Code generation for accurate array redistribution on automatic distributed-memory parallelization

Bo Zhao
State Key Laboratory of Mathematical Engineering and Advanced Computing
Zhengzhou, Henan, China
E-mail: zhaobo197359@gmail.com

Rui Ding
State Key Laboratory of Mathematical Engineering and Advanced Computing
Zhengzhou, Henan, China
E-mail: dr2012earth@gmail.com

Lin Han
State Key Laboratory of Mathematical Engineering and Advanced Computing
Zhengzhou, Henan, China

Jinlong Xu
State Key Laboratory of Mathematical Engineering and Advanced Computing
Zhengzhou, Henan, China
E-mail: longkaizh@126.com

Received 19 February 2013

Accepted 19 November 2013

Abstract

Code generation belongs to the backend of parallelizing compiler, and is for generating efficient computation and communication code for the target parallel computing system. Traditional research resolve array redistribution mainly by generating communication code that each processor sends all data defined in its local memory to all processors, but this will bring large amount of communication redundancy, which increase with the growth of number of processors. Focusing on this problem, this paper presents an accurate code generation algorithm of array redistribution for distributed-memory architecture. The algorithm determines source processor and goals processor of each array element's migration in array redistribution by the transformation of data decompositions, then generate accurate communication code. The experimental results show that algorithm proposed by this paper can effectively reduce communication redundancy with the processor scale growth, and improve the parallel performance of applications.

Keywords: automatic parallelization; data decomposition; array distribution; communication code generation;

1. Introduction

Every computing node has its own memory on distributed-memory parallelizing computers, and needs explicit message passing to exchange data between nodes. This means that automatic parallelization not only have to partition computation and data onto each computing node, but also have to generate communication code to keep data consistency. Since local memory access speed of computing node is much faster than remote memory access, the efficiency of communication code has a direct impact on performance of parallel programs. Code generation belongs to backend of parallelizing compiler, and its task is based on program's parallelism analysis result of frontend to generate suitable parallel code for execution on target parallel system.

For the communication code generation on automatic distributed-memory parallelization, people have done a lot of research. Ancourt and Irigoin use a series of projections of Fourier-Motzkin elimination (FME) to generate loop nests after loop transformation[1]. Amarasinghe and Lam represent data decompositions, computation decompositions and the data flow information all as systems of linear inequalities, and showed that the problems of communication code generation and communication optimization can all be solved by projecting polyhedra represented by sets of inequalities onto lower dimensional spaces [2]. Based on the theories of Ref.3, Ferner built an open source parallelizing compiler *Paraguin* to generate message-passing code for distributed-memory computer systems[3]. In addition, to reduce the number of inter-processor messages, he extended these algorithms to incorporate the mapping of virtual processors to physical processors[4]. In Refs. 5 and 6, Martin proposes a method on suppressing independent loops in packing/unpacking loop nest to reduce message size for message-passing code. In Ref 7, Griebl provides a discussion on distributed-memory automatic parallelization using the polyhedral framework. In Ref 8, Classen et al. construct communication polytopes for each flow-dependence to complete distributed memory code generation scheme of Ref 7, though with very limited implementation and experimental evaluation. Bondhugula reported an end-to-end automatic distributed-memory parallelization and code generation framework on Ref 9, and presents techniques for optimizing communication code, such as the communication set is not sent to processors that do not need any value from this communication set.

When handling pipeline communication of intra-loop, the above studies can generate accurate communication code by using the result of dependence testing. However, the dependence test mainly is data flow analysis within loop, so this method is not suitable to generate accurate communication for inter-loops data exchange.

Works that translate OpenMP to MPI address a subset of problems of communication code auto-generation. In Refs. 10, Kwon et al. introduce a hybrid compiler-runtime translation system, which analyses accurate array access section on runtime and generate communication code in communication point.

As a matter of fact, a great many large-scale scientific computing applications contain irregular problems and it is necessary to discuss the data decomposition and code generation for irregular instances. For example, when dealing with the programs with sparse matrix, the index of the data array should be implemented through other array's value and such indirect index leads to the data access mode greatly irregular. This subscript expression relies on variable or non-affine functions so we can only confirm the data accesses under data access mode are irregular array references or not. Basumallik and Eigenmann propose techniques that create inspectors to analyze actual data access patterns for irregular accesses at runtime, and enable computation-communication overlap by restructuring irregular parallel loops[11]. In Refs 12 and 13, Ravishankar et al. propose a code generation approach for effective parallel execution of a class of irregular loop computations in a distributed-memory environment, using a combination of static and runtime analysis and generating inspector/executor (I/E) code. The inspector captures the data-dependent behavior of the computation in parallel and the executor performs the computation in parallel. In Ref 14, Kim et al. present automatic pipelined parallelization for distributed memory with speculation.

Characteristics of above studies are generating communication code on runtime, although it is possible to make an accurate judgment on the flow of array element, the cost of run-time analysis will eventually be passed on to the parallel performance of the program.

If array has different distribution strategies between loops, then array redistribution must be brought to ensure processors reference data in its local memory. Array redistribution is the most common inter-loops communication in parallel programs. The traditional researches solve it by generating redundancy distribution communication, which each processor conservative sends all data defined in its local memory to all processors. This native approach provides a very clean way to generate communication code and guarantees that each processor's data access pattern will be satisfied in local. But this means a processor may receive more data than necessary, and processor that need not receive any data may receive some[9]. Redundancy distribution communication is shown in figure 1. We use cubes to represent array elements, and assign cubes with same color to same processor. Then the grey planes represent data decomposition to partition data space, which is the set of array elements.

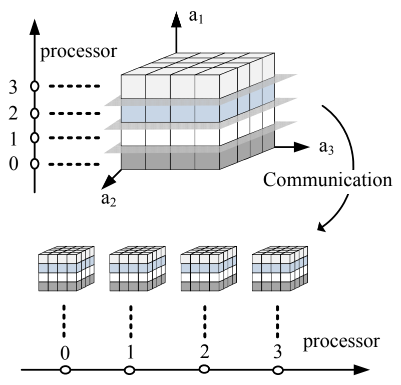


Figure 1. redundancy distribution communication

Suppose the data size that array defined in loop is *N*, then the local data size that assign to processor is *N*/*np*, *np* is the total number of processors. In redundancy redistribution communication, each array element not only migrate from its producer (processor define it) to consumer (processor use it), but also be sent to processor without the producer-consumer relation, then increase (*np*-1) communication redundancy per data. So each processor generate (*N*/*np*)*(*np*-1) communication redundancy, and total amount is (*N*/*np*)*(*np*-1)**np*= *N**(*np*-1). This indicates that with the growth of number of processors, communication redundancy will continue to increase. For the expensive cost of remote data exchange on distributed-memory architecture, increasing communication redundancy will undoubtedly reduce the parallel benefit of program.

To solve this problem, this paper proposes an accurate communication code generation algorithm for distributed-memory architecture. The algorithm confirms source processor and goals processor of each array element's migration in array redistribution by the transformation of data decompositions, then generate accurate array redistribution communication code and can effectively reduce communication redundancy with the processor scale growth.

The rest of the paper is organized as follows. Section 2 provides some necessary notion and formal description. Section 3 introduces our accurate communication code generation algorithm of array redistribution. Section 4 is the experiment and analysis. Finally, we conclude in Section 5 with a summary of the contributions of this paper.

2. Background and Notation

To describe code generation algorithm better, we provides some necessary notion and formal description in this section.

2.1. *Affine decomposition*

Affine decomposition, first proposed by Anderson and Lam in Ref 15, is an effective method to represent and find computation partition and data distribution, also is the basis of code generation algorithm proposed in this paper. It's for the domain of dense matrix code where the loop bounds and array subscripts are affine functions of the loop indices and symbolic constants. Most of the practical applications satisfy this condition[15].

