A Super-Programming Approach For Mining Association Rules in Parallel on PC Clusters[*]

Dejiang Jin and Sotirios G. Ziavras
Department of Electrical and Computer Engineering
New Jersey Institute of Technology
Newark, NJ 07102, USA

Abstract: PC clusters have become popular in parallel processing. They do not involve specialized inter-processor networks, so the latency of data communications is rather long. The programming models for PC clusters are often different than those for parallel machines or supercomputers containing sophisticated inter-processor communication networks. For PC clusters, load balancing among the nodes becomes a more critical issue in attempts to yield high performance. We introduce a new model for program development on PC clusters, namely the *Super-Programming Model (SPM)*. The workload is modeled as a collection of *Super-Instructions (SIs)*. We propose that a set of SIs be designed for each application domain. They should constitute an orthogonal set of frequently used high-level operations in the corresponding application domain. Each SI should normally be implemented as a high-level language routine that can execute on any PC. Application programs are modeled as *Super-Programs (SPs)*, which are coded using SIs. SIs are dynamically assigned to available PCs at run time. Because of the known granularity of SIs, an upper bound on their execution time can be estimated at static time. Therefore, dynamic load balancing becomes an easier task. Our motivation is to support dynamic load balancing and code porting, especially for applications with diverse sets of inputs such as data mining. We apply here SPM to the implementation of an apriori-like algorithm for mining association rules. Our experiments show that the average idle time per node is kept very low.

Keywords: mining association rules, cluster computing, load balancing, parallel processing.

[*] This work was supported in part by the U.S. Department of Energy under grant DE-FG02-03CH11171.

1. Introduction

Targeting cost-effective parallel computing [1-4], researchers have recently shifted their attention to the adoption of clusters containing commercial off-the-shelf (COTS) PC nodes. These systems are suitable for large-scale problems, such as data mining of very large databases [1]. PC clusters are decentralized systems. Each node is an independent PC running its own general-purpose operating system. A general-purpose inter-node network, such as the Ethernet, connects these nodes together. Data communication among the PCs is controlled by application layer software rather than by lower-level system software or hardware. Thus, the latency of data communications is usually longer than that of parallel computers and supercomputers that contain specialized hardware to implement communication networks. As a result, the programming models for applications running on PC clusters should differ from those targeting parallel machines or supercomputers. For PC clusters, it is more difficult to exploit low-level or fine-grain parallelism potentially existing in programs. It is more appropriate to adopt coarse- or medium-grain programming models for PC clusters [2, 18]. This approach corresponds to fewer data transfers, thus it reduces the adverse effect of long communication delays. Load balancing among PCs becomes a critical issue in attempts to yield high performance.

In existing medium- to coarse-grain programming models, the way to decompose applications is normally function-oriented. For code reuse in a given application domain, these functions are implemented as library routines. For example, BLAS (Basic Linear Algebra Subprograms) contains "building block" routines for performing basic vector and matrix operations [5]. They are commonly used to develop high quality linear algebra software. Each routine completes a single operation, such as matrix-matrix multiplication, no matter how large the input matrices are. Workloads are controlled through data partitioning. Load balancing is based on workload estimation. In current programming models used with PC clusters, such as message-passing models, these routines may need to communicate with each other at run time through the communication layer. This makes their execution time much less predictable.

To effectively balance the workload and also support application portability, we introduce here the *super-programming model (SPM).* In SPM, the parallel system is modeled as a single virtual machine. SPM decomposes applications in a workload-oriented manner. It suggests that an effective instruction set architecture (ISA) be developed for each application domain. The frequently used operations in that domain should belong to this ISA. Their operand sizes are assumed limited by predefined thresholds. Application programs are modeled as *super-programs (SPs)* coded with SIs. Then, as long as an efficient implementation exists for each SI on given computing platforms, code portability is guaranteed. SIs are dynamically assigned to available processing units at run time. When the degree of parallelism in an SP is much larger than the number of nodes in the cluster, any node has little chance to be idle. Good load balancing becomes more feasible by focusing on scheduling at this coarser SI level. Another major advantage of SPM is that, because of the similarity between SIs for PC clusters and instructions for micro-processors, SPM can "borrow" many mature technologies developed for sequential computing. For example, utilizing cache technology, SPs can reduce the adverse effect of remote data loads.

The SPM approach can actually be adopted for any application. We have applied it to the simpler case of matrix multiplication [21]. In this paper, we apply the SPM approach to an apriori-like algorithm for mining association rules, in order to prove that it can address the load balancing problem very well even for this highly demanding application. Data mining involves data analysis and discovery; under acceptable computational times, it produces an enumeration of patterns over the data. Mining association rules is a typical data-mining problem. Techniques for discovering association rules have been studied extensively [6-8,10-12, 14, 19]. Data mining often requires high-performance computer systems [8, 9, 14-18]. In mining association rules, the workload is dynamically created. Thus, static load balancing is not expected to yield good performance. Several techniques have been developed to dynamically balance this problem using some form of load estimate [9, 15]. For example, when counting the support (i.e., frequency of appearance) of patterns, the CD algorithm allocates the workload by evenly partitioning candidate patterns [9]. It assumes that counting the support of different patterns has the same amount of work. It is expected to balance the workload only if this assumption is held and the number of candidate patterns is a good estimate of the real workload. However, transaction data may be loaded with some form of skewing [15]; the efficiency of the algorithm is then reduced. Additionally, the workload of several subtasks, like pruning a candidate pattern against the database to determine its support or generating next round candidate patterns, depends heavily on the already mined data. Therefore, better load balancing is needed with a good estimation of the assigned workload.

The next section presents our SPM model and focuses on its support for dynamic load balancing. Section 3 introduces the mining association rules problem and the Apriori algorithm. Section 4 introduces our SPM approach for an Apriori-like algorithm. Section 5 presents experimental results on a PC-cluster and relevant analysis. Section 6 carries out a theoretical analysis of SPM for this application and a comparison with another parallel approach to this data mining problem.

2. Super-Programming Model (SPM)

2.1 Super-Data Blocks (SDBs)

In SPM, the PC cluster is modeled as a single virtual machine (VM) with a single super-processor; the latter includes one instruction fetch/dispatch unit (IDU) and multiple instruction execution units (IEUs). The super-processor can handle a set of "build-in" data types and a set of basic abstract operators that can be applied to the former. These data types are stored in SDBs. The operations are performed by super-instructions (SIs). SDBs are high-level abstract data types. On each PC, they are expressed in their local format. Implementers are free to adopt any data structure supported by the languages used to implement VM. As long as the SDB formats have been set up along with the SDB exchange protocols, nodes with different architectures can freely exchange SDBs. This feature makes it very easy to work with heterogeneous clusters.

To execute an SI, the super-processor should first load its operands. This may involve remote loads if the operands are not cached locally. Since the communication latency for member nodes in a cluster is large, to prevent IEUs from starvation the execution time of SIs should be generally higher than the load latency. Thus, operands should have coarse enough granularity. Since SDBs are operands of SIs, SIs can have predictable execution time only if SDBs have limited size. SDBs include not only original data but also meta-data to characterize themselves. Objects in object-oriented languages are good candidates to be expressed with SDBs. To summarize, SDBs are application-domain specific.

2.2 Super-Instructions (SIs)

In SPM, SIs are high-level abstract operations applied to SDBs. They collectively constitute the basic operations of VM. Similarly to instructions for microprocessors: 1) SIs are subject to data dependencies. There is no communication logic embedded in their body. Dependencies are handled only at the beginning and end of an SI's execution. Once all operands are locally available, an SI can be executed without any interruption. 2) SIs are atomic. Each SI can only be assigned to and be executed on a single IEU. 3) The workload of each SI has a quite accurate upper bound for a given computer. The operands, of course, have limited size.

