David Skillicorn
Queen's University

Strategies for Parallel Data Mining

This article presents 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 parallel implementation strategies for data-mining techniques without benchmarking each one.

One definition of data mining is the deep understanding of variation in large data sets. At its simplest, data mining is concerned with finding the variations in complex data. At its most difficult, it is concerned with finding the most convincing or most useful hypothetical explanations for observed variations. These explanations can be *opaque*—providing a means of classifying variation without any explanation of why it works—or *transparent*—describing what creates the observed variation.

For two reasons, data mining could be the killer application that parallel computing has been seeking. First, analyzing variation appears to be algorithmically complex and hence might require levels of computing power that only parallel computers can provide in a timely way. (See the sidebar, "Why parallelize data mining?") Second, the data sets involved are large and rapidly growing larger, and parallel computers are organized to handle such large volumes effectively, although some data-mining problems are already taxing their limits.

Data-mining algorithms can potentially be parallelized in many different ways. Because designing and implementing parallel programs is expensive, it is impractical to test all of these by building implementations and comparing them. Fortunately, practical complexity measures for parallel programming are rapidly maturing.

This article shows how to use such measures to assess different parallelization strategies for data-mining algorithms. We also show how the structure of some algorithms results in a double-speedup phenomenon. A high-level cost analysis greatly simplifies the search for an effective algorithm.

Strategies for parallelization

A typical data-mining application starts from a data set describing interactions between individual units and a central multifaceted unit. The individuals might be customers and the central unit a sales organization; the individuals might be fabricated objects and the central unit a test rig; or the individuals might be radiation sources and the central unit a telescope. The essential requirement is that the individual units exhibit *variation* and the central unit measures this variation. The data miner wants to understand the deeper processes that create this variation—to better market products in the first case, to increase production yield in the second, and to discover something about the universe in the third. Often this means clustering the individuals into groupings that are more homogeneous than the entire set, and sometimes this involves classification based on the clustered structure.

1092-3063/99/$10.00 © 1999 IEEE

Why parallelize data mining?

Data-mining applications fall into two groups based on their intent. In some applications, the goal is to find explanations for the most variable elements of the data set—that is, to find and explain the *outliers*. In other applications, the goal is to understand the variations of the majority of the data set elements, with little interest in the outliers. Scientific data mining seems to be mostly of the first kind ("find the unexpected stellar object in this sky sweep"), whereas commercial applications seem to be of the second kind ("understand the buying habits of most of our customers"). In applications of the first kind, parallel computing seems to be essential. In applications of the second kind, the question is still open because we do not know how effective sampling from a large data set might be at answering broader questions. 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.

We can obtain some idea of how parallel computing is already being used for data mining by examining the uses to which the world's largest supercomputers are put. The number of "industrial" users in the TOP500 list of the world's most powerful installed computers (http://www.netlib.org/benchmark/top500.html) increased from 153 in November 1997 to 207 in November 1998. A look at a few of the names of the organizations on this list (see Table A) suggests that some, at least, are using supercomputers for data mining. Data-mining success stories tend not to be publicized.

Table A. Some commercial applications in the TOP500 list.

Computer ranking list (by computational power)	Owner
64	State Farm
79	Charles Schwab
91	Oracle/IBM
117	Chase Manhattan
119	Sears
173–176	Commerzbank
178–179	Deutsche Morgan Grenfell
206	Prudential
211	Lexis Nexis
224	SMVG Bern

For example, a bank wants first to cluster its customers into good and bad credit risks and then to use knowledge of what these clusters are like to decide whether or not to approve loans to new customers. For concreteness, let's suppose that a typical data set is a table with n rows and m columns. Each of the n rows describes an individual unit; each of the m columns describes a facet of the interactions with the central unit; and each entry in the table describes the result of an interaction between an individual and the central unit. For a sales organization, the rows might be customer visits; the columns, products; and the entries, the number of each product sold to each customer.

The output of the data-mining algorithm is information distilled from this data in one of the following forms:

- A set of *concepts* (that is, predicates) about the interactions ("80% of customers buy apples and tuna on the same visit"). We consider algorithms that produce concepts transparent, because the concepts are usually intelligible in organizational terms.
- A set of *model parameters*. We typically use these to build classifiers that do things such as distinguishing profitable from unprofitable customers. We consider algorithms of this kind opaque, because it is not usually clear why the classifier computes the answers it does.

