Strategies for Parallel Data Mining

D.B. Skillicorn
skill@cs.queensu.ca

May 1999
External Technical Report
ISSN-0836-0227-
1999-426

Department of Computing and Information Science
Queen's University
Kingston, Ontario, Canada K7L 3N6

Document prepared May 10, 1999

Abstract

We present a set of cost measures that can be applied to parallel algorithms to predict their computation, data access, and communication performance. These measures make it possible to compare different possible parallel implementation strategies for data mining techniques without the necessity to benchmark each one. We give general cost expressions for three common parallelizing strategies, and show how to instantiate these cost expressions for a particular technique, neural networks.

Keywords: cost measurement, parallelization, complexity, association rules, neural networks, genetic algorithms.

1 Introduction

The role of parallel computing in successful data mining is still an open question. Data mining algorithms are expensive, with high demands for computation and data access. It is plausible that the more sophisticated the algorithm, the greater the commercial advantage, so the demand for computation is likely to grow; the rate of growth of data stored online suggests that the demand for data access is likely to grow as well. Parallel computing seems to have a natural role to play since parallel computers are good at large computations, and also good at handling large volumes of data.

On the other hand, data mining may not turn out to be as hard as it looks. For example, if sampling from a large dataset provides results as good, or almost as good, as those obtained by using the entire dataset, then high-performance computation and data access may be unnecessary. Furthermore, the bottleneck in sequential data mining is access to disk storage. Parallel computers with multiple striped disks have an advantage here, but few parallel computers scale up the performance of their disk access system as fast as their computational power (or their price). The cost/benefit trade-off may not favor parallel computing in the long run.

Parallel computing thus has considerable *potential* as a tool for data mining, but it is not yet completely clear whether it represents the *future* of data mining.

Some idea of how parallel computing is already being used for data mining can be obtained by examining the uses to which the world's largest supercomputers are put. The number of 'industrial' users in the TOP500 list [13] increased from 153 (Nov'97) to 207 (Nov'98). The names of some of the organizations who use these systems suggest that some, at least, are being used for data mining (data mining success stories tend not to be publicized).

Commercial applications in TOP500	
Rank	Owner
64	State Farm
79	Charles Schwab
91	Oracle/IBM
117	Chase Manhattan
119	Sears
173-6	Commerzbank
178-9	Deutsche Morgan Grenfell
206	Prudential
211	Lexis Nexis
224	SMVG Bern

Designing and implementing parallel programs is expensive. Data mining algorithms could be parallelized in many different ways. It is impractical to test all of these by building implementations and comparing them. Fortunately, practical complexity measures for parallel programming are rapidly maturing. We show how to assess different parallelization strategies using such cost measures. The search space of effective parallel data mining algorithms is greatly reduced by using high-level cost analysis.

Section 2 presents different parallelization strategies and discusses their performance intuitively. Section 3 introduces a robust set of cost measures that can be used to predict the performance of an algorithm across platforms. In Section 4 we review some standard data mining algorithms. In Section 5, we give an abstract sequential cost expression for such algorithms. In Section 6, we describe and cost several different techniques for parallelizing data mining algorithms. In Section 7, we give a detailed example of how these techniques work for neural networks, and in Section 8 explore a double speedup phenomenon that occurs in some algorithms.

2 Strategies for parallelizing data mining algorithms

A typical data mining application starts from a dataset describing interactions between customers and an organization. For concreteness, let us suppose that this dataset is a table with n rows and m columns. Each column describes a specific possible interaction ('buying apples') and each row describes a set of interactions that occurred together ('one visit to a supermarket').

The output of the data mining algorithm is some information distilled from this data. The results may be:

- A set of concepts (i.e. predicates) about the interactions ('80% of customers buy apples and tuna on the same visit'). Algorithms that produce concepts are said to be transparent, since the concepts are usually intelligible in organizational terms.
- A set of model parameters. These are typically used to build classifiers that, for example, can distinguish profitable from unprofitable customers. Algorithms of this kind are said to be opaque, since it is not usually clear why the classifier computes the answers it does.

