A Parallel, Multithreaded Decision Tree Builder

Girija J. Narlikar

December 1998
CMU-CS-98-184

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Parallelization has become a popular mechanism to speed up data classification tasks that deal with large amounts of data. This paper describes a high-level, fine-grained parallel formulation of a decision tree-based classifier for memory-resident datasets on SMPs. We exploit two levels of divide-and-conquer parallelism in the tree builder: at the outer level across the tree nodes, and at the inner level within each tree node. Lightweight Pthreads are used to express this highly irregular and dynamic parallelism in a natural manner. The task of scheduling the threads and balancing the load is left to a space-efficient Pthreads scheduler. Experimental results on large datasets indicate that the space and time performance of the tree builder scales well with both the data size and number of processors.

This research is supported by ARPA Contract No. DABT63-96-C-0071. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright notation thereon. Views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of ARPA or the U.S. Government.

Keywords: Decision tree building, multithreading, parallel C4.5, scalable data mining, lightweight Pthreads, space efficiency, dynamic scheduling.

1 Introduction

The increasing availability of massive amounts of data in science and industry has made data mining applications such as data classification highly prevalent. Data classification is used to predict the value of a class (a dependent attribute) in a data record based on the values of its other attributes. A set of data records with known class values, called the *training set*, is used to construct a model for enabling this prediction. In particular, decision tree-based models have become a popular mechanism for data classification, because decision trees are relatively fast to compute, they are fairly simple to interpret by humans [30], and they obtain accuracies comparable to other classification techniques [21]. Larger training sets enable the construction of more accurate decision trees, but also increase the processing and memory requirements of the tree building phase. Therefore, techniques that speed up decision tree building are useful. Recent work has focused on parallelizing decision tree builders to meet this goal [22, 18, 33, 12]. Many businesses are now investing in cost-effective symmetric multiprocessors (SMPs) with large amounts of memory to speed up their data classification tasks.

Decision tree building algorithms have a large amount of inherent parallelism. However, the structure of the final decision tree and the work involved in building each subtree is highly variable and data-dependent. Therefore, implementing a parallel decision tree builder that can make effective use of multiple processors by balancing the load and yet provide reasonable data locality is a difficult problem. Some recent parallel methods have focused on building the decision tree from out-of-core training data, that is, data that is too large to fit in main memory [22, 33, 18]. However, the availability of cost-effective symmetric multiprocessors (SMPs) equipped with a few gigabytes of main memory is increasing, and the demand for interactive packages requiring fast decision tree builders is rising (*e.g.*, [35, 29, 1]). Therefore, we believe that building decision trees from in-core data remains an important problem.

This paper presents a parallel implementation of a decision tree builder for in-core data that uses fairly fine-grained, lightweight threads. The job of efficiently implementing the program is left to the thread scheduler, allowing the expression of simple, high-level code. Since the program is expressed in terms of a large number of dynamic, lightweight threads, the thread scheduler can automatically balance the load across the processors. However, if the scheduler creates too much active parallelism, the high-level, fine-grained parallel program can often end up requiring much more memory than its serial counterpart [6, 32, 31, 26]. Increased memory requirements limit the size of the largest dataset than can be used to construct the tree without paging to disk. In general, the running time can be affected even before the physical memory is exhausted [26]. Therefore, we use a provably-efficient thread scheduler [27] that provides good space and time performance for the multithreaded decision tree builder. In effect, this thread scheduler partitions the work involved with the upper part of the tree across all the processors. However, when the working set size decreases below a user-defined threshold (*e.g.*, when it fits in a processor's cache), it automatically schedules the work for an entire subtree on a single processor to get good locality. This scheduling strategy was developed to execute nested parallel programs in general; in particular, this paper shows that it is results in good performance for a high-level decision-tree building code.

We have parallelized the original C4.5 decision tree building algorithm as proposed by Quinlan [30]. We exploit divide-and-conquer parallelism in C4.5 at two levels: at the outer level of building the tree node-by-node in a top-down, recursive manner, and at the inner level of sorting data records within a single node using a parallel quicksort. The expression of this high degree of fine grained parallelism has several advantages, including simpler code, adaptability to a changing number of available processors, and automatic load balancing. We feel that this approach of using fine grained threads with an efficient thread scheduler could potentially be extended to other decision tree building techniques.

