Available online at www.prace-ri.eu

Partnership for Advanced Computing in Europe

Petascaling Machine Learning Applications with MR-MPI

R. Oguz Selvitopi[a], Gunduz Vehbi Demirci[a], Ata Turk[a], Cevdet Aykanat[a*]

[a]*Bilkent University, Computer Engineering Department, 06800 Ankara, TURKEY*

Abstract

This whitepaper addresses applicability of the Map/Reduce paradigm for scalable and easy parallelization of fundamental data mining approaches with the aim of exploring/enabling processing of terabytes of data on PRACE Tier-0 supercomputing systems. To this end, we first test the usage of MR-MPI library, a lightweight Map/Reduce implementation that uses the MPI library for inter-process communication, on PRACE HPC systems; then propose MR-MPI-based implementations of a number of machine learning algorithms and constructs; and finally provide experimental analysis measuring the scaling performance of the proposed implementations. We test our multiple machine learning algorithms with different datasets. The obtained results show that utilization of the Map/Reduce paradigm can be a strong enhancer on the road to petascale.

1. Introduction

Large-scale machine learning problems exist in a large number of social and industrial applications, such as social network analysis, consumer/voter preference analysis, anomaly detection, credit card fraud control/management systems, and postal automation. On the other hand, Map/Reduce is a framework originally developed at Google to ease development of parallel and distributed codes [1]. There are many studies that aim to harness the processing power and ease provided by the Map/Reduce framework for tackling large-scale machine learning problems [16]. The MapReduce paradigm has proved its success by being realized and efficiently utilized on many large-scale projects [2-5]. In this study, we test the scalability and applicability of the Map/Reduce parallel programming paradigm on fundamental data mining approaches with the aim of exploring/enabling processing of terabytes of data on PRACE Tier-0 supercomputing systems. To this end, we first test the usage of MR-MPI library, a lightweight Map/Reduce implementation that uses the MPI library for inter-process communication, on PRACE HPC systems; then propose MR-MPI-based implementations of a number of machine learning algorithms and constructs; and finally provide experimental analysis measuring the scaling performance of the proposed implementations.

2. Background

2.1 Map/Reduce Framework

Map/Reduce is a paradigm originating from functional programming, where higher order functions map and reduce are applied to a list of elements to return a value. The Map/Reduce framework provides a runtime system that manages mapper and reducer tasks, providing automatic scalability and fault tolerance. With the help of this framework, it is possible to ignore complex parallel programming structures like message passing and synchronization and the programmer only needs to design a mapper and a reducer function for each distinct map/reduce phase. Along with reducing programming complexity, another important feature of Map/Reduce is that it can operate on massive data sets. That is, Map/Reduce is designed for scalability instead of speedup.

* Corresponding author. *E-mail address:* aykanat@cs.bilkent.edu.tr

The most widely used open-source Map/Reduce implementation is the Apache Hadoop Project [2]. Hadoop is developed in Java and makes use of TCP/IP ports for communication. Since PRACE Tier-0 systems fail in supporting the requirements of Hadoop, alternative Map/Reduce implementations are considered. Among these, the Map/Reduce-MPI (MR-MPI) library [10] developed at Sandia National Labs seems to be the most appropriate choice for supercomputing systems.

2.2 MR-MPI Library

The scientific computing community tried to exploit the benefits of the MapReduce programming model [6-11]. In [10], the authors developed a lightweight library called MR-MPI. MR-MPI is developed in C++. It uses the MPI library for inter-process communication. These properties enable MR-MPI to be used on HPC platforms without an extra overhead. It also has additional functionalities that can be utilized for speed-up. In the original Map/Reduce framework, it is required to submit each Map/Reduce phase as a separate job, which causes a decrease in performance. In contrast, MR-MPI library does not have such requirements, which leads to a performance increase especially in iterative algorithms like graph algorithms. In the original Map/Reduce framework, initial key-value pairs produced by a map phase are all written to the disk system waiting for the reducer tasks to read their own partitions via remote procedure calls. In the MR-MPI library, whenever a mapper task produces all its key-value pairs, it is not obligated to write all of these key-value pairs to the disk but instead, it is possible to communicate these key-value pairs with reducer tasks while storing them in memory. MR-MPI also provides additional functions to manipulate key-value pairs between map tasks and reduce tasks. For example one can reduce some of the key-value pairs and produce new key-values from them. Later it is possible to unite old key-value pairs which are not reduced with the new key-value pairs for further reduction operations. With MR-MPI lots of further optimizations can be achieved while designing new efficient Map/Reduce algorithms.

