8.1 Parallel Redistribution of Multidimensional Data

Tore Birkeland and Tor Sørevik

NIC Series Volume 38:
Parallel Computing: Architectures, Algorithms and Applications
C. Bischof, M. Bücker, P. Gibbon, G.R. Joubert, T. Lippert, B. Mohr and F. Peters (Eds.)
Parallel Redistribution of Multidimensional Data

Tore Birkeland, Tor Sørevik

published in


© 2007 by John von Neumann Institute for Computing
Permission to make digital or hard copies of portions of this work for personal or classroom use is granted provided that the copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise requires prior specific permission by the publisher mentioned above.

http://www.fz-juelich.de/nic-series/volume38
Parallel Redistribution of Multidimensional Data

Tore Birkeland and Tor Sørevik

Dept. of Mathematics, University of Bergen, Norway
E-mail: \{tore.birkeland, tor.sorevik\}@math.uib.no

On a parallel computer with distributed memory, multidimensional arrays are usually mapped onto the nodes such that only one or more of the indexes becomes distributed. Global computation on data associated with the remaining indexes may then be done without communication. However, when global communication is needed on all indexes a complete redistribution of the data is needed. In higher dimension \((d > 2)\) different mappings and subsequent redistribution techniques are possible. In this paper we present a general redistribution algorithm for data of dimension \(d\) mapped on to a processor array of dimension \(r < d\).

We show by a complexity analysis and numerical experiments that while using a 1D processor grid is the most efficient for modest number of processors, using 2D processor grid has better scalability and hence work best for higher number of processors.

1 Introduction

A common situation in parallel computation on multidimensional data is to perform calculations which involve all data along a specific dimension, while the calculations are completely decoupled along the other dimensions. For such problems it is possible to simplify the parallel algorithms by keeping at least one dimension local, while distributing the other dimensions among the processors. In most cases, however, it is necessary to perform calculations along all dimensions sequentially, which introduces the problem of redistributing the data among the processors. A prime example of this problem is the multidimensional fast Fourier transforms (FFT), where the full FFT can be calculated by performing 1D FFTs along all dimensions sequentially\(^1\)–\(^4\).

The standard way of dealing with problems of this type, is to distribute only one dimension of the data set at a given time. Redistribution can then be performed with the calls to the MPI function MPI_Alltoall. As other authors have pointed out\(^5\),\(^6\), this approach has limited scalability as the number of processors, \(P\), is limited by the smallest dimensional grid size \(P < \min N_i\). Furthermore, many massively multiprocessor computers utilizes special network topologies (i.e. toroidal), which this technique is not able to exploit.

In this paper we describe a generalized algorithm for data redistribution and its implementation using the MPI. We also analyse the computational complexity of the algorithm, and discuss in which cases it is favourable over the standard approach.

2 Problem Definition and Notation

Consider a \(d\)-dimensional dataset of size \(N_0 \times N_1 \times \cdots \times N_{d-1}\) which is mapped onto an \(r\)-dimensional processor array of size \(P_0 \times P_1 \times \cdots \times P_{r-1}\), \(1 \leq r < d\). The mapping is done by splitting the data set along \(r\) dimensions in equal pieces. We get different splittings depending on which dimensions we choose to split. There is of course \(\binom{d}{r}\)
possible mappings. In a computation, as exemplified by the d-dim FFT, the mapping will
to have to change during the computation.

For convenience we assume that \( N_i \mod P_j = 0 \) for all \( i = 0, \ldots, d - 1 \) and \( j = 0, \ldots, r - 1 \). This requirement is easy to overcome in practise, but it simplifies the notation
significantly in this paper. In practise we have some control over \( r \) and the \( P_j \)'s, while \( d \) and
\( N_i \) are defined by the problem. A straight forward way to deal with the \( N_i \mod P_j = 0 \)
requirement is to set \( P_1 = P_2 = \cdots = P_r = P \), pad the data array with zeroes to
satisfy \( N_i \mod P = 0 \). Another way (which we have used in our implementation) is to
modify the algorithm slightly so that it can work with different amounts of data residing
on each processor.