At the single IEU level, SIs can be implemented with any ordinary programming technique. SIs are portable throughout PCs in the cluster. For homogeneous cluster systems, this requirement is easy to achieve. But for a heterogeneous system, it becomes a major task. In a heterogeneous PC cluster, there must be multiple implementations of an SI. They correspond to software routines for each kind of computer in the cluster. The SI set is completely application domain dependent. Also, it is open to expansion to match the application's requirements, as deemed appropriate. For an effective ISA under SPM within a given application domain, the SI set should provide all required basic operations. This is called completeness of the SI set. The SI set encapsulates the underlying support system. Thus, all applications in the domain can be described with these SIs. This enhances software component reuse. Also, the SI set should be orthogonal for better program maintenance, ease of algorithm development, efficiency and good portability.

2.3 Super-Functions (SFs) and Parallelism

To facilitate ease of application development, programs are usually developed using high-level structures. The latter combine many simple low-level operations to form high-level abstract operations, such as reusable subprograms or functions. Entire programs consist of lists of such operations; they describe how a computer system should perform computations to solve the respective problem. In SPM, these functions are called SFs. SFs are "binary" executable procedures for VM that can be executed on the super-processor. Application programs are modeled as SPs which are implemented as collections of SFs. When an SP runs on a PC cluster, its SFs are called as they are encountered sequentially. The IDU fetches SIs in SFs and dispatches them to the IEUs. We apply multithreading so that each PC node may have, at any time, up to six active threads corresponding to different SIs. This number of threads is the result of our experimentation.

SFs play a critical role in the parallelization of SPs. The VM provides the mechanism to execute multiple SIs simultaneously. Super-instruction level parallelism (sILP) is supported by the super-processor "hardware." To sufficiently exploit this capability of the VM, SFs should provide sufficient SIs that can be executed in parallel. However, the IDU should check data dependencies among SIs in order to issue them correctly. The execution of these SIs is subject to stalls if there exist SDB dependencies. To increase the potential run-time parallelism, SFs can co-operate with the IDU to denote intrinsic parallelism in the problem. Also, dynamic information can be collected by the IDU to maximize the parallelism in already issued SIs so that SFs can efficiently utilize multiple IEUs and exploit the parallel capabilities of the system.

2.4 Runtime Environment

Let us now summarize the programming environment that we have developed to support SPM. To support VM at runtime, a set of processes are launched on member PCs. An IEU is represented as such a working process that provides a runtime environment to execute SIs. The IDU is represented as a separate process or a set of function threads embedded in IEUs. In the former case, the IDU centrally dispatches all SIs. In the latter case, the IDU is distributed; its member threads will co-operate to control the dispatching of SIs to IEUs. Collectively, these processes provide the runtime environment to execute SPs. This environment provides the basic SPM implementation layer. It is completely independent of any application domain.

Since both SDBs and SIs are high-level abstractions in SPM, a global name space is essential. In a PC cluster, computer nodes are independent processing units that run their own copy of the operating system. IEU processes running on these computers have their own independent virtual logical address space. Data with the same address in these IEUs on different PCs may be of completely different nature. But an SI may execute on any member computer in the cluster. Thus, SIs cannot reference their operands using logical addresses in IEUs. There must be a global logical name space. All data objects exist in this space and are identified with distinct IDs. SIs can reference operands in this space. Each object may physically exist in a file or be cached in the memory of a member node. This global space is independent of the logical spaces of IEUs. It does not even map directly to the latter. When an SI is assigned to an IEU, the runtime environment of this IEU maps the corresponding objects into its own logical address space. This is similar to mapping the logical address space of a program to the physical address space in the memory of a PC. The runtime environment provides the service to load SDBs into the IEU address space. We can imagine the file system being shared by all nodes in the "main memory" of VM. Nodes also share information at their local-memory level which can be viewed as "Level-2 (L2) cache". Once all operands (SDBs) of an SI have been loaded into the local IEU, the SI can be executed by launching a thread within the IEU process to execute a local procedure that operates on these objects. Besides IDs, these references of SDBs in an SI may also include some meta-data about the respective data objects. Of course, SIs may also use immediate data.

2.5 Load Balancing

In SPM, all parallelism in an SP is mapped to SI-level parallelism, so load balancing focuses on SI scheduling. Load balancing is controlled completely by the IDU. In the VM, all IEUs are general-purpose and can execute any SI. Therefore, the IDU can assign any SI to any IEU. On the other hand, unlike functional units in micro-processors, IEUs work together in the asynchronous mode. When and only when an IEU is free to execute an SI, does it notify the IDU for the assignment of a new SI, if the instruction queue of the IDU is not empty. Of course, this SI should not be subject to any SDB dependencies that can stall it at that time. Thus, as long as SPs provide enough parallelism, no IEU will be under-loaded. This load balancing mechanism is based on a producer-consumer protocol. It does not pre-allocate tasks among member computers based on their performance and the estimated workload of tasks. The controller (IDU) always gets feedback from the IEUs. A computer that can finish the execution of SIs assigned to it more quickly than others will be assigned more work. The system can be unbalanced only if SPs do not have enough parallelism. For a good program, such a problem usually appears when the last SI has been assigned before a synchronization operation. However, SIs have limited workload. So, this situation does not last long time. To facilitate simplicity and efficiency in practice, SIs are not required to have identical workloads. SIs should be designed with two relevant properties in mind: large tasks must be divided into smaller ones and small to medium diversity should be allowed among the workloads of SIs. Despite the displayed generosity in this SI set design, a scheduling policy using a producer-consumer protocol will balance the workload well for an application with abundant parallelism.

To conclude, all parallelism in SPs is mapped to SI-level parallelism. The IDU can issue simultaneously SIs belonging to different SFs as long as they do not have data dependencies. Then, good load balancing becomes more feasible by scheduling operations at this coarser SI level. This load balancing mechanism does not distinguish among IEU architectures and their performance. Thus, it works in both heterogeneous and homogeneous environments. As long as an efficient implementation exists for each SI on each given computer, the difference between architectures is not important.

3. An Algorithm for Mining Association Rules

3.1 Basic Concepts

Let $I = \{ a_1, a_2, a_3, \dots , a_m \}$ be a *set of items* and $DB = \langle T_1, T_2, T_3, \dots ,T_n\rangle$ be a *transactions* database with items in I. A *pattern* is a set of items in I. The number of items in a pattern is called the *length of the pattern*. Patterns of length k are sometimes called *k-item patterns*. The *support s(A) of a pattern* A is defined as the number of transactions in DB containing A. Thus,

$$s(A) = |\{ T|\ T \in DB, A \subseteq T\}|.$$

A pattern A is a *frequent pattern* (or a *frequent set*) if A's support is not less than a predefined *minimum support threshold* s_{min}.

A rule is an expression of the form R: $X \rightarrow_{s(R),\alpha(R)} Y$, where X and Y are exclusive patterns of I ($X \cap Y = \varnothing$). X and Y are called the *pre-pattern* and *post-pattern* of R respectively. s(R) and $\alpha(R)$ are the support and confidence of the rule R, respectively. The *support s(R) of rule* R is defined as the support s(Z) of the joint pattern $Z = X \cup Y$. The *confidence* $\alpha(R)$ *of rule* R is defined as s(Z)/ s(X). An *association rule* is a rule with support not less than a minimum threshold. Given a transactions database, a minimum support threshold s_{min} and a minimum confidence threshold α_{min}, the problem of finding the complete set of association rules $M = \{ R: X \rightarrow_{s(R),\alpha(R)} Y \mid s(R) \geq s_{min}$ and $\alpha(R) \geq \alpha_{min} \}$ is called the *association rules mining problem*. Also, given a transactions database and a minimum support threshold s_{min}, the problem of finding the complete set of frequent patterns is called the *frequent patterns mining problem*.