Affine decomposition first maps the computation and data onto a virtual processor space which scale is not limited. The computation decomposition of the loop nest onto n-dimensional processor space is an affine function $\vec{c}: I$ ® P, $c(\vec{i}) = C\vec{i} + \vec{g}$, where C is an $n \times l$ linear transformation matrix, $\vec{\gamma}$ is a constant vector, $\vec{i}$ is an index vector for a loop nest and I is a l-dimensional iteration space. The data decomposition of the array onto n-dimensional processor space is an affine function $\vec{d}: A \to P$, $d(\vec{a}) = D\vec{a} + \vec{\delta}$, where D is an $n \times m$ linear transformation matrix, $\vec{\delta}$ is a constant vector, $\vec{a}$ is index vector for an array and A is a *m*-dimensional array space[15].

After virtual mapping stage, affine decomposition maps the processors of the virtual processor space onto the physical processors of the target machine by the physical mapping function $M_s(\vec{p})$. To use *Block* mapping, $M_s(\vec{p})$ can be expressed as equation (1)[4]:

$$M_s(\vec{p}) = \overrightarrow{pid} = \left\lfloor \vec{p} - \overrightarrow{lb_p} / \overrightarrow{bk} \right\rfloor = \left\lfloor \vec{p} / \overrightarrow{bk} \right\rfloor \qquad (1)$$

where $\vec{p}$ is virtual processor, $\overrightarrow{bk}$ is block size and $\overrightarrow{lb_p}$ is the lower bound of virtual processors. Because physical mapping always start from virtual processors $\vec{0}$, so $\overrightarrow{lb_p}$ always equal to $\vec{0}$ in $M_s(\vec{p})$. Physical processors space is non-negative integer space, so we can equivalent transform the equation of $M_s(\vec{p})$ to inequality (2)[4]:

$$\overrightarrow{bk} \times \overrightarrow{pid} \le \vec{p} \le (\overrightarrow{pid} + 1) \times \overrightarrow{bk} - 1 \qquad (2)$$

where $\overrightarrow{pid}$ is physical processors. According to definition of data decomposition, mapping data space of array to physical processors can be expressed as inequalities group (3):

$$\begin{cases} \overrightarrow{bk} \times \overrightarrow{pid} \le D\vec{a} + \vec{\delta} \le (\overrightarrow{pid} + 1) \times \overrightarrow{bk} - 1 \\ \overrightarrow{lb} \le \vec{a} \le \overrightarrow{ub} \end{cases} \qquad (3)$$

where $\overrightarrow{lb}$ is lower bound of array's data space and $\overrightarrow{ub}$ is upper bound. The processor in the conventional sense refers to the physical processor. So the "processor" is "physical processor" in the following if no special instructions.

2.2. *Symbol linear inequalities*

To generate computation and communication code in parallelizing compiler, we need to represent the iteration space of the loop nests and the data space of the arrays as different multi-dimensional integer spaces. Since many of the iteration and data spaces found in practice are multi-dimensional convex regions, these spaces can be represented as convex polyhedrons [17].

We represent all possible values of a set of integer variables ($v_1,\ldots,v_k \in Z^n$) as an *n*-dimensional discrete cartesian space, where the k-*th* axis corresponds to variable v_k. Coordinate $[x_1,\ldots,x_k] \in Z^n$ corresponds to the value $v_1=x_1,\ldots,v_n=x_n$. A parameterized convex polyhedron in the n-dimensional space of the variables $v_1,\ldots,v_n$, parameterized by symbolic constants $u_1,\ldots,u_k$, is represented by a system of linear inequalities with the variables and $v_1,\ldots,v_n$ and the symbolic constants $u_1,\ldots,u_k$. All the solutions satisfying the inequalities correspond to the integer points within the polyhedron [17].

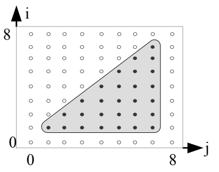
A parameterized convex polyhedron $S^n : Z^k \to P(Z^n)$ of *n* dimensions and *k* parameters is represented by the system of inequalities (4):

$$S^n(u_1,\ldots,u_k) = \left\{ \begin{array}{l} a^1 + b_1^1 u_1 + \ldots + b_k^1 u_k + c_1^1 v_1 + \ldots + c_n^1 v_n \ge 0 \\ \cdots \\ a^m + b_1^m u_1 + \ldots + b_k^m u_k + c_1^m v_1 + \ldots + c_n^m v_n \ge 0 \end{array} \right\} \quad (4)$$

where all *a*'s, *b*'s and *c*'s are integers, $u_1,\ldots,u_k$ are integer symbolic constants and $v_1,\ldots,v_n$ are integer variables[17].

Take code in figure 2(a) as example, data space of array *b* is shown in figure 2(b). The shaded area in figure 2(b) is the data space accessed by loop nest, we can represent it by linear inequalities group in figure 2(c), which is obtained based on inequalities system (4).

```
double a[9][9], b[9][9];
for (i = 1; i < 8; i++){
    for (j = i; j < 8; j++){
        a[i][j]=b[i][j];
    }
}
```



$$\begin{cases} i - 1 \ge 0 \\ n - 1 - i \ge 0 \\ j - 1 \ge 0 \\ n - 1 - j \ge 0 \end{cases}$$

(a) Sample code (b) Data space of *b*[*i*][*j*] (c)Linear inequalities

Figure 2. Linear inequalities system

Adding the inequalities relation (2) of *Block* mapping into inequalities system of figure 2(c), code generation can confirm the local data space of array distributed onto each processor. Assume data decomposition of array *b* is $\begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \end{bmatrix}$, then $bk \times pid \le i \le (pid + 1) \times bk - 1$ can be obtained according to (2). Combining it with inequalities system of figure 2(c), we can get inequalities system in figure 3(a). Loop nest in figure 3(a) can be generated by using FME to this system. Symbol *bk* and *pid* is unknown in compile time, therefor we call it symbol linear inequalities.

$$\begin{cases} i - 1 \ge 0, n - 1 - i \ge 0, \\ j - i \ge 0,\ n - 1 - j \ge 0, \\ i - bk \times pid \ge 0, \\ (pid + 1) \times bk - 1 - i \ge 0 \end{cases}$$

```
lb_i=max(pid*bk, 1);
ub_i=min(pid*bk+bk -1, 8);
for (i = lb_i; i≤ub_i; i++){
        for (j = i; j < 8; j++)
            {…=b[i][j] }}
```

(a) symbol linear inequalities (b) Loop nest of generated

Figure 3. Symbol linear inequalities system

2.3. *Two stage mapping model*

The data distribution algorithm in this article will adopt two level mapping model, which is usually used by most of the data distribution research. The two level mapping model is shown in figure 4. The first stage of the model is virtual mapping and it maps the computation and data into a size unlimited virtual processor space. The second part of the model is physical mapping through which we can map the virtual processor space into physical processor according to the data distribution equation. Virtual processor space offers an intermediate-layered and hardware platform independent mapping templet for automatic computation partition and data distribution. The templet focuses our attention on the design and optimization of the algorithm in first stage mapping.

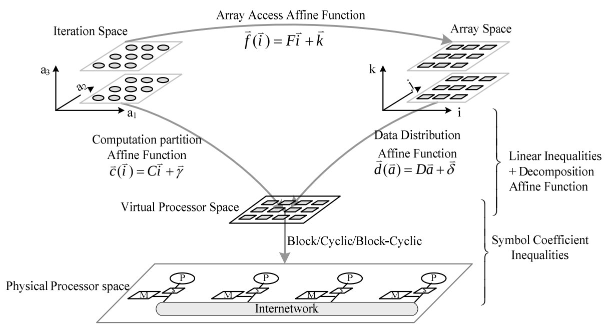


Figure 4. Two level mapping model

Virtual mapping

Virtual mapping comes down to the mapping through three spaces, that is iteration space, data space and processor space. Through the first stage of virtual mapping, data decomposition affine function describes the mapping from data space to virtual processor space and computation decomposition affine function represents the mapping from iteration space to virtual processor space while array access affine function explains the mapping from iteration space to virtual data space. The relationship of the mapping through three spaces is shown in figure 5.