Let \( S = \{i_0, i_1, \ldots, i_r\} \) be an index set where \( 0 \leq i_j < d \) for \( j = 0, \ldots, r - 1 \). Then \( S_{\text{now}} \) denotes the dimensions which are distributed among the \( r \)-d processor array. A
dimension can only be distributed over one set of processors, which gives \( i_j \neq i_k \), if
\( j \neq k \). If we want to do computation on dimension \( k \), where \( k \in S_{\text{now}} \), a redistribution is
required.

Let \( S_{\text{next}} \) be a distribution where \( k \notin S_{\text{next}} \). The dimensions \( S_{\text{now}} \setminus S_{\text{next}} \) will be
distributed, while the dimensions \( S_{\text{next}} \setminus S_{\text{now}} \) will be gathered.

3 Algorithm

3.1 Redistribution of One Dimension

We will now assume that the difference between \( S_{\text{now}} \) and \( S_{\text{next}} \) is exactly one index, i.e.
the operation to be performed is an all-to-all along one dimension. For such an operation,
the processors can be organised in groups, where a processor only communicates with
other processors in the same group. For redistribution along different dimensions in the
processor grid, different groups will have to be formed. In general, one set of groups
will be formed for each dimension in the processor array. A processor \( P_\alpha \), where \( \alpha = (\alpha_0, \alpha_1, \ldots, \alpha_{r-1}) \), will be a part of the groups \( G^j_{\alpha_j} \), for \( j = 0, 1, \ldots, r - 1 \) (Figure 3.1).
For communication within one group, an algorithm similar to the standard implementation of all-to-all is used. Below is an implementation of this algorithm in simplified Python-like syntax. \texttt{inData} and \texttt{outData} are the input and output data arrays local to the current processor. \texttt{fullShape()} returns the shape of the global array, and \texttt{shape(x)} returns the local size the array \texttt{x}. \texttt{inDistrib} and \texttt{outDistrib} are the dimensions of the data set which is distributed at the beginning and end of the algorithm respectively. \texttt{groupSize} is the number of processors in the communication group.

```python
sendSize = fullShape(inDistr)/groupSizeecvSize = fullShape(outDistr)/groupSize
for i in range(groupSize):
    sendProc = (groupRank + i) % groupSize
    recvProc = (groupRank + groupSize - i) % groupSize
    sendSlice = shape(inData)
    sendStart = sendProc*sendSize
    sendEnd = (sendProc+1)*sendSize
    sendBlock = inData[sendSlice]
    recvSlice = shape(outData)
    recvStart = recvProc*recvSize
    recvEnd = (recvProc+1)*recvSize
    recvBlock = outData[recvSlice]
    irecv(fromProc, recvBlock)
    isend(toProc, sendBlock)
wait()
```

A test implementation of the above algorithm has been made in C++. To set up the processor groups we have used the Cartesian topology routines in MPI, and set up a communicator for each processor group. This is an easy way to set up a r-dimensional processor grid, and allows for optimised MPI implementations to exploit locality in the underlying network topology without user interaction.
For handling multidimensional data in C++ we have used the excellent blitz++ library\(^7\). Using blitz++ and MPI datatypes we have been able to hide the details of sending and receiving a strided hyperslab, which has simplified the implementation of the redistribution considerably.

### 3.2 Redistribution of Several Dimensions

If \( r = 1 \) or \( r = d - 1 \) the above algorithm cover all possibilities. However, for \( 1 < r < d - 1 \) one might pose the problem of how to redistribute more than one dimension at the time. One alternative is to apply the above algorithm several times. Assume that \( S\text{\textunderscore now} \rightarrow S\text{\textunderscore next} \) is a simple redistribution, that is if \( s_{i}\text{\textunderscore now} \neq s_{i}\text{\textunderscore next} \rightarrow s_{i}\text{\textunderscore next} \notin S\text{\textunderscore next} \) \( s_{i}\text{\textunderscore now} \notin S\text{\textunderscore now} \). For such a redistribution, the following simple algorithm will change distribution from \( S\text{\textunderscore now} \) to \( S\text{\textunderscore next} \).

```python
for i in range(r):
    if sNow[i] != sNext[i]:
        redistribute(data, groupIndex=i,
        inDistrib=sNow[i] outDistrib=sNext[i])
```

### 4 Lower Dimensional Projections

An alternative method for redistributing several dimensions simultaneously, is to map the data set to a lower dimensional data set, and distribute that array on a low dimensional processor array. In the extreme, one could map the data to a 2 dimensional data set and use a 1 dimensional processor array. However, if \( N_i \mod P_j \neq 0 \), one risks splitting up a dimension in the data set in an unpredicted way, and one must therefore take care to use the correct indexes in such situations.

