NIMBLE: A Toolkit for the Implementation of Parallel Data Mining and Machine Learning Algorithms on MapReduce

Amol Ghoting, Prabhanjan Kambadur, Edwin Pednault, and Ramakrishnan Kannan IBM Thomas J. Watson Research Center, Yorktown Heights, NY {aghoting, pkambadu, pednault, ramkrish}@us.ibm.com

ABSTRACT

In the last decade, advances in data collection and storage technologies have led to an increased interest in designing and implementing large-scale parallel algorithms for machine learning and data mining (ML-DM). Existing programming paradigms for expressing large-scale parallelism such as MapReduce (MR) and the Message Passing Interface (MPI) have been the de facto choices for implementing these ML-DM algorithms. The MR programming paradigm has been of particular interest as it gracefully handles large datasets and has built-in resilience against failures. However, the existing parallel programming paradigms are too low-level and ill-suited for implementing ML-DM algorithms. To address this deficiency, we present NIMBLE, a portable infrastructure that has been specifically designed to enable the rapid implementation of parallel ML-DM algorithms. The infrastructure allows one to compose parallel ML-DM algorithms using reusable (serial and parallel) building blocks that can be efficiently executed using MR and other parallel programming models; it currently runs on top of Hadoop, which is an open-source MR implementation. We show how NIMBLE can be used to realize scalable implementations of ML-DM algorithms and present a performance evaluation.

Categories and Subject Descriptors

D.2.2 [Design Tools and Techniques]: Software libraries

General Terms

Design, Performance, Algorithms

Keywords

Data mining, Machine learning, MapReduce, Tasks

1. INTRODUCTION

In the last decade, advances in data collection and storage technologies have allowed organizations to accumulate vast amounts of data. Organizations, ranging from small Internet companies to large businesses, are collecting terabytes of data on a daily basis

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

KDD'11, August 21–24, 2011, San Diego, California, USA. Copyright 2011 ACM 978-1-4503-0813-7/11/08 ...\$10.00.

with the intent of understanding and improving their business processes. Hence, there is a growing need to run machine learning and data mining (ML-DM) algorithms on very large datasets. To complicate matters, recent trends in hardware have brought new challenges to the programming community; multi-core and many-core systems have become ubiquitous and advances in single-threaded processor performance have reached a standstill. Increasing dataset sizes and the shift towards parallel scaling in hardware have necessitated that ML-DM algorithms be parallelized in the immediate future.

While the need for parallelization in ML-DM is evident, few solutions to satisfy this need exist today. For more than two decades, parallel database products such as those offered by IBM, Oracle, Teradata, and Netezza have provided a partial means to realize a parallel implementation of ML-DM algorithms. However, it is difficult to restructure parallel ML-DM algorithms to suit the declarative SQL-based programming interface exposed by these database products. Furthermore, large-scale installations of these products are extremely expensive and are not an affordable option in most cases. Consequently, the past few years have seen researchers moving to parallelization of ML-DM based on imperative specification that use open-source products. One approach has been to use low-level abstractions for parallelization such as those provided by Pthreads [10], OpenMP [9], TBB[5], and PFunc [19] for sharedmemory systems, and MPI [8] for distributed-memory systems. As these programming abstractions originated in the high performance computing community, they are expressive and powerful, but are arcane; hence, they make implementation cumbersome.

In recent years, an increasing number of programmers have migrated to the MapReduce (MR) programming model. The MR programming model was designed to simplify the processing of large files on a parallel system through user-defined Map and Reduce primitives. A MR job consists of two phases — a Map phase and a Reduce phase. During the Map phase, the user-defined Map primitive transforms the input data into (key, value) pairs in parallel. These pairs are stored and then sorted by the system so as to accumulate all values for each key. During the Reduce phase, the user-defined Reduce primitive is invoked on each unique key with a list of all the values for that key; usually, this phase is used to perform aggregations. Finally, the results are output in the form of (key, value) pairs. Each key can be processed in parallel during the Reduce phase. Hadoop [1], an open-source implementation of the MR programming model, has emerged as a vastly popular platform for parallelization in industry and academia. A user can perform parallel computations by submitting one or more MR jobs to Hadoop. One of the key advantages of Hadoop is that it is capable of running on large commodity clusters and recovering from both data as well as compute node failures. Companies like Facebook, Yahoo!, Amazon, Baidu, AOL, and IBM use Hadoop on a daily basis. Furthermore, many data management platforms such as HBase [2], Hive [3], PIG [22], and JAQL [6] build off it as well. Hence, we conjecture that Hadoop will be a stable, growing, and well-maintained platform.

While the above mentioned programming models are very popular in their particular domains, we believe that they have a set of limitations that make them ill-suited to the implementation of parallel ML-DM algorithms. To get a better grasp of these limitations, we first characterize the requirements of the ML-DM community when it comes to implementing their parallel algorithms.

Parallelization with limited effort: Efficient parallelization of an existing sequential implementation of an algorithm is a non-trivial process as factors such as communication, data management, and scheduling have to be carefully considered¹. Parallelization using existing frameworks such as PThreads [10], OpenMP [9], TBB [5], and PFunc [19] for shared-memory systems, and MPI [8] for distributed-memory systems is time-consuming and requires indepth knowledge of parallel programming. Most of these frameworks have been designed for "optimal parallelization"; that is, they allow the user to control data placement, scheduling, and communication so as to realize the best possible parallel implementation. However, the requirements of ML-DM researchers and developers are different — they need a means to both implement and evaluate their algorithms in a limited amount of time and they do not necessarily want to spend time concentrating on an "optimal parallelization". Therefore, ML-DM researchers and developers have shied away from implementing parallel versions of their algorithms. Ideally, parallel implementation should require little additional work relative to the sequential implementation; this requires that the parallelization interface should support programming patterns that are typically used during sequential implementation. For example, serial implementations of decision tree induction typically follow a divide-and-conquer approach and operate by first finding the best splitting criterion for a dataset, then partitioning the dataset on this criterion, and finally recursively processing these partitions until the recursion terminates. In this case, to make things simple and natural for the implementer, the parallelization framework should support recursive programming patterns.

