Parallel Levenberg-Marquardt-based Neural Network Training on Linux Clusters - A Case Study

N. N. R. Ranga Suri and Dipti Deodhare
AI & Neural Networks Group
Centre for Artificial Intelligence & Robotics
Bangalore
{nnrsuri,dipti}@cair.res.in

P. Nagabhushan
DOS in CS
University of Mysore
Mysore
pnagabhushan@hotmail.com

Abstract

This paper addresses the problem of pattern classification using neural networks. Applying neural network classifiers for classifying a large volume of high dimensional data is a difficult task as the training process is computationally expensive. A parallel implementation of the known training paradigms offers a feasible solution to the problem. By exploiting the massively parallel structure of the Levenberg-Marquardt algorithm for non-linear optimization a training algorithm for neural networks has been implemented on a Linux cluster using LAM (Local Area Multi-computer) MPI (Message Passing Interface). The implementation, besides facilitating the main objective of maximising computational speedup, is also portable and scalable. A standard benchmark for neural network training comprising a sufficiently large volume of satellite image data has been utilized to present and discuss the properties of the implementation.

1. Introduction

The goal of pattern classification is to assign input patterns to one of a finite number of classes. Parametric Bayesian classifiers, described in [4], are the most widely used as they are simple to implement. On the other hand, the applied research in this field has paid more attention to practical issues of implementing classifiers for real world problems. Further explorations in this direction have revealed that adaptive non-parametric neural network classifiers are well suited for real world problems. As explained in [10], neural network classifiers allow selection of differing practical characteristics and provide reduced error rates. This special feature exhibited by neural network classifiers makes them suitable for handling complex problems like analyzing and classifying satellite image data.

Satellite image analysis involves tasks like preprocessing, segmentation and edge detection. Moreover acquiring such data involves recording the intensity values at different spectral bands. Due to this and other such reasons, satellite image data tends to be high dimensional. Results reported by some researchers for satellite image analysis, [16], show that neural network classifiers have performed well for various tasks involved in the analysis process.

For applications like satellite image analysis, as the input dimension grows large, handling such data for training neural networks becomes a time consuming task as shown in [15]. The solution would be to go in for some kind of parallel implementation of the neural network training process. Implementations like [11, 17] are effective in terms of training speed, but are hardware specific, hence lack portability to a variety of architectures. Given that there is wide applicability of neural networks for pattern classification [10], there is a lot of need for portable implementations of parallel neural network training algorithms.

Linux clusters are fast becoming popular platforms for the development of parallel & portable programs. Establishing a Linux cluster involves connecting computers running Linux operating systems with an appropriate network switch and then installing the requisite parallel libraries on each of them. This method is basically a way of establishing a loosely coupled parallel computing environment in which different processes communicate with each other by means of messages. Message passing is a paradigm widely used on parallel machines since it can be efficiently and portably implemented. This way of developing parallel programs has caught the attention of many application developers as it offers a cost effective solution.

This paper describes parallel implementation of a neural network training algorithm based on the Levenberg-Marquardt (LM) algorithm for non-linear optimization. In section 2 we describe the structure of the algorithm and some important issues related to its implementation. Section 3 includes a brief note on the Linux cluster that was established and the LAM MPI parallel libraries used. Section 4 and 5 present our results in terms of classification accuracy and computational speedups attained through implementation.

2. The Levenberg-Marquardt Algorithm

Pattern Classification with neural classifiers basically involves understanding the class boundaries by the classifier. To attain this capability, the classifier has to undergo a training phase. The same is achieved with the help of a training algorithm.

Gradient-based training algorithms, like back-propagation, are most commonly used by researchers. They are not efficient due to the fact that the gradient vanishes at the solution. *Hessian-based* algorithms used as reported in [8], allow the network to learn more subtle features of a complicated mapping. The training process converges quickly as the solution is approached, because the Hessian does not vanish at the solution.

To benefit from the advantages of Hessian based training, we focused on the Levenberg-Marquardt Algorithm reported in [9, 12, 6]. The LM algorithm is basically a Hessian-based algorithm for nonlinear least squares optimization. For neural network training the objective function is the error function of the type

$$e(\mathbf{x}) = \frac{1}{2}\sum_{k=0}^{p-1}\sum_{l=0}^{n_o-1}(d_{kl} - a_{kl})^2 \qquad (1)$$

where a_{kl} is the actual output at the output neuron l for the input k and d_{kl} is the desired output at the output neuron l for the input k. p is the total number of training patterns and n_o represents the total number of neurons in the output layer of the network. $\mathbf{x}$ represents the weights and biases of the network.