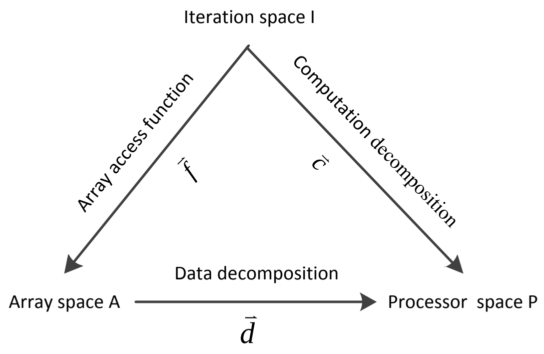


Figure 5. The mapping relationship through three spaces

The essence of the mapping process is space partition and alignment through different spaces. Space partition can be represented in matrix form, for example, one two dimension data space use vector $\vec{a} = \begin{pmatrix} a_1 \\ a_2 \end{pmatrix}$ to represent the nodes in the data space where a_1 and a_2 represent the index of the array's first and second dimension.

Physical mapping

The mapping mode between virtual processor set and physical processor set includes *Block mapping*, *Cyclic mapping* and *Block_Cyclic loop mapping*. *Block* mapping divides the virtual processor space object into several pieces equally with virtual processor in each piece consecutive and then map the pieces into physical processor one-to-one. *Cyclic mapping* circularly places the virtual processor space object on each physical processor according to index-increasing order. *Block_Cyclic loop mapping* divides the virtual processor space object into several pieces equally first and then circularly places the virtual processor space object on each physical processor according to index-increasing order. Figure 6 is the case of a one-dimensional sixteen virtual processors mapped into a one-dimensional four physical processors in the pattern of *Block mapping*, *Cyclic mapping* and *Block_Cyclic loop mapping*.

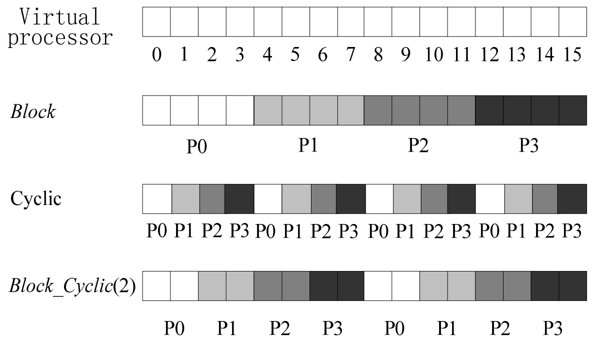


Figure 6. Case of physical mapping pattern

When dealing with the mapping from virtual processor to physical processor, we should confirm the mapping mode first. The choice of mapping mode refers to the loop index type of nest loops, the existence of parallelism in distributed loops and the existence of load balance in nest loops etc.

After completing the virtual mapping, two stage mapping model maps virtual processor space into physical processor space according to fuction $M_s(\vec{p})$ .Mapping fuction $M_s(\vec{p})$ is represented by equality (5) with *Block mapping* mode.

$$M_s\left(\vec{p}\right)=\overrightarrow{pid}=\left\lfloor \vec{p}-\overrightarrow{lb_p}/\overrightarrow{bk}\right\rfloor=\left\lfloor \vec{p}/\overrightarrow{bk}\right\rfloor \quad (5)$$

Where $\vec{p}$ is the vector of mapped virtual processor, $\overrightarrow{bk}$ is the vector of piece size which is determined by the ratio upper bound of virtual processor number and run-time physical processor number. That is $\overrightarrow{bk}$ = $\left\lceil \overrightarrow{Np}/\overrightarrow{Npid}\right\rceil$=

$\left\lceil\left(\overrightarrow{ub_p}-\overrightarrow{lb_p}+1\right)/\left(\overrightarrow{ub_{pid}}-\overrightarrow{lb_{pid}}+1\right)\right\rceil$=
$\left\lceil\left(\overrightarrow{ub_p}+1\right)/\left(\overrightarrow{ub_{pid}}+1\right)\right\rceil$, where $\overrightarrow{Np}$ is the virtual processor number, $\overrightarrow{Npid}$ is the physical processor number, $\overrightarrow{ub_p}$ and $\overrightarrow{lb_p}$ are the upper bound and lower bound vector mapped into the virtual processor by decomposition, $\overrightarrow{ub_{pid}}$ and $\overrightarrow{lb_{pid}}$ are upper bound and lower bound vector of physical processor. When dealing with physical mapping we always begin the partition with virtual processor $\vec{0}$, so the value of $\overrightarrow{lb_p}$ in mapping function $M_s(\vec{p})$ is always $\vec{0}$. Due to the physical processor spaces are all non-negative and integeral, we can transform the equality of $M_s(\vec{p})$ into corresponding inequality (6) equaivalently.

$$\overrightarrow{bk}\times\overrightarrow{pid}\le\vec{p}\le\left(\overrightarrow{pid}+1\right)\times\overrightarrow{bk}-1 \quad (6)$$

According to the definition of data decomposition, we can get the inequalities (7) that array's data space mapped into physical processor in *Block mapping* mode.

$$\begin{cases}\overrightarrow{bk}\times\overrightarrow{pid}\le D\vec{a}+\vec{\delta}\le\left(\overrightarrow{pid}+1\right)\times\overrightarrow{bk}-1\\ \overrightarrow{lb}\le\vec{a}\le\overrightarrow{ub}\end{cases} \quad (7)$$

Where $\overrightarrow{lb}$ is the lower limit of the divided array's index while $\overrightarrow{ub}$ is the upper limit.

3. Array redistribution code generation

Communication will occur when the data is non-local to the processor that references that data. If array has different distribution modes in loops of define point and use points, then needs communication to redistribute, so it can make sure the data accessed by the processor is in local. According to the data decomposition, communication of array redistribution can be classified into two classes:

(a) data-reorganization communication. In case of that the data decomposition matrix *D* changes, it makes array distribution mode reorganized inter-dimensions, and requires general movement of the entire data structure.

(b) nearest-neighbor communication. In case of that the data decomposition matrix *D* is steady, but the offset $\vec{\delta}$ changes, it makes array distribution mode shift in intra-dimension, only boundary data is needed to transfer between nearest-neighbor processor.

Next is the approach of accurate communication code generating for the data-reorganization communication and the nearest-neighbor communication.

3.1. *Data-reorganization communication*

Data-reorganization communication will occur when the data decomposition of array has inter-dimension changes, the data elements would be mapped on processors through the new data distribution strategy. The code segment in figure 7 is derived from the kernel of the program BT, an application in NAS Parallel Benchmark (NPB). We based on this code segment to explain the data-reorganization communication and the accurate communication code generation.

Supposing that the target parallel computing system has 6 processors numbered from 0 to 5, the block size *bk* is 1,and the data decomposition of the three-dimensional array *rhs* in loop 1 is[1 0 0]+[0], in loop 2 is [0 0 1]+[0]. At here a data-reorganization communication for *rhs* is necessary between loop 1 and loop 2. The figure 8(a) is the data distributed status of *rhs* in loop 1 when the partition happens in the first dimension, while the figure 8(b) is the result of data reorganization for *rhs* in loop 2 when partition the third dimension. The gray plane in figure means the partition of data decomposition to data space, the array element with the same color in each data space are distributed to the same processor; when the data's color changes in figure 8(a) and (b), it means the data must migrate to corresponding color processor.

```
#define n 7
double rhs[n][n][n]; int A, B;
for (i = 1; i < n; i++)                          //Loop 1
    for (j = 1; j < n; j++)
        for (k = 1; k < n; k++){
            rhs[i][j][k]= rhs[i][j][k]-A* rhs[i][j-1][k];
}
for (i = 1; i < n; i++)                          //Loop 2
    for (j = 1; j < n; j++)
        for (k = 1; k < n; k++){
            rhs[i][j][k]= rhs[i][j][k]-B* rhs[i][j][k-1];
}
```

Figure 7. Code to illustrate data-reorganization communication

After data redistribution of array *rhs*, all of the processors exchange data with each other, and according to the transformation of data decomposition, the data transfer in a regular way. Using the data decomposition of before and after data-reorganization communication, we can analysis the "producer - customer" relation of the data's migration between processors and generate the accurate communication code.