We will speak as if the results of data mining were concepts, but it makes no difference to the conclusions we draw.

The size of the datasets used for data mining can be very large. A dataset might contain information about 10^9 customer interactions, each involving perhaps 1000 attributes.

Parallel data mining requires dividing up the work, so that processors can make useful progress towards a solution as fast as possible. The question is how to do this division of labor. Clearly, it is a good idea to divide up the computation; but it is equally, and sometimes more, important to divide up the accesses to the dataset, and to minimize communication between the processors while they are working. In data mining applications, we want to minimize the resources spent developing concepts that seem to be valid locally, based on the limited amount of data available to each processor, but are not valid globally. A good technique is a balancing act between local, speculative, computation which may turn out to be wasted, and expensive, but reassuring communication.

There are three basic strategies for parallelizing data mining algorithms. They are:

1. **Independent Search.** Each processor has access to the whole dataset, but each heads off into a different part of the search space, starting from a randomly-chosen initial position.

2. **Parallelize a sequential data mining algorithm.** There are two common variants. In the first, the set of concepts is partitioned across processors, and each processor examines the entire dataset to determine which of its local concepts is globally correct. Because generating new concepts usually requires knowing which smaller or simpler concepts are correct, processor must regularly exchange information about their concepts. In the second, the dataset is partitioned by columns, and each processor computes those partial concepts that hold for the columns it can see. Once again, regular exchanges are needed to determine which partial concepts can be fitted together to make globally-correct concepts.

3. **Replicate a sequential data mining algorithm.** Each processor works on a partition of the dataset (by rows) and executes (more or less) the sequential algorithm. Because the information it sees is only partial, it builds entire concepts that are locally correct, but may not be globally correct. We call these *approximate concepts*. Processors exchange these approximate concepts, or facts about them, to check if they are globally correct. As they do so, each learns about the parts of the dataset it cannot see.

There are, of course, many other possibilities and combinations, but these three distinguish the main lines of attack.

Independent search is a good strategy when the desired output is one optimal solution, so it works well for minimization problems. The parallelized approaches have attracted most attention, because applying parallel computing seems to require a parallel algorithm. However, both variants have their problems. The first requires each processor to access the entire dataset. Access to terabyte datasets is slow compared to processor speeds, and growing relatively slower, so we can afford to do a lot of computation to avoid such accesses. The information exchanged between processors also tends to be large; typically the concepts themselves. The second of the parallelized approaches suffers because partial concepts can often not be extended to become complete concepts; many more facts are true about a few attributes than can be true globally, and it is hard to winnow these quickly enough. (Of course, if we knew which attributes 'belonged' together we could assign these to the same processor and get good results – but it is exactly this that we want to discover.)

The replicated approach is not particularly novel, but it is often the best way to add high performance to a data mining application. It has two significant advantages: the dataset is partitioned so the access cost is spread across processors; and the data that must be exchanged between phases is often much smaller than the concepts themselves, so communication is cheap. It is still possible to generate local concepts that do not hold globally, but the fact that they must be internally consistent, as well as consistent with the local subset of the dataset means that this does not happen often.

The results of a data mining algorithm can sometimes be larger than its input dataset (although surely the *interesting* results are not). It follows that the concept set can be extremely large, at least at intermediate stages of the algorithm. When this happens, the replication approach will not perform so well, because it requires each processor to store the whole of the current concept set.

This analysis is based on intuitions about the inherent costliness of communication and data access. We will make these intuitive ideas formal; but first we must define a realistic, but tractable, way of measuring the costs of computation, communication, and data access.

3 Parallel Complexity Theory

Standard parallel complexity theory is based on the PRAM, an abstract architecture with a single shared memory, accessible in constant time. It is the programmer's responsibility to ensure that no two processors access the same location simultaneously. Real architectures must pay some penalty for accessing locations that are distant, whether they are physically shared-memory or distributed-memory. The requirement that programmers prevent interference is too strong so, in practice, architectures allow arbitrary access patterns to appear in programs, and pay some overhead at run-time to prevent conflicting accesses. Both latency costs (the costs of accessing at a distance) and conflict costs (preventing simultaneous accesses) cause large discrepancies between the theoretical costs of the PRAM model and those observed when programs run on real machines.