MR-MPI enables the utilization of native MPI functions, thus deviating from conventional Map/Reduce implementations. MR-MPI supports in-core and out-of-core modes of operations, which allow processors to utilize disks in situations where data is too big to fit in main memory. However, it does not provide any fault tolerance or data redundancy capability.

MR-MPI provides several operations that perform various tasks. Since these are important for our work, we review them here briefly:

- *map*: Generates key-value (KV) pairs by calling a user program. This is serial and requires no communication between processors. In default settings, each processor is assigned an equal number of mappers.

- *add:* Adds KV pairs from one object to another. Requires no communication and is performed serially.

- *convert:* Converts KV pairs into key- multivalue (KMV) pairs. Prior to calling the convert function, the KV pairs may contain duplicate keys and their related values. After calling convert, the values of the same key are concatenated to be a single KMV pair.

- *collate:* Aggregates KV pairs across processors and then converts them into KMV pairs. This operation requires communication and is actually equivalent to an aggregate operation followed by a convert operation.

- *reduce:* Calls back the user program to process KMV pairs. This operation requires no communication and processes a KMV pair to generate KV pair. The KMVs owned by processors are guaranteed to be unique. At the end of the reduce operation, each processor will own a list of unique KV pairs.

3. Machine Learning with MR-MPI

Within this project, we test the MR-MPI library to see its performance on machine learning algorithms and constructs such as All Pairs Shortest Path (APSP), All Pair Similarity Search (APSS), and Decision Trees (DT). APSP is generally used for measuring importance (via metrics such as betweenness centrality) in social networks. APSS is a widely used kernel in many data mining and machine learning domains. The output of the APSS -a similarity graph- can be used as input to graph transduction [18], graph clustering [19] or near-duplicate detection algorithms. Moreover, the computed similarity graph can also be utilized in the widely used classical k-means [20] or k-nearest neighbor searches [21]. DTs are among the most widely used learning algorithms in machine learning since they are adaptable, easy to interpret, and known to produce highly accurate models.

3.1 All Pairs Shortest Path (APSP)

The input graph to the APSP problem consists of n vertices and can be represented by an n by n adjacency matrix (also referred to as distance matrix). There are two basic algorithms we have implemented using the Map/Reduce paradigm for solving this problem. The first one is the *Repeated Squaring* method in which the adjacency matrix is multiplied by itself $\log n$ times to compute the shortest path between all pairs of vertices. The asymptotic complexity of a single multiplication is $O(n^3)$, thus repeated squaring takes $O(n^3 \log n)$-time. The second approach is the *Floyd-Warshall* algorithm which uses a dynamic programming formulation to obtain a better asymptotic complexity, $O(n^3)$. The Floyd-Warshall algorithm consists of n iterations, where in iteration i, row i and column i are used to compute and update possibly all entries of the distance matrix. Although repeated squaring is more amenable to parallelization, the sparse matrix multiplied by itself in this method becomes denser as the computation proceeds.

The repeated squaring algorithm utilizes matrix multiplication of the form AB as a kernel operation. Based on 1D matrix partitioning, the matrix multiplication problem can be solved in two different ways with the Map/Reduce paradigm. The first method requires a single Map/Reduce phase and the second method requires two consecutive Map/Reduce phases. The former method requires rowwise partitioning of matrix A, whereas the latter one requires columnwise partitioning of matrix A together with conformal rowwise partitioning of matrix B. We opted to use the latter one because it scales better than the single-phase approach. In the two-phase approach, each column i of matrix A is multiplied by row i of matrix B. This operation is called outer product and each outer product produces a matrix consisting of partial results which are later combined to produce the resulting similarity matrix. On the other hand, in the one-phase approach, an inner product operation is performed between each row i of matrix A and all columns of matrix B. Each inner product operation produces a single element of the resulting output matrix. The one-phase approach necessitates replication of *all* columns of matrix B among the processors that participate in multiplication, whereas the two-phase requires *no* replication at all [17]. Hence, the one-phase approach incurs more communication, which hinders its scalability. Moreover, even though the second approach needs two phases, MR-MPI library does not need intermediate key-value pairs to be written to the distributed file system, which provides low latency between two consecutive Map/Reduce phases. Hence, the two-phase approach is more amenable to parallelization.

