Load Balancing on PC Clusters with the Super-Programming Model *

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

ABSTRACT

Recent work in high-performance computing has shifted attention to PC clusters. For PC-clusters, member nodes are independent computers connected by general-purpose networks. The latency of data communications is long and load balancing among the nodes becomes a critical issue. We introduce a new model for program development on PC clusters, namely the *Super-Programming Model (SPM)* to address this issue. In SPM, PC clusters are modeled as a single virtual machine with PCs as the processing units. The workload is modeled as a collection of *Super-Instructions* (SIs). Each SI can achieve a limited workload. Application programs are coded using SIs. SIs are dynamically assigned to available PCs at run time. For limited workload, no SI overloads any PC. Therefore, dynamic load balancing becomes an easier task. We apply SPM to mining association rules. Our experiments show that under normal conditions the workload is balanced very well. A performance model is also developed to describe the scalable behavior of SPM.

1. Introduction

High-performance computing has recently shifted its attention to PC clusters containing commercial off-the-shelf (COTS) nodes, for cost-effective parallel computing [1-3]. This trend makes high-performance computing much less expensive and more accessible. These systems are suitable for large-scale problems, such as data mining of very large databases [1]. Each node of a PC cluster is an independent computer running a general-purpose operating system. A general-purpose interconnection network, that most often is an Ethernet-based, connects these nodes together. Data communication among the PCs is controlled by application layer software rather than by lower-level system software or hardware. The latency of data communications on a PC cluster is usually longer than on parallel processing systems that contain specialized hardware for communication networks. Therefore, programming models developed for the latter are not suitable for programming PC clusters. For PC clusters, it is more difficult to exploit low-level or fine-grain parallelism existing in programs. It is more appropriate to adopt coarse- or medium-grain programming. This reduces the adverse effect of long communication delays. For PC clusters, load balancing among member computers becomes a critical issue for high performance. It tries to "appropriately" assign tasks among processing nodes so that minimize the idle time of the processing nodes while other nodes are busy. Only if the workload on these member computers is well balanced, we will be able to achieve high performance. Otherwise, some computer nodes will be idle for significant periods of time during the computation and the overall efficiency of the system will diminish.

In existing programming models, the way to decompose applications is normally function-oriented. Applications are decomposed into function units. To reuse code in a given application domain area, these units are implemented as library functions. For example, BLAS (Basic Linear Algebra Subprograms) has "building block" routines for performing basic vector and matrix operations [4]. They are commonly used in the development of high quality linear algebra software. Each subprogram completes a single operation, such as matrix-matrix multiplication, no matter how large the matrices are. In this fashion, the workload of each "block" is various, it is very difficult to balance.

For this reason, we introduce the *Super-Programming Model (SPM)* for cluster computing. The workload is modeled as a collection of *Super-Instructions* (SIs), that have limited atomic workload. Application programs are modeled as *Super-Programs (SPs)* coded with SIs. At run time, SIs are dynamically assigned to available PCs. The maximum execution time for each SI is well estimated and adjusted with parameters. SPM makes the workload easier to balance among the PC nodes. If the degree of

* This work was supported in part by the U.S. Department of Energy under grant ER63384.

parallelism in the super-program is much larger than the number of nodes in the cluster, then nodes have little chance to be idle. The workload will be balanced very well.

SPM can be adopted for any parallel application. To effectively support application portability among different computing platforms and to also balance the workload in parallel systems, we suggest that an effective instruction set architecture (ISA) be developed for each application domain. For a particular application domain, it is not difficult to determine a set of frequently used operations. These operations then become part of the chosen ISA. SPs utilize these SIs in the coding process. Then, as long as an efficient implementation exists for each of these SIs on given computing platforms, code portability is guaranteed and good load balancing becomes more feasible by focusing on scheduling SIs. Ideally, the chosen ISA should be orthogonal, containing as few SIs as possible that are also adequate to develop any program in the corresponding application domain with the smallest possible number of general-purpose instructions. In this paper, we apply the SPM model to a data mining problem, in order to prove that it can address the load balancing problem very well.

