Aerospace, Information Technology, and Communications Seminar (SEMASITT'03).
May 7, 2003. Johore Bahru, Malaysia.

DISTRIBUTED DATA PARTITIONING INTERFACE FOR HOMOGENEOUS CLUSTERS IN PROTEIN SECONDARY STRUCTURE PREDICTION

Satya Nanda Vel Arjunan[1], Safaai Deris[1], Rosli Md Illias[2], Mohd Saberi Mohamad[1]
[1]Department of Software Engineering, Faculty of Computer Science and Information Systems
Universiti Teknologi Malaysia, 81310 Skudai, Johore, Malaysia
Tel: 07-553243, E-mail : Safaai@fsksm.utm.my, satyanandavel@hotmail.com, berie_ext2@lycos.com
[2] Department of Bioprocess Engineering, Faculty of Chemical and Natural Resources Engineering
Email : r-rosli@utm.my

Abstract : *The effort required to write efficient parallel programmes or to parallelize existing sequential algorithms remains daunting. This is true even for regularly structured problems. Much of the work takes place when partitioning and distributing workloads over processors in a distributed computing environment. To alleviate this task, we present a data parallel interface called Distributed Data Partitioning Interface (DDPI). Its simple interface permits parallel implementation even by users with little understanding of parallel computing technicalities. In this work we evaluate the performance of DDPI in several computationally intensive problems such as matrix multiplication, data clustering and neural network batch training. Through these problems, we demonstrate that it is possible to achieve almost ideal speedups when they are parallelized with DDPI.*

Keywords: *data partitioning interface, parallel k-means, parallel k-harmonic means, parallel batch training*

1. Introduction

Although commodity clusters and parallel computers are becoming widespread now, the effort required to write efficient parallel programmes or to parallelize existing algorithms remains daunting. This is true even for regularly structured problems. Much of the work takes place when partitioning and distributing workloads over processors in the distributed computing environment. There are two main approaches to relieve this effort off of the user: automatic parallelizing compilers (Agarwal et al., 1995; Prechelt and Hänßgen, 2002) and workload distributing libraries or tools (Hendrickson and Leland, 1994; Carpenter et al., 1997; Karypis and Kumar, 1998; Boniface et al., 1999; Chen and Taylor, 2002). Unfortunately in the former, even though it is a well-established research field, the fundamental issue of optimal partitioning remains unsolved. On the other hand, for regularly structured problems, the libraries and tools appear to be either overkills (Hendrickson and Leland, 1994; Carpenter et al., 1997; Karypis and Kumar, 1998) or too specialized (Boniface et al., 1999; Chen and Taylor, 2002), and therefore are substantially cumbersome when they are used to parallelize existing sequential algorithms. Certainly the problem at hand should be the center of focus instead of being concerned with the intricacies of parallel programming. Moreover, even for experienced parallel programmers, the development of good parallel implementations with these tools is still more tedious than writing efficient serial programmes. For these reasons, we are motivated to look at a general solution and derive the following requirements in this work:

(i) *Low learning threshold*. Ideally, in order to reduce the effort required for parallelization, it is not expected of the user to acquire additional skills pertaining to parallelism nor to learn extraneous language constructs. Hence, the low level parallelization details should be hidden from the user.

(ii) *Simple implementation*. The overall structure of the sequential program should be preserved such that the user would be able to focus on the original algorithm flow of the problem even after parallelization.

(iii) *Portability*. The system should be implemented in a widely accepted and standard programming language to ensure portability to all target platforms and machines. For better portability, assumptions about the distributed computing environment's specific network topology should be avoided. Nonetheless, the system should cater for homogeneous processors and networks since they are more commonly available.

(iv) *Maintainability*. Although initially the solution may be intended for regularly structured problems, it should however have the facility to be extended for more complicated problems.

(v) *Effectual*. The system's performance should be comparable to more specialized and sophisticated implementations.

It was found that an interface using the data parallel approach fulfills the above requirements. The design and evaluation of the interface, referred to as the Distributed Data Partitioning Interface (DDPI), will be presented in the following sections.

2. Scope and Limitations

DDPI is designed to parallelize problems dealing with regularly distributed data or iterative in nature. Although it can still be used with irregularly structured problems, the performance may not be optimal because it may bring about communication overhead to distribute workloads evenly. DDPI is targeted for users with little or no prior experience in parallel programming. It is implemented in an object oriented fashion in C++ and utilizes the Message Passing Interface (MPI) (MPI Forum, 1998). Even though one of the objectives is to avoid learning additional language constructs, it is still reasonable to expect the user to know the basic MPI functions since they are also implemented in both C and C++. This tool, which addresses the problem of data partitioning, assumes that a single processor with sufficient memory is available to partition the complete data.

3. Design of DDPI

Table 1 lists the description of symbols used in this work. Figure 1 displays the three major parallelization steps with the DDPI programming interface. In order to distribute the computational workload, DDPI provides a small set of routines to spread data across the processes. The data, which can be either locally or globally accessible, is contained in a two-dimensional matrix constructor. It is partitioned according to one of several available techniques in DDPI and shipped to the processes in the process grid. Each process will then be able to perform computations concurrently using their local data. When required, the processes can communicate with each other using existing MPI functions. During the computational procedure, there will be situations in which information pertaining to the distributed data is needed. DDPI provides a convenient access to this information through several essential routines. Finally, the local data can also be gathered and reduced for global use with MPI or DDPI routines. Specific details of the above steps will be explored in the following sections.

Table 1: Description of Symbols.

Symbol	Description	Symbol	Description
nProcs	total number of processes	*lclCols*	local columns
prRows	total process rows	*rowBlk*	row block size
prCols	total process columns	*colBlk*	column block size
prRow	process row coordinate	*startPrRow*	starting process row
prCol	process column coordinate	*startPrCol*	starting process column
gblRows	global rows	*nSamples*	number of data samples
gblCols	global columns	*nDimension*	dimension size
lclRows	local rows	*contxt*	context of the process grid

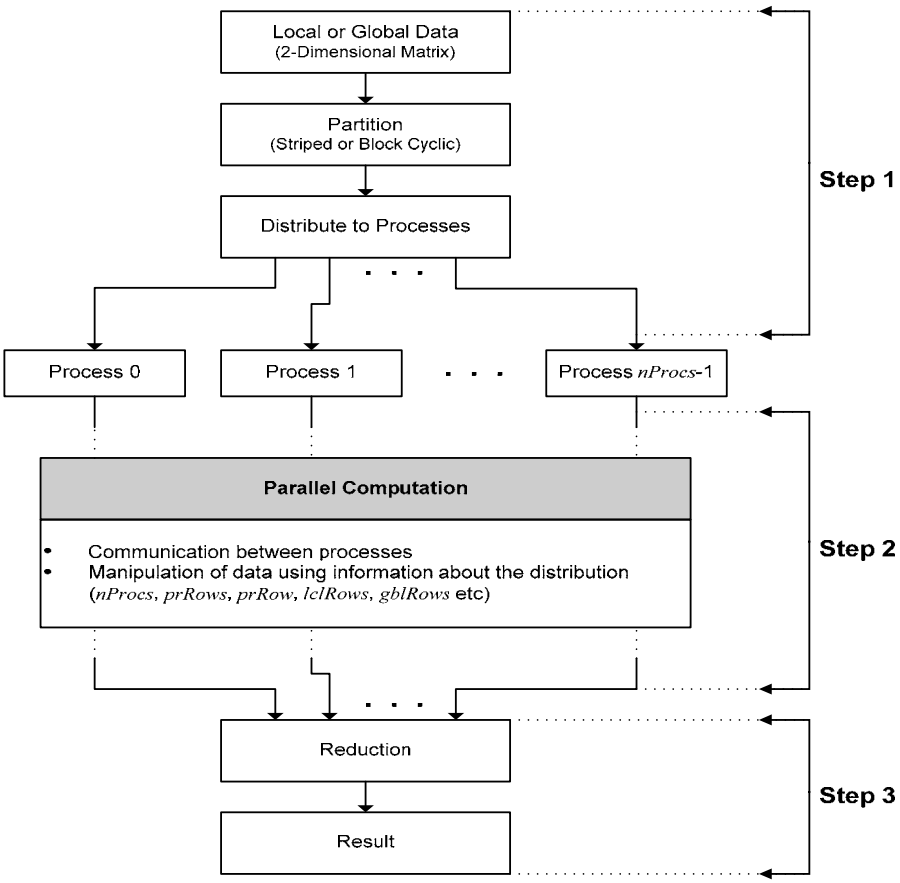


Fig 1: The three main parallelization steps of DDPI

3.1 Step 1: Initializing, Partitioning and Distributing Data

The first parallelization step with DDPI relieves most of the effort from the user by automatically partitioning and distributing a given set of computational workload to the processors. The user begins the parallelization procedure with a one time initialization step of MPI and DDPI libraries:

```
MPI_Init();
DDPI_Init();
```

This is followed by allocating the data using the DDPI's Matrix object constructor

```
Matrix::Matrix(i,j,data);
```

where, *i* and *j* are the row and column sizes of the source data respectively. If the source data is locally owned, it should belong to the root process (process 0) because DDPI will distribute the data to other processes from the root process. The root process can be verified using the MPI function, MPI_Comm_rank which returns the process label of the calling process. The data can now be distributed by issuing the DDPI scatter command:

```
Matrix::scatter(partition);
```

In the above command, *partition* represents one of DDPI's three identifiers for the partitioning technique that will be used to distribute the data. Table 2 lists the identifiers and their corresponding partitioning techniques. The three methods are commonly used in general parallel computing applications.

