About the Capability of Some Parallel Program Metric Prediction Using Neural Network Approach

Vera Yu. Goritskaya, Nina N. Popova
Department of Computer Science, Moscow State University

***Abstract*—Parallel program execution on multiprocessor system is affected by many factors. Often it is rather difficult to estimate how program would behave when running on some definite number of processors. Most tools designed for performance and other parallel program characteristics evaluation affect source code of a program. In this paper we consider neural network approach to predict job run time (when using a queue-based submission system). All analysis that is needed for the prediction can be provided without any knowledge about program source code.**

I. INTRODUCTION

Recently the complexity of multiprocessor systems rises increasingly. Systems like heterogeneous clusters and distributed systems that can include clusters and other multiprocessors as its components constantly appear. Since parallel program execution strongly depends on multiprocessor system characteristics and job scheduling mechanisms details, program performance evaluation becomes more and more complex problem.

Parallel program execution on multiprocessor system is affected by many factors. Often it is rather difficult to estimate how program would behave when running on some definite number of processors. Most tools designed for performance and other parallel program characteristics evaluation affect source code of a program. In this paper we consider neural network approach to predict job flow time (when using a queue-based submission system). All analysis that is needed for the prediction can be provided without any knowledge about program source code.

A. Metrics of the parallel program commonly used for performance evaluation

During performance evaluation of parallel programs different metrics are used [1]. The first one is the parallel run time or the flow time (the time from the moment when computation starts to the moment when the last processor finishes its execution). Generally the flow time is composed of three components: computation time, communication time and idle time. The parallel run time of a parallel algorithm depends not only on the size of the problem but also on the complexity of the interconnection network and the number of processors used.

One of the most commonly used metric is speedup, which captures the relative benefit of solving a given problem using a parallel system (the ratio of the time needed to solve the problem on a single processor to the time required to solve the same problem on a parallel system with ”p” processors; depending on the way in which sequential time is measured we can distinguish absolute, real and relative speedups). Theoretically, speedup cannot exceed the number of processors used during program execution.

Both above mentioned performance metrics do not take into account the utilization of processors in the parallel system. While executing a parallel algorithm processors spend some time on communicating and some processors can be idle. Then the efficiency of a parallel program is defined as a ratio of speedup to the number of processors. In the ideal parallel system the efficiency is equal to one but in practice efficiency is between zero and one.

The next measure, which is often used in the performance evaluation of parallel programs, is the cost of solving a problem by the parallel system. The cost is usually defined as a product of the parallel run time and the number of processors.

The next useful measure is the scalability of the parallel system. It is a measure of its capacity to increase speedup in proportion to the number of processors. Parallel system is scalable when the efficiency is the same for increasing the number of processors and the size of the problem.

To measure all above described metrics we need to measure the run time of sequential and parallel programs during experimental performance evaluation. Performing multiple runs of a parallel application usually can be extremely time-consuming in real multiprocessor systems with great number of users running their tasks on them and complex scheduling mechanisms [2].

We consider neural network approach to predict flow time of a parallel program. As long as parallel application execution is strongly connected with multiprocessor system on which this execution takes place we cannot restrict ourselves to considering only parallel application characteristics: besides, to predict run time of an applicationwe use both multiprocessor utilization information and job flow statistics. There is no need in application source code availability.

II. RELATED WORK

The problem of application performance prediction is very difficult and useful. Usage of artificial neural network

mechanism for application performance prediction is described in [3]. They implemented this algorithm using information about multiple executions of one definite application and got very good and accurate results.

Analytic modeling for performance tuning of parallel programs [2] provides predictive power, but is not widely used in practice, due to the difficulty of developing accurate models that work for real-world programs.

The other approach to task scheduling is described in [4]: authors estimate job execution time basing on program source analysis. The serious constraint is that application source code should be available - and sometimes it is impossible.

III. Our Approach

So, the problem is to predict time that job would spend inside the multiprocessor system (flow time). All data that is needed for the analysis can be gathered using standard utilities and commands of operating system or Job managing (scheduling) systems running on multiprocessor.

To be more precise, job completion moment is predicted. Neural network mechanism helps us to answer the question: "During what time interval job would be "executing ""inside" the computational system?). In this work it is considered that computer works in a batch regime (it means that any task can be preempted for another with higher priority can use necessary resources and be executed) and is managed by job scheduling system.

The problem we examine here is a part of another more complex and general problem – task scheduling in distributed systems, when it is needed to define the node on which concrete job would be executed.

To predict job characteristics the following input data are used:

- job submission moment,
- number of required processors,
- the highest possible execution time (after it job is to be interrupted),
- system loading at the moment of job submission,
- size of executable file,
- etc…

These data can be gathered without changing application code and using only standard OS and job managing commands.

Another feature of our project is that we gather "execution history". Using only "single execution" data can lead to very inaccurate results. To improve prediction accuracy the information about multiple application executions is used to predict waiting and execution time.

Here are some "historical" characteristics:

- average execution time for definite application with fixed required processors number,
- average required time for definite application with fixed required processors number,
- average size of executable,
- etc…

A. Neural network architectures

Artificial Neural Networks (ANNs) are commonly used for solving different prognosis problems; but their application for parallel systems is not enough investigated. Neural networks are a class of machine learning models that map a set of input parameters to a set of target values. Fig. 1 and Fig. 2 represent sample neural network architectures. The network is composed of a set of units that process the value at their inputs and produce a single scalar value. These values are then multiplied by a set of weights and communicated to other units within the network. Each edge on these pictures represents a weight, and each node represents a unit. The set of incoming edges at each unit indicates the set of values communicated to it. In these specific network architectures, the input parameters are placed at the first layer, and information flows from the left to the right. The units that produce the final predictions are output units, and those that receive input parameters are input units (input units simply pass incoming values to all of their outgoing edges). In addition, one or more layers of hidden units may be part of the network architecture. Hidden units process the outputs of other units, and, in turn pass their own outputs to another set of (hidden or output) units. The representational power of a neural network (the set of functions it can represent) can be increased by adding hidden units and layers.

