Chapter 5
Machine Learning Algorithm Acceleration Using Hybrid (CPU-MPP) MapReduce Clusters

Sergio Herrero-Lopez and John R. Williams

Abstract The uninterrupted growth of information repositories has progressively led data-intensive applications, such as MapReduce-based systems, to the mainstream. The MapReduce paradigm has frequently proven to be a simple yet flexible and scalable technique to distribute algorithms across thousands of nodes and petabytes of information. Under these circumstances, classic data mining algorithms have been adapted to this model, in order to run in production environments. Unfortunately, the high latency nature of this architecture has relegated the applicability of these algorithms to batch-processing scenarios. In spite of this shortcoming, the emergence of massively threaded shared-memory multiprocessors, such as Graphics Processing Units (GPU), on the commodity computing market has enabled these algorithms to be executed orders of magnitude faster, while keeping the same MapReduce-based model. In this chapter, we propose the integration of massively threaded shared-memory multiprocessors into MapReduce-based clusters, creating a unified heterogeneous architecture that enables executing Map and Reduce operators on thousands of threads across multiple GPU devices and nodes, while maintaining the built-in reliability of the baseline system. For this purpose, we created a programming model that facilitates the collaboration of multiple CPU cores and multiple GPU devices towards the resolution of a data intensive problem. In order to prove the potential of this hybrid system, we take a popular NP-hard supervised learning algorithm, the Support Vector Machine (SVM), and show that a $36\times - 192\times$ speedup can be achieved on large datasets without changing the model or leaving the commodity hardware paradigm.

S. Herrero-Lopez (✉)
Technologies, Equities and Currency (TEC) Division, SwissQuant Group AG, Kuttelgasse 7, 8001 Zurich, Switzerland
e-mail: sergherrero@gmail.com

J.R. Williams
Massachusetts Institute of Technology, 77 Massachusetts Avenue, 02139 Cambridge, MA, USA
e-mail: jrw@mit.edu

A. Gkoulalas-Divanis and A. Labbi (eds.), *Large-Scale Data Analytics*,
DOI 10.1007/978-1-4614-9242-9_5,

5.1 Introduction

The data mining community has often assumed that performance increase on existing techniques would be given by the continuous improvement of processor technology. Unfortunately, due to physical and economic limitations, it is not recommendable to rely on the exponential frequency scaling of CPUs anymore. Furthermore, the low price and ubiquity of data generation devices not only has led to larger datasets that need to be digested on a timely manner, but also to the growth of dimensionality, categories and formats of the data. Simultaneously, an increasingly heterogeneous computing ecosystem has defined three computing families:

1. *Commodity Computing*: It encompasses large-scale geographically distributed commodity machine clusters running primarily open source software. Its reliability to host batch processing systems, such as Hadoop [12], and storage systems, such as BigTable [5] or Cassandra [17], across tens of thousands of nodes and petabytes of data, have made commodity computing the foundation of internet-scale companies and the cloud.
2. *High Performance Computing/Supercomputing*: It refers to centralized multi-million computer systems capable of delivering high throughput for complex tasks that demand large computational power. Typically, these are funded and operated by governments or large corporations, and are utilized for the resolution of scientific problems.
3. *Appliance Computing*: It refers to highly specialized systems exclusively designed to carry out one or few similar tasks with maximum performance and reliability. These nodes combine state-of-the-art processor, storage and interconnect technologies and cost one order of magnitude less than supercomputers. These computing appliances have been successfully utilized for large-scale analytics and enterprise business intelligence operations.

Under these circumstances, it is required for the research community to investigate the adaptation of classic and novel data intensive algorithms to this heterogeneous variety of parallel computing ecosystems and the technologies that compose them. This adaptation process can be separated into two phases: The *Extraction Phase*, in which the parallelizable parts, called parallel *Tasks*, of the algorithm are identified and separated; and the *Integration Phase*, in which these tasks are implemented for the most suitable parallel computing platform or combination of them.

5.1.1 Extraction Phase

The *extraction of parallelism* on a data intensive algorithm can be carried out at different levels, with different impacts on performance and increased programming complexity:

1. *Independent Runs*: This is the most common technique; it simply runs the same algorithm with different configuration parameters on different processing nodes. Each of the runs is independent and parallel execution does not speed up individual runs.
2. *Statistical Query & Summation*: This technique decomposes the algorithm into an adaptive sequence of statistical queries, and parallelizes these queries over the sample [16]. This approach is satisfactory in speeding up slow algorithms, in which little communication is needed.
3. *Structural Parallelism:* This technique is based on the exploitation of fine-grained data parallelism [15]. This is achieved by handling each data point with one or few processing threads.

These three techniques are complementary and are often combined to yield maximum performance on a given target parallel computing platform. Successful parallelization transforms a computationally limited problem into a bandwidth bound problem, in which communication between processing units becomes the bottleneck and optimizing for minimum latency gains critical importance. The full exposure of the complexity of parallel programming will result in the largest performance gain.

Individual parallel tasks extracted through both Statistical Queries & Summation and/or Structural Parallelism, can be directly modeled using the MapReduce programming paradigm [8]. The MapReduce framework is illustrated in Fig. 5.1.

The Map and Reduce operators are defined with respect to structured (key, value) pairs. Map (M) takes one pair of data with a type in one domain, and returns a list of pairs in a different domain:

$$M\,[k_1, v_1] \rightarrow [k_2, v_2] \tag{5.1}$$

The Map operator is applied in parallel to every item in the input dataset. This produces a list of (k_2, v_2) pairs for each call. Then, the framework collects all the pairs with the same key and groups them together. The Reduce (R) operator is then applied to produce a v_3 value.

$$R\,[k_2, \{v_2\}] \rightarrow [v_3] \tag{5.2}$$

The advantage of the MapReduce model is that makes parallelism explicit, and more importantly, language or platform agnostic, which allows executing a given algorithm on any combination of platforms in the parallel computing ecosystem. M or R tasks are distributed dynamically among a collection of *Workers*. The Workers is an abstraction that can represent nodes, processors or Massively Parallel Processor (MPP) devices.

Researchers have focused their effort on the decomposition of Machine Learning algorithms as iterative flows of Map and Reduce tasks. Next, the decomposition of three classic Machine Learning algorithms into flows of Map (M) and Reduce (R) tasks is explained:

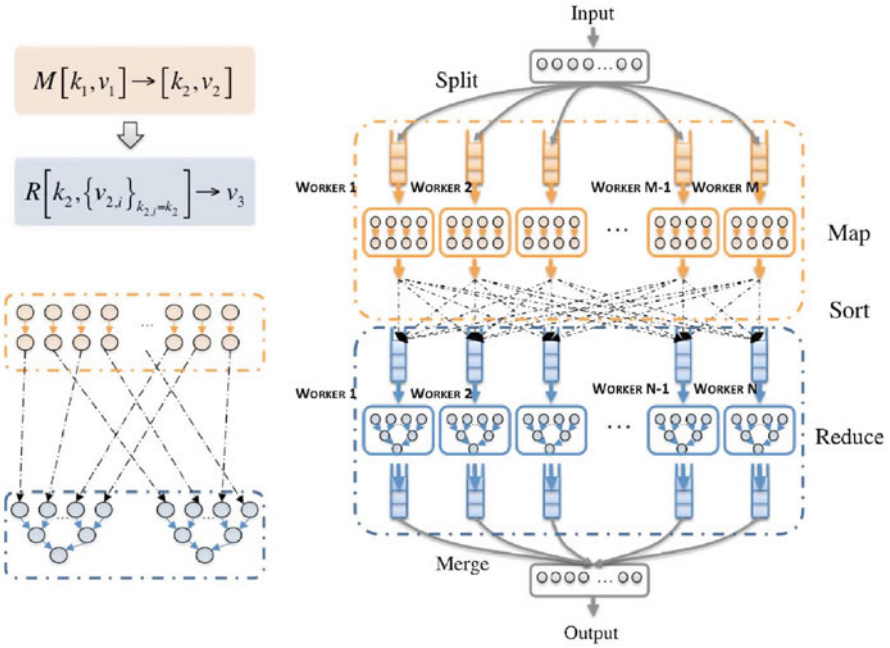

Fig. 5.1 MapReduce primitives and runtime

- *K-means*: The K-means clustering algorithm can be represented as an iterative sequence of (M, R) tasks that run until the stop criteria are met. M represents the assignation of points to clusters and R the recalculation of the cluster centroids. K-means is illustrated in Fig. 5.2.
- *Expectation Maximization*: The EM algorithm for Gaussian mixtures is represented by iterations of (M, R, R, R) tasks running until convergence. M corresponds to the E-step of the algorithm, while (R, R, R) correspond to the M-step that calculates the mixture weights a_i, means $\bar{\mu_k}$ and covariance matrices Σ_k, respectively. EM is illustrated in Fig. 5.3.
- *Support Vector Machine*: The resolution of the dual SVM problem using the Sequential Minimal Optimization (SMO) [20] is represented by iterations of (M, M, R, M) tasks running until convergence. These tasks reproduce the identification of the two Lagrange multipliers to be optimized in each iteration, and their analytic calculation. The SVM is illustrated in Fig. 5.4.