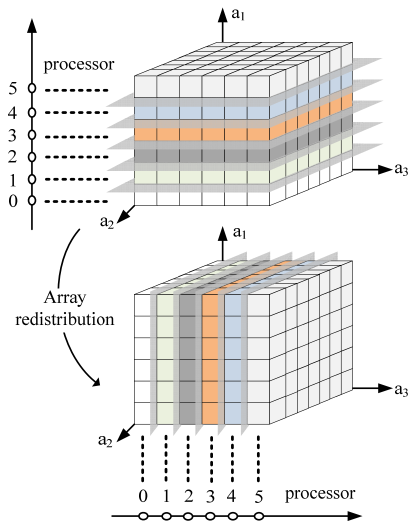
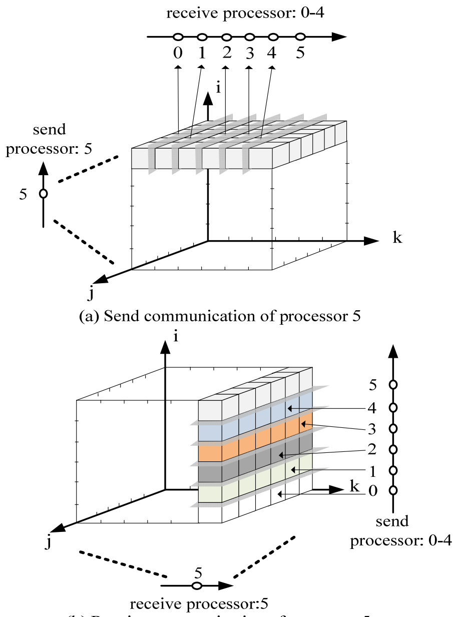


Figure 8. Data-reorganization communication

Communication code consists of three main parts: packing code, unpacking code and the communication primitive. The packing code set up the number of the data which the local processer send to the others and the offset of the data in the send-buffer, then copy the data sent into the send-buffer. The unpacking code set up the number of the data which the local processer receive from the others and the offset of the data in the receive-buffer, then copy the data received back to array from the receive-buffer. The communication primitive is used for inter-processors data exchange with the setup of the buffer. The accurate packing and unpacking code is the core part of the communication code generation.

3.1.1. *Packing code generation.*

The primary function of the packing code is filling the data sent into the send-buffer, and the crucial point is for each one of processor to judge which array element will mapping on the local and which will needs to be sent to other processors. First of all, the current processor needs to send data only from local data space of array mapped to this processor, and it is easy to acquire the local data space according to global data space and former data decomposition of array in the define point loop. Secondly, after array redistribution, the local data space mapped into processor by former data decomposition will be partitioned by the new data decomposition and be distributed into different processor. The array element remapped into other processor from the current processor is the data that needs to be sent to the processor.



(a) Send communication of processor 5

(b) Receive communication of processor 5

Figure 9. Data-reorganization communication

Supposing *n*-dimensional array *a* needs a data-reorganization communication. For the data space of *a* in define point loop, the lower and upper bound is ($\overrightarrow{lb}$, $\overrightarrow{ub}$), and data decomposition is $d(\vec{a}) = D\vec{a} + \vec{\delta}$; For the data space of *a* in use point loop, the lower and upper bound is ($\overrightarrow{lb'}$, $\overrightarrow{ub'}$), and data decomposition is $d'(\vec{a}) = D'\vec{a} + \vec{\delta}'$. Assuming target parallel computer system consists of *nprocs* processors numbered from 0 to *nprocs*-1. According to the *block* mapping function $M_s(\vec{p})$, data decomposition $d(\vec{a})$ and the array bounds ($\overrightarrow{lb}$, $\overrightarrow{ub}$), we based on the inequality (3) can acquire the local data space L_k mapped into the current processor *k*(0≤*k*≤*nprocs*-1) before array redistribution, and the boundary of L_k is ($\overrightarrow{plb_k}$, $\overrightarrow{pub_k}$). In the same way, according to $M_s(\vec{p})$, $d'(\vec{a})$ and ($\overrightarrow{lb'}$, $\overrightarrow{ub'}$) we can acquire the local data space L'_{pid} mapped into any processor *pid* (0≤ *pid* ≤*nprocs*-1) after array redistribution, and the boundary of L'_{pid} is ($\overrightarrow{plb'_{pid}}$, $\overrightarrow{pub'_{pid}}$).

```
/* Set sendbuf */
sdipls[0]=count=0;
for (i = 0; i < comm_size; i++)
    sendcouts[i]=(min(bk*(mpid+1)-1,n-1)-max(bk*mpid,1)+1)
                         *(min(bk*(i+1)-1,n-1)-max(bk*i,1)+1)*(n-1);
for (i = 0; i < comm_size; i++)
                   sdipls[i] = sdipls[i-1] + sendcouts[i-1];
/* Fill sendbuf */
for (np = 0; np < comm_size; np++)
   for (i = max(bk*mpid, 1); i < min(bk*(mpid+1)-1, n-1); i++)
      for (j = max(bk*np, 1)+1); j < min(bk*(np+1)-1, n-1); j++)
         for (k = 1; k < n; k++)
                    sendbuf[count++] = rhs[i][j][k];
/* Set recvbuf */
rdipls[0]=count=0;
for (i = 0; i < comm_size; i++)
    recvcouts[i]=(min(bk*(mpid+1)-1,n-1)-max(bk*mpid,1)+1)
                         *(min(bk*(i+1)-1,n-1)-max(bk*i,1)+1)*(n-1);
for (i = 0; i < comm_size; i++)
                   rdipls[i] = rdipls[i-1] + recvcouts[i-1];
/* MPI Collective Primitive */
MPI_Alltoallv(sendbuf, recvbuf, sdipls, rdipls);
/* Get data from recvbuf */
for (np = 0; np < comm_size; np++)
   for (i = max(bk*np, 1); i < min(bk*(np+1)-1, n-1); i++)
      for (j = max(bk*mpid, 1)+1); j < min(bk*(mpid+1)-1, n-1); j++)
         for (k = 1; k < n; k++)
                    rhs[i][j][k] = recvbuf[count++] ;
```

Figure 10. Code of data-reorganization communication

For current processor *k*, the data remapped from L_k into L'_{pid} is the communication set needs to be sent from *k* to *pid*. If restrain L_k by the boundary ($\overrightarrow{plb'_{pid}}$, $\overrightarrow{pub'_{pid}}$) of L'_{pid} , then as *pid* change from 0 to *nprocs*-1, it means partition the data space L_k of processor *k* by $d'(\vec{a})$, and the local data distributed to *pid* is need to be remapped from *k* into *pid*. In the data-reorganization communication, the sending communication set of processor *k* can be represented as:

$$comm\_set(\vec{l},\ \vec{u})_{[k,pid]} = \{(\vec{l},\ \vec{u}) \left| \left((\vec{l},\ \vec{u}) = (\overrightarrow{plb_k}, \overrightarrow{pub_k}) \cap (\overrightarrow{plb'_{pid}}, \overrightarrow{pub'_{pid}}) \right) \bigcap (k \neq pid) \right.\} \tag{8}$$

Expression (8) indicates that $(\vec{l},\ \vec{u})$ is the data set sent to the other processor from the current processor *k*. With expression (8), we can calculate the sending communication set of *k* to any processor, and generate the packing code of *k* to fill the sending data into send-buffer. For example, figure 9(a) present the local data space mapped into processor 5 before array redistribution and the sending communication set generated by the partition of new data decomposition $d'(\vec{a}) = [0\,1\,0] + [0]$.

After determined sending communication set of each processor through expression 5, we can express it as inequalities, and use the FME to generate the accurate packing code, including the setting and filling the send-buffer. Figure 9 is the data-reorganization communication code of the program in Figure 7, from first to 13 lines is the packing code, where *mpid* is the current processor number, *comm_size* is the total number of processors in the communication, array *sendcounts* records the size of the sending data to each processor, array *sdipls* records the offsets of the sending data to each processor in the buffer. Setting and filling code of the send-buffer is ahead of the communication primitive, buffer filling code is to place the array element into continuous buffer and the processor sends data according the buffer setup.