Table 2: DDPI identifiers for data partitioning techniques.

Identifier	Partitioning Technique
ROW	Row Striped
COL	Column Striped
UNI	Block Cyclic

The data matrix is partitioned by mapping blocks of rows of size *rowBlk* and blocks of columns of size *colBlk* to the process grid. The partitioning techniques can be classified based on the block sizes and the mesh of the process grid. In the row and column striped partitioning techniques, the data matrix is divided into groups of complete rows or columns (Figure 2). Each process is allocated these contiguous rows or columns as workloads. DDPI

employs the following functions to determine the block sizes:

$$rowBlk = \text{int}\left(\frac{gblRows + prRows - 1}{prRows}\right) \quad (1)$$

$$colBlk = \text{int}\left(\frac{gblCols + prCols - 1}{prCols}\right) \quad (2)$$

In these functions, *gblRows* and *gblCols* are the total number of rows and columns in the undistributed data matrix respectively. The block sizes can be computed using the process row and column sizes listed in Table 3.

Table 3: Process grid meshes for striped partitioning techniques.

Process Grid	Row Striped	Column Striped
process rows (*prRows*)	*nProcs*	1
process columns (*prCols*)	1	*nProcs*

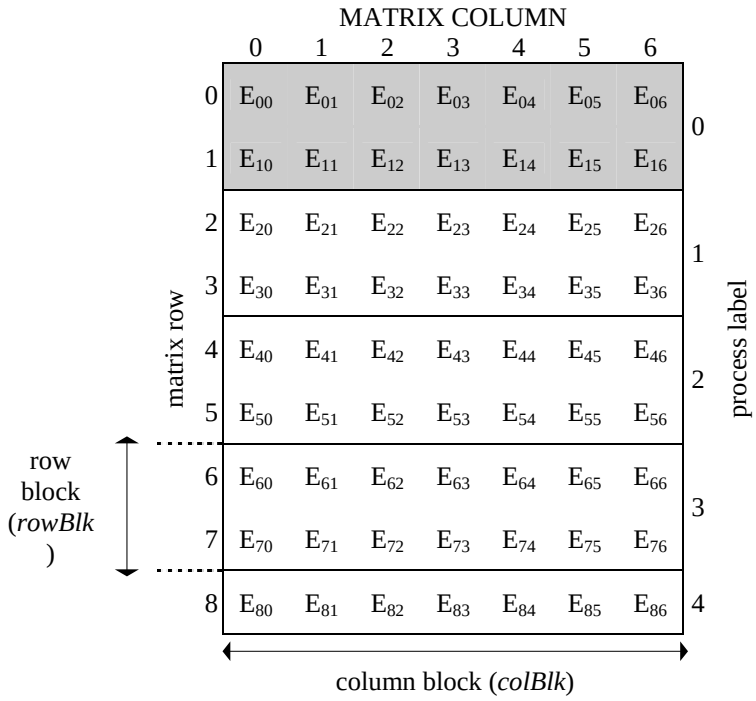
An example of each scheme is displayed in Figure 2. The examples illustrate the partitioned layouts of a data matrix *E* of size 9×7 that is distributed over 6 processes. It can be observed from both of the examples that not all of the 6 processes receive the distributed matrix. Nevertheless, there is a partitioning strategy which can distribute the same data matrix *E* to all the processes yet achieve better load balancing. This technique, called checkerboard block cyclic partitioning, is illustrated in Figure 3.

The checkerboard block cyclic partitioning scheme, which has also been incorporated in the High Performance Fortran standard (High Performance Fortran Forum, 1997), distributes blocks of rows and columns among the processes in a wraparound manner. Unlike the striping techniques, the block cyclic partitioning technique does not have rigid process grid meshes. Furthermore, the row and columns blocks can be of any size as well. Small block sizes in this scheme will provide better load balancing but at the cost of frequent interprocessor communications. Conversely, large block sizes will reduce the communication latency but may cause load imbalance among the processes. With DDPI, by default the block sizes are computed using the following conventions:

$$rowBlk = \text{int}\left(\frac{gblRows}{prRows}\right) \quad (3)$$

$$colBlk = \text{int}\left(\frac{gblCols}{prCols}\right) \quad (4)$$

The sizes however, can be changed to suit the computational problem. In Figure 3 the data matrix *E* is partitioned into 4×2 blocks and mapped onto a 2×3 row-major order process grid. The largest workload assigned to a process is a 15-element matrix owned by the root process (Figure 3 (a)). Comparing this workload with the largest one from the example in Figure 3 (b) (18-element matrix), it is evident that the block cyclic partitioning scheme achieves better load balancing than the column striped partitioning technique. It should be noted however, as it will be demonstrated in the experimental results section, in some problems the striping techniques perform better than the checkerboard block cyclic scheme.

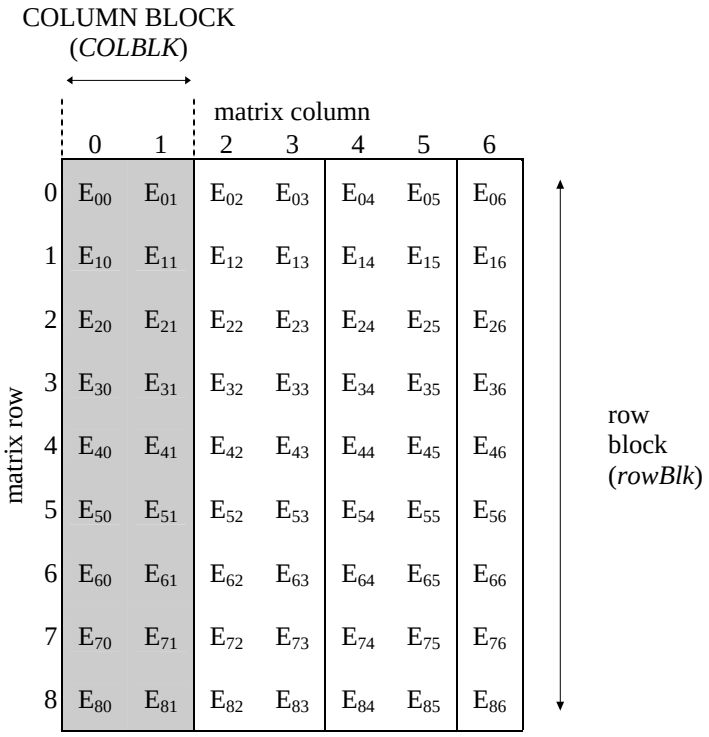


Parameter	Size
gblRows	9
gblCols	7
nProcs	6
prRows	6
prCols	1
rowBlk	2
colBlk	7

$$rowBlk = \text{int}\left(\frac{9+6-1}{6}\right)$$
$$= 2$$
$$colBlk = \text{int}\left(\frac{7+1-1}{1}\right)$$
$$= 7$$

Note: Shaded block indicates the workload assigned to the root process

Fig 2 (a) : Example of a row striped partitioning distribution

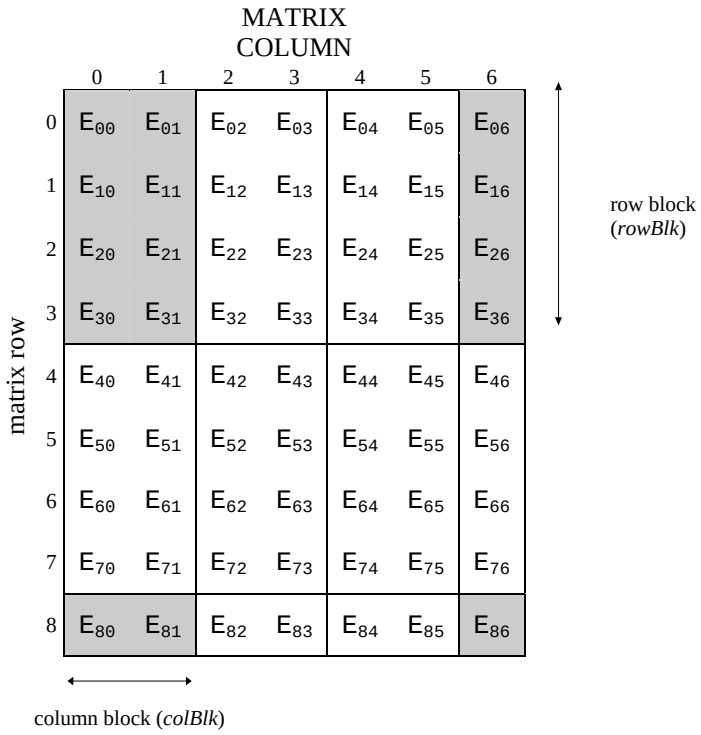


Parameter	Size
gblRows	9
gblCols	7
nProcs	6
prRows	1
prCols	6
rowBlk	9
colBlk	2

$$rowBlk = \text{int}\left(\frac{9+1-1}{1}\right)$$
$$= 9$$
$$colBlk = \text{int}\left(\frac{7+6-1}{6}\right)$$
$$= 2$$

Note:
Shaded block indicates the workload assigned to the root process.