The decision tree building program uses a lightweight, user-level implementation of Posix standard threads or Pthreads [17]; the Pthreads implementation has been modified to use the space-efficient scheduler [27]. The results of executing the program on an 8-processor SMP indicate that the implementation scales well with both the number of processors and the number of data instances (records). For example, for an input dataset with 1.6M instances on 8 processors, we obtain a speedup of 6.32 with respect to an efficient, serial version of the program. The results also indicate that the modified, space-efficient Pthreads scheduler provides better space and time performance for the tree builder, in comparison to the original Pthreads scheduler.

The remainder of this paper is organized as follows. Section 2 describes how our parallel formulation of the C4.5 tree building algorithm. Section 3 provides an overview of the hardware and software systems used to implement the algorithm. The experimental results are presented in Section 4. Section 5 presents an overview of related work in parallel decision tree building, followed by a summary and discussion in Section 6.

2 Parallelizing C4.5

Recent tree building algorithms [20, 33], designed mainly for out-of-core data, result in equivalent performance as Quinlan's C4.5 [30] for in-core data. Since the focus of this work was on building the decision tree from in-core data, that is, data that fits in the main memory of the machine, we decided to parallelize the original C4.5 algorithm.

C4.5 is a top-down divide-and-conquer algorithm that builds a decision tree from instances with both discrete and continuous attributes. Our implementation currently handles only continuous attributes, and therefore builds binary trees. At each node, the best attribute (and best split) is picked according to the highest gain ratio. Computing the gain ratio involves sorting the instances according to each attribute at each tree node. Since the time for pruning the tree is negligible ($< 1\%$) compared to the time required for the tree building phase [20], we have not currently implemented the pruning phase.

We took advantage of the intrinsic divide-and-conquer nature of the C4.5 algorithm to parallelize it using lightweight threads; Figure 1 shows our parallel formulation of the algorithm. A new child thread is dynamically created to execute each of the two branches of the `in parallel` construct (which include the recursive calls to `Tree_Build`). The child threads may execute in parallel; after finishing the parallel recursive call, they synchronize with the original (parent) thread and terminate. In practice, since threads are typically more expensive than simple function calls, we amortize thread overheads by switching to serial recursion when the dataset size becomes very small. To increase the degree of parallelism in the code, the work executed at each tree node can also be parallelized. We executed each sort at a node using a parallel divide-and-conquer quicksort; again, each recursive call to quicksort is executed by a new thread. Thus, our code has two levels of divide-and-conquer parallelism: at the outer level over the decision tree, and at the inner level (within each tree node) in the quicksort. The resulting computation graph is highly irregular and data dependent, and it is left to the thread implementation to schedule it efficiently at runtime. Note that this use of a large number of lightweight threads allows the program to be written at a fairly high level, with code that reflects the natural parallelism in the program. Further, the load can be balanced automatically by the thread scheduler, and the program can adapt to a changing number of available processors. A coarse-grained version in which the programmer explicitly partitions and maps the parallelism onto the processors and dynamically balances the load would be significantly more complex. A more detailed description of the advantages of using lightweight threads in general can be found elsewhere [26].

The sort was executed in place and did not involve any dynamic allocation of heap memory. However, in the outer level of the divide-and-conquer parallelism, heap memory is dynamically allocated at each tree node to store the subset of instances that filter down to that node. Thus, the instances used at each node are stored in contiguous memory, leading to simpler code and efficient use of the processor cache. Since the threads constructing the node's children subsequently access these instances in parallel (to extract their

```
Tree_Build(data) {
   if Is_Leaf(data) return (Make_Leaf(data));
   (attr, split) = Find_Best_Split(data);
   cell = Make_Cell(attr, split);
   in parallel {
      {
         data1 = Extract_Data(data, attribute, split, lesser_or_equal);
         cell->left = Tree_Build(data1);
      }
      {  data2 = Extract_Data(data, attribute, split, greater);
         cell->right = Tree_Build(data2);
      }
   }
  return (cell);
}


Find_Best_Split(data) {
   for each attr {
      Sort(data, attr);         /* parallel quicksort */
      find best split in sorted data;
   }
   pick attr a with best split s;
   return (a,s);
}
```

Figure 1: Pseudocode for the parallelized version of C4.5. `Extract_Data()` allocates space for and extracts instances with values of the given attribute that are either less-than-or-equal, or greater than the given split value. `Sort()` sorts the instances according to the given attribute using a parallel, divide-and-conquer quicksort.

subset), this heap memory is freed after the entire subtree under the node has been constructed[1]. Thus, due to the large numbers of threads expressed and the big amounts of dynamic memory allocated, we decided to use a space-efficient thread scheduler [27], as described in Section 3.