The steps involved in training a neural network in batch mode using the Levenberg-Marquardt algorithm are as follows:

1. Present all inputs to the network and compute the corresponding network outputs and errors. Compute the mean square error over all inputs as in equation 1.
2. Compute the Jacobian matrix , $J(\mathbf{x})$ where $\mathbf{x}$ represents the weights and biases of the network.
3. Solve the Levenberg-Marquardt weight update equation to obtain $\Delta\mathbf{x}$. (Refer to equation 2)
4. Recompute the error using $\mathbf{x} + \Delta\mathbf{x}$. If this new error is smaller than that computed in step 1, then reduce the *training parameter* μ by μ^-, let $\mathbf{x} = \mathbf{x} + \Delta\mathbf{x}$, and go back to step 1. If the error is not reduced, then increase μ by μ^+ and go back to step 3. μ^+ and μ^- are predefined values set by the user. Typically μ^+ is set to 10 and μ^- is set to 0.1.
5. The algorithm is assumed to have converged when the norm of the gradient is less than some predetermined value, or when the error has been reduced to some error goal.

In the above algorithm, the weight update vector $\Delta\mathbf{x}$ is calculated as

$$\Delta\mathbf{x} = [J^T(\mathbf{x})J(\mathbf{x}) + \mu I]^{-1}J^T(\mathbf{x})R \qquad (2)$$

where R is a vector of size pn_o calculated as follows

$$R = \begin{pmatrix} d_{11} - a_{11} \\ d_{12} - a_{12} \\ \dots \\ \dots \\ d_{21} - a_{21} \\ d_{22} - a_{22} \\ \dots \\ \dots \\ d_{pn_o} - a_{pn_o} \end{pmatrix}$$

$J^T(\mathbf{x})J(\mathbf{x})$ is referred to as the Hessian matrix. Let the dimension of the input space be n_i. Suppose there are n_o classes that the input data is to be classified into. Also let the total number of patterns be p. If the popular Multi-Layer Perceptron (MLP) is used to do the classification it will necessarily comprise an n_i dimensional input layer and an n_o dimensional output layer. If n_h is the number of hidden neurons that the single hidden layer of the MLP has, then the total number of weights and biases in the MLP will be

$$w = n_i n_h + n_h n_o + n_h + n_o \qquad (3)$$

With the above notation the dimension of the Jacobian will be pn_o x w while that of the Hessian will be w x w where w is as computed in equation 3.

Since the properties of the algorithm and its parallel implementation are best discussed with reference to a sufficiently high-dimensional, large volume data, the neural network benchmark comprising satellite data [1] has been used. The database consists of the multi-spectral values of pixels in 3x3 neighborhoods in a satellite image and the classification associated with the central pixel in each neighborhood. The aim is to predict this classification given the multi-spectral values. In the sample database, the class of a pixel is coded as a number. There are a total of 36 numerical attributes in the range 0 to 255 and there are 6 decision classes. The training set consists of 4435 patterns while the test set consists of 2000 patterns. Appropriately mapping these values to the notation discussed above we have $p = 4435$, $n_i = 36$ and $n_o = 6$.

The best results in our experiments were obtained with $n_h = 27$ i.e. 27 hidden layer neurons. Therefore, if the Levenberg-Marquardt algorithm is employed for the neural network training, the dimension of the Jacobian will be 26610x1167 while that of the Hessian will be 1167x1167.

For practical problems (particularly comprising image data) these dimensions will be even higher since the dimensions for the Jacobian scale both with the input dimension (n_i) and the training data size (p). Needless to say the computation time also increases. Consequently, both for constraining space requirements as well as for adequate speedups parallel distributed computing is not only desirable but essential. The computation of the Jacobian lends itself to parallelization and this allows for practical high dimensional large volume problems to be handled using the LM-based training algorithm. In what follows we discuss one such implementation on a 17 node LAM MPI-based Linux cluster.