2. Relevant Research

2.1 The Mining Association Rules

Mining association rules is a typical data mining problem. It can be modeled as follows: Let $I = \{ a_1, a_2, a_3, \ldots, a_m \}$ be a *set of items* and $DB = \langle T_1, T_2, T_3, \ldots, T_n \rangle$ be a *transactions* database with items in I. A *pattern* is a set of items in I. The term *itemset* is used interchangeably with the term *set of items* or *pattern*. The *transaction* represents an itemset that occurs in a database. 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 the database DB containing A. An *association rule* is an association relationship between a pair of patterns expressed in the form R: $X \rightarrow_{s,\alpha} Y$, which means X implies Y, where X and Y are exclusive patterns ($X \cap Y = \varnothing$) of I. X and Y are called the *pre-pattern* and *post-pattern* of rule R, respectively. s and α are the support and confidence of the rule R, respectively. The *support s of rule* R is defined as the support of the pattern obtained by joining X and Y ($s = s(X \cup Y)$). Finally, the *confidence a of rule* R is defined as $s(X \cup Y)/ s(X)$. Given a transactions database DB, a minimum support threshold s_{min} and a minimum confidence threshold α_{min}, the problem of finding the complete set of association rules with support and confidence no less than these support and confidence thresholds, respectively, is called the *association rules mining problem.* 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}. Also, given a transaction 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.*

Techniques for discovering association rules have been studied extensively [5-7, 9, 11]. Many approaches transform the association rules mining problem to the frequent patterns mining problem. The most popular one is the Apriori algorithm [6]. Many relevant studies adopt an Apriori-like approach [10,13]. The Apriori algorithm is based on the following property: if any k-length pattern is not a frequent pattern in the database, then any (k+1)-length pattern that includes this pattern can never be a frequent pattern in the database. Using this property, any verified short frequent patterns can help in screening longer candidate patterns. The algorithm works as follows:

```
I' = {x | x ∈ I and x is a frequent item}
// Find all frequent items by scanning DB
P1= { 1-length patterns {x} | x ∈ I'}
For (k = 2; Pk-1 ≠ ∅ ; k++) do begin
  Ck = apriori_gen(Pk-1)
  For any pattern c ∈Ck  {c.count = 0}
  For all transactions T ∈ DB{
    For any pattern c ∈Ck { if (c ⊆ T)  c.count++}
  }
  Pk  = {c | c ∈Ck, c.count  ≥  smin }
End
Answer = È Pk
```

Initially, P_1 gets all frequent patterns with a single item. After that, iteratively the algorithm calls the function "apriori_gen" that generates a complete set C_k of k-length candidate patterns; then, their support is counted by scanning transactions containing these candidate patterns; the set P_k of all k-length patterns is generated by pruning C_k to eliminate infrequent patterns. Once P_k is empty, the iteration is terminated. The union of variable length frequent patterns, $\cup P_k$, forms the complete set of frequent patterns in which association rules can be identified. Each transaction is checked to see if it supports some candidate patterns. To speed up this checking operation, a hash tree of candidate patterns is used.

2.2 Parallel Algorithms and Load Balancing

Several efforts have focused on the development of algorithms for data mining on parallel platforms [7, 8, 11-14]. Several techniques have been developed for dynamic load balancing using some form of load estimate [8, 12]. In previous related research, all consideration of load balancing is based on an estimation of the computation workload. For example, the workload of the join operation was estimated in [8] based on the size of equivalence

classes. To count the support of candidate patterns, it was also suggested to estimate the related workloads. In [11], static load balancing was embedded in the data partition algorithm.

Many studies have proved that computing the counts of candidate patterns is the most computationally expensive step in the algorithm. The only way to compute these counts is to scan the entire transactions database. Most algorithms focus on computing the support of patterns in parallel [7, 8, 11]. These algorithms can be classified into two basic types based on what types of data are partitioned. They are either *count distribution (CD) algorithms* or *data distribution (DD) algorithms*. In a *CD algorithm* [2], the entire candidate set is copied into all the nodes. Transaction data are partitioned and each node is assigned an exclusive partition. The allocation of workload is controlled by partition of transaction. A *DD algorithm* partitions the set of candidate patterns for exclusive assignment to processing nodes [2]. This partitioning is 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. The allocation of workload is controlled by partition of candidate. But both partition of candidates and partition of transactions are well workload estimation of the count.