Algorithm 1 displays the two-phase matrix multiplication algorithm. At the beginning of the algorithm, two MR-MPI library objects named as A and B are initialized. Object A stores the key-value objects which are read from the distributed file system using parallel I/O functions of the MPI library. Key-value pairs are of the form $(i,j)(a_{ij})$ and correspond to nonzero entries of matrix A. Here, (i,j) is the key that stores the row and column indices of the respective nonzero element a_{ij}. The B object is initially empty. To obtain a columnwise partition of matrix A object, we use a mapper function which transforms the key-value pairs $(i,j)(a_{ij})$ into $(j)(i,a_{ij})$. In a similar manner, to obtain a conformal rowwise partitioning of the B object, we use a mapper function which transforms the key-value pairs $(i,j)(b_{ij})$ into $(i)(j,b_{ij})$. A collate operation is then performed where all the key-value pairs are hashed according to key fields and are distributed among processors. When the collate phase is complete, nonzero entries of columns of matrix A and rows of matrix B with the same key field are all collected by the

Input: MapReduce object A = non-zero coordinates with weights: $Key = (row_i, col_j), Value = (a_{ij})$
MapReduce object B = non-zero coordinates with weights: $Key = (row_i, col_j), Value = (b_{ij})$

Map columns using A as input:
Input: $Key = (row_i, col_j), Value = (a_{ij})$
Output: $Key = (col_j), Value = ('A', row_i, a_{ij})$
Map rows using B as input:
Input: $Key = (row_i, col_j), Value = (b_{ij})$
Output: $Key = (row_i), Value = ('B', col_j, b_{ij})$

Collate conformal columns of A and rows of B: Row and column numbers are Key

Reduce:
Input: $Key = (col_i)\ or\ (row_i), MultiValue = ('A', row_j, a_{ji}) or ('B', col_j, b_{ij})$
ColumnVectorA, rowVectorB = split MultiValues according to the tags in the values
$C = columnVectorA \times rowVectorB$
For each $c_{ik} \in C$ do
Output: $Key = (row_i, col_k), Value = (c_{ik})$

Map columns using C as input:
Input: $Key = (row_i, col_k), Value = (c_{ik})$
Output: $Key = (col_k), Value = ('C', row_i, c_{ik})$

Collate: collect all partial results in the same column
Input: $Key = (col_k), MultiValue = ('C', row_i, c_{ik})$
Output: $Key = (col_k), Value = MultiValue$

Reduce:
Input: $Key = (col_k), MultiValue = ('C', row_i, c_{ik})$
For each $c_{ik} \in C_i$ do
$r_{ik} += c_{ik}$
Output: $Key = (row_i, col_k), Value = (r_{ij})$

Algorithm 1: Two-phase matrix multiplication algorithm that computes AB.

same processor, thus achieving conformal columnwise and rowwise partitioning of matrices A and B, respectively. Then, all key-value pairs with the same key are merged into a single key-multi value object of the form $(j)(i, [a_{ij}])$ and a reduce operation is performed with a user defined function that contains a similarity operator within the outer product operation forming a new local intermediate matrix (denoted as C) with partial results. The formed entries of the local intermediate matrices are of the form $(i, k)(c_{ik})$, which are then used to be mapped in a columnwise fashion to generate the output key-value pairs $(k)(i, c_{ik})$. Finally, a collate operation is performed to distribute each column of the local intermediate matrices produced by the outer product operations. After this phase, partial results of different intermediate matrices that have the same column indices are gathered by the same processor. Therefore, it is possible to compute columns of the resulting matrix locally by all processors.

Algorithm 2 displays the high-level MR-MPI pseudo-code of the repeated squaring algorithm. Note that each iteration of Algorithm 2 effectively implements the matrix multiplication algorithm given in Algorithm 1, where the same input matrix is multiplied by itself at each iteration and the output matrix of the current iteration becomes the input matrix of the following iteration.

```
Require: A = (i, j)(a_ij), n
 1: Init MapReduce objects A
 2: k = 1
 3: while k < n − 1 do
 4:    A ← A.map(mapper1DCW)
 5:    B ← A.map(mapper1DRW)
 6:    A ← B.add( )
 7:    delete B
 8:    A.collate(NULL)
 9:    A.reduce(reducerOP)
10:    A ← A.map(mapper1DCW)
11:    A.collate(NULL)
12:    A.reduce(reducerSP)
13:    k ← k × 2
14:  return A
```

Algorithm 2: High-level MR-MPI pseudo-code for APSP with repeated squaring.

```
Require: A = (i, j)(w_ij), n
 1: A ← A.map(mapper1DCW)
 2: A.collate(NULL)
 3: A.reduce(reducer1DCW)
 4: for k = 0 to n − 1 do
 5:    T ← A.map(kthColMapper)
 6:    T.collate(NULL)
 7:    T.reduce(kthColReducer)
 8:    A.convert(NULL)
 9:    A.reduce(kthIterationReducer)
10: return A
```

Algorithm 3: High-level MR-MPI pseudo-code for APSP with Floyd-Warshall.