Code complexity. We provide the number of lines of code as a measure of the complexity of writing the parallel tree builder using the high-level model of lightweight threads. The initial serial version was under 700 lines of code, including comments and code for I/O, debugging and testing; around 50-100 of these lines could have been omitted if the code was not written with the intention of being subsequently parallelized. Starting from the 700 lines of serial code, three functions, totaling to under 150 lines of code, were duplicated to convert them into their parallel versions; we retained the original serial versions for serial recursion near the leaves of the decision tree. Less than 40 new lines of code were added to parallelize these three functions using Pthreads; the majority of these 40 lines involved packing and unpacking arguments for the Pthreads. Thus, the final code had less than 900 lines of code.

[1] A more complex method of coding the algorithm can free the memory at an earlier stage, once all the child threads have extracted their subsets.

3 Execution Platform

The hardware platform for the experiments described in this paper is an 8-processor Sun Enterprise 5000 SMP running Solaris, with 2GB of main memory. Each processor is a 167MHz UltraSPARC with a 512KB L2 cache. All the experiments described in this paper have a space requirement of less than 2GBs, so that the data is always resident in the main memory.

Posix threads or Pthreads [17] support a standard interface adopted by a number of SMP vendors [36, 37, 14, 7, 15]. We use the native Pthreads package on Solaris 2.5 [37], which implements Pthreads at the user level. This allows common thread operations such as creation and synchronization to be cheaper than corresponding operations on kernel-level threads [28, 26]. Thus, thousands of threads can be dynamically created and destroyed during the execution of the program.

We use two versions of the native Solaris Pthreads implementation[2]. Both versions use a default thread stack size of a page (8KB) instead of the original 1MB, because a small stack is more appropriate for our lightweight, short lived threads. Setting the library's default stack size to 8KB allows the library to cache and reuse previously allocated stacks of that size. The first version of the Pthreads implementation uses a simple FIFO scheduling queue, the only option currently available in the native Solaris Pthreads implementation. We call this version of the Pthreads implementation the "original" version. In general, a FIFO queue may result in a large number of threads being simultaneously created and made active. Consequently, the program may require more memory than necessary, and hence, also suffer from poor performance [26]. Therefore, our second Pthreads version uses a space-efficient scheduler [27, 25]. In general, this scheduler reduces the total amount of memory required to execute a parallel program, while also resulting in good locality and low scheduling overheads. We call this version of Pthreads the "new" version. In the Section 4, we will present experimental results using both Pthreads versions. Note that the standard Posix interface supported by both versions is identical, and therefore we run the same decision tree building program using both versions. We now describe the space-efficient scheduler in more detail.

The space-efficient scheduler

Unlike distributed memory machines, SMPs have smaller (a few megabytes in size) hardware-coherent off-chip caches, and they do not support explicit, user-controlled distribution of data in these caches. Therefore, if some task makes sequential passes over data that is too large to fit in the cache, which processor the task is scheduled on has little impact on its performance (provided the load is balanced). The tasks of creating the nodes in the upper part of the decision tree make sequential passes over large amounts of data, and therefore have such characteristics. However, once the data that filters down to a subtree fits in a processor's cache, it is more efficient to construct the entire subtree on that processor (provided there is sufficient work to keep all processors busy). This characteristic is common to a number of nested, divide-and-conquer parallel applications. The space-efficient scheduling algorithm used in this paper was designed to efficiently execute such applications while keeping total memory requirements low.

The space-efficient scheduler conserves memory requirements by prioritizing threads in their serial, depth-first execution order. Further, the scheduler preempts threads when they allocate too much memory; such threads are essentially lowered in priority. Note that in the parallel C4.5 formulation described in Section 2, threads constructing the nodes in the upper levels of the decision tree allocate large amounts of memory to store data instances. Therefore, these threads may get preempted and subsequently rescheduled on different processors. However, when a thread and its descendent threads do not allocate a large amount of memory, the scheduler allows all of them to execute on a single processor (unless other processors are idle). Since the decision tree in our program is built at each stage by recursively forking new child threads

[2] We had access to the source code for the original Solaris Pthreads implementation, allowing us to make modifications to it.

