Array-level Collective Communications

Shuang ZHANG and Yi-feng CHEN

School of Electrical Engineering and Computer Science, Peking University, China

*Corresponding author

Keywords: MPI, Distributed data, Collective communication.

Abstract. The Message Passing Interface (MPI) provides some well-defined collective interfaces for a set of messages between multiple processes. However, MPI can be improved. The collective interface in MPI is too specialized, that many collective communication patterns still require instantiation by programmers. In this paper, we describe a novel tool Parallel Dimension (PADM). PADM is defined to extend MPI for a more general class of collective communications. Therefore, in PADM, both the MPI interfaces and other collective communications are performed with the same general solution. PADM provides array style abstraction for both process topology and data distribution from the global view. Unlike the highly Partitioned Global Address Space (PGAS) model, PADM allows programmer to explicitly control the distribution and communication. And PADM provides more concise communication sentences, and allows greater scope for optimization.

Introduction

MPI is the most popular and ubiquitous choice for distributed communication. It provides well-defined interfaces for some specialized collective communication, such as broadcast, gather, scatter and etc. For programmers, applying the collective interfaces instead of the point-to-point sentences will boost the productivity. However, these collective interfaces are too specialized to cover various patterns. For example, the group-wise collective operation is undefined in MPI. To apply these interfaces in multiple groups, the programmers have to use the MPI communicator to construct partitioned communication context and process groups. However, communicator creation is an operation that exchanges information between processes with expensive synchronization cost. Another undefined example is the circular neighbor communication. This communication pattern is commonly used in stencil problems, such as scientific simulation and image processing. Each process exchanges data with its neighbors. The programmers have to use the point-to-point sentences, like MPI_Irecv and MPI_Send, or the MPI topologies such as MPI_Cart_shift interface to perform this kind of communications.

Furthermore, it is possible to design specific optimization for each collective interface according to its definition. The implemented libraries for MPI, such as OpenMPI and MPICH, have been optimized in several interfaces. There are many studies focused on the optimization of collective operations to improve the performance. For each interface, they use multiple algorithms depending on the message size, such as [1], [2]. The optimization only applies to these specialized patterns in MPI. Actually, many various patterns are suitable for these algorithms.

In this paper, we describe our work PADM on providing a single general solution for collective communication operations. This general interface is based on an array-level algebraic representation dimension. The construction of dimension follows several simple rules which is easy to learn. Dimension contains both the process topology and data distribution. Therefore, each communication is simplified as a pair of dimensions and a copyto sentence in PADM. For programmers, our work is able to boost the productivity. In addition, the dimension is a binomial tree structure. It is possible to analyze the structure and identify the communication pattern. Then we can design the optimized algorithm for each pattern.

Early work on simplifying communications focused on the PGAS. PGAS is the basis model of a large number of languages, such as Unified Parallel C, Co-array Fortran and Global Arrays [11]–[14]. It assumes a global memory address space that is logically partitioned and a portion of it is
local to each process. These languages boost programmer productivity by providing shared variables. However, the distribution is implicit in most of the languages. The runtime performs one-sided communication sentences automatically according to the data requirement on each process. Considering the performance, programmers should be aware of the data locality. Thus, compared with PGAS, our work provides explicit distribution and communication.

The rest of the paper is organized as follows: Section II introduces the construction rules for dimension and the general copyto sentence, then Section III compares the MPI and PADM codes in several small cases to show that our work provides more concise program, and investigates into a case study on matrix multiplication. Section IV introduces the implementation and optimization. Section V illustrates and discusses the cost model and performance. Section VI concludes our work.

Parallel Dimension

PADM is an extension of MPI to provide a general interface for various collective communication. This interface is based on our conception dimension. In this section, we introduce the construction rules in PADM to provide an intuitive understanding.

