maximum value in the scheduling table **S**.

The aims for GEN_BLOCK redistribution are:

(1) To minimal number of communication steps. Clearly, the minimum number of communication steps is lower-bounded by larger of the maximum fan-out and maximum fan-in of the processor nodes. That means, to minimal the total time step *n* of the scheduling table **S**.

(2) To minimal communication time of the communication Schedule. The total communication time is proportional to the sum of the size of each communication step. In order to reduce the total communication time, one way is to minimize the size of communication schedule as opposed to minimize the number of communication steps. The numbers of fan-in and fan-out of a processor are the numbers of processors the processor receives and sends a message, respectively. That means, to minimal the sum of the first column of the scheduling table **S**, $a_{00}, a_{10}, \ldots, a_{n0}$.

(3) To avoid conflicts. Since one processor can not send/receive two messages at the same time, it is important to efficiently redistribute data without any conflicts.

Before we discuss how to minimize the size of the communication schedule, we introduce some useful definitions [5].

Definition 1. (Neighboring Message Set) The messages (more than one message, absolutely) sent from same processor or received by same processor are in same neighboring message set (NMS). In a redistribution table, an NMS is a set of elements in the same row or same column. An NMS can be labelled accordingly as NMS$_k$, ($k = 1, 2, 3, \ldots$), from left-up side to right-down side in a data redistribution table.

Theorem 1. The minimal number of communication steps is the maximum number of elements of an NMS. Proof. If we have the minimal number of communication steps *m* in a data redistribution, and there is an NMS which has $m + 1$ elements, there must be one message can not be scheduled.

Definition 2. (Link Message) The message which belongs to two different neighbor message sets, NMS$_k \cap$NMS$_{k'}$ ($k \neq k'$), is a link message. Each neighbor message set may have at most two *link messages*.

Therefore, the GEN_BLOCK redistribution problem has the following properties:

(1) The total number of the neighbors of an element $m_{i,j}$, $B_{left} + B_{right} + B_{up} + B_{down} \leq 2$. Here *B* is the Boolean value to judge whether there is a neighbor or not. B_{left}, B_{right}, B_{up}, and B_{down} are the values related to its neighbor in left, right, up, and down directions, respectively. If $m_{i,j}$ has a neighbor message in one direction, *B* in that direction is equals to 1, otherwise it is equals to 0. It also indicates that one message can belong to at most two different NMSs, as we have described above.

(2) In a data 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.

(3) Elements in the same row or the same column of a data redistribution table can not be scheduled in the same time step. Clearly, it is because elements in the same row or column represent messages sent by the same processor or received by the same processor.

3. GEN_BLOCK Redistribution Algorithms

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

Definition 3. (Conflict) There is a conflict if two messages in the same NMS are scheduled into the same time step S.

In Ref. [5], we proposed a divide-and-conquer redistribution algorithm to improve the efficiency of communication among source processors and target processors. The divide-and-conquer algorithm can generate suboptimal communication scheduling tables. However, as we have discussed in Ref. [5], the disadvantage in the divide-and-conquer algorithm is that many *bad* link messages (see its definition in [5]) can be generated during the merging process. We suggested to include the list scheduling in developing new efficient redistribution algorithms. So in this paper, we develop new algorithms based on the list scheduling.

The list scheduling is the process of sorting all communication messages in descending order. The messages in descending order are saved in an array A. According to the message orders in A, communication messages are allocated one by one accordingly. Sorting messages in descending order is to guarantee that the larger size message can be allocated earlier.

At the beginning of the list scheduling, the maximum size message is assigned to the time step S_0. Next, the second maximum size message is assigned to the time step S_0. If there is a conflict between the second maximum size message and the maximum size message message, or say, both two messages belong to one NMS$_k$ (k can be any number), the second maximum size message will be reassigned to time step S_1. Suppose there are n time step $S_0, S_1, \ldots, S_{(n-1)}$, where n is the element number of NMS$_k$, which has the maximum number of elements. To assign a message $m_{i,j}$ to any time step S_l, the following problems must be clarified:

1) In which time step should the message $m_{i,j}$ be allocated? Since all already assigned messages have larger size than $m_{i,j}$, it is possible to allocate $m_{i,j}$ in any time step S_l if no conflicts between $m_{i,j}$ and all already assigned messages in S_l. Here we suppose $m_{i,j}$ will be allocated in the first order of time step S_l.