A number of algorithms have been developed for the association rules mining problem. Many are frequent-pattern based. They first solve the frequent patterns mining problem, and then check the confidence of all candidate association rules which are built with frequent sets and their subsets. Since the supports of the pre-pattern X and post-pattern Y of a rule are not less than the support of their joint pattern Z ($X \subset Z$, $Y = Z - X$), if Z is a frequent pattern then X and Y also must be frequent patterns. A rule can only be found between a frequent pattern Z and its subset (X).

3.2 Apriori Algorithm

In frequent-pattern based algorithms, the Apriori algorithm is one of the most popular. It was first proposed by Agrawal et. al. [7]. Many relevant studies [13,16] adopt an Apriori-like approach. The Apriori algorithm is based on the anti-monotone property of frequent patterns. This property states that any super-set of a non-frequent pattern can never be a frequent pattern. Of course, all sub-sets of frequent patterns are frequent patterns. Thus, any verified short frequent patterns can help in screening longer candidate patterns. The Apriori algorithm can be represented by the following pseudo-code that finds all frequent patterns:

```
I’ = {x | $x \in I$ and x is a frequent item}  // Find all frequent items by scanning DB
$P_1$= { 1-length patterns p |  p ={x} and $x \in I'$}
For (k = 2; $P_{k-1} \neq \emptyset$ ; k++) do begin
    $C_k$ = apriori_gen($P_{k-1}$)
    For any pattern $c \in C_k$  {c.count = 0}
    For all transactions $T \in DB${
        For any pattern $c \in C_k$   {  if ($c \subseteq T$ ) c.count++  }
    }
    $P_k$ = {c | $c \in C_k$, c.count $\geq s_{min}$ }
End     Answer = $\cup P_k$
```

3.3 Parallel Algorithms for Mining Association Rules

Since mining association rules is a very costly computational process, many relevant parallel algorithms have been developed. The most popular ones exploit data parallelism. Many studies have proved that computing the counts of candidate patterns is the most computationally expensive step of the algorithm. The only way to compute these counts is to scan the entire transactions database. Each transaction is checked to see if it supports some candidate patterns. Thus, most algorithms focus on computing the support of patterns in parallel [8,9,14]. To speed up this operation, a hash tree of candidate patterns is used.

The algorithms that count supports in parallel can be classified into two basic types based on what types of data are partitioned. They are classified as either *count distribution (CD)* or *data distribution (DD*) algorithms [8]. In a CD algorithm, the entire candidate set is copied into all the nodes. However, transaction data are partitioned and each node is assigned an exclusive partition. Each node computes the support of all candidate patterns against its locally stored partition (a subset of DB). The global supports of candidate patterns are computed by summing up these individual local support counts. The count distribution algorithm has two potential problems related to memory scalability and load balancing. When the problem size increases, then the set of candidate patterns also increases. This requires the memory size of all the nodes to increase as well. The system, however, usually increases in size by increasing the number of nodes rather than improving the structure of individual nodes. Thus, if the cardinality of the set of generated candidate patterns increases, some or all of the nodes may no longer be able to hold the entire hash tree. Then, the algorithm has to partition the hash tree and scan the assigned database partition many times, once for each partition in the hash tree. Since the entire algorithm is developed in the synchronous co-operation mode, the global support cannot be computed until all local supports have been computed. Therefore, load balancing becomes a more critical issue and the workload should be balanced in each processing round.

DD algorithms address the memory scalability problem by partitioning the set of candidate patterns for exclusive assignment to the processing nodes [8]. This partitioning may be done in a round-robin fashion. Each node is responsible for computing the counts for its locally stored subset of the candidate patterns for all the transactions in DB. After the supports of all candidate patterns have been computed, each node finds the frequent patterns in its local set of candidate patterns, prunes the set of candidates to its set of frequent patterns, and then sends these frequent patterns to all the other nodes in order to generate the set of candidates for the next round. To achieve local counting, each node must scan all components of DB that are distributed throughout all the nodes. Therefore, each node receives the portions of the transactions data in DB stored in the other nodes. DD algorithms do not address load balancing very well. Although they partition the set of candidate patterns among the nodes and then use the same transaction data, the workload for determining the supports of the candidate patterns is, however, data dependent. Therefore, these workloads are irregular. Each node may need different time and its assigned workloads cannot be estimated before the actual operations are performed.

4. Our SPM approach for Mining Association Rules

Since our objective is not to compare algorithms for the association rules mining problem, but to build a programming model for balancing the workload in parallel implementations of such algorithms, we choose the Apriori algorithm as a typical example for discussion.

4.1 Data Blocks for Mining Association Rules

We have designed the following types of SDBs.

1. **BlockOfItems:** includes a list of distinct items which are sorted in a predefined order. Each block covers a continuous, exclusive partition.
2. **BlockOfTransaction:** includes a list of transaction data. Each transaction is a sorted list of distinct items.
3. **BlockOfJoinResult:** includes sorted candidates of frequent patterns. They have not been screened to see if their sub-patterns are frequent patterns.
4. **BlockOfCandidates:** includes sorted candidates of frequent patterns that passed the sub-set screening.
5. **BlockOfFrequentSet:** includes sorted frequent patterns and their supports.
6. **BlockOfRules:** includes a limited number of association rules.

4.2 SI Set for Mining Association Rules

For mining large transaction databases, we have designed the following SI set.

1. **LoadDataBlock (in DS, in s, out rawId):** gets a block of data outside of the system. “DS” is the reference to the external data source; “s” is the maximum size of the fetched data; “rawId” is the global ID of a BlockOfTransaction block to hold incoming data.

2. **CountItemSupport (**in **rawBlockId,** in/out **itemsId):** extracts all distinct items and counts the number of times each item appears in a block of raw transactions (identified by "rawBlockId"); with the help of a global mapping object (identified by "itemsId"), it also merges results into respective global data blocks.
3. **ShrinkItemBlock (**in/out **itemsBlockId,** int **minSupport):** This SI prunes items in a BlockOfItems block (identified by "itemsBlockId") by removing all items with counts less than the minimum support "minSupport."
4. **GetFrequentItemsBlock (**in **items,** out **fItemBlockId,** out **fSetBlockId,** in/out **fMapId):** collects all data contained in a list of pruned blocks of items, creates an SDB (identified by "fItemBlockId") of frequent items and an SDB (identified by "fSetBlockId") of 1-length frequent patterns, and then sends appropriate information to the global mapping object (identified by "fMapId"). "items" is the list of global IDs of pruned blocks of items..
5. **ShrinkTransactionBlock(**in **rawId,** out **itemsId):** shrinks a block of transactions based on a block of frequent items. It removes all non-frequent items from every transaction. The parameter "rawId" identifies an SDB of transactions. The parameter "itemsId" identifies an SDB of frequent Items.
6. **MergeTransactionBlock (**in/out **list):** merges a list of pruned SDBs of transactions. The parameter "list" is a list of global IDs of pruned blocks of transactions. The first ID in the list is also used as the ID of the generated data block.
7. **GenxxxCandidatesFamilyBlock (**in **itemSet,** in **frequentBlockId,** in **frequentMapId,** out **candsBlockId):** joins an n-length frequent pattern with all n-length patterns that follow and have the same prefix in the SDB identified by "frequentBlockId"; it generates a "BlockOfJoinResult" type SDB of (n+1)-length candidate patterns that is identified by "candsBlockId" . This SI has multiple versions. "xxx" may be "Large", "Middle" or "Small" based on the format of the first parameter "itemSet" that is the first set of n-length patterns. The parameter "frequentMapId" is the ID of the global mapping object that maps the partitions of n-length frequent patterns to IDs of SDBs.
8. **FilterCandidates (**in/out **candsBlockId,** in **frequentBlockId,** in **frequentMapId):** screens candidate patterns in an SDB identified by "candsBlockId" with an SDB of n-length frequent

patterns identified by “frequentBlockId”. The parameter “frequentMapId” is the ID of the global mapping object that maps the partitions of n-length frequent patterns to IDs of SDBs.