Support for rapid prototyping: The knowledge discovery process is iterative in nature — researchers and users often need to evaluate a variety of algorithms before picking the most effective algorithm for a given task. Typically, algorithms tend to have several common components; thus, the parallelization framework should allow the programmer to reuse as much of their existing code as possible to facilitate rapid prototyping. For example, consider the implementation of a parallel ensemble method such as bagging [12] that uses the above mentioned decision tree induction algorithm for its base classifier. Bagging generates several random samples of the training data and builds a different classifier using each of these random samples to create an ensemble of classifiers. It then takes the majority class label from the labels predicted by each individual classifier on the test data to produce the final prediction. Ideally, one should be able to compose a parallel bagging predictor using a parallel random sampler and a pluggable parallel classifier.

Domain-specific parallelization: Most of the parallelization platforms that we have mentioned in this Section are generic; that is, they can efficiently parallelize applications from a wide variety of application domains. However, their genericity comes at a high price — these parallelization frameworks require that the parallelization be expressed at a very low-level. As the implementers of

ML-DM algorithms often do not have expertise in low-level parallel programming, it is essential that the parallelization framework employed to parallelize ML-DM algorithms be domain-specific. Furthermore, given the varying nature of ML-DM workloads, the framework should support efficient parallelization of both compute and I/O-intensive workloads.

We believe that no existing parallelization infrastructure, including MR, is capable of addressing all these requirements. As is, MR programming is very low level and the implementation process continues to be time consuming. Its one-input, two stage data flow is extremely rigid and does not directly match with the nature of ML-DM computations. Its deficiencies are as follows:

- Users need to write custom code to manage larger computations that require multiple MR jobs (e.g., iterative and recursive computations). Also, they are responsible for managing data across jobs and scheduling these jobs.
- When multiple computations can be performed inside a single MR job, users are responsible for co-scheduling and pipelining these computations inside a single job.

These factors lead to code that is difficult to maintain and reuse. Furthermore, the burden of optimizing these executions falls in the hands of the implementer. Ideally, the parallel programming tools should be palatable to the ML-DM community and the optimization strategies should be aligned with the nature of parallel ML-DM algorithms.

To address these challenges, we present an infrastructure named NIMBLE whose programming abstractions have been designed with the intention of parallelizing ML-DM computations. These programming abstractions allow users to specify data parallel, iterative, task parallel, and even pipelined computations. Furthermore, NIMBLE provides built-in support to process data stored in a variety of formats; it also allows facile implementation of custom data formats. The abstraction and optimization strategies have been designed hand-in-hand to deliver high performance on MR based runtimes — the infrastructure however is portable and can support other runtimes (e.g., MPI, X10). NIMBLE is fully implemented and is being used by programmers at IBM Corporation and will be available for public consumption in the immediate future.

The rest of this paper is organized as follows. Details of the NIMBLE interface are presented in section 2. The runtime and the execution engine are presented in section 3. Sample implementations and their performance evaluations are presented in section 4. Related work is presented in section 5 and conclusions in section 6

2. SOFTWARE ARCHITECTURE

The primary goal of NIMBLE is to enable rapid development of parallel ML-DM algorithms that run portably on distributed- and shared-memory machines. To realize this goal, every design decision in NIMBLE is geared towards both productivity and portability; e.g., NIMBLE is implemented in JAVA, a high productivity language. In this section, we describe the design of NIMBLE in detail.

Software architecture overview: NIMBLE is organized into three distinct layers (Figure 1): (1) The *User API layer*, which provides the programming interface to the users. It primarily consists of abstract classes that allow users to represent *tasks* and directed acyclic graphs (DAGs) of tasks to express dependencies between tasks. A task can take one or more datasets as input, may process the input in parallel, and produce one or more datasets as output. Additionally, NIMBLEś API also provides means to allow users to spawn other tasks/DAGs from inside other tasks and wait on their completion.

¹ There are a small class of embarrassingly parallel algorithms, which can be parallelized without much effort.

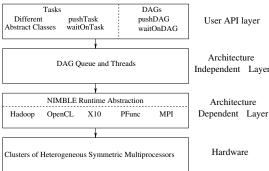


Figure 1: An overview of the software architecture of NIMBLE. At the very top is the user-level API, followed by the architecture independent DAG control layer. The last layer is the architecture dependent layer that allows portable execution of NIMBLE programs.

(2) The Architecture independent layer, which acts as the middle-ware between the user specified tasks/DAGs, and the underlying architecture dependent layer. The primary structures in this layer are a DAG queue along with worker threads that process this queue to ensure progress. This layer is primarily responsible for the smart scheduling of tasks and DAGs, and delivering the completion notifications of these tasks and DAGs to the users. (3) The Architecture dependent layer, which consists of harnesses that allow NIMBLE to run portably on various runtimes. Currently, NIMBLE only supports execution on the Hadoop platform.

<u>Software view of hardware:</u> NIMBLE presents a master/slave perspective of the hardware to the user-level tasks (Figure 2). In this model, the main thread of execution begins on the master node. When a task is spawned, depending on its requirements, it is either directly executed on the master node or the task and its data are split into smaller chunks and each chunk is executed on a different slave node (i.e., a different virtual address space). This allows NIMBLE to use a single physical machine as both the master and slave nodes.

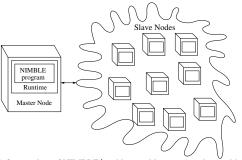


Figure 2: Software view of NIMBLEs architectural layout. A task can either execute on the master node, or be split and executed among slave nodes. Split tasks do not share any state as they execute in disjoint virtual address spaces.

User API layer: The primary mechanism for users to express an algorithm is by implementing one or more tasks. A task represents an independent, reusable piece of computation. In programming terms, a task is a JAVA class that is a subclass of one of the abstract task classes provided by NIMBLE. Several abstract task types are provided; each of which represents a different control/data flow (Figure 3); also each abstract class has some properties that dictate its co-scheduling eligibility. NIMBLEs tasking model is designed to let users write simple, reusable components that can be composed together to implement a wide variety of algorithms at a high-level of abstraction; thus NIMBLE effectively decouples the functionality provided by the task from its execution.



Figure 3: The task hierarchy in NIMBLE. Tasks that are dashed (AbstractTask and WrapperTask) are not directly extensible. The rest of the tasks can be extended by the users to create their own custom tasks.

All NIMBLE tasks derive from AbstractTask; however, this class is not directly extensible by the users. In NIMBLE tasks can be divided into one of two kinds: *Data tasks* that have at least one input dataset and can generate one or more output datasets, and *Non-Data tasks* that represent pure computations and have neither input nor output datasets.