Construction Rules

The construction follows several simple rules, as shown in Equation (1) and (2). These rules are flexible enough to describe the common distribution.

\[
D = \text{tag} (\text{size, step, disp}) \tag{1}
\]

\[
S = D_0 * ... * D_n | \text{array}(D_0 * ... * D_n) | \text{array} \ (S_{\text{ref}}, S_{\text{base}}) \tag{2}
\]

First, in Equation (1), each basic dimension(D) has one tag and three fundamental components: size, step and disp. There are three keywords for tag, dim(dummy), proc(process) and data that represent different elements. Dummy is a void type without specific data or process information. Size is the number of elements with default value 0. Step is the distance between two elements in sequence, which is also called stride in PGAS [14]. Its default value is 1, which means the continuous situation. And disp means the displacement with default value 0. Elements’ offset will move at a certain distance according to this. For each index in dimension, \( i \in (0, \text{size} - 1) \), the offset is calculated as:

\[
\text{offset}(i) = (i \% \text{size} + \text{disp}) \times \text{step}. \tag{3}
\]

Since the default values mentioned before can be omitted. Let us look at a simple construction according to the first rule in Equation (2):

\[
\text{dim} \ A = \text{proc}(4) * \text{data}(4, 16) * \text{data}(16).
\]

The multiplication operator ‘*’ is used to construct multi-dimensions. This dimension \( A \) describes distributed data on 4 processes, and each process contains 64 continuous data elements. It has three dimensions, but it is also a two-dimensional array with 16*16 elements, or a one-dimensional array with 256 elements. Each dimension is constructed as a binomial tree according to the multiplication order. Using the brackets to change the order will produce different trees as shown in Figure 1.

![Dimension trees with different multiplication order.](image)

Figure 1. Dimension trees with different multiplication order.
To describe a continuous sequence, we have to set the correct step value for each dimension, like 
\(data(4, 16)\) in \(A\). Or we can use the second rule in Equation (2) to fill the step values automatically for 
continuous situation as follow:

\[
dim A = array(proc(4) \times data(4) \times data(16)).
\]

The last rule in Equation (2) is called reference mechanism. It is used to describe more flexible 
distributions. A reference contains two dimensions, the ref dimension \(S_{\text{ref}}\) and the base one \(S_{\text{base}}\), 
which is constructed as follow example:

\[
dim R = array(dim(8) \times dim(8), A).
\]

In this case, \(A\) is the base one, and \(dim(8) \times dim(8)\) is the new description based on \(A\). We use the 
dim type here without specific data or process information. Since \(A\) is configured as a \(16 \times 16\) two-
dimensional matrix. Then \(R\) describes an \(8 \times 8\) sub-matrix in the top left corner of \(A\). The offsets are 
calculates in nested way, \(R\), offset(i) = base.offset(ref.offset(i)).

Actually, there are two kinds of offset: process number and local data offset. We provide function 
node and offset to calculate them respectively, like \(A\) node(i) and \(A\) offset(i). Moreover, for 
simplicity, we use ‘+’ operator to set the disp value. For example, the rank 1 process can be 
described as proc(1) + 1, which is better than proc(1,1,1).

Copyto

The general communication interface is defined as:

\[
copyto(dim S_s, void *s, dim S_t, void *t, int elem_size);
\]

where \(S_s\) is the source distribution, and \(S_t\) is the target one of the same size. The void pointers are the 
source and target data addresses. If there is no read-write conflict, the programmers can use the same 
address for both pointers. The last parameter requires the size of data element, such as sizeof(float).

Column and Row

Since every multi-dimension is constructed as a binomial tree, it is also a two-dimensional array. 
And the two children of the root/parent node represent the column and row dimension, respectively. 
We can use functions \(col(A)\) and \(row(A)\) to get them from dimension \(A\). The default order is row-
major, using the function \(col\_major(A)\) will change it into column-major.

Transform Functions

The reference is able to describe different order, re-division, portion and repetition of the basic 
one. PADM provides well-defined transform functions to simplify the reference declaration, and 
some frequently used functions are shown in Table 1. The first four functions will treat the input \(D\) 
as one dimension, while the others require that \(D\) is two-dimensional. The function multi will return 
a collapsed dimension, which means that the reference size is larger than the base size. We provide 
function \(dsize\) and \(psize\) to get the actual process and local data size. For example, \(multi(proc(1), 4)\). \(psize()\) returns 1.