to build each subtree, entire subtrees near the bottom of the tree are executed on a single processor. Further, the threads within such a subtree are executed by the scheduler in a depth-first order on one processor, which typically results in good cache locality. Similarly, child threads in each invocation of the parallel quicksort (which does not require any dynamic memory allocation) tend to be executed in depth-first order on the same processor as their parent thread. The depth-first execution order is achieved by each processor storing its threads in a LIFO stack; such stacks are globally ordered by the scheduler according to the serial (depth-first) execution order of their threads. Note that whenever a processor becomes idle, it selects a thread to steal at random from high-priority stacks. The thread is stolen from the bottom of the selected stack, and typically corresponds to the computation of a node higher in the decision tree. Such a stealing policy results in higher granularity of work being stolen, leading to fewer steals. For a purely nested-parallel program with a depth (critical path length) of D and a serial (depth-first) space requirement of S_1, the scheduler guarantees that the parallel execution on p porcessors will require $S_1 + O(p \cdot D)$ space [3]. Further details on the design and analysis of the space-efficient scheduler, along with experimental comparisons with other thread schedulers, can be found elsewhere [25, 27].

Note that Pthreads support a wide range of standard functions, and are implemented as a separate library (instead of being part of the programming language). Consequently, they incur overheads for operations such as thread creation and deletion that are two orders of magnitude higher than the cost of a C function call. In our experiments, we amortize such basic Pthread overheads by switching to serial recursion once the dataset at a node reaches a size of 4000 instances (or less). Similarly, in the parallel quicksort, we switch to serial recursion once the number of examples to be sorted falls below 4000. Note that such serialization of subtrees at the lowest levels of the recursion tree (in both the tree build and quicksort) allows both versions of the Pthread scheduler to obtain data reasonable locality near the leaves. However, the space-efficient scheduler can obtain additional locality by scheduling multiple Pthreads close together in the recursion tree on one processor.

4 Experimental Results

We now describe the experiments to evaluate our C4.5 implementation. This section begins by describing the input datasets and the accuracy of our final decision tree for the inputs. Next, we describe the space and time performance of our program.

4.1 Input datasets

Due to the lack of availability of real datasets with more than 150,000 (150K) examples, we decided to use the synthetic dataset proposed by Agrawal *et al.* [3]. Each instance (record) in the synthetic dataset holds information about a customer. The dataset has 8 continuous attributes, each representing some information about the customer[4]. To allow a meaningful comparison with previous work, we used function *pred7* from the synthetic dataset generator [16] to generate the dataset. The binary class declares whether or not the customer has any disposable income that he/she is likely to spend. The function *pred7* computes the class for each customer (record) as shown below.

[3]For example, our parallel formulation of the decision tree builder in Figure 1 is a purely nested parallel program. However, the use of mutexes and condition variables within the Pthread implementation itself adds non-nested-parallel computation to the final Pthreads-based decision tree builder.

[4]We did not use the ninth attribute, namely the zipcode, and we treated the numeric "car" attribute as continuous.

function *pred7*:
disposable := $(2 \times (\text{salary+commission})/3 - \text{loan}/5 - \text{education\_level} \times 5000 + \text{equity}/5 - 10000)$;
if (disposable > 0) **then return** TRUE;
else return FALSE

Performance results for a real, but smaller speech-recognition dataset are reported elsewhere [27, 25].

We verified the correctness of the decision tree builder by measuring its accuracy for the synthetic dataset. Note that since we have not yet implemented tree pruning, our program produces larger-than-optimal trees with a slight cost to accuracy. For example, for the a training set with 100K instances and a test set with 20K instances, our program achieves an accuracy of 96.2%. Data-generating functions simpler than function 7 result in accuracies of 100% for even very small training sets, because in this case, the tree produced by our program is small and pruning is not required. We expect the accuracy of our program to improve after tree pruning is implemented.

4.2 Serial performance

All the speedups reported in this paper are calculated with respect to the serial C version of the program that does not use any threads. To ensure that this base case is competitive with previous work for in-core input sizes, we measured the performance of this serial version for the synthetic dataset. On a 167MHz UltraSPARC with 128MB of main memory, building the tree for one of the biggest problem sizes (300,000 or 300K input instances) that ran without paging required 266 seconds. On a similar machine with more memory, the serial program built a tree from 800K instances in 816 seconds, and took 1748 seconds to build a tree from 1600K instances. The same program ran for an input size of 100K in 155 seconds on a 80MHz PowerPC processor[5]. Note that these numbers are competitive with previous decision tree building implementations [33, 22] on data generated by a function slightly simpler than *pred7* [6]. For example, Zaki *et al.* [39] report a running time of around 1750 seconds for 1000K input instances of a synthetic dataset generated using the simpler function on a 112MHz PowerPC.