9. **CountBlockCandidatesSupport (**in/out **candsBlockId,** in **transBlockId):** counts the partial support of the candidate patterns in a candidate block identified by “candsBlockId” in an SDB of transactions identified by “transBlockId”.
10. **PruneCandidatesBlock(**in/out **candBlockId,** in **minSupport):** prunes a block of candidates identified by “candBlockId” by removing candidates with support less than the threshold “minSupport”.
11. **GetFrequentSetBlock (**in **list,** out **fid,** in/out **fMapId)**: merges a list of pruned SDBs of candidate patterns, generates a permanent SDB of frequent patterns, sends the range information to the global mapping object and publishes the block in the global space. The parameter “list” is a list of IDs of pruned SDBs. The parameter “fid” is the ID of a generated SDB of frequent patterns. The parameter “fMapId” is the ID of a global mapping object that maps a range of partitions of frequent patterns to an SDB of frequent patterns.
12. **CheckConfidenceInBlock (**in **preFSetsBlockId,** in **postFSetsBlockId,** in/out **rulesBlockId,** in **minConfidence):** extracts association rules by checking frequent patterns in an SDB with their sub-patterns in another SDB against a threshold “minConfidence”. The parameter “preFSetsBlockId “ is the ID of a data block that includes the (n-i)-length sub-patterns, where i is an integer. The parameter “postFSetsBlockId” is the ID of the SDB that includes n-length frequent patterns. The parameter “rulesBlockId” is the ID of an SDB of rules.
13. **StoreResult (**in **rulesId,** out **des):** stores a list of generated rules in the external storage. The parameter “rulesId” is the ID of an SDB of rules. The parameter “des” is the reference to a destination in the external storage.

4.3 Super-Programming for Mining Association Rules

The high-level description of the SP is:

```
$P_1$ = initial(DS, $s_{min}$)    //DS is the external data source
For (k = 2; $P_{k-1} \neq \emptyset$ ; k++) do begin
    $C_k$ = gen_candidate($P_{k-1}$)
    $P_k$ = gen_frequentSet ($C_k$, $s_{min}$ )
    $R_k$ = find_rules($\cup_{j<k} P_j$, $P_k$ )
**End**
Answer = $\cup R_k$
```

Some SFs may run in parallel during the execution of the SP. Once “gen_candidate” is finished, both “gen_frequentSet” and “find_rules” are called. And when “gen_frequentSet” is done, we do not have to wait for “find_rules” to finish before the next iteration begins.

1. <u>The “initial” SF</u>

“initial()” finds the value domain of the items, identifies all distinct items and stores them in a sorted list of SDBs; it also counts them and prunes these blocks. It then generates a list of data blocks of 1-length frequent patterns. In an SP, its implementation is:

```
while (there is more data){ LoadDataBlock (DS, s, $rawBlockId_i$); }
parallel do  for all $rawBlockId_i${    CountItemSupport ($rawBlockId_i$, itemsId); }
parallel do for all blocks in item list { ShrinkItemBlock( $itemsBlock_i$, s); }
parallel do for all blocks $list_i${ GetFrequentItemsBlock ($list_i$, $fItemBlock_i$, $fSetBlock_i$, fMap); }
parallel do for all blocks in transactions list $rawBlockId_i$ {
    while (there are more $itemsBlock_i$ blocks that must be checked)
        ShrinkTransactionBlock($rawBlockId_i$, $itemsBlock_i$); }
```

2. <u>The “gen_candidate” SF</u>

The implementation of this SF is:

```
parallel do all data blocks of $frequentBlock_i$ {
    parallel do all same and following data blocks $frequentBlock_j$ {  //$i \leq j$
        if (can be joint) {
            GenxxxCandidatesFamilyBlock (itemSet in $frequentBlock_i$, $frequentBlock_j$,
                        frequentMapId, $candsBlock_m$) //version of SI based on data
            while ($candsBlock_m$ still need check by $frequentBlock_p$) {
                FilterCandidates ($candsBlock_m$, $frequentBlock_p$, frequentMapId) }
        }
    }
}
```

3. **The "gen_frequentSet" SF**

The implementation is:

```
parallel do for all data blocks of k-length candidate patterns $candsBlock_m$ {
    parallel do for all blocks in transactions list $transBlockId_i$ {
        CountBlockCandidatesSupport ($candsBlock_m$, $transBlockId_i$) }
    PruneCandidatesBlock($candsBlock_m$, $S_{min}$) }
parallel do for all partition $list_i$,{ GetFrequentSetBlock ($list_i$, $fid_i$, fMapId); }
```

4. **The "find_rules" SF**

The implementation is:

```
parallel do for each data model of k-length frequent patterns block $frequentBlock_p$ {
    parallel do for each SDB of frequent patterns block $frequentBlock_m$ {
        if (there are more SDBs that include sub-patterns needed to be checked){
            CheckConfidenceInBlock ($frequentBlock_m$, $frequentBlock_p$, $rulesBlock_i$,
                minConfidence) }
    } }
```

5. Experimental Results and Analysis

5.1 Experimental Setup

All the experiments were performed on a PC cluster with six nodes. Each node contains two AMD Athlon processors running at 1.2 GHz. Each node has 1GB of main memory, a 64K Level-1 cache and a 256K Level-2 cache. All the nodes are connected via an Ethernet switch. Each link has 100Mbps bandwidth. All the PCs run Red Hat 7.3. All nodes have the same view of the file system by sharing files via an NFS file server.

We used synthetic databases with sizes ranging from 2MB to 400MB; they were generated using the program provided in [20]. The generated databases are listed in Table 1. The parameters for the databases are implied from their name. The number of transactions follows the letter "D". The average number of items per transaction follows "T". The average length of the maximal pattern follows "I". The number of distinct items N is 1000; the number of patterns is 10000. Each database is stored in a single file. Thus, loading them is a sequential process.

Table 1. Synthetic databases used.

Name
T25.I10.D3K
T25.I10.D10K
T25.I10.D30K
T25.I10.D100K
T25.I10.D500K

Table 2. Parameters of workload.

Name	Value
Maximum size of block of items	1000
Maximum size of block of transactions	1000
Maximum size of block of candidate patterns	1000
Maximum size of block of rules	1000
Maximum number of hosted data blocks	10000
Maximum number of cached data blocks	10000

The SP and SFs were manually written using the SIs described in Section 4. "LoadDataBlock" SIs in the "initial" SF load all external transaction data into SDBs. Then, the SP runs on VM. The runtime environment and the SIs were implemented in the Java language. The IDU was implemented in a mixture of modes. It works as follows. Each IEU maintains a local pool of SIs. It is a priority queue. SIs are put into the pool by the SFs that are executed on the node when the SIs can be issued. SIs are assigned a priority by SFs to control the order of issue. Besides these SI pools, a centralized process records the highest priority in each node. When an IEU is free to receive another SI, it compares the global highest priority with the local one. If they are equal, the IEU fetches an SI from the local pool. Otherwise, it gets an SI from another node that has the highest priority SI.

A few special functions were added in the code that implements the runtime environment; they collect information at runtime about the utilization of individual PC nodes. When no SI is executed on a node, the node is considered to be idle. The information includes the total time elapsed, the time for the execution of sequential code, the total time for the execution of SIs in parallel and the total idle time. During the experiments, each member computer launches a runtime environment process. The VM caches SDBs as much as possible by processes run on the host. The runtime environment also caches recently used SDBs. When an IEU needs some data that are not cached locally, it first tries to find them in its peers. Table 2 shows the values of relevant parameters.