3.1.2 *Unpacking code generation.*

The primary function of the unpacking code is copy the data in the receive-buffer back into array after the communication, the crucial point is for each one of processor to judge which data element will remapping on the local and which processor is the data's producer. First of all, the data needing to be received by the current processor is the data non-local and used for computing in the loop. In another word, it is array's local data space remapped into processor by the new data decomposition after the redistribution, and it is easy to acquire this space according to the global data space and new data decomposition of array in used point loop. Secondly, if partitioning local data space with the former decomposition, then the data distributed apart from local memory is need to be received from processors that the data remapped to.

According to the *block* mapping function $M_s(\vec{p})$, data decomposition $d(\vec{a})$ and the array bounds ($\overline{lb}$, $\overline{ub}$), we can acquire the local data space L_{pid} mapped into the any processor *pid* (0≤*pid*≤*nprocs*-1) before array redistribution, and the boundary of L_{pid} is ($\overline{plb_{pid}}$, $\overline{pub_{pid}}$). In the same way, according to $M_s(\vec{p})$, $d'(\vec{a})$ and ($\overline{lb}', \overline{ub}'$) we can acquire the local data space L'_k mapped into any processor *k* (0≤*k*≤*nprocs*-1) after array redistribution, and the boundary of L'_k is ($\overline{plb'_k}$, $\overline{pub'_k}$). For the current processor *k*, the data remapped from L_{pid} into L'_k is the communication set needs to be received from *pid* to *k*.

If restrain L'_k by the boundary ($\overline{plb_{pid}}$, $\overline{pub_{pid}}$) of L_{pid}, then as *pid* changes from 0 to *nprocs*-1, it means partition the data space L'_k of processor *k* by $d(\vec{a})$, and the local data distributed to *pid* is need to be remapped from *pid* into *k*. In the data-reorganization communication, the receiving communication set of processor *k* can be represented as:

$$comm\_set(\vec{l},\ \vec{u})_{[pid,k]} = \{(\vec{l},\ \vec{u}) \\ \left| \left((\vec{l},\ \vec{u}) = (\overline{plb_{pid}}, \overline{pub_{pid}}) \cap (\overline{plb'_k}, \overline{pub'_k}) \right) \bigcap (k \neq pid) \right\} \quad (9)$$

Expression (9) indicates that $(\vec{l},\ \vec{u})$ is the data set received from the other processor by the current processor *k*. With expression (9), we can calculate the receiving communication set of processor *k* from any processor, and generate the unpacking code of processor *k* to copy the receiving data back into array from the receive-buffer. For example, figure 9(b) present the local data space mapped into processor 5 after array redistribution and the receiving communication set generated by the partition of former data decomposition $d(\vec{a}) = [1\,0\,0] + [0]$.

After determined receiving communication set of every processor through expression 8, we can express it as inequalities, and use the FME to generate the accurate unpacking code, including the setting and copying data from receive-buffer. In figure 10, the code from 14 to 20 lines and from 23 to 28 lines is the unpacking code, where array *recvcounts* records the size of the receiving data from each processor, array *rdipls* records the offsets of the receiving data from each processor in the buffer. Setting code of the receive-buffer is ahead of the communication primitive, by which the processor receives data according the buffer setup. While the filling code of the receive-buffer is behind the communication primitive, and copy the data from buffer back to array.

After generating the packing and unpacking code, the code generation algorithm inserts the *alltoall* communication primitive of MPI message passing library between the receive-buffer setting and data writing back, to complete accurate communication code generation for data-reorganization communication. In Figure 10, the code of line 21 to 22 is the communication primitive generated by the algorithm.

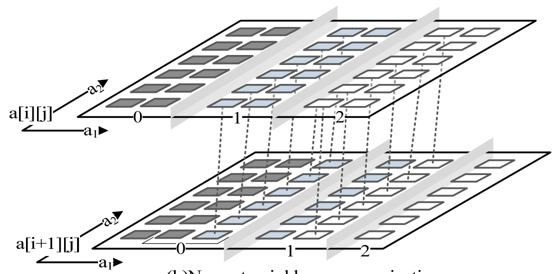
3.2. *Nearest-neighbor communication generation*

Nearest-neighbor communication will occur when the data decomposition of array has intra-dimension shift, a little of boundary data should be transferred. It is just slight inter-processor data exchange compared with the data reorganization communication, and has better data locality. The next, we take Jacobi method as example to introduce the nearest-neighbor communication and its accurate communication code generation. Jacobi is a common iterative method, the new value of one point is the average value of the old value of this point and its neighbors', its parallelization brings typical nearest-neighbor communication. Figure 12(a) represents the core code of the Jacobi iterative method, figure 12(b) show the nearest-neighbor communication for the data decomposition shifting in one dimension of array *a*.

Assuming the target parallel computer system consists of 3 processors numbered from 0 to 2, the block size *bk* is 2, the data decomposition of the array reference *a*[*i*][*j*] in loop 1 is [1 0] + [0], and the data decomposition of the array reference *a*[*i+1*][*j*] in loop 2 is $[1\,0]+\begin{bmatrix}-1\\0\end{bmatrix}$. The figure 10(b) represents the data distributed status of *a*[*i*][*j*] and *a*[*i+1*][*j*],the gray plane in figure means the partition of data decomposition to data space, the array element with the same color in each data space are distributed to the same processor. When the data distributed to different processor after remapping, it should migrate among processors according as the decomposition displacement in dimension, the dotted line represents the nearest-neighbor communication. Figure 12 show that, each processor only exchange data with its adjacent processors in nearest-neighbor communication, if which handled as the data-reorganization communication will cause a large amount of redundant communication. Therefore, the code generation needs to judge the data exchanged from one processor with its neighbor according to the displacement of data decomposition in dimension.

```
#define n 6
  double a[n][n], b[n][n];
for (it = 1; it <= itmax; it++) {
        for(i = 1; i < n-1; i++)                    //Loop 1
            for(j = 1; j < n-1; j++){
                b[i][j]=0.25*(a[i-1][j]+a[i+1][j]
                            +a[i][j-1]+a[i][j+1]);
        }
        for(i = 1; i < n-1; i++)                    //Loop 2
            for(j = 1; j < n-1; j++){
                a[i][j]= b[i][j];
        }
}
```

(a)Critical code of Jacobi iterative



(b)Nearest-neighbor communication

Figure 11. Jacobi interative

Supposing that we only partition one dimension of array, or partition multi dimensions but just one dimension there has decomposition displacement. Data decomposition of n-dimensional array *a* is $d(\vec{a})=D\vec{a}+\vec{\delta}$ in define point loop, and is $d'(\vec{a})=D\vec{a}+\vec{\delta}'$ in use point in the loop. These two data decompositions have displacement just in *i* (0≤*i*≤*n*) dimension. According to the definition of data decomposition, the array elements $\vec{a}$ shall be mapped on the virtual processor $\vec{p}=d(\vec{a})$ in define point loop and be mapped on the virtual processor $\vec{p}'=d'(\vec{a})$ in use point loop. If $\Delta\vec{p}=\vec{p}'-\vec{p}$, that is $\Delta\vec{p}=d'(\vec{a})-d(\vec{a})=(D-D)\vec{a}+\quad(\vec{\delta}'-\vec{\delta})=\Delta\vec{\delta}$ So we need to move $\vec{a}$ through $\Delta\vec{p}=\Delta\vec{\delta}$ virtual processors along the *i* dimension to complete the redistribution from $d(\vec{a})$ to $d'(\vec{a})$.