In our previous work this method has been deployed with great success\(^8,9\). It is simple and in many cases efficient. It does however has restricted scalability. The reason for this is twofold. First there is a theoretical limit on the number of processors used which require us to have \( \min(N_1, N_2) \geq P \). The prime example of this is for \( d = 3 \) and the data array is of the size \( N \times N \times N \). In that case we will have \( N^3 \) data and only being able to use \( P \leq N \) processors.

Secondly when all processors are involved in the same all-to-all communication this is a more severe stress for the bisectional bandwidth than when they are divided into smaller groups, each group doing an internal all-to-all simultaneously. This effect is increasing with \( P \) and may also dependent on the network topology.

### 5 Complexity Analysis

This analysis is not meant as a detailed analysis, suitable for accurate prediction of the communication time. It only indicates the pro- and con’s of the different strategies, and their dependence on the actual parameters.

For simplicity we assume that \( N_0 = N_1 = \cdots = N_{d-1} = N \) and \( P_0 = P_1 = \cdots = P_{r-1} = P \) and \( N \mod P = 0 \). Then each processor will store \( N^d/P^r \) data items. The
algorithm of Section 3.1 will send pieces of \( N^d / P^{r+1} \) data in each of the \( P - 1 \) steps. With latency \( t_s \) and reciprocal bandwidth \( t_w \) the complexity of this algorithm becomes:

\[
(P - 1)(t_s + t_w \frac{N^d}{P^{r+1}})
\]

(5.1)

Initially \( d - r \) directions are not splitted, and local work in these dimensions can be carried out. Then, in each step, the algorithm provides one new direction for local computation. Thus to get data associated with the remaining \( r \) indexes to appear locally, the redistribution algorithm needs to be repeated \( r \) times. Thus for a complete sweep of local computation in all \( d \) direction the total work becomes:

\[
W_1 = r(P - 1)(t_s + t_w \frac{N^d}{P^{r+1}}) \approx r(P t_s + t_w \frac{N^d}{P^r})
\]

(5.2)

Alternatively we might map the data to a 2D data set and the processor array to a 1D array of \( P^r \) processors. Each processors will still have the same total amount of data. However when applying our algorithm the piece of data to be sent is now only \( N^d / P^{2r} \) and the number of loop iterations becomes \( P^r - 1 \). However the upside is that only one redistribution is needed for a full sweep. The complexity becomes

\[
W_2 = (P^r - 1)(t_s + t_w \frac{N^d}{P^{2r}}) \approx P^r t_s + t_w \frac{N^d}{P^r}
\]

(5.3)

The amount of data transferred is less in the second case, but the number of start-ups might be much higher. We notice that the problem parameters \((N, d)\), system parameters related to processors configuration \((P, r)\) as well as those related to communication network \((t_s, t_w)\) all play a role in determining the fastest communication mode.

We would like to caution about using the above formula for predicting the fastest way of organising your computation. Additional factors such as network topology as well as system and communication software will also be of importance. Note also that when arranging the processors as a 1D-array one more easily get significant load imbalance when the \( N \text{ mod } P = 0 \) condition is violated.

6 Numerical Experiments

We have performed several numerical experiments to determine the properties of the redistribution algorithms. The implementation has been tested on different platforms with different network topologies to determine if we are able to effectively exploit special topologies more efficiently than with the previously used algorithm. For measuring efficiency, we have used the wall clock time it takes to complete two complete sweeps of redistribution. That is \( 2r + 1 \) redistributions with the algorithm presented in 3.1.

In order to minimise any effect of function calls to the timer, the process has been repeated 10 times and the averaged time is recorded. Each of these tests was run 10 times and to minimise the effect background processes in the operating system, we report the minimal time. Furthermore the complete test was run twice with several days separation in order to detect if any of the tests were influenced by other jobs running on the computer.