Figure 2 (b): Example of a column striped partitioning distribution.

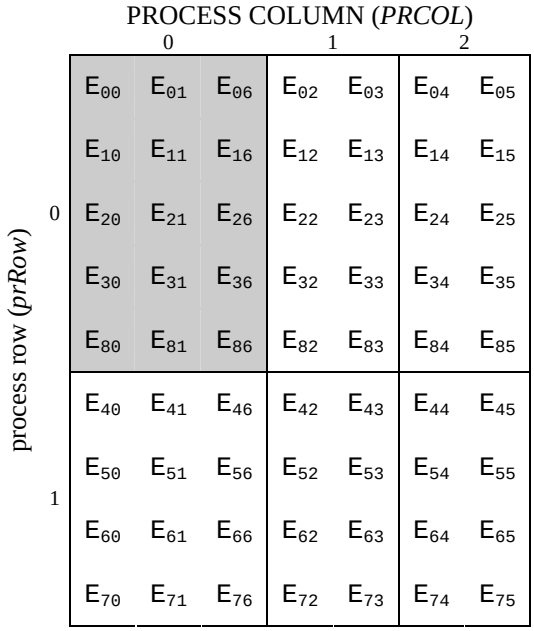


Note:
A 9×7 matrix partitioned into 4×2 blocks. Shaded blocks are sent to the root process of a 2×3 process grid.

Parameter	Size
gblRows	9
gblCols	7
nProcs	6
prRows	2
prCols	3
rowBlk	4
colBlk	2

$$rowBlk = \text{int}\left(\frac{9}{2}\right)$$
$$= 4$$
$$colBlk = \text{int}\left(\frac{7}{3}\right)$$
$$= 2$$

Figure 3 (a): Example layout of the checkerboard partitioning scheme.



Note:
A 9×7 matrix partitioned into 4×2 blocks partition mapped onto 2×3 row-major ordered process grid. Shaded blocks are on the root process.

Figure 3 (b): Distributed matrix of the checkerboard partitioning example.

3.2 Step 2: Computing concurrently using Distributed Data

Once the data is partitioned and distributed, each process can use its local data matrix to perform computations. Nevertheless, each process will require essential information pertaining to the distributed data such as the local rows and columns, the corresponding global matrix cell of its local cell, its location on the process grid, etc. DDPI accommodates this by providing several routines that return such information. Table 4 lists the summaries of available DDPI routines. Although these routines provide complete information pertaining to the distributed data, fundamental message passing functions may still be needed for more elaborate parallel programming. These functions are available from MPI (Table 5).

Table 4: Summary of DDPI routines.

Routine	Function
getGblRows getGblCols	Returns the global rows/columns, *gblRows*/*gblCols* of the partitioned matrix.
getLclRows getLclCols	Returns the local rows/columns, *lclRows*/*lclCols* of the partitioned matrix.
gbl2lclRow gbl2lclCol	Converts a global row/column into its corresponding local row/column and returns the process row/column, *prRow*/*prCol* in which the global row/column is located. Another overloaded version of these routines returns a predefined identifier, OUTSIDE if the global row/column to be converted resides out of the local matrix.
lcl2gblRow lcl2gblCol	Converts the process' local row/column into its corresponding global row/column.
gbl2lcl	Converts a global coordinate (*gblRow*,*gblCol*) of a matrix cell into its corresponding local coordinate (*lclRow*,*lclCol*) and returns the coordinate of the process (*prRow*,*prCol*) that locally owns the matrix cell.
getContxt	Returns the context, *contxt* of the process grid in which the matrix is distributed. The *contxt* serves as a reference for the unique process grid and the partitioning technique used by the processes. Two sets of

	data can be distributed in an identical fashion by using the context of one of them as the partitioning technique identifier for the scatter method of the other: Matrix::scatter(contxt);
descriptor	A one-dimensional array containing information about the distributed matrix: *contxt*, *gblRows*, *gblCols*, *rowBlk*, *colBlk*, *startPrRow*, *startPrCol* and *lclRows*. Analogous to the descriptor used by the ScaLAPACK parallel linear algebra library (Blackford et al., 1997).

Table 5: Summary of MPI routines.

Routine	Function
MPI_Send	Sends data from the calling process to another process identified by the process label.
MPI_Receive	Inverse operation of MPI_Send. Data is received by the calling process from another process identified by the process label.
MPI_Scatter	Distributes distinct uniform-sized blocks of data in an array from the calling process to distinct members of a process group. It is a primitive form of the DDPI's scatter method; it neither partitions disingenuously nor maps the data onto a process grid.
MPI_Gather	Inverse operation of MPI_Scatter. Collects distinct uniform-sized blocks of data from all members of a process group into an array of the calling process. It is a primitive form of the DDPI's gather method; it does not take into account the partitioning technique or the process grid.
MPI_Bcast	Sends local data from the root process to all members of a process group.
MPI_Reduce	Reduces data elements from all members of a process group into a single value and places the result on the root process.
MPI_Allreduce	Similar to MPI_Reduce but the reduced result is distributed to all members of a process group.

3.3 Step 3: Assembling Local Computational Results

At the completion of local computations, the processes may need to synchronize, gather and reduce their local computation outcomes to reflect the overall result of the parallel computation. To synchronize the processes, the function MPI_Barrier can be used. The data gathering procedure can be as simple as assembling the local data of processes into a single array while the reduction process may include operations such as multiplication and summation. For the former, MPI provides a data assembler routine called MPI_Gather. Alternatively, DDPI provides an advanced version of this function which is also the inverse operation of its scatter routine:

```
Matrix::gather();
```

The routine assembles the previously partitioned and distributed data matrix into its original form and places it on the root process. The reduction process on the other hand can be executed using two of the MPI reduction routines listed in Table 5 (MPI_Reduce and MPI_Allreduce). Finally, the resources allocated for the parallel computation can be released and the computation can be terminated by issuing the exit commands of both MPI and DDPI libraries:

```
DDPI_Exit();
MPI_Finalize();
```

The presented three major steps of parallelization are a simple outline of the parallelization strategy with DDPI. They can be extended for more complex parallel computing solutions such as in cases with multiple sets of distributed data, multiple types of partitioning techniques and multiple topologies of process grids.

4. Experimental Results and Discussion

In this section, parallelization results of three problems, namely matrix multiplication, data clustering and neural network batch training are presented. The experiments were conducted on a Linux cluster consisting of two computers with each having two 1.6 GHz Athlon SMP CPUs interconnected by a 1 Gbps gigabit ethernet switch.

The computers have 2 GB and 1 GB of memory respectively. The cluster's performance reached 6.435 Gflops when measured using the Linpack benchmark (Dongarra, 2002) with Basic Linear Algebra Subprograms (BLAS) library (Dongarra et al., 1990) optimized by Automatically Tuned Linear Algebra Software (ATLAS) (Whaley et al., 2001). Its maximum performance could not be measured because it was limited by the amount of physical memory.