5.1.2 Integration Phase

The *integration* of parallel MapReduce tasks into diverse computing platforms spans a wide and heterogeneous variety of parallel system architectures. Originally, internet-scale companies decomposed indexing and log-processing jobs into Map

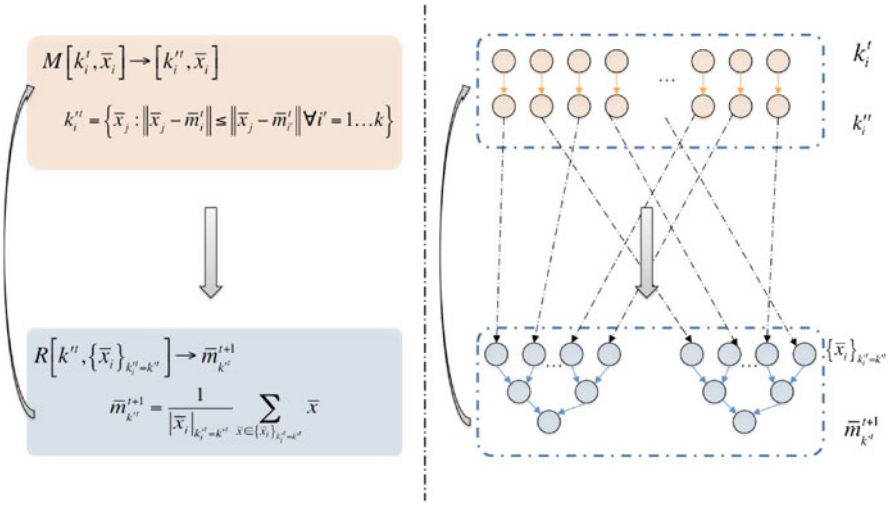

Fig. 5.2 Decomposition of K-means into MapReduce tasks

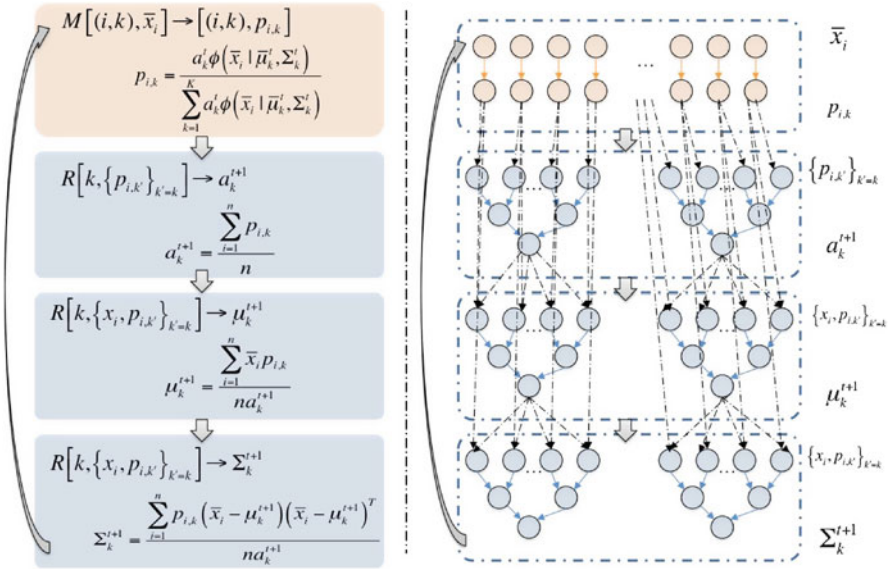

Fig. 5.3 Decomposition of EM using Gaussian mixtures into MapReduce tasks

and Reduce tasks that were executed in batches on top of a distributed file system hosted by hundreds or thousands of commodity nodes. Its proven reliability in production, along with its symbiosis towards virtualized environments, led the MapReduce model to be one of the key data processing paradigms of cloud service infrastructures. Research initiatives have investigated the applicability of

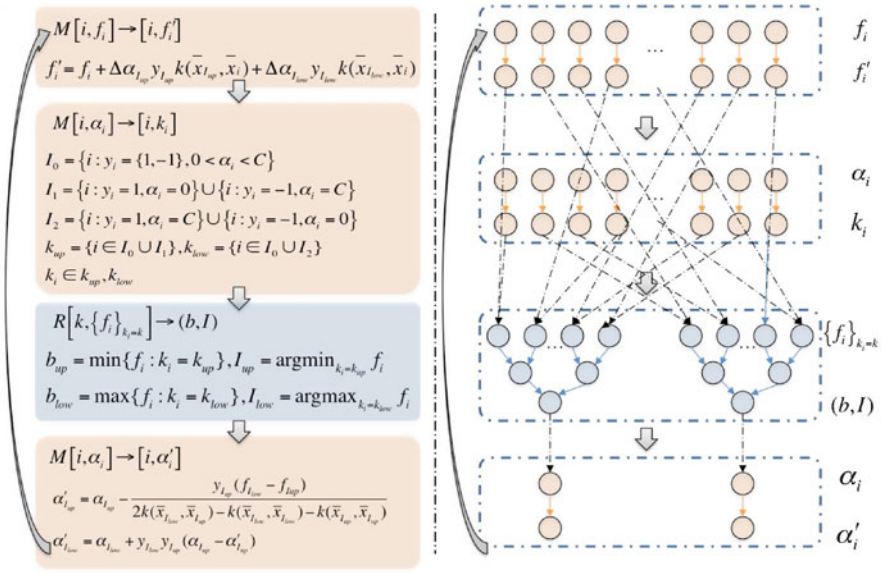

Fig. 5.4 Decomposition of SVM into MapReduce tasks

this model in scientific environments and enterprise analytics, and have tested the implementation of MapReduce tasks on alternative platforms, such as multicore or MPPs (GPUs, Cell microprocessors or FPGAs), aiming to boost the performance of computationally expensive jobs.

In this chapter, a hybrid solution that boosts the computational throughput of commodity nodes is proposed, based on the integration of multiple MPPs into the MapReduce runtime. For this purpose, a programming model to orchestrate MPPs is developed. In order to test the computational capabilities of this solution, a multiclass Support Vector Machine (SVM) is implemented for this hybrid system and its performance results for large datasets reported.

The rest of this chapter is organized as follows: Sect. 5.2 reviews previous initiatives that accelerate the execution of data intensive MapReduce jobs, either by optimizing the cluster runtimes or exploiting the capabilities of massively parallel platforms. Section 5.3 enumerates the research contributions presented by this work. Our proposed unified heterogeneous architecture, is described in Sect. 5.4. The decomposition of the SVM problem into MapReduce tasks and its integration into the GPU cluster architecture is explained in Sect. 5.5. Section 5.6 contains details of the performance gain provided by our massively threaded implementation. The conclusions of this work are presented in Sect. 5.7.

5.2 Related Work

In general, the research efforts for the performance improvement of large-scale Machine Learning algorithms, expressed as MapReduce jobs, can be classified into two families, namely *cluster category* and *multiprocessor category*.

Cluster Category: These efforts focus on the adaptation of the cluster runtime to satisfy the particular needs of Machine Learning algorithms and facilitate their integration into clusters. These needs have been identified as: (1) support for iterative jobs, (2) static and bariable data types, and (3) dense and sparse BLAS operations. Ekanayake et al. [10] presented *Twister*, a modified runtime that accommodates multiple Machine Learning algorithms by supporting long-running stateful tasks. Ghoting et al. [11] designed *System-ML*, a declarative language to express Machine Learning primitives and simplify direct integration into clusters. The Apache *Mahout* [1] project compiled a library of the most popular MR-able algorithms for the standard Hadoop implementation of MapReduce.