3. Linux Cluster Setup

A cluster of 17 personal computers working with the Linux (Debian GNU/Linux 3.0, Woody binary) operating system was established to carry out the implementation. Connectivity between the computers was achieved via a 3-Com switch and the Ethernet protocol. As already mentioned Message Passing Interface (MPI) is a paradigm that provides the facility to develop parallel and portable algorithms. An MPI program consists of autonomous processes, executing their own code, in an MIMD style, as described in [13]. The codes executed by each process need not be identical. The processes communicate via calls to MPI communication primitives. Typically, each process executes in its own address space, although shared-memory implementations of MPI are possible. LAM stands for Local Area Multicomputer and is an implementation of the MPI standard. It is a parallel processing environment and development system, described in [2], for a network of independent computers. It features the MPI programming standard for developing parallel programs. The LAM MPI parallel libraries were installed on each computer in the cluster to implement parallel constructs based on MPI.

One of these computers was designated as the *master*. The master monitors the overall execution of the application program. The rest of the computers were designated as *slave nodes*. (Henceforth a slave node will simply be referred to as node.) Basically a setup consisting of a master-slave environment with 16 slave nodes was established.

4. Parallelizing the LM Algorithm

As already discussed, the Jacobian and the Hessian matrix computations are computationally expensive. Moreover, even for moderately sized problems and moderately sized neural network architectures, the size of the Jacobian can become prohibitively large. Addressing this issue, the LM algorithm is parallelized for neural network training by appropriately distributing the computation and space requirements over the cluster. SPMD (Single Program and Multiple Data) strategy was used in the parallelization process.

Steps involved in training a neural network in batch mode using the parallelized LM algorithm are as follow:

1. Read the training data/patterns for which the neural network classifier has to be designed.

2. Consider a neural network architecture (MLP) with n_i input layer neurons, n_h hidden layer neurons and n_o output layer neurons, by making use of the details like input dimension and the number of classes into which the training data have to be classified.

3. If the number of nodes in the cluster under consideration is n, divide the training data into n groups, such that each group has a number of patterns equal to $s = p/n$.

4. Present all inputs to the network and compute the corresponding network outputs and errors. Compute the mean square error over all the inputs as shown in the equation 1.

5. For each node in the cluster, execute the following steps

   (a) Let $Q^i(\mathbf{x})$ be a matrix corresponding to s number of the training patterns made available at node i represented as

   $$Q^i(\mathbf{x}) = \left[q^i_{kl}\right],\ 0 \leq l < n_o\ and\ 0 \leq k < s$$

   where $q^i_{kl} = (d^i_{kl} - a^i_{kl})$ represents the error value at the output neuron l for the input pattern k. Construct a vector R^i of size sn_o by taking the elements of the matrix $Q^i(\mathbf{x})$ in column major order.

   (b) Compute the row block of the Jacobian matrix corresponding to the node i as

   $$J^i = \begin{pmatrix} \frac{\partial q_{11}(\mathbf{x})}{\partial \mathbf{x}_1} & \frac{\partial q_{11}(\mathbf{x})}{\partial \mathbf{x}_2} & \cdots & \frac{\partial q_{11}(\mathbf{x})}{\partial \mathbf{x}_w} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial q_{1n_o}(\mathbf{x})}{\partial \mathbf{x}_1} & \frac{\partial q_{1n_o}(\mathbf{x})}{\partial \mathbf{x}_2} & \cdots & \frac{\partial q_{1n_o}(\mathbf{x})}{\partial \mathbf{x}_w} \\ \frac{\partial q_{21}(\mathbf{x})}{\partial \mathbf{x}_1} & \frac{\partial q_{21}(\mathbf{x})}{\partial \mathbf{x}_2} & \cdots & \frac{\partial q_{21}(\mathbf{x})}{\partial \mathbf{x}_w} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial q_{2n_o}(\mathbf{x})}{\partial \mathbf{x}_1} & \frac{\partial q_{2n_o}(\mathbf{x})}{\partial \mathbf{x}_2} & \cdots & \frac{\partial q_{2n_o}(\mathbf{x})}{\partial \mathbf{x}_w} \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial q_{s1}(\mathbf{x})}{\partial \mathbf{x}_1} & \frac{\partial q_{s1}(\mathbf{x})}{\partial \mathbf{x}_2} & \cdots & \frac{\partial q_{s1}(\mathbf{x})}{\partial \mathbf{x}_w} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial q_{sn_o}(\mathbf{x})}{\partial \mathbf{x}_1} & \frac{\partial q_{sn_o}(\mathbf{x})}{\partial \mathbf{x}_2} & \cdots & \frac{\partial q_{sn_o}(\mathbf{x})}{\partial \mathbf{x}_w} \end{pmatrix}$$