Thus, load balancing based on load estimation cannot be perfect. To conclude, past approaches to load balancing for mining association rules in databases did not demonstrate the versatility of the dynamic load balancing technique that we propose in this paper. Our results also in Sections 4 and 5 support our claim.

2.3 Parallel Implementation of Data Mining

Many researchers have implemented relevant algorithms and evaluated their performance on parallel machines or supercomputers, such as the IBM SP2 [7], SGI Power Challenge [8], and Cray T3D [11]. Some researchers experimented on both parallel machines and PC cluster [1, 13]; But they applied identical algorithms. They did not consider adjusting the algorithm for the chosen computing platform; for example, to reduce the effect of long delay on PC clusters, they tried to improve the PC-interconnection network.

3. A Super-Programming Model for Mining Association Rules

3.1 The Super-Programming Model

In the super-programming model (SPM), the system is modeled as a single virtual machine with PCs as processing units. The workload is modeled as a collection of SIs. Like instructions for processors, SIs are expected to complete a task with limited workload that can make the execution time quite predictable. i.e. SPM is workload-oriented. An example of such an SI is "compute the supports for a set of candidate patterns, where the number of patterns is no more than k, against a block of transaction data"; k is a design parameter. SIs model atomic workload units The maximum execution time for each SI is well estimated. It is determined by design parameters. Designers can choose the parameters so that all SIs have similar maximum execution time. Any large task is implemented by executing more than one SI. Application programs are modeled as *Super-Programs (SPs)*. They are composed of SIs.

A runtime environment supports the execution of SPs. At run time, each SI is dynamically assigned to a PC to execute if and only if the PC has resource. Each SI can only be executed on a single node. SIs are executed parallel if they do not depend on each other. Extending the functional unit to handle multiple SIs makes the high-level parallelism not only to be determined by the algorithm but also by the ISA designer. Increasing the degree of high-level parallelism makes easier the task of balancing the workload.

3.2 Design Issues for the Super-Instruction Set

There exist three main issues :

1) Portability of SIs. SIs are implemented by software routines that can be executed on PC nodes. Since SIs are dynamically assigned to PCs, they could be executed on any PC. This requires implementing SIs that are portable throughout PCs in the cluster. For a heterogeneous system, it becomes a major task.

2) Completeness and orthogonality of the SI set. The SI set creates an abstract layer. It should encapsulate the underlying support system. An application should be described completely by using these SIs. Thus, the SI set should provide all basic operations to support such abstractions. Considering the storage capacity and the programming capability of general-purpose computers, the number of SIs can be unlimited. Also SIs can be as robust as needed. There is no real need to improve any resource in order to provide a larger instruction set. Thus, the SI set is open to expansion to match the application domain's requirements, as needed. In other words, the SI set is completely application dependent. To enhance software component reuse, it heavily depends on the application domain. Another issue is the orthogonality of the SI set. That is, SIs should not have any functionality overlap in terms of the types of major tasks that they carry out. This way, the SI set will have reasonable size without any redundancy for better program maintenance, ease of algorithm development, efficiency and good portability.

3) High-level name space for data references. Member PCs are completely independent processing units.

They may have different independent logical space. SIs are dynamically assigned to a PC. Thus, SP running on a cluster needs a global logical space. SIs should only reference data with names (or Ids) in this space rather than reference their operands with local addresses on the underlying procedures implemented SIs.. The runtime environment will map the global Object to local space.

3.4 Data Blocks for Mining Association Rules

The operands for SIs are blocks of data. Such a data block is called a *super-data block* (SDB). SDBs are primary entities for high-level super-programming; they are used as build-in data in ordinary programming. High-level super-programs build their data structures using SDBs. SIs manipulate these SDBs. Each SDB has its own *global ID*, as data in an ordinary program have their own address. The data included in an SDB can be loaded/cached/stored at any node by runtime support systems. The data blocks have limited maximum size. Thus the workload of SIs is limited. This way, assigning a significantly large task to a single node is avoided. For mining association rules, we have designed a set of super-data blocks as shown in Table 1.