AbstractOnePassTask, AbstractOnePassMergeTask,

AbstractOnePassKeyedTask, and AbstractIterativeTask are all examples of data tasks. Specifically, they can access their input datasets in parallel and write to one or more output datasets in parallel. There are two types of datasets, *regular datasets* and *keyed datasets*. Regular datasets consist of one or more records that are independent, and hence, can be processed one record at a time. Keyed datasets will be described shortly. All the data tasks have initializeTask () and finalizeTask () methods. The initializeTask () method is called at the beginning of a task's execution and is executed only once on the master node before commencing with the data parallel execution on the slave nodes. Similarly, the finalizeTask () method is invoked only once on the master node after a task has completed processing all its input data. We now describe the individual data tasks in more detail.

```
Input: t: task
    Output: taskSet: A set of tasks
   dataPartition = partition(getInputDatasets(t));
   taskSet = \emptyset;
 3
    /* Create as many tasks as data partitions */;
 4
    foreach partition \in dataPartition do
 5
         t' = makeReplica(t);
         {\tt setInputData}(t', partition);\\
 6
7
         taskSet = taskSet \bigcup t';
    /* Process each partition */
    foreach task \in taskSet do
10
         foreach dataset \in getInputDatasets(task) do
              foreach record \in \mathsf{getRecords}(dataset) do
11
12
                   t.processRecord(dataset, record):
13 return taskSet;
```

Algorithm 1: executeTask

AbstractOnePassTask is the simplest of the data tasks (see Algorithm 2) and it requires users to implement a processRecord() method. Once the initializeTask () method is invoked on the master node, conceptually, the data is broken into a number of partitions. We make a copy of the task for each of these partitions, and then process each partition with its associated copy of the task. The partition is processed by feeding the task one record at a time using the processRecord() method (see Algorithm 1). The task also has the ability to write to one or more output datasets inside the processRecord() method. Once all the partitions are processed, we invoke the finalizeTask () method on the original task. While this programming abstraction does not make parallelism explicit, it emulates a doAcross() loop over the entire data, where each partition is

processed in parallel. An example of an operation that maps well to AbstractOnePassTask is matrix scaling, which can be carried out independently on each element.

```
Input: t: task
1 t.initializeTask():
\mathbf{2} executeTask(t);
3 t.finalizeTask():
                            Algorithm 2: executeOnePassTask
```

AbstractOnePassMergeTask extends AbstractOnePassTask with an additional mergeTasks() method (see Algorithm 4); this method can be used to merge/aggregate the state of all the tasks that processed independent partitions of data during processRecord(). The mergeTasks() method is akin to a *reduction* operation, hence can be invoked on two or more instances of a task in parallel; it is an ideal candidate when partial results computed during processRecord() need to be accumulated. For example, consider computing the mean of all the elements of a sparse matrix, which would involve accumulating both the number of non-zeros and the sum of all values in each partition during processRecord(). The mergeTasks() method can then be used to merge all the partial results and produce the final result.

```
Input: taskSet: A set of tasks
    Output: t: A merged task
1 t = pop(taskSet);
\mathbf{2} \ \text{ for each } t' \in taskSet \ \text{do}
\textbf{3} \qquad \qquad t.\mathsf{mergeTasks}(t');
4 return t:
                                  Algorithm 3: mergeTasks
   Input: t: task
1 t.initializeTask();
2 taskSet = executeTask(t):
   t = mergeTasks(taskSet):
4 t.finalizeTask():
                          Algorithm 4: executeOnePassMergeTask
```

AbstractIterativeTask extends AbstractOnePassMergeTask by adding iterative control flow using beginlteration () and endlteration () methods (see Algorithm 5). Iterations continue till the beginlteration () method returns false. The endlteration () method is invoked at the end of every iteration to perform computations to setup the following iterations. AbstractIterativeTask task should be used for the specification of iterative, data parallel computations. Consider the implementation of k-Means clustering in NIMBLE — beginlteration () returns true till convergence is reached. processRecord() is used to assign data points to one of k centers and maintain sufficient statistics, mergeTasks() is used to accumulate sufficient statistics across partitions, and endlteration () is used to compute the new k centers (see Section on k-Means clustering).

```
Input: t: task
1 t.initializeTask();
2 while t.beginIteration() do
3
        taskSet = executeTask(t);
        t = mergeTasks(taskSet);
        t.endIteration();
6 t.finalizeTask();
                          Algorithm 5: executeIterativeTask
```

AbstractOnePassKeyedTask is used to process keyed datasets, where each record is associated with a key. AbstractOnePassKeyedTasks require that their input records be sorted on each key across all their input datasets. The task is then provided with all records across all datasets that match a certain key through the reduceRecord()

method. When all the records with the same key have been reduced, NIMBLE invokes finishProcessingForCurrentKey() to help which allows key-wide reduction. This task should be used for the specification of data parallel computations where aligned access to one more inputs is needed. For example, consider the addition of two matrices — we need to ensure that we add elements with the same row and column indices; this requires a sort operation.

Non-data tasks derive from WrapperTask. As these are designed to represent computations without data access, depending on where the task is to be executed, one can either use WrapperTaskMaster tasks (for execution on the master node) or WrapperTaskWorker tasks (for execution on a slave node). The WrapperTaskMaster can be used to implement work flows consisting of other serial and parallel tasks. The WrapperTaskWorker can be used to embed existing binaries that can be executed on the slaves node. It allows one to leverage task parallelism using existing binaries.

```
Input: t: task
 1 t.initializeTask();
2 keyedDatasets = getInputDatasets(t):
   /* Sort all datasets on key */:
    sortedKevedDatasets = sortOnKev(kevedDatasets):
   foreach key \in sortedKeyedDatasets do
         foreach dataset \in getInputDatasets(t) do
              foreach record \in getRecords(key, dataset) do
8
                 \begin{tabular}{ll} t. {\tt reduceRecord}(key, dataset, record); \\ \end{tabular}
              t.finishProcessingForCurrentKey();
10 t.finalizeTask();
```

Algorithm 6: executeOnePassKevedTask

Task Spawning: In NIMBLE, tasks can spawn and wait on completion of their spawned tasks. The outputs of any completed task can be piped in as input to the spawned task as well. As has been noted, a task represents data parallel computations; with the ability for tasks to spawn other tasks, NIMBLE also supports task parallelism. In fact, NIMBLEs API elegantly supports algorithms that require both task and data parallelism concurrently. An example where task spawning is beneficial is the implementation of divide-and-conquer algorithms, such as decision tree induction algorithms. Task spawning also facilitates the writing of reusable parallel codes - each task should be used to represent an independent piece of computation and tasks can spawn other tasks akin to function calls in programming languages to realize reusable parallel codes.