We will speak as if all the results of data mining were concepts, but it makes little difference to the conclusions we draw.

The data sets used for data mining can be very large. A data set might contain information about $n = 10^9$ customer interactions, each involving perhaps $m =$ 1,000 attributes.

Parallel data mining requires dividing up the work so that processors can make useful progress toward a solution as fast as possible. The essential question is how to divide the labor.

There are three components to the work: computation, access to the data set, and communication between the processors. In modern parallel computers, access to the data set is likely to be most costly, followed by communication, with computation being relatively cheap. The cost of computation is decreasing much faster than the other two costs, which increasingly dominate achievable performance. These three components are in mutual tension: dividing up the computation to make it faster creates more communication and often more data set accesses as well.

Finding the most effective parallel algorithm requires carefully balancing the three kinds of operations and their costs. Parallel data-mining algorithms often do this by having each processor compute concepts that are valid locally, and then using communication to decide whether the concepts are valid globally. The local computations are therefore speculative. Allowing the amount of speculation to grow increases the ratio of computation to communication. On the other hand, some speculation will turn out to be wrong and hence waste computational resources. Speculation also often requires extra data set accesses. A good parallel technique is a balancing act between local, speculative computation—which may turn out to be wasted—and expensive, but reassuring, communication.

We distinguish three basic strategies for parallelizing data-mining algorithms:

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

- *Parallelized sequential data-mining algorithm.* Each processor restricts itself to generating a particular subset of the set of possible concepts. There are two variants: In the first, each processor generates complete concepts, but with restrictions on the variable values in some positions. Validating such concepts means examining only a subset of the rows of the data set. In the second, each processor generates partial concepts—concepts involving some subset of the variables—but the variables can take any values. Validating such concepts requires examining a subset of the columns.
- *Replicated sequential data-mining algorithm.* Each processor works on a partition of the data set (by rows) and executes what is more or less the sequential algorithm. Because each processor sees only partial information, it builds entire concepts that are locally, but possibly not globally, correct. We call these *approximate concepts*. Processors exchange approximate concepts, or facts about them, to check their global correctness. As they do so, each learns about the parts of the data set it cannot see.

There are, of course, many other possibilities and combinations, but these are the three 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.

Parallelized approaches try to reduce both the amount of memory each processor uses to hold concepts and the fraction of the data set that each processor must access. However, they tend to be unsuccessful on both counts. Because the number of potential concepts is huge, large concepts are often best generated from smaller ones known to be valid. Generating new concepts, therefore, requires processors to exchange information on their existing valid concepts, a bandwidth-intensive operation that requires significant spare memory at each processor, at least temporarily. When the algorithm uses whole-concept partitioning, it cannot always easily tell which rows of the data set it needs to validate each concept, so each processor typically accesses the entire data set anyway. When the algorithm uses partial concepts, each processor needs to see only certain columns. On the other hand, partial concepts are harder to assemble into globally valid whole concepts, so this approach is inherently more speculative. (Of course, if we knew which attributes "belonged" together, we could assign them to the same processor and get good results—but this is exactly what we want to discover.)

The replicated approach is not particularly novel, but it is often the best way to increase performance in a data-mining application.

The replicated approach is not particularly novel, but it is often the best way to increase performance in a data-mining application. It has two significant advantages: First, it necessarily partitions the data set and so spreads the access cost across processors. Second, the data that must be exchanged between phases, typically frequencies, is often much smaller than the concepts themselves, so communication is cheap. This technique can, however, generate local concepts that do not hold globally, but because the concepts must be internally consistent, as well as consistent with the local subset of the data set, this does not happen often.

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

So far, this analysis has been based on intuitions about the inherent costliness of communication and data access. Now let's make these intuitive ideas formal. To do this, we must define a realistic but tractable way of measuring the costs of computation, communication, and data access.

Parallel complexity theory

Standard parallel complexity theory is based on the Parallel Random-Access Machine, an abstract architecture with a single shared memory, accessible in constant time. In the PRAM model, the prgrammer must ensure that no two processors access the same location simultaneously. This theory does not reflect real-world costs well. Real architectures, whether they have a physically shared or distributed memory, pay some penalty for accessing distant locations. Also, the requirement that programmers prevent interference is too strong. In practice, architectures allow arbitrary access patterns to appear in programs and pay some overhead at runtime to check for and prevent conflicting accesses. Costs for both latency (accessing at a distance) and conflict (preventing simultaneous accesses) cause large discrepancies between the PRAM model's theoretical costs and those observed when programs run on real machines.