Table 1 Summary of Super-data blocks types

Name	Content
BlockOfItems	A list of distinct items covering a continuous, exclusive partition
BlockOfTransaction	A set of transaction data
BlockOfJoinResult	A list of candidate patterns without checking for frequent sub-patterns
BlockOfCandidates	A list of candidate patterns. All of their sub-patterns are frequent patterns
BlockOfFrequentSet	A list of frequent patterns with the same length
BlockOfRules	A set of alreadymined association rules

3.5 ST Set for Mining Association Rules

We have designed an SI set for mining association rules in large transaction databases. A summary is shown in Table 2.

Table 2 Summary of a super-instruction set for mining association rules

SI name	Parameters	Function
LoadDataBlock	Reference to the data source; an SDB of transactions; maximum size of the SDB	Gets a block of data residing outside of the system
CountItemSupport	ID of the extracted transaction block; ID of the map model	Extracts all distinct items and counts the support of items appearing in a block of raw transaction data
ShrinkItemBlock	SDB of items; the threshold of support	Prunes items in an SDB of items
GetFrequentItems Block	A list of SDBs of items; SDB of frequent items; generated SDB of 1-length frequent patterns; mapping object	Creates an SDB of frequent items and an SDB of 1-frequentItemSet
ShrinkTransaction Block	ID of the original SDB of transactions; ID of the result SDB of transaction;	Shrinks a block of transactions
MergeTransaction Block	A list of IDs of pruned SDBs of transactions	Merges a list of pruned SDB blocks of transactions into one SDB
GenCandidates Block	[An frequent pattern] or [A list of frequent patterns]; an SDB of frequent patterns; a global mapping object; the generated SDB	Generates an SDB of candidate patterns.
FilterCandidates	An SDB of candidates; an SDB of frequent patterns; the global mapping object	Screens candidate patterns in a SDB by comparing their sub-patterns with frequent patterns stored in another SDB .
CountCandidates Block	ID of an SDB of candidate patterns; ID of an SDB of transactions	Counts the partial support of the candidate patterns in an SDB of candidates
PruneCandidates Block	An SDB of candidate patterns; the threshold of support	Prunes an SDB of candidates
GetFrequentSet Block	List of SDBs of candidate patterns; a global mapping object	Generates an SDB of frequent patterns from a list of SDBs of counted candidates
CheckConfidence InBlock	an SDB containing post-patterns; an SDB of pre-pattern; an SDB of rules; mapping object	Extracts rules from a pair of SDBs of frequent patterns
StoreResult	SDB of rules	Stores an SDB of rules in external storage

3.6 Super-Program for Mining Association Rules

The high level description of the Super-Program for mining association rules is:

P_1 = initial(DS, s_{min})
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 = $\cup R_k$

1. Initial: the first super-function
   While (there is more data){
     LoadDataBlock (DS, s, $rawBlockId_i$); }
   parallel do { CountItemSupport; }
   parallel do { ShrinkItemBlock; }
   parallel do { GetFrequentItemsBlock;}
   parallel do for all SDBs of transactions {
     sequential do for all SDBs of frequent Items{
       ShrinkTransactionBlock;}
   }
2. "gen_candidate" super-function
   parallel do all SDBs of frequent patterns {
     parallel do all same and following SDBs of
       frequent patterns{
         GenCandidatesBlock ;
         FilterCandidates ;
     }
   }
3. "gen_frequentSet" super-function
   parallel do for all data blocks of k-length candidates
   patterns $candsBlock_m$ {
     parallel do for all blocks in transaction 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. "find_rules" super-function
   parallel do all SDBs of k-length frequent patterns{
     parallel do all SDBs of m-length frequent patterns
     { CheckConfidenceInBlock }
   }

4. Experimental Results

4.1 Experimental Setup

All the experiments were performed on a PC cluster with six nodes. Each dual-processor node is equipped with AMD Athlon processors running at 1.2 GHz. Each node has 1GB of main memory, a 64K Level-1 cache and a 256K L2 cache. All the nodes are connected through a switch to form an Ethernet LAN. 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.

Table 3 Synthetic databases used

set	transactions	average items per transaction	average length of the maximal pattern
A	3K	25	10
B	10K	25	10
C	30K	25	10
D	100K	25	10
E	500K	25	10

We used synthetic databases with sizes ranging from 2MB to 400MB; they were generated using the program provided in [15]. The generated databases are listed in Table 3. 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 4 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 super-program was manually written using the SIs described in Section 3. We implemented a simple runtime environment, which supports our super-programming model. The runtime environment and the SIs were implemented in the Java language. Table 4 shows the values of the parameters used to control the maximum workload within SIs.