4.1 Concurrent Matrix Multiplication

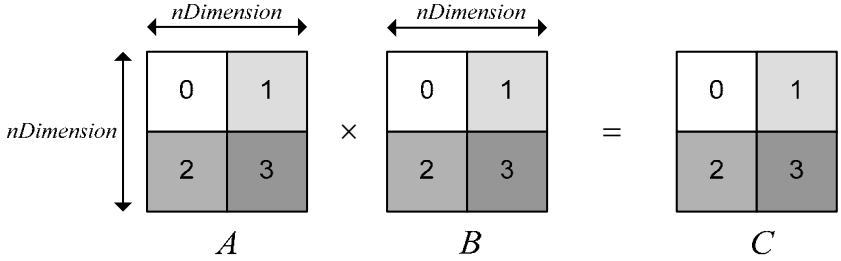
Conventional dense matrix multiplication has the computational complexity of

$$O\left(nDimension^3\right)$$

where *nDimension* is the dimension of the square matrices. With parallelization, it is possible to reduce the complexity to (Comino and Narasimhan, 2002):

$$O\left(\frac{nDimension^3}{nProcs}\right)$$

Even though parallelization of dense matrix multiplication algorithm has been studied quite extensively (Kumar et al., 1994), due to its prominent role in scientific computing applications, new algorithms and implementations are also continuously being developed (Valsalam and Skjellum, 2002; Whaley et al., 2001; Aberdeen and Baxter, 2001; Gunnels et al., 2001; Chatterjee et al., 1999). The PBLAS (Parallel Basic Linear Algebra Subprograms) library, a subset of the ScaLAPACK (Scalable Linear Algebra Package) library (Blackford et al., 1997), is widely used to compute matrix multiplication in a parallel computing environment. The product of two real matrices is computed using its pdgemm (parallel double precision generalized matrix multiply) routine through either its Fortran or C/C++ interfaces. In order to distribute the computational workload to the processes, PBLAS requires the two matrices to be partitioned beforehand using the block cyclic partitioning technique. Figure 4 depicts the block cyclic partitioning layout for the matrices with a process running on each of



Note:The matrices are mapped onto 2×2 row-major order process grid. The number in each block indicates the process label.

Fig 4: Parallel matrix multiplication with block cyclic partitioned matrices.

the four processors in the cluster. In this illustration, the multiplicands *A* and *B* are square matrices of dimensions *nDimension*×*nDimension* that produce a solution matrix *C* of the same size.

For the Fortran interface, users can utilize the existing directives in the High Performance Fortran to partition and distribute the workload prior to calling the pdgemm routine. However, for the users employing the C/C++ interface, they need to separately set up the data partitioning and spreading procedures. This would impose a significant amount of effort on the users without parallel programming

```
1   // function executes C <-- A×B using PBLAS pdgemm routine with DDPI interface
2   void execute_pdgemm(int nDimension) { // for square matrix, rows=cols=nDimension
3      Matrix *A;
4      Matrix *B;
5      Matrix *C;
6      int    rows = nDimension;
7      int    cols = nDimension;
8
9      MPI_Init();
10     DDPI_Init();
11     A = new Matrix(rows,cols);
12     B = new Matrix(rows,cols);
13     C = new Matrix(rows,cols);
14     A->scatter(UNI); // partition multiplicand A block cyclically and distribute
15     B->scatter(UNI); // partition multiplicand B block cyclically and distribute
16     C->scatter(UNI); // partition product C block cyclically and distribute
17
18     // convert C <-- alpha×A×B+beta×C of PBLAS pdgemm routine into C <-- A×B
19     char   transposeA = 'N'; // set matrix A as not transposed
20     char   transposeB = 'N'; // set matrix B as not transposed
21     int    p1 = 1; // increment index for traversing the elements in the matrices
22     double alpha = 1.0;
23     double beta = 0.0;
24
25     // execute C <-- A×B using PBLAS pdgemm routine
26     pdgemm_(&transposeA, &transposeB, rows, cols, rows, alpha,
27           A->data, p1, p1, A->descriptor,
28           B->data, p1, p1, B->descriptor, beta,
29           C->data, p1, p1, C->descriptor);
30
31     C->gather(); // local results are assembled to form complete product matrix C
32     delete A;
33     delete B;
34     delete C;
35     DDPI_Exit();
36     MPI_Finalize();
37  }
```

Fig 5: Function execute_pdgemm that partitions, distributes and multiplies using PBLAS pdgemm routine and DDPI interface.

expertise. In fact, this drawback in ScaLAPACK has motivated the development of a similar library but with a simpler MPI like interface called PLAPACK (van de Geijn et al., 1997). Unfortunately, PLAPACK does not have the amount of user base and influence which ScaLAPACK has in high performance computing applications. DDPI addresses this requirement in ScaLAPACK elegantly with its simple interface as shown in Figure 5. Furthermore, the matrix descriptor used in DDPI is also fully compatible with the one required by ScaLAPACK. Therefore, over 100 remaining double precision routines in ScaLAPACK can also use DDPI as the interface to partition and distribute data across processes.

The pdgemm routine with DDPI interface was experimented with four different dimensions of multiplicands. Figure 6 displays the results of the execution time when multiplying the matrices with varying number of processors. The execution time indicates the

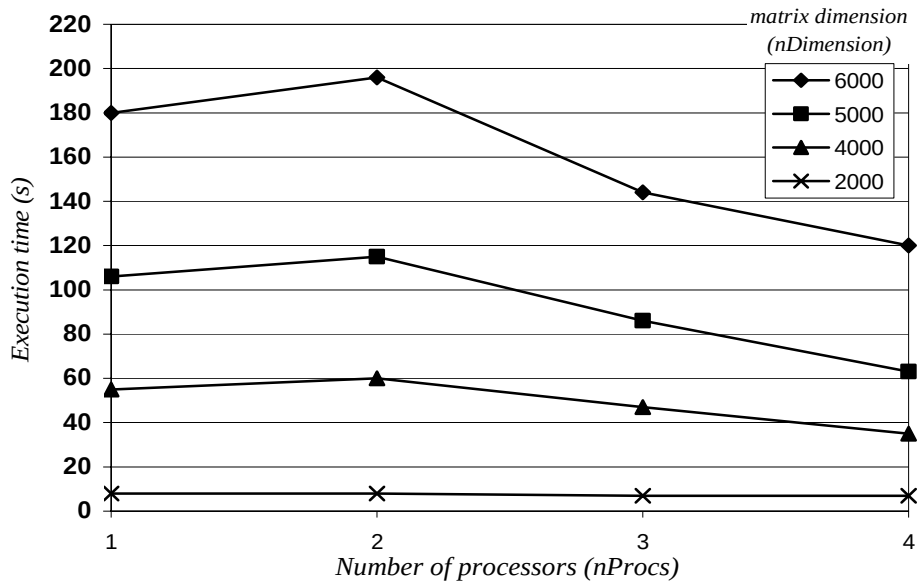
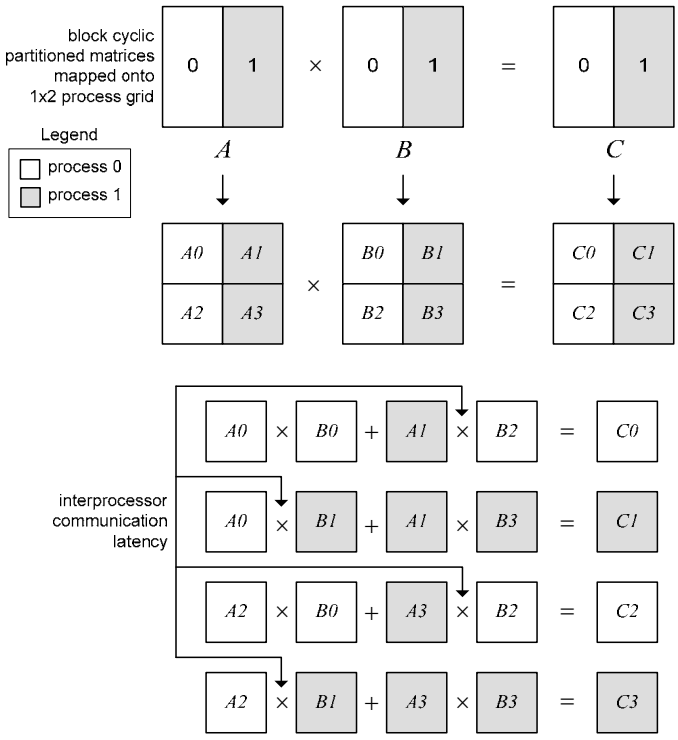


Fig 6: Execution times of parallel matrix multiplication using pdgemm and DDPI interface.

time spent to partition, distribute, compute and assemble the matrices. Generally, it can be observed that the computation time decreases when the processors are added. However, there is an unanticipated increase in the execution time when two processors are utilized as compared to only one. This is possibly due to the communication latency when multiplying local data across the two processors in the block cyclic partition as indicated in Figure 7.



Note: The matrices are mapped onto a 1×2 row-major order process grid.

Fig 7: Communication latency when multiplying block cyclic partitioned matrices

To verify this conjecture, another set of experiments is conducted with the row striped partitioning technique that eliminates the need to communicate at the indicated sections. This partitioning layout, illustrated in Figure 8, shows that it is unnecessary for further interprocessor communication once the data has been distributed to the processors.