<table>
<thead>
<tr>
<th>Function</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>dim get_low(dim D, int k)</td>
<td>get (k) units from (D) that near the origin</td>
</tr>
<tr>
<td>dim get_high(dim D, int k)</td>
<td>get (k) units from (D) that far from the origin</td>
</tr>
<tr>
<td>dim multi(dim D, int k)</td>
<td>repeat (D \times k) times</td>
</tr>
<tr>
<td>dim cyclic(dim D, int k)</td>
<td>cyclic (D \times k) units</td>
</tr>
<tr>
<td>dim cyclicRow(dim D, int k)</td>
<td>cyclic row dimension of (D)</td>
</tr>
<tr>
<td>dim cyclicCol(dim D, int k)</td>
<td>cyclic column dimension of (D)</td>
</tr>
<tr>
<td>dim block_major(dim D, int k)</td>
<td>block-major order with (k \times k) block size</td>
</tr>
</tbody>
</table>

Table 1. Transform Function.
**Concise Code**

We have discussed some cases in Section I that is hard to write in MPI. In this section, we take a simple case as example to compare the MPI and PADM codes. The common code is omitted in both versions to show the difference. It is clear that the PADM program is concise and easy to understanding. Then we will discuss the distribution-independent algorithm and a case study, matrix multiplication.

**Circular Neighbor Communication**

The first case is the circular neighbor communication. Each process sends 100 float elements to its right neighbor, and receives the same count from the left one. There are \( pnum \) processes in total. The MPI version is shown in Figure 2(a). Programmers should consider many details, such as the destination process number, blocking or non-blocking interfaces, avoiding deadlock (non-blocking first) and ensuring the completion (MPI_Wait). Even using the MPI topology interface will not reduce the amount of code. While in PADM version, as shown in Figure 2(b), there is only one task that is to design the source and target dimensions for \textit{copyto} function. We use the transform function \textit{cyclic} here to describe the destination processes.

![Figure 2. MPI vs PADM neighbor communication.](image)

**Distribution-independent**

The second case is a library function of matrix transpose. When designing a library function in MPI, the programmers must be aware of the original distribution of input matrix, such as block or row scattered data distribution. The different original distributions result in different message passing patterns. However, in PADM, it is able to design a \textit{distribution-independent} algorithm. It means the specific data layout is unnecessary in subroutine. Our function is shown in Figure 3. It requires that the column and row dimensions in \texttt{A} represent the column and row of the matrix, respectively. We exchange the column and row dimensions to describe the target distribution \texttt{B}. Also we can use \texttt{col_major(A)} to get it alternatively.

![Figure 3. Distribution-independent transpose in PADM.](image)

**Case Study: Matrix Multiplication**

In the traditional 2D algorithm of matrix multiplication that \( C = A \times B \), the computation task is partitioned to a 2D processes grid. Before the local multiplication can be carried out, each process
must get multiple rows and columns from A and B. In each matrix distribution of A and B, if the size of process grid is P, the communication size is $P^{1/2}$ times of the matrix size.

A three-dimensional approach [4]–[6] to parallel matrix multiplication has a factor $P^{1/6}$ less communication than the 2D algorithm. In this algorithm, the processes are configured as a 3D cube. Each process gets a single sub-matrix from A and B, then performs a local matrix multiplication. After the computation, several sub-matrices of this product must be sent to their destination processes and then summed together with the resulting matrix C. The computation is described in Figure 4. There are three communications in this algorithm. The first one distributes the sub-matrices of A to all processes. The second one distributes those of B with different destinations. And the third one gathers all the products for each result on each process. The communications are complicated. Furthermore, in MPI, the programmers must be aware of the original distributions of data.

![Figure 4. Cube algorithm for matrix multiplication.](image)

