

Facilitating data mining on a network of workstations

Srinivasan Parthasarathy*

Department of Computer Science
University of Rochester,
Rochester, NY-14620
srini@cs.rochester.edu

Ramesh Subramonian

Microcomputer Research Laboratory,
Intel Corporation
2200 Mission College Blvd, MS RN6-35
Santa Clara CA 95052-8119
subramon@gomez.sc.intel.com

August 4, 1999

Abstract

The computationally-intensive nature of many data mining algorithms and the size of the datasets involved has motivated efforts to use parallel computing to produce timely results. A particularly cost effective computing platform for such parallelizations is a network of workstations (NOW). However, there are many problems associated with efficient parallelizations on a NOW, including data transmission over a low bandwidth network, load-balancing, fault-tolerance, interactivity, programming complexity, etc.

To address some of these problems, in this paper, we propose the *programmable, distributed doall*, a generic mechanism, similar to the *doall* primitive on SMPs, which schedules a set of independent tasks on a NOW. It allows incremental reporting of results, which is used to allow the user to monitor and, if necessary, interrupt the operation. Most importantly, it seeks to reduce communication bandwidth requirements by allowing specification of resource requirements of the tasks at the application programming level. This information is used by a “resource-aware” scheduler to reduce communication.

We evaluate the performance of this mechanism using a template application that resembles, both in its computational and data access patterns, several common data mining applications. We also evaluate the primitive on clustering, a common data mining technique.

Keywords: Distributed Algorithms, Data Mining, Network of Workstations, Programming Primitives

*Work described in this article was done while at Intel

1 Introduction

As our ability to collect, store, and distribute huge amounts of data increases with advancing technology, discovering the knowledge hidden in these ever-growing databases has become a pressing problem. This problem is referred to as data-mining, an effort to derive interesting conclusions from large bodies of data automatically. Extracting knowledge from these massive databases is a computationally expensive process, which is amenable to and can benefit from being parallelized. This has prompted work in parallelizing data mining algorithms [SAM96, ZPOL97, CHN⁺96].

Modern-day enterprises usually contain a cluster of shared memory workstations connected by some (intra-enterprise)network. Such a cluster of shared-memory symmetric multi-processors (SMPs) is a cost effective computational resource. It is particularly attractive as it is a zero-cost resource that if properly harnessed can provide a powerful computational platform. Leveraging this enterprise wide resources for data mining poses several problems.

First, the programming model in cluster computing is based on message passing. In terms of programmability this model is more complicated than the intra-workstation shared memory model. Furthermore, parallel programming primitives like *doall* constructs that exist on a shared memory system [Hig93], are not available on a cluster platform. Corresponding primitives for parallelization on an SMP cluster can alleviate this problem.

Second, due to the large datasets involved, distributing the data across a limited bandwidth interconnection network is expensive. Techniques that can reduce the communications costs, by reducing the input data size, are essential for distributed data mining. Ideally, such techniques should compromise on result quality and comprehensiveness as little as possible.

Third, the very nature of the knowledge-discovery process requires the user to be tightly integrated into it. Providing interactivity in the form of monitoring, computational steering and fast response times in an asynchronous, distributed environment is difficult, but useful. In order for this to happen algorithms have to be re-architected in a fashion that permits such desirable features.

1.1 Contributions

In this paper we present a programmable data parallel primitive D-DOALL that addresses each of the above problems in the following way.

- **Usability** Our solution takes the form of the traditional *doall* primitive, a popular way to express data parallelism in shared memory multiprocessors. The runtime system handles executing it across a network of workstations.
- **Limited Bandwidth and Large Datasets** The regularity of many data mining algorithms means that one can reason about the resource (input data) requirements of individual independent tasks. A good scheduling algorithm can take advantage of this information to minimize communication. Our runtime system incorporates a global affine scheduler which does this.
- **Interactivity** Our primitive supports partial (incremental) result reporting in a timely fashion and also permits clients to terminate execution at any point in the computation.

We evaluate the primitive on two commonly used data mining applications: discretization and clustering.

1.2 Organization

In Section 2, we describe the architecture of the distributed data mining system we have implemented to set our primitive in context. In Section 3, we describe the distributed *doall* (D-DOALL) primitive. In Section 4 we highlight alternative scheduling policies for the D-DOALL primitive. In Section 5, we evaluate the relative merits of these policies using a simulation of a template application. In Section 6, we present actual speedups on a NOW for our applications. Finally, in Section 7, we present our conclusions and outline directions for future work.

2 Architecture

In this section we sketch the architecture of a distributed data mining system we have implemented. The design of our system took into account the interactivity and large datasets involved in mining applications. In addition it is often possible to provide succinct descriptions of the input e.g. “partition dataset X into 4 clusters” or “discretize the continuous attributes in dataset Y”. This permitted us to decouple task description from the actual data required by the task and is reflected in our decoupled architecture. This decoupling is important as it potentially enables different tasks to obtain data from multiple sources simultaneously.

Our architecture consists of the following logical components:

- **Client** consisting of the GUI, a task manager that directs the mining process, and a local cache of data/results of prior computations. It is responsible for the interacting with the data mining engine in terms of invoking, guiding and monitoring computations as well as visualization of the results.
- **Compute Servers**, each consisting of a task manager, a compute module, which is the core data mining engine, and a local data cache. All compute servers are indistinguishable in terms of structure and code base.
- **Data Server** consisting of a data distiller and the source database. The data distiller reads data from the database and performs appropriate data compaction transformations before passing it to the compute servers.

