A Parallelizing Interface for K-Means Type Clustering Algorithms and Neural Network Batch Training

Satya Arjunan[1] Safaai Deris[2] Rosli Md Illias[3] Mohd Saberi Mohamad[2]

[1]Laboratory of Bioinformatics, Institute for Advanced Biosciences
Tsuruoka Town Campus of Keio University, 14-1, Baba-cho, Tsuruoka, Yamagata, 997-0035 Japan
Tel: +81-235-29-0800, Fax: +81-235-29-0809, Email: satya@ttck.keio.ac.jp

[2]Department of Software Engineering, Faculty of Computer Science and Information Systems
Universiti Teknologi Malaysia, 81310 Skudai, Johore, Malaysia
Tel: +60-7-557-6160 x 3860, Fax: +60-7-556-5044, Email: safaai@fsksm.utm.my, berie_ext2@lycos.com

[3]Department of Bioprocess Engineering, Faculty of Chemical and Natural Resources Engineering
Universiti Teknologi Malaysia, 81310 Skudai, Johore, Malaysia
Tel: +60-7-550-5313, Fax: +60-7-558-1463, Email: r-rosli@utm.my

Abstract

The k-means clustering algorithm and neural network batch training becomes computationally intensive when the manipulated data is large. One way to reduce the computational demand of such techniques is to execute them in a concurrent manner. Unfortunately, the effort required to implement these techniques in a distributed computing environment remains daunting. Much of the work takes place when partitioning and distributing workloads over processors in the 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 of k-means type clustering algorithms and neural network batch training even by users with little understanding of parallel computing technicalities. In this work we demonstrate that it is possible to achieve near ideal speedups when k-means and k-harmonic means clustering algorithms and multilayer perceptron batch training are parallelized with DDPI.

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

1 Introduction

The k-means clustering algorithm and neural network batch training becomes computationally intensive when the manipulated data is large. One way to reduce the computational demand of such techniques is to execute them in a concurrent manner. Although commodity clusters and parallel computers are becoming widespread now, the effort required to write efficient parallel programmes or to parallelize these techniques remains daunting. 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 (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 data clustering and neural network batch training, the libraries and tools appear to be either overkills (Carpenter et al., 1997; Karypis and Kumar, 1998) or too specialized (Boniface et al., 1999; Chen and Taylor, 2002). For these reasons, we are motivated to look at a general solution and derive the following requirements in this work:

1. *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.
2. *Simple implementation*. The overall structure of the data clustering and neural network batch training should be preserved such that the user would be able to focus on the original algorithm flow of the problem even after parallelization.
3. *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.
4. *Maintainability*. Although initially the solution may be intended for data clustering and neural network

batch training, it should however have the facility to be extended for more complicated problems.

5. *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. In the data parallel approach, the computational workload is spread to processors by distributing partitions of the large manipulated data. The design and implementation 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 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 in data clustering and neural network batch training algorithms, 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
nProcs	total number of processes
prRows	total process rows
prCols	total process columns
prRow	process row coordinate
prCol	process column coordinate
gblRows	global rows
gblCols	global columns
lclRows	local rows
lclCols	local columns
rowBlk	row block size
colBlk	column block size
startPrRow	starting process row
startPrCol	starting process column
nSamples	number of data samples
nDimension	dimension size
contxt	context of the process grid

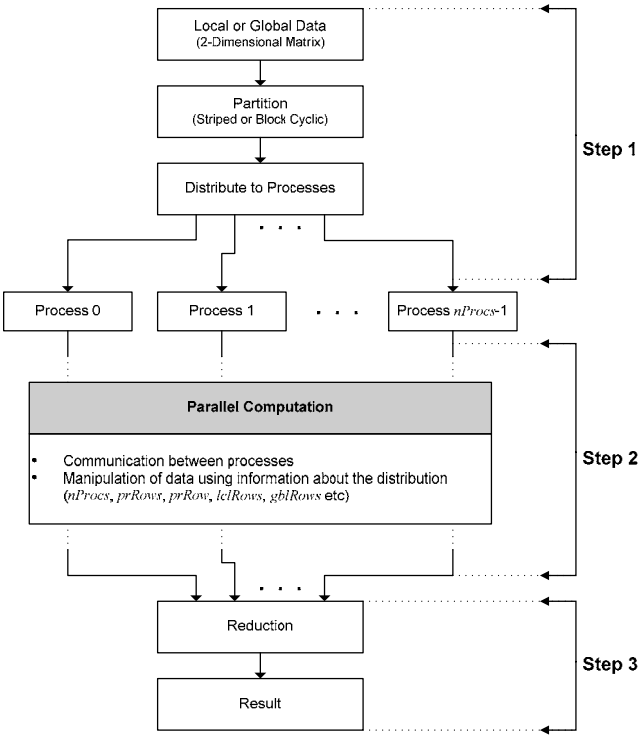


Figure 1: The three main parallelization steps of DDPI

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:

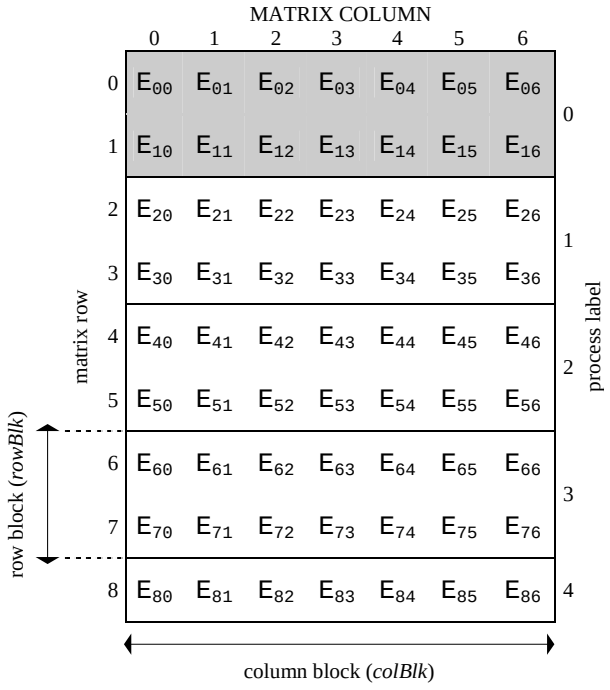
```
Matrix::scatter(partition);
```

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: 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



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

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

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

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

Figure 2: Row striped partitioning distribution

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

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

An example of row striped partitioning is displayed in Figure 2. The example illustrates the partitioned layout of a data matrix *E* of size 9×7 that is distributed over 6 processes. In addition to striped partitioning, DDPI can also be used to distribute data using a partitioning strategy called checkerboard block cyclic partitioning. This technique will not be discussed in this work because it is not used for either of the data clustering or neural network batch training algorithms.

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 summary of available DDPI routines. Although these routines

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*.

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.

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).

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. DDPI 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 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 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

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:

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

Figure 3: The sequential k-means clustering algorithm

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 3 depicts the k-means clustering algorithm which finds k clusters in a data set of size $nSamples \times nDimension$. 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 4 lists the steps in the data parallel approach.

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.

Figure 4: 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 5 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.

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

sequential k-means	parallel k-means

`data = X;`
`// 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);`
	`MPI_Init();`
	`DDPI_Init();`
	`Matrix::Matrix(nSamples,nDimension,X);`
	`Matrix::scatter(ROW);`
	`data = Matrix::data;`
	`myNode = MPI_Comm_rank();`
	`if (myNode == 0)`
	`  // 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();`

Figure 5: Sequential and parallel k-means comparison

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 (3) with respect to the execution time of the sequential implementation is shown in Figure 6.

$$speedup = \frac{\text{executionTime}(nProcs = 1)}{\text{executionTime}(nProcs)} \quad (3)$$

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 impaired when the data set is relatively small (*nSamples* = 80,000).

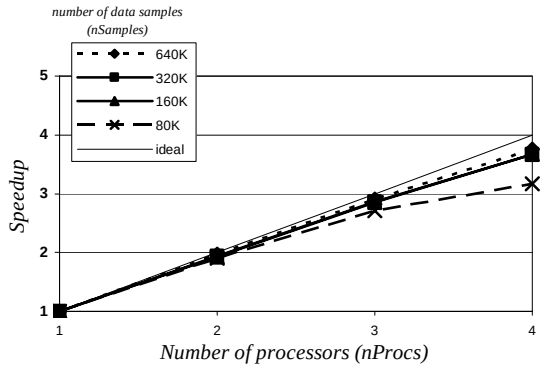


Figure 6: The k-means speedup after parallelization

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 7 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.

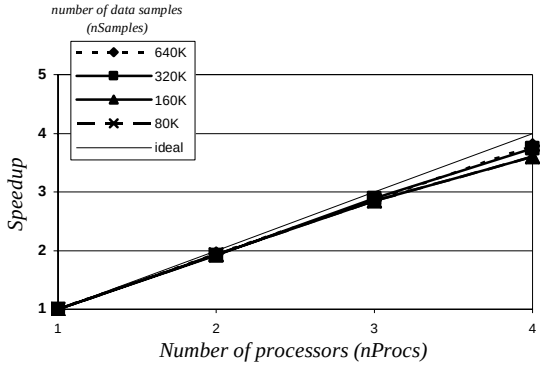


Figure 7: The k-harmonic means parallelization speedup

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 8. 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.

Figure 8: Parallelization steps of batch training

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 9. 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.

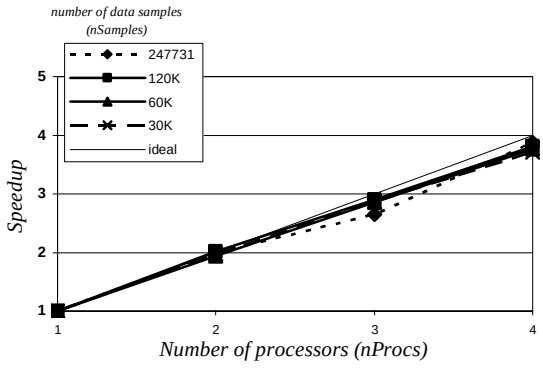


Figure 9: The MLP batch training speedup

5 Conclusion

A simple yet effective interface for parallelizing k-means type clustering algorithms and neural network batch training has been described in this work. DDPI's almost ideal speedup performances appear to be consistent on large data which are comparable to dedicated hand coded implementations or other existing sophisticated solutions. DDPI's simplicity of implementation, 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.

References

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**. 943-962.

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.

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.

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**. 67-79.

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.

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**. 1-17.

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

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.

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.

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.

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

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**. 359-392.

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

MPI Forum. (1998). "Special Issue: MPI2: A Message-Passing Interface Standard." *The International Journal of High Performance Computing Applications*. **Vol. 12 No. 1-2**. 1-299.

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

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**. 167-178.

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.

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.

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

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.

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.

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.