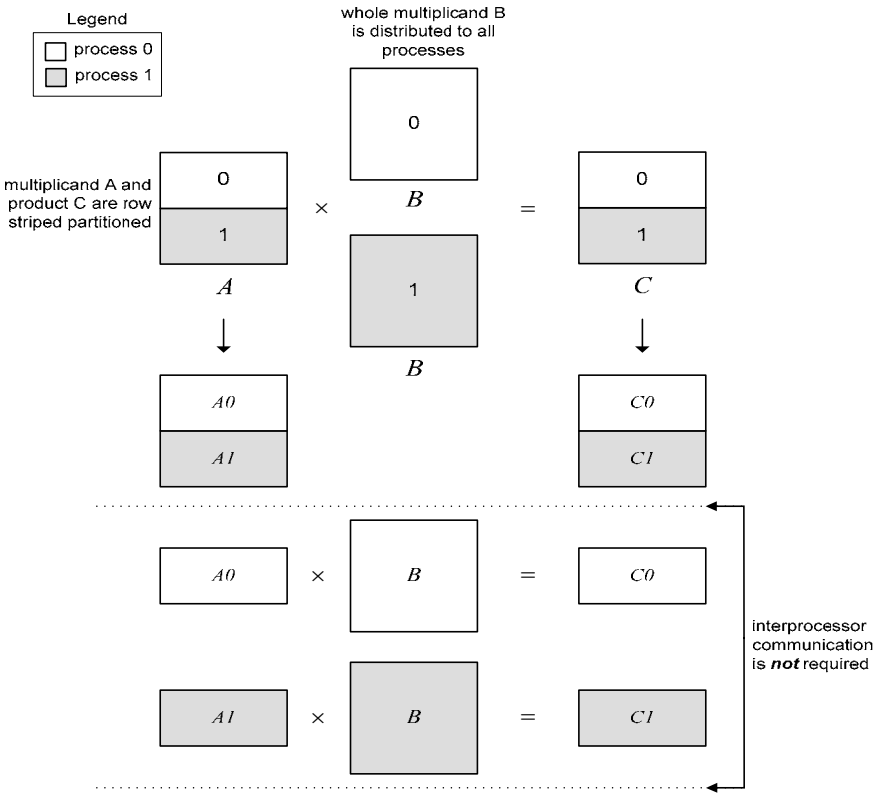


Fig 8: Concurrent multiplication of a row striped matrix with a whole matrix without interprocessor communications.

In the proposed technique, multiplicands *A* and *B* have *nProcs*×1 and 1×1 process grids respectively. Apparently, the PBLAS library prohibits simultaneous operations on matrices having different process grids (conflicting contexts). As a consequence, the experiment is conducted using the sequential version of the pdgemm routine, dgemm (double precision generalized matrix multiply), which is available from the BLAS library. Figure 9 illustrates the simplicity of DDPI in parallelizing even such sequential matrix multiplication algorithm.

The timing results of the proposed technique are illustrated in Figure 10. As expected, the execution time decreased by almost twofold when the number of processors increased from one to two. This outcome clearly exhibits the influence of the communication latency in the overall computation speed. However, the reduction in the execution speed is not linearly proportionate to the addition of processors because the time spent to transfer the whole matrix *B* to each processor also increases with the number of processors. It can also be seen that when the size of matrix is small (*nDimension* = 2000), the speedup (ratio of the execution time on 1 processor to the execution time on *nProcs* processors) is almost negligible. This is due to the small granularity of the problem, in which, the speedups gained from the higher number of processors are negated by the amount of time spent to transfer the data to each processor.

```
1   // function executes C <-- A×B using BLAS
2   dgemm routine with DDPI interface
3   void execute_dgemm(int nDimension) { // for
4   square matrix, rows=cols=nDimension
5      Matrix *A;
6      Matrix *B;
7      Matrix *C;
8      int    rows = nDimension;
9      int    cols = nDimension;
10
11     MPI_Init();
12     DDPI_Init();
13     A = new Matrix(rows,cols);
14     B = new Matrix(rows,cols);
15     C = new Matrix(rows,cols);
16     A->scatter(ROW); // partition multiplicand
17  A using row striping and distribute
18     B->scatter(WHOLE); // distribute whole
19  matrix B
20     C->scatter(A->getContxt()); // partition and
21  distribute product C indentical to A
22
23     // convert C <-- alpha×A×B+beta×C of
24  BLAS dgemm routine into C <-- A×B
25     char   transposeA = 'N'; // set matrix A as
26  not transposed
```

```
27      char    transposeB = 'N'; // set matrix B as
28    not transposed
29      int     p1 = 1; // increment index for
30    traversing the elements in the matrices
31      double  alpha = 1.0;
32      double  beta = 0.0;
33
34      // execute C <-- A×B using BLAS dgemm
35    routine
36      dgemm_(transposeA, transposeB, C-
37    >getLclRows(), C->getLclCols(),
38              A->getLclCols(), alpha,
                A->data, C->getLclRows(),
                B->data, C->getLclCols(), beta,
                C->data, C->getLclRows());

      C->gather(); // local results are assembled
    to form complete product matrix C
      delete A;
      delete B;
      delete C;
      DDPI_Exit();
      MPI_Finalize();
    }
```

Fig 9: Function execute_dgemm that partitions, distributes and multiplies using BLAS dgemm routine and DDPI interface.

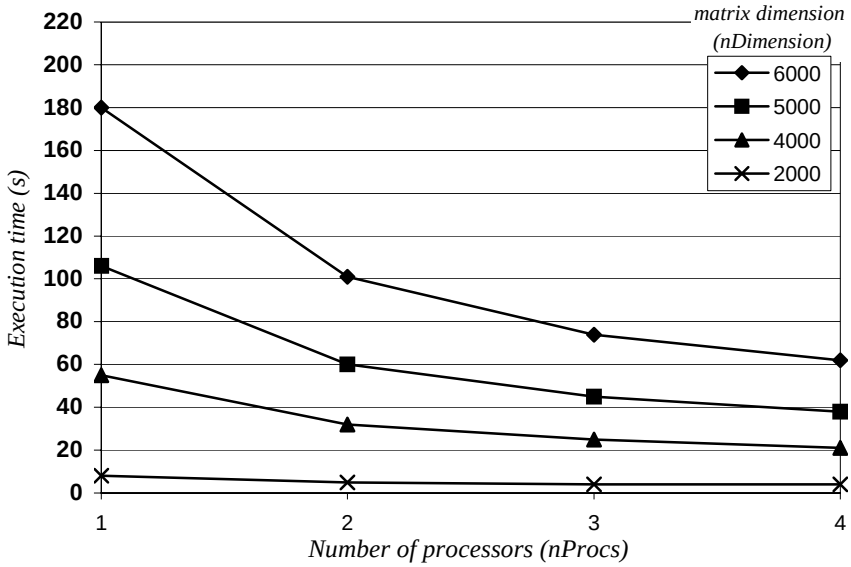


Fig 10: Execution times of parallel matrix multiplication using dgemm and DDPI interface.

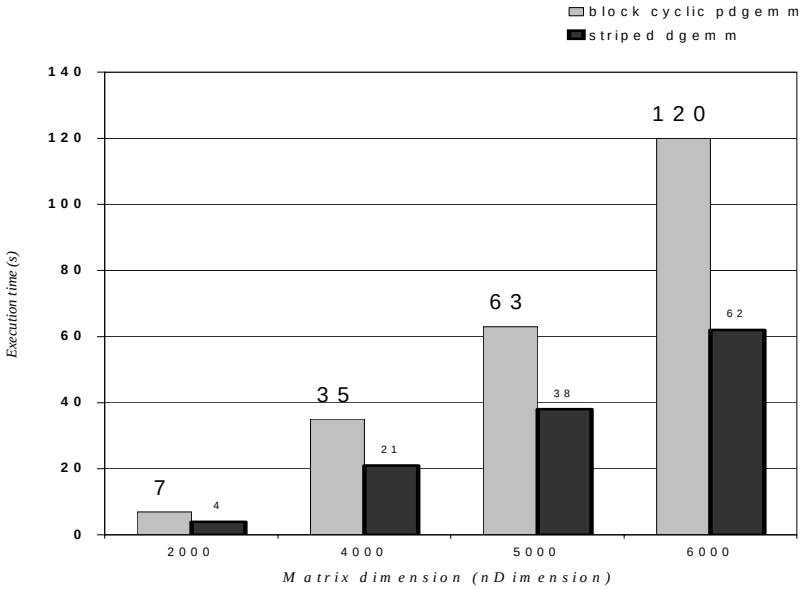


Figure 11: Comparison of execution times between block cyclic pdgemm and striped dgemm.

Figure 11 compares the performance of the two parallel multiplication techniques when all four processors of the cluster are utilized. In terms of the execution time, it is evident that the striped dgemm technique outperforms the block cyclic partitioning scheme by almost twofold for all the matrix sizes. The result demonstrates that with DDPI it is possible to implement a parallel matrix multiplication procedure that outperforms dedicated parallel implementations such as the PBLAS pdgemm. It appears that the drawback of the PBLAS routine is in its restriction to use identical process grid topologies for all the matrices manipulated in the operation.