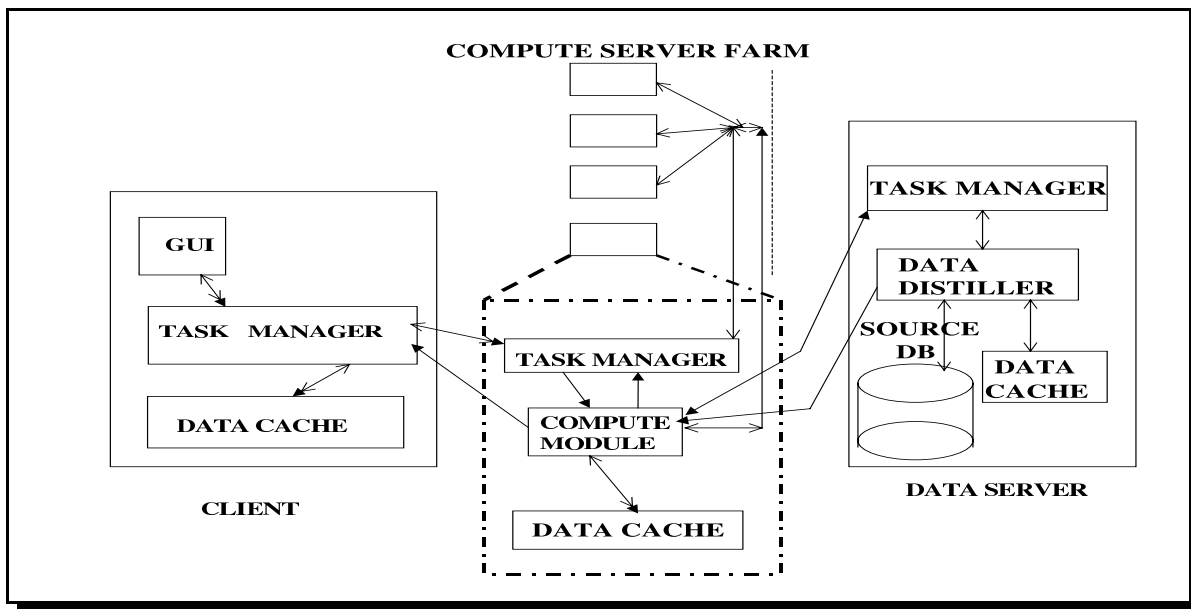


Figure 1: IntelliMiner Architecture

The physical layout of these logical components depends on available resources. In a fully distributed mode the components, i.e. the client, each compute server and the data server are physically separated. When the data sets are relatively small and the client is relatively powerful, all components could be resident on the client. Figure 1 depicts the overall architecture with arrows indicating communication patterns.

2.1 Communication Protocol

In this section, we describe the basic protocol for communicating across any two tiers in our architecture. For exposition, we describe with an example how the communication between a client and compute server transpires.

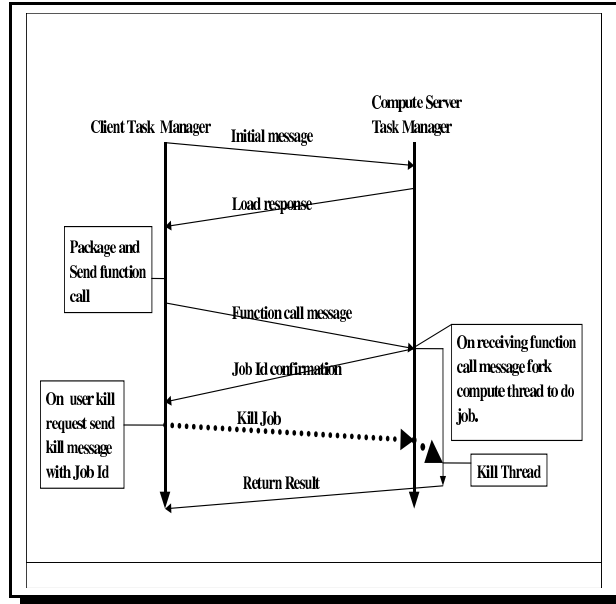


Figure 2: Communication Protocol

Figure 2 depicts a typical communication between a client and a compute server. The client task manager (caller), that needs to execute the remote service first pings the task manager on the compute server to see if it is overloaded. If the server is not overloaded, then the client packages the function call and transmits it to the compute server. On receiving the packed function call, the task manager forks a thread to execute the job, and returns a job identifier to the caller. This permits the caller to stop/kill the compute server job (callee) if and when required. The compute module thread that executes the job unmarshals the function call, invokes the appropriate function, and generates the results. When the function has completed its execution the results are packaged by the callee and returned to the caller.

The communication protocol is implemented on top of C++ sockets. We evaluated using distributed object technology like COM/CORBA but found sockets to be the cheapest (performance wise). We are investigating alternative message passing primitives to improve communication overheads (www.via.org) and to permit portability across different architectures [Mes94].

3 Distributed DOALL primitive

A *doall* loop is one in which the loop iterations are independent [Wol96], i.e. , where there are no conflicts between iterations. In other words where an element that is assigned is used on that iteration only then the loop is referred to as a *doall* loop. In such a case , executing the iterations sequentially or in parallel in any order is legal, since the result does not depend on the order. It is a simple mechanism that is often adequate to express parallelism. Loops that are not strictly *doall* loops (such as a loop that sums the elements of an array) can often be cast as *doall* loops by performing partial summations in parallel and then sequentially combining the results of the partial sums. A sample *doall* loop is shown below.

```
for ( i = 0; i < N; i++ ) { A[i] = B[i]; }
```

The *doall* primitive we adopt is loosely based on on the **parallel for** defined in OpenMP (www.openmp.org), an industry-wide initiative to unify parallel programming constructs on SMPs, and is similar in flavor to the **forall** construct in High Performance Fortran [Hig93] (although **forall** is more of a compile-time construct, whereas the *doall* in figure 3 is a pure runtime construct). Despite the restrictiveness of *doall*, its adherence to an industry standard, ease of use and portability make it an attractive choice on the SMP platform.

Figure 3 presents the invocation of the *doall* construct on our sample *doall* loop.

```

doall((void (*)(void))body, iters, ..., N, A, B); /* do in parallel */
void body1(int *start, int *iters, int A[], int B[])
{
    int i;
    for ( i = *start; i < start + *iters; i++ ) {
        A[i] = B[i];
    }
}

```

Figure 3: Sample parallelization using doall

```

Distributed-DOALL(
    int T; /* number of independent tasks */
    void **inputsList; /* [T][] */
    void **outputsList; /* [T][] */
    FuncPtr body; /* body of doall loop */
);

```

Figure 4: Basic Distributed DOALL Interface

The run time system is responsible for scheduling different iterations of the loop body on different processors by assigning appropriate values to `start` and `iters` (a variable to chunk several consecutive iterations on the same processor) and invoking the function body.

Providing a variant of the *doall* mechanism on a cluster of SMPs would reduce the complexity of distributed programming. However the implicit assumption of shared memory in the *doall* primitive is an obstacle to a direct transformation into the distributed, memory model. To overcome this, we designed the distributed *doall* primitive, the D-DOALL. The D-DOALL is invoked as follows. The application specifies a set of tasks to be executed in parallel. For each task it marshals the function to be executed and the input arguments to the function as a linear array of bytes. The result of each task is a linear array of bytes, the interpretation of which is the responsibility of the calling routine. Figure 4 specifies the calling mechanism. Note that the input/output semantics of the D-DOALL prevent passing of parameters by value.