A special function was added in the code to collect information at runtime about the utilization of the individual PC nodes. The information includes the total time elapsed, the total time executing sequential code, the total time executing parallel SIs and the total idle time. When no SI is running on a node, the node is considered to be idle. We run repeatedly the same program 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 among all nodes and for multiple runs. The minimum support is 0.6% and the minimum confidence is 40%.

4.2 Results

Table 5 presents a summary of the total program running time, actual execution time and the average idle time of nodes for each mining problem in our experiments. Since the initial part of the super-program loads data sequentially, we separated the calculation of the total execution and parallel computing times. Fig 1 shows the percentage of total idle time (relative to the total execution time) under various conditions. Fig 2 shows the percentage of idle time in computing (relative to the total computation time) under various conditions.

Fig 3 shows the total speedup for the super-program executing on the PC cluster as a function of the total number of PC nodes. Fig 4 shows the speedup in parallel computing as a function of the total number of PC nodes.

Table 5 Summary of execution times and idle times in milliseconds for SPM

Data Set	Number of nodes	1	2	3	4	5	6
A	Average idle time	191	3556	5101	7613	8881	10572
	idle time in computing*	191	1266	3800	6126	6441	7837
	Total time	71170	63209	53217	54884	53295	51285
	Computing time	68757	59901	50775	52406	49483	47183
B	Average idle time	440	5478	8178	9030	11049	11196
	idle time in computing*	440	2716	3864	4333.9	5400	5737
	Total time	146062	103904	97505	85210	88830	86648
	Computing time	139947	96999	89417	77384	80003	78460
C	Average idle time	217	9340	14090	14580	18543.8	17745
	idle time in computing*	217	2617	4397	3977	5714	4820
	Total time	376079	231358	211133	158794	162268	157674
	Computing time	361751	214551	192959	141123	142221	138286
D	Average idle time	306	19484	32092	37448	373045	49967
	idle time in computing*	306	2696	8015.5	3438	8070	13158
	Total time	1234613	701748	515564	547617	377532	431875
	Computing time	1189756	656980	467411	487153	328808	372980
E	Average idle time	1912	85293	134191	144945	182876	196851
	idle time in computing*	1912	53622	78463	32467	90942	91268
	Total time	6883521	4825186	3081771	3263902	2325782	2585250
	Computing time	6652822	4571818	2747404	2664023	1866108	2078452

* excluding the sequential load data stage

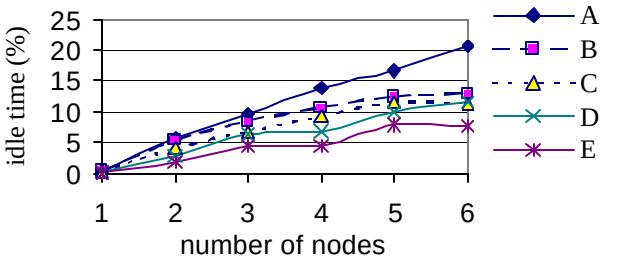

Fig. 1 Average idle time vs the number of nodes.

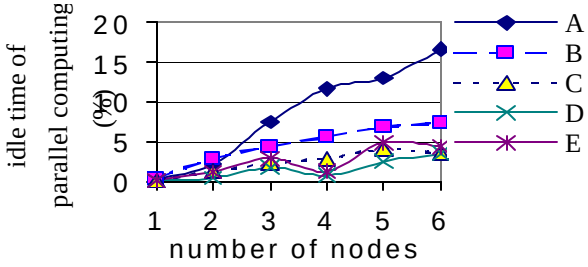

Fig. 2 Average idle time in parallel computing vs the total number of PC nodes

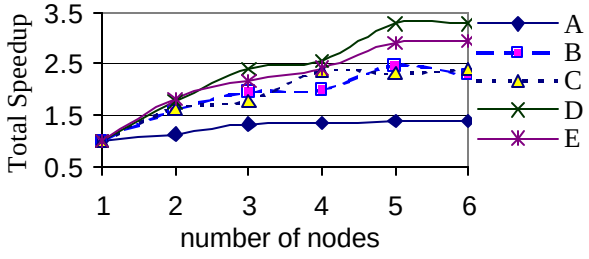

Fig. 3 Total speedup of the super-program vs the total number of PC nodes.