Recall that we switch to serial recursion in our code once the number of examples falls below 4000. This ensures that thread overheads are under 1% of the total running time of the program. Consequently, the multithreaded version on one processor runs at most 1% slower than the serial, C program.

4.3 Parallel performance

We now present the results of running the parallel, multithreaded decision tree builder on multiple processors. Parallel implementations of programs often require more memory than their serial counterparts, especially when the parallelism is expressed in a high-level and fine-grained manner. Minimizing the memory requirement of the parallel tree builder allows the use of larger datasets for building the tree without paging to disk. Note that a lower memory requirement also typically results in better time performance due to fewer memory-related system calls, and fewer TLB misses. We therefore present both the time and space performance of the parallel tree builder.

The numbers reported in this section were measured on the 8-processor Enterprise 5000 SMP described in Section 3. In all the figures, "Orig" refers to the Pthreads implementation with the original FIFO schedul-

[5] Because this machine has only 32MB of memory, we did not run bigger problem sizes on it.

[6] The previous implementations were evaluated using a data generating function "function 7" that differs slightly from *pred7*, and is defined elsewhere [33]. Our tree builder ran around 20% faster on the data generated by this simpler function (function 7), compared to data generated by *pred7*, which we use in all our experiments.

ing queue, while "New" refers to the implementation that uses the space-efficient Pthreads scheduler. We ran our program on datasets ranging in size from 100K instances to 1600K (1.6M) instances.

Time performance

Figure 2 shows the speedup of our program for with respect to the serial C version of the program. Although both thread libraries perform fairly well, the space-efficient library scales better with the number of processors. This is probably because it results in a lower memory requirement (less contention between threads for memory allocation), and because it schedules threads in an attempt to get better locality at lower levels of the recursion tree. As expected, the speedups are higher for the larger problem size.

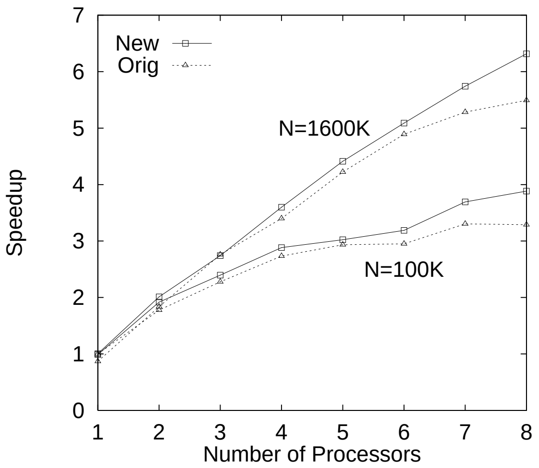

Figure 2: The speedups achieved on up to 8 processors for two different input sizes (100K and 1600K instances). "Orig" denotes the Solaris Pthreads implementation with its existing FIFO scheduler; "New" denotes the implementation modified to use a space-efficient Pthreads scheduler.

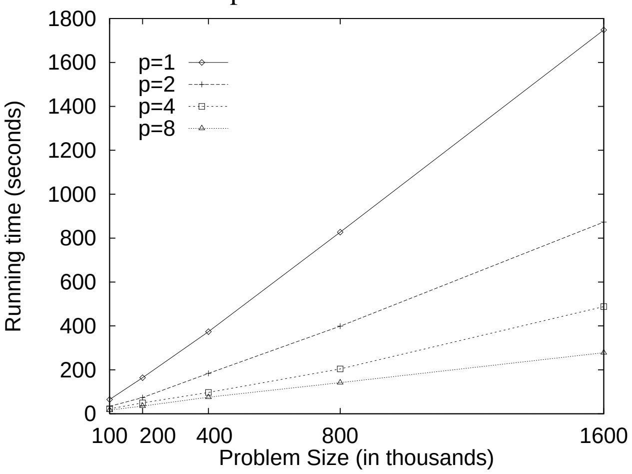

Figure 3: The variation of the running time with the input size for different numbers of processors (p = 1, 2, 4, 8). These experiments used the space-efficient scheduler.

Figure 3 shows the increase in running time with the size of the input dataset using the new Pthreads scheduler. Recall that we use quicksort to pick the best attribute at a decision tree node; quicksort sorts n records in $O(n \log n)$ time in the expected case. Therefore, the running time grows slightly faster than linearly with the input size.

