Efficient Development of Parallel NLP Applications

Prateek Jindal Dan Roth L.V. Kale

Dept. of Computer Science, UIUC

{jindal2,danr,kale}@illinois.edu

Abstract

Parallel programming is becoming increasingly popular. Computers have increasingly many cores (processors). Also, large computer-clusters are becoming available. But there is still no good programming framework for these architectures, and thus no simple and unified way for NLP applications to take advantage of the potential speed up. In this paper, we develop a broadly applicable parallel programming method to NLP problems. Our work is in distinct contrast to the tradition of designing (often ingenious) ways to speed up a single algorithm at a time. Specifically, we show how the problems which can be expressed in LBJ framework [13] take advantage of parallelization. We use Charm++ [7] to demonstrate the speed up of NLP applications.

1. Introduction

This paper describes a way to facilitate the development of natural language applications in parallel. Natural language applications are computation-intensive. With increasing availability of the resources for parallel computing (multicore desktops and computer-clusters), it is natural to use these resources to solve the NLP problems. Unfortunately, when one starts developing natural language applications in parallel, one is fraught with many difficulties as described below.

Today's NLP systems are growing more complex with the need to incorporate a wider range of language resources and more sophisticated statistical methods. In many cases, it is necessary to learn a component with input that includes the predictions of other learned components or to assign simultaneously the values that would be assigned by multiple components with an expressive, data dependent structure among them. As a result, the design of systems with multiple learning components is inevitably quite technically complex, and implementations of conceptually simple NLP systems can be time consuming and prone to error.

So, we see that developing NLP applications is hard. Developing parallel NLP applications is harder. There are a number of steps involved in creating a parallel algorithm (e.g. *task decomposition, mapping and taking care of communication / synchronization issues*). None of these steps are necessary for a sequential program. Not only are there additional steps involved when creating parallel programs, but these steps are very prone to errors. If the wrong task-decomposition is chosen, you might not see any performance increases from parallel programming. Also, there is lack of knowledge of parallel programming systems among the application programmers. Parallel programming is not yet the mainstream. The languages used for parallel programming are often on a very low level, as communication or synchronization operations need to be managed by the programmer. Testing parallel programs is even trickier. Bug can be in 2 places. Either the algorithm is faulty or there may be synchronization problems. Another aspect is the lack of good libraries. Programming becomes way easier, when the programmer can rely on powerful libraries to encapsulate complex behavior. Parallel programming libraries to facilitate the development of NLP applications are largely missing.

In this paper, we address both types of difficulties - difficulties due to complex NLP systems and difficulties due to parallel computing. We use LBJ ([13]) and Charm++ ([7]) to develop parallel NLP systems. LBJ is a framework for developing natural language applications. LBJ has already been used to build state of the art NLP systems (e.g. [12]). Charm++ is a parallel programming paradigm that we use to add the capabilities of parallel programming in LBJ. The programs written in Charm++ can run efficiently on both multicore desktops and computer-clusters without any change. Charm++ has already been very successful in developing numerous parallel applications (e.g. NAMD, OpenAtom etc.). Thus, by integrating Charm++ with LBJ, we show how NLP programmers can use the increased computational power without having to deal with the complexity of parallel computing.

We make three major contributions through this paper. First, we describe a framework for developing parallel NLP applications which doesn't require the NLP programmer to

SPLASH 2010 Workshop on Concurrency for the Application Programmer October 18, Reno

deal with any parallel programming issues. Second, the parallel NLP applications written using this framework run on both multicore desktops and computer-clusters. Third, LBJ provides a very rich library for developing NLP applications. So, now the parallel NLP applications can also make use of this rich library.

This paper is organized as follows. Section 2 presents the related work. Section 3 & 4 describe the main features of LBJ and Charm++ respectively. In Section 5, we show through an example how Charm++ can be integrated into LBJ. Section 7 shows the performance results of a case-study for parallelizing a learning algorithm (one of the components of LBJ) using Charm++. Finally, we conclude in Section 8.