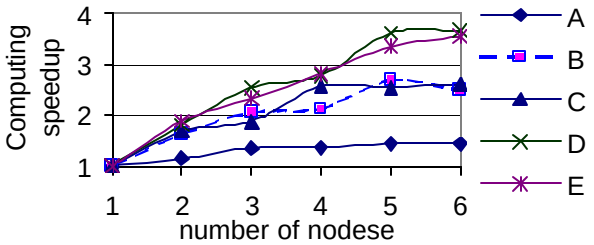

Fig. 4 Speedup of parallel computing vs the total number of PC nodes.

5. Analysis of Results

5.1 Performance of Parallel Computing

From Fig. 3 and 4, we can see that the speedup increases with the number of nodes. However, the rate of increase depends on the size of the mining problem. For the problem A, the speedup with 6 nodes is less than 1.5. For larger problems, the speedup with 6 nodes is larger. When the size of the problem is large enough, such as problem D or E, however, the speedup is stable.

This behavior of the performance improvement as a function of the problem size can be explained. Although most super-functions are intrinsically parallel, we may still encounter the "last task" imbalance problem. This kind of problem appears when the last task for a super-function that ends with synchronization of the nodes, still continues on a computer node while all other nodes are wait for synchronization. For smaller problems, keeping the size of super-data blocks fixed, the number of super-data blocks will be smaller. If the degree of data parallelism is lower, there is more chance to encounter the "last task" imbalance problem. And the chance is larger when the number of nodes is larger. Thus, the speed up of the system is kept low we should choose smaller sizes for super-data blocks to directly reduce the effect of the "last task" imbalance problem. Reducing the size of super-data blocks also increases the parallelism among super-functions. This indirectly reduces the chance for an imbalance to occur.

For large problems, the "last task" effect will be small. For mining a given database with given criteria (i.e., minimum support and minimum confidence), the total number of SIs does not depend on how we schedule them and assign them to PC nodes. Assume that the total number of SIs is N_{SI} and that there are n identical computer nodes in the PC cluster. Assume that these nodes will be idle for a percentage p_{idle} of time and will execute SIs all other times. Then, the total time to solve the problem is estimated by:

$$T = (t_{avg} * N_{SI}) / (n * (1 - p_{idle}))$$

where t_{avg} is the average time to execute an entire SI, including loading/storing SDBs.

Assume all data blocks are stored in the main memory of the nodes and all nodes have identical memory capacity. 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 time to execute an SI when all operands are available in local cache is t_{exe}, the average time to load a local super-data block is $t_{local\text{-}load}$ and the average time to load a super-data block from a remote node is $t_{remote\text{-}load}$. Then, the expected total time to execute an SI that loads local operands is $t_{exe\text{-}local} = t_{exe} + t_{local\text{-}load}$. The average total time to execute an SI that loads remote operands is $t_{exe\text{-}remote} = t_{exe} + t_{remote\text{-}load}$. Finally, the total time to execute an SI is estimated by:

$$t_{avg} = (1/n) * t_{exe\text{-}local} + (n-1)/n * t_{exe\text{-}remotel} \quad (5\text{-}1)$$

We define "a" to be equal to $t_{exe\text{-}remote} / t_{exe\text{-}local}$, 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:

$$T_n = (t_{avg} * N_{SI}) / n$$
$$= ((n-1)*a + 1) * t_{exe\text{-}local} * N_{SI} / n^2 \quad (5\text{-}2)$$

When n = 1, then the super-program is executed sequentially in a single node. The execution time in this case is:

$$T_1 = t_{exe\text{-}local} * N_{SI} \quad (5\text{-}3)$$

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

$$S(n) = T_1 / T_n = n^2 / ((n-1)*a + 1) \quad (5\text{-}4)$$

We have $a = (t_{exe} + t_{remote\text{-}load}) / (t_{exe} + t_{local\text{-}load})$. It varies with $t_{remote\text{-}load}$. $t_{remote\text{-}load}$ depends on the amount of data transmitted in the network, which, in turn, indirectly depends on the number of nodes in the cluster. For a higher system speedup, the communication workload in the network will probably increase within a time unit. By increasing n, we also increase the chance that a super-data block 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, $t_{remote\text{-}load}$ does not vary significantly with n and its effect on "a" is weakened by t_{exe}. So, in our model, we assume that "a" is a constant.