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

What we need is a parallel complexity measure that is correct to within a constant factor. Such a measure must account for three factors:

- time for computation by each processor,
- time for memory access by each processor, and
- time for communication among the processors.

The flaw in the PRAM model is that it only accounts for the first of these costs. The issue of memory access arises in sequential cost modeling, 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 becomes more important. A simple way to take the cost of memory access into account is to multiply the number of words read from memory during each program phase by a transfer time per word, r.

What makes costing communication difficult is that it is highly nonlinear: the cost of sending a message in an empty network is very different from that 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 show, however, that congestion in parallel computers almost always happens at the boundaries of the network, rather than in the middle.[1] 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.

We can obtain accurate measures of communication performance if we assume that the network is always heavily loaded. The worst nonlinearities occur at intermediate loads, because then a message *might* encounter other messages and take a long time, but it might 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, we can determine expected delivery times with small variance. This forms the basis for a robust measure of communication performance that will be valid as long as communication always takes place when the network is busy. This, in turn, will be true if processors interleave computation phases with communication phases in rough synchrony. Network performance under heavy load can be captured by the network's *permeability* (called g).

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

> Experimental studies show that congestion in parallel computers almost always happens at the boundaries of the network, rather than in the middle.

Suppose we express the transfer time from memory in units of instruction execution times per word transferred, and the network permeability in units of instruction execution times per word moved. Then all three terms—computation time, memory access time, and communication time—are in the same units. We could simply compare algorithms using each of these three values, but we can conveniently join them in a single cost expression. Slightly surprisingly, taking the sum of the three values provides the best prediction of actual execution cost. The cost expression becomes

$$\text{cost} = \max_{\text{processors}} (w_i + a_i r) + \max_{\text{processors}} h_i g.$$

Here, w_i is the number of instructions executed by processor i, and a_i is the number of memory accesses it makes. The first term captures the cost of the computation and memory access, while the second captures the cost of the intervening communication. A sum is the best form for this expression because many programs have an explicit phased structure: a processor reads a block of data from memory, performs a complex calculation on that data, and shares result with other processors. This structure is especially common in data-intensive applications. Variations in the cost expression's structure expression to reflect, for example, overlapped fetch and compute, result in a difference in cost of at most a factor of three. More importantly, these variations tend not to affect the *relative* cost of different algorithms, our focus of interest.

This form of cost model has its origins in the Bulk Synchronous Parallelism programming model,[2] where researchers have verified its fundamental accuracy over a wide range of applications.[1,3-5] The BSP cost model is not a theoretical construct but a practical cost model whose predictions on most programs with phased structures (including most data-mining algorithms) fall within a few percent of actual observed costs.

The cost model we've presented is likely to produce accurate estimates of data-mining algorithms' 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 great confidence, because the model's basis is counting instructions executed, memory locations accessed, and bytes communicated: it is hard to see how a model could do better. For most data-mining applications, different parallelization strategies execute approximately the same number of instructions. What separates them is how much communication and data access they do. Clearly an algorithm that communicates more will be more expensive regardless of the details of how we cost that communication.

Of course, a detailed understanding of system issues is sometimes important to understanding a program's behavior. However, as we shall see, the distinctions

Data-mining algorithms

Researchers have investigated many data-mining techniques; we'll cover a sample of three here:

- association rules,[1]
- neural networks,[2] and
- genetic algorithms,[3]

Other frequently used techniques are decision trees,[4] inductive logic programming,[5] and singular value decomposition (especially for text, where it called latent semantic indexing).[6]

Association rules

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

The algorithm computes rules of the form

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

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

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

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

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.

Neural networks

In contrast, neural-network data-mining algorithms are opaque. The result of training a neural network is a black box 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 that 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.

We train a neural network by presenting each row of the data set to the inputs, comparing the resulting net output to the desired output and using the difference as an error that propagates back through the network, altering the internal weights. Again, there are many variants.

We usually feed the data set to the network many times; each time is called an *epoch*.

Genetic algorithms

Genetic algorithms find concepts using a computational analogy to Darwinian evolution. The algorithm randomly generates an initial population of concepts. Then, it rates the fitness of each concept by how well it describes the data set. Concepts that "survive" (that is, those that are sufficiently fit) remain in the concept set, where they replicate according to their fitness, mutate randomly, and cross over by exchanging parts of their substructures. The algorithm then evaluates the new concept set for fitness, and the process repeats.