2. Related Work

There have been several attempts in the past to parallelize NLP applications. Elsayed et al. presents a MapReduce algorithm for computing pairwise document similarity in large document collections. Brants et al. [2] report the benefits of large-scale statistical language modeling in machine translation using map-reduce. A distributed infrastructure is proposed which is used to train on up to 2 trillion tokens, resulting in language models having up to 300 billion n-grams. Das et al. [3] describes an approach to collaborative filtering for generating personalized recommendations using map-reduce for users of Google News. Kazama et al. [8] propose using large-scale clustering of dependency relations between verbs and multiword nouns (MNs) to construct a gazetteer for named entity recognition (NER). They use MPI for parallelization. We see that most of the systems to parallelize NLP applications use map-reduce paradigm for parallelization. This necessitates the application to be specified in terms of map-reduce framework. Our system is more general because it does not place such restrictions on the application. Another major limitation of the above approaches is that they are very specific in their scope. However, LBJ is a very expressive modeling language that can express a number of NLP problems. Thus, our parallelization techniques benefit a very broad range of NLP problems.

3. Introduction to LBJ

Many software systems are in need of functions that are simple to describe but that no one knows how to implement. Recently, more and more designers of such systems have turned to machine learning to plug these gaps. Given data, a discriminative machine learning algorithm yields a function that classifies instances from some problem domain into one of a set of categories. For example, given an instance from the domain of email messages (i.e., given an email), we may desire a function that classifies that email as either "spam" or "not spam". Given data (in particular, a set of emails for which the correct classification is known), a machine learning algorithm can provide such a function. We call systems that utilize machine learning technology learning based programs.

Modern learning based programs often involve several learning components (or, at least a single learning component applied repeatedly) whose classifications are dependent on each other. There are many approaches to designing such programs; here, we focus on the following approach. Given data, the various learning components are trained entirely independently of each other, each optimizing its own loss function. Then, when the learned functions are applied in the wild, the independent predictions made by each function are reconciled according to user specified constraints. This approach has been applied successfully to complicated domains such as Semantic Role Labeling.

Learning Based Java is a modeling language for the rapid development of software systems with one or more learned functions, designed for use with the JavaTM programming language. LBJ offers a convenient, declarative syntax for classifier and constraint definition directly in terms of the objects in the programmer's application. With LBJ, the details of feature extraction, learning, model evaluation, and inference are all abstracted away from the programmer, leaving him to reason more directly about his application.

The LBJ compiler accepts the programmer's classifier and constraint specifications as input, automatically generating efficient Java code and applying learning algorithms (i.e., performing training) as necessary to implement the classifiers' entire computation from raw data (i.e., text, images, etc.) to output decision (i.e., part of speech tag, type of recognized object, etc.). The details of feature extraction, model evaluation (i.e., evaluating the function that the learning algorithm returned), and inference (i.e., reconciling the predictions in terms of the constraints at runtime) are abstracted away from the programmer.

A classifier may be defined by:

- coding it explicitly in Java,
- using operators to build it from existing classifiers, or
- identifying feature extraction classifiers and a data source to learn it over.

Under the LBJ programming philosophy, the designer of a learning based program will first design an object oriented internal representation (IR) of the application's raw data using pure Java. A classifier is then any method that produces one or more discrete or real valued classifications with respect to a single object from the programmer's IR. Using LBJ, these classifications are easily interpretable either at face value as the application requires or as features amenable for input to a learning algorithm. Learning algorithms are employed to create learning classifiers, which are classifiers that can change their representation with experience. Once the LBJ compiler has generated these representations from their specifications and user supplied training objects, the application, written in pure Java, simply invokes any classifier

on an IR object just like any other method. Programming with LBJ, the practitioner reasons in terms of his data directly, disregarding the cumbersome implementation details of feature extraction and learning.

LBJ is supported by a library of interfaces and classes that implement a standardized functionality for features and classifiers. The library includes learning and inference algorithm implementations, general purpose and domain specific internal representations, and domain specific parsers.

The LBJ compiler also operates similarly to a makefile. When changes are made to one or more supporting classifiers, the compiler only re-trains those learned classifiers that were affected by the changes.

LBJ has been used to develop several state-of-the-art NLP systems. The LBJ POS tagger reports a competitive 96.6% accuracy on the standard Wall Street Journal corpus. In the named entity recognizer of [12], non-local features, gazetteers, and wikipedia are all incorporated into a system that achieves 90.8 F1 on the CoNLL-2003 dataset, the highest score we are aware of. Finally, the co-reference resolution system of [1] achieves state-of-the art performance on the ACE 2004 dataset while employing only a single learned classifier and a single constraint.