Furthermore, the PRAM model cannot even act as an approximation to or foundation for a more accurate cost model. It is not, in general, possible to tell when the PRAM model is going to be in error, because it depends on details of the memory access pattern and target architecture's network. Worse still, errors, when they occur, can be polynomial in the problem size (rather than the constant factor errors of sequential complexity theory).

A parallel complexity measure that is correct to within a constant factor is needed. There are two important aspects to choosing such a measure. First, it must take into account costs associated with the memory hierarchy. This issue arises in sequential cost modelling as well, but only for a relatively small subclass of memory-bound or out-of-memory algorithms whose execution time is not dominated by the number of instructions they execute. In a parallel setting, a much greater proportion of programs use significant amounts of memory, so getting this right is more critical.

Second, a measure must accurately reflect the costs of communication, whether explicitly, as in message-passing programs, or implicitly, as in shared-memory programs. What makes costing communication difficult is that it is highly non-linear: the cost of sending a message in an empty network is very different from the cost of sending it in a heavily-loaded network. The reason, of course, is congestion; when a network is busy, a given message has a much greater chance of being blocked by other traffic during its trip. Experimental studies [5] show, however, that congestion in parallel computers almost always happens at the boundaries of the network, rather than in the middle. In other words, the problem is getting into, and even more getting out of, the network. When multiple messages arrive at a processor, it can only extract one of them at a time; the others block back in the network.

Accurate measures of communication performance can be obtained if we assume that the network is always in a heavily-loaded state. The worst non-linearities occur at intermediate loads. For then a message *may* encounter other messages, and take a long time, but it may also be lucky and travel straight through. When the network is busy, all messages encounter other messages, and the effects on individual delivery times average out. If every processor sends messages to random destinations, it is possible to determine expected delivery times with small variance. This forms the basis for a robust measure of communication performance which will be valid as long as communication always takes place when the network is busy; this will be true if processors interleave computation phases with communication phases in rough synchrony. It is convenient to capture network performance in terms of the network's *permeability* (called g), in units of time per byte delivered.

As we mentioned earlier, it is the link between processor and network that is the actual bottleneck. Thus if all processors begin sending data, it will be the processor that sends or receives the most data that will take the longest to finish. If each processor i sends or receives h_i bytes, then an accurate estimate of the cost of a communication phase will be the maximum of the $h_i g$'s.

This assumes that all processors spend all of their time communicating, and not all programs have that behavior. However, many data mining algorithms do, at least approximately. They are structured in a number of phases, each of which involves a local computation, followed by an exchange of data between processors. The cost of such a phase is described by an expression of the form:

$$\text{cost} \;=\; \underset{\text{processors}}{\mathsf{MAX}}\; w_i \;+\; \underset{\text{processors}}{\mathsf{MAX}}\; h_i\, g$$

where w_i is the number of instructions executed by processor i.

This cost model is derived from BSP [11] where its fundamental accuracy has been verified over a wide range of applications. (BSP enforces this alternating computation/communication structure on programs, and works to reduce the effective value of g, but that doesn't affect the basic point.) Notice that both terms are the in the same units, time. This avoids the need to decide how to weight the cost of communication relative to computation, and makes it possible to compare algorithms with different mixes of computation and communication.

We have not yet addressed the issue of compute-bound versus memory-bound computations. The hard situation to model is when the time taken for computation is close to that required for memory access, for then it is hard to predict which will dominate. A detailed cost model is required and the precise interactions of instructions and cache may be critical. In most data mining applications it is straightforward to tell when computation will dominate memory access, and the memory access pattern is typically a single pass through the dataset on each phase whose cost is predictable.