The main thread that invokes the D-DOALL first identifies the available compute servers. This is identical to the first part of the communication protocol described in Section 2.1. It then spawns a *local doall thread* for each available remote compute server. The main thread then blocks until all tasks are done. Each *local doall thread* selects a task from the task queue and sends that task to its associated compute server and blocks until it receives the output. On receiving the result (output) it selects another task and repeats the process. When all tasks are done the *local doall threads* terminate and the main thread that invoked the D-DOALL is resumed.

Figure 4 shows the basic structure of the D-DOALL interface. We now describe how additional features such scheduling and support for interactivity can be added to this basis.

3.1 Supporting Interactivity

We extend the D-DOALL mechanism to support interactivity in two ways (Figure 5). One is by providing a *terminate* mechanism. The other is by providing a mechanism which allows the calling thread to be made aware of the completion of a particular

task.

```
typedef void ( *FuncPtr )( void *, int);
DistributedDOALL(
    int T; /* number of independent tasks */
    void **inputsList; /* [T][] */
    void **outputsList; /* [T][] */
    FuncPtr body; /* body of doall loop */
    int &interrupt_flag; /* global interrupt flag */
    FuncPtr fp /* functional processing */);
```

Figure 5: Interactive Resource-aware D-DOALL

Termination is handled as follows. We permit the application developer to pass as a parameter a location in global memory (`interrupt_flag`) that initially contains the boolean `False`. If the D-DOALL is to be terminated by the client the `interrupt_flag` is set to the value `True`. The main thread spins on this flag while waiting for all its tasks to complete. When this flag is set to `True`, the main thread simply cancels all tasks which have not been scheduled and waits for any outstanding scheduled tasks to complete. Once these outstanding tasks complete, the main thread exits from the D-DOALL call.

We permit the application developer to pass as a parameter to the D-DOALL, a function pointer `fp` that takes as its argument a `void*` pointer. Essentially we modified the way each *local doall thread* behaves on receiving the output from a compute server. After scheduling the next job it executes the function pointed to by `fp` with the result output as the parameter. Within this function the *local doall thread* can notify the GUI of incremental progress made in an application specific manner.

4 Scheduling

4.1 Previous Work

Given a set of T tasks and a set of P processors, the scheduling problem can be informally stated as deciding which processor executes which task when. One solution is to statically partition the work among the processors (Static Scheduling) at compile time [Wol96, Pol88, LP93]. Such schemes have been implemented on NOWs [CLZ95, CR92, Gea94]. Another solution is to hand out tasks one at a time to a free processor requesting work (Simple (Dynamic) Scheduling) [Pol88, ML94]. More complicated dynamic strategies have also been proposed (in some cases application specific) [LK87, NS93, Bea96].

The scheduling policy for the D-DOALL specified in Figure 4 is the simple, dynamic policy. We currently support a simple dynamic scheduling policy. Tasks are scheduled in the order specified, thus allowing the user to have some control on the order in which partial results will be reported. This feature is used in Section 6.2.1.

4.2 Desired Scheduling Properties

A good scheduling algorithm should be able to satisfy certain basic requirements such as fault tolerance and load balancing. In addition, a desirable property is “resource-aware” scheduling, also called “affinity-scheduling” [ML94]. A “resource-aware” scheduling algorithm takes into account the resources (in terms of data or partial results) that a processor possesses in its local cache when it determines what task to assign to that processor. Using matrix-multiply ($C \leftarrow A \times B$) as an example, the creation of $C_{i,j}$ is ideally assigned to the processor that possesses row i of A and column j of B .

This feature is especially desirable in data-intensive applications such as data mining. In such applications, it is often the case that the format in which data is stored for report generation is not the most appropriate format for mining the data. In such cases the data has to be transformed into a form which is acceptable to the application. Furthermore, several applications like

Discretization [FI93, SVC97], Clustering [SGIF97], and Similarity Analysis [AF93], may accept a distilled/compressed form of the output. Reduction of the output data prior to transmission reduces the bandwidth requirement of the interconnection network and storage requirements at the compute server. The need to minimize communication is especially important in a distributed environment where communication bandwidth is relatively small and one seeks to reduce demands on the database server. The overall cost of obtaining a resource needs to factor in the cost of such transformations along with the cost of communicating the transformed or distilled data. Therefore, in such applications, the overall cost of obtaining a resource can be very expensive.

A good scheduling algorithm, which analyzes resource requirements and schedules tasks in resource-aware manner, can prove to be beneficial to such applications. Clearly this analysis will take time. However, if the time spent scheduling is small when compared with cost of obtaining such resources, then resource-aware scheduling should outperform traditional schedulers for such applications. In the ensuing sections we present such resource-aware algorithms and examine the performance of the scheduling algorithms both qualitatively and quantitatively.

4.3 Application Programming Interface

```
Distributed-DOALL(
    int T; /* number of independent tasks */
    void **inputsList; /* [T][] */
    void **outputsList; /* [T][] */
    FuncPtr body /* loop body */
    int &interrupt_flag; /* global interrupt flag */
    FuncPtr fp /* functional processing */
    int R; /* number of resources */
    int *num_resources_per_task; /* [T] */
    int **resources_per_task; /* [T][] */
);
```

Figure 6: API for resource-aware D-DOALL

In this section, we describe how the D-DOALL API changes from that of Figure 5 to that of Figure 6 in order to become “resource-aware”. The application needs to inform the scheduling algorithm about the resource requirements of each task as follows. The user specifies (i) a list of R resources, numbered $1 \dots R$, (ii) a list of T tasks, numbered $1 \dots T$, and (iii) for each task, a list of resources needed to perform that task.

We explain the above specification using matrix multiply as an example. Consider $C[m][p] \leftarrow A[m][n] \times B[n][p]$. There are $m + p$ resources, the m rows of A and the p columns of B . There are $m \times p$ tasks corresponding to the creation of each element of C . Task $(i \times m) + p$ is the creation of $C_{i,j}$ and requires resources i and $m + j$ where resource i is the i th row of A and resource $m + j$ is the j th column of B .

In Section 4.4, we will show how specifying the resource requirements of each task, enables the scheduler to make educated guesses as to the resources possessed by a processor and hence, the suitability of assigning a specific task to a specific processor. We seek to answer the question: “can we do a better job of scheduling than the algorithms of Section 4.1 without imposing an onerous burden either on the programmer or on the scheduler?”

4.4 Algorithms

In Sections 4.4.1, 4.4.2 and 4.5, we present two different “resource-aware” scheduling algorithms, and a discussion of their respective advantages and disadvantages. In Section 5, we present an empirical evaluation of their performance.