We run repeatedly the same SP for each mining problem, for different numbers of nodes in the cluster. Each case was run many times. The data presented here are average times of multiple runs. In this paper, our study focuses on load balancing. We also implemented the HPA (Hash Partitioned Apriori) algorithm that appeared in [17] for the sake of comparing their effectiveness in load balancing. HPA adopts a modified DD strategy. It uses a hash function instead of a round-robin approach to partition the candidate patterns among the member nodes. All the nodes know the hash function. Thus, it is not necessary to broadcast all the transaction data to all the nodes. When generating k-length candidate patterns, each node gets all (k-1)-length frequent patterns in order to generate the candidate patterns, and applies the hash function to determine the destination node for every candidate pattern. If the destination is this node, it inserts the candidate pattern into its hash table of candidates; otherwise, it is discarded. After generating all candidate patterns, each node reads the transactions database (the local partition) from its local disk, generates all k-length patterns for each transaction and applies the hash function used in the candidate generation stage. It derives the destination node ID for each k-length pattern and sends the pattern to this node. Every node receives the patterns sent from other nodes and uses them, including those generated locally, to count the candidate patterns in its candidate hash table. It then finds the k-length frequent patterns among them. When the number of nodes is larger than the average number of k-length patterns generated from the transactions, the load of transmitting data among the nodes is reduced. We also added trace code to find the idle time of member nodes. Comparison was performed by running both programs to solve the same cases. The transactions in our data set were longer than those in [17].

The cache hit ratio depends on the cache size, the actual data and the access patterns. To prevent useful cached data from swapping out of the local disk, the number of cached objects is limited by the maximum number of cached data blocks. Thus, in our experiments the cache size is controlled by this parameter rather than by the available space in the local memory. For a fair comparison with HPA, our experiments employ the same datasets for SPM and HPA, and the maximum number of cached data blocks is constant (as shown in Table 2) in the cached data cases. The differences in cache hit rates for the SPM and HPA techniques result from their access patterns.

To also examine the effect of the SDB size on the performance of SPM, we run our SP program for various SDB sizes. We investigated SDBs having sizes 100, 200, 300, 500, 1000, 2000, 3000, 5000, 10000, 20000, 30000 and 50000, if there was enough data. The SDB size affects the performance in many ways. Firstly, if it is too large then the number of tasks may be small and it will be difficult for the scheduler to balance the workload. Secondly, it affects the communication overhead. If the SDB is very small, the communication overhead may be longer than the computation time and, therefore, it cannot be hidden by running multiple SIs on each node. On the other hand, for large SDBs the overhead of loading operands may be comparable to actual computation times. However, multi-threading reduces tremendously the overall effect of this overhead. Our program runs showed that the plot of the total execution time as a function of the SDB size has a U shape. That is, the best execution times result from choosing SDBs of medium size, such as 1000. The exact plot also depends on the dataset size. The lower part of the curve becomes wider for larger datasets because many tasks can still be created with larger SDBs. Thirdly, increasing the SDB size significantly for very large datasets may inadvertently affect the performance due to cache memory limitations resulting from the SPM runtime environment and virtual memory implementation. The former environment caches a limited number of objects in each member PC, so for large datasets a cache replacement policy will have to be applied many times. Also, the virtual memory mechanism provided by the operating system of member PCs will require a large number of page replacements. However, the SPM and virtual memory environments view the data differently. SPM applies a global policy for SDBs while the operating system applies a local page replacement technique for a given PC. We should like ideally the access of disk data to be initiated by SPM. However, the actual size of SDBs is parameterized and the same type of SDBs may occupy different amounts of memory. Thus, the object cache technique cannot completely hide the virtual memory technique, even if we change the SDB size for fixed cache size. For larger SDB sizes this effect appears more often. Our experiments show that SDBs of size 1000 are a good choice for small- to medium-sized datasets. For very large data sets (e.g., T25.I10.D100K), the best SDB size may be larger but the improvement in performance is less than 10%.

5.2 Experimental Results

Table 3. Summary of execution times and idle times in milliseconds for SPM.

Problem	Number of nodes	1	2	3	4	5	6
T25.I10. D3K minSupport =0.6%	Average idle time	191	3556	5101	7612.75	8881	10571.8
	Idle time in computing*	191	1266.3	3798.6	6125.95	6441.32	7837.17
	Total time	71170	63209	53217	54884	53295	51285
	Computing time	68757	59901	50775	52406	49483	47183
T25.I10. D10K minSupport =0.6%	Average idle time	440	5478	8177.67	9029.5	11049.2	11195.5
	Idle time in computing*	440	2716	3864.07	4333.9	5399.92	5736.83
	Total time	146062	103904	97505	85210	88830	86648
	Computing time	139947	96999	89417	77384	80003	78460
T25.I10. D30K minSupport =0.6%	Average idle time	217	9340	14089.7	14580	18543.8	17745
	Idle time in computing*	217	2617.2	4396.87	3977.4	5713.72	4819.67
	Total time	376079	231358	211133	158794	162268	157674
	Computing time	361751	214551	192959	141123	142221	138286
T25.I10. D100K minSupport =0.6%	Average idle time	306	19484	32092	37448.5	37304.8	49967.2
	Idle time in computing*	306	2696	8015.5	3437.5	8070.4	13157.8
	Total time	1234613	701748	515564	547617	377532	431875
	Computing time	1189756	656980	467411	487153	328808	372980
T25.I10.D500K minSupport =0.6%	Average idle time	1912	85293	134191	144945	182876	196851
	Idle time in computing*	1912	53622	78463.2	32467.4	90941.6	91267.8
	Total time	6883521	4825186	3081771	3263902	2325782	2585250
	Computing time	6652822	4571818	2747404	2664023	1866108	2078452

* excluding the sequential load data stage

Table 3 presents a summary of the total program running time, actual computation time, total idle time of nodes during actual computation and average idle time of nodes for each mining problem in our experiments. We present the computation time because the initial part of the super-program loads data sequentially. Fig. 1 shows the percentage of total idle time (relative to the total execution time) under various conditions. Also, Fig. 2 shows the percentage of idle time in the computing stage (relative to the total computation time) under various conditions.

In both cases, the idle time increases with the number of nodes. The percentage of idle time in computing is always significantly less than the overall percentage of idle time. This is because when a node is loading transaction data sequentially, no other node can begin execution of any SI. With increases in the number of nodes, the number of potentially idle nodes increases. Thus, the accumulated idle time of the whole system increases. With increases in the number of nodes, the total execution time is reduced and the effect of the sequential component on the percentage of idle time increases. If we only consider the idle time in computing, we can see that for a small problem (e.g., D3K) the percentage of idle time is significant. The longer idle time for a small problem may be due to the fact that there are not enough parallel SIs available for execution. For larger problems, with the same size of data blocks, the data parallelism increases. The previous effect is reduced in scale. Experimental results show that, for a larger problem the idle time increases much slower and the increase is less than linear. In our experiments, the percentage of idle time is not more than 8% for large problems.

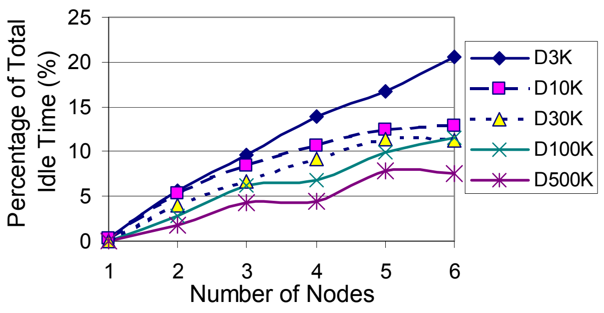

Fig. 1. Percentage of average node idle time as a function of the total number of nodes.