Multiprocessor Category: In this category MapReduce jobs are scattered and gathered among multiple processing cores on a shared-memory multiprocessor device or multiple devices hosted by interconnected nodes. Typically the processing units in these systems are constructed to run tens of threads simultaneously reducing the load of MapReduce tasks assigned to each thread, while increasing the degree of parallelism. Communication between cores is carried out through the shared-memory hierarchy. Popular systems of this category are Phoenix (multicore) [27], Mars (GPU) [13], CellMR (Cell) [21] and GPMR [22].

The hybrid MapReduce runtime proposed in this chapter is unique in the sense that it combines the best of both worlds to deliver an efficient framework that meets the specific needs of Machine Learning algorithms, and produces up to two orders of magnitude of acceleration using massively threaded hardware.

Particularly for the case of SVM implementations for shared-memory multiprocessors, Chu et al. [23] provide an SVM solver for multicore based on MapReduce jobs obtained through Statistical Query & Summation. Similarly, researchers have been focused on the GPU adaptation of dual form SVMs for binary and multiclass classification [3, 14]. Specifically for the case of SVM implementations in clusters, Chang et al. [6] provide the performance and scalability analysis of a deployment of their PSVM algorithm on Google's MapReduce infrastructure.

5.3 Research Contributions

Typically, both system categories, *Cluster* and *Multiprocessor*, have shown complementary characteristics. Batch processing systems running on commodity clusters provide high reliability through redundancy and near-linear scalability by adding nodes to the cluster for a low cost. Nevertheless, by nature, its intensive cross-machine communication leads to higher latencies and increased complexity for

computer cluster administrators. On the contrary, shared-memory multiprocessors do not have any built-in reliability mechanism and their scalability is limited by the number of processing cores and the capacity of the memory hierarchy in place. In these devices, latencies in cross-processor communication are orders of magnitude lower, and single-node execution drastically reduces the system administration complexity. Ideally, a unified system including the benefits of both solutions and meeting the needs of Machine Learning algorithms is desired, in order to execute these algorithms on large scale datasets and obtain the results on a timely manner.

The authors of this chapter believe that both categories can be merged to create a unified heterogeneous MapReduce framework and increase the computational throughput of individual nodes. The contributions of this new hybrid system are the following:

1. *Runtime Adaptation*: The original MapReduce runtime was not designed specifically for Machine Learning algorithms. Even though libraries, such as Mahout [1], implement a variety of classic algorithms, the framework has inherent inefficiencies that prevent it from providing timely responses. Our proposed hybrid design integrates a series of modifications to accommodate some of the common needs of Machine Learning algorithms. These runtime modifications are introduced next:
   - *Iterative MapReduce Jobs*: Most Machine Learning algorithms are iterative. The state of the algorithm is maintained through iterations and is reutilized towards the resolution of the problem in each step. Like *Twister* [10], our solution enables executing long-running iterative jobs that keep a state between iterations.
   - *Static and Variable Data Support*: Most iterative Machine Learning algorithms define two types of data: *static* and *variable*. Static data is read-only and is utilized in every iteration, while variable data can be modified and is typically of smaller size. In order to minimize data movements and memory transfers, our runtime allows specifying the nature of the data.
   - *Dense & Sparse BLAS*: The execution of a task may require as input the results of a dense or sparse BLAS operation. Our solution enables interleaving massively threaded BLAS operations to prepare the input data of M and R steps.
2. *MPP Integration*: As opposed to Mars [13], Phoenix [27] and CellMR [21], which were constructed to run MapReduce jobs within a single isolated multiprocessor and not designed to scale out, our solution takes a different approach based on the integration of MPPs into the existing MapReduce framework as coprocessors. GPMR [22] follows the same direction, but keeping the same runtime and not optimizing it for Machine Learning algorithms.
3. *MPP Orchestration*: A programming model to manage multiple MPPs towards the execution of MapReduce tasks is presented. We use an abstraction, called *Asynchronous Port-based Programming*, which allows creating coordination primitives, such as Scatter-Gather.

4. *Massively Threaded SVM*: While implementations of SVM solvers for multiprocessors and clusters provided satisfactory performance as part of isolated experiments, to the best of our knowledge, this work pioneers the execution of a multiclass SVM on a topology of multiple MPPs intertwining tens of CPU threads and thousands MPP threads collaboratively towards an even faster resolution of the SVM training problem.

5.4 A Unified Heterogeneous Architecture

In this section we provide an overview of the foundations of MapReduce-based batch processing systems. We take the Hadoop architecture as a reference due to its popularity and public nature. First, we explain the characteristics and principles of operation of currently existing data processing nodes, called *Data Nodes (DN)*. Then, we proceed to introduce our modifications by integrating more powerful nodes composed by multiple Massively Parallel Processor (MPP) devices; we call these nodes *MPP Nodes* (MPPN). DNs and MPPNs may coexist within a MapReduce cluster, nevertheless, they are meant to address MapReduce jobs with different requirements: DNs should work on batch, high latency jobs, whereas MPPNs would take responsibility of compute intensive jobs.

5.4.1 MapReduce Architecture Background

Typically, the architecture of MapReduce and MapReduce-like systems consists of two layers: (i) a data storage layer in the form of a Distributed File System (DFS) responsible of providing scalability to the system and reliability through replication of the files, and (ii) a data processing layer in the form of a MapReduce Framework (MRF) responsible of distributing and load balancing tasks across nodes. Files in the DFS are broken into blocks of fixed size and distributed among the DNs in the cluster. The distribution and load balancing is managed centrally in a node called *NameNode* (NN). The NN does not only contain metadata about the files in the DFS, but also manages the replication policy. The MRF follows a master-slave paradigm. There is a single master, called *JobTracker*, and multiple slaves, called *TaskTrackers*. The JobTracker is responsible of scheduling MapReduce jobs in the cluster, along with maintaining information about each TaskTracker's status and task load. Each job is decomposed into MapReduce tasks that are assigned to different TaskTrackers based on locality of the data and their status. In general, the output of the Map task is materialized to the disk before proceeding to the Reduce task. The Reduce task may get shuffled input data from different DNs. Periodically, TaskTrackers sent a heartbeat to the JobTracker to keep it up to date. Typically, TaskTrackers are single or dual threaded and consequently, can launch one or two Map or Reduce tasks simultaneously. Hence, each task is single-threaded and work on a single block point by point sequentially. The architecture is illustrated in Fig. 5.5.

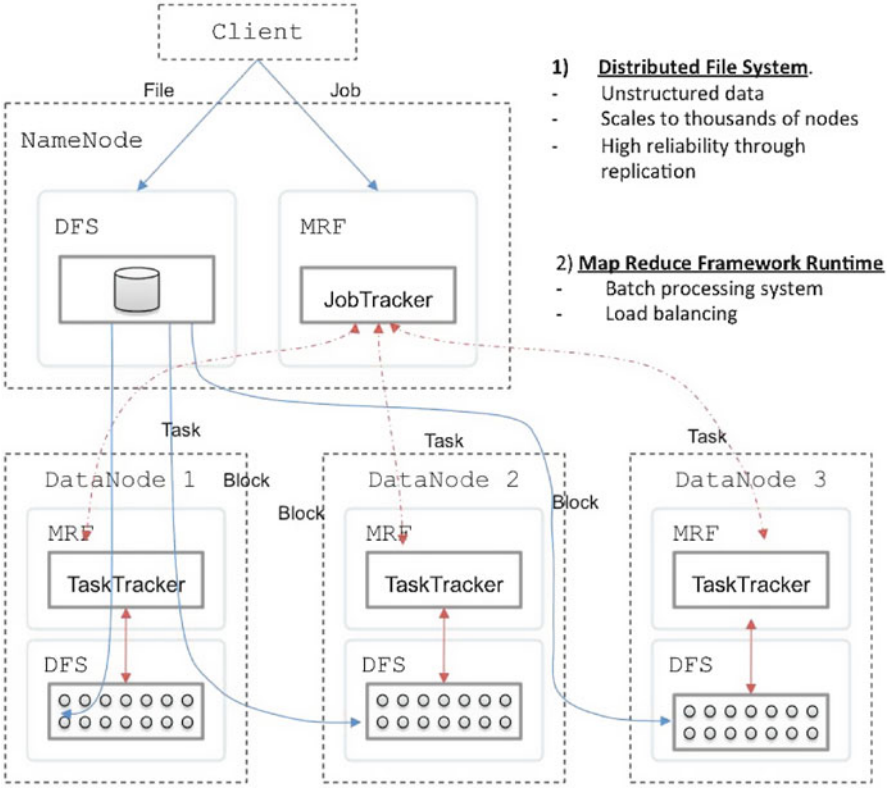

Fig. 5.5 The MapReduce architecture

5.4.2 MPP Integration

We propose the addition of massively parallel processors in order to increase the computational capabilities of the DNs. Currently, DNs use a single thread to process the entire set of data points confined on a given block. This is shown in Fig. 5.6. Parallelism is achieved through the partitioning of data into blocks and the concurrent execution of tasks on different nodes, nevertheless, this setup does not leverage fine-grained parallelism, which can be predominant on data mining algorithms. Fortunately, the introduction of MPPs enables MapReduce tasks to be carried out by hundreds or thousands of threads, giving to each thread one or few data points to work with. This is described in Fig. 5.7. The main differences between DNs and MPPNs are the following:

- *Multithreading*: In DNs the TaskTracker assigns the pair *(Task, Block)* to a single core. Then, the thread running on that core executes the MapReduce function point by point in the block sequentially. On the contrary, in MPPNs the

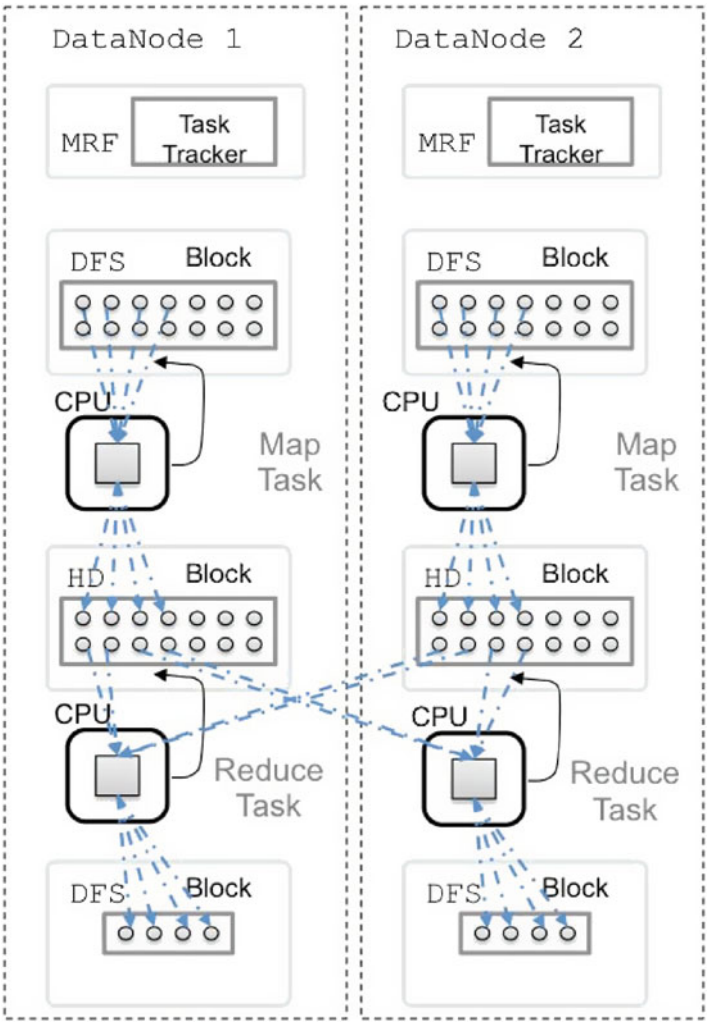

Fig. 5.6 Data node (DN)

TaskTracker assigns the pair *(Task, Block)* to a massively threaded multiprocessor device. Then the device launches simultaneously hundreds or thousands of threads that will execute the same task on multiple data points simultaneously.

- *Pipelining*: In DNs the intermediate result generated by the Map task is materialized by writing the result locally in the node. Before the execution of the Reduce task, the intermediate result is read from the disk and possibly transmitted over the network to a different DN as part of the shuffling process. On the contrary, MPPNs do not materialize the intermediate result. The output of the Map task is kept on the MPP memory and, if necessary, is forwarded to a different device as part of the shuffling process.

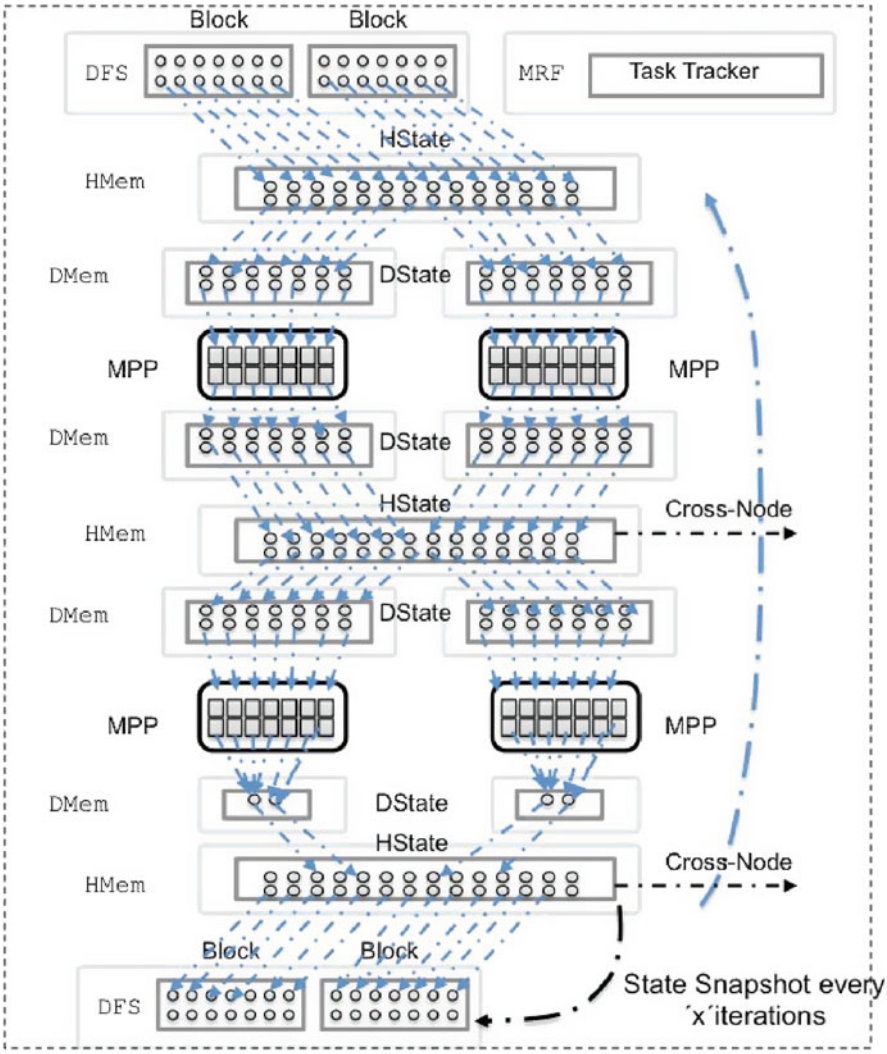

Fig. 5.7 Massively parallel processor node (MPPN)

- *Communication*: In DNs the shuffling process requires slower cross-machine communication leading to increased latency between MapReduce operators. On the contrary, the shuffling process in MPPNs is carried out through message passing between host CPU threads.
- *Iteration*: In DNs a job is terminated after the conclusion of the *R* step. Any additional iteration would be executed as an independent job. MPPNs provide support for iterative algorithms allowing repeatable tasks to be part of the same long-running iterative job.

5.4.3 MPP Orchestration

In general, DNs running on commodity hardware are single or dual threaded. Each CPU thread operates on a different data block, and since the results of each task are materialized to the disk, synchronization between CPU threads is not necessary. Nevertheless, the introduction of MPPs into data nodes requires the interaction of two different threading models, the classic CPU threads and the MPP threads.

As opposed to CPU threads, which are heavy, MPP threads are lighter, they have fewer registers at their disposal and will be slower, but can be launched simultaneously in groups toward the execution of the same task. Furthermore, the fact that MPP threads will run distributed across multiple devices within the same node, raises a challenge not only on the efficient coordination of thousands of these threads towards the collaborative execution of an algorithm, but also on the responsiveness and error handling of the devices running these threads.