The fundamental idea behind resource-aware scheduling is to use the knowledge of the resources possessed by a processor to determine which task should be assigned to it. We hope to be able to have an approximate idea of the resources that a processor possesses based on tasks executed by it in the past. Knowing the resources a processor possesses allows us to define a metric to evaluate the relative suitability of different tasks. In Section 4.4.1 and 4.4.2, we will make the assumption that each processor has an infinite cache. Hence, at any point in time, it possesses all resources for all tasks performed by that time. We will relax this assumption in Section 4.5.

We introduce some useful notation. Let S be the set of processors. Let R be the set of resources. Let T be the set of tasks. We shall also use T to refer to the set of *incomplete* tasks at any point in time.

Notation 4.1 $R(t) \subseteq R$ is the set of resources required to execute task t .

Notation 4.2 $R(s) \subseteq R$ is the set of resources possessed by server s .

Notation 4.3 $R(r, s) = 1$ if $r \in R(s)$ (i.e., processor s possesses resource r); else, 0 .

We define the *cost*, $C_{t,s}$ (Equation 1), of executing task, t , on server, s , as the sum of resources that need to be obtained before the task can be executed (Equation 1). For instance, if a processor s already has resources x and y and task t requires resources w, x, y, z the cost of evaluating t on s is 2, corresponding to obtaining resources w and z . Note that we are making the implicit assumption that all resources are equally expensive to obtain for all processors. Note that by *cost* we mean *only* the cost of acquisition of resources, not the computational time.

$$C_{s,t} = \sum_{r \in R(t)} (1 - R(r, s)). \quad (1)$$

For complexity analysis, we assume that each task requires a constant number of resources i.e., $\forall t : |R(t)| = 1$.

4.4.1 Local Algorithm

In this section, we describe a scheduling algorithm which operates under the infinite-cache assumption. The algorithm is *local* i.e., each server assesses the value of a task without consideration of other servers. While it is simple, it suffers, in performance (Section 5), a limitation we address in Section 4.4.2.

The local scheduler under the infinite cache assumption, works as follows. Assume server s requests a task. From the set of uncompleted tasks, T , assign task t' to processor s such that $C_{t',s}$ is minimum over all $t \in T$.

Theorem 4.1 *The local algorithm requires $O(|T|)$ time per scheduling decision.*

Proof : Every time a server requests a task, there are $O(T)$ tasks to be evaluated. Each evaluation requires $O(1)$ work since, by assumption, $\forall t : |R(t)| = 1$. The proof follows. \square

4.4.2 Global Algorithm

In this section, we show how one can make scheduling decisions from a global viewpoint, as opposed to a purely local viewpoint (Section 4.4.1). In this algorithm, if processor, s' , requests a task and there exists a task, t' , of zero cost for it, ($C_{t',s'} = 0$), we assign t' to s' . When no such task exists, the scheduling decision becomes more complex. Let the smallest cost to perform task t be C_t (Equation 2).

$$C_t = \min_{s \in S} (C_{t,s}) \quad (2)$$

At a given point in time, let, C^{old} (Equation 3), be the sum of the smallest cost to perform all remaining tasks.

$$C^{old} = \sum_{t \in T} C_t \quad (3)$$

Consider a tentative assignment of t' to s' . The cost or *investment* involved in making this assignment is $C_{t',s'}$. Based on this assignment, let the new cost (similar to Equation 3) be $C^{new}(t', s')$. Therefore, the gain or *return on investment* of this assignment is $G(t', s') = C^{old} - C^{new}(t', s')$. We choose t' so as to maximize the profit, $P'_{t',s'}$ (Equation 4) which is the difference between the investment and the return on investment.

$$P'_{t',s'} = (C^{old} - C^{new}(t', s')) - C_{t',s'} \quad (4)$$

However, Equation 4 does not capture the reality that the ability of s' to perform t'' at a small, or zero, cost, does not guarantee that it will in fact be assigned t'' . This is because of load balancing requirements which might require the scheduler to assign t'' to s'' even if $C_{t'',s''} > C_{t'',s'}$. Hence, we need to temper our estimate of the return on investment. We do so by using a factor $\beta(0 < \beta < 1)$. Determining the optimal value of β remains an open problem. We now replace Equation 4 with Equation 5.

$$P_{t',s'} = \beta(C^{old} - C^{new}(t', s')) - C_{t',s'} \quad (5)$$

Note that when $\beta = 0$, maximizing $P_{t',s'}$ is the same as minimizing $C_{t',s'}$, which is precisely the local algorithm(Section 4.4.1). We now discuss a tie-breaking mechanism invoked when more than one task has the same gain for a given processor. The intuition behind this approach is to reduce the overlap between the resources possessed by different processors. N_r (Equation 6) is the number of processors that possess r . Let $AR(s', t')$ (Equation 7) be the set of *Additional Resources* that s' must acquire to perform t' . We choose t' so as to minimize $RO(t', s')$ (Equation 8), which measures the increase in Resource Overlap when s' acquires $AR(t', s')$.

$$N_r = \sum_{s \in S} R(s, r) \quad (6)$$

$$AR(s', t') = R(t) - R(S) \quad (7)$$

$$RO(t') = \sum_{r \in AR(s', t')} N_r \quad (8)$$

The global scheduler under the infinite cache assumption works as follows. When a processor s requests a task, Assign it task t' if $\exists t' : C_{t',s} = 0$. Else, assign task t' such that Equation 5 is maximized. In either case, if more than one task qualifies, assign the task for which $RO(t)$ is minimum.

Lemma 4.1 $C_{t,s}$ does not increase. C_t does not increase.

Proof : Follows from fact that $R(r, s)$ does not change once it becomes 0 and from Equation 4.3. □

Theorem 4.2 The global algorithm requires $O(T^2)$ time per scheduling decision.

Proof : Let $C_{t,s}$ and C_t be constructed initially at a one-time cost of $O(ST)$. At each request by processor s' , we have to (i) evaluate $C_{s',t}$ for each $t \in T$, (ii) update $C_t^{new} = \min_{s \in S} C_{s',t}$, and (iii) calculate $C^{new}(t', s') \leftarrow \sum_{t \in T} C_t^{new}$. Using Lemma 4.1, we see that step (ii) above requires us to concern ourselves only with this particular server, s' . Hence, all three steps above require $O(T)$ time. Note that once the choice of t' is decided upon, it requires, no additional time to update $C_{t,s'}$ and C_t appropriately. The above three steps have to be performed for each possible value of t' , yielding a time complexity of $O(|T|^2)$. □

4.5 Removing the Infinite Cache Assumption

In Sections 4.4.1 and 4.4.2, we made the arguably unreasonable assumption that a processor has an infinite cache i.e., once it acquires a resource, it possesses it for all subsequent time. In this section, we seek to relax that assumption. This requires reasoning about what resources a processor is likely to relinquish. While this reasoning is rife with approximation, we show (Section 5.3) that it can improve performance.