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

References

1. H. Toivonen, *Discovery of Frequent Patterns in Large Data Collections*, Tech. Report A-1996-5, Dept. Computer Science, Univ. of Helsinki, 1996.
2. C. Bishop, *Neural Networks for Pattern Recognition,* Oxford Univ. Press, Oxford, England, 1995.
3. D.E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, Reading, Mass., 1989.
4. J.R. Quinlan, *C4.5: Programs for Machine Learning,* Morgan-Kaufmann, San Francisco, 1993.
5. S. Muggleton, "Inverse Entailment and Progol," *New Generation Computing Systems,* Vol. 13, 1995, pp. 245–286.
6. M.W. Berry, S.T. Dumais, and G.W. O'Brien, "Using Linear Algebra for Intelligent Information Retrieval," *SIAM Rev.*, Vol. 37, No. 4, Dec. 1995, pp. 573–595.

between different parallelization strategies are usually quite large; it would be a pathological architecture indeed that would alter these distinctions.

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

Costing sequential data-mining algorithms

The algorithms described in the "Data-mining algorithms" sidebar all have the property that their global structure is a loop, building more accurate or more complex 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 the formula

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

where *STEP* gives the cost of a single iter-

ation of the loop, and *ACCESS* is the cost of accessing the data set once (say, $nm \times r$, the data transfer rate given earlier). We must make the dependence of *STEP* on σ explicit, because it is possible for the set of derived concepts to be larger than the input.

Parallel complexity

We can construct similar cost expressions for each of the parallelization strategies discussed earlier.

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 processor 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 algorithm selects the best description at the end. This is the computational equivalent of evolution to fill equivalent niches.

The cost of an independent-search strategy algorithm has the form

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

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. Clearly, this approach only makes sense if we have reason to expect that $k_i \leq k_s/p$, which is characteristic of searching for a single optimum.

Complexity of parallelized algorithms

The basic structure of these algorithms is first to partition the initial concepts into p subsets and then to repeat the following steps:

- Execute a special variant of a data-mining algorithm on the entire data set, 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 data set. A chromosome's total fitness depends on the fitness of its pieces.

For association rules, the partial-concept parallelization approach is called the Data Distribution technique. This technique partitions the possible frequent sets across the processors, and each processor examines the entire data set. (For practical reasons, the algorithm usually divides the data set into p pieces that circulate by each processor in turn.) A partitioned-concept parallelization approach called Candidate Distribution has also been investigated.

When the concept set is partitioned, the cost of a parallelized algorithm has the form

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

where σ is the size of the concept set.

When the concepts themselves are partitioned, the cost of a parallelized algorithm has the form

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

In these expressions, *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.

Complexity of replicated algorithms

The basic structure of these algorithms is first to partition data into p subsets, one per processor, and then to repeat the following steps:

- 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 that the algorithm partitions the data set among the processors; computes candidate sets locally and measures their support in the local partition; exchanges these support values and computes the total support for each candidate; and repeats. 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 data set 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 the other processors to compute the global fitness. The algorithm computes the next generation of concepts based on this universally known fitness, perhaps requiring some small data exchanges if it permits crossover across processors.

The cost of a replicated algorithm has the form

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

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

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

since it is usually easy to divide the computational work and data access evenly across processors. So we get

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

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 often improves the rate at which an algorithm derives concepts, so for some algorithms, k_r is much less than 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 later on.

Comparisons

The relative costs of these different performance improvement strategies depend on

- the number of iterations of the basic loop structure (the relative size of the ks),
- the amount of data the processors must access during each iteration, and
- 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, $k_r \approx k_p$. That is, the replicated and parallelized versions require roughly the same number of iterations of the basic loop. Thus, there may not be much reason to choose between and replicated and parallelized algorithms on this basis. The relative performance of independent search is sensitive to the problem structure. For minimization problems k_i may be much smaller than any of the other ks; for other problems it may be about the same as k_s, the number of iterations required by a sequential algorithm.

For data access, independent search suffers from the drawback that each processor must access the entire data set. 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 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 because it requires only a single global communication 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 that replicated algorithms require is typically less work than for parallelized algorithms.

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.