4.2 Concurrent Data Clustering

Data clustering, which is an NP-complete problem (Garey et al., 1982) of finding groups in heterogeneous data by minimizing some measure of dissimilarity, is one of the fundamental tools in data mining, machine learning and pattern classification solutions. Of all the many available clustering techniques, the k-means center based clustering algorithm, despite of its local minimum solutions, stands out as a popular tool due to its low computational complexity and straightforward implementation (Estivill-Castro and Houle, 2001). Figure 12 depicts the k-means clustering algorithm which finds *k* clusters in a data set of size *nSamples*×*nDimension*. For a single iteration of the search space (steps 2 to 4), the k-means algorithm has the computational complexity of

$$O\left(nSamples \times nDimension \times k\right)$$

The k-means primary advantage of low computational complexity will therefore be inhibited when the number of samples is large. Motivated by this shortcoming when using k-means with large databases, several parallel implementations of the technique have been introduced (Dhillon and Modha, 1999; Kantabutra and Couch, 2000; Ng, 2000; Zhang et al., 2000). According to the analysis by Kantabutra and Couch, their algorithm requires heavy network loading due to rebroadcasts of the data set and therefore only about half of the CPU time is utilized. On the other hand, the data parallel approaches adopted by the other three implementations are superior since only essential local statistics are broadcasted at each iteration, which substantially reduces the interprocessor communication latency. Figure 13 lists the steps in the data parallel approach.

Input

k : number of clusters

X : data set $\in \Re^{nSamples \times nDimension}$

Output

centers : cluster centers $\in \Re^{k \times nDimension}$

Step 1: **Initialization**

Select a set of *k* starting points, the initial cluster centers $\overrightarrow{centers}^{j}$ where:

$$j = 1, \cdots, k$$

$$\overrightarrow{centers}^{j} = \left(centers_1^j, \cdots, centers_{nDimension}^j \right)^T \in \Re^{k \times nDimension}$$

The selection may be done using the Forgy or the random partitioning technique.

Forgy technique:

- set $\overrightarrow{centers}^{j}$ as *k* random samples of the data set

Random partitioning technique:

- partition the data set into *k* segments randomly
- assign each $\overrightarrow{centers}^{j}$ as the centroid of those segments, where centroid is the mean value of the samples assigned to it

Step 2: **Data membership computation**

For each sample $\vec{X}^n$,

$$n = 1, \cdots, nSamples$$

$$\vec{X}^n = \left(X_1^n, \cdots, X_{nDimension}^n \right)^T \in \Re^{nSamples \times nDimension}$$

compute its membership:

$$m(\overrightarrow{centers}^{j} \mid \vec{X}^n) = \begin{cases} 1; if\ l = \arg\min_j \left\| \vec{X}^n - \overrightarrow{centers}^{j} \right\|^2 \\ 0; otherwise \end{cases}$$

Step 3: **Data membership weight assignment**

For each sample $\vec{X}^n$, set its weight to unity:

$$w\left(\vec{X}^n \right) = 1$$

Step 4: **Center recalculation**

For each center $\overrightarrow{centers}^{j}$, recalculate its location from all samples $\vec{X}^n$, according to their membership and weights:

$$\overrightarrow{centers}^{j} = \frac{\sum_{n=1}^{nSamples} m(\overrightarrow{centers}^{j} \mid \vec{X}^n) w(\vec{X}^n) \vec{X}^n}{\sum_{n=1}^{nSamples} m(\overrightarrow{centers}^{j} \mid \vec{X}^n) w(\vec{X}^n)}$$

Step 5: **Convergence condition**

Repeat steps 2 to 4 until convergence. The objective function that the k-means algorithm minimizes is:

$$Perf_{KM}\left(\vec{X}^n \mid \overrightarrow{centers}^{j} \right) = \sum_{n=1}^{nSamples} \min_{j \in \{1 \ldots k\}} \left\| \vec{X}^n - \overrightarrow{centers}^{j} \right\|^2$$

Fig 12: The sequential k-means clustering algorithm.

Step 1: **Initialization**
Partition the data set into *nProcs* partitions and distribute them to the local memory of the respective processes. On the root process, initialize centers values and make them global values by broadcasting them to all processes.

Step 2: **Local computation**
On each process, compute local data memberships, local centers and local performance using local data sets and global centers.

Step 3: **Global center recalculation**
Compute new global centers using summed local centers and summed local data memberships. Compute the global performance by summing local performances.

Step 4: **Convergence condition**
If global performance has converged, terminate computation and return global centers, otherwise start next iteration from step 2.

Fig 13: The data parallel approach to parallelize k-means type clustering algorithms.

With this approach, it is possible to reduce the k-means computational costs of each iteration (steps 2 to 4) to

$$O\left(\frac{nSamples \times nDimension \times k}{nProcs}\right)$$

provided that $nSamples >> nProcs$ (Zhang et al., 2000). By exploiting the similarity of the data parallel approach adopted by DDPI, a parallel k-means algorithm can be implemented in a straightforward manner using DDPI.

Figure 14 compares the sequential implementation of k-means with its parallel counterpart which is implemented via DDPI's row striped partitioning interface. It is evident that with only several additional lines, the k-means algorithm can be converted for concurrent computations with DDPI. The original algorithm flow is still preserved which permits further modifications of the algorithm even by users with little understanding of parallel computing.

In order to empirically evaluate the performance of the parallel k-means, several experiments were conducted with varying number of data samples. For this purpose, synthetic data sets were generated using an algorithm presented by Zhang (Zhang, 2001). The number of clusters (*k* = 8), the dimension size (*nDimension* = 8) and the data set sizes are similar to the ones adopted by Ng (Ng, 2000) since his hardware performance is within the range of the Linux cluster used in this research. The speedup (5) with respect to the execution time of the sequential implementation is shown in Figure 15.

$$speedup = \frac{\text{executionTime}\left(nProcs = 1\right)}{\text{executionTime}\left(nProcs\right)} \quad (5)$$

It can be observed that the speedups gained from the parallel k-means are almost equal to the ideal case which indicates linear speedup. In the largest data set (*nSamples* = 640,000), the speedup is 3.76 on 4 processors. The speedup is only suppressed when the data set is relatively small (*nSamples* = 80,000).

Input
k : number of clusters
X : data set matrix
nSamples : number of data samples
nDimension : data dimension
Output
centers : cluster centers
Variable
meanSE : the k-means performance, based on its objective function

sequential k-means	parallel k-means
data = X**;**	MPI_Init**();** DDPI_Init**();** Matrix**::**Matrix**(**nSamples**,**nDimension**,**X**);** Matrix**::**scatter**(**ROW**);** data = Matrix**::**data**;** myNode = MPI_Comm_rank**();** **if (**myNode == 0**)**

```
// intialize centers


meanSE = BIG_NUM;
do {
  oldMeanSE = meanSE;
  meanSE = 0;
  for j = 1 to k
    dataCnt_j = 0;
    for col = 1 to nDimension
      centers_j,col = 0;
    endfor
  endfor
  for row = 1 to nSamples
    minDistance_row = BIG_NUM;
    for j = 1 to k
      sumDistance = 0;
      for col = 1 to nDimension
        sumDistance = sumDistance +
            (data_row,col – centers_j,col)^2;
      endfor
      if (sumDistance < minDistance_row)
        minDistance_row = sumDistance;
        centerLabel_row = j;
      endif
    endfor
    crow = centerLabel_row;
    for col = 1 to nDimension
      centers_crow,col = centers_crow,col +
          data_row,col;
    endfor
    dataCnt_crow = dataCnt_crow + 1;
    meanSE = meanSE + minDistance_row;
  endfor;




  for j = 1 to k
    for col = 1 to nDimension
      centers_j,col =
          centers_j,col/max(dataCnt_j,1);
    endfor
  endfor
} while (meanSE < oldMeanSE);
```

```
  // intialize centers
endif
MPI_Bcast(centers, k);
meanSE = BIG_NUM;
do {
  oldMeanSE = meanSE;
  meanSE_ = 0;
  for j = 1 to k
    dataCnt_j = 0;
    for col = 1 to nDimension
      centers_j,col = 0;
    endfor
  endfor
  for row = 1 to Matrix::getLclRows();
    minDistance_row = BIG_NUM;
    for j = 1 to k
      sumDistance = 0;
      for col = 1 to nDimension
        sumDistance = sumDistance +
            (data_row,col – centers_j,col)^2;
      endfor
      if (sumDistance < minDistance_row)
        minDistance_row = sumDistance;
        centerLabel_row = j;
      endif
    endfor
    crow = centerLabel_row;
    for col = 1 to nDimension
      centers_crow,col = centers_crow,col +
          data_row,col;
    endfor
    dataCnt_crow = dataCnt_crow + 1;
    meanSE_ = meanSE_ + minDistance_row;
  endfor;
  MPI_Barrier();
  MPI_Allreduce(centers_,centers,MPI_SUM);
  MPI_Allreduce(dataCnt_,dataCnt,MPI_SUM);

MPI_Allreduce(meanSE_,meanSE,MPI_SUM);
  for j = 1 to k
    for col = 1 to nDimension
      centers_j,col =
          centers_j,col/max(dataCnt_j,1);
    endfor
  endfor
} while (meanSE < oldMeanSE);
DDPI_Exit();
MPI_Finalize();
```