Space performance

We use the high-water mark of total heap memory allocation (including the memory required to store the input data) as a measure of the memory requirement of the program. This high-water mark is essentially the maximum of the total heap memory allocated by all the threads at any time during the execution of the program. Recall that heap memory is dynamically allocated to store subsets of the original input dataset that filter down to individual tree nodes. Figure 4 shows the variation in the memory requirement of the decision tree builder with the number of processors, for different input sizes. The results indicate that the original Pthreads scheduler requires more memory than the space-efficient scheduler as the problem size increases. Figure 5 shows the variation of the memory requirement using the space-efficient scheduler, as the problem size increases. Both Figures 4 and 5 show that the total memory required by the decision tree builder does not increase with the number of processors.

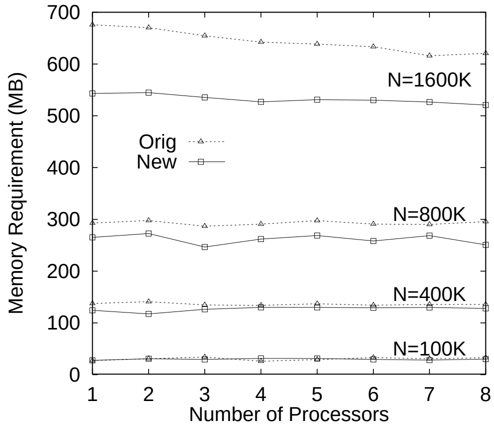

Figure 4: The variation of the memory requirement with the number of processors for different problem sizes.

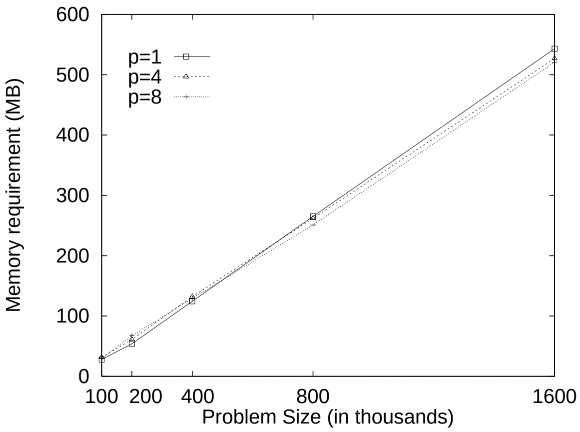

Figure 5: The variation of the memory requirement with the size of the input data, for different numbers of processors.

5 Related Work

In the recent past, there has been a fair amount of research on parallelizing data mining applications in general [11, 34, 19, 2, 4, 8, 5, 9, 24, 38, 10, 23]. However, a fewer number of parallel decision tree builders have been presented [33, 12, 22, 18]. For example, the SPRINT tree building algorithm [33] is designed to efficiently handle out-of-core data. Because sorts on out-of-core data are expensive, it pre-sorts the data at the beginning of the execution rather than at each tree node, and uses a hash table to map a data record to the node it currently belongs to. In the parallel version of SPRINT for distributed memory machines [33], processors synchronously construct each node of the tree. Each of the P processors begins with $1/P$ of the data instances, but the partitioning can soon become highly unbalanced. Explicit load balancing involves significant complexity [39]. Further, SPRINT uses per-processor hash tables, and these hash tables are large near the top of the tree; therefore, the memory requirement of SPRINT increases with the number of processors. Parallel algorithms based on SPRINT have also been proposed for shared memory SMPs [39]. One of these algorithms (MWK) constructs the tree in a breadth-first manner, with pipelined parallelism between nodes at the same level in the tree. However, since data is always accessed level-by-level, the algorithm cannot take advantage of the good cache usage that can be obtained by executing an entire, lower-level subtree on one processor. A second algorithm (SUBTREE) exploits task parallelism by dynamically assigning computations of subtrees to processor groups. In both the algorithms, the partitioning and scheduling is done explicitly by mechanisms specific to the algorithms. In comparison, our algorithm expresses more fine-grained parallelism both across tree nodes and within a node, and relies on the underlying, general-purpose Pthreads implementation to transparently schedule and balance the work.