In this section, we propose an event-driven model to orchestrate both CPU and MPP threads towards the execution of MapReduce tasks. Unlike ordinary event-driven libraries, which usually directly build upon the asynchronous operations, the method proposed in this chapter is based on the principles of Active Messages [25] and the abstraction layer provided by the Concurrency and Coordination Runtime (CCR) [7]. These abstractions are: the *Port*, the *Arbiter* and the *Dispatcher Queue*.

Figure 5.8 illustrates the three abstractions. The *Port* is an event channel in which messages are received. Posting a message to a port is a non-blocking call and the calling CPU thread continues to the next statement. The *Arbiter* decides which registered callback method should be executed to consume the message or messages. Once the method is selected, the arbiter creates the pair *(Task, Block)*, which is passed to the Dispatcher Queue associated to the port. This is an indirection that enables the creation of high-level coordination primitives. Some of the possible primitives are discussed later in this section. Each port is assigned a *Dispatcher Queue* and multiple ports can be associated with the same Dispatcher Queue. The Dispatcher Queue consists of a thread pool composed by one or more CPU threads. Available threads pick *(Task, Block)* pairs passed by the Arbiter and proceed to the execution of the task on the corresponding data block in the MPP. The MPP is stageful. It keeps a state in the memory of the device (DState) across iterations to minimize memory transfers. If necessary, it synchronizes with the state in the host memory (HState).

Next, some of the coordination primitives that can be constructed using these three abstractions are introduced:

- *Single Item Receiver*: It registers callback X to be launched when a single message of type M is received in Port A.
- *Multiple Item Receiver*: Registers callback X to be launched when n messages are received in Port A. p messages will be of type M (success) and q messages of exception type (failures), so that $p + q = n$.
- *Join Receiver*: Registers callback X to be launched when one message of type M is received in Port A and another in Port B.

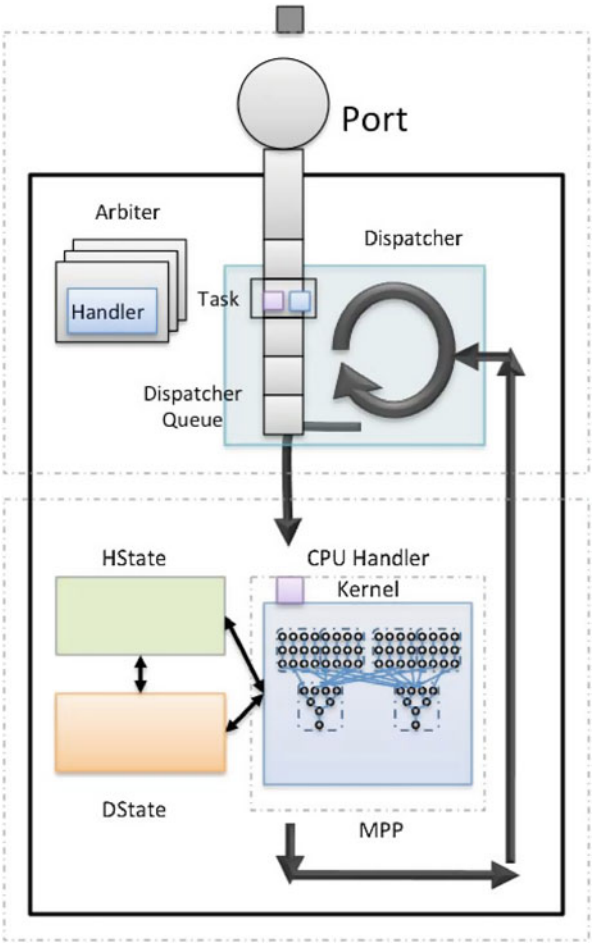

Fig. 5.8 Port abstraction and its components

- *Choice*: Registers callback X to be launched when one message of type M is received in Port A and registers callback Y to be launched when one message of type N is received in Port A.

In the context of the MapReduce framework, these abstractions are utilized to construct a Scatter-Gather mechanism in which a master CPU thread distributes MapReduce tasks among available MPP devices, and, upon termination, these return the control and the results back to the master thread. Each MPP device will have a Port instance for every type of MapReduce task, a single Arbiter and a single Dispatcher Queue. The Arbiter will register each Port following the *Single Item Receiver* primitive with the assigned callback method that represents the MapReduce task. Requests to launch a task will contain a pointer to the data block to be manipulated and the response port in which all the responses need to be gathered. The callback method will contain the invocation of the computing kernel,

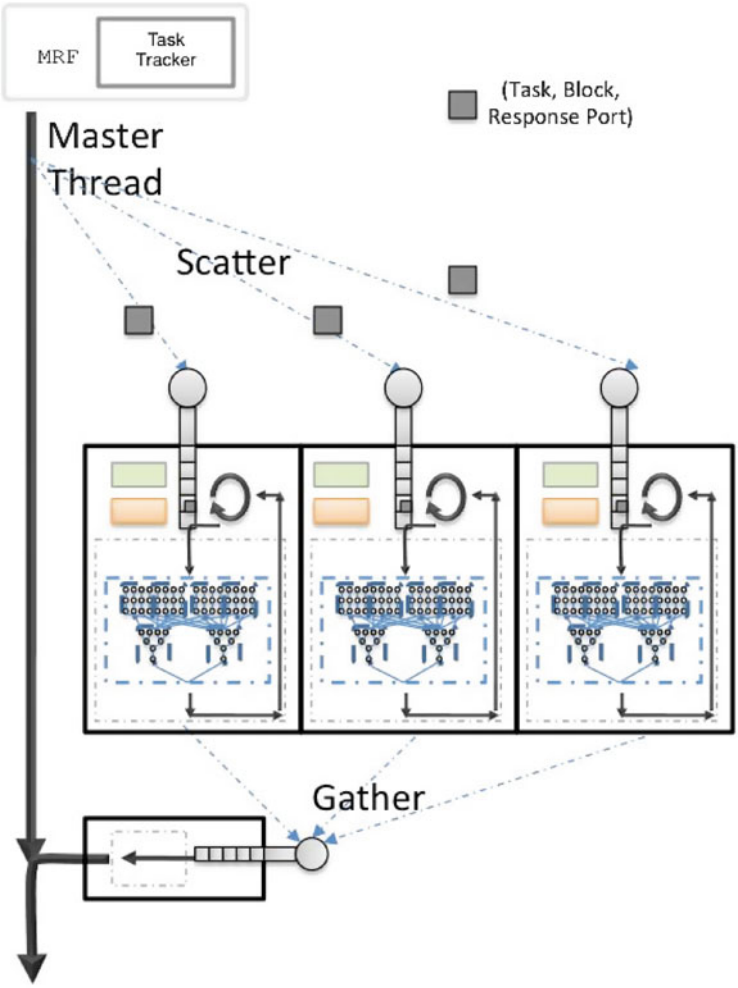

Fig. 5.9 Scatter-Gather using ports and MPPs

which spawns hundreds of MPP threads that operate the data simultaneously. The response Port follows a *Multiple Item Receiver* primitive and is registered to launch a callback method in the master thread when all the devices have answered.

This Scatter-Gather mechanism is illustrated in Fig. 5.9. One of the key benefits of this event-driven model is that not only enables coordinating multiple MPP devices towards the execution of MapReduce tasks, but also deals with the potential failure of any device, which is fundamental to conserve the robustness of the MapReduce framework.

5.5 Massively Multithreaded SVM

In order to investigate the performance of the architecture proposed in Sect. 5.4, in this section we take the SVM classifier, explore its decomposition into a MapReduce job, and launch it on a MPPN composed by multiple MPP devices. First, we provide a brief introduction to the SVM classification problem. Second, we describe the MapReduce job that solves the training phase on a single MPP device. Third, we coordinate various MPP devices using the Scatter-Gather primitive to solve a larger classification problem. Figures in this section hide the Port, Arbiter, and Dispatcher Queue components to facilitate the understanding of MapReduce task sequences.