DAGs of tasks: NIMBLE supports the chaining of tasks to create task DAGs and ensures their efficient execution; this is an important feature of NIMBLE that promotes modular design of reusable tasks without sacrificing performance. Unlike task spawning, where the dependency between tasks is available only when a task is executed, by task chaining, the dependencies between tasks are made explicit a priori. The benefit of making these dependencies available early is that NIMBLEś runtime system can co-schedule many of these chained tasks inside a single MR job, thereby minimizing the number of I/O scans and the overheads of starting MR jobs. An example where specifying DAGs of tasks is beneficial is the implementation of matrix computations, which are common when implementing ML-DM algorithms.

One issue that arises when dealing with DAGs of tasks is that of data alignment. As a rule, an AbstractOnePassKeyedTask requires its input to be sorted on its input datasets' keys. However, once inputs have been sorted, a parent task of type AbstractOnePassKeyedTask does not need to have its output re-sorted before feeding its output to its children provided that it has not altered the alignment of the keys. Thus, each edge in a DAG has a property that specifies whether the input to a task needs to be sorted or can be used as

is. The benefit of this property is that it allows one to co-schedule AbstractOnePassKeyedTasks in a situation where one provides the input to the other without having to sort on the keys.

In summary, the NIMBLEś API layer exposes an abstraction that allows programmers to express different forms of parallelism without requiring explicit data management, communication, or scheduling. This allows users to write near serial code, while reaping the benefits of parallel execution. Furthermore, NIMBLE encourages composition of complex algorithms using smaller parallel building blocks. Algorithms that have been written against this API are then presented to the lower layers that orchestrate execution on Hadoop while managing control and data.

<u>Datasets:</u> An important aspect of using NIMBLE involves moving data in and out of tasks. In NIMBLE, all data are treated as a stream of records (essentially black boxes); the only requirement being that records must be convertible to and from strings (TextRecords) or bytes (BinaryRecords). To facilitate users, NIMBLE provides several adaptors that allow data stored in many popular formats to be used without any modifications. For example, NIMBLE has built-in support for LibSVM [13] and several other sparse matrix formats such as (i, j, v), and compressed sparse row/column (CSR/CSC) formats. Furthermore, it is trivial to write adaptors that allow use of data stored in any format (text or binary) with NIMBLE.

3. RUNTIME

One of the primary goals of NIMBLE is to enable the user's algorithms to run on a wide-variety of parallel platforms in a performance-portable fashion. In order to realize this goal, NIMBLEs runtime is split into two parts; the architecture independent layer, and the architecture dependent layer. Figure 1 depicts the high-level structure of the runtime composition of NIMBLE; the user's applications always interact with the architecture independent layer, ensuring complete portability. In this section, we briefly describe the architecture independent layer. In addition, we also describe the Hadoop architecture dependent layer, which allows NIMBLE applications to run on the Hadoop MR framework.

Architecture independent layer: This light-weight layer sits between the user programs and runtime platforms such as Hadoop, MPI, and X10. It consists of a DAG queue and a thread pool; threads wait for users to push DAGs or individual tasks into the DAG queue. As there is a single DAG queue that services all the threads in the thread pool, the architecture independent layer of NIMBLE can be thought of as a work-sharing environment. Once a thread pulls a DAG from the DAG queue, it executes it to completion using the pre-specified architecture dependent layer. ² Algorithm 7 shows the naive version of the loop run by each architecture independent thread. Briefly, each thread gets a runtime (e.g., the Hadoop architecture dependent layer) and the DAG queue as input; then, it continually removes DAGs from this queue and executes them. DAGs are executed by repeatedly extracting the maximal sub DAG that can be executed by the given runtime as one job and executing it (see Section on Co-scheduling of tasks).

The Hadoop architecture dependent layer: The primary goal of this layer is to provide the services needed by the architecture independent layer; i.e., allowing smooth execution of NIMBLE programs on the underlying framework. To enable running NIMBLE applications for a specific runtime, we need to implement an interface for that runtime, which can take NIMBLE DAG specifications and execute these DAGs as efficiently as possible. We selected Hadoop as the target runtime for our first architecture dependent

```
 \begin{array}{c|c} \text{Input: } Q\text{: queue of DAGs, } R\text{: Architecture-specific runtime} \\ \textbf{1} & \text{while not shutdown() do} \\ \textbf{2} & \text{repeat} \\ \textbf{3} & d = \text{getNextDAG}(Q); \\ \textbf{4} & \text{until not } d = NULL \;; \\ \textbf{5} & \text{while executionIncomplete}(d) \; \text{do} \\ \textbf{6} & s = \text{getExecutableSubDAG}(R, d); \\ \textbf{7} & s = \text{executeDAG}(R, s); \\ \textbf{8} & \text{reintegrateDAG}(d, s); \\ \end{array}
```

Algorithm 7: runArchitectureIndependentThread

layer because of its inherent support for handling large datasets, which are an important consideration for data mining and machine learning algorithms. Briefly, each architecture dependent layer has to expose the following functionality: (1) co-scheduling individual tasks found in the DAG using one or more jobs, (2) executing the job, which includes piping intermediate outputs between tasks, and (3) returning the executed DAG (with changed internal state) back to the architecture independent layer.

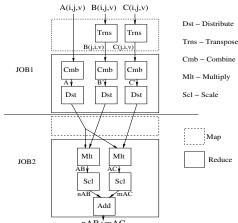


Figure 4: Co-scheduling for a naive implementation of the matrix equation $n \times A \times B + m \times A \times C$ such that the entire operation is completed in two MR jobs.

Co-scheduling of tasks: An important advantage of NIMBLE is its ability to design an entire application as a series of encapsulated, reusable tasks. However, executing each individual task as a separate "job" can result in excessive startup costs and lost opportunities for data reuse between tasks. To alleviate such performance bottlenecks, NIMBLE allows users to spawn and execute entire DAG computations instead of just individual tasks. Apart from being an elegant and powerful programming abstraction, the ability to spawn and execute DAG computations also allows NIM-BLE to co-schedule appropriate tasks within a DAG in a single MR operation, thereby overcoming the high overhead involved in executing Hadoop jobs. However, there is not a one to one correlation between the user-level DAGs and those that can be executed by Hadoop in one MR job. In this section, we describe the algorithms used to select the maximal sub-DAG from the user-level DAG that can be executed as one MR job.