Researchers have implemented several of these strategies for computing association rules. The replicated approach is called Count Distribution.[6] Provided the system can keep the candidate sets in memory, Count Distribution outperforms two parallelized techniques: Data Distribution, which partitions the candidate set and circulates the data set among the processors; and Candidate Distribution, which partitions both the candidate set and the data set. 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 situations where the intermediate results can be larger than the data set (because the frequent sets are elements of the powerset of the attributes). Scalability, therefore, becomes an issue. Count Distribution's drawback is that it requires every processor to store all the candidates.

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. We can easily instantiate each strategy's high-level cost expressions for particular data-mining techniques to give a more refined basis for decision.

Detailed example: neural networks

To make our conclusions more concrete, we turn now to neural networks, for which we will compare more detailed cost expressions. It becomes very clear that replicated approaches are much more effective than the others for this kind of data-mining algorithm.[7]

Consider a neural network with l layers, m neurons per layer, and 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 data set is n.

A replicated approach to learning is *exemplar parallelism*—each processor trains an identical initial network on $1/p$ of the examples. At the end of each epoch—that is, when some processor has processed each row of the data set—processors exchange their error vectors and combine them. We call this *deterministic learning*. This continues for sufficient epochs to ensure convergence.

The cost for a single training epoch is

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

Here, 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 data set (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 make each processor responsible for some subset of the neurons. A natural starting place is to assume that each processor takes responsibility 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. When we divide the network into layers (*layer parallelism*), the cost for a single epoch is

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

When we divide a network 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.

For both of these approaches, the computation term is about the same magnitude as it is for exemplar parallelism, except for an added term that 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, by far the biggest term present in data-mining applications. As a result, these techniques perform poorly even for quite small data sets.

An alternative approach is to allocate neurons randomly to processors (*neuron parallelism*). Not surprisingly, this approach is worse than either layer or column parallelism. Its cost is

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

The argument about required communication is an information-theoretic one, and so is not sensitive to variations in the precise neural-net training technique or to architectural variations. The only exceptions are architectures for which the assumptions about communication costs do not hold. For example, Vipin Kumar, Shashi Shekhar, and Minesh Amin[8] define a neural net parallelization that exploits carefully scheduled, point-to-point communication in a mesh and is faster than our layer parallelism expression would suggest. However, such techniques require special-purpose hardware rather than the off-the-shelf hardware that is the norm today.

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.

Many proposals for neural net parallelization use even finer partitioning, some even partitioning the edge set across processors. However, our formulae make it clear that this can only increase the cost of communication.

Double speedup

Several data-mining techniques exhibit a double-speedup phenomenon. We might expect a parallel implementation using p processors to take time t_s/p if it could be arranged without overheads—with perfect speedup, in other words. But some parallel data-mining implementations execute even faster. This does not happen because of some idiosyncratic architectural effect, but because of fundamental properties of the algorithms.

To exhibit double speedup, a data-mining algorithm must be able to use the information it obtains from one phase to reduce the work required by subsequent phases. Suppose that the first phase of an algorithm requires work (computation) w, but that the work in subsequent phases can be reduced by a multiplicative factor α. Then a sequential algorithm has a computation cost of the form

$$(1 + \alpha + \alpha^2 + \alpha^3 + \alpha^4 + \ldots)w.$$

A parallel algorithm using, say, four processors that exchanges information after each (parallel) phase, takes less time overall. The first parallel phase takes time w, but the second phase takes only time $\alpha^4 w$, and so on, provided that the reduction is independently additive. This reduction is a function of w, which in turn is a function of the size of the data set. Such a reduction can be large even when p is quite small.

Data-mining algorithms that satisfy these conditions include those with the goal of constructing a set of concepts that "covers" the whole data set. In other words, every element of the data set is explained by one concept. Once the algorithm has found an explanation for part of the data set, it does not need to examine that part again, so there is less work for the next phase. Yu Wang, in an unpublished work, has verified this phenomenon for inductive logic data mining in the Progol style.

Another type of data-mining algorithm where double speedup occurs is the training of neural networks using exemplar parallelism. Each processor learns, in a condensed way, what every other processor has learned from its data because the processors encode this information in the error vectors they exchange. 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.

So far, we have described exemplar parallelism as if it used deterministic learning, generating an error vector only after the entire data set has been seen by some processor. Using an alternative technique, *batch learning*, the algorithm computes and exchanges error vectors after processing some subset, called a

batch, of the entire data set. Each processor sees only $1/p$ of each batch.