Algorithm 3 presents the high-level MR-MPI pseudo-code for the Floyd-Warshall algorithm. The input to this algorithm is an n by n matrix. This input matrix is represented with the Map/Reduce object A whose key-value pairs are of the form $(v_i, v_j)(w_{ij})$, where v_i and v_j are the source and the target nodes, respectively, and w_{ij} is the distance between these two nodes. Lines 1-3 in the algorithm perform a columnwise partitioning of the matrix by transforming the key-value pairs into the form $(v_j)(v_i, w_{ij})$. The key-value pairs that correspond to the same column entries in the distance matrix are stored and processed by the same processor. Hence, the distance computation in each iteration can be performed locally. The main loop in Algorithm 3 is executed n times and at each iteration k, the k^{th}column is replicated to all reducer tasks. This is achieved by the user-defined function *kthColMapper*, adding these key-value pairs to the Map/Reduce object *T*, performing a collate operation to distribute column k to all reducers, and finally performing a reduce operation (*kthColReducer*) to gather the corresponding entries of k^{th} column in a single contiguous vector (lines 5-7). Recall that the convert function converts all values of a key to a single key-multivalue object by aggregating these values in a contiguous data structure. The convert operation in line 8 converts distinct key-value objects to key-multivalue objects, which is an obligation of the MR-MPI library since reduce operations can only be performed on key-multivalue objects in MR-MPI. Finally, the reduce operation is performed with the user-defined function *kthIterationReducer* to update all key-value pairs using k^{th} column and the recursive formulation of the Floyd-Warshall algorithm. The updated values are stored in Map/Reduce object *A* and of the form $(v_j)(v_i, w_{ij})$.

Require: $A = (i, j)(a_{ij})$
1: Init MapReduce object A
2: $A \leftarrow A.\text{map}(mapper1DCW)$
3: $A \leftarrow A.\text{map}(similarityOP)$
4: $A.\text{collate}(\text{NULL})$
5: $A.\text{reduce}(reducerSP)$
6: return A

Algorithm 4: High-level MR-MPI pseudo-code for APSS.

3.2 All Pairs Similarity Search (APSS)

The APSS problem is defined for a given set of vectors as finding all vector pairs the similarities of which are above a given similarity threshold. In this problem, a given set of m-dimensional n vectors are represented by an n by m input matrix A by placing vectors which represent high dimensional data at the row positions of the matrix. In this representation, APSS reduces to multiplication of the input matrix A with its transpose A^T. Matrix multiplication is performed by using a similarity operator instead of each scalar multiply-and-add operation that is used in the standard multiplication.

The basic difference between the APSP algorithm with repeated squaring and the APSS algorithm is that the APSP involves the matrix multiplication of the form AA whereas APSS involves the matrix multiplication of the form AA^T. A columnwise partition of matrix A already induces a conformal rowwise partition of matrix A^T. Hence, in the APSS algorithm, we can easily avoid the expensive map, collate and reduce operations in the first phase of the two-phase matrix multiplication algorithm given in Algorithm 1. Another important difference is that the APSP algorithm with repeated squaring performs $\log n$ successive matrix multiplications whereas APSS performs only once. So, the APSS algorithm reduces to the algorithm given in Algorithm 4.

3.3 Decision Trees

Decision Trees (DTs) are tree-structured plans of a set of attributes used for addressing learning-based prediction problems. In decision tree learning, as training data we are given a training set of (*x*, *y*) pairs where *x* are *d*-dimensional feature vectors (also called attributes or input vectors) and *y* are associated output variable (label), and during testing, we are given an input vector and asked to predict the output variable. DTs perform this task by splitting the data at each internal node and making a prediction at each leaf node. At the root of the tree, the entire training dataset is examined to find the best split predicate for the root. The dataset is then partitioned along the split predicate and the process is repeated recursively on the partitions to build the child nodes.