Consider the execution of the computation $X=(n\times A\times B+m\times A\times C)$, where A,B, and C are sparse square matrices represented in the (i,j,v) format, and n and m are scalars. One way to compute this equation using MR is to first convert the matrices from their (i,j,v) representation into individual compressed sparse rows (CSR); of course, matrices B and C, are first transposed into (j,i,v) before converting them into their CSR format. Once this step is complete, we have A,B^T , and C^T represented as CSR vectors; now, the given equation can be computed quite easily. The DAG of the tasks that are involved in computing X is shown in

²Individual tasks are internally wrapped around in a DAG for uniformity.

```
Innut: DAG D
   Output: List of executable tasks.
   execTaskSet = \emptyset;
2
3
        noNewAdditions = true:
        foreach task \ t \in unchosenTaskList(D) do
4
5
            removeDependencies(t.\ executableTaskSet):
6
            if noUnsatisfiedDependency(t) then
                 execTaskSet = execTaskSet \bigcup t;
8
                 if instanceOf(OnePassKeyedTask,t) or
                 (instanceOf(OnePassTask,t) and lastParentPhase(t) ==
                  reduce) then
                  setExecutionPhase(t, reduce);
9
                 else
10
11
                     setExecutionPhase(t, map);
12
                 markTaskAsChosen(t):
                 noNewAdditions = false;
13
14 until noNewAdditions:
15 return execTaskSet;
                       Algorithm 8: getExecutableSubDAG
```

Figure 4. The matrix transpose is calculated by using two separate task types; the Transpose task is of type AbstractOnePassTask; it takes (i, j, v) entries as input and generates (j, i, v) entries that are keyed on j, the original column numbers. The Combine task is of type AbstractOnePassKeyedTask that takes these (j, i, v) matrix elements and recombines them to form CSR column-vectors. The next task is to distribute the matrices A, B^T , and C^T so that $A \times B$ and $A \times C$ can be computed using inner products. This is done using the Distribute task of type AbstractOnePassTask which takes in a row-vector and generates an output dataset, which is keyed based on the row-number. In addition, the Distribute task also replicates the right hand side (matrices B^T and C^T) so that each row in B^T and C^T (i.e., each column of B and C) is sent to each row of A. The next tasks are self-explanatory; the two Multiply tasks of type AbstractOnePassKeyedTask compute $A \times B$ and $A \times C$ one row at a time respectively. Similarly, the two Scale tasks of type AbstractOnePassTask compute $n \times A \times B$ and $m \times A \times C$ respectively, and finally the Add task of type AbstractOnePassKeyedTask adds the two results from the Scale tasks together to compute X.

If each task in the above DAG were executed as an individual MR job, it would take 13 MR jobs to compute X; since many elements are needlessly read from and written to the disk multiple times, a large overhead is incurred. For example, A is common in computing $A \times B$ and $A \times C$; the result of these computations can be directly streamed to be scaled and added up resulting in computation of X without ever having to write intermediate results to disk. This intuitive execution is precisely what NIMBLE achieves, thereby computing X in just two MR jobs (see Figure 4). This has two benefits: (a) we only pay Hadoop's overhead cost once and (b) we directly pipe data from one task to another *in memory* as long as both tasks are executing in the same phase (i.e., Map or Reduce).

Algorithms 8 and 9 elaborate on the exact procedure used to pick the maximal sub-DAG in a DAG that can be executed in one MR job. From the time a DAG is spawned for execution until it has completed its execution, NIMBLE's runtime engine repeatedly queries getExecutableSubDAG() to get the maximal executable sub-DAG of the spawned DAG. When invoked, getExecutableSubDAG() returns a list of tasks that can be run in the Map phase and a list of tasks that can be run in the Reduce phase of the MR job. This is achieved by first adding all those tasks in the DAG that have no dependencies (as checked by noUnsatisfiedDependency()) to the appropriate list (all but tasks of type AbstractOnePassKeyedTask are marked as Map phase tasks). Once these initial tasks are added to the Map and the Reduce task lists, we iterate over the remaining

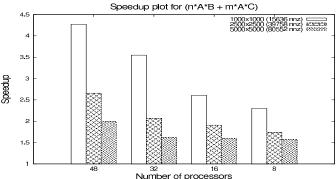


Figure 5: Co-scheduling speedups achieved when computing $(n \times A \times B + n \times A \times C)$, where A, B, and C are sparse matrices. Matrix multiplication was carried out (naively) using the inner-product formulation. Number of non-zeros in each matrix are given in brackets. The experiments were run on IBM's Nadal cluster (see Section 4).

tasks until a fixed point is reached (i.e., no new tasks are added). Additional tasks are added by judiciously removing dependencies that are satisfied by the execution of the tasks that are already in the Map and the Reduce task lists. This procedure is shown in removeDependencies() and the logic is as follows. If a child task t_c depends on the completion of a parent task t_p , then we mark such an edge as satisfied iff t_c can be executed after t_p in the same MR job. As can be seen in Algorithm 9, marking an edge (t_p, t_c) as satisfied depends on the type of both t_p and t_c . When t_p is a AbstractIterativeTask, we cannot mark any of its outgoing edges as satisfied because the termination criteria is determined only at runtime. When t_p is a AbstractOnePassTask, all its outgoing edges can be marked as satisfied because t_p can be completely executed in the Map or Reduce phase. When t_p is a AbstractOnePassMergeTask, an edge to t_c , which is a AbstractOnePassKeyedTask, can be marked as satisfied because t_p 's execution can be completed during the Reduce phase, and t_c can be completely executed after t_p in completed in the same Reduce phase. Finally, when both t_p and t_c are AbstractOnePassKeyedTasks, an edge from t_p to t_c can be marked as satisfied iff t_p 's output is aligned with t_c 's input — in other words, if the output of t_p does not have to be sorted again. Once the maximal sub-DAG is found, the tasks in this sub-DAG are executed in topological order. Note that tasks of type WrapperTask do not appear in Algorithms 8 and 9 because they have no input or output. Hence, they are perceived as singleton nodes in the DAG.

Figure 5 presents the speedups achieved by co-scheduling tasks when executing the task DAG shown in Figure 4. By co-scheduling tasks, we consistently get speedups across all matrix sizes. For smaller matrices, the speedups are significant as most of the execution time can be attributed to the I/O overhead involved in MR. As the number of computations required increase, the speedups are lower as the execution time is dominated by the $O(n^3)$ operations required for matrix multiplication. The importance of coscheduling tasks is more pronounced when the executions are I/O bound — which is true for a large class of ML-DM algorithms.