Assuming that the processor number increases in left-to-right order and starting from 0 to *P*. The local space of array *a* mapped on processor *k* (0≤*k*≤*P*) is L_k before the redistribution and changes to L_k' after the redistribution. The lower and upper bound in *i* dimension of L_k is (lb_k, ub_k), and the boundary of L_k' in *i* dimension is (lb_k' , ub_k'). For every array element needs to move through $\Delta\vec{p}=\Delta\vec{\delta}$ virtual processors while remapping, and a virtual processor only corresponds to an array elements, we can draw the following conclusions: i. when $\Delta\vec{\delta} > \vec{0}$, the data set $(ub_k - |\Delta\vec{\delta}| + 1\ ,\ ub_k)$ of L_k will be remapped to the right-neighbor processor of *k* (has lager number than *k*), the data set $(lb_k',\ lb_k' + |\Delta\vec{\delta}| - 1)$ of L_k' is remapped from the left-neighbor processor of *k* (has smaller number than *k*), as shown in figure 12(a); ii. when $\Delta\vec{\delta} < \vec{0}$, the data set $(lb_k,\ lb_k + |\Delta\vec{\delta}| - 1)$ of L_k will be remapped to the left-neighbor processor of *k*, the data set $(ub_k' - |\Delta\vec{\delta}| + 1,\ ub_k')$ of L_k' is remapped from the right-neighbor processor of *k*, as shown in figure 12(b).

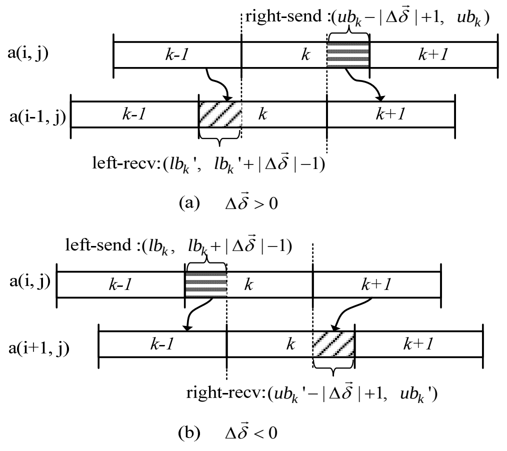

right-send :$(ub_k - |\Delta\vec{\delta}| + 1,\ ub_k)$
a(i, j)
k-1
k
k+1
a(i-1, j)
k-1
k
k+1
left-recv:$(lb_k',\ lb_k' + |\Delta\vec{\delta}| - 1)$
(a) $\Delta\vec{\delta} > 0$
left-send :$(lb_k,\ lb_k + |\Delta\vec{\delta}| - 1)$
a(i, j)
k-1
k
k+1
a(i+1, j)
k-1
k
k+1
right-recv:$(ub_k' - |\Delta\vec{\delta}| + 1,\ ub_k')$
(b) $\Delta\vec{\delta} < 0$

Figure 12. Nearest-neighbor communication

Therefore, the sending communication set of processor *k* in the nearest-neighbor communication can be expressed as:

$$\begin{aligned} & if\left(\Delta\vec{\delta} > \vec{0} \cap k < P\right) \\ & \quad send\_com(l_k^i, u_k^i)_{L_k} = (\text{send to } k{+}1) \cap (ub_k - |\Delta\vec{\delta}| + 1, ub_k)_{L_k} \\ & elseif\left(\Delta\vec{\delta} < \vec{0} \cap k > 0\right) \\ & \quad send\_com(l_k^i, u_k^i)_{L_k} = (\text{send to } k{-}1) \cap (lb_k, lb_k + \Delta\vec{\delta}| - 1)_{L_k} \end{aligned} \qquad (10)$$

The expression (10) expresses that, if we restrain the data space L_k of processor *k* by the boundary $(l_k^i,\ u_k^i)$ in *i* dimension, the data set we got is the sending communication set of *k*. Moreover, the receiving communication set of processor *k* can be expressed as:

$$\begin{aligned} & if\left(\Delta\vec{\delta} > \vec{0} \cap k > 0\right) \\ & \quad recv\_com(l_k^i, u_k^i)_{L_k} = (\text{recv from } k{-}1) \cap (lb_k', lb_k' + |\Delta\vec{\delta}| - 1)_{L_k'} \\ & elseif\left(\Delta\vec{\delta} < \vec{0} \cap k < P\right) \\ & \quad recv\_com(l_k^i, u_k^i)_{L_k} = (\text{recv from } k{+}1) \cap (ub_k' - |\Delta\vec{\delta}| + 1, ub_k')_{L_k'} \end{aligned} \qquad (11)$$

The expression (11) expresses that, if we restrain the data space L_k' of *k* by the boundary $(l_k^i,\ u_k^i)$ in *i* dimension, the data set we got is the receiving communication set of *k*.

When $|\Delta\vec{\delta}| < bk_i$, the data mapped on processor *k* only will be redistributed to the neighbor processors *k*-1 or *k*+1,so *k* only needs to exchange data with these two processors, (10) and (11) expressed in this default condition. In some extreme conditions, would occur $|\Delta\vec{\delta}| > bk_i$, all data in L_k will be redistributed to the other processor along *i* dimension, and all data in L_k' is redistributed from the other processor. According to the symbol $\Delta\vec{\delta}$ and the value of $|\Delta\vec{\delta}| \bmod bk_i$, we can get the data set which needed to be exchanged with the neighbor-processor by the boundary of each processor. In this paper, we consider $|\Delta\vec{\delta}| < bk_i$ is a default condition. Nearest-neighbor communication consists of three main parts: packing code, unpacking code and the communication primitive. Getting the sending communication set by expression (10) and adding the boundary of the offset dimension in communication set as constraint to the inequalities system of processors' local data space, we can use the FME method to generate the packing code, including the setting and filling the send-buffer. Similarly, according the expression (11), we can generate the unpacking code, including the setting and copying data from receive-buffer. Then, insert the *alltoall* communication primitive of MPI message passing library between the receive-buffer setting and data writing back, to complete accurate communication code generation for nearest-neighbor communication.

According to 3.1 and 3.2, the accurate code generation algorithm of array redistribution is shown in figure 14.

```
/* Set sendbuf and recvbuf*/
sdipls[0]=rdipls[0]=0;
|Δδ|=0, ubj=n-2, lbj=1, count=0;
for (i = 0; i < comm_size; i++)
   sendcouts[i]=recvcouts[i]=0;
if(pid >0)sendcouts[pid-1]=|Δδ|*(ubj-lbj);
for (i = 0; i < comm_size; i++)
    sdipls[i] = sdipls[i-1] + sendcouts[i-1];
/* Fill sendbuf */
if(pid > 0){
  for (i = max(bk*pid, 1); i < max(bk*pid, 1)+|Δδ|; i++)
      for (j = 1; j < n; j++)
          sendbuf[count++]=a[i][j];
}
/* Set sendbuf and recvbuf*/
if(pid <comm_size-1)recvconuts[pid+1]=|Δδ|*(ub0-lb0);
for (i = 0; i < comm_size; i++)
    rdipls[i] = rdipls[i-1] + recvcouts[i-1];
/* MPI Collective Primitive */
MPI_Alltoallv(sendbuf, recvbuf, sdipls, rdipls);
/* Get data from recvbuf */
count=0;
if(pid <comm_size-1){
  for (i = min(bk*(pid+1)-1; i < min(bk*(pid+1)-1+|Δδ|; i++)
      for (j = 1; j < n; j++)
          a[i][j] = recvbuf[count++];
}
```