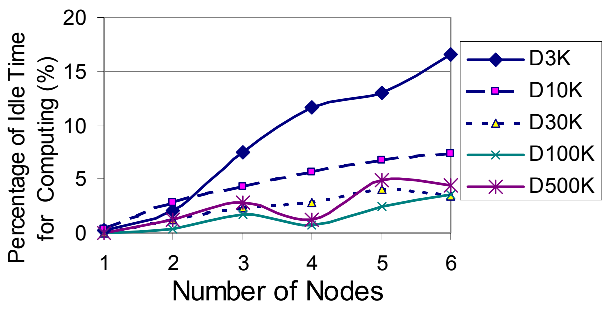

Fig. 2. Percentage of idle time in computing as a function of the total number of nodes.

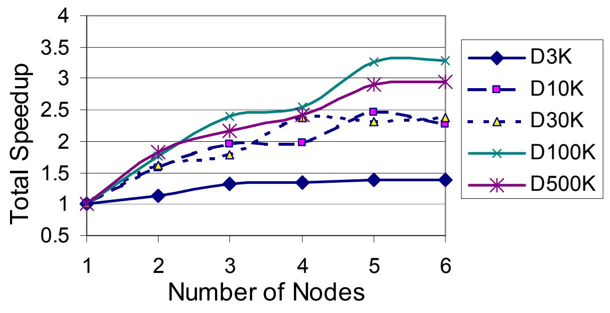

Fig. 3. Total speedup as a function of the total number of nodes.

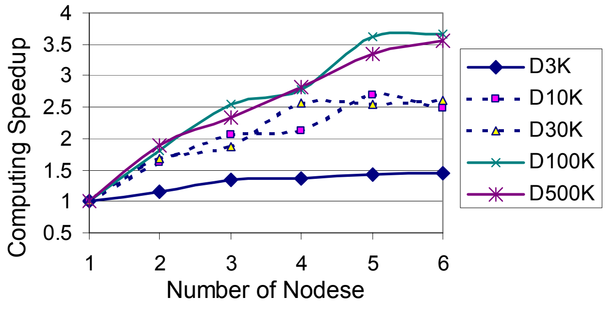

Fig. 4. Speedup in computing as a function of the total number of nodes.

Fig. 3 shows the total speedup of our super-program executing on the PC cluster as a function of the total number of PC nodes. Fig. 4 shows the speedup in the actual computing phase as a function of the total number of PC nodes. For all cases, the speedup increases with the number of nodes. However, the rate of increase depends on the size of the mining problem. For the T25.I10.D3K problem, the speedup with 6 nodes is less than 1.5. However, for larger problems, such as T25.I10.D100K or T25.I10.D500K, the speedup with 6 nodes is greater than 3.6. A comparison of SPM with HPA is shown in Fig. 5. We can conclude that SPM results in lower idle times. SPMt and SPMc correspond to idle times for the entire SP and its computing phase in the SPM approach, respectively.

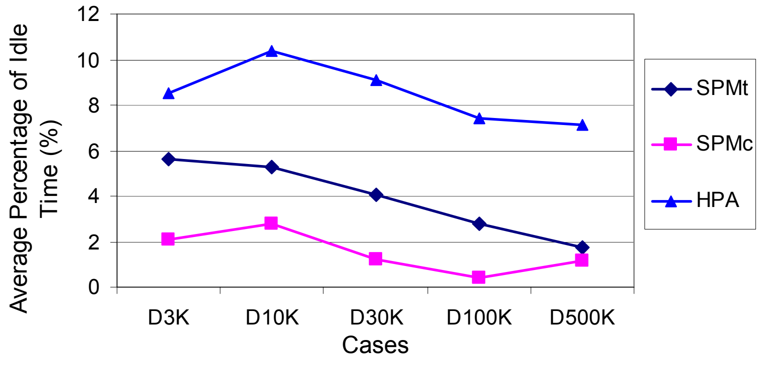

Fig. 5. Percentage of idle times for the SPM and HPA algorithms.

6. Discussion and Further Comparisons

6.1 Parallelism in SFs

6.1.1. Parallelism in the “initial” SF

This SF loads external data, counts distinct items, selects frequent items, creates 1-length frequent patterns and builds internal SDBs of transactions. As described in Section 4, the first step (load external transaction data) may be completely sequential, if no special arrangement is made for the portability of input data. Despite this, the other steps are parallel. To count the support of items, if the total number of transactions is n_{tran} and the size of the SDB of transaction data is s_{tran}, then the number n_{d_tran} of SDBs of transaction data is n_{tran} / s_{tran}. The degree of parallelism in counting the support is n_{d_tran}. To shrink an SDB of items, if the total number of distinct items is n_{item} and the size of the SDB of items is s_{item}, then the degree of parallelism is n_{item} / s_{item}. If the number of frequent items is $n_{f\text{-}item}$, then the degree of parallelism in generating 1-length frequent patterns is $n_{f\text{-}item} / s_{item}$. The degree of parallelism for shrinking transactions is also n_{d_tran}. In our experiments, $s_{tran} = 1000$. For D3K, $n_{tran} = 3000$ and the degree of parallelism is as low as 3. This example shows the effect of the block size on parallelism.

6.1.2. Parallelism in the "gen_candidate" SF

The process of generating candidate patterns is highly parallel. If the total number of joined candidate patterns is n_{j_can} and the size of SDBs of candidates is s_{j_can}, then the degree of parallelism is n_{j_can} / s_{j_can}. However, n_{j_can} is not known in advance. It strongly depends on the criteria used for mining and the actual data being mined. It is not simply determined by the number of frequent patterns in the preceding round.

6.1.3. Parallelism in the "gen_frequentSet" SF

The process of generating frequent patterns is highly parallel. If the total number of candidate patterns is n_{can} and the size of SDBs of candidates is s_{can}, then the number of SDBs of pruned candidates n_{d_can} is (n_{can} / s_{can}). The degree of parallelism in this SF is $n_{d\_can} * n_{d\_tran}$.

6.1.4. Parallelism in the "find_rules" SF

This SF for finding the rules can also be done in parallel. To avoid useless checking operations, we, however, do the test of confidence partially in parallel. For an SDB of frequent patterns, only the blocks that include their longest sub-patterns are checked to extract association rules. If and only if some association rules are discovered for k-length pre-patterns, data blocks that include their shorter (k-1)-length sub-patterns are then checked. Thus, blocks including the same sub-pattern length are checked in parallel. Blocks including different lengths of sub patterns are checked sequentially. The degree of parallelism is $k*n_{d\_freq}$, where k is the length of the frequent patterns (it is also the number of the longest sub-patterns) and n_{d_freq} is the number of SDBs of frequent patterns generated in the k-th round.

6.1.5. Parallelism Among SFs

It seems that only “gen_frequentSet” and “find_rules” can be executed in parallel. However, by unfolding the loop multiple “find_rules” SFs may be executed in parallel. Even though there is only a low degree of parallelism at this level, by merging the degrees of parallelism of member SFs we increase the total parallelism. Although each SF may have less parallelism at the end of its execution, this lack of parallelism may appear at different times. When an SF lacks parallelism, another SF may still have abundant parallelism at that time. Then, the entire program can satisfy the requirement of high parallelism.

For mining association rules, in the first few rounds there is usually a very large number of candidate patterns. The large values of n_{j_can} and n_{can} make the functions of generating the candidate and frequent patterns to have a very large degree of parallelism. At this stage, the length of frequent patterns is very small and the number of sub-patterns for each pattern is also small. Thus, at this stage the degree of parallelism in the SF that finds rules is very low. When the process reaches the last few rounds, the volume of candidate patterns definitely drops dramatically. The small values of n_{j_can} and n_{can} make the functions of generating the candidate and frequent patterns to have very small parallelism. However, our experiments show that the length of frequent patterns is then very long and the number of sub-patterns for each pattern is very large. Thus, the degree of parallelism in the SF that finds rules is then very high. This reverse change makes the parallelism of the entire application rather stable throughout execution.