Executing a DAG: After selecting the maximally executable sub-DAG, the Hadoop architecture dependent layer first determines all the inputs that are needed for the current sub-DAG to execute, and creates output file paths for all the leaf tasks' outputs. Once this is done, the job is submitted to Hadoop with our custom Map and Reduce classes; these classes know how to read the (key, value) pairs that are inputs to both the Map and the Reduce phases, interpret these pairs as records and marshal them to the tasks that need these inputs. Since we execute more than one task in both the Map and the Reduce phases, the NIMBLE Hadoop layer also redirects the intermediate output from the non-leaf tasks to other tasks.

```
Input: t_c: task, T: set of tasks
   foreach task t_p \in T do
        if isParent(\hat{t}_p,t) and not instanceOf(IterativeTask,t) then
2
             if instanceOf(OnePassMergeTask,\ t_p) then
3
                  if instanceOf( OnePassKeyedTask,t_c) \ {\bf or} \\
4
                  \dot{(}instanceOf(\dot{O}nePassTask,\!t_{c}) and not
                  instanceOf(OnePassMergeTask,\ t_c)) then
5
                       markEdgeAsSatisfied(t_p,t_c);
             else
                  if instanceOf(OnePassTask,t_p) then
                       if getExecutionPhase(t_p) = reduce then
 8
                            if (instanceOf(OnePassKeyedTask,\ t_c) and
 9
                            alignedWithParents(t_c)) or
                            instanceOf(OnePassTask, t_c) then
10
                             markEdgeAsSatisfied(t_p, t_c);
12
                            markEdgeAsSatisfied(t_p, t_c);
13
                       if instanceOf(OnePassKeyedTask, t_p) then
15
                            if instanceOf(OnePassKeyedTask,t_c) &&
                            alignedWithParents(t_c) then
16
                                 markEdgeAsSatisfied(t_p, t_c);
17
                            else
                                 if (instanceOf(OnePassTask,t_c) and not
18
                                 instanceOf(OnePassMergeTask,\ t_c)) then
                                      markEdgeAsSatisfied(t_p, t_c);
19
```

Algorithm 9: removeDependencies

Returning the executed task: In NIMBLE users express computation as tasks, which are JAVA objects. As these tasks are executed in Hadoop by replicating the objects, the state in each of these tasks may need to be merged and returned to the users. This is done in the following manner. At the end of the Map phase, the tasks that were used in that Mapper are emitted as values with a special set of keys (taskKeys). These are directed to a set of reducers who proceed to merge these tasks together and write the serialized, partially merged tasks to the file system. These partially merged tasks are then recreated in memory on the master node by de-serializing the tasks and merging them to finally create a single task that is returned to the user.

4. ALGORITHMS AND EVALUATIONS

To demonstrate the expressiveness of NIMBLE, we detail the implementation of two popular data mining algorithms — k-Means Clustering and Pattern Growth-based Frequent Itemset Mining. We also evaluate the performance of these algorithms together with three other data mining algorithms – k-Nearest Neighbors, Random Decision Trees, and RBRP-based Outlier Detection (implementation details omitted due to space constraints) – to illustrate gains in productivity and performance. Experiments were performed on IBM's Nadal cluster, which has 48 processors distributed across 12 nodes with 4 GB of RAM per processor. All the nodes were running RHEL 5 (kernel 2.6.18-164.el5) with IBM Java v1.6.0 and Hadoop 0.20.2.

<u>k-Means Clustering:</u> Given a dataset D consisting of n data points, each with d dimensions, the data clustering problem is to partition this dataset into k subsets such that each subset behaves "well" under some measure. The popular k-Means clustering algorithm can be briefly described as follows. First, it begins with k random centers, $C^0 = \{C_1^0, \cdots, C_k^0\}$. Next, for each of the n data points, it finds its closest center in C^0 and maintains the sum of all points and number of points that were assigned to each center. The data points are partitioned into k subsets based on their closest centers.

The center of mass for each of these k subsets is used to find the new set of k centers, $C^1 = \{C_1^1, \cdots, C_k^1\}$. This process continues iteratively until we encounter an iteration i such that the centers C^i and $C^{(i+1)}$ are identical.

It is well known that data parallelization is a means of parallelizing k-Means clustering. A parallel implementation of k-Means clustering in NIMBLE is provided in Figure 6. k-Means clustering is implemented using just one class that extends AbstractIterativeTask. The initializeTask () method is used to initialize the centers randomly. The beginlteration () method returns true provided the process has not converged. The processRecord() method is responsible for assigning each data point to its closest center and maintaining sufficient statistics that will be used to determine the new centers. The mergeTasks() method sums up the sufficient statistics. The endIteration () method computes new centers based on the accumulated sufficient statistics and also checks for convergence. The finalizeTask () method is used to output centers after convergence. Looking at this example, it is evident that NIMBLE provides a natural and near-serial way of implementing algorithms. The only additional method needed for parallelization is the mergeTasks() method that is straightforward to implement in this case. Data management, communication, and control are absent. The implementation is far more succinct relative to the corresponding MR implementation that would span several classes and require explicit data management (to provide input and read output of a MR job) and job submission for each iteration of k-Means clustering.

```
class kMeansTask extends AbstractIterativeTask {
 float [][] kCenters = new float [k][dimensions];
  float [][] newkCenters = new float [k][dimensions];
  int [] kPoints = new int [k];
  float [][] kSumPoints = new float [k][dimensions];
  boolean converged = false:
  public void initializeTask() {initializeCenters(kCenters);}
  public boolean beginIteration() {return converged;}
  public void processRecord(BasicRecordl record) {
    final int closestCenter = findClosestCenter(kCenters,record);
    ++(kPoints[closestCenter]):
     \textbf{for (int } i = 0; i < \texttt{record.getNumFields()}; i++) \ \texttt{kSumPoints[closestCenter][i]} += \texttt{record[i]}; \\ 
  public void mergeTasks(AbstractBasicTask mergeTask) {
    for (int i = 0; i < kCenters.length; i++) {
      for (int j = 0; j < kCenters[i].length; <math>j++)
        kSumPoints[i][j] += ((KMeansTask)mergeTask).kSumPoints[i][j];
      kPoints[i] += ((KMeansTask)mergeTask).kPoints[i];
  public void endIteration() {
    for (int i = 0; i < newkCenters.length; i++)
      for (int j = 0; j < newkCenters[i].length; <math>j++)
        newkCenters[i][j] = kSumPoints[i][j] / kPoints[i];
    converged = (difference(newkCenters, kCenters) < THRESHOLD) ? true: false;
    kCenters = newkCenters:
  public void finalizeTask() { outputCenters(kCenters); }
```