Figure 13. Nearest-neighbor communication code

Algorithm 1: AccurateCommunicaiton_CodeGenreation($\vec{d}_1$,$\vec{d}_2$, *a*, ln_1, ln_2)

```
InPut  : $ln_1$, $ln_2$: loop of define point and use point;
         $a$: arbitrary array redistributed between $ln_1$ and $ln_2$;
         $\vec{d}_1(\vec{a}) = D_1(\vec{a}) + \vec{\delta}_1$ : data decomposition of $a$ in $ln_1$;
         $\vec{d}_2(\vec{a}) = D_2(\vec{a}) + \vec{\delta}_2$ : data decomposition of $a$ in $ln_2$;
OutPut : Communication of array redistribution
Begin
    calculate boundary $(\vec{lb},\vec{ub})$ of $a$'s data space in $ln_1$;
    calculate $(\vec{lb}',\vec{ub}')$ of $a$'s data space in $ln_2$;
    use $k$ to represent current processor;
    use $pid$ to represent arbitrary processor;
    calculate local space $L_{pid}$ of $pid$ by $M_s(\vec{p})$, $\vec{d}_1$ and $(\vec{lb},\vec{ub})$;
    calculate local space $L_{pid}$ of $pid$ by $M_s(\vec{p})$, $\vec{d}_2$ and $(\vec{lb}',\vec{ub}')$;
    if $(D_1 \neq D_1)$
        calculate $(\overrightarrow{plb_k}, \overrightarrow{pub_k})$ of $L_k$ and $(\overrightarrow{plb'_k}, \overrightarrow{pub'_k})$ of $L'_k$ by (3);
        get inequalities system $G_1$ of $k$'s sending communicaiton set by (5);
        use FME on $G_1$ to generate packing code;
        get $G_2$ of $k$'s receiving communicaiton set by (6);
        use FME on $G_2$ to generate unpacking code;
    else if $(\vec{\delta}_1 \neq \vec{\delta}_2)$
        $\Delta\vec{\delta} = \Delta\vec{p} = d_2(\vec{a}) - d_1(\vec{a}) = \vec{\delta}_2 - \vec{\delta}_1$;
        calculate $(lb_k, ub_k)$ of $L_k$ in offset dimension by (3);
        calculate $(lb_k', ub_k')$ of $L'_k$ in offset dimension by (3);
        calculate $(l_k^i, u_k^i)$ of $k$'s sending communicaiton set by (7);
        get $G_3$ by using $(l_k^i, u_k^i)$ to restrain $L_k$;
        use FME on $G_3$ to generate packing code;
        calculate $(l_k^i{}', u_k^i{}')$ of $k$'s receiving communicaiton set by (8);
        get $G_4$ by using $(l_k^i{}', u_k^i{}')$ to restrain $L'_k$;
        use FME on $G_4$ to generate unpacking code;
    insert collective communication primitive;
```

Figure 14. Codegen algorithm of array redistribution

4. Experimental evaluation

We conducted experiments on an 8-node Sunway cluster and a 4-processor Supermicro server. Sunway cluster has 1 service node, 8 compute nodes with 2 quad-Core Xeon processors per node, and 96GB of main memory. MPICH-1.2.7 was used for MPI communication on this cluster. Supermicro server has 4 Intel Xeon X5670 processors with 6 cores per node and a clock speed of 2.93 GHz. On this sever, main memory is 36 GB and MPICH2-1.3a2 was used for MPI communication.

To verify the correctness and effect of our algorithm, we evaluate performance on four selected commonly used applications: BT, SP, 2D-Heat and Jacobi. BT and SP come from Nas Parallel Benchmarks (NPB) and mimic the computation and data movement in computational fluid dynamics (CFD) applications; 2D-Heat is a practical application of the alternating direction implicit (ADI) method, which is a finite difference method for solving parabolic, hyperbolic and elliptic partial differential equations; Jacobi is most commonly used method with a lot of parallelism for solving large sparse linear systems.

Our communication code generation algorithm has been implemented on Open64 compiler. For comparison, the compiler generated two kinds of parallel programs. First one was generated based on traditional researches, and used the redundancy distribution communication to redistribute array. Therefore we marked this parallel program as *RPEM* (Redundancy Parallel Execution Model) in experimental result. Second parallel program that generated based on our algorithm used the accurate communication to redistribute array and was marked as *Accurate* in experimental result. Table 1 listed communication data amount of these two kind parallel programs for each application.

TABLE I. COMMUNICATION AMOUNT COMPARISON

Program	**Communication amount**		
	RPEM	***Accurate***	***Redundancy reduced***
BT	$2*np*N^3*5$	$(np+1)*N^3*5$	$(np-1)*N^3*5$
Jacobi	$np*N^2$	$np*2$	$np*(N^2-2)$
2D-Heat	$2*np*N^2$	$2*N^2$	$2*(np-1)*N^2$
SP	$2*np*N^3*5$	$(np+1)*N^3*5$	$(np-1)*N^3*5$

Firstly, we did a speedup test for 2D-Heat and Jacobi and listed the result on figure 12. 2D-Heat solves a 2D heat equation: $u_t = b_1 u_{xx} + b_2 u_{yy}$ using the alternating direction implicit (ADI) method, which reduces two-dimensional problems to a succession of one-dimensional problems[16]. Kernel code of 2D-Heat is divided into column sweep and row sweep. 2D array *u* and *v* has one data-reorganization communication between two sweeps respectively. Compared with the redundancy distribution method, our algorithm reduced $2*(np-1)*N^2$ amount of communication redundancy, and the experiment achieved significant speedup upgrade on Supermicro server and Sunway cluster. The detailed code of 2D-Heat can refer to Ref 16.

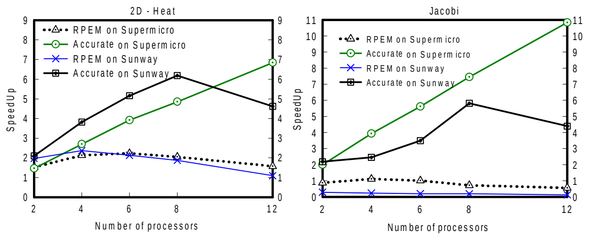


Figure 15. Expermental result of 2D-heat and Jacobi

Jacobi has good locality, thus we focus its use on testing the effects of nearest-neighbor communication. Figure 10(a) represents the kernel code of Jacobi. After partitioning data into blocks, except adjacent elements between blocks needs to be communicated, each block can be completely independent parallel computed. The computation volume of Jacobi is relatively small, so is more sensitive to the communication. Due to parallel program *RPEM* using redundancy distribution communication to redistribute array, thus brought more communication redundancy, and speedup achieved during the experiment was bad or even less than 1. Program that generated by our algorithm reduced $np*(N^2-2)$ amount of communication redundancy, therefore significantly improved the parallel performance.

Secondly, we did a speedup test for BT and SP on class W, A and B. The experiment result was listed on figure 13. These two applications belong to NPB benchmarks, which is widely used in parallel computer performance evaluation. BT is block tri-diagonal solver. The kernel code of this application can be divided into three main parallel regions. Each parallel region contains a lot of loops, and dominant array *Rhs* has two inter-regions data-reorganization communication. After the optimization of our algorithm, the amount of communication redundancy reduced from $2*np*N^3*5$ to $(np-1)*N^3*5$, thus the speedup of parallel program *Accurate* is higher than *REPM*. SP is scalar Penta-diagonal solver. SP and BT are similar in many respects, but there is a fundamental difference with respect to the communication to computation ratio. SP has lesser computation than BT, thus its speedup in experiment was overall lower than BT.

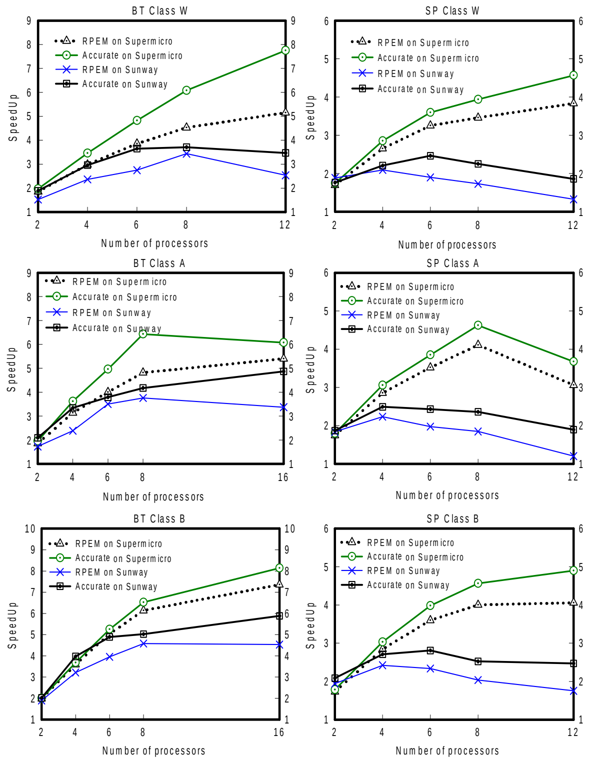


Figure 16. Expermental result of BT and SP

For Supermicro server and Sunway cluster, there is a difference with respect to parallel performance. That is why the speedups of each application in two environments are not entirely consistent. But comparing two parallel programs for each application, programs generated by our algorithm always achieve better performance and faster upward trend in speedup. On the one hand, this shows that communication redundancy has enormous impact on performance of parallel programs under the mismatching of communication and computing capacity. On the other hand, this also shows that the communication code generation algorithm proposed by this paper can effectively reduce communication redundancy of array redistribution and can improve performance of parallel programs.