The main performance test is to redistribute a 3D data set of size \( N \times N \times N = N^3 \), on processor grid of \( P \times P = P^2 \) processors. The test is run for several values of \( P \) and \( N \), comparing the time it takes to redistribute the data set on a 2D vs. a 1D processor grid.
Platform 1 | Platform 2
---|---
Provider | NTNU | Argonne National Lab
System | IBM p575+ | IBM Blue Gene/L
CPU | IBM Power5+ 1.9 GHz | PowerPC 440 700 MHz
Cores pr. node | 16 | 2
Total nodes | 56 | 1024
Interconnect | HPS | Blue Gene/L Torus
Topology | Fat tree | 3D Torus

Table 1. The platforms used for testing the redistribution algorithms

Figure 3. Platform 1 redistribution time as a function of number of processors $P^2$ plotted for different grid sizes. The left panel shows wall time using a 1D processor grid, and the right panel shows the results for a 2D processor grid.

To our disposal we have had 2 different platforms. The key hardware features of these are described in Table 1.

Platform 1 uses a high speed HPS interconnect. We have not detected any significant dependence on how the processor grid is mapped onto the physical processors, suggesting that the network topology is not an important factor for this platform. In Figure 3 the results for the main performance test is shown both for a 1D and a 2D processor grid. As expected, for few processors, the 1D processor grid is superior. However, the 2D configuration gives better scaling and eventually becomes faster than the 1D configuration. The crossover-point appears to be $P^r \approx N/4$. The jagged form of the curves in left figure is a consequence of the fact that when the data cannot be distributed evenly among the processors, the performance will be dictated by the processor with most data rather than any of the latency effects described in Section 5. The case $N = 256$ and $r = 1$ illustrate this explanation. Here we observe local minima for $P = 128$ and $P = 256$.

Platform 2 is an IBM Blue Gene/L system, and uses a special 3D toroidal network for bulk data transfer. The network topology is interesting as it should fit well to a logical 2D configuration of the processors, and by carefully mapping the logical 2D processor grid onto the physical 2D toroidal node grid, one might expect some performance improvement compared to the 1D processor grid. In order to test this hypothesis, we have run four test cases. One reference test using a 1D processor grid, and three tests with a 2D processor grid. Using the 2D processor grid, we have varied the mapping of the processor grid on the
scientific results

(a) Platform 2 redistribution time as a function of number of processors $P^r$ plotted for grid size $N = 256$ for a 1D and 2D processor grid.

(b) Relative performance of a 2D processor grid compared to 1D. $N = P^r$ for different values of $N$. Values larger than 1 means the 2D processor grid is faster.

Figure 4. Numerical results from Platform 2

physical nodes in order to detect if the performance is dependent on the topology. However, through all our tests, we have not been able to observe any difference in performance between the three 2D processor grids used.

Figure 4 shows results from Platform 2. In essence, the results are similar results to that of Platform 1. For a given $N$, a 1D processor grid performs best for few processors, but a 2D processor grid scales better. Interestingly, we observe that for $P^r = N$ (Figure 4(b) (Which is the highest possible processor count for the 1D processor grid) the 2D processor grid performs better, even though twice the amount of data is being transferred. This means that not only does the 2D processor grid allow one to use more processors, it also enables more efficient utilization of the the network, most likely due to larger blocks of data being sent at each step in the redistribution. The relative decrease in efficiency of the 2D processor grid seen for increasing values of $N$ and $P^r$, can be explained from the fact that block blocksize increases cubically with $N$, and decreases linearly with $P^r$. As the blocksize increase, we expect the startup effects for the 1D processor grid to decrease. In virtual processor mode, the performance of the 2D processor grid suffers most, because the bandwidth of each process is effectively halved compared to co processor mode.

7 Concluding Remarks

We have designed and implemented and algorithm for redistribution of multidimensional arrays. Complexity analysis and experiments show that our algorithm has better scalability than the standard algorithm which view the processors organised as a 1D-grid. As the current trend in design of HPC-system is that number of processors increases much faster than their individual speed, sustained petaflop computing can only be achieved through highly scalable algorithms.

Our implementation is a generalization of the standard algorithm, making it easy to change the dimension of the processor grid at runtime depending on the size of the problem and the number of processors available at runtime.
Acknowledgement

We gratefully acknowledge the support of NOTUR, The Norwegian HPC-project, for access to their IBM p575+ at NTNU, and to Argonne National Laboratory for access to their Blue Gene/L System.

References