The cost model presented above is likely to produce accurate estimates of running times on existing parallel computers. However, for *comparing* different algorithms, its absolute accuracy is not as important as its relative accuracy. And here we can have high confidence, because the model is based on counting: instructions executed, and bytes communicated. For most data mining applications, different parallelization strategies will execute approximately the same number of instructions. The aspect that separates them is how much communication they do. And clearly an algorithm that communicates more is going to be more expensive regardless of the details of how that communication is costed.

Because the cost model depends only on high-level properties of algorithms, it can be applied to an algorithm in the abstract. So we can compare different parallelization strategies without having to develop different implementations and benchmarking them against each other (provided that the costs are clearly different). In particular, strategies that are likely to be unproductive can be ruled out cheaply.

4 Data Mining Algorithms

Many data mining techniques have been investigated. We will concentrate on three:

- Association rules [12]
- Neural networks [3]
- Genetic algorithms [4]

Other techniques that are often used are decision trees [8], inductive logic programming [7], and singular value decomposition (especially for text, where it is known as latent semantic indexing) [2].

4.1 Association Rules

Association rules were one of the earliest data mining algorithms. Given a dataset, a support s, and a confidence c, the first step of the algorithm is to find all of the *frequent sets*, those subsets of the attributes that appears in at least s of the rows of the dataset.

Using these, rules of the form

$$A, B, C \Rightarrow D$$

are computed from frequent sets $\{A, B, C, D\}$ provided that they have sufficient confidence, that is

$$\frac{\text{support of}\{A, B, C, D\}}{\text{support of}\{A, B, C\}} \geq c$$

Most algorithms make use of the insight that a set can only be frequent if all of its subsets are frequent. The frequent sets can therefore be computed by alternately generating candidate sets of a certain size, checking which of the candidates was in fact frequent by a pass through the dataset, then using the surviving candidates to generate candidates of size one greater. Thus the algorithm goes in phases, each computing a candidate set of size i, and then pruning it using a pass through the dataset.

The output of an association rule algorithm is a set of rules capturing information about how likely it is that certain patterns of attributes occur with other attributes in customer transactions. This is an example of a transparent algorithm.

4.2 Neural networks

In contrast, neural network data mining algorithms are opaque. The result of training a neural network is a black box which is capable of answering interesting questions ("is this person a good candidate for a mortgage") but not of explaining why it gives the answers it does.

A neural net consists of layers of units which sum their inputs and transmit an output if the weighted sum exceeds a threshold. There are many different possible arrangements, but we will assume the most general, that the output of a node in one layer is connected to the inputs of all nodes in the following layer, and that each edge has an associated weight.

Neural networks are trained by presenting each row of the dataset to the inputs, comparing the resulting net output to that desired and using the difference as an error that is propagated back through the network, altering the internal weights. (Again, there are many variants.)

The dataset is usually fed to the network many times; each one is called an *epoch*.

4.3 Genetic algorithms

Genetic algorithms find concepts using a computational analogy to Darwinian evolution. An initial population of concepts is generated randomly. The fitness of each concept is evaluated by how well it describes the dataset. Those concepts that 'survive' (that is, are sufficiently fit) remain in the concept set, where they are replicated according to their fitness, allowed to mutate randomly, and allowed to crossover by exchanging parts of their substructure. The new concept set is then evaluated for fitness, and the process repeats.

The strength of genetic algorithms is that they are not dependent on the problem structure; their weakness is that it is hard to know when to stop the process. In practice, most algorithms seem to stop when there is little change in the concept set, or after some fixed number of iterations.

5 Costing sequential data mining algorithms

The algorithms described in the previous section all have the property that their global structure is a loop, building more accurate or more detailed concepts from those of previous iterations. Suppose that this loop executes k_s times, and generates σ concepts. We can describe the sequential complexity of this algorithm by a formula:

$$\text{cost}_s \;=\; k_s\,[STEP(nm,\sigma) + ACCESS(nm)]$$

where *STEP* gives the cost of a single iteration of the loop, and *ACCESS* is the cost of accessing the dataset once. *STEP* depends on σ as well as nm since it is possible for the set of derived concepts to be larger than the input.