Fig 14: Comparison of sequential and parallel implementations of k-means

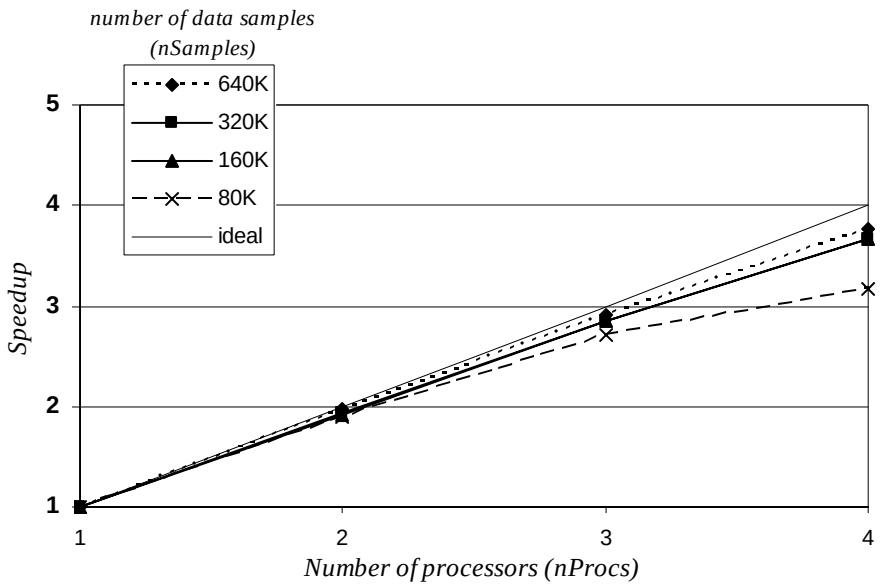


Figure 15: The k-means speedup after parallelization with DDPI

Recently, Hamerly and Elkan have evaluated another center based clustering algorithm called k-harmonic means and found it to be superior to the k-means algorithm in terms of the computed centers' quality (Hamerly and Elkan, 2002). It appears from their findings that, on the contrary to the k-means algorithm, the k-harmonic means algorithm (Zhang, 2001) is robust to initial starting points of the centers. A parallel implementation of the k-harmonic means technique with DDPI is conducted to evaluate the consistency of the DDPI's performance in varied clustering problems. Hence, a concurrent k-harmonic means algorithm was implemented with the DDPI's row striped partitioning interface and a set of experiments was executed similar to that of the k-means algorithm. Figure 16 shows the results of this set of experiments. The results also demonstrate that it is possible to achieve almost linear speedups with the DDPI's parallelizing interface for other clustering techniques such as the k-harmonic means algorithm.

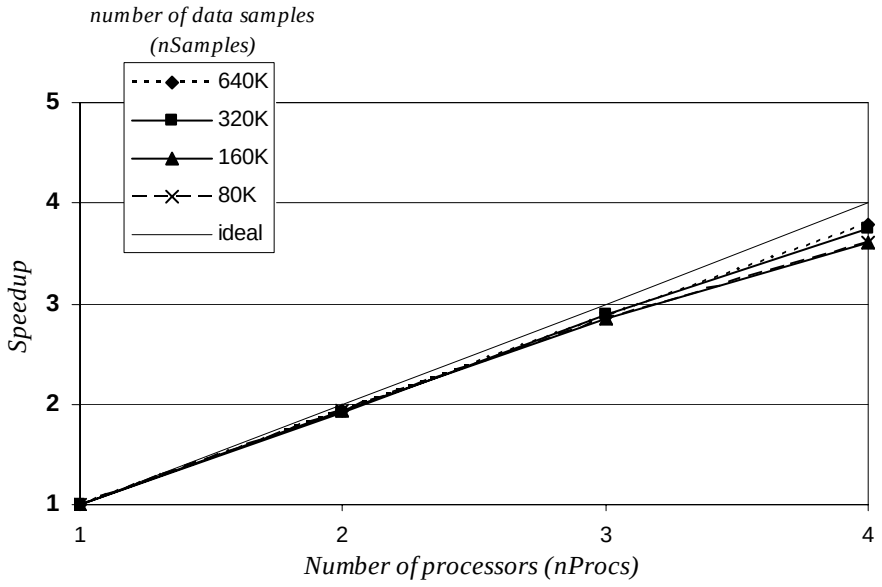


Figure 16: The k-harmonic means speedup after parallelization with DDPI.

4.3 Concurrent Batch Learning for Neural Networks

The learning phase of a neural network is computationally intensive especially when the batch training is employed as opposed to the stochastic technique. With batch training, at each iteration, the entire data set needs to be considered in order to compute the parameters' gradient for an iterative gradient based optimization scheme (such as the commonly used error backpropagation algorithm). Conversely, for the stochastic training, at each iteration, the gradient is computed after considering only a single sample of the data set. There are however, some instances when the batch learning is preferred over the stochastic technique (LeCun et al., 1996).

When large data sets are considered for batch training, the training phase can be parallelized to reduce the computational costs. Parallelization strategies that are available include training each network of a multi-neural network architecture on a dedicated processor, parallelization at the neuron or synapse level, and parallelization using the data parallel approach (Sundararajan and Saratchandran, 1998). Interestingly, akin to the data clustering problem, the data parallel approach appears to be the most favourable technique due to its simplicity and performance (Schikuta and Weidmann, 1997; Rogers and Skillicorn, 1998). The parallelization steps of a general neural network batch training algorithm with the DDPI's interface are shown in Figure 17. In addition to saving memory space by only allocating a portion of the data set on the local memories, the approach can also be applied for both single and multiple neural network architectures.

Step 1: **Initialization**
- Let *nProcs* be equivalent to the number of processors available in the homogeneous parallel computing environment.
- Place the training data set on an *nSamples*×*nDimension* matrix accessible by the root process. Partition the matrix into *nProcs* partitions using DDPI's row striped partitioning technique and distribute them to all processes.
- On the root process, initialize the neural network parameter values and make them global values by broadcasting them to all processes.

Step 2: **Local gradient computation**
- On each process, compute local empirical error and local accumulated gradients using the local data and global parameter values.

Step 3: **Global parameter value adjustment**
- Sum all local accumulated gradients and divide them by the total number of samples (*nSamples*) to obtain the effective global gradient.
- Sum all local empirical errors to obtain global empirical error.
- Adjust the parameter values using the global gradients through an iterative gradient based optimization procedure.
- Broadcast the new global parameter values to all processors.

Step 4: **Convergence condition**
- If global empirical error has converged, terminate computation and return global parameter values, otherwise start next iteration from step 2.

Fig 17: Parallelization steps of batch training with DDPI's interface.

In order to assess the performance of the parallel batch training algorithm, a set of experiments was conducted with the classic Multilayer Perceptron (MLP) and the error backpropagation algorithm. The training was done on a data set with varying number of data samples and fixed number of iterations. The batch training speedup with respect to the execution time of the sequential implementation is shown in Figure 18. It is clear that DDPI's performance is also consistent in the batch training problem. Furthermore, a dedicated neural network parallelization library by Boniface et al. (Boniface et al., 1999) was reported to only achieve speedup of 3.6 on 8 processors whereas with DDPI it is possible to attain speedup up to 3.87 on only 4 processors (*nSamples* = 247731). However it should be noted that their experiment was conducted with the Kohonen Self-organizing Map on a network system more than 3 years ago. Their poor performance is also possibly due to their neuron parallelism strategy which causes heavy network loading.