ScalParC is another parallel decision tree builder designed for handling out-of-core data on distributed memory machines [18]. As with SPRINT, the ScalParC algorithm pre-sorts the data, but uses a distributed hashtable instead of a per-processor hashtable to map records to nodes; this reduces the memory required to store the hash table. To balance the load across processors, the tree is built level-by-level in a breadth-first order. This scheme is well-suited for handling out-of-core data, where disk accesses dominate the total execution time. However, for in-core datasets, good cache locality is important at lower levels of the recursion tree, which cannot be obtained with a breadth-first tree construction. Srivastava *et al.* [12] propose a parallel formulation in which processors work together to synchronously build the nodes in the top few levels of the decision tree; the tree nodes in lower levels are gradually partitioned across groups of processors as the number of data instances filtering down to the nodes decreases. Note that by using our general-purpose, space-efficient thread scheduler, we automatically get a similar implementation while using a simple, high-level program.

6 Summary and Discussion

Decision tree building is a difficult application to parallelize, due to a large amount of data-dependent, dynamic parallelism. We have presented a high-level parallel formulation of Quinlan's C4.5 decision tree building algorithm which uses an underlying, space-efficient implementation of lightweight Pthreads to get good performance. Because the lightweight-threads model allows threads to be dynamically created and destroyed often, we are able to express the dynamic parallelism inherent within C4.5 in a natural manner. The use of lightweight threads also allows us to express a large amount of parallelism, so that the load can be dynamically balanced by the thread implementation. We showed that with an efficient, general-purpose thread scheduler, our high-level code can achieve serial and parallel performance competitive with previous approaches. We feel that this lightweight threads approach can be applied to other data mining approaches, leading to simpler parallel formulations for cost-effective SMPs.

Future work involves adding tree pruning and the ability to handle discrete attributes. Currently we parallelize only the sorts at each tree node. We plan to express more of the parallelism inherent within the work performed at each node; for example, the calculation of the best gain ratio from a sorted sequence can also be parallelized. We feel that an efficient implementation of lightweight threads will also be useful to parallelize decision tree building algorithms designed for out-of-core data; the threads could be used to hide a significant portion of the I/O latency in such problems. We also plan to investigate the potential of applying an efficient implementation of lightweight threads to other data mining techniques.

References

[1] AbTech Corporation. *ModelQuest MarketMiner*^{TM}. `http://www.abtech.com`.

[2] R. Agrawal and J. C. Shafer. Parallel mining of association rules. *Ieee Trans. On Knowledge And Data Engineering*, 8:962–969, 1996.

[3] Rakesh Agrawal, Sakti P. Ghosh, Tomasz Imielinski, Balakrishna R. Iyer, and Arun N. Swami. An interval classifier for database mining applications. In *Proc. 18th Int. Conf. Very Large Databases, VLDB*, pages 560–573, 23–27 August 1992.

[4] S. S. Anand, C. Shapcott, D. Bell, and J. Hughes. Data mining in parallel. In *Proceedings of WoTUG-18: Transputer and occam Developments*, volume 44, pages 113–124, April 1995.

[5] D. W. Cheung, J. Han, V. T. Ng, A. W. Fu, and Y. Fu. A fast distributed algorithm for mining association rules. In *Fourth International Conference on Parallel and Distributed Information Systems (PDIS '96)*, pages 31–43, December 1996.

[6] D. E. Culler and G. Arvind. Resource requirements of dataflow programs. In H. J. Siegel, editor, *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 141–151, Honolulu, Hawaii, May–June 1988. IEEE Computer Society Press.

[7] Digital Equipment Corporation. *Digital UNIX (Version 4.0): New and Changed Features*. Order number: AA-QTLMA-TE.

[8] S. H. Freitas and A. A. Lavington. A data-parallel primitive for high-performance knowledge discovery in large databases. Technical Report Internal Report CSM-242, University of Essex, UK, May 1995.

[9] Gehad Galal, Diane J. Cook, and Lawrence B. Holder. Improving scalability in a scientific discovery system by exploiting parallelism. In David Heckerman, Heikki Mannila, Daryl Pregibon, and Ramasamy Uthurusamy, editors, *Proceedings of the Third International Conference on Knowledge Discovery and Data Mining (KDD-97)*, page 171. AAAI Press, 1997.

[10] S. Goil and A. Choudhary. High performance data mining using data cubes on parallel computers. In *Proceedings of the 1st Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing (IPPS/SPDP-98)*, pages 548–555, Los Alamitos, March30–April3 1998. IEEE Computer Society.

[11] Eui-Hong Han, George Karypis, and Vipin Kumar. Scalable parallel data mining for association rules. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 26(2):277–288, 1997.

[12] Eui-Hong (Sam) Han, Vipin Kumar, and Vineet Singh. Parallel formulations of decision-tree classification algorithms. In *Proc. Int. Conf. on Parallel Processing*, 1998.