Figure 6: k-Means Clustering using NIMBLE

Frequent Pattern Mining: Frequent pattern mining, also known as frequent itemset mining (FIMI), plays an important role in a range of data mining tasks. The frequent pattern mining problem was formulated by Agrawal *et al.* [11] for association rule mining. Briefly, the problem description is as follows: Let $I = \{i_1, i_2, \cdots, i_n\}$ be a set of n items, and let $D = \{T_1, T_2, \cdots, T_m\}$ be a set of m transactions, where each transaction T_i is a subset of I. An itemset $i \subseteq I$ of size k is known as a k-itemset. The support of i is $\sum_{j=1}^m (1:i \subseteq T_j)$, or informally speaking, the number of transactions in D that have i as a subset. The frequent pattern min-

ing problem is to find all $i \in D$ that have support greater than a minimum support value.

Our parallel FIMI implementation in NIMBLE uses a pattern growth approach that is similar to the one proposed by Li et al. [21] and is provided in Figure 7. The approach extends all the given prefixes in the input dataset by 1 and prunes away infrequent extensions. It then groups these extended prefixes and builds a projected dataset for each group. Finally, it recursively invokes the same task on these projected datasets until no more extensions are possible. The recursive implementation of FIMI in NIMBLE needs a single class called fimiTask that extends AbstractIterativeTask. The task performs two iterations. In the first iteration, the processRecord() method maintains counts for items as extensions of all itemsets in prefixes. Then the mergeTasks() method is used to aggregate counts across tasks. Finally, the endlteration() method is used to prune infrequent extensions, group frequent extensions into itemsetGroups, and allocate datasets for each group in itemsetGroups. In the second iteration, the processRecord() method projects the input dataset for each group in itemsetGroups. Finally, the endIteration() method spawns a child task for each group in itemsetGroups, provided it can be extended.

Looking at this example, as was the case with k-Means, the implementation is natural, succinct, and near-serial – the only additional method needed for parallelization is the mergeTasks() method that is straightforward to implement in this case. Furthermore, support for recursion does indeed simplify implementation.

```
class fimiTask extends AbstractIterativeTask {
  int iteration = 0:
  HashMap<Vector<Long>,HashMap<Long,Long>> itemCounts=
       new HashMap<Vector<Long>,HashMap<Long,Long>>();
  Vector<Vector<Long>>> itemsetGroups=new Vector<Vector<Vector<Long>>>();
  //Projected datasets for an itemset group
  HashMap<Integer,OutputDataset> itemsetProjectedDatasets =
        new HashMap<Long, OutputDataset>();
  Vector<Vector<Long>> prefixes;
  public boolean begin|teration() {return (iteration == 0 || iteration == 1) ? true; false;}
  public void processRecord(BasicRecordI record) {
   if(iteration == 0) countItems(record.itemCounts):
    else if(iteration == 1) {
      for (int i=0; i < itemsetGroups.size(); i++)</pre>
       project(itemsetGroups.get(i),record,itemsetProjectedDatasets.get(i));
 public void mergeTasks(AbstractBasicTask mergeTask) {
    \textbf{if} (iteration == 0) \ mergeItemCounts (itemCounts, ((fimiTask)mergeTask).itemCounts); \\
  public void endIteration() {
    if(iteration == 0){
      pruneInfrequentItems(itemCounts);
      groupItems(itemsetGroups, itemCounts);
      \textbf{for (int } i=0; \ i{<} itemsetGroups.size(); i{+}{+}) \{
        SparseDataset od = new SparseDataset();
        itemsetProjectedDatasets.put(i,od);
   } else if (iteration == 1) {
      for (int i= 0; i<itemsetGroups.size();i++){
        fimiTask childTask = new fimiTask();
        childTask.prefixes = new Vector < Vector < Long >> ();
       childTask.prefixes.addAll(itemsetGroups.get(i));
        childTask.addInputDataset(itemsetProjectedDataset.get(i));
        if (childTask.prefixes.size > 0 && itemsetProjectedDataset.get(i).hasRecords())
          this.getTaskQueue().pushTask(childTask);
    ++iteration;
```

Figure 7: Frequent Pattern Mining using NIMBLE

k-Nearest Neighbors: Given a dataset, the k-Nearest Neighbors (k- $\overline{\text{NN}}$) algorithm finds the k nearest neighbors for each point in the

dataset Nearest neighbors can be computed using any distance metric; for our implementation we used euclidean distances. k-NN is often used in machine learning as a simple classification mechanism where each unlabeled item is labeled based on a majority vote of the labels of its nearest neighbors. The process of calculating k-NN for each data point consists of computing the distance from that point to every other point and maintaining a MaxHeap that keeps track of only the k nearest neighbors of the point. The procedure followed to compute k-NN is an iterative one and is implemented as an AbstractlterativeTask. During each iteration, we perform distance computations to find the k nearest neighbors for a subset of the data points by streaming through and measuring distances against all other data points. This process continues iteratively until we have determined the k nearest neighbors of all input points.

Random Decision Trees: We implemented a classifier based on Random Decision Trees [15] in NIMBLE. The algorithm constructs multiple decision trees randomly. When constructing each tree, the algorithm picks a feature randomly when expanding a node without any purity check (such as information gain). When picking a categorical feature, it ensures that the feature-value combination has not been chosen previously on the path from that node to the root node. Continuous features on the other hand can always be chosen and are set with a random threshold. We stop expanding a node when the number of instances that reach that node are below a threshold. The approach is implemented using an AbstractIterativeTask - each task builds one random decision tree. The task initializes a subset of the nodes of the tree randomly based on the range of values of the features and then gathers the number of instances in the training set that reach each node in the tree. Nodes that are not reached by a sufficient number of instances are pruned away. Nodes that can be further expanded are grown with a new random subtree and this process continues iteratively until the process terminates.

Outlier Detection: We implemented the RBRP algorithm [16] for distance-based outlier detection in NIMBLE. The algorithm operates in two phases. The first phase partitions the dataset into bins such that points that are closer to each other are more likely to be assigned to the same bin. Every point is assigned to exactly one bin and each bin must be less than a certain size. The second phase finds outliers in each bin separately and then merges these outliers into a global set of outliers. Both these phases have been parallelized using an AbstractIterativeTask.