4. Introduction to Charm++

Charm++ [7] is an object-oriented asynchronous message passing parallel programming paradigm. By programming paradigm, we mean Charm++ is a way of writing a program (a programming model). Charm++ is not a programming language in and of itself. Instead, Charm++ uses the C++ programming language as it's base language. Charm++ adds additional functionality and structure on top of C++ that allows the programmer to solve the problem at hand. In Charm++, there are special objects called chares which are used for communication among different processes. Each chare object may contain some state (i.e. data), send and receive messages, and will perform some task in response to receiving a message (that is, execute a special member function called an entry method).

4.1 A Charm++ Program

At a high-level, from the programmer's perspective, a Charm++ program is simply a collection of chare objects. Each chare object has some state associated with it. The chare objects communicate by sending messages to one another. When a particular chare object receives a message, it will execute an entry method to processes the message. This entry method may perform one or more operations/calculations, it may send more messages to other chare objects, it may buffer the contents of the message for later processing, or it may do nothing at all. This is how forward progression is made in the overall application. One chare sends a message to another chare, the receiving chare does some computation and then sends out more messages to other chares, and so on, and so on. Execution begins with a special chare called the main chare (similar to how execution of a C++ program begins with the execution of a special function called main).

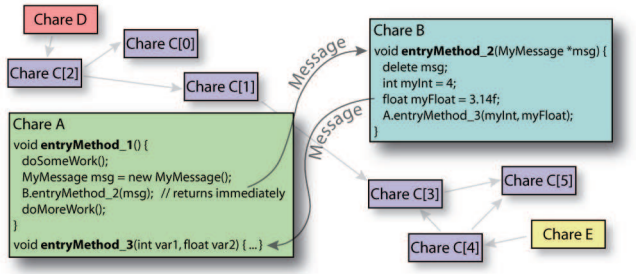

Figure 1. Charm user view.

4.2 Charm++ Runtime System

When a programmer writes a Charm++ application, they write it in terms of chare objects and how the chare objects communicate with one another through method invocation (or message passing). Details such as the number of processors, types of processors, type of interconnect, and so on are not considered. The programmer simply writes the application as a collection of interacting objects. This view of a Charm++ application is referred to as the user's view of a Charm++ application (see Figure 1).

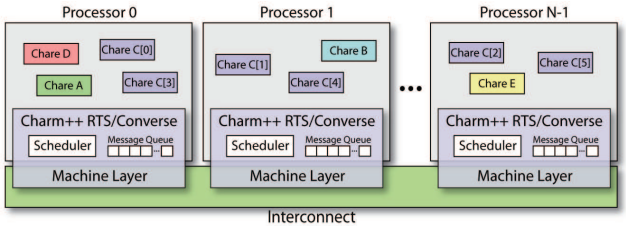

Figure 2. Charm system view.

Alternatively, when the application is actually compiled there is a specific target platform including type of processor, type of interconnect, and so on. Additionally, when the application is executed, there is a specific set of physical resources that are made available to the application such as the number of physical processors, and so on. The purpose of the Charm++ Runtime System is to manage as many of the details of the physical resources on behalf of the application, and thus, on behalf of the programmer. This view of the Charm++ application is referred to as the system's view of a Charm++ application (see Figure 2). Management decisions that the Charm++ Runtime System can make on behalf of the application include (but are not limited to):

Mapping Chare Objects to Physical Processors Through various methods, the Charm++ Runtime System can assign the chare objects to physical processors.

Load-Balancing Chare Objects The Charm++ Runtime System can dynamically migrate chare objects between physical processors as the application executes allowing the application to utilize the physical processors more efficiently.

Routing of Messages As chare objects are assigned to physical processors and migrated between physical processors, the Charm++ Runtime System keeps track of where the chare objects live. Messages being sent to a chare object are dynamically routed to the physical processor containing the chare object.

Checkpointing Because the Charm++ Runtime System, through the use of PUP Routines, can migrate a chare object's state between physical processors, checkpointing is fairly trivial. The Charm++ Runtime System can simply migrate all of the chare objects' states to disk.