6 Parallel Complexity

We can now construct similar cost expressions for the parallelization strategies discussed above.

6.1 Complexity of Independent Search

The independent search strategy is straightforward: do not partition the data; instead execute the same algorithm p times, using some randomization technique to direct each to a different part of the search space of concepts.

For genetic algorithms, an independent search approach requires running multiple copies of the sequential algorithm from different random starting chromosomes. The best description is selected at the end. This is computational equivalent of evolution to fill equivalent niches.

The cost of an independent search strategy algorithm has the form

$$\text{cost}_i \;=\; k_i\left[STEP(nm, \sigma) + ACCESS(nm)\right] + \sigma pg + \sigma$$

Here k_i is the number of iterations of the whole program, $STEP$ and $ACCESS$ are the costs of the algorithm and its data accesses as before, σpg is the cost of sharing the answers among the processors at the end, and σ is the cost of computing the best solution. It is clear that this approach only makes sense if we have a reason to expect that $k_i \leq k_s/p$, which is characteristic of searching for a single optimum.

6.2 Complexity of Parallelized Algorithms

The basic structure of these algorithms is:

- Partition the initial concepts into p subsets.
- Repeat
  - Execute a special variant of a data mining algorithm on the entire dataset or perhaps a segment of it, and the current partial set of concepts, to derive a new partial set of concepts.
  - Exchange the partial concepts with other processors, deleting concepts that are not globally correct.

For genetic algorithms, a partial concept parallelization approach divides the chromosomes into pieces and assesses the fitness of each piece against the relevant features (columns) of the dataset. The total fitness of a chromosome depends on the fitness of its pieces.

For association rules, the partial concept parallelization approach is called the Data Distribution technique. The possible frequent sets are partitioned across the processors, and the entire dataset is examined by each processor. (For practical reasons, the dataset is usually divided into p pieces which are circulated by each processor in turn.) A partitioned-concept parallelization approach called Candidate Distribution has also been investigated.

The cost of a parallelized algorithm has the form

$$\text{cost}_p = k_p\left[SPECIAL(n, m, r/p) + ACCESS(n, m) + EXCH(n, m, r, p)g + RES(n, m, r, p)\right]$$

when the concept set is partitioned, and

$$\text{cost}_p = k_p\left[SPECIAL(n, m/p, r) + ACCESS(n, m/p) + EXCH(n, m, r, p)g + RES(n, m, r, p)\right]$$

when the concepts themselves are partitioned. *SPECIAL* is the complexity of a single step of the special algorithm, *EXCH* is the cost of exchanging the partial concepts, and *RES* is the cost of resolving the partial concepts or partial concept set into a consistent set.

6.3 Complexity of Replicated Algorithms

The basic structure of these algorithms is:

- Partition data into p subsets, one per processor.
- Repeat
  - Execute (some variant of) the sequential algorithm on each subset.
  - Exchange information about what each processor learned with the others.

For the frequent set part of association rule computation this means: partition the dataset among the processors; compute candidate sets locally and measure their support in the local partition; exchange these support values and compute the total support for each candidate; and repeat. Notice that each processor keeps the same candidate sets and replicates the computation of new candidate sets from old; but the volume of data exchanged is very small – integers.

A genetic algorithm replicated implementation has a similar structure. Each processor has a subset of the dataset and a full set of the current concepts. Each measures the local fitness of the concepts against its partition, and exchanges this data with all of the other processors to compute the global fitness. The next generation of concepts is computed based on this universally-known fitness (perhaps requiring some small data exchanges if crossover is permitted across processors).

The cost of a replicated algorithm has the form

$$\text{cost}_r = k_r \left[STEP(nm/p, r) + ACCESS(nm/p) + rpg + RES(rp) \right]$$

where k_r is the number of iterations required by the parallel algorithm, r is the size of the data about approximate concepts generated by each processor, rpg is the cost of a total exchange between the processors of these approximate concepts, and $RES(rp)$ is the computation cost of using these approximations to compute better approximations for the next iteration.