5.5.1 Binary SVM

The binary SVM classification problem is defined as follows: Find the classification function that, given l examples $(\bar{x}_1, y_1), \ldots, (\bar{x}_l, y_l)$ with $\bar{x}_i \in R^n$ and $y_i \in \{-1, 1\}$ $\forall i$, predicts the label of new unseen samples $\bar{z}_i \in R^n$. This is achieved by solving the following regularized learning problem, where the regularization is controlled via C.

$$\min_{f \in H} C \sum_{i=1}^{l} (1 - y_i f(\bar{x}_i))_+ + \frac{1}{2} \| f \|_k^2 , \tag{5.3}$$

where $(k)_+ = \max(k, 0)$. Then slack variables ξ_i are introduced to overcome the problem introduced by its non-differentiability:

$$\min_{f \in H} C \sum_{i=1}^{l} \xi_i + \frac{1}{2} \| f \|_k^2 \tag{5.4}$$

subject to: $y_i f(x_i) \geq 1 - \xi_i$ and $\xi_i \geq 0, i = 1, \ldots, l$. The dual form of this problem is given by:

$$\max_{\alpha \in R^l} \sum_{i=1}^{l} \alpha_i - \frac{1}{2} \alpha^T K \alpha \tag{5.5}$$

subject to: $\sum_{i=1}^{l} y_i \alpha_i = 0$ and $0 \leq \alpha_i \leq C$, $i = 1, \ldots, l$, where $K_{ij} = y_i y_j k\left(\bar{x}_i, \bar{x}_j\right)$ is a kernel function. Equation 5.5 is a quadratic programming optimization problem and its solution defines the classification function:

$$f(x) = \sum_{i=1}^{l} y_i \alpha_i k(\bar{x}, \bar{x}_i) + b, \tag{5.6}$$

where b is an unregularized bias term.

5.5.2 MapReduce Decomposition of the SVM

The binary SVM problem can be solved using the *Sequential Minimal Optimization* (SMO) algorithm [20]. SMO converts the dual form of the SVM problem into a large scale *Quadratic Programming* (QP) optimization that can be solved by choosing the smallest optimization problem at every step, which involves only two Lagrange multipliers $(\alpha_{I_{low}}, \alpha_{I_{up}})$. For two Lagrange multipliers the QP problem can be solved analytically without the need of numerical QP solvers. Next, we present SMO as an iterative sequence of MapReduce operators.

First, there are two consecutive Map operators. The first Map updates the values of the classifier function f_i based on the variation of the two Lagrange multipliers, $\Delta\alpha_{I_{low}} = \alpha'_{I_{low}} - \alpha_{I_{low}}$, $\Delta\alpha_{I_{up}} = \alpha'_{I_{up}} - \alpha_{I_{up}}$, their label values $\left(y_{I_{up}}, y_{I_{low}}\right)$ and their associated kernel evaluations:

$$\begin{aligned} f_i' = f_i &+ \Delta\alpha_{I_{up}} y_{I_{up}} k(\bar{x}_{I_{up}}, \bar{x}_i) \\ &+ \Delta\alpha_{I_{low}} y_{I_{low}} k(\bar{x}_{I_{low}}, \bar{x}_i) \end{aligned} \tag{5.7}$$

with $i = 1 \ldots l$. The initialization values for the first Map of the iterative sequence are: $f_i = -y_i$, $\Delta\alpha_{I_{up}} = \Delta\alpha_{I_{low}} = 0$, $\alpha_{I_{low}} = \alpha_{I_{up}} = 0$, $I_{low} = I_{up} = 0$.

$$M\,[i, f_i] \rightarrow \left[i, f_i'\right] \tag{5.8}$$

The second Map classifies the function values f_i' into two groups, k_{up} and k_{low}, according to these filters, in which C is the regularization parameter, $k_i \in k_{up}, k_{low}$.

$$\begin{aligned} I_0 = &\{i : y_i = 1, 0 < \alpha_i < C\} \cup \\ &\{i : y_i = -1, 0 < \alpha_i < C\} \end{aligned} \tag{5.9}$$

$$I_1 = \{i : y_i = 1, \alpha_i = 0\} \tag{5.10}$$

$$I_2 = \{i : y_i = -1, \alpha_i = C\} \tag{5.11}$$

$$I_3 = \{i : y_i = 1, \alpha_i = C\} \tag{5.12}$$

$$I_4 = \{i : y_i = -1, \alpha_i = 0\} \tag{5.13}$$

$$k_{up} = \{i \in I_0 \cup I_1 \cup I_2\} \tag{5.14}$$

$$k_{low} = \{i \in I_0 \cup I_3 \cup I_4\} \tag{5.15}$$

$$M\,[i, \alpha_i] \rightarrow [i, k_i] \tag{5.16}$$

The Reduce operator takes the list of values generated by the Maps and applies a different reduction operator based on the group they belong to. For k_{up} min and arg min are used, while k_{low} requires max and arg max.

$$b_{up} = \min\left\{f_i : k_i = k_{up}\right\} \tag{5.17}$$

$$I_{up} = \arg\min_{k_i = k_{up}} f_i \tag{5.18}$$

$$b_{low} = \max\left\{f_i : k_i = k_{low}\right\} \tag{5.19}$$

$$I_{low} = \arg\max_{k_i = k_{low}} f_i \tag{5.20}$$

The indices $\left(I_{up}, I_{low}\right)$ indicate the Lagrange multipliers that will be optimized.

$$R\left[k, \{f_i\}_{k_i = k}\right] \rightarrow [b, I] \tag{5.21}$$

The last Map uses these indices to calculate the new Lagrange multipliers:

$$\alpha'_{I_{up}} = \alpha_{I_{up}} - \frac{y_{I_{up}}\left(f_{I_{low}} - f_{I_{up}}\right)}{\eta} \tag{5.22}$$

$$\alpha'_{I_{low}} = \alpha_{I_{low}} + s\left(\alpha_{I_{up}} - \alpha'_{I_{up}}\right) \tag{5.23}$$

where

$$s = y_{I_{up}} y_{I_{low}} \tag{5.24}$$

$$\eta = 2k\left(\bar{x}_{I_{low}}, \bar{x}_{I_{up}}\right) - k\left(\bar{x}_{I_{low}}, \bar{x}_{I_{low}}\right) - k\left(\bar{x}_{I_{up}}, \bar{x}_{I_{up}}\right) \tag{5.25}$$

$$M\left[i, \alpha_i\right] \rightarrow \left[i, \alpha'_i\right] \tag{5.26}$$

Convergence is achieved when $b_{low} < b_{up} + 2\tau$, where τ is the stopping criteria.

5.5.3 *Single-MPP Device SVM*

As we advanced in Sect. 5.4.2, unlike single or dual core based MapReduce-like systems, MPP devices can carry out multithreaded MapReduce tasks. For the case of the single-device SVM, the data block provided by the TaskTracker represents the entire training dataset. This data block is further split into subblocks that are passed to the processors in the device. Typically, each processor can run several threads simultaneously, which enables a large number of Map or Reduce tasks being executed in parallel. Figure 5.10 schematically shows the flow of MapReduce tasks on a MPP device. Two versions of the SVM MapReduce job were

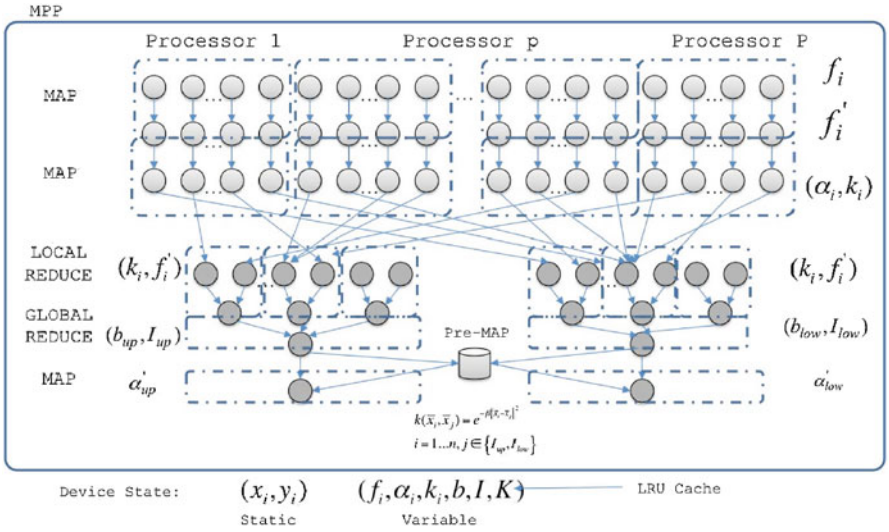

Fig. 5.10 Binary SVM decomposed into MapReduce tasks

constructed, one for each data structure type: *dense* and *sparse*. In general, sparse data structures can reduce memory utilization and data transfer times, which benefits communication within the MapReduce framework. Nevertheless, the performance of sparse algebraic operations in the MPP directly depends on the degree of sparsity of the data and the effect might be adverse, since the duplication of memory accesses caused by the additional indirection can have a negative effect on performance. The dense and sparse matrix-vector multiplications on MPPs used in this work are based on Bell et al. [2] and Vazquez et al. [24], respectively. Their impact on the SVM speed is reflected in Sect. 5.6.3.