Fault-Tolerance If a physical processor is experiencing problems or has already crashed, the Charm++ Runtime System can dynamically recreate the chare objects on the failed physical processor on the remaining physical processors.

Dynamic Re-Allocation of Physical Resources A cluster being used to execute a Charm++ application may suddenly receive more jobs (or have several jobs finish). The Charm++ Runtime System can dynamically migrate chare objects from (or to) physical processors allowing the application to dynamically shrink to use fewer physical processors (or expand to use more physical processors) based on the cluster's overall load.

Each processing element has its own Charm++ Runtime System running on it. The various instances of the Charm++ Runtime Systems are responsible for their local processing element. They may also communicate with one another for collective operations (such as checkpointing, fault-recovery, load-balancing, and so on).

Charm++ has been successfully used for the parallelization of numerous applications.

- NAMD - Molecular Dynamics [11]
- ChaNGa - Computational Cosmology [5]
- OpenAtom - Ab Initio Molecular Dynamics [15]
- LeanMD - Protein Folding Peta-Flop class machines [10]
- Rocket Simulation
- Computational Science and Engineering (CSE) Applications

4.3 Tools

Several tools are available to facilitate the programming with Charm++ as listed below:

- Projections: Performance analysis tool.
- Faucets: Job submission tool.
- CharmDebug: Debugging tool for Charm++ programs.
- BigSim - Simulating PetaFLOPS Supercomputers
- BigNetSim - Parallel InterConnection Network Simulation

5. Integration of LBJ with Charm++

LBJ helps to develop NLP applications by providing an easy way to develop and use classifiers as features. In addition to the simple, hard-coded classifiers that come packaged with LBJ, a constantly growing suite of learned classifiers is available. These classifiers can be imported into LBJ or Java source code, and used just like methods. Figure 3 shows the syntax for learning classifier expressions.

Table 1 shows some of the NLP applications that have been developed using LBJ. All these applications require the training of classifiers.

Learning classifiers can be computationally very expensive. Classifiers are trained over some training data. This training data can contain millions of training examples. Also, classifiers features can themselves number in millions. Most of the learning algorithms that are used to train classifiers need to take multiple rounds on the training data for learning accurate models. Long learning times for training classifiers significantly hurts the efficiency of application developers. It is also irritating for the application users.

Using Charm++, we provide a way of learning classifiers efficiently using multicore-desktops and computer-clusters. The application programmer using LBJ simply needs to specify that he would like to use the parallel version of the classifier. It automatically invokes the Charm++ Runtime system which trains the classifier in parallel and stores the learned model in a file. This learned model can subsequently be used in the application to make classifications.

Next section demonstrates the parallelization of a learning algorithm using Charm++. The learning algorithm chosen is called Liblinear [4].

6. Parallel Liblinear

LBJ provides a number of learning algorithms to train classifiers. These learning algorithms include Perceptron, Average Perceptron, Winnow, Stochastic Gradient Descent, Naive Bayes, Liblinear etc. Here, we focus on Liblinear to study the performance gains obtained by parallelization. It is a novel dual coordinate descent method for linear SVM with L1- and L2-loss functions. Liblinear has been shown to be much faster than state of the art solvers such as Pegasos [14], TRON [9], SVMperf [6], etc.

Support Vector Machines (SVMs) are useful for data classification. Given a set of instance-label pairs $(x_i, y_i), i = 1, ..., l$, $x_i \in R^n$; $y_i \in -1, +1$, SVM requires the solution of the following unconstrained optimization problem:

Table 1. NLP applications developed using LBJ

Application	Description
Illinois Part of Speech Tagger	This is an implementation of our SNoW-based POS tagger for use with LBJ.
Illinois Chunker	A classifier that partitions plain text into sequences of semantically related words, indicating a shallow (i.e., non-hierarchical) phrase structure.
Illinois Coreference	A Coreference Resolver, based on LBJ, trained on the ACE 2004 corpus.
Illinois Named Entity Tagger	This is a state of the art NE tagger that tags plain text with named entitites (people / organizations / locations / miscellaneous). It uses gazetteers extracted from Wikipedia, word class model derived from unlabeled text and expressive non-local features. The best performance is 90.8 F1 on the CoNLL03 shared task data.