2) What is the order of time step S to assign $m_{i,j}$? Assign $m_{i,j}$ in the order of S_l with l from 1 to 2, . . ., or l from n to n-1, . . ., or pick up l randomly? If no conflicts between messages, the results of assigning $m_{i,j}$ in the order of S_l with l from 1 to 2, . . ., are equal to these of assigning $m_{i,j}$ in the order of S_l with l from n to $n-1, \ldots$. However, it is difficult to say when there are conflicts. Here we suppose that assigning $m_{i,j}$ in the order of S_l with l from 1 to 2, ... If there is a conflict between $m_{i,j}$ and any message in S_l, $m_{i,j}$ will be assigned to next time step S_{l+1} without violating the minimal number of communication steps.

3) How to relocate message $m_{i,j}$ if there are conflicts? In order to satisfy the minimal number of communication steps, which has been described in Sect. 2, the maximum time step is n, the maximum size of NMS$_k$ ($k = 1, 2, \ldots$). So it is possible that one message has conflicts with any other message in each time step S_l.

List algorithm involves two phases:

1. List-scheduling phase: Sorting all messages in descending order. And then, assign message $m_{i,j}$ into S_0. If a conflict exists in the time step S_0, it shall move to next step without violating the minimal step condition. If a message can't be allocated, it will be discarded. The discarded messages will be allocated in the relocation phase after finishing the list scheduling phase.

2. Relocation phase: Relocating discarded messages. The matrix is divided into three parts according to the position of the discarded message: middle, left-hand side, and right-hand side. The middle part includes both the column and the row close to the discarded message. The costs for each part are calculated and compared. The smallest cost method is chosen to complete the relocation phase.

The list-scheduling phase of the list algorithm is the following:

```
List Algorithm
    Inputs: R
    Outputs: S
    1. S = $\emptyset$.
    2. A $\Leftarrow$ sorting $m_{ij} \in$ R in descending order;
       $N_A$=length.$A$ .
    3. $s_{00} \Leftarrow a_0, (a_0 \in$ A, $s_{00} \in$ S).
    4. for (i=1; i<N; i++) {
         m=0;
         j=0;
         while ($s_{mj} \neq \emptyset$) {
             if ($s_{mj}, a_i \in$ same NMS) {
                 m++;
                 j=0;
                 continue;
             }
             j++;
         }
         $s_{jk} \Leftarrow a_m$;
       }
    (To be continued)
```

Definition 4. (Discarded Message) After the list-scheduling phase of the list algorithm, if total time step of **S** is larger than the minimal number of communication steps N_S, a message located in time step $S_k (k \geq N_S)$ is a discarded message.

Theorem 2. A discarded message must be a link message.

If a message m is not a link message, it belongs to only one NMS, say NMS$_j$. The element number of NMS$_j \leq N_S$. So we can find a position for m inside N_S time steps, absolutely.

Lemma 1. A discarded message belongs to two NMSs, and both of them have N_S element number.

A link message belongs to two NMSs, so a discarded message belongs to two NMSs. If any of NMSs have the element number $\leq N_S$, the discarded message can be filled inside N_S time steps.

Lemma 2. A discarded message is the shortest message of the NMSs it belongs.

If both a discarded message m_j and a message m_k ($m_j > m_k$) belongs to same NMS, m_j should be filled into a time step $< N_S$ since it has priority to be filled than m_k. So m_j is not a discarded message. This conflicts with the precondition.

Lemma 3. All discarded messages are filled in the (N_S + 1)th time step only.

If a discarded message m_j has conflicts with one discarded message m_k, which has already been filled in (N_S + 1)th time step, m_j and m_k belong to the same NMS. Since a discarded message is the latest message to be filled, m_j can not be a discarded message. So any two discarded messages can't conflict with each other.

In **S**, there are only two rows, say S_i and S_j, that each has only one message, say s_{ik} and s_{jl}, which belongs to the same NMS as the discarded message. Other rows, each has two. It can be concluded from Lemma 1.