When n ? 8, S(n) = n/a. the speedup for n nodes is n/a and the efficiency of the system is

$$E(n) = S(n) = 1/a \quad (5\text{-}5)$$

Therefore, the well-known iso-efficiency property is satisfied. The efficiency does not depend on the number of nodes and the problem size. The system will increase its speedup linearly when there is a large number of nodes. However, the remote delay coefficient will reduce the value of the speedup.

5.2 Idle Time of Member Nodes (PCs)

From Fig. 1 and 2, we can see that the idle time increases with the number of nodes. The percentage of idle time in parallel 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 parallel computing, we can see that for a small problem (e.g. A) 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%.

Executing high-level super-functions in parallel may further reduce the effect of the "last task" problem, even if there is only a low degree of parallelism in the program. This is because, although each super-function may have an unbalanced workload at the end of its execution, these imbalances may appear at different times. At a given time, a super-function may be lacking parallelism. However, another super-function may still have abundant parallelism at that time. The balanced workload for the latter may hide the imbalance for the former and make the overall system look well balanced.

For mining association rules, in the first few rounds there is usually a very large number of candidate patterns. This makes the functions of generating the candidate patterns and frequent patterns to have 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 super-function that finds rules is very low. When the process reaches the last few rounds, the volume of candidate patterns definitely drops dramatically. This makes the functions of generating the candidate patterns and the frequent patterns to have very small parallelism. However, it is expected 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 super-function that finds rules is then very high. This reverse change between parallel super-functions makes the parallelism of the entire application rather stable. Thus, the percentage of average idle time for nodes is low.

6. Conclusions

In this paper, we have proposed a programming model for parallel computing on PC clusters. Super-programs based on this programming model can efficiently execute on parallel systems or PC clusters. With our model, a system can efficiently schedule tasks, balance the workload and fully utilize the computing capacity of each computer node. When the super-program has enough parallelism, the nodes have a little chance to be idle. For the sake of illustration, a complete set of SIs for mining association rules in parallel on PC clusters was developed. A performance model was also developed to describe the scalable behavior of such a system. The scalable behavior of our approach was observed in our experiments.

References

[1] M. Oguchi, T. Shintani, T. Tamura, M. Kitsuregawa, "Optimizing Protocol Parameters to Large Scale PC Cluster and Evaluation of its Effectiveness with Parallel Data Mining," *High Performance Distributed Computing, Proc. The Seventh Int. Symposium*, p34 –41, Jul 1998.

[2] T. Fahringer, 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. on Java Grande*, p8-17, Nov. 2002.

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

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

[5] R. Agrawal and R. Srikant, "Mining Association Rules Between Sets of Items in Large Databases ," *Proc. ACM (SIGMOD)*, p207-216 May 1993.

[6] R. Agrawal, T. Imielinski, and A. Swami, "Fast Algorithms for Mining Association Rules in Large Databases," *Proc. 20th Very Large Database Conf.*, p487-499 Sept. 1994.

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

[8] 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. Supercomputing*, July 1996.

[9] 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.

[10] D. Lin and Z. M. Kedem, "Pincer_Serch: An Efficient Algorithm for Discovering the Maximum Frequent Set," *IEEE Trans. Knownl. Data Eng.*, Vol.14(3), p553-566, Dec. 2002.

[11] E. Han, G. Karypis, and V. Kumar, "Scalable Parallel Data Mining for Association Rules," *Proc. ACM Special Interest Group on Management of Data (SIGMOD)*, p277-88, May 1997.

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

[13] T. Shintani and M. Kitsuregawa, "Parallel Mining Algorithms for Generalized Association Rules with Classification Hierarchy," *Proc. ACM (SIGMOD)*, p25-36, 1998.

[14] 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.

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