The Map/Reduce-based construction of the decision tree from a given input set is performed in multiple Map/Reduce iterations. There can be various Map/Reduce implementations but here we describe the approach presented in [15]. In this algorithm, each Map/Reduce iteration builds one level of the decision tree. In an iteration, a mapper considers a number of possible splits on its subset of the data and for each split it stores partial statistics, which are sent to reducers. A reducer collects all partial statistics and sends them back to the master node to determine the best split. A master node grows the respective level of the tree and also decides if nodes should be split further. We refer to mappers and reducers as worker nodes. The stopping criteria are based on two main conditions: (i) when the number of records falls below a pre-determined threshold or (ii) information gain at the corresponding node is below a predetermined value, the node is not further expanded.

The main stages of a single iteration are described below:

1. The master node distributes data and decision tree information to worker nodes.
2. Each worker maps its own portion of data by computing split information using split values of all attributes.
3. A collate operation is performed to distribute the KV pairs (generated by mappers) to reducers.
4. Each reducer finds the best local split for each active node of the tree and sends them back to master node.
5. The master node receives partial local best split attributes from worker nodes and chooses best attribute for the currently active nodes. Then, it decides which tree nodes to expand.

4. Experiments

To test the MR-MPI library, we first ported this library to Hermit (Cray XE6) system [12] and developed MR-MPI based implementations of the APSP, APSS and DT algorithms. We should note that compiling and testing MR-MPI in almost all HPC systems can be performed with ease since MR-MPI is only dependent on the MPI library, which is already well linked and optimized in most HPC systems. To obtain peak performance, MR-MPI must be tuned with the parameters of the parallel file system (Hermit provides Lustre [14] parallel file system). We made use of parallel MPI-I/O for data transfers performed between the memory and the disk.

4.1 All Pairs Shortest Path

For the APSP experiments we have used three random graphs that are generated recursively with power-law degree distributions. These kinds of graphs are commonly used to represent social networks. The properties of these random graphs are displayed in Table 1. APSP algorithms are run on 128, 256 and 512 cores for the rgraph1, rgraph2 and rgraph3 datasets, respectively, to conduct weak scaling performance tests. The obtained results for these datasets are displayed in Table 2.

Name	Type	Nodes	Edges	Description
rgraph1	Directed	256	2560	Randomly generated network
rgraph2	Directed	512	5120	Randomly generated network
rgraph3	Directed	1024	10240	Randomly generated network

Table 1: APSP Dataset properties

For APSP, we considered two alternative implementations, one based on repeated squaring (RSQ) and one based on the Floyd-Warshall (FW) algorithm. The results show that RSQ obtains better running times compared to FW. The reason is that the number of Map/Reduce iterations is $\log n$ for RSQ, whereas it is n for the FW algorithm.

Considering the cubic complexity of the APSP problem, we note that when we double the input data size, the problem size (as defined in [22]) increases $2^3 = 8$ times. So, when we double both the input data size and the number of processors, the computational load of a single core increases four-fold. As seen in Table 2, parallel running times increase less than four-fold, thus showing that the our MR-MPI-based APSP algorithms exhibit successful weak scaling performance.

Dataset:	**rgraph1**	**rgraph2**	**rgraph3**
Number of processors:	**128**	**256**	**512**
RSQ	21.36	57.22	152.47
FW	147.91	534.19	1187.81

Table 2: APSP runtimes (in seconds) for weak-scaling experiments.

4.2 All Pairs Similarity Search

For APSS experiments, we have used a realistic social graph named LiveJournal. The LiveJournal social graph is taken from the Stanford University Large Network Dataset Collection [13]. LiveJournal is an on-line community in which a significant fraction of members is highly active. It allows members to maintain journals and to declare which other members are their friends. Each node in the graph corresponds to a member in the network and an edge is added between nodes if the respective pair of members become friends. The properties of these datasets are presented in Table 3.

Name	Type	Nodes	Edges	Description
LiveJournal (LJ)	Directed	4,847,571	68,993,773	LiveJournal online social network

Table 3: APSS dataset properties.

The execution time results gathered from the APSS experiments are displayed in Figure 1. The APSS algorithm is run on 1024, 2048 and 4096 cores for the LiveJournal dataset. As seen from the results, APSS scales nicely

until 4096 cores where parallelization overheads prevent linear strong scaling after 4096 cores. For this dataset, a super linear speedup is achieved between 1024 and 2048 cores since the size of the data that is assigned to each processor is bigger than the size of the memory available at each core which causes I/O operations to swap pages of key-value pairs between memory and distributed file system. Additionally, as one can see the huge datasets can be processed on the HPC systems in a very small period of time which might not be possible with other parallel computing systems.