If no discarded message exists after the list-scheduling phase of list algorithm, **S** is a completed solution. But if there are any discarded messages, we must reassign these discarded messages from (N_S + 1)th time step to any of steps from 1st to (N_S)th in order to keep the minimal number of communication steps condition. Suppose a discarded message m_n belongs to two NMSs, where NMS$_i$ has { m_0,m_1,..., m_k }, and NMS$_j$ has {n_0, n_1, ..., m_k }, according to Lemma 1 (link messages are boxed for easily identifying). We can distinguish all situations into three different situations (see the relocation phase of list algorithm below), and select the situation with minimum additional cost.

The relocation phase of the list algorithm is the following:

(Continue the list algorithm)

```
5. n=0;
   while ($s_{(Ns+1)n} \neq \emptyset$) {
      $s_{(Ns+1)n} \in$ NMS$_a$, $s_{(Ns+1)n} \in$ NMS$_b$,
      finding $s_ik \in$ NMS$_a$, $\exists s_ik', k' \neq k$ not $\in$ NMS$_a$;
      finding $s_jl \in$ NMS$_b$, $\exists s_jl', l' \neq l$ not $\in$ NMS$_b$;
      $\triangle 1 = s'_{j0} + s'_{i0} - s_{j0} - s_{i0}$:
      $s_{jj'} \Leftarrow s_{ik}$
      $s_{ik} \Leftarrow s_{(Ns+1)n}$
      while (conflict) {
         if $s_{jj'}$ conflicts with $s_{jj''}$, move $s_{jj''}$ to $s_{ii'}$;
         if $s_{ii'}$ conflicts with $s_{ii''}$, move $s_{ii''}$ to $s_{jj'''}$;
      }
      $\triangle 2 = s'_{j0} + s'_{i0} - s_{j0} - s_{i0}$:
      $s_{ii'} \Leftarrow s_{jl}$
      $s_{jl} \Leftarrow s_{(Ns+1)n}$
      while (conflict)
         if $s_{ii'}$ conflicts with $s_{ii''}$, move $s_{ii''}$ to $s_{jj'}$;
         if $s_{jj'}$ conflicts with $s_{jj''}$, move $s_{jj''}$ to $s_{iii'''}$;
      }
      $\triangle 3 = \sum S'_i - \sum S_i$:
      {$\boxed{s_{ii'}}$,$s_{jj'}$, . . . ,$\boxed{m_k}$}=NMS$_a$;
         // a link message is boxed
      {$\boxed{s_{kk'}}$,$s_{ll'}$, . . . ,$\boxed{m_k}$}=NMS$_b$ ;
      sorting $s_{jj'}$, . . . , $\boxed{m_k}$, $s_{ll'}$, . . . ,;
      assigning $s_{jj'}$, . . . , $\boxed{m_k}$, $s_{ll'}$, . . . to time step
         from $s_0$ to $s_{Ns}$, but neither $S_i$ nor $S_k$;
      selecting minimum($\triangle 1$, $\triangle 2$, $\triangle 3$);
      i++;
   }
```

(End of the list algorithm)

Clearly, there are differences between this algorithm and a relocation algorithm developed previously in Ref. [6]: (1) In the list-scheduling phase, the list algorithm discards the conflict message and continuously allocates the next message. But in the relocation algorithm, it may change from the first phase to the second phase when the conflict message appears; (2) In the relocation phase, we will allocate the discarded messages through relocation methods used in the divide-and-conquer algorithm and the relocation algorithm. The method that requires the smallest relocation cost will be selected to process the relocation.

From the algorithm shown above, we can estimate the time complexity of the list algorithm is $O(n^2 log_2 n)$ because its list scheduling phase needs $O(n^2)$, while its relocation phase requires $O(n^2 log_2 n)$ in the worst cases.

4. Results and Discussion

The simulation model and the scheduling algorithms considered were implemented under SGI Origin 2000 system with 24 processors. 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 among source processors and target processors, we also suppose the total size of messages in both source processors and target processors are equal.

Inputs of the program are two arrays: the source processors set and the target processors set. According to these arrays, a matrix which represents the messages can be generated. Then the algorithm presents a scheduling table, which contains the time step for each message.