   and the same is represented as

   $$J^i = \left[j^i_{kl}\right],\ 0 \leq k < sn_o\ and\ 0 \leq l < w$$

where w is as calculated in equation 3 and j^i_{kl} is the element at kth row and lth column of the Jacobian matrix computed at node i.

(c) Then compute the matrix H^i as follows

$$H^i = (J^i)^T * J^i$$

(d) Also compute the gradient of the error corresponding to the node i as

$$g^i = (J^i)^T * R^i$$

6. Construct the Hessian matrix H, by adding the corresponding components of the matrix elements computed at the nodes in the cluster as

$$H = \sum_{i=0}^{n} H^i$$

7. Similarly construct the gradient as

$$g = \sum_{i=0}^{n} g^i$$

8. Solve the equation below to obtain the weight update vector $\Delta\mathbf{x}$ corresponding to the network weights $\mathbf{x}$ as

$$\Delta\mathbf{x} \ = \ [H + \mu I]^{-1} g$$

9. Recompute the error using $\mathbf{x} + \Delta\mathbf{x}$. If this new error is smaller than that computed in step 4, then reduce μ by μ^-, let $\mathbf{x} = \mathbf{x} + \Delta\mathbf{x}$, and go back to step 4. If the error is not reduced, then increase μ by μ^+ and go back to step 8. μ^+ and μ^- are predefined values set by the user. Typically μ^+ is set to 10 and μ^- is set to 0.1.

10. The algorithm is assumed to have converged when the norm of the gradient is less than some predetermined value, or when the error has been reduced to less than some predefined error.

The implementation program has been developed in the C language using MPI programming constructs. The program basically consists of two procedures named as $master()$ and $node()$ that are intended to run on the master computer and the node computers respectively. These two procedures are called by the main program within the MPI environment to complete the algorithm. Pseudo code indicative of the algorithmic organization of the implementation, as shown in Figure 1, is presented below.

```
mpi_lm() //...main program on MPI for LM algorithm
    MPI_Initialize();
    master() //...tasks to be done by the master
        get_network_info();
        initialize_weights();
        load_training_data();
        calc_error();
        distribute_data();
        recv_node_Hessian();
        update_weights();
    end
    node() //...computation at each node
        recv_node_data();
        calc_node_Jacobian();
        calc_node_Hessian();
        send_node_Hessian();
    end
    MPI_Finalize();
end
```

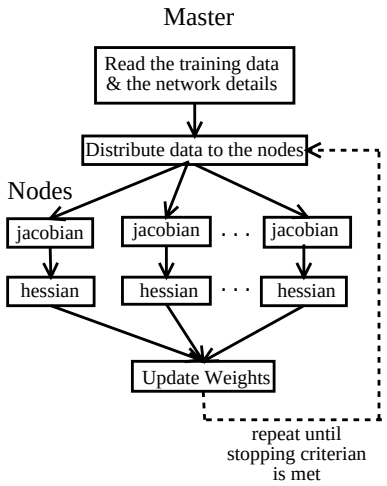

Figure 1: SPMD parallel strategy for LM algorithm

5. Performance Accuracy

As already mentioned in section 2, the satellite image benchmark from the UCI Machine Learning Repository has been used for discussing the cluster setup and the parallel implementation. An multi layer perceptron consisting of 36 input neurons and 6 output neurons to handle the 36 dimensional, 6 class classification problem was considered. The multi layer perceptron had a single hidden layer. The standard sigmoid activation, defined as

$$f(x) = \frac{1}{1 + e^{-x}}$$

was used for all the neurons.