Assume the requests by processor s to the scheduler to occur at times $0, 1, 2 \dots$ where 0 refers to the most recent request. The scheduler can keep track of the last time, $V(r, s)$, at which resource r was acquired by processor s . Initially, $\forall r \forall s : V(r, s) = \infty$. Assume that server s has just been assigned task t . This means that $\forall r \in R(t), V(r, s) = 0$. Assume that s had been assigned t' at the previous point in time. This means that $\forall r \in (R(t') - R(t)), V(r, s) = 1$.

In Sections 4.4.1 and 4.4.2, $R(s, r) = 1$ if s had been assigned a task that required r at some previous time and 0 otherwise. Now, we make a probabilistic estimate of $R(s, r)$ as $\alpha^{V(r, s)}$, where $0 < \alpha < 1$ is an aging parameter. A drawback of our current approach is that the value of α is set somewhat arbitrarily.

Both the local and global algorithms can be modified to remove the infinite cache assumption by simply changing the way $R(r, s)$ is calculated as shown above. Unfortunately, since Lemma 4.1 does not apply, this introduces a factor of $O(ST)$ with each scheduling decision.

In the next section we qualitatively compare the three policies on a simple template application and show why we believe resource-aware scheduling is so important for minimizing communication.

5 Scheduling Policy Analysis

In this section, we evaluate the pros and cons of the various scheduling policies using a template application that closely mirrors, in its computational and data access patterns, several data mining applications.

In Section 5.1, we describe the template application. In Section 5.2, we compare the algorithms of Section 4.4.2 and 4.5 and a simple dynamic scheduler. Later in Appendix 5.4, we discuss the primary limitation of the resource aware strategies, scheduling time complexity, and outline two ways in which this limitation can be alleviated.

5.1 (Upper Diagonal) Matrix Multiply

The template application is an upper diagonal matrix multiply. The tasks are: $\forall i, \forall j < i$, compute $C_{i,j} \leftarrow \sum_k A_{i,k} \times B_{k,j}$. The resources are the rows of A and the columns of B . The resource requirements of task $C_{i,j}$ is row i of A and column j of B . The *cost*, to a server, of performing $C_{i,j}$ is 2 if it possesses neither row i of A nor column j of B ; 1, if it possesses either one and 0 if it possesses both.

We now motivate our choice of this particular template application by showing that it mirrors the computational and data access patterns of many commonly used data mining algorithms. We list a few examples below.

- **Feature Selection:** The goal of feature selection [DKS95] is to select the smallest subset of features, $\{X_1, \dots, X_n\}$, that best determines the class label, Y . Let $f(X_i, X_j)$ be the ability of features X_i and X_j to jointly predict Y . For expository simplicity, limit the number of features selected to 2. Feature selection can be rephrased as evaluating $\forall i \forall j < i : f(X_i, X_j)$, which is identical to the template application.
- **Diff:** The diff primitive [Sub98] provides a high-level view of the differences between two databases that share the same set of attributes, $\{X_1, \dots, X_n\}$. Again for expository simplicity, assume that we are interested only in pair-wise differences. Let $f(X_i, X_j)$ be the difference between the data sets when each is projected onto attributes X_i and X_j . The diff problem is to rank $\forall i \forall j < i : f(X_i, X_j)$.
- **Decision Trees:** In the process of growing a decision tree [Qui93], the problem is to determine which leaf node to split and for each leaf node, which attribute to use for the decision at that node. Consider the computation at a node. While

typically, a single attribute is used as the decision variable, one can well consider extensions to more than one attribute (e.g., $X > 5 \wedge Y < 6$) as long as the decisions remain simple [PSV98]. We [PSV98] clearly demonstrate that this approach is comparable to, and in some instances better than, the state of the art work in discretization, at a fraction of the computational cost. Limiting oneself to selecting attributes pair-wise, the problem is to determine $f(X_i, X_j)$ for all pairs X_i, X_j , where $\{X_i, X_j\}$ are the attributes and $f(X_i, X_j)$ measures the goodness of X_i, X_j as a decision attribute.

Another important problem in this domain is in visualizing the results of discretization (also clustering). Since current limitations in graphics pretty much restrict you to 2 or 3D (two base attributes and one goal attribute) visualizations, an important task of the discretizer is to decide what to display to the user, i.e, identify which of several base attribute pairs best display separations between discrete regions. In order to do this it has been suggested in past work that entropy [SVC97] could be used to pick the pair of attributes that best meet the desired objective. Once again the formulated problem (computing and comparing the entropy for all base attribute pairs) resembles our template application.

We note that while we are limiting our analysis, and thereby our examples from the data mining world, to datacubes (resources per task) of size two for expository simplicity, our scheduling algorithm is capable of handling larger datacube problems. Furthermore, while the examples we outlined above are such that can take advantage of our resource-aware scheduling algorithms, our primitive is capable of addressing other data mining applications as well. Many distributed data mining tasks like association mining (ECLAT [ZPOL97]), sequence mining (SPADE [Z98]), and clustering [SGIF97], take an approach of dividing or replicating the data set (or the dataset is pre-divided) and selectively broadcasting the data at the beginning of the process, thus eliminating the need for communication of the data set during the learning process. Our architecture and primitive allows us to handle these scenarios as well. By separating the data acquisition from the task acquisition, in our architecture, and having distributed data servers we actually permit multiple compute servers to obtain their data simultaneously, as opposed to the traditional centralized broadcast approach. If on the other hand the application requires a central broadcast of data, then our D-DOALL primitive supports this as well via the *inputsList* parameter (see figure 6).

5.2 Comparison of scheduling strategies

To compare the scheduling policies, we used our template application, upper-triangular matrix multiply where A, B, C are 8×8 matrices and 4 servers, so that $|R| = 16, |T| = \binom{8}{2} = 28, |S| = 4$. (The small numbers are merely to simplify presentation of results.) . The assumptions under which the scheduling strategies are evaluated are:

- Servers request for new tasks in a fixed order. Without any loss of generality we assume that servers 1, 2, 3, and 4 request for tasks in that order.
- All servers are equally fast and equally loaded.
- Time to make a scheduling decision is the same for all policies.

Figure 7 depicts the schedule generated by Simple, Local resource aware, and the global resource aware scheduling policies. Numeric values in the corresponding task box indicate which server executed that particular task. Below each schedule, we indicate how many resources (broken down into rows of A and columns of B) each server acquired and the total resources acquired by all servers. For example from the analysis we see that under Simple Scheduling, server 4 obtained 5 columns and 6 rows.