In our experiments, we first estimate the effects of the up-bound of the array size on the experimental results of message scheduling. Normally, a random number is generated according to the up-bound of the array size. In Figs. 1 and 2, we show the event percentage as a function of the

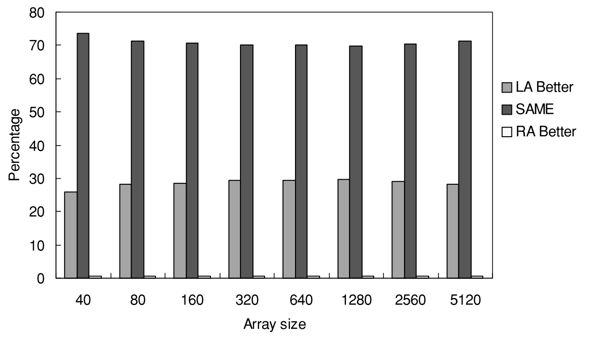


Fig. 1 Event percentages of the list algorithm and the relocation algorithm as a function of the array size.

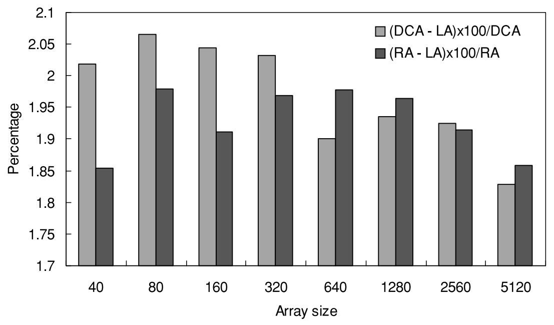


Fig. 3 The normalized difference percentages of average communication times of the list algorithm and the divide-and-conquer algorithm.

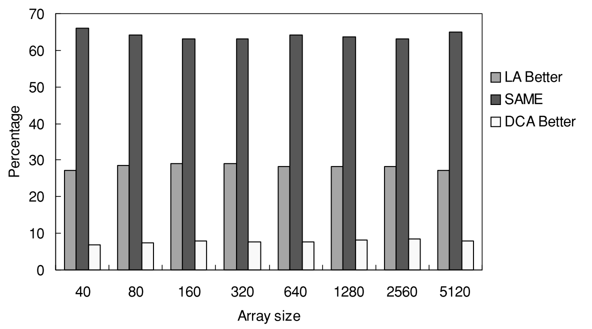


Fig. 2 Event percentages of the list algorithm and the divide-and-conquer algorithm as a function of the array size.

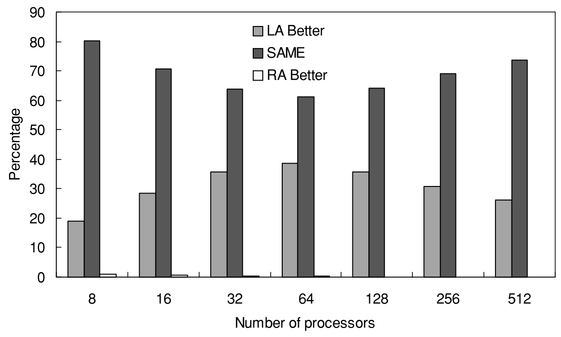


Fig. 4 Event percentages of the list algorithm and the relocation algorithm as a function of the number of processors.

the array size in histogram. The simulation results in both Figs. 1 and 2 are generated with 16 processors as an example. The total number of generated events is 100000 for each array size. For each different array size, 3 weighted histograms are compared. In Fig. 1, "LA Better" represents the percentage of the number of events that the list algorithm has the lower total communication time than the relocation algorithm, while "RA Better" gives the reverse situation. If both algorithms have same total communication time, events are collected into "SAME" cylinder. Results shown in Fig. 1 indicates that the array size does not play an important role in distinguishing the list algorithm and the relocation algorithm since results are very similar for different array sizes. In Fig. 2, "LA Better" represents the percentage of the number of events that the list algorithm has the lower total communication time than the divide-and-conquer algorithm, while "DCA Better" gives the reverse situation. If both algorithms have same total communication time, events are collected into "SAME" cylinder. From Fig. 2, we can see the array sizes from 40 to 5120 give quite similar results. Generally, the array size of 160 is selected for generating redistribution events in the rest of this paper. Before that, Fig. 3 is plotted to validate whether it is reasonable or not to use the array size of 160. In Fig. 3, the normalized difference percentages of average communication times of the list algorithm and the relocation algorithm, $(RA - LA) \times 100/RA$, and these of the list algorithm and the divide-and-conquer algorithm, $(DCA - LA) \times 100/DCA$ are shown as a function of the array size from 40 to 5120. Results show that both the divide-and-conquer algorithm and the relocation algorithm require more time than the list algorithm. Similar to the conclusion derived from both Figs. 1 and 2, results with different array sizes from 40 to 5120 are very close. So it is not a bad idea to choose 160 as the array size in the following experiments.