In PADM, it is possible for programmer to write the distribution-independent algorithm from an intuitive way. In our implementation, as shown in Figure 5, the input dimensions are considered as two-dimensional. Each process performs local sub-matrices multiplication of size len*len. And the blocks number of A, B and C is m*n, n*k and m*k, respectively. Therefore, we need a process cube of size m*n*k. The declaration of each cube dimension is separate, as shown in Line 7-9. Before the local computation, the sub-matrices of A are scattered among the $Pm*Pn$ processes, and are repeated along the $Pk$ dimension. Similarly, the matrix B is scattered among the $Pn*Pk$ grid, and is repeated along the $Pm$ direction.

```c
void cubeGemm(int len, dim A, float* a, dim B,
    float* b, dim C, float* c){
    int elem_size = sizeof(float);
    int m = col(A).size/len;
    int n = row(A).size/len;
    int k = row(C).size/len;
    //proc dims
    dim Pn = proc(m, n+k);
    dim Pk = proc(n, k);
    //local data
    dim Blk = data(len,len)-data(len);
    float *ina, *inb, *amb; //malloc...
    //copy A
    dim A_Blk = multi(proc(1),k) x block_major(A,len);
    copyto(A_Blk, a, INA, ina, elem_size);
    //copy B
    dim B_Blk = multi(proc(1),m) x block_major(B,len);
    copyto(B_Blk, b, INB, inb, elem_size);
    //compute products
    gemm(BLK, ina, inb, amb);
    //copy AMB
    dim AMB = Pn x Pm x Pk x Blk;
    //copy C
    dim CP = data(n, C.size); //malloc...
    copyto(CP, inb, CSM, csum, elem_size);
    //sum to C
    for(int i=0; i<CP.size; i++)
        c[i] += csum[CP.offset[i]];}
```

![Figure 5. Cube algorithm in PADM.](image)
The local sub-matrix is described as Blk in Line 11, which is continuous in memory. But the corresponding input data may be not entirely continuous or on multiple processes. Considering the performance, the design of the dimensions must ensure that the communication continuous length is long enough. Therefore, the local data description Blk is at the rightmost side of the target dimension. At least, the target dimension has the longest continuous length. And we use the block major function to adjust the original row-major input dimensions into block-major order. By using the multi-function, the adjusted offsets are repeated several times. From Line 14-20, matrix A and B are distributed and repeated on the cube processes.

After the local computation gemm, the products along the \( P_n \) direction must be summed together. We use the \( CP \) dimension to enlarge the data offsets scope, and allocate memory to receive these products on local. The \( P_n \) dimension corresponds to the \( CP \) dimension. And the product blocks on \( Pm*P*P_n \) correspond to the block-major of \( C \), as shown in Line 24-26. Therefore, the products are sent to their respective destination processes and then summed together with the resulting matrix \( C \).

In our code, only the malloc and free sentences are omitted. The code is concise and easy to understand. Especially, this algorithm accepts different original distribution. In Figure 6, we describe three different distributions for matrix A. Case one is row scattered data distribution. Case two is a block scattered pattern on \( r*r \) process grid, and each block is a square sub-matrix. Case three is also a block distribution with different grid pattern, where block is a rectangle part of the matrix. These patterns are also available for matrix B and C. Different distributions result in different performance because of the different communication continuous length. If the matrix multiplication of size 8192*8192 is parallel on 64 processes, the performance is shown in Table 2. The last column \( cnt \) is the number of continuous data in communication. The third case performs best because of the largest \( cnt \) value. It is possible to design a distribution-independent algorithm in PADM by using the reference mechanism and general copyto function. Therefore, tuning the performance with different inputs is easy.

```c
1 //matrix n*n; processes p = r*r = q*r*q
2 //case1: row
3 dim A = (proc(p) * data(n/p,n)) * data(n);
4 //case2: block
5 dim E = proc(r) * data(n/r);
6 dim A = array(E*E);
7 //case3: rectangle
8 dim Row = proc(q) * data(n/q);
9 dim Col = proc(q) * data(n/q*q);
10 dim A = Col+Row;
11 //call
12 cubeGemm(n/q, A, B, B, C, c);
```

Figure 6. Various input distribution for cube algorithm.

<table>
<thead>
<tr>
<th>Time(s)</th>
<th>Copy A</th>
<th>Copy B</th>
<th>Copy C</th>
<th>Total</th>
<th>cnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>case 1: row</td>
<td>0.223</td>
<td>0.254</td>
<td>0.233</td>
<td>0.710</td>
<td>2^{11}</td>
</tr>
<tr>
<td>case 2: block</td>
<td>0.282</td>
<td>0.338</td>
<td>0.320</td>
<td>0.940</td>
<td>2^{10}</td>
</tr>
<tr>
<td>case 3: rectangle</td>
<td>0.180</td>
<td>0.179</td>
<td>0.157</td>
<td>0.516</td>
<td>2^{20}</td>
</tr>
</tbody>
</table>

**Implementation and Optimization**

PADM is implemented by using MPI, OpenMP and C++. We defined it as a library tool that compatible with MPI. It is a lightly extension that allows programmers to use an abstract and logic description of data which is separates from the actually physical memory. It is easy to use, PADM provides a head file to be included. The programmer only need to add some options such as `-fopenmp` when compiling.