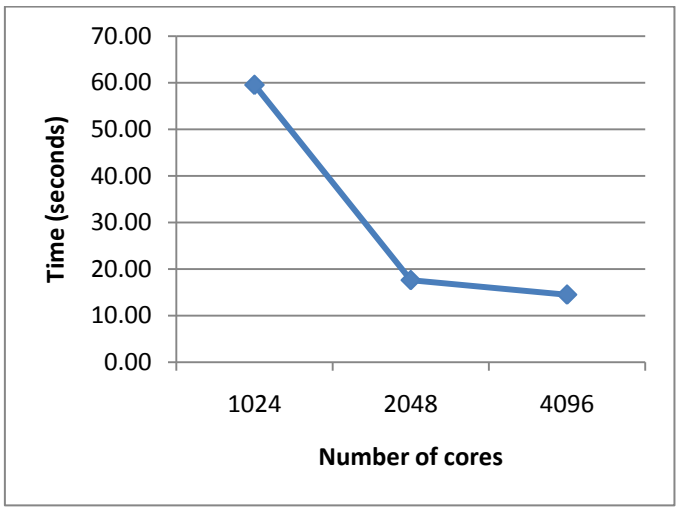

Figure 1: Running time of APSS algorithm with increasing number of cores.

Figure 2 shows the dissection of the running time of the APSS algorithm (described in Section 3.2) into different stages. Recall that only the collate stage incurs communication whereas the other stages contain computational load. In the figure, the map #2 stage involves the expensive local outer product operations. For scalable performance beyond 4096 cores, intelligent partitioning algorithms that consider reducing communication overhead incurred during the collate operation while maintaining computational balance during the map stage are required.

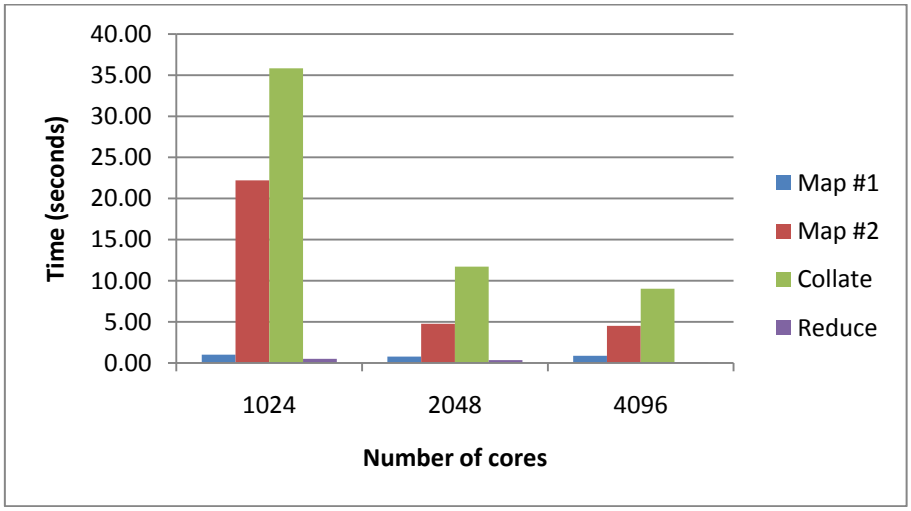

Figure 2: APSS running time dissection for the LiveJournal dataset.

4.3 Decision Tree

We have tested our implementation on a real-life dataset with 500,000 records and 90 attributes [23]. Our test is based on regression trees. In regression trees, attributes and the target attribute(s) are both ordered. Building of the decision tree is performed with 64, 128, …, 2048 cores. We stop building the decision tree when the number of levels in the tree exceeds 20. We used an SGI Altix 8600 (vilje) machine for our tests. Figure 3 displays the execution times obtained for varying number of cores.