Performance Evaluation: The results of our performance evaluation for all five algorithms are provided in Figure 8. The speedup numbers are relative to the execution time measured when using 8 processors. Thus, the maximum possible speedup when using 48 processors is 6. k-Means clustering was evaluated using large synthetic datasets – 10D-100M implies a 10 dimensional dataset with 100 million points. FIMI was evaluated using a large synthetic dataset (T10I4D100M) with 100 million transactions and a large real dataset (Webdocs) with 1.7 million transactions. k-NN was evaluated using the MNIST [20] datasets that have dimensionality 784 and the number points were varied from 20K to 60K. Random decision trees and RBRP were evaluated using large synthetic datasets - the naming convention for their datasets is same as that used for the k-Means datasets. It is easy to see that the NIMBLE implementations are indeed efficient and speedup improves with increasing dataset size and dimensionality.

5. RELATED WORK

Although there has been extensive research into *ad-hoc* parallelization of ML-DM algorithms for over two decades, we focus our attention on efforts that have looked at building a generic framework for parallelizing ML-DM algorithms. Jin and Agrawal

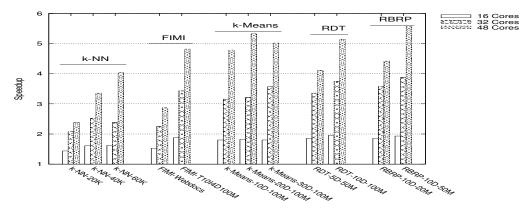


Figure 8: Speedup for algorithms implemented in NIMBLE. Speedup is relative to execution time on 8-cores.

[18] first studied the problem of building a common framework for implementing parallel data mining algorithms. They showed that these algorithms share a common structure that is amenable to MR-type parallelization. Based on this structure, the authors presented an interface for data parallelization of these algorithms that requires one to make simple modifications to sequential code to realize a parallelization. However, in addition to data parallelism, NIMBLE provides task parallelism, task DAGs, and efficient execution of these DAGs, which results in greater I/O efficiency. IBM's Parallel Machine Learning Toolbox (PML) [4] incorporates an infrastructure for implementing and running ML-DM algorithms on large parallel systems by leveraging aggregate memory for data storage and fast collective communication to perform reductions. However, unlike NIMBLE, PML does not support task parallelism.

Recently, the MR abstraction has been used to implement parallel machine learning algorithms. Chu *et al.* [14] have used MR to express a class of machine learning algorithms that fit the Statistical-Query model. In another study, Panda *et al.* presented a framework for the implementation of decision tree ensembles [23] on top of MR. We have already discussed the deficiencies of using MR for implementing parallel ML-DM algorithms.

Implementations such as Dryad [17] and Dryad-LINQ [24] support DAG-based abstractions for implementing ML-DM algorithms. While the DAG-based abstraction permits rich computational dependencies, it does not naturally express iterative, data parallel, task parallel, and dynamic data driven algorithms that are prevalent in ML-DM. Furthermore, Dryad's DAG abstraction is far too low level and we can in fact use it as an alternative runtime to Hadoop in NIMBLE. Apache's Mahout project [7] is closely related to NIMBLE in that it provides a library of machine learning implementations; however, it does not provide an infrastructure for the specification of parallel ML-DM algorithms.

6. CONCLUSION

In this paper, we presented NIMBLE, a novel, domain-specific parallelization framework that allows rapid development of parallel ML-DM algorithms. In particular, NIMBLE has been designed to scale to large datasets and processor counts as its runtime targets (e.g., Hadoop) are themselves scalable. NIMBLE cleanly separates process choreography and control from actual analytics. This al-

lows algorithm designers to focus more on algorithmic issues and worry less about parallelization issues. Furthermore, we have designed a DAG execution engine that optimizes execution of DAGs of tasks by packing as many tasks as possible into one runtime job. Results on several algorithms demonstrate both the efficiency and the ease of use of NIMBLE.

7. REFERENCES

- [1] Hadoop. http://hadoop.apache.org.
- [2] HBase. http://hadoop.apache.org/hbase.
- [3] Hive. http://hadoop.apache.org/hive.
- [4] IBM Parallel Machine Learning Toolbox. http://www.alphaworks.ibm.com/tech/pml.
- [5] Intel Threading Building Blocks. http://www.threadingbuildingblocks.org.
- [6] JAQL. http://www.jagl.org.
- [7] Mahout. http://lucene.apache.org/mahout/.
- [8] $MPI.\ http://www.mpi-forum.org.$
- [9] OpenMP.http://www.openmp.org.
- $\begin{tabular}{ll} [10] \begin{tabular}{ll} PThreads. \begin{tabular}{ll} https://computing.llnl.gov/tutorials/pthreads. \end{tabular} \label{tabular}$
- [11] R. Agrawal et al. Mining association rules between sets of items in large databases. ACM SIGMOD, 22(2), 1993.
- [12] L. Breiman. Bagging predictors. Machine Learning, 24(2), 1996.
- [13] C.-C. Chang and C.-J. Lin. LIBSVM: a library for support vector machines, 2001.
- [14] C. Chu et al. Map-reduce for machine learning on multicore. In NIPS, 2007.
- [15] W. Fan et al. A general framework for accurate and fast regression by data summarization in random decision trees. In ACM SIGKDD, 2006.
- [16] A. Ghoting et al. Fast mining of distance-based outliers in high-dimensional datasets. DMKD, 16(3), 2008.
- [17] M. Isard et al. Dryad: distributed data-parallel programs from sequential building blocks. In SIGOPS Operating System Review, 2007.
- [18] R. Jin and G. Agrawal. Shared Memory Parallelization of Data Mining Algorithms: Techniques, Programming Interface, and Performance. In SDM, 2002.
- [19] P. Kambadur et al. PFunc: Modern Task Parallelism For Modern High Performance Computing. In SC, 2009.
- [20] Y. LeCun et al. Gradient-based learning applied to document recognition. In Intelligent Signal Processing, 2001.
- [21] H. Li et al. Pfp: parallel fp-growth for query recommendation. In ACM RecSys, 2008.
- [22] C. Olston et al. Pig latin: a not-so-foreign language for data processing. In ACM SIGMOD, 2008.
- [23] B. Panda et al. PLANET: massively parallel learning of tree ensembles with MapReduce. Proceedings of the VLDB Endowment, 2(2), 2009.
- [24] Y. Yu et al. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In OSDI, 2008.