The general `copyto` function requires source dimension and the same size target one. By scanning all the indexes in the size scope only once time, we can calculate the element locations at both ends, and then copy this element from source to target by using point-to-point message sentences. This is the most basic core solution.
Overlapping and Multi-threads

A common optimization method is overlapping the computing and communication. In our solution, the computing refers to the location calculation for each index, called scan. We choose OpenMP to launch multiple threads on each process, but only one thread is allowed to use the message passing sentences. However, the scan work is partitioned among several threads. Each computing thread scans a portion of the dimension and calculates the locations. If the source or target location is on the current process, then we push a send or receive request into global queue. Once the queue is not empty, the communication thread will pop up each request from the queue and turn it into a MPI_Isend or MPI_Irecv sentence. Then we use MPI_Test to ensure that these requests are completed. Moreover, we provide a function setTnum to set the number of scan threads.

PADM-led Optimization

The PADM dimension is an analyzable binomial tree. We can get information from the structure without scanning indexes. Now we are going to discuss some optimizations that based on the analysis.

Continuous Length. Each dimension represents a sequence of data offsets. These offsets may be continuous in address, such as proc(4)*data(100). In other cases, they may not be entirely continuous, like data(10,20)*data(10), that every 10 elements is continuous in memory. The number of regular continuous elements on each process is configured as the continuous length. We can use A.cntLen() to get it.

The two dimensions in one communication may have different continuous length. The greatest common divisor of these two values is the continuous length cnt of this communication. It means that every cnt indexes refer to the same source and target process number, and their offsets are continuous in memory at both ends. Therefore, only the first index in each continuous segment is need to be scanned. Meanwhile, the message length is increased to cnt. This solution called scanComm.

Granularity. To optimize the communication when cnt is small, we introduce another argument gran that means the granularity. The message length is set to this granularity rather than cnt. Usually, the value of gran is several times the size of cnt. Therefore, each process packs several continuous segments with the same destination into one buffer block and send it. The destination process provides another buffer to receive the message. Once a receive message is completed, the destination process will unpack the buffer and move each continuous segment to the right offset. In this situation, we use same size of threads to perform packing and unpacking seperately. In the communication thread, semaphore is used to notice the unpacking thread that a buffer block is ready to be unpacked. This solution called bufComm.

In addition, we design a buffer pool to recycle the buffer blocks. The pool is partitioned into many blocks of size gran. And these block addresses are managed by a global queue. When a new block is required, a buffer address is popped up from the queue. If the queue is empty, we will allocate a temporary block which is also recycled after use. After the communication, both the pool and temporary blocks are released.

The copyto function will analyze the input dimensions and set arguments automatically including cnt, gran, the size of buffer pool and etc. Also we provide functions for programmers to control some of them, such as setGranBytes and setBufBytes.

Locality. By analyzing the source and target dimensions, we can discuss the locality. If the data dimensions correspond to each other between source and target, and the process numbers at both ends are in the same sequence, it means that the data movement only occurs on local, such as:

\[ \text{dim } S = \text{proc}(4) * \text{data}(10, 10) * \text{data}(10); \]

\[ \text{dim } R = \text{proc}(4) * \text{data}(10) * \text{data}(10, 10). \]
In this situation, the movement is performed by `memcpy` instead of message passing. This branch solution is called `localCopy`. The `copyto` function chooses the proper optimized branch solution according to the dimensions’ information.

**Identification and Optimization for Collective Operation**

We take broadcast operation as an example to discuss the algorithm level optimization. There is an optimization for long message broadcast. The message is first divided up and scattered among the processes. The scattered data are then collected back to all processes, similar to an allgather. We measured the un-optimized MPI Beast of OpenMPI and this new algorithm on 32 processes. The new algorithm reduces up to 74.5% time when the message length is 1GB. This existing optimization can be applied for both standard and more various scenarios in PADM. For these more general broadcast, the source and target dimension have some common features. It is possible to analyze the dimensions’ features to identify the communication pattern.