It is reasonable to assume that

$$STEP(nm/p, r) = STEP(nm, r)/p$$

and

$$ACCESS(nm/p) = ACCESS(nm)/p$$

so we get

$$\text{cost}_a \;\approx\; \text{cost}_s/p + k_r(rpg + RES(rp))$$

In other words, we get a p-fold speedup, except for an overhead term, provided k_s and k_r are of comparable size.

Exchanging results frequently often improves the rate at which concepts are derived so for some algorithms $k_r \ll k_s$. This gives a "double" speedup

$$\text{cost}_a \;=\; \frac{k_r}{k_s}\frac{\text{cost}_s}{p} + \text{overhead}$$

We will discuss this phenomenon in Section 8.

6.4 Comparisons

The relative costs of these different performance improvement strategies depends on:

- the number of the iterations of the basic loop structure (the relative size of the ks);
- the amount of data that must be accessed during each iteration;
- the amount of communication that must take place among the processors.

It is hard to be dogmatic about the number of iterations required in general. For some algorithms, it will be the case that $k_r \approx k_p$. Thus there may not be much to choose between parallelized and replicated algorithms on this basis. The relative performance of independent search is sensitive to problem structure; for minimization problems k_i may be much smaller than any of the other ks; for other problems it may be that $k_i \approx k_s$.

For data access, independent search suffers from the drawback that the entire dataset must be accessed by each processor. Parallelized algorithms have the potential to require access to all n rows, but only m/p columns of each, the same nm/p access that replicated algorithms require. However, all of the parallelized algorithms known to me require much more data than this, usually all nm values. This makes parallelized algorithms much more expensive.

For communication, independent search is a clear winner since only a single global communication is required at the end to determine the best solution. Both parallelized and replicated algorithms communicate at the end of every phase. However, parallelized algorithms tend to have to transmit (parts of) concepts while replicated algorithms tend to have to transmit only facts *about* concepts which are much smaller. For the same reason, the resolution between concepts required by replicated algorithms is typically less work than for parallelized algorithms.

Thus, although it is not possible to say that one strategy is the best overall, there is a strong tendency for replicated strategies to be better than parallelized strategies. The high-level cost expressions for each strategy can be easily instantiated for particular data mining techniques to give a more refined basis for decision.

For example, several of these strategies have been implemented for computing association rules. The replicated approach is called Count Distribution [1]. It has been shown to outperform two other techniques provided that the candidate sets can be kept in memory: Data Distribution, in which the candidate set is partitioned, and the dataset circulated among the processors; and Candidate Distribution, in which the candidate set and dataset are both partitioned. The reasons are exactly those that the cost expressions make plain: the two poorer techniques require much greater data access and communication than Count Distribution.

However, frequent set calculation is one of those cases where the intermediate results can be larger than the dataset (since the frequent sets are elements of the powerset of the attributes). Scalability therefore becomes an issue. Count Distribution has the drawback that it requires all of the candidates to be stored at every processor.

In the next section, we illustrate the use of cost expressions by doing a detailed analysis of the relative costs of neural net training using different performance improvement techniques.

7 Detailed Example – Neural networks

To make these cost expressions more concrete, we turn now to neural networks, where we will display more detailed cost expressions and compare them. It becomes very clear that replicated approaches are much more effective than the others for data mining algorithms with these general characteristics [10].

Consider a neural network with l layers, and m neurons per layer, with full connections from each layer to the next and preceding layers. The number of weights (one per connection) is $W = lm^2$. The number of examples in the dataset is n.

A replicated approach to learning is *exemplar parallelism* – each processor trains an identical initial network on a pth fraction of the examples. At the end of each epoch, that is when all rows of the dataset have been processed by some processor, processors exchange their error vectors and combine them (deterministic learning). This continues for sufficient epochs to ensure convergence.