The fewer the total number of resources acquired, the better the overall performance. The global scheduling policy (Section 4.4.2) outperforms the local scheduling policy which in turn outperforms the simple, dynamic scheduling policy. Interestingly enough, for this example, the schedule generated by the global policy is optimal.

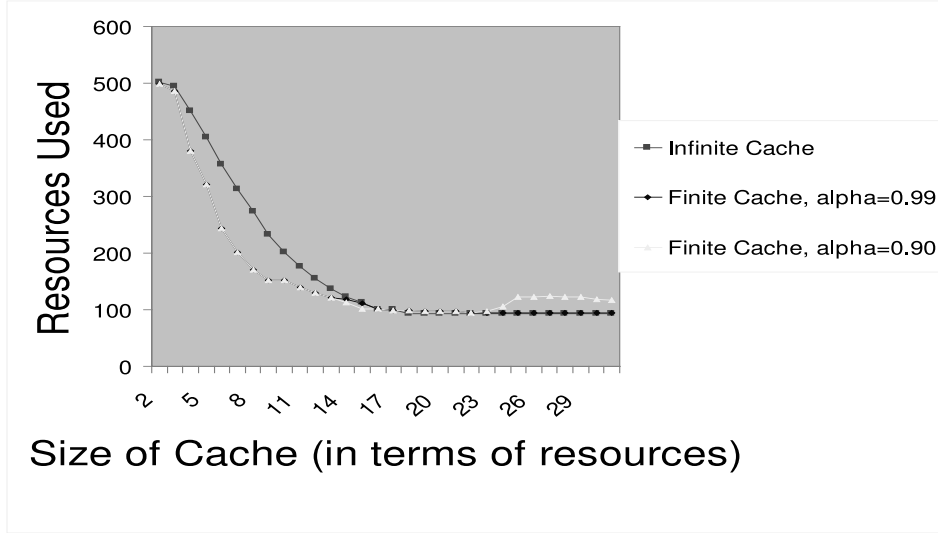


Figure 8: Effect of Finite Cache on Scheduling Effectiveness

5.4.1 Reducing the number of tasks

Merging tasks into a smaller number of mega-tasks can reduce the time complexity. While we want $|T| > |S|$ for load-balancing purposes, having $|T| \gg |S|$ makes scheduling harder without substantially improving load-balancing. The intuition underlying the merging is to aggregate those tasks into mega-tasks that have as much commonality in their resource requirements as possible.

Problem Definition. Let $G = (T \cup R, E)$ be a bipartite graph with vertices $T \cup R$ and edges E . An edge $e = (x, y) \in E \Leftrightarrow x \in T \wedge y \in R$. Nodes in T represent tasks, nodes in R represent resources and an edge from t to r indicates that task t requires resource r . Let $C < |T|$ be the desired number of mega-tasks. Let T' be an exhaustive and mutually exclusive set of subsets of T i.e., $T' \subset 2^T$ such that $t'_i, t'_j \in T' \Rightarrow t'_i \cap t'_j = \phi \wedge \cup_i t'_i = T$ and 2^T is the power set of T . Let $r(t) \subseteq R$ be the set of resources required by task t i.e., $r \in r(t) \Rightarrow (t, r) \in E$. Ideally, there would be a high degree of overlap between the resources required by the tasks in a mega-task. This is formalized in Problem 5.1.

Problem Definition 5.1 Given $G = (T \cup R, E)$, C , and g , find $T' \subset 2^T$ such that $|T'| = C$ and $\sum_{t' \in T'} |\cup_{t \in t'} r(t)|$ is minimum

Algorithm for Problem 5.1 We have been unable to find an efficient solution to Problem Definition 5.1. Our heuristic approach is a simple, extension to the algorithm we used for resource-aware scheduling. We create as many “dummy servers” as the number of mega-tasks required. We make these dummy servers requests tasks in order. On completion, the set of tasks assigned to a given “dummy server” constitutes a mega-task. While expensive, this processing step can be done off-line and needs to be done only once for a given instantiation of R , T and S and the resource dependency $R(t)$.

5.4.2 Overlapping Scheduling with Task Execution

The scheduler could determine the next task to execute for *each* processor while waiting for a task request. Alternatively, it could order the processors in the likelihood of their being the next processor to make a request (based on when the processor last requested a task) and select the best task for the processors in that order.

6 Experimental Analysis

In this section, we examine the performance of D-DOALL scheduler on 2-dimensional Discretization and Clustering. We also provide a glimpse of why support for interactivity and specifying task order is essential for applications like clustering.

All experiments were performed on a client and upto four compute servers each a dual 200 MHz Intel Pentium Pro system running Windows NT 4.0 with 256 MB RAM. For every experiment, numbers reported were averaged over 16 runs at different times of the day.

6.1 2-D discretization

In the process of growing a decision tree [Qui93], the problem is to determine which leaf node to split and for each leaf node, which attribute to use for the decision at that node. Consider the computation at a node. While typically, a single attribute is used as the decision variable, one can well consider extensions to more than one base attribute (e.g., $X > 5 \wedge Y < 6$) as long as the decisions remain simple [PSV98]. Limiting oneself to selecting base attributes pair-wise, the problem is to determine $f(X_i, X_j)$ for all X_i and for all pairs X_i, X_j , where $\{X_i\}$ are the attributes and $f(X_i, X_j)$ measures the goodness of X_i, X_j as a decision attribute. This problem is referred to as 2-Dimensional Discretization. We evaluate our scheduling algorithms on a single node split using 2-Dimensional (all pairs X_i, X_j) Discretization where the evaluation function $f(X_i, X_j)$ used is Entropy. Evaluating any pair of attributes involves 3 steps, obtaining the data involving the two attributes, computing the probability density (pdf) estimate, searching for the optimal (determined by goodness function; Entropy) cut-point.