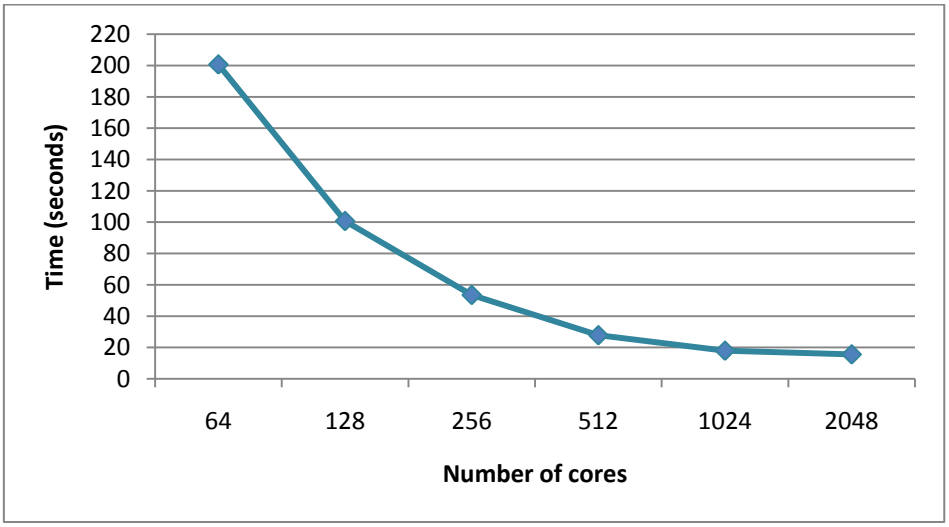

Figure 3: Running time of building decision tree with increasing number of cores.

The obtained execution times indicate a good scalable performance for building decision tree. Up to 1024 cores, we almost get linear speedup. The performance starts to deteriorate at 2048 processors, which is due to the increasing communication requirements. Even though decision tree building algorithm is highly iterative and sequential in nature due to dependencies between successive levels of the tree, the execution times show that a fast and optimized MapReduce library can attain quite good performance.

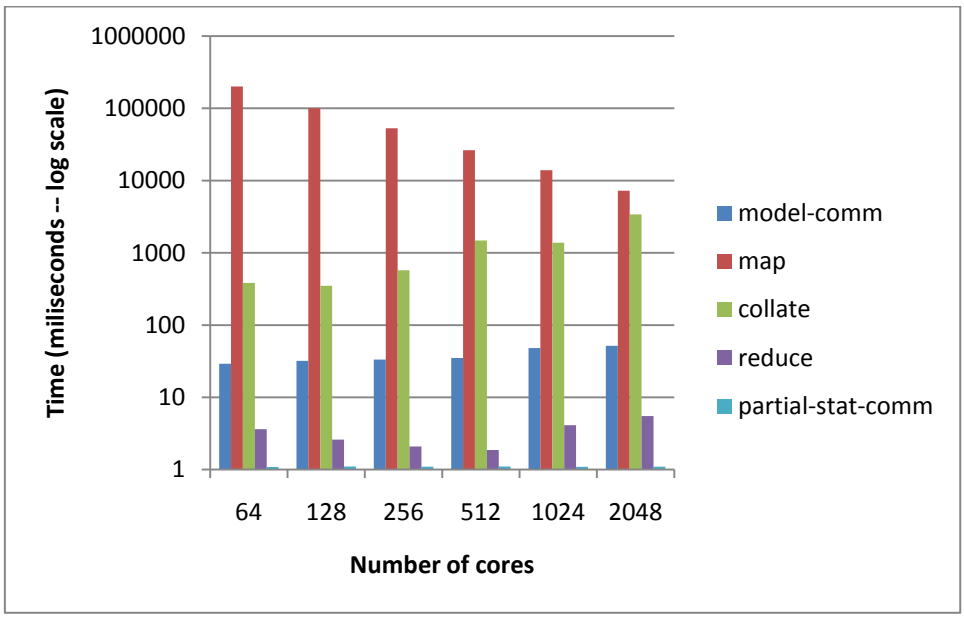

Figure 4: Decision tree build execution time dissection.

We present the running time dissection of building the decision tree in Figure 4. As seen from the figure, most of the time is spent in either map stage or collate stage. At low core counts, map stage constitutes a high percentage of total execution time. With increasing number of processors, the time spent in the collate stage increases while the time spent in the map stage decreases. At 2048 cores, they almost become equal. This indicates that communication requirements of the parallel decision tree building algorithm become bottleneck in obtaining a scalable performance. Note the other two stages (model-comm and partial-stat-comm) which also require communication. Since the data communicated in these stages are not large, they do not affect the parallel efficiency of the algorithm as drastically as the collate stage.

5. Conclusions

We have implemented MR-MPI-based machine learning algorithms that are able to utilize the Map/Reduce programming model in high performance computing systems. MR-MPI depends on only a few well-known and widely used libraries and thus we did not have any problems in porting our code from one machine to another.