The cost for a single training epoch is:

$$C_{EP} \;=\; n(AW)/p + W(p-1)g$$

where A is a constant that depends on the particular training algorithm. The first term is the cost of computation: a constant number of weight adjustments for each row of the dataset (a term of size nW), divided equally among the processors. The second term is the cost of communication: the total exchange of sets of errors (one per weight) among the processors.

Parallelized approaches have each processor responsible for some subset of the neurons. A natural starting place is to assume that each processor is responsible for some rectangular block of neurons. Even very simple analysis of this possibility makes it clear that blocks that are the full width or depth of the neural network save communication. The cost when the network is divided into layers (*layer parallelism*) is

$$C_{LP} \;=\; [(n-1) + 2(l-1)]\left[\frac{AW}{p} + 2mg\right]$$

for a single epoch. When a network is divided into columns (*column parallelism*), the cost is

$$C_{CP} \;=\; n\left[\frac{AW}{p} + (2m + (m - \frac{m}{p})(l-1))g\right]$$

Layer parallelism is a form of concept set partitioning, while column parallelism is direct concept partitioning, since each processor is responsible for a subset of the inputs.

Notice that the computation term is about the same magnitude for all of these approaches as for exemplar parallelism (except for an added term which arises from the initial filling and emptying of the pipeline when there are layers). However, the communication term now contains a factor of size n, which is far the biggest term present in data mining applications. As a result, these techniques perform poorly even for quite small datasets.

An alternative approach is to allocate neurons randomly to processors. Not surprisingly, this approach is worse than either of those above. Its cost is

$$C_{NP} \;=\; [(n-1) + 2(l-1)]\left[\frac{AW}{p} + \frac{2Wl}{p}g\right]$$

The argument here is an information-theoretic one, and so is not sensitive to variations in the precise neural net training technique, nor to architectural variations. The only exception are architectures for which the assumption about communication costs do not hold. For example, Kumar, Shekhar, and Amin [6] give a neural net parallelization that exploits carefully-scheduled, point-to-point communication in a mesh and is faster than the layer parallelism expression above would suggest. However, such techniques require special-purpose hardware, rather than the off-the-shelf hardware that is the norm today.

There have been many proposals for neural net parallelization that use even finer partitioning, including even partitioning the edge set across processors. It is clear from the formulae above that this can only increase the cost of communication.

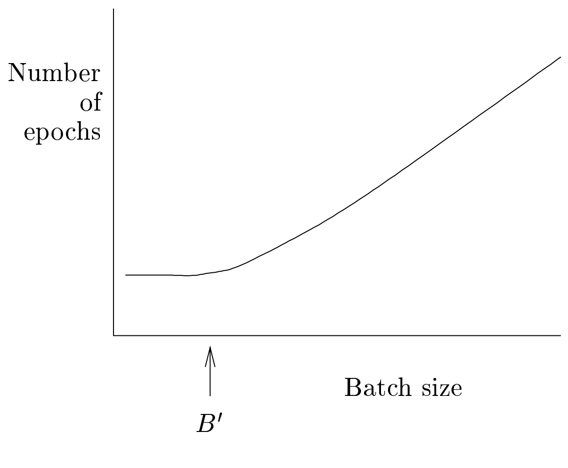
8 Double Speedup

The interesting phenomenon of double speedup occurs in exemplar parallel training of neural nets. Each processor learns, in a condensed way, what every other processor has learned from its data, whenever communication phases take place. This information has the effect of accelerating its own learning and convergence. The overall effect is that k_r is much smaller than k_s would have been, and this in turn leads to a double speedup.

We have described exemplar parallelism so far as if it used deterministic learning, where an error vector is generated only after the entire dataset has been seen by some processor. There is an alternative, batch learning, in which error vectors are computed and exchanged, after some subset, called a batch, of the entire dataset has been processed. Notice that each processor sees only a pth fraction of each batch.

If each batch is sufficiently large, the error vectors computed by each processor are good approximations to the deterministic error vector, but they have required much less processing to discover. The result is that each processor makes progress based on the work of *all* processors during the processing of a batch.