[13] S. R. Hedberg. Parallelism speeds data mining. *IEEE parallel and distributed technology: systems and applications*, 3(4):3–6, Winter 1995.

[14] Hewlett Packard. *HP-UX 11.00: An Extended Product Brief.*

[15] IBM. *AIX Version 4.3 General Programming Concepts: Writing and Debugging Programs.*

[16] IBM Datamining Research. *Quest Synthetic Data Generation Code*. Can be downloaded from `www.almaden.ibm.com/cs/quest/syndata.html`.

[17] IEEE. Information Technology–Portable Operating System Interface (POSIX)–Part 1: System Application: Program Interface (API) [C Language]. IEEE/ANSI Std 1003.1, 1996 Edition.

[18] M. Joshi, G. Karypis, and V. Kumar. ScalParC: A new scalable and efficient parallel classification algorithm for mining large datasets. In *Proc. 1st Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing (IPPS/SPDP-98)*, pages 573–579, Los Alamitos, March30–April3 1998.

[19] Bin Li and Dennis Shasha. Free parallel data mining. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 27(2):541–??, 1998.

[20] M. Mehta, R. Agrawal, and J. Rissanen. SLIQ: A fast scalable classifier for data mining. *Lecture Notes in Computer Science*, 1057:18–??, 1996.

[21] D. Michie, D. J. Spiegelhalter, and C. C. Taylor, editors. *Maschine Learning, Neural and Statistical Classification*. Ellis Horwood, Hertfordshire, 1994.

[22] Rakesh Agrawal Mohammed J. Zaki, Ching-Tien Ho. Scalable parallel classification for mining on shared-memory systems. In *Proc. 1st Workshop on High Performance Data Mining*, March 1998.

[23] J. E. Moreira, S. P. Midkiff, M. Gupta, and R. D. Lawrence. Parallel data mining in java. Technical Report RC 21326, IBM T. J. Watson Research Center, 1998.

[24] Andreas Mueller. Fast sequential and parallel algorithms for association rule mining: A comparison. Technical Report CS-TR-3515, Dept. of Computer Science, Univ. of Maryland, College Park, MD, August 1995.

[25] Girija J. Narlikar. *Space-Efficient Multithreading*. PhD thesis, Carnegie Mellon University, School of Computer Science, 1999. To appear.

[26] Girija J. Narlikar and Guy E. Blelloch. Pthreads for dynamic and irregular parallelism. In *CD-ROM Proceedings of Supercomputing '98*. IEEE, November 1998.

[27] G.J. Narlikar. Scheduling threads for low space requirement and good locality, October 1998. Submitted for publication.

[28] M. L. Powell, Steve R. Kleiman, Steve Barton, Devang Shah, Dan Stein, and Mary Weeks. SunOS multi-thread architecture. In *Proceedings of the Winter 1991 USENIX Technical Conference and Exhibition*, pages 65–80, Dallas, TX, USA, January 1991.

[29] Quadstone Limited. *Decisionhouse Product Architecture*. `http://www.quadstone.com`.

[30] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Mateo, CA, 1993.

[31] Jr. R. H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Trans. on Programming Languages and Systems*, 7(4):501–538, 1985.

[32] C. A. Ruggiero and J. Sargeant. Control of parallelism in the manchester dataflow machine. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, pages 1–16. Springer-Verlag, Berlin, DE, 1987.

[33] John C. Shafer, Rakesh Agrawal, and Manish Mehta. SPRINT: A scalable parallel classifier for data mining. In *Proc. 22nd Int. Conf. Very Large Databases, VLDB*, September 1996.

[34] Takahiko Shintani and Masaru Kitsuregawa. Parallel mining algorithms for generalized association rules with classification hierarchy. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 27(2):25–36, 1998.

[35] SiliconGraphics. *MineSet Users's Guide*. Part No.: 007-3214-004.

[36] SiliconGraphics. *Topics in IRIX Programming (IRIX 6.4)*. Part No.: 007-2478-004.

[37] Sun Microsystems. *Multithreaded Programming Guide (Solaris 2.5)*.

[38] Zaki, Parthasarathy, and Ogihara. Parallel algorithms for discovery of association rules. In *Data Mining and Knowledge Discovery*, volume 1. Kluwer Academic Publishers, 1997.

[39] M. J. Zaki, C. T. Ho, and R. Agrawal. Parallel classification for data mining on shared-memory multiprocessors. Technical report, IBM Research Report, 1998.