**Identification.** We choose three cases of broadcast as shown in List 7. The first case describes broadcast on all processes with not entirely continuous message, and the data layout is changed during the broadcast. The second one only broadcast the root data to even number processes. And the third one describes a group-wise broadcast. The features of broadcast dimensions are concluded as follows, where A is the source dimension and B is the target one:

a) \(A.psize() < B.psize()\);
b) \(A.dsize() = B.dsize()\);
c) only A contains a collapsed dimension;[

d) A’s data and process dimension corresponds to that of B respectively;[n

e) both the data offsets are in a successive segment without overlap or jump.[n

As we mentioned before, the function `p size` and `d size` return the actual size of process and local data. Since the data on root process is sent to multiple destinations, the actual process size of the source dimension is definitely less than that of the target one. And the actual data at both ends is the same size. The repetition of root message results in collapsed proc or data dimension. If there is a collapsed data dimension, we will separate the repetition description from data and turn it into a collapsed process dimension for identification.

If one’s data corresponds to another one’s process, it means that data on one process will be distributed to multiple processes, which is definitely not a broadcast pattern, this is why we have feature (d). The offsets may not be entirely continuous, but they must be in the successive memory segment without overlap and jump.

**Optimization Algorithm.** If the pattern is identified as a broadcast, we separate data and process descriptions for both the source and target dimensions. Therefore, we get four new dimensions represent source process, source data, target process and target data, respectively. These dimensions are used to describe the scatter-allgather communication. If the message is long enough, the new algorithm solution is performed. Otherwise, it performs the general `copyto`.

Therefore, all the standard, group-wise, not entirely continuous data, and many other various broadcast can be optimized with this new algorithm in PADM. The performance is discussed in the next section.

**Cost model and Performance**

In this section, we discuss performance from several cases. The performance is measured on a 32-nodes test set of the Mole-8.5 Supercomputing [3]. And the MPI library is the OpenMPI 2.1 version.

**Cost Model**

We use a simple model to estimate the performance. We assume that the time taken for each process to complete the communication can be modeled as \(m\alpha + 2ny + N\beta\), where \(m\) is the number of messages, \(\alpha\) is the startup time per message, independent of message size, \(n\) is the bytes per message,
\( \gamma \) is the packing or unpacking time per byte, \( N \) is the total number of bytes transferred, \( N = m \cdot n \), and \( \beta \) is the transfer time per byte. We assume further that the communication time always overlapped the packing and unpacking time, like Figure 7(a).

![Figure 7(a)](image)

When tuning granularity for specific case, the value of \( N \beta \) is constant, and \( m \) is inversely proportional to \( n \). If \( cnt \) is large enough that there is no packing or unpacking, the cost model turns to \( m \alpha + N \beta \), which means that less messages is better.

**General Copyto**

We choose a not entirely continuous circular neighbor communication case as follows:

\[
dim P = \text{proc(pnum)};
\]

\[
dim A = P \cdot \text{data(2, 1024)} \cdot \text{data(n, 2048)} \cdot \text{data(1024)};
\]

\[
dim B = \text{cyclic(P, 1)} \cdot \text{data(n * 2048)}.
\]

**copyto(A, s, B, r, sizeof(float));**

We have introduced the optimized solution \text{scanComm} and \text{bufComm} for \text{copyto} function. In this case, \( cnt \) is 4KB. We set the granularity as 1/64 of the local data. If \( n=128 \), each process holds 1MB data, then \( \text{gran} \) is 16KB. The performance is shown in Figure 7(b). The ‘cnt8’ represents an 8-process communication with \( \text{cnt} \) message length. Since the 4KB continuous length is not long enough, the \text{bufComm} solution increases the message length and performs better. Furthermore, we discuss different granularities in another point-to-point communication case that rank 0 process sends 256MB data to the rank 1 process, and the continuous length is also 4KB.

\[
dim D = \text{array((data(256)*data(256))*data(1024))};
\]

\[
dim A = \text{proc(1)*col\_major(col(D))*row(D)};
\]

\[
dim B = (\text{proc(1)+1}) \cdot D
\]

**copyto(A, s, B, r, sizeof(float));**

The performance is shown in Figure 7(c). The threads number includes both the packing and unpacking group. It achieves better performance from 1MB to 8MB across different threads number.
If the granularity is too large, the $2n\gamma$ cost a lot. If the granularity is too small, we have to manage a lot of short messages which increases the startup cost $ma$.