> learn [classifier-expression] // Labeler
> using classifier-expression // Feature extractors
> [from instance-creation-expression [int]] // Parser
> [with instance-creation-expression] // Learning algorithm
> [evaluate Java-expression] // Alternate eval method
> [cval [int] split-strategy // K-Fold Cross Validation
> [alpha double] // Confidence Parameter
> [testingMetric
> instance-creation-expression]] // Testing Function
> [preExtract boolean] // Feature Pre-Extraction
> [progressOutput int] // Progress Output Frequency
> End

The first classifier expression represents a classifier that will provide label features for a supervised learning algorithm. The classifier expression in the using clause does all the feature extraction on each object, during both training and evaluation. When the from clause appears, the LBJ compiler retrieves objects from the specified parser until it finally returns null. The instance creation expression in the with clause should create an object of a class derived from the LBJ2.learn.Learner class in the library.

Figure 3. Syntax for learning classifier expressions.

$$min_w \frac{1}{2} w^T w + C \sum_{i=1}^{l} \xi(w; x_i, y_i) \quad (1)$$

where $\xi(w; x_i, y_i)$ is a loss function, and $C > 0$ is a penalty parameter. The above problem is often referred to as the primal form of SVM. Liblinear solves the dual version of the problem:

$$min_\alpha f(\alpha) = \frac{1}{2} \alpha^T \overline{Q} \alpha - e^T \alpha \quad (2)$$

$$\text{subject to } 0 \leq \alpha_i \leq U, \forall i$$

where $\overline{Q} = Q + D$, D is a diagonal matrix, and $Q_{ij} = y_i y_j x_i^T x_j$.

6.1 Working of Liblinear

Algorithm 1 A dual coordinate descent method for Linear SVM

Given α and the corresponding $w = \sum_i y_i \alpha_i x_i$
while α is not optimal **do**
  for $i \in \{1, 2, \ldots, l\}$ **do**
    $G = y_i W^T x_i - 1 + D_{ii} \alpha_i$
    $$PG = \begin{cases} min(G, 0) & \text{if } \alpha_i = 0, \\ max(G, 0) & \text{if } \alpha_i = U, \\ G & \text{if } 0 < \alpha_i < U. \end{cases}$$
    if $|PG| \neq 0$ **then**
      $\overline{\alpha_i} \leftarrow \alpha_i$
      $\alpha_i \leftarrow min(max(\alpha_i - G/\overline{Q_{ii}}, 0), U)$
      $w \leftarrow w + (\alpha_i - \overline{\alpha_i}) y_i x_i$
    end if
  end for
end while

Liblinear algorithm [4] is described in Algorithm 1. The optimization process starts from an initial point $\alpha^0 \in R^l$ and generates a sequence of vectors ${\alpha^k}_{k=0}^{\infty}$. Hsieh et al., 2008 refer to the process from α^k to α^{k+1} as an outer iteration. In each outer iteration, there are l inner iterations, so that sequentially $\alpha_1, \alpha_2, \ldots, \alpha_l$ are updated. Each outer iteration thus generates vectors $\alpha^{k,i} \in R^l$, $i = 1, \ldots, l+1$, such that $\alpha^{k,1} = \alpha^k$, $\alpha^{k,l+1} = \alpha k + 1$, and

$$\alpha^{k,i} = [\alpha_1^{k+1}, \ldots, \alpha_{i-1}^{k+1}, \alpha_i^k, \ldots, \alpha_l^k]^T, \forall i = 2, \ldots, l. \quad (3)$$

6.2 Limitations of Liblinear

Although Liblinear is able to efficiently learn the linear classifiers, it doesnt scale well for very large problem sizes. Liblinear stores all the training vectors into the memory as ¡Feature, Value¿ pairs. As a result, Liblinear requires the memory space which increases linearly with the size of the dataset. Figure 4 shows the memory consumed by Liblinear as a function of the training data size. The total number of features used in this experiment was 100,000. As we increase the number of training vectors from 2000 to 20,000, the memory requirements increase from 0.74 GB to 7.4 GB. So, it is clear that we cant use Liblinear for training very large datasets.