5.5.4 Multiple-MPP Device SVM

In Sect. 5.5.3 a data block representing the entire training set was forwarded to one MPP device where MapReduce tasks would be executed iteratively until convergence, without any interaction with other devices. In this section we enhance the decomposition of MapReduce tasks to be able to break the SVM problem into multiple MPP devices. Figure 5.11 describes the interactions between four MPP devices to solve a single SVM problem. The training dataset is split into four data blocks stored in the distributed file system. The TaskTracker, that manages the master thread, forwards the corresponding block to each device. Each device performs the Map operator and a local Reduce on its local data block. The results of the reduce are gathered by the master thread, which carries out a *Global Reduce*

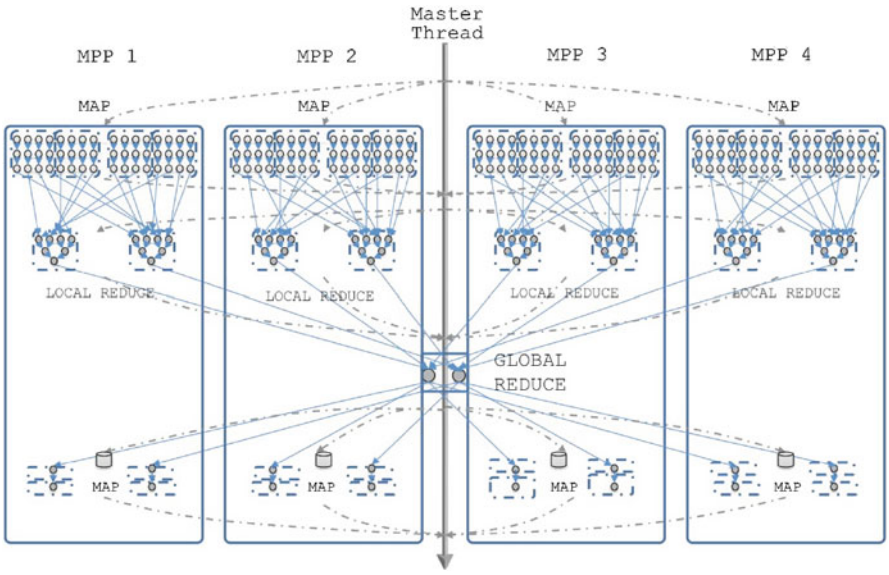

Fig. 5.11 Multiple-MPP device SVM

in order to find (b_{up}, I_{up}) and (b_{low}, I_{low}). Then these values are scattered to the devices in order to update the lagrange multipliers $\left(\alpha'_{I_{up}}, \alpha'_{I_{low}}\right)$. Finally, the master thread synchronizes, checks for convergence, and if required, proceeds to scatter the next Map task to the MPP group.

5.6 Implementation and Experimental Results

In this section we provide implementation details and performance results for the MapReduce jobs presented in Sect. 5.5, along with the incremental performance gain from one method to another. As a baseline for comparison, we take a popular SVM solver, LIBSVM [4], which is a single-threaded version of the SMO algorithm. Then, we compare LIBSVM to the SVM algorithm running on the standard Hadoop platform. Having evaluated these two popular options, we proceed to assess the performance boost obtained from the inclusion of GPUs in MapReduce cluster nodes. Throughout all the experiments the same SVM kernel functions $k(\bar{x_i}, \bar{x_j})$, regularization parameter C, and stopping criteria τ were used.

Table 5.1 Datasets

Dataset	# Training points	# Testing points	# (Features, classes)	(C, β)
WEB	49,749	14,951	(300,2)	(64, 7.8125)
MNIST	60,000	10,000	(780,10)	(10, 0.125)
RCV1	518,571	15,564	(47,236,53)	(1, 0.1)
PROTEIN	17,766	6,621	(357,3)	(10, 0.05)
SENSIT	78,823	19,705	(100,3)	(1, 0.7)

5.6.1 Datasets

SVM training performance comparisons are carried out over five publicly available datasets, WEB [20], MNIST [18], RCV1 [19], PROTEIN [26] and SENSIT [9]. These datasets were chosen based on their computational complexity since they have hundreds of features per data sample. The sizes of these datasets and the parameters used for training are indicated in Table 5.1. The Radial Basis kernel, $k\left(\bar{x}_i, \bar{x}_j\right) = e^{-\beta\left\|\bar{x}_i - \bar{x}_j\right\|^2}$, was used for the training phase throughout all the experiments, as well as $\tau = 0.001$. Multiclass datasets, such as MNIST, RCV1, PROTEIN and SENSIT are decomposed into binary SVM problems following the One-vs-All (OVA) output code. Then, the resulting collection of binary SVMs is solved in parallel as independent MapReduce jobs.

5.6.2 Implementation and Setup

The measurements collected in the next subsection (i.e. Sect. 5.6.3) were carried out in a single machine with a dual socket Intel Xeon E5520 @ 2.26 GHz (8 cores, 16 threads) and 32 GB of RAM.

Hadoop Setup: Using this machine as a host, the SVM algorithm running on Hadoop was executed on 4 Virtual Machines (VMs), with a single core and 4 GB of RAM each. The host ran the Master Node, which contained the NameNode and the JobTracker, while the four VMs ran the DataNodes with the TaskTrackers.

Multiprocessor Setup: This machine also accommodated four GPUs. The multiprocessors utilized are NVIDIA Tesla C1060 GPUs with 240 Stream Processors @ 1.3 GHz. Each GPU has 4 GB of memory and a memory bandwidth of 102 GB/s. Similar to the Hadoop case, the host machine ran the Master Node, which contained the NameNode with the JobTracker, and a MPP Node with four GPU devices. The computing kernels representing MapReduce tasks were implemented using NVIDIA CUDA.

The distribution of resources across different experiments is summarized in Table 5.2. $a + b$ represents a master threads and b device threads. We denote *SD* to the single-GPU experiment and *MD* to the multi-GPU experiment.

Table 5.2 SVM experiments

Experiment	# Host threads	# Virtual machines	# GPU devices	# GPU threads	Host mem (GB)	Device mem (GB)
LIBSVM	1	–	–	–	4	–
Hadoop	1 + 4	4	–	–	4 × 4	–
SD	1 + 1	–	1	1,024	4	4
MD	1 + 4	–	4	4,096	4 × 4	4 × 4

Table 5.3 Performance results for SVM training

Dataset (Non-zero %)		LIBSVM	Hadoop SVM	SD SVM (Dense)	SD SVM (Sparse)	MD SVM (Dense)	MD SVM (Sparse)
WEB	Time (s)	2,364.2	1,698.7	154.3	107.35	73.6	57.3
	Gain (x)	1	1.39	15.32	22.02	32.12	41.26
(3 %)	Accuracy (%)	82.69	82.69	82.69	82.69	82.69	82.69
MNIST	Time (s)	118,943.5	66,753.5	2,010.3	2,321.75	726.9	923.16
	Gain (x)	1	1.78	59.16	51.23	163.63	128.84
(19 %)	Accuracy(%)	95.76	95.76	95.76	95.76	95.76	95.76
RCV1	Time (s)	710,664	231,486	N/A	N/A	N/A	3,686
	Gain (x)	1	3.07	N/A	N/A	N/A	192.75
(0.1 %)	Accuracy(%)	94.67	94.67	N/A	N/A	N/A	94.67
PROTEIN	Time (s)	861	717.5	32.93	39.09	16.06	20.71
	Gain (x)	1	1.2	26.14	22.02	53.61	41.57
(29 %)	Accuracy(%)	70.03	70.03	70.03	70.03	70.03	70.03
SENSIT	Time (s)	8,162	4,295.78	134.670	539.32	58.29	273.96
	Gain (x)	1	1.9	60.61	15.13	140.02	29.79
(100 %)	Accuracy(%)	83.46	83.46	83.46	83.46	83.46	83.46

5.6.3 Experimental Results

In this subsection we provide the performance gain obtained by each architecture/MapReduce task flow compared to the reference implementation for all the datasets: WEB, MNIST, RCV1, PROTEIN and SENSIT. For each of the experiments we present its training time, the measured acceleration with respect to the reference implementation and the accuracy obtained from testing the calculated Support Vectors (SVs) with the test dataset. These results are collected in Table 5.3. The acceleration of the testing phase falls out of the scope this work due to its triviality.