**Standard Collective Operations**

We take the alltoall instance as the basic collective communication to measure the performance. The algorithm for MPI Alltoall in OpenMPI does not attempt to schedule communication. Instead, each process posts all the MPI Irecv in a loop, then all the MPI Isend in a loop, followed by an MPI_Waitall. This is similar to the scanComm solution.

Figure 7(d) shows the performance for MPI Alltoall versus PADM *copyto* with different threads. Each process contains 1GB data. The granularity is the same as the *count* size. In PADM cases, the performance of 8 threads is slightly better than the performance of 4 threads, which shows the advantage of using multiple threads. The overhead in PADM does not significantly degrade the performance. Actually, our *copyto* function performs better when the process number is less than 32.

**Optimized Various Broadcast**

First, we measure the performance of standard broadcast on 32 processes. The algorithms of MPI Bcast, *copyto* and the new one(scatter-and-allgather) are different. Their performance is shown in Figure 7(e) and 7(f). Here the message length equals to the total local data size. We see that for short messages($\leq$4MB), *copyto* performs better than both the MPI and the new one. And for long messages($\geq$8MB), the new algorithm performs best.

Then, we measured three various cases in Figure 8 on 32 processes with different message size. In the list, the value of $n$ is set to 512, and the continuous length of $Dt$ dimension is 1/1024 of the local data size. The performance of the optimized solution and general *copyto* is shown in Table 3. Each *copyto* contains one synchronization. The BcastOpt solution thus is synchronized twice. Therefore, when the message is longer, the optimized algorithm performs better. When the message length is 512MB, the BcastOpt reduces 88.3% communication time compared with the general solution in the first case.

```
int pnum, root, n, num;
2 dim Root = proc(1)+root;
3 dim P = proc(pnum);
4 dim D = array(data(n)*data(2))\text{data}\{n+2\};
5 dim Dt = col_major(col(D)) * row(D);
6 //case 1, not entirely continuous
7 dim A1 = multi(Root, pnum) * Dt;
8 dim B1 = P * Dt;
9 //case 2, broadcast to even number processes
10 dim A2 = multi(Root, pnum/2) * Dt;
11 dim B2 = proc(pnum/2) * D;
12 //case 3, two group of broadcast
13 dim A3 = multi(proc(2), pnum/2) * Dt;
14 dim B3 = P + D;
```

Figure 8. Various broadcast patterns.

<table>
<thead>
<tr>
<th>Message length</th>
<th>Case 1</th>
<th>Case 2</th>
<th>Case 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>opt</td>
<td>copyto</td>
<td>opt</td>
</tr>
<tr>
<td>2MB</td>
<td>0.221</td>
<td>0.426</td>
<td>0.205</td>
</tr>
<tr>
<td>8MB</td>
<td>0.225</td>
<td>0.537</td>
<td>0.239</td>
</tr>
<tr>
<td>32MB</td>
<td>0.256</td>
<td>0.572</td>
<td>0.260</td>
</tr>
<tr>
<td>128MB</td>
<td>0.326</td>
<td>1.620</td>
<td>0.290</td>
</tr>
<tr>
<td>512MB</td>
<td>0.778</td>
<td>6.640</td>
<td>0.684</td>
</tr>
</tbody>
</table>

**Conclusion**

PADM provides a general solution for various collective patterns. The abstraction of distribution is followed by several simple rules which is intuitive and flexible. In PADM, the programmers are
aware of the data locality and able to control the performance related factors. It provides more concise codes and greater scope for optimization. Our work is open source on GitHub, https://github.com/zs192001/padmV0. The future work is to discuss more scientific cases.

Acknowledgment
This research is supported by National Key R&D Program of China under Grants No. 2016YFB0200502; and National Natural Science Foundation of China Grants No. 61672208, 61432018. Thanks to the Mole-8.5 Supercomputing System developed by Institute of Process Engineering, Chinese Academy of Sciences.

References