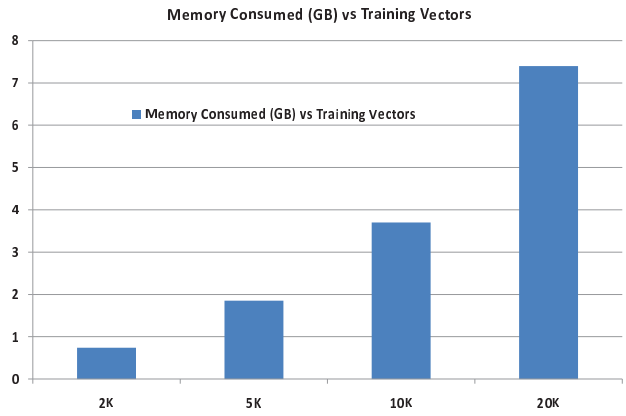

Figure 4. Limitations of Liblinear.

6.3 Scaling Liblinear to very large datasets

One solution to overcome the problem of insufficient memory is to use the disk space. We note that in each inner iteration, Liblinear needs the weight vector and 1 training vector. Thus, we can swap the training vectors into and out of memory alternately and thus use only a constant amount of memory. But the problem here is that the Liblinear needs all the training vectors for every outer iteration. This would lead to a lot of swaps into and out of memory and thus would be very inefficient. An alternative solution is to use multiple processors in parallel.

We have implemented 2 solutions to using multiple processers. Each of the solutions is described below.

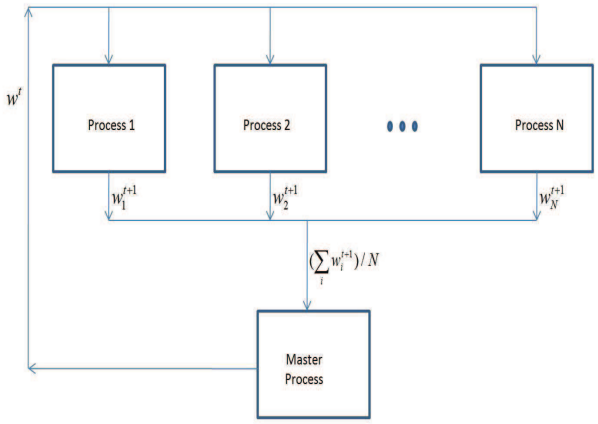

Figure 5. Solution 1.

6.3.1 Solution 1

In the first solution, we implemented the exact version of Liblinear in parallel. Figure 5 shows the schematic view of parallel liblinear. This figure shows that we maintain a set of processes. Each of the processes is responsible for a subset of the data. In each outer iteration of Liblinear, we see that the entire training set is scanned once. The training vectors are considered one by one in the inner iterations. Each inner iteration updates the weight vector. The next iteration uses the modified weight vector. This makes the exact implementation inherently sequential.

The master process initializes the weight vector to zero. It sends the weight vector to the first process. This process runs one outer iteration on its part of the training set. The modified weight vector is passed on to 2nd process. In this way, the weight vector is passed along the chain of processes. Along with the weight vector, we also pass the maximum and the minimum projected gradient that has been obtained so far. The last process in the chain passes the weight vector back to the master process. The master process keeps track of the total number of iterations that have executed so far. From the maximum and minimum projected gradient, the master process determines whether the required convergence has been reached or not according to the following relation:

$$\text{if (PGmax - PGmin)} \leq \text{eps then stop.}$$

where eps is provided by the user. A value of 0.1 is good for most purposes. The problem with the exact implementation is that only one of the processes is active at any moment. But it solves the problem of insufficient memory by distributing the training set among different processors.

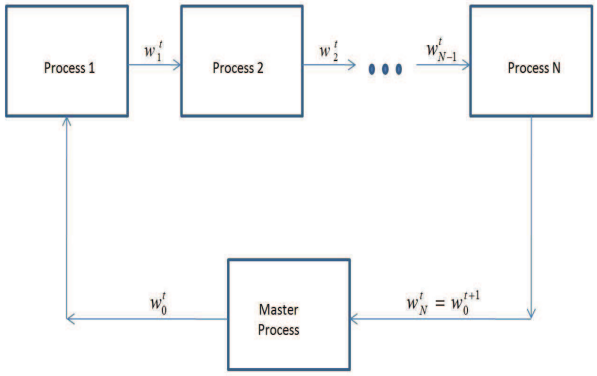

Figure 6. Solution 2.