The execution of the Map and Reduce operators, introduced in Sect. 5.5.3, on the standard Hadoop infrastructure yielded a modest performance improvement in the range of ($1.20 \times -3.07\times$) when compared to LIBSVM. Nevertheless, the results obtained from running these same operators on a SD SVM produced an order of magnitude of acceleration in the range of ($15.13 \times -60.61\times$), which is consistent with the values obtained by Catanzaro et al. [3] and Herrero-Lopez et al. [14]. Scaling out the problem to four GPUs (MD SVM) and using the GPU orchestration

model presented in this paper outperformed all the previous solutions producing an overall acceleration in the range of $29.79\times - 192.75\times$. These results also show that the use of sparse data structures is beneficial for cases with high degree of sparsity (WEB and RCV1), while results adverse for the rest. The execution of the sparse MD SVM on the WEB dataset produced a $1.28\times$ gain compared to the dense MD SVM on the same dataset, while the SVM for the RCV1 dataset could not be solved on its dense SVM versions nor single device SVM form since data structures would not fit in the GPU memory. The SVM for the RCV1 dataset was solved only for the sparse MD SVM version, which produced the highest acceleration ($192.75\times$) for this set of experiments. Finally, it is necessary to point out that no accuracy loss was observed and that the same classification results were obtained on all the testing datasets across all the different systems.

5.7 Conclusions and Future Work

In this chapter, our goal was to accelerate the execution of Machine Learning algorithms running on a MapReduce cluster, while maintaining the reliability and simplicity of its infrastructure. For this purpose, we integrated massively threaded multiprocessors into the nodes of the cluster, and proposed a concurrency model that allows orchestrating host threads and thousands of multiprocessor threads spread throughout different devices so as to collaboratively solve MapReduce jobs. In order to verify the validity of this system, we decomposed the SVM algorithms into MapReduce tasks, and created a combined solution that distills the maximum degree of fine-grained parallelism. The execution of the SVM algorithm in our proposed system yielded an acceleration in the range of $29.79\times - 192.75\times$, when compared to LIBSVM and in the range of $15.68\times - 91.83\times$, when compared to the standard Hadoop implementation. To the best of our knowledge this is the shortest training time reported on these datasets for a single machine, without leaving commodity hardware nor the MapReduce paradigm. In the future, it is planned to explore the possibility of maximizing the utilization of the GPUs in the MPP Node through the execution of multiple MapReduce tasks concurrently in each device.

Acknowledgements This work was supported by the Basque Government Researcher Formation Fellowship BFI.08.80.

References

1. Apache.org: Apache mahout: scalable machine-learning and data-mining library. http://mahout.apache.org/
2. Bell, N., Garland, M.: Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008–004, NVIDIA Corporation (2008)

3. Catanzaro, B., Sundaram, N., Keutzer, K.: Fast support vector machine training and classification on graphics processors. In: ICML'08: Proceedings of the 25th International Conference on Machine Learning, Helsinki, pp. 104–111. ACM, New York (2008). doi:http://doi.acm.org/10.1145/1390156.1390170
4. Chang, C.C., Lin, C.J.: LIBSVM: a library for support vector machines (2001). Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm
5. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: a distributed storage system for structured data. In: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation – Volume 7, OSDI'06, Seattle, pp. 205–218. USENIX Association, Berkeley (2006)
6. Chang, E.Y., Zhu, K., Wang, H., Bai, H., Li, J., Qiu, Z., Cui, H.: Psvm: parallelizing support vector machines on distributed computers. In: NIPS (2007). Software available at http://code.google.com/p/psvm
7. Chrysanthakopoulos, G., Singh, S: An asynchronous messaging library for c#. In: Proceedings of the Workshop on Synchronization and Concurrency in Object-Oriented Languages, OOPSLA 2005, San Diego (2005)
8. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008). doi:http://doi.acm.org/10.1145/1327452.1327492
9. Duarte, M.F., Hu, Y.H.: Vehicle classification in distributed sensor networks. J. Parallel Distrib. Comput. **64**, 826–838 (2004)
10. Ekanayake, J., Li, H., Zhang, B., Gunarathne, T., Bae, S.H., Qiu, J., Fox, G.: Twister: a runtime for iterative mapreduce. In: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, HPDC'10, Chicago, pp. 810–818. ACM, New York (2010)
11. Ghoting, A., Krishnamurthy, R., Pednault, E., Reinwald, B., Sindhwani, V., Tatikonda, S., Tian, Y., Vaithyanathan, S.: Systemml: declarative machine learning on mapreduce. In: Proceedings of the 2011 IEEE 27th International Conference on Data Engineering, ICDE'11, Hannover, pp. 231–242. IEEE Computer Society, Washington (2011). doi:http://dx.doi.org/10.1109/ICDE.2011.5767930.
12. Hadoop: hadoop.apache.org/core/
13. He, B., Fang, W., Luo, Q., Govindaraju, N.K., Wang, T.: Mars: a mapreduce framework on graphics processors. In: PACT'08: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, Toronto, pp. 260–269. ACM, New York (2008). doi:http://doi.acm.org/10.1145/1454115.1454152
14. Herrero-Lopez, S., Williams, J.R., Sanchez, A.: Parallel multiclass classification using svms on gpus. In: GPGPU'10: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, Pittsburgh, pp. 2–11. ACM, New York (2010). doi:http://doi.acm.org/10.1145/1735688.1735692
15. Hillis, W.D., Steele, G.L., Jr.: Data parallel algorithms. Commun. ACM **29**(12), 1170–1183 (1986). http://doi.acm.org/10.1145/7902.7903
16. Kearns, M.: Efficient noise-tolerant learning from statistical queries. J. ACM **45**(6), 983–1006 (1998). doi: http://doi.acm.org/10.1145/293347.293351
17. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. SIGOPS Oper. Syst. Rev. **44**, 35–40 (2010). doi:http://doi.acm.org/10.1145/1773912.1773922
18. Lecun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proc. IEEE **86**, 2278–2324 (1998)
19. Lewis, D.D., Yang, Y., Rose, T.G., Li, F.: Rcv1: a new benchmark collection for text categorization research. J. Mach. Learn. Res. **5**, 361–397 (2004)
20. Platt, J.C.: Fast training of support vector machines using sequential minimal optimization, pp. 185–208. MIT, Cambridge (1999)
21. Rafique, M.M., Rose, B., Butt, A.R., Nikolopoulos, D.S.: Cellmr: a framework for supporting mapreduce on asymmetric cell-based clusters. In: IPDPS'09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing, Rome, pp. 1–12. IEEE Computer Society, Washington (2009). doi:http://dx.doi.org/10.1109/IPDPS.2009.5161062

22. Stuart, J.A., Owens, J.D.: Multi-gpu mapreduce on gpu clusters. In: Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium, IPDPS'11, Anchorage, pp. 1068–1079. IEEE Computer Society, Washington (2011)
23. tao Chu, C., Kim, S.K., an Lin, Y., Yu, Y., Bradski, G., Ng, A.Y., Olukotun, K.: Map-reduce for machine learning on multicore. In: Proceedings of NIPS, pp. 281–288 (2007)
24. Vazquez, F., Ortega, G., Fernandez, J., Garzon, E.: Improving the performance of the sparse matrix vector product with gpus. In: 2010 IEEE 10th International Conference on Computer and Information Technology (CIT), Bradford, pp. 1146–1151 (2010). doi:10.1109/CIT.2010.208
25. von Eicken, T., Culler, D.E., Goldstein, S.C., Schauser, K.E.: Active messages: a mechanism for integrated communication and computation. SIGARCH Comput. Archit. News **20**, 256–266 (1992)
26. Wang, J.Y.: Application of support vector machines in bioinformatics. Master's thesis, National Taiwan University, Taipei, Taiwan (2002)
27. Yoo, R.M., Romano, A., Kozyrakis, C.: Phoenix rebirth: scalable mapreduce on a large-scale shared-memory system. In: IISWC'09: Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC), Austin, pp. 198–207. IEEE Computer Society, Washington (2009). doi:http://dx.doi.org/10.1109/IISWC.2009.5306783