If each batch is sufficiently large, the error vectors that each processor computes are good approximations of the deterministic error vector but have taken much less processing to discover. As a result, 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 in Figure 1. 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. Owen Rogers has verified this general behavior experimentally for several data sets.[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

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

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

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

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 about 20, so speedups are significant even when p is small.

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 might not be accurate; it need only be helpful. It will often have the character of a hint rather than an answer. But hints, as we know, can often shorten searches dramatically.

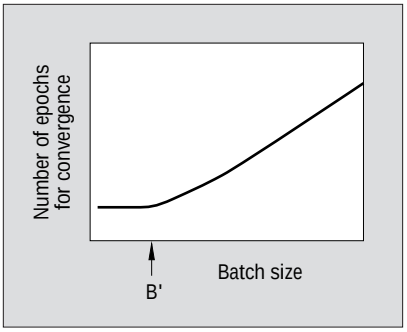

Figure 1. Training of neural networks using exemplar parallelism and batch learning. The algorithm computes and exchanges error vectors after processing a batch of the data set.

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 find the best approach. Cost measures based on counting computations, data accesses, and communication, however, let us compare algorithms and predict their performance. Although we cannot expect these cost measures to be completely accurate, they are expressive enough to rule out techniques that are not worth exploring.

Our cost expressions for three general implementation strategies suggest that replication is both the simplest and the one likely to perform the best. In our neural-networks example, the replicated implementation outperforms other parallelized techniques. Further, replicated implementations often have a work-reducing property that enables double speedup. One speedup factor is proportional to the number of processors used, but the other depends on how quickly the algorithm exploits collective knowledge. The net effect is that modestly parallel implementations can achieve significant performance gains.

ACKNOWLEDGMENTS

This work was supported by the Natural Sciences and Engineering Research Council of Canada.

References

1. J.M.D. Hill, S. Donaldson, and D. Skillicorn, *Stability of Communication Performance in Practice: From the Cray T3E to Networks of Workstations,* Tech. Report PRG-TR-33-97, Oxford Univ. Computing Laboratory, Oxford, England, 1997.
2. D.B. Skillicorn, J.M.D. Hill, and W.F. McColl, "Questions and Answers about BSP," *Scientific Programming*, Vol. 6, No. 3, Fall 1997, pp. 249–274.
3. M. Goudreau et al., "Towards Efficiency and Portability: Programming the BSP Model," *Proc. Eighth Ann. Symp. Parallel Algorithms and Architectures*, ACM Press, New York, 1996, pp. 1–12.
4. J.M.D. Hill, P.I. Crumpton, and D.A. Burgess, *The Theory, Practice, and a Tool for BSP Performance Prediction Applied to a CFD Application,* Tech. Report TR-4-96, Programming Research Group, Oxford Univ. Computing Laboratory, Oxford, England, 1996.
5. J. Reed, K. Parrott, and T. Lanfear, "Portability, Predictability and Performance for Parallel Computing: BSP in Practice," *Concurrency: Practice and Experience,* Vol. 8, No. 10, Dec. 1996, pp. 799–812.
6. R. Agrawal and J. Shafer, *Parallel Mining of Association Rules: Design, Implementation and Experience,* IBM Research Report RJ10004, IBM Almaden Research Center, San Jose, Calif., Feb. 1996.
7. R.O. Rogers and D.B. Skillicorn, "Using the BSP Cost Model for Optimal Parallel Neural Network Training," *Future Generation Computer Systems*, Vol. 14, 1998, pp. 409–424.
8. V. Kumar, S. Shekhar, and M. Amin, "A Highly Parallel Formulation of Backpropagation in Hypercubes," *IEEE Trans. Parallel and Distributed Systems*, Vol. 5, No. 10, Oct. 1994, pp. 1073–1091.
9. R.O. Rogers, *Data Mining with Parallel Neural Networks: Bulk Synchronous Parallelism and the Optimality of Batch Learning,* master's thesis, Dept. Computing and Information Science, Queen's Univ., Kingston, Canada, 1997.

David Skillicorn is a professor in the Department of Computing and Information Science at Queen's University in Canada. His research interests are primarily in parallel computing, ranging from theoretical aspects such as models, to applications in areas such as data mining, structured text, and hypermedia. He was involved in the development of the Bulk Synchronous Parallelism (BSP) model and library. He is a member of the IEEE Computer Society. Contact him at the Dept. of Computing and Information Science, Queen's Univ., Kingston, ON, Canada K7L 3N6; skill@cs.queensu.ca; www.cs.queensu.ca/home/skill.