To begin with, a smaller number of neurons were introduced in the hidden layer. The training set consists of 4435 patterns. Since clusters of 4, 8, 16 nodes were considered for presenting our results it would be convenient

to have the number of patterns in the training set as a multiple of 16. The number of patterns belonging to class 6 in the training was larger as compared to the other classes. Therefore the last 3 patterns in the training set belonging to class 6 were dropped. As a result the only 4432 patterns were used for training. When all 16 nodes of the cluster were used to train the neural network with LM algorithm, each node had a share of 277 training patterns. As a result the dimension of the Jacobian matrix at each node was 1662x1167 only. As already discussed, if all the training patterns were trained on one single node then the dimension of the Jacobian matrix would have been 26592x1167 which is significantly larger. After training, the performance of the neural network on the prescribed test set, consisting of 2000 patterns was observed. Gradually the number of hidden neurons was increased. The performance accuracy of the trained neural network on the test set for 20, 24 and 27 hidden nodes has been presented in Table 1. No significant improvements in the performance accuracy were observed on increasing the number of neurons in the hidden layer beyond 27.

Num. of hidden layer neurons	Classification accuracy
20	78.65%
24	79.1%
27	82.6%

Table 1: Classification accuracy with varying number of hidden layer neurons.

We have employed the threshold-margin criterion advocated by Fahlman [5], to present our results. As the criterion suggests, the actual output of a neuron in the output layer, namely a was taken to match with that of the desired output d if $|a - d| < 0.4$. Therefore for neurons with desired output as $d = 1$ the actual output was considered to match only if $a > 0.6$. Similarly for $d = 0$, the actual output was considered to match only if $a < 0.4$. This criterion is more stringent as compared to the simplistic criterion of the highest output node indicating the class to which the pattern belongs, employed by most researchers. For example, in the MLP with 27 hidden neurons the classification accuracy using the simplistic approach of the highest neuron indicating the class membership of the input pattern would lead to an accuracy of 86.05%. On the test set this is significantly better than the accuracy of 82.6% reported using the threshold-margin criterion. The MLP with 27 hidden nodes converged in 20 iterations with a mean squared error (MSE) of 0.02. Training was terminated because the decrease in the MSE in successive iterations after 20 iterations was not significant.

Many researchers have used the satellite data to benchmark their algorithms. In [3], the authors have reported the results of their implementation on satellite data for different algorithms like SMIFE2, MMIP, MIFS, PCA and LDA. Methods like PCA and LDA have been discussed and tested on satellite data as shown in [14]. In Table 2, we give a comparison of the classification accuracy obtained with our Parallel implementation of Levenberg-Marquardt training algorithm for Neural Network with that of the other reported results on satellite data. We denote our algorithm with the string PLMNN. Needless to say that the Hessian information utilized by the Levenberg-Marquardt algorithm for optimization plays a significant role in the quality of the solution obtained.

Algorithm name	Num. of hidden layer neurons	Classification accuracy
PLMNN	27	82.6%
SMIFE2	150	77.6%
MMIP	150	79.0%
MIFS	150	79.3%
IG	150	85.1%
GR	150	85.1%
PCA	150	78.9%
LDA	150	78.7%

Table 2: Comparison with other algorithms. The classification accuracy of PLMNN will be 86.05% if the highest neuron is considered for indicating class membership.

6. Speedup Measure

The next phase of our experimentation focused on measuring the time speedup of our parallel implementation of the training algorithm discussed in the previous section. As discussed in the implementation, the training algorithm is parallelized to compute Jacobian and Hessian matrices in parallel at each node of the cluster. Different number of nodes in the cluster were considered in different trials to run the training algorithm and the turnaround times of those trials for one complete iteration of the training process were recorded. This is shown in Table 3.

Num. of Nodes in Cluster	Training Time (in seconds)
1	1270.296
4	319.019
8	169.566
16	100.993

Table 3: Training times with varying number of nodes

Speedup is a measure of the cluster setup that shows the computational advantage gained by using several nodes in the cluster over the amount of computation needed to run the same program on a single processor. The speedup (S_n) value can be calculated as below.

$$S_n = \frac{T_1}{T_n}$$

where T_1 is the execution time on single processor and T_n is the execution time on a cluster. The implementation performed well on the cluster setup with considerably good computational speedups, as is evident from the Table 4.