The number of epochs required to achieve a given level of convergence as a function of batch size has the shape shown below:



As batch sizes get smaller, the total number of epochs required for convergence gets smaller as each processor gets error information more frequently. When the batch size reaches some size bound, however, the error information becomes less accurate and hence less helpful, and the effect disappears. This general behavior has been verified experimentally for a number of datasets [9].

Let B be the batch size and $b = n/B$ be the number of batches in each epoch. The total cost of exemplar parallelism training for E epochs is

$$C_{EP}^{\text{total}} = E\left[\frac{nAW}{p} + b(p-1)Wg\right] \qquad (1)$$

In the range where convergence depends linearly on B, we can write $E = c/b$, for some constant c and get:

$$C_{EP}^{\text{total}} = \frac{c}{b}\frac{nAW}{p} + c(p-1)Wg \qquad (2)$$

Minimizing the overall cost requires minimizing the computation term, by making b as large as possible, that is $b = n/B'$. The value of this b on the denominator is often $\sim$20, so speedups are significant even when p is small.

Note that it does not follow that, because an application exhibits double speedup, it will necessarily be solvable by sampling. The information that, shared among processors, improves overall completion may not be accurate; it need only be helpful. It will often be of the character of a hint rather than an answer; but hints, as we know, can often reduce searches dramatically.

9 Conclusions

The best way to reap the benefits of the performance of parallel computers for data mining problems is not obvious. Implementing many different parallel variants of each data mining algorithm and benchmarking them is an expensive and unattractive way to decide which approach is best. We have shown that cost measures based on counting computations, data accesses, and communication allow algorithms to be compared in the abstract. While these cost measures cannot be expected to be completely accurate, they are expressive enough to rule out some possibilities.

We have developed cost expressions for three general implementation strategies and suggested that replication is both the simplest and likely to be the best performing. We have illustrated how these general cost expressions can be instantiated for particular data mining techniques, using the example of neural networks. Here it is very clear that replicated implementation outperforms other parallelized techniques.

References

[1] R. Agrawal and J. Shafer. Parallel mining of association rules: Design, implementation and experience. Technical Report RJ10004, IBM Research Report, February 1996.

[2] M.W. Berry, S.T. Dumais, and G.W. O'Brien. Using linear algebra for intelligent information retrieval. *SIAM Review*, 37(4):573–595, 1995.

[3] C. Bishop. *Neural networks for pattern recognition.* Oxford University Press, 1995.

[4] D.E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, 1989.

[5] Jonathan M. D. Hill, Stephen Donaldson, and David Skillicorn. Stability of communication performance in practice: from the Cray T3E to networks of workstations. Technical Report PRG-TR-33-97, Oxford University Computing Laboratory, October 1997.

[6] V. Kumar, S. Shekhar, and M. Amin. A highly parallel formulation of backpropagation in hypercubes. *IEEE Transactions of Parallel and Distributed Systems*, 5, No.10:1073–1091, October 1994.

[7] S. Muggleton. Inverse entailment and Progol. *New Generation Computing Systems*, 13:245–286, 1995.

[8] J.R. Quinlan. *C4.5: Programs for Machine Learning.* Morgan-Kaufmann, 1993.

[9] R.O. Rogers. Data mining with parallel neural networks: Bulk synchronous parallelism and the optimality of batch learning. Master's thesis, Department of Computing and Information Science, Queen's University, 1997.

[10] R.O. Rogers and D.B. Skillicorn. Using the BSP cost model for optimal parallel neural network training. *Future Generation Computer Systems*, 14:409–424, 1998.

[11] D.B. Skillicorn, J.M.D. Hill, and W.F. McColl. Questions and answers about BSP. *Scientific Programming*, 6(3):249–274, 1997.

[12] H. Toivonen. Discovery of frequent patterns in large data collections. Technical Report A-1996-5, Department of Computer Science, University of Helsinki, 1996.

[13] Top500 list. In *Supercomputing '98*, November 1998. `www.netlib.org/benchmark/top500.html`.