6.2 Performance Model

Let us now carry out theoretical analysis of performance. Assume a total number of N_{SI} SIs for mining a given database with given criteria (i.e., minimum support and minimum confidence) and a homogeneous cluster with n PCs. Assume that these PCs will be idle for a portion p_{idle} of time and will execute SIs all other times. The total time to solve the problem is estimated as $T= (t_{avg}(n) * N_{SI})/ (n * (1- p_{idle}))$, where $t_{avg}(n)$ is the average time to execute an entire SI. p_{idle} is determined by the parallelism in SP and the number of PCs in the cluster. The experimental results show that it is low. Thus, we can ignore it. $t_{avg}(n)$ includes not only the actual time to execute the SI but also the time to load the operands for SDBs. The former time may only depend on the parameters used by the SI, but not on the number of nodes and the problem itself. The latter, however, strongly depends on the bandwidth of the network in the cluster and the actual location of these data blocks. We distinguish among different implementations below, based on data accesses.

1. Completely cached model

For small- to medium-sized problems, all data blocks may be stored in the main memory of the nodes (i.e., cached by VM). We assume that all nodes have identical memory capacity. Also, assume that each node hosts 1/n-th of the data blocks. Every SI has 1/n chance to load a data block from the local memory and (n - 1)/ n chance from the memory of a remote node. Assume that the average CPU time to execute an SI is t_{exe}, the CPU average time to load a local SDB is $t_{local-load}$ and the average CPU time to load an SDB from a remote node is $t_{remote-load}$. Then, the expected total time to execute an SI that loads local operands is $t_{exe-local} = t_{exe} + t_{local-load}$. The average total time to execute an SI that loads remote operands is $t_{exe-remote} = t_{exe} + t_{remote-load}$. Finally, the total time to execute an SI is estimated by:

$$t_{avg}(n) = (1/n)^* t_{exe-local} + (n-1)/n * t_{exe-remote} \tag{6-1}$$

We define "a" to be equal to $t_{exe-remote} / t_{exe-local}$ and we call it the *remote delay coefficient*. It is greater than one. If we temporarily ignore p_{iddle}, then the total execution time of the program on a cluster with n nodes is:

$$T_n = (t_{avg}(n) * N_{SI}) / n = ((n-1)*a + 1)* t_{exe-local} * N_{SI} / n^2 \tag{6-2}$$

When n = 1, the super-program is executed sequentially on a node. The execution time is:

$$T_1 = t_{exe-local} * N_{SI} \tag{6-3}$$

Thus, the speedup of computing with n nodes compared to sequential implementation is:

$$S(n) = T_1 / T_n = n^2/((n-1)*a + 1) \tag{6-4}$$

We have $a = (t_{exe} + t_{remote-load})/(t_{exe} + t_{local-load})$. It varies with $t_{remote-load}$ that depends on the amount of data transmitted in the network, which, in turn, indirectly depends on the number of nodes. By increasing n, we also increase the chance that an SDB is loaded from a remote location. These factors make the effect of the number of nodes n on "a" to be complicated. For example, if a = (n+1), then S(n) = 1; the system will never achieve a speedup. However, this dependence is weak. Under normal operating conditions, $t_{remote-load}$ does not vary significantly with n and its effect on "a" is weakened by t_{exe}. So, we assume that "a" is a constant.

When $n \rightarrow \infty$, S(n) = n/a. The efficiency of the system is:

$$E(n) = S(n)/n = 1/a \tag{6-5}$$

Therefore, the well-known isoefficiency property is satisfied.

Considering level-1 cache, we assume that each node not only hosts part of the data but also redundantly caches some data hosted by peer nodes. If the local cache missing rate is d, where $0 < d < 1$, then the effect on the execution time of an SI with operands located in remote nodes is:

$$t'_{exe\text{-}remote} = (1-d) * t_{exe\text{-}local} + d * t_{exe\text{-}remote} = ((1-d) + d*a) * t_{exe\text{-}local} = a' * t_{exe\text{-}local} \qquad (6\text{-}6)$$

where $a' = 1 + d*(a-1)$. (6-7)

"d" is less than 1, so "a' " will be less than "a". Thus, the cache strategy can reduce the value of the remote delay coefficient. From (6-4), we know that this can increase the speedup.

2. Incompletely cached model

For very large problems, not all SDBs can be cached by VM. Some may exist in its "main memory". When an SI refers to an SDB which is not in the local memory of the node (i.e., it results in a VM cache miss), the data block must be loaded. This involves a slow I/O operation. Assume that the average CPU time to load an SDB in this case is $t_{e\text{-}load}$ and the rate of cache misses is "b". Then, the average time to execute an SI is estimated as:

$$t'_{avg}(n) = t_{avg}(n) + b * t_{e\text{-}load} \qquad (6\text{-}8)$$

Since the rate of cache misses depends on the cache size of the entire system, "b" varies with the number of nodes in the cluster (i.e., b(n)). With an increase in the number of nodes in the cluster, the total size of the main memory will increase. Thus, the cache space for SDBs increases. The rate of cache misses "b" will then be reduced. If we define "c" as $t_{e\text{-}load}/t_{avg}(n)$, then the total execution time of the program is:

$$T'_n = (t'_{avg}(n) * N_{SI}) / n = (1 + c*b(n)) * (t_{avg}(n) * N_{SI}) / n \qquad (6\text{-}9)$$

Similarly to "a", we can assume that "c" is a constant. In this case, the speedup of computing with n nodes compared to sequential implementation is:

$$\begin{aligned} S'(n) &= T'_1 / T'_n = S(n)*(1 + c*b(1))/(1+c*b(n)) \\ &= [n^2/((n-1)*a+1)]*(1 + c*b(1))/(1+c*b(n)) \qquad (6\text{-}10) \end{aligned}$$

When $n \rightarrow \infty$, $b(n) \rightarrow 0$. So, $S'(n) \rightarrow n * (1+c*b(1)) / a$ (6-11)

From (6-4) and (6-11), we can find out that the remote delay coefficient "a" reduces the value of the speedup; the speedup decreases linearly with increases in "a". Therefore, for large systems we need good techniques for caching SDBs or SIs with large execution times. This reduces the idle time of PCs for better performance. This is the objective of our dynamic load balancing approach.

6.3 Further Comparison with HPA

The intrinsic need for multiple scans is the large data volume rather than the partitioning of the data. In general, independently of the chosen compression no system can cache all the data for a large database. Not only transaction data cannot be fully cached, but also candidate patterns may not be fully cached in the main memory of member PCs. If both the transaction data and candidate patterns cannot be fully cached, then multiple scans are inevitable. On the other hand, if either of them can be fully cached, then multiple scans can be avoided. These situations are very similar to join operations in relational databases. When multiple scans are inevitable for a large problem, then the performance is affected by the pattern used to access data.