6.1.1 Effect of Locality Based Scheduling

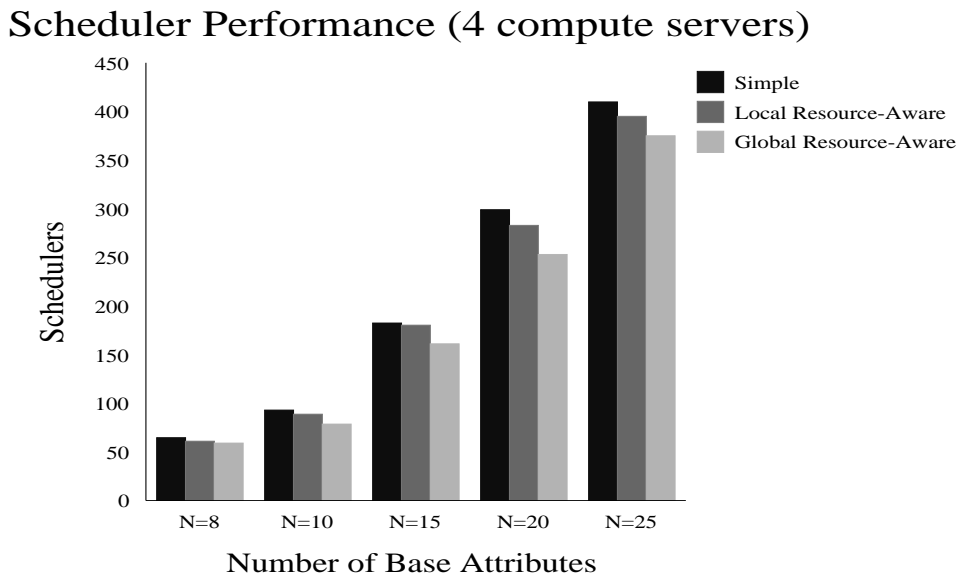


Figure 9: Locality Based Scheduling Performance

In this experiment we evaluated the effect of our scheduling strategies on a synthetic dataset XOR [PSV98]¹ with N (base) + 1 (goal) attributes, where N was varied from 8 to 25, and 100000 instances. Each pair-wise evaluation of the N base attributes is independent from the other and is executed as a D-DOALL loop. Resources (data) for any evaluation required the corresponding base attribute columns and the goal attribute column which were obtained from the dataserver. The corresponding task-resource matrix was encoded and passed to the D-DOALL primitive for the locality based scheduling algorithms.

¹ details of dataset generation can be found from the cited article

Once a task (evaluating a base pair) is assigned to a compute server, that compute server determines if the resources required are present locally. If not, it goes and fetches the resources from a central data server. On receiving a request, the data server (refer section 2) generates the resource (row/column) on the fly and ships it to the corresponding compute server. Once the resources are acquired, the compute server places it on a local data cache and executes the task.

Results are presented (refer Figure 9) for 3 scheduling algorithms, i.e., simple scheduling, local resource aware scheduling and global resource-aware scheduling. From the results we see that local resource scheduling outperforms simple scheduling by 5-10%. Global resource aware scheduling outperforms local resource aware scheduling anywhere from 5-15% and outperforms simple scheduling by as much as 20%. The speedups obtained by the global scheduling algorithm in spite of the increase in scheduling time underline the importance of global affinity scheduling for such applications. Note that these results reflect total running time without the optimizations for reducing scheduling time presented in Section 5.4.

In terms of raw speedup global resource scheduling generates a speedup of 3 resulting in an efficiency of 75%. This lower efficiency is due to two reasons. First, there is contention at the data server. Second, with more number of processors the total number of resources acquired is more (attribute columns are replicated).

6.2 Clustering

Clustering is a commonly used data mining technique which partitions the attribute space, from which the data is drawn, into regions of similarity [FPSS96]. The assignment of instances to classes (or clusters) could be deterministic (all instances in a region belong entirely to that class) or probabilistic (every instance belongs to every class to an extent determined by its location in the attribute space). Determining how similar two instances can be done either by defining a metric in the attribute space or by postulating the existence of probability density models. Search techniques are often used to find the best clustering, since efficient algorithms do not exist in most cases. These search algorithms are in general, computationally expensive. Searching for the best number of clusters to describe the data is one of the more expensive aspects of the search process. In this paper, we use the cross-validated likelihood strategy of [SGIF97] with $M = 40, \beta_c = 0.5$. M is the number of searches over which the cross-validated likelihood is averaged. β_c is the fraction of the data used to estimate the model and $1 - \beta_c$ is the fraction used to evaluate the cross-validated likelihood.

6.2.1 The Need for Incremental Reporting and Task Ordering

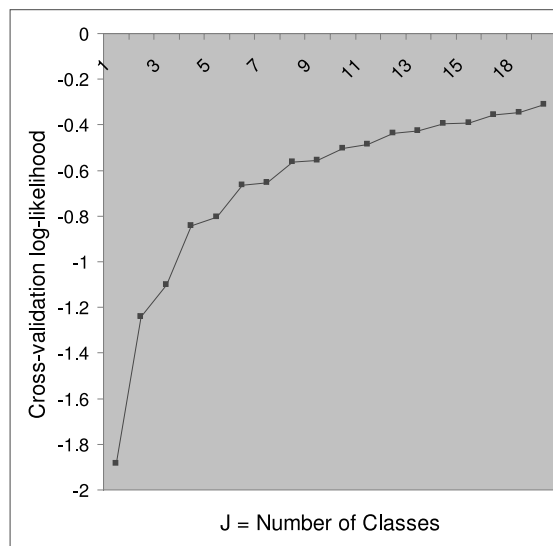


Figure 10: Improvement in cross-validated likelihood as J increases

The motivation for making data mining algorithms report their results incrementally and having parallelization primitives that provide this feature is evident from the following study. The search for the most appropriate number of classes is parallelized over different workstations using the interactive D-DOALL primitive (Section 3). We generated a synthetic data set with $I = 2^{16}$ instances. Each of $K = 3$ independent attributes was modeled independently as a mixture of $J = 8$ Gaussians. We searched for the best J in $[1..32]$. None of the searches for $J > 19$ converged. Given that the tasks are handed out in order of increasing J , we can see that the marginal improvement in the ability to describe the data set begins to fall off around $J = 8$. Providing feedback to the user in the form of Figure 10 allows the user to terminate searches for higher J when he/she feels that the incremental improvement is not worth the additional complexity.

Being able to specify the desired order of task execution is important since one is often biased towards choosing simpler clusters (small J). This is an important scheduling characteristic of many data mining algorithms. Namely, that while there may be a large number of hypotheses to be explained, one often desires to impose an ordering in the exploration of those hypotheses.