Figure 4 shows the comparison among the scheduling results of the list algorithm and the relocation algorithm. The results are divided into three tuples: "LA Better" (the list algorithm has better performance than the relocation algorithm), "RA Better" (the relocation algorithm has better performance than the list algorithm), and "SAME" (same performance). In average, 30% results of the list algorithm have better performance than these of the relocation algorithm, while around 70% results have same performance. Events with "LA Better" are difficult to find in Fig. 4. It indicates that list algorithm has better performance than the relocation algorithm. The comparison among scheduling results of the list algorithm and the divide-and-conquer algorithm also shows that the list algorithm has better performance than the divide-and-conquer algorithm in average.

The comparison among the scheduling results of the list algorithm and the divided-and-conquer algorithm is shown in Fig. 5. Figure 5 shows that 40% results of the list algorithm have better performance than these of the divided-and-conquer algorithm in average, while 50% results of the list algorithm have the same performance as these of

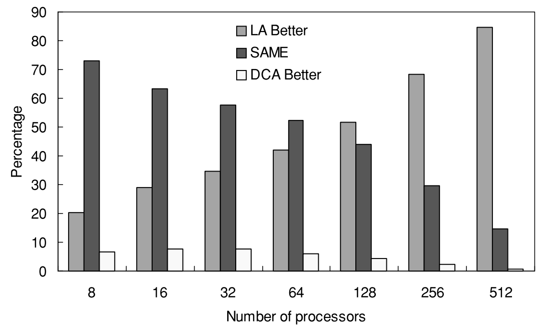


Fig. 5 Event percentages of the list algorithm and the divide-and-conquer algorithm as a function of the number of processors.

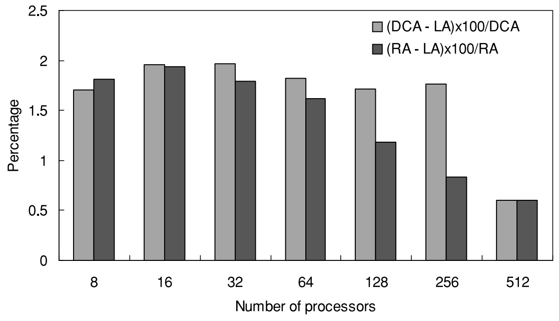


Fig. 6 The normalized difference percentages of average communication times of the list algorithm (LA) and the relocation algorithm (RA).

the divided-and-conquer algorithm in average. Events with "DCA Better" are less than 10% in Fig. 5. It indicates that the list algorithm has better performance than the divide-and-conquer algorithm. From the results, we can find that the list algorithm is the best one among those scheduling algorithms.

We also compare the normalized difference percentages of average communication times of the list algorithm and the relocation algorithm, $(RA - LA) \times 100/RA$, and these of the list algorithm and the divide-and-conquer algorithm, $(DCA - LA) \times 100/DCA$, as a function of the number of processors in Fig. 6. Obviously, both the divide-and-conquer algorithm and the relocation algorithm require more time than the list algorithm. The differences among results are between 1 and 2. It means that the list algorithm really improve the performance of GEN_BLOCK data redistribution.

5. Related Work in Array Redistribution

Recent work on array redistribution has been divided into two categories: regular and irregular redistributions. A non-blocking communication algorithm presented by Thakur [3] can perform the computation and communication in parallel which makes it more efficient than blocking communication. A redistribution technique based on descriptors called pitfalls has been presented in Ref. [7].