SPM partitions candidate patterns differently from HPA [17] or other algorithms. HPA produces a fixed number of partitions for candidate patterns. The number of partitions is the same as the number of PCs. Each PC holds a partition that may be a monolithic structure. In SPM, all candidate patterns are partitioned into many SDBs. The number of SDBs for candidate patterns may be much larger than the number of PCs. Each PC may host or cache more than one SDB. The "partitions" in the PC include all SDBs processed by it. When the partition is larger than the node's capacity, further decomposition of the partition for that PC is inevitable. SPM provides a straightforward way to do so. It just simply puts some SDBs in the central file system. Decomposition of a monolithic structure used by HPA will be more difficult. SPM provides a way to access data locally. When partitioning candidate patterns, each node needs to insert candidate patterns that belong to it into its data structure. The mode of accessing data is irregular. If the data structure is not fully cached, repeatedly loading pages cannot be avoided. But SPM groups such operations in an SI. The implementation of an SI targets exclusively one SDB. Thus, swap loading can be avoided. The number of candidate patterns can dramatically change among rounds. The assignment of an SI in SPM involves only a pair of candidate and transaction SDBs, instead of comparing an SDB against the entire database. When the number of candidate patterns is reduced, the SF to count the support is automatically reduced to a more efficient CD algorithm. Actually, when the number of candidate patterns is not very small, the SF may work in a mixture of the CD and DD modes. In contrast, HPA does not display this flexibility.

7. Conclusions

We proposed the SPM parallel programming model for PC clusters. It facilitates code portability and dynamic load balancing. For the purpose of illustration, we used our model to implement the Apriori algorithm for mining association rules on PC clusters. A complete set of super-instructions for this application was developed. Super-programs based on this model can efficiently execute on PC clusters. Following this model, a system can efficiently balance the workload among computer nodes. When the super-program has enough parallelism, the member nodes have a little chance to be idle. Minimizing the idle time of nodes in scalable PC clusters should be the objective for good performance according to our underlying theory. The scalable behavior of our approach was observed in our experiments. We will apply SPM to other application areas as well.

References

[1] M. Oguchi, T. Shintani, T. Tamura, and M. Kitsuregawa, "Optimizing Protocol Parameters to Large Scale PC Cluster and Evaluation of its Effectiveness with Parallel Data Mining," *Proc. 7th Int. Symp. High Perf. Distri. Comp.,* p34 –41, Jul 1998.

[2] T. Hiroyas, M. Miki, and Y. Tanimura, "The Differences of Parallel Efficiency Between the Two Models of Parallel Genetic Algorithms on PC Cluster Systems," *Proc. High Perf. Comp. Asia-Pacific Region,* Vol. 2, p945 -948, 2000.

[3] T. Fahringer and A. Jugravu, "JavaSymphony: New Directives to Control and Synchronize Locality, Parallelism, and Load Balancing for Cluster and GRID-computing" *Proc. 2002 joint ACM-ISCOPE conf. Java Grande*, Nov. 2002.

[4] T.G. Mattson, "High Performance Computing at Intel: the OSCAR Software Solution Stack for Cluster Computing" *Proc. First IEEE/ACM Intern. Symp.* Cluster *Comp. Grid*, 2001, p22 –25, 2001.

[5] http://www.netlib.org/blas/.

[6] R. Agrawal and R. Srikant, **“**Mining Association Rules Between Sets of Items in Large Databases ,” *Proc. ACM SIGMOD*, p207-216 May 1993.

[7] R. Agrawal, T. Imielinski, and A. Swami, “Fast Algorithms for Mining Association Rules in Large Databases,” *Proc. 20th Very Large Database Conf.,* Sept. 1994.

[8] R. Agrawal and J. Shafer, “Parallel Mining of Association Rules*,” IEEE Trans. Knowl. Data Eng.*, Vol. 8, No. 6, p962-969, Dec. 1996.

[9] M. J. Zaki, M. Ogihara, S. Parthasarathy, and W. Li “Parallel Data Mining for Association Rules on Shared-Memory Multi-Processors,” *Proc. ACM/IEEE Conf. Supercomp.,* July 1996.

[10] R. Agrawal, C. Aggarwal, and V.V.V. Prasad, “A Tree Projection Algorithm for Generation of Frequent Itemsets,” *J. Parall. Distr. Comp.*, Vol 61, No. 3, p350-371, 2001.

[11] S. Brin, R. Motwani, J. Ullman, and S. Tsur, “Dynamic Itemset Counting and Implication Rules for Market Basket Data ,” *Proc. ACM SIGMOD*, p255-264, May 1997.

[12] J. Han, J. Pei, and Y.Yin, “Mining Frequent Patterns Without Candidate Generation ,” *Proc. ACM SIGMOD*, p1-12, May 2000.

[13] D. Lin and Z.M. Kedem, “Pincer_Serch: An Efficient Algorithm for Discovering the Maximum Frequent Set,” *IEEE Trans. Knowl. Data Eng.*, Vol.14, No.3,p553-566, Dec.2002.

[14] E. Han, G. Karypis, and V. Kumar, “Scalable Parallel Data Mining for Association Rules,” *Proc. ACM SIGMOD,* p277-88, May 1997.

[15] D.W. Cheung, S.D. Lee and Y. Xiao, “Effect of Data Skewness and Workload Balance in Parallel Data Mining,” *IEEE Trans. Knowl. Data Eng.*, Vol. 14, No. 3, p498-514, Dec. 2002.

[16] T. Shintani and M. Kitsuregawa, “Parallel Mining Algorithms for Generalized Association Rules with Classification Hierarchy,” *Proc. ACM SIGMOD*, p25-36, 1998.

[17] T. Shintani and M. Kitsuregawa, “Hash Based Parallel Mining Algorithms for Mining Association Rules,” *Proc. IEEE 4th Conf Paral. Distr. Info. Systems*, p19-30, Dec 1996.

[18] D.W. Cheung, K. Hu, and S. Xia, “Asynchronous Parallel Algorithm for Mining Association Rules on a Shared-Memory Multi-Processors,” *Proc. 10th Annual ACM Symp. Paral. Alg. Archit.,* p279-288, 1998.

[19] J. Han and Y. Fu, “ Discover of Multiple-Level Association Rules from Large Database,” *Proc. 21th Very Large Database Conf.*, Sept. 1995.

[20] http://www.almaden.ibm.com/cs/quest/syndata.html

[21] D.Jin and S.G. Ziavras, "A Super-Programming Technique for Large Sparse Matrix Multiplication on PC Cluster," *Worksh. Hardware/Software Support High Perf. Sci. Eng. Comp.*, New Orleans, Sept. 2003.

Biographies

Dejiang Jin received the B.Sc. in Chemical Physics from the University of Science and Technology of China, Hefei, China, in 1986 and the M.Sc. in Materials Science from Wuhan University of Technology, China, in 1991. From 1991 to 1995 he worked in the industry in the development of new materials. From 1995 to 1997 he carried out research related to smart materials in the Advanced Materials Center of Wuhan University of Technology. From 1997 to 1999 he worked in the area of environmental processes as a visiting researcher in the Civil Engineering Department at New Jersey Institute of Technology (NJIT). He also received an M.S. degree in Computer Science from NJIT in 2000. He is currently a Ph.D. student in Computer Engineering at NJIT.

Dr. **Sotirios G. Ziavras** received the Diploma in EE from the National Technical University of Athens, Greece, in 1984, the M.Sc. in Computer Engineering from Ohio University in 1985, and the Ph.D. degree in Computer Science from George Washington University in 1990. He was a Distinguished Graduate Teaching Assistant at GWU. He was also with the Center for Automation Research at the University of Maryland, College Park, from 1988 to 1989. He was a visiting Assistant Professor at George Mason University in Spring 1990. He joined in Fall 1990 the ECE Department at NJIT as an Assistant Professor. He was promoted to Associate Professor and then to Professor in 1995 and 2001, respectively. He is currently the Associate Chair for Graduate Studies in the ECE Department at NJIT. He also has a joint appointment in the Computer Science Department. He is an Associate Editor of the Pattern Recognition journal. He is an author/co-author of about 100 research and technical papers. He is listed, among others, in Who's Who in Science and Engineering, Who's Who in America, Who's Who in the World, and Who's Who in the East. His main research interests are conventional and unconventional processor designs, field-programmable gate arrays, embedded computing systems, parallel and distributed computer architectures and algorithms, network router design, and computer architecture design.