6.3.2 Solution 2

To improve the resource utilization of parallel Liblinear, we modified the above version by training the different parts of the dataset concurrently. Figure 6 shows the schematic view of this implementation. In this implementation, we divide the training set among different processes as in the previous solution. But, the master process doesnt send the weight

vector to process 1 as in the previous solution. Instead, the master process broadcasts the weight vector to all the processes. Now, each of the processes carries out one outer iteration on its part of the training set concurrently. After one outer iteration, all the processes send their weight vector to the master process. The master process computes the new weight vector from the received weight vectors according to the following equation:

$$w^{t+1} = \sum_i (w_i^{t+1})/N$$

where w_i^{t+1} is the weight vector output by i^{th} process at the end of tth iteration. And w^{t+1} is the weight vector to be broadcasted to all the processes at the beginning of $(t+1)^{th}$ iteration. As in the previous solution, all the sub-processes also send their maximum and minimum projected gradient to the master process. The master process determines whether the convergence has been reached or not by using maximum and minimum projected gradients as in the previous solution. If the convergence has not been reached yet, the weight vector is broadcasted again and the process continues until convergence.

7. Results

Next, we compare the convergence behavior of the 2 parallel solutions to Liblinear. The machine used for these experiments had the following characteristics:

CPU: Intel Xeon 2.00 GHz 8-core
Memory: 5 GB
Processors Used for Experiments: 4

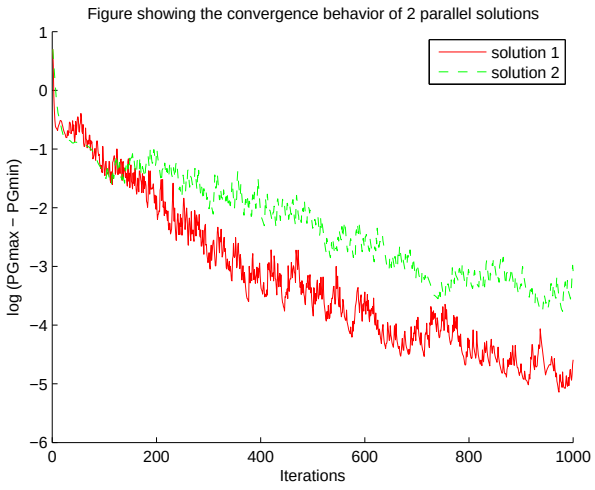

Figure 7. Convergence behavior of two parallel solutions.

Table 2 shows the characteristics of the datasets used in the experiments. We trained both versions of parallel Liblinear on these datasets. Figures 7 and 8 above show the convergence behavior of the 2 solutions. The steeper descent

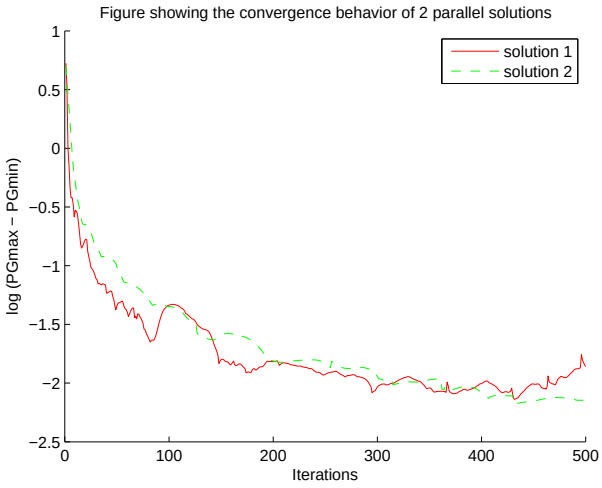

Figure 8. Convergence behavior of two parallel solutions.

Table 2. Characteristics of the datasets used for experiments

	Features	**Training Vectors**	**Non-zero Elements**
Dataset 1	10,000	4000	20,000,000
Dataset 2	2,000	8,000	8,000,000

indicates the faster convergence. We find that solution 1 converges faster for 1st dataset. For 2nd dataset, both the solutions converge almost equally fast. These figures reveal that both the solutions have good convergence properties. Table shows the time taken by 2 parallel implementations. We find that 2nd implementation is about 2.6 times faster than the first.