Kalns and Ni [8] presented a mapping technique which can map data to logical processor to minimize the total amount of communication data. Algorithm from Desprez [9] concentrates on organizing the message exchanges into structured communication steps to minimize contention. Guo [10] developed techniques to reduce overheads in both index computation and inter-node communication. Guo [11] also presented an efficient index computation method to generate a schedule that minimizes the number of communication steps and eliminates node contention in each communication step.

There are relatively few papers on irregular array redistribution. Leair [12] have implemented the GEN_BLOCK data distribution in PGHPF, a High Performance Fortran compiler [13]. Simple benchmark results show that the GEN_BLOCK distribution increasing the speed up to twice over regular distribution. Yook and Park [6] proposed an algorithm for the redistribution of one-dimensional arrays in GEN_BLOCK. Their algorithm exploits a spatial locality in message passing from seemingly irregular array redistribution. Lee [14] focused on reducing the communication cost in GEN_BLOCK redistribution using a logical processor reordering method.

A divide-and-conquer algorithm was developed previously [5]. The main routines of our divide-and-conquer algorithm includes: (1) separating the matrix into the smallest unit, a neighbor message set (NMS); (2) grouping each pair of NMSs, $\{NMS_1, NMS_2\}, \cdots, \{NMS_{2i+1}, NMS_{2i+2}\}, \cdots$. The separating and grouping processes are quite simple. All separations are taken place at *link* messages. It is reasonable since the link message is belong to both related row and related column, which makes conflicts possible happen. After obtaining the pairs of NMSs, we begin the merging processes recursively to produce the scheduling. The generating scheduling table processes are quite same for each group in any phases. First, we detect the conflict information, and kick out the link message and those messages scheduled with it. Then sort remain messages to make an optimal solution. If a message has more possible positions in the scheduling table, it stands with its closest ceiling. Finally, the link message and those related messages are included and added into the scheduling table. The divide-and-conquer algorithm provides comparable performance to the relocation algorithm.

6. Conclusion

Recent work on GEN_BLOCK redistribution is quite little. In this paper, algorithms for GEN_BLOCK redistribution are developed to obtain near optimal scheduling while satisfying the minimal communication costs of total steps condition and/or the minimal step condition.

Since both the divide-and-conquer algorithm and the relocation algorithm have advantages and disadvantages, we further develop new efficient algorithms based on the list-scheduling. In the list-scheduling phase of a list algorithm, we discard the conflict messages and complete the list-scheduling first. Then we allocate discarded conflict messages through relocation methods used in a divide-and-conquer algorithm and a relocation algorithm developed previously. Method with the smallest relocation cost is selected to guarantee the enhanced algorithm is more efficient than others in average. The list algorithm provides more chance for conflict messages than the relocation algorithm in the list-scheduling phase. We compared the experimental results among these algorithms. Among them, the list algorithm provides more efficient solution than others.

References

[1] Y. Pan and J. Shang, "Efficient and scalable parallelization of time-dependent Maxwell equations solver using high performance Fortran," 4th IEEE International Conference on Algorithms & Architectures for Parallel Processing, pp.520–531, Hong Kong, Dec. 2000.

[2] High Performance Fortran Forum, High Performance Fortran Language Specification version 2.0, Rice University, Houston, Texas, Jan. 1997.

[3] R. Thakur, A. Choudhary, and G. Fox, "Runtime array redistribution in HPF programs," Proc. Scalable High Performance Computing Conference, pp.309–316, May 1994.

[4] H. Wang, Algorithms for Irregular Redistributions in HPF-2, Georgia State University Thesis, Atlanta, Georgia, USA, Aug. 2002, .

[5] H. Wang, M. Guo, and D. Wei, "A divide-and-conquer algorithm for irregular redistributions in parallelizing compilers," J. Supercomputing, vol.29, no.2, pp.157–170, Aug. 2004.

[6] H. Yook and M. Park, "Scheduling GEN_ BLOCK array redistribution," Proc. IASTED International Conference Parallel and Distributed Computing and Systems, MIT, Boston, USA, Nov. 1999.

[7] S. Ramaswamy, B. Simons, and P. Banerjee, "Optimizations for efficient array redistribution on distributed memory multicomputers," J. Parallel Distrib. Comput., vol.38, pp.217–228, 1996.

[8] E.T. Kalns and L.M. Ni, "Processor mapping techniques toward efficient data redistribution," IEEE Trans. Parallel Distrib. Syst., vol.6, no.12, pp.1234–1247, 1995.