In allusion to the irregular problems, we did a test for CG and IS on class A and B. The testing platform is Sunway Blue light. Two kinds of parallel programs were generated in the test. First on the foundation of traditional data decomposition and code generation algorithm we produced parallel program for irregular problems that could not be managed before optimization and marked it with *Affine* in the testing result. Then we used redundance replica technique to generate parallel program for irregular loops and marked it with *Optimized*. In order to explain the parallel performance, we did a test for the parallel version of MPI offered by NASA and marked it with *Manual* in the testing result. The testing results for CG and IS are shown in figure17 and figure 18.

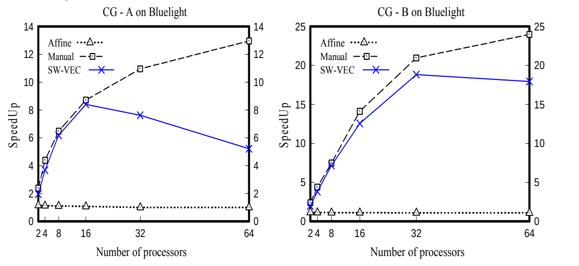


Figure 17. Expermental result of CG

In the benchmark of CG, a conjugate method is used to compute an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix. This kernel is typical of unstructured grid computations in that it tests irregular long-distance communication and employs sparse matrix-vector multiplication. The kernel loops are irregular loops and occupy 93% execution time of conjugate gradient methods. Parallel program Optimized is generated after using the algorithm raised in this article. It is able to divide the loop and generate correct correspond code to get promise speedup.

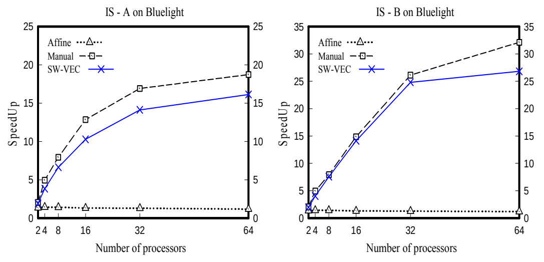


Figure 18. Expermental result of IS

The benchmark of IS tests a sorting operation that is important in *particle method* codes. This type of application is similar to particle-in-cell applications of physics, wherein particles are assigned to cells and may drift out. This benchmark tests both integer computation speed and communication performance. This problem is unique in that floating point arithmetic in not involved. In the kernel function Rank (), the nest loops occupying leading status contain array write reference that used indirect array. We got a rather good speedup for the Optimized program after partition and correspond code generation and had effect close to the manual result on class B.

The Manual parallel program offered by NASA not only used algorithm level optimization but also has no corresponding redundance. As a result, it has a better speedup relative to the Optimized program applying redundance replica technique for corresponding code generation. But our algorithm in this article cannot deal with irregular loops therefor we could not parallel the kernel loops in CG and IS. Hence the speedup is always 1 pace up and down.

5. Summary and conclusions

The traditional code generation algorithm use redundancy distribution communication to deal with array distribution. However, the resulting problem is to generate communication redundancy will continue to increase with the growth of number of processors, and significantly reduces the parallel benefits of parallel programs. Focus on this problem, this paper proposes a code generation algorithm of accurate communication. The algorithm based on data decomposition results to handle two classes communication of array redistribution respectively. For data-reorganization communication, we partition the local data space of each processor by data decomposition to get communication set; for nearest-neighbor communication, we obtain communication set according to the moving distance of local data space boundary along the dimension has offset in data decomposition; Finally, we generate packing code, unpacking code and communication primitive by using communication sets. The experiment shows that, communication data amount of applications always held remains unchanged with the growth of processors, and our algorithm effectively inhibit communication redundancy and achieve better speedups. Next, we will study the overlapping technique of computation and communication to improve the effective of communication.

6. References

1. Ancourt C, Irigoin F. Scanning polyhedra with do loops. Proceedings of the third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP ´91). NY, USA: ACM New York, 1991. 39-50.
2. Amarasinghe SP and Lam MS. Communication optimization and code generation for distributed memory machines. Proceedings of The ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation (PLDI ´93). NY, USA: ACM New York, 1993. 126-138.
3. Ferner CS. The Paraguin Compiler-Message-Passing Code Generation Using SUIF. Proceedings IEEE SoutheastCon. Washington DC: IEEE Computer Society, 2002:1-6.
4. Ferner CS. Revisiting Communication Code Generation Algorithms for Message-passing Systems. International Journal of Parallel, Emergent and Distributed Systems, 2006, 21(5):323-344.
5. Martin PJ. and Ferner CS, “Suppressing independent loops in packing/unpacking loop nests to reduce message size for message-passing code,” in the *Proceedings of the* PDPTA’07 – The 2007 International Conference on Parallel and Distributed Processing Techniques and Applications (as part of WORLDCOMP’07), Las Vegas, NV, June 15-18, 2007. p98-104.
6. Martin PJ. Suppressing Independent Loops in packing/unpacking Loop Nests to Reduce Message Size for Message-Passing Code. University of North Carolina Wilmington, USA, Wilmington NC, 2010, Master
7. M. Griebl, Automatic Parallelization of Loop Programs for Distributed Memory Architectures. FMI, University of Passau, 2004.
8. M. Classen and M. Griebl. Automatic code generation for distributed memory architectures in the polytope model. In *IEEE IPDPS*, Apr. 2006.
9. Bondhugula U. Automatic Distributed-Memory Parallelization and Code Generation using the Polyhedral Framework. Technical Report, Report No.IISc-CSA-TR-2011-3, Bangalore: Indian Institute of Science, 2011.
10. Kwon O, Jubair F, Eigenmann R and Midkiff S. A Hybrid Approach of OpenMP for Clusters. Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP ´12). NY, USA: ACM New York, 2012. 75-84.
11. Basumallik A and Eigenmann R. Optimizing irregular shared-memory applications for distributed-memory systems. Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP ´06). NY, USA: ACM New York, 2006. 119-128.
12. Ravishankar M, Eisenlohr J, Pouchet LN, Ramanujam J, Rountev A, Sadayappan P. Code generation for parallel execution of a class of irregular loops on distributed memory systems. The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC12). CA, USA: IEEE Computer Society Press Los Alamitos, 2012.
13. Ravishankar M, Eisenlohr J, Pouchet LN, Ramanujam J, Rountev A, and Sadayappan P. Code generation for parallel execution of a class of irregular loops on distributed memory systems. Technical Report, Report No.OSU-CISRC-5/12-TR10. The Ohio State University, 2012.
14. Kim H, Johnson NP, Lee JW, Mahlke SA, August DI. Automatic Speculative DOALL for Clusters. Proceedings of the 10th International Symposium on Code Generation and Optimization (CGO ´12). NY, USA: ACM New York, 2012. 94-103.
15. Anderson JM and Lam MS. Global optimizations for parallelism and locality on scalable parallel machines. In: Cartwright R, ed. Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation. Albuquerque: ACM New York, 1993.112–125.
16. Lee PZ, Kedem ZM. Automatic Data and Computation Decomposition on Distributed Memory Parallel Computers. ACM Transactions on Programming Languages and Systems, 2002, 24(1): 1-50.
17. S. Amarasinghe. Parallelizing Compiler Techniques Based on Linear Inequalities. PhD thesis, Stanford University, 1997.