We realized MapReduce-based implementations for all pairs shortest path, all pairs similarity search and decision tree building problems. Experimenting with different algorithms, we showed that achieving scalability via the Map/Reduce paradigm for machine learning problems is quite viable. Experimental results also show that for approaching petascale and exascale performance, we need intelligent partitioning methods that are centered around minimizing communication requirements of the underlying MapReduce application. We believe that the MR-MPI library has the possibility to make a great impact in scientific computing since it eases parallel programming while providing high scalability for HPC platforms.

References

[1] Dean, J., & Ghemawat, S. (2008). Map/Reduce: simplified data processing on large clusters. *Communications of the ACM*, *51*(1), 107-113.

[2] White, T. (2009). Hadoop: The Definitive Guide: The Definitive Guide. O'Reilly Media.

[3] Olston, C., Reed, B., Srivastava, U., Kumar, R., & Tomkins, A. (2008, June). Pig latin: a not-so-foreign language for data processing. In Proceedings of the 2008 ACM SIGMOD international conference on Management of data (pp. 1099-1110). ACM.

[4] Thusoo, A., Sarma, J. S., Jain, N., Shao, Z., Chakka, P., Anthony, S., ... & Murthy, R. (2009). Hive: a warehousing solution over a map-reduce framework. Proceedings of the VLDB Endowment, 2(2), 1626-1629.

[5] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., ... & Gruber, R. E. (2008). Bigtable: A distributed storage system for structured data. ACM Transactions on Computer Systems (TOCS), 26(2), 4.

[6] Cohen, J. (2009). Graph twiddling in a Map/Reduce world. Computing in Science & Engineering, 11(4), 29-41.

[7] Ekanayake, J., & Fox, G. (2010). High performance parallel computing with clouds and cloud technologies. In Cloud Computing (pp. 20-38). Springer Berlin Heidelberg.

[8] Kang, U., Tsourakakis, C. E., & Faloutsos, C. (2009, December). Pegasus: A peta-scale graph mining system implementation and observations. In Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on (pp. 229-238). IEEE.

[9] Tu, T., Rendleman, C. A., Borhani, D. W., Dror, R. O., Gullingsrud, J., Jensen, M. O., ... & Shaw, D. E. (2008, November). A scalable parallel framework for analyzing terascale molecular dynamics simulation trajectories. In High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for (pp. 1-12). IEEE.

[10] Plimpton, S. J., & Devine, K. D. (2011). Map/Reduce in MPI for large-scale graph algorithms. Parallel Computing, 37(9), 610-632.

[11] http://Map/Reduce.sandia.gov/doc/Manual.html, Map/Reduce-MPI (MR-MPI) Library Documentation

[12] http://www.hlrs.de/?id=1546

[13] http://snap.stanford.edu/data/

[14] P. Schwan. Lustre: Building a file system for 1000-node clusters. In Proceedings of the 2003 Linux Symposium, July 2003.

[15] Biswanath Panda, Joshua S. Herbach, Sugato Basu, and Roberto J. Bayardo. 2009. PLANET: massively parallel learning of tree ensembles with Map/Reduce. *Proc. VLDB Endow.* 2, 2 (August 2009), 1426-1437.

[16] https://mahout.apache.org/

[17] A. Rajaraman and J. D. Ullman, Mining of Massive Datasets. New York, NY, USA: Cambridge University Press, 2011.

[18] Joachims T (2003) Transductive learning via spectral graph partitioning. In: In ICML, pp 290–297

[19] Brandes U, Gaertler M, Wagner D (2003) Experiments on graph clustering algorithms. In: Di Battista G, Zwick U (eds) Algorithms - ESA 2003, Lecture Notes in Computer Science, vol 2832, Springer Berlin / Heidelberg, pp 568–579, URL http://dx.doi.org/10.1007/978-3-540-39658-1_52

[20] Lloyd SP (1982) Least square quantization in pcm. IEEE Transactions on Information Theory Originally published in 1957 in Bell Telephone Laboratories Paper

[21] Cover T, Hart P (1967) Nearest neighbor pattern classification. IEEE Transactions on Information Theory

[22] Kumar, V., Grama, A., Gupta, A., & Karypis, G. (1994). *Introduction to parallel computing* (Vol. 110). Redwood City: Benjamin/Cummings.

[23] https://archive.ics.uci.edu/ml/datasets/YearPredictionMSD.

Acknowledgements

This work was financially supported by the PRACE project funded in part by the EUs 7th Framework Programme (FP7/2007-2013) under grant agreement no. RI-312763.