[9] F. Desprez, J. Dongarra, A. Petitet, C. Randriamaro, and Y. Robert, "Scheduling block-cyclic array redistribution," IEEE Trans. Parallel Distrib. Syst., vol.9, no.2, pp.192–205, 1998.

[10] M. Guo and I. Nakata, "A framework for efficient data redistribution on distributed memory multicomputers," J. Supercomputing, vol.20, no.3, pp.243–265, Nov. 2001.

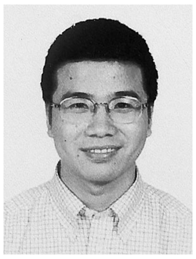
[11] M. Guo, I. Nakata, and Y. Yamashita, "Contention-free communication scheduling for array redistribution," Proc. International Conference on Parallel and Distributed Systems, pp.658–667, Dec. 1998.

[12] M. Leair, D. Miles, V. Schuster, and M. Wolfe, "Flexible data distribution in PGHPF," Euro-Par99 Parallel Processing 5th International Euro-Par Conference, Proc., Springer-Verlag, LNCS, Toulouse, France, Aug./Sept. 1999.

[13] PGHPF: A High Performance Fortran compiler, http:// www.pgroup.com/ products/ pghpfindex.htm

[14] S. Lee, H. Yook, M. Koo, and M. Park, "Logical processor reordering toward efficient GEN_BLOCK redistribution," 4th Annual HPF User Group meeting, Tokyo, Japan, Oct. 2000.

[15] S. Lee, H. Yook, M. Koo, and M. Park, "Processor reordering algorithms toward efficient GEN_BLOCK redistribution," Proc. 2001 ACM symposium on Applied computing, pp.539–543, Las Vegas, Nevada, USA, 2001.



Hui Wang is currently a researcher at the Department of Computer Software, The University of Aizu, Japan. His research interests include parallel and distributed computing, cluster and grid computing, high-performance computing and simulation.

Minyi Guo is currently an associate professor at the Department of Computer Software, The University of Aizu, Japan. From 2001 to 2005, he was a visiting professor of Georgia State University, USA, Hong Kong Polytechnic University, University of New South Wales, Australia, and National Sun Yet-sen University, Taiwan. Dr. Guo has served as general chair, program committee or organizing committee chair for many international conferences. He is the editor-in-chief of the Journal of Embedded Systems. He is also in editorial board of International Journal of High Performance Computing and Networking, Journal of Embedded Computing, Journal of Parallel and Distributed Scientific and Engineering Computing, and International Journal of Computer and Applications. He is also the author of books "New Horizons of Parallel and Distributed Computing" and "High Performance Computing: Paradigm and Infrastructure". Dr. Guo's research interests include parallel and distributed processing, parallelizing compilers, data parallel languages, pervasive computing and embedded systems, molecular computing and software engineering. He is a member of the ACM, IEEE, and IEEE Computer Society.

Daming Wei graduated from Department of Mathematics and Mechanics, Tsinghua University, Beijing, China. He received the M. Eng. degree in Computer Engineering from Shanghai Institute of Technology (Shanghai University), and Ph.D. in Biomedical Engineering from Zhejiang University, China. He was a deputy director of the Biomedical Engineering Section in Zhejiang University before joined Tokyo Institute of Technology in 1986. Since then, he has been with industry and universities in Japan. He is currently a professor at faculty of Computer Science and Engineering and a director of Department of Computer Software, the University of Aizu, Fukushima, Japan. Prof. Wei is well-known for his achievements in developing a state-of-the-art computer heart model and computer simulation of electrocardiogram. Recent directions in his research group include biomedical modeling and computer simulation, and visualization and virtual environment, biomedical informatics, e-health and mobile healthcare. He is leading a number of large-scale research projects funded by Japan Ministry of Education, Science and Technology, Japan Ministry of Economy and Industry, and other domestic organizations. He serves as a council member of the International Society of Bioelectromagnetism. He is editor of the International Journal of Bioelectromegnetism, the International Journal of Bioinformatics. He is a council member of Japan Biomedical Engineering Society Tohoku Branch. He is guest professor at several universities in Japan and China.