6.2.2 Speedup obtained using D-DOALL

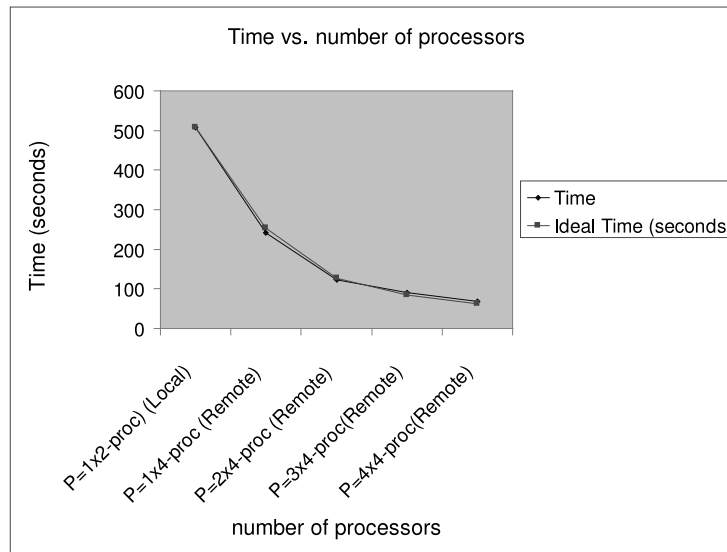


Figure 11: Speedup in clustering using D-DOALL

In this section, we present our results on the use of D-DOALL to parallelize the search for the best number of classes. The experimental setup is as follows. We searched for the best number of classes in $[2..8]$ using the cross-validated likelihood strategy with $M = 20, \beta_c = 0.5$ [SGIF97]. The data set used was the *adult* data set with $I = 32561$ instances [MM96]. The client was a 2×200 MHz Pentium Pro machine running Windows NT 4.0. The remote compute servers were 4×200 MHz Pentium Pro machine running Windows NT 4.0. Our results are in Figure 11. The ideal time line in the figure uses the time when the entire computation happens on the client (1×2 -proc (Local)) as the baseline and then computes the ideal time (assuming linear speedup) for the other configurations. We see from the figure that the overhead of D-DOALL is small and that speedup is almost linear.

7 Conclusions and Future Work

In Section 4.4.2 and Section 4.5, we introduced scheduling parameters α and β , the values of which were set somewhat arbitrarily. It would be interesting to see if the scheduler could *learn* the optimal values of α and β based on the actual scheduling decisions it makes and the quality of those decisions. The algorithms of Sections 4.4.2, 4.5 and 4.5 could be

improved and their analysis tightened. An exact solution to Problem Definition 5.1 would also be useful.

As part of ongoing work we are evaluating the primitive proposed on Feature Selection, and Diff Discovery and hope to have results for a later version of this paper. We also plan to evaluate the effectiveness of this primitive on applications from other domains (such as interactive vision).

We would like to thank Ramana Venkata for many useful discussions on this work.

References

- [AF93] R. Agrawal, C. Faloutsos, and A. Swami. Efficient similarity search in sequence databases. In *Foundations of Data Organization and Algorithms*, Washington DC, 1993.
- [Bea96] F. Berman and et al. Application-level scheduling on dist. het. networks. *Supercomputing*, November 1996.
- [CLZ95] M. Cierniak, W. Li, and M. J. Zaki. Loop scheduling for heterogeneity. In *4th IEEE Intl. Symp. on High-Performance Distributed Computing, also TR 540, U. Rochester*, August 1995.
- [CHN⁺96] D. Cheung, J. Han, V. Ng, A. Fu, and Y. Fu. A fast distributed algorithm for mining association rules. *4th Intl. Conf. Parallel and Distributed Info. Systems*, December 1996.
- [CR92] A. L. Cheung and A. P. Reeves. High performance computing on a cluster of workstations. *1st IEEE Intl. Symp. High-Performance Distributed Computing*, July 1992.
- [DKS95] James Dougherty, Ron Kohavi, and Mehran Sahami. Supervised and unsupervised discretization of continuous features. In *Proceedings of the 12th International Conference on Machine Learning*, pages 194–202, 1995.
- [FI93] Usama. M. Fayyad and Keki B. Irani. Multi-interval discretization of continuous-valued attributes for classification learning. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pages 1022–1027, 1993.
- [FPSS96] U. Fayyad, G. Piatetsky-Shapiro, and P. Smyth. The KDD process for extracting useful knowledge from volumes of data. *Communications of the ACM*, 39(11):27–34, November 1996.
- [Gea94] A. S. Grimshaw and et al. Metasystems: An approach combining parallel processing and heterogeneous distributed computing systems. *Journal of Parallel and Dist. Computing*, 21(3), 1994.
- [Hig93] High Performance Fortran Forum. High Performance Fortran language specification. *Scientific Programming*, 2(1-2):1–170, 1993.
- [LP93] W. Li and K. Pingali. Access normalization: Loop restructuring for NUMA compilers. *ACM Trans. on Computer Systems*, 11(4), November 1993.
- [LK87] F. Lin and R. Keller. The gradient model load balancing method. *IEEE Transactions on Software Engineering* (13), 1987.
- [Mes94] Message Passing Interface Forum. MPI: A message-passing interface standard, version 1.0, May 1994.
- [ML94] E. Markatos and T. Leblanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. *IEEE TPDS*, pages 379–400, April 1994.
- [MM96] C. J. Merz and P.M. Murphy. *UCI Repository of learning databases*, 1996. <http://www.ics.uci.edu/mllearn/MLRepository.html>.
- [NS93] H. Nishikawa and P. Steenkiste. A general architecture for load balancing in a distributed-memory environment. *13th IEEE Int. Conf. on Distributed Computing*, May 1993.
- [PSV98] S. Parthasarathy, R. Subramonian, and R. Venkata. Generalized discretization for summarization and classification. In *2nd International Conference on Practical Applications of Knowledge Discovery and Data Mining*, pages 219–239, 1998.
- [Pol88] C. D. Polychronopoulos. *Parallel Programming and Compilers*. Kluwer Academic Pub. 1988.
- [Qui93] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, Los Altos CA, 1993.
- [SAM96] J. Shafer, R. Agrawal, and M. Mehta. Sprint: A scalable parallel classifier for data mining. In *VLDB*, pages 544–555, 1996.

- [SGIF97] P. Smyth, M. Ghil, K. Ide, and A. Fraser. Detecting atmospheric regions using cross-validated clustering. In *Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining*, pages 61–66, 1997.
- [Sub98] R. Subramonian. Defining *diff* as a data mining primitive. In *Fourth International Conference on Knowledge Discovery and Data Mining*, 1998. to appear.
- [SVC97] R. Subramonian, R. Venkata, and J. Chen. A visual interactive framework for attribute discretization. In *Third International Conference on Knowledge Discovery and Data Mining*, pages 82–88, 1997.
- [Wol96] Michael Wolfe. High performance compilers for parallel computing. *Addison Wesley*, 1996.
- [ZPOL97] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New parallel algorithms for fast discovery of association rules. *Data Mining and Knowledge Discovery: An International Journal*, December 1997.
- [Z98] M. J. Zaki. Efficient enumeration of frequent sequences. In *7th Intl. Conf. on Information and Knowledge Management*, November 1998.