Num. of nodes in Cluster	Speedup
4	3.982
8	7.49
16	12.578

Table 4: Cluster speedups with varying number of nodes.

When the number of nodes in the cluster is small (4 nodes) the speedup achieved with the implementation is linear. This characteristic remained consistently till up to 8 nodes. As the number of nodes is increased to a large number (16 nodes), the speedup observed is sub-linear. This is because as the number of nodes increases the amount of computation done by each node reduces and at the same time the communication overheads between the larger number of nodes increases. This observation is consistent with Minsky's conjecture discussed in [7].

The parallel implementation of the LM-algorithm was made in such a way that there was no inter-process communication needed between the nodes. However, the Hessian matrix and the gradient were accumulated at the master by receiving the corresponding values computed at the nodes. This accumulation process introduced some amount of communication between the master and the nodes. Hence with the increase in the number of nodes in the cluster the communication overhead also increased.

7. Conclusion

The proposed parallel neural network training algorithm for classifying large volume of training data speeds up the training process without the requirement for any special hardware. The speed up capability of this technique is evident from the implementation results. Since the data is distributed over all the nodes of the cluster, this method can also bring down the space requirements on a single computer to manageable limits.

Acknowledgments

The authors are thankful to Dr. Subrata Rakshit, Sc 'D', CAIR, Bangalore for providing some reference papers relevant to this work. The authors would like to thank Director, CAIR for facilitating and supporting this research.

References

[1] UCI Machine Learning Repository. `http://www.ics.uci.edu/mlearn/MLSummary.html`.

[2] MPI Primer / Developing With LAM. Technical Report Version 1.0, Ohio Supercomputer Center, The Ohio State University, November 1996.

[3] K. D. Bollancker and J. Ghosh. Linear Feature Extractors Based on Mutual Information. In *Proc. ICNN*, volume 3, pages 1528 – 1533, Washington D.C., June 1996.

[4] R. Duda and P. Hart. *Pattern Classification and Scene Analysis*. John Wiley and Sons, NY, 1973.

[5] S. E. Fahlman. An Empirical Study of Learning Speed in Backpropagation Neetworks. Technical report, June 1988.

[6] M. T. Hagan and M. B. Menhaj. Training Feedforward Networks with the Marquardt Algorithm. *IEEE Trans. on Neural Networks*, 5(6):989 – 993, November 1994.

[7] K. Hwang and F. A. Briggs. *Computer Architecture and Parallel Processing*. MGH, 1985.

[8] E. P. A. Jr. and T. J. Bartolac. Parallel Neural Network Training. In *Proc. AAAI Spring Symposium on Innovative Applications of Massive Parallelism, Stanford Univ*, March 1993.

[9] K. Levenberg. A Method for the solution of certain nonlinear problems in least squares. *Quart. Appl. Math*, 2:164 – 168, 1944.

[10] R. P. Lippmann. Pattern Classification using Neural Networks. *IEEE Communications Magazine*, 27(11):47 – 64, November 1989.

[11] N. Mache and P. Levi. Parallel neural network training and cross validation on a cray T3E System and Application to Splice Site Prediction in Human DNA. Technical report, Institute of Parallel and Distributed High Performance Systems, Stuffgart, Germany, 1995.

[12] D. W. Marquardt. An Algorithm for least squares estimation of nonlinear parameters. *SIAM J.*, 11:431 – 441, 1963.

[13] Message Passing Interface Forum. MPI:A Message-Passing Interface Standard. Technical Report Version 1.0, University of Tennessee, Knoxville, Tennessee, June 1995.

[14] J. R. Quinlan. *Programs for Machine Learning*. Morgan Kauffmann, San Mateo, CA, 1993.

[15] G. V. Trunk. A Problem of Dimensionality: a Simple Example. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI–1(3):306 – 307, 1979.

[16] K. Waldemark, T. Lindblad, V. Becanovic, J. L. Guillen, and P. L. Klingner. Patterns from the sky: Satellite image analysis using pulse coupled neural networks for pre-processing, segmentation and edge detection. *Pattern Recognition Letters*, 21:227 – 237, 2000.

[17] M. Whitbrock and M. Zagha. An Implementation of Backpropagation Learning on GF11, a Large SIMD Parallel Computer. *Parallel Computing*, 14:317 – 327, 1990.