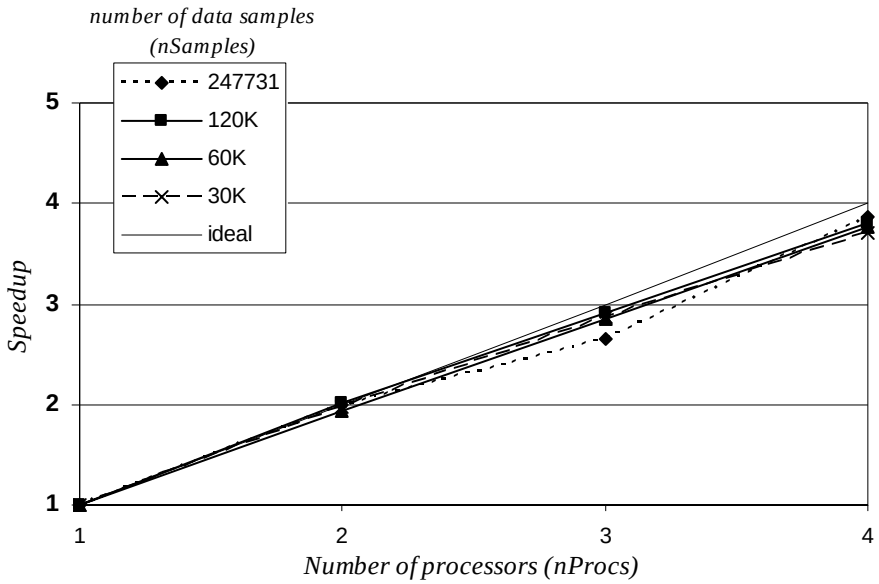


Fig 18: The Multilayer Perceptron batch training speedup after parallelization with DDPI

2.6 Conclusion

A simple yet effective solution for parallelizing iterative or large data problems has been described in this work. DDPI's parallelization versatility has been demonstrated through a wide range of problems. Its almost linear speedup performances appear to be consistent on large data problems which are comparable to dedicated hand coded implementations or other existing sophisticated solutions. DDPI's simplicity of implementation, demonstrated through some of the studied problems, promotes adoption by users with little understanding of parallel computing technicalities. In the future, DDPI can be extended for applications on a heterogeneous cluster by partitioning the workload according to the performance and resources of the individual nodes in the cluster. Additionally, DDPI can also be improved by providing support for complex and irregularly structured problems.

References

[1] Aberdeen, D. and Baxter, J. (2001). "Emmerald: a fast matrix-matrix multiply using Intel's SSE instructions." Concurrency and Computation: Practice and Experience. **Vol. 13 No. 2,** pp. 103-119.

[2] Agarwal, A., Kranz, D. A. and Natarajan, V. (1995). "Automatic Partitioning of Parallel Loops and Data Arrays for Distributed Shared-Memory Multiprocessors." IEEE Transactions on Parallel and Distributed Systems. **Vol. 6 No. 9**, pp. 943-962.

[3] Blackford, L. S., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J. J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D. and Whaley, R. C. (1997). "ScaLAPACK Users' Guide." Philadelphia, USA: SIAM Publications.

[4] Boniface, Y., Alexandre, F. and Vialle, S. (1999). "A Library to Implement Neural Networks on MIMD Machines." Proc. of the 5th International Euro-Par Conference on Parallel Processing (Euro-Par '99). Toulouse, France. 935-938.

[5] Carpenter, B., Zhang, B. and Wen, Y. (1997). "NPAC PCRC Runtime Kernel Definition." Technical Report CRPC-TR97726. Center for Research on Parallel Computation, Rice University, USA.

[6] Chatterjee, S., Lebeck, A. R., Patnala, P. K. and Thottethodi, M. (1999). "Recursive Array Layouts and Fast Parallel Matrix Multiplication." Proc. of the 11th ACM Symposium on Parallel Algorithms and Architectures (SPAA '99). Saint-Malo, France. 222-231.

[7] Chen, J. and Taylor, V. E. (2002). "Mesh Partitioning for Efficient Use of Distributed Systems." IEEE Transactions on Parallel and Distributed Systems. **Vol. 13 No.1**, pp. 67-79.

[8] Comino, C. and Narasimhan, V. L. (2002). "A Novel Data Distribution Technique for Host-Client Type Parallel Applications." IEEE Transactions on Parallel and Distributed Systems. **Vol. 13 No. 2**, pp. 97-110.

[9] Dhillon, I. S. and Modha, D. S. (1999). “A Data-Clustering Algorithm on Distributed Memory Multiprocessors.” Large-Scale Parallel Data Mining. *Lecture Notes in Computer Science*. **Vol. 1759**. 245-260.

[10] Dongarra, J. J., Croz, J. D., Hammarling, S. and Duff, I. S. (1990). “A Set of Level 3 Basic Linear Algebra Subprograms.” ACM Transactions on Mathematical Software. **Vol. 16 No. 1**, pp. 1-17.

[11] Dongarra, J. J. (2002). “Performance of Various Computers Using Standard Linear Equations Software.” Technical Report CS-89-85. University of Tennessee, USA.

[12] Estivill-Castro, V. and Houle, M. E. (2001). “Robust Distance-Based Clustering with Applications to Spatial Data Mining.” *Algorithmica*. **Vol. 30 No. 2**. 216-242.

[13] Garey, M. R., Johnson, D. S. and Witsenhausen, H. S. (1982). “The Complexity of the Generalized Lloyd-Max Problem.” *IEEE Trans. Inform. Theory*. **Vol. 28 No. 2**. 255-256.

[14] Gunnels, J. A., Henry, G. M. and van de Geijn, R. A. (2001). “A Family of High-Performance Matrix Algorithms.” Part 1, Computational Science – 2001. *Lecture Notes in Computer Science*. **Vol. 2073**. 51-60.

[15] Hamerly, G. and Elkan, C. (2002). “Alternatives to the k-means algorithm that find better clusterings.” Proc. of the 11th ACM International Conference on Information and Knowledge Management (CIKM 2002). McLean, USA. 600-607.

[16] Hendrickson, B. and Leland, R. (1994). “The Chaco User's Guide: Version 2.0.” Technical Report SAND94-2692. Sandia National Laboratory, USA.

[17] High Performance Fortran Forum. (1997). “High Performance Fortran language specification, Version 2.0.” Center for Research on Parallel Computation, Rice University, USA.

[18] Kantabutra, S. and Couch, A. L. (2000). “Parallel K-means Clustering Algorithm on NOWs.” NECTEC Technical Journal. **Vol. 1 No. 6**.

[19] Karypis, G. and Kumar, V. (1998). “A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs.” SIAM Journal on Scientific Computing. **Vol. 20 No. 1,** pp. 359-392.

[20] Kumar, V., Grama, A., Gupta, A. and Karypis, G. (1994). “Introduction to Parallel Computing.” Redwood City, USA: Benjamin/Cummings.

[21] LeCun, Y., Bottou, L., Orr, G. B. and Müller, K-R. (1996). “Effiicient BackProp.” Neural Networks: Tricks of the Trade. 9-50.

[22] MPI Forum. (1998). “Special Issue: MPI2: A Message-Passing Interface Standard.” The International Journal of High Performance Computing Applications. **Vol. 12 No. 1-2**, pp. 1-299.

[23] Ng, M. K. (2000). “K-Means-Type Algorithms on Distributed Memory Computer.” International Journal of High Speed Computing. **Vol. 11 No. 2**, pp. 75-91.

[24] Prechelt, L. and Hänßgen, S. U. (2002). “Efficient Parallel Execution of Irregular Recursive Programs.” IEEE Transactions on Parallel and Distributed Systems. **Vol. 13 No. 2**, pp. 167-178.

[25] Rogers, R.O. and Skillicorn, D.B. (1998). “Using the BSP Cost Model to Optimize Parallel Neural Network Training.” *Future Generation Computer Systems*. **Vol. 14**. 409-424.

[26] Rost, B. (2002). “Rising Accuracy of Protein Secondary Structure Prediction.” Protein structure determination, analysis and modeling for drug discovery. Chasman, D. (Ed.). New York, USA: Dekker. 207-249.

[27] Schikuta, E. and Weidmann, C. (1997). “Data Parallel Simulation of Self-organizing Maps on Hypercube Architectures.” Proc. of the Workshop on Self-Organizing Maps (WSOM ‘97). Helsinki, Finland. 142-147.

[28] Sundararajan, N. and Saratchandran, P. (1998). “Parallel Architectures for Artificial Neural Networks.” Los Alamitos, USA: IEEE Computer Society Press.

[29] Valsalam, V. and Skjellum, A. (2002). “A framework for high-performance matrix multiplication based on hierarchical abstractions, algorithms and optimized low-level kernels.” *Concurrency and Computation: Practice and Experience*. **Vol 14 No. 10**. 805-839.

[30] van de Geijn, R. A. (1997). “Using PLAPACK: Parallel Linear Algebra Package.” Massachusetts, USA: MIT Press.

[31] Whaley, R. C., Petitet, A. and Dongarra, J. J. (2001). “Automated Empirical Optimization of Software and the ATLAS Project.” *Parallel Computing*. **Vol. 27 No. 1-2**. 3-25.

[32] Zhang, B. (2001). “Generalized K-Harmonic Means – Boosting in Unsupervised Learning.” Proc. of the 1st SIAM International Conference on Data Mining (SDM ‘01). Chicago, USA.

[33] Zhang, B., Hsu, M. and Forman, G. (2000). “Accurate Recasting of Parameter Estimation Algorithms Using Sufficient Statistics for Efficient Parallel Speed-Up: Demonstrated for Center-Based Data Clustering Algorithms.” Proc. of the 4th European Conference on Principles of Data Mining and Knowledge Discovery (PKDD 2000). Lyon, France. 243-254.