Table 3. Comparison of time taken (in seconds) by two parallel implementations

	Solution 1	**Solution 2**
Dataset 1	196	75
Dataset 2	39	15

8. Conclusions

Natural Language Processing is a challenging task. To deal with the complexity of natural language, todays NLP systems incorporate a wider range of language resources and more sophisticated statistical methods. NLP systems can derive great benefits from parallel computing which is becoming increasingly popular. In this paper, we have shown how to integrate parallel programming techniques into LBJ, which can be used to express a wide variety of NLP problems. LBJ provides a very rich library to help in developing NLP applications. Our solution enables the parallel NLP applications to use this rich library. Specifically, we presented a parallel version of a learning algorithm which can run

both on multicore-desktops and computer clusters. We also demonstrated the convergence results of the parallel learning algorithm.

References

[1] E. Bengtson and D. Roth. Understanding the value of features for coreference resolution. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 294–303. Association for Computational Linguistics, 2008.

[2] T. Brants, A. Popat, P. Xu, F. Och, and J. Dean. Large language models in machine translation, June 22 2007. US Patent App. 11/767,436.

[3] A. Das, M. Datar, A. Garg, and S. Rajaram. Google news personalization: scalable online collaborative filtering. In *Proceedings of the 16th international conference on World Wide Web*, page 280. ACM, 2007.

[4] C. Hsieh, K. Chang, C. Lin, S. Keerthi, and S. Sundararajan. A dual coordinate descent method for large-scale linear SVM. In *Proceedings of the 25th international conference on Machine learning*, pages 408–415. ACM, 2008.

[5] P. Jetley, L. Wesolowski, F. Gioachin, L. V. Kalé, and T. R. Quinn. Scaling Hierarchical N-body Simulations on GPU Clusters. In *Proceedings of the ACM/IEEE Supercomputing Conference 2010 (to appear)*, 2010.

[6] T. Joachims. Training linear SVMs in linear time. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, page 226. ACM, 2006.

[7] L. V. Kale and G. Zheng. Charm++ and AMPI: Adaptive Runtime Strategies via Migratable Objects. In M. Parashar, editor, *Advanced Computational Infrastructures for Parallel and Distributed Applications*, pages 265–282. Wiley-Interscience, 2009.

[8] J. Kazama and K. Torisawa. Inducing gazetteers for named entity recognition by large-scale clustering of dependency relations. *Proceedings of ACL-08: HLT*, pages 407–415, 2008.

[9] C. Lin, R. Weng, and S. Keerthi. Trust region Newton method for logistic regression. *The Journal of Machine Learning Research*, 9:627–650, 2008.

[10] V. Mehta. LeanMD: A Charm++ framework for high performance molecular dynamics simulation on large parallel machines. Master's thesis, University of Illinois at Urbana-Champaign, 2004.

[11] J. C. Phillips, G. Zheng, S. Kumar, and L. V. Kalé. NAMD: Biomolecular simulation on thousands of processors. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–18, Baltimore, MD, September 2002.

[12] L. Ratinov and D. Roth. Design challenges and misconceptions in named entity recognition. In *CoNLL '09: Proceedings of the Thirteenth Conference on Computational Natural Language Learning*, pages 147–155, Morristown, NJ, USA, 2009. Association for Computational Linguistics. ISBN 978-1-932432-29-9.

[13] N. Rizzolo and D. Roth. Learning Based Java for Rapid Development of NLP Systems. In *Proceedings of the International Conference on Language Resources and Evaluation (LREC)*, Valletta, Malta, May 2010. URL `http://l2r.cs.uiuc.edu/~danr/Papers/RizzoloRo10.pdf`.

[14] S. Shalev-Shwartz, Y. Singer, and N. Srebro. Pegasos: Primal estimated sub-gradient solver for svm. In *Proceedings of the 24th international conference on Machine learning*, page 814. ACM, 2007.

[15] R. V. Vadali, Y. Shi, S. Kumar, L. V. Kale, M. E. Tuckerman, and G. J. Martyna. Scalable fine-grained parallelization of plane-wave-based ab initio molecular dynamics for large supercomputers. *Journal of Comptational Chemistry*, 25(16): 2006–2022, Oct. 2004.