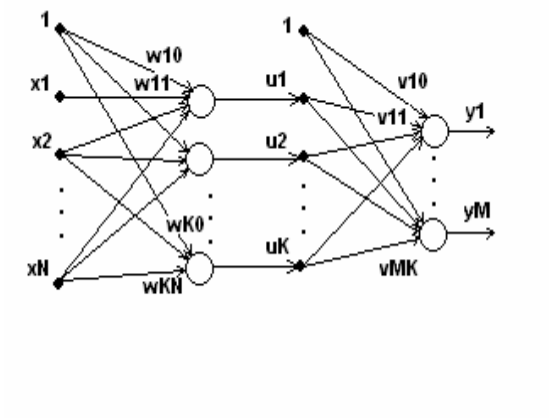

Figure 1. Multilayer feedforward network with sigmoid transfer function (1 hidden layer)

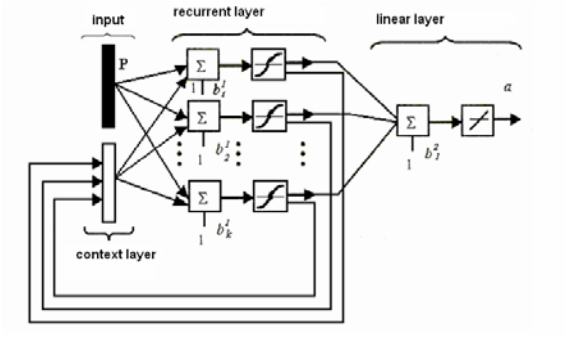

Figure 2. Elman backpropagation network

Fig. 1 shows a feedforward neural network with one hidden layer, and Fig. 2 – Elman backpropagation network. Elman network includes two hidden layers – recurrent and linear. These two kinds of Neural Networks were used for waiting and execution time prediction.

To make results of neural network work acceptable, input data must be properly processed before it would be used.
First, we define parameter which represents the program category – it depends on application execution time. In described case all tasks have been divided into four groups (according to rough evaluation of job execution time distribution):

- the tasks with average execution time less than 100 seconds;
- the tasks with average execution time more than 100 seconds and less than 1000 seconds;
- the tasks with execution time between 1000 seconds and 5000 seconds;
- the tasks with execution time more than 5000 seconds.

For each group neural networks with different parameters were implemented.

Then, samples with maximum and minimum execution time values for each job were removed from input data sets. (Reducing the noise level dictates that the difference between performance results from two different runs with the same input parameters be kept as small as possible.) Samples corresponding to cases when user rejected his or her job were also excluded.

At least, both inputs and targets were scaled so that they fall into the range of [-1, 1].

IV. EXPERIMENTAL RESULTS

A. *Training*

At each step during training, a new example is presented at the network's input layer. At each layer, every unit forms a weighted sum of the incoming values and associated weights. This sum is then processed by an activation function that produces the output of that unit. In this study, we use fully connected feedforward neural networks with the sigmoid function as the activation function. In case of feedforward network, for example, the operation of the sigmoid activation function on the weighted sum of inputs forms the unit result output. After a prediction on the current example, the weights in the network are updated in proportion to their contribution to the error.

On Fig. 3 and Fig. 4 you can see examples of training accuracy in cases of both considered networks for one of the task groups (networks were simulated using Matlab 7 package). These graphics represent

Table 1 represents training accuracy approached for different task groups in cases of both considered networks. Performance is the mean square error (backpropagation training method uses mean square error as a default performance function). As one can see there is no significant difference between feedforward and elman networks training results. As it will be described further, this can't be said about predicting accuracy (but not for percentage error).

Table 1. Training results comparison

Job group (according to the average execution time)	Performance (multiplayer feedforward network with sigmoid transfer function)	Performance (elman backpropagation network)
< 100 sec	$10^{-2} – 10^{-3}$	10^{-3}
100 – 1000 sec	10^{-4}	10^{-4}
1000 – 5000 sec	$10^{-4} – 10^{-5}$	10^{-4}
> 5000 sec	10^{-5}	10^{-5}

B. *Testing*

On Fig. 3 and 4 the predicting accuracy on testing input set is shown. Results are get using feedforward neural network (Fig. 3) and Elman neural network (Fig 4) (on all figures red line represents predicted values, blue line – real - actual – values).

The percentage error is almost the same for both networks (25% and 18% for feedforward and Elman networks respectively). It is rather high. Poor accuracy may result from the following factor: the training algorithm that adjusts network weights is not very suitable for reducing percentage

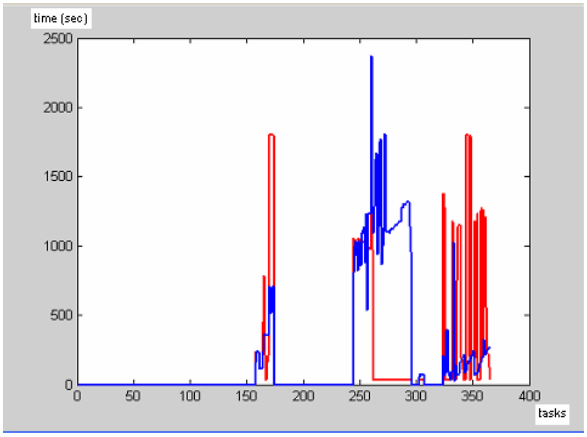

Figure 3. Execution time prediction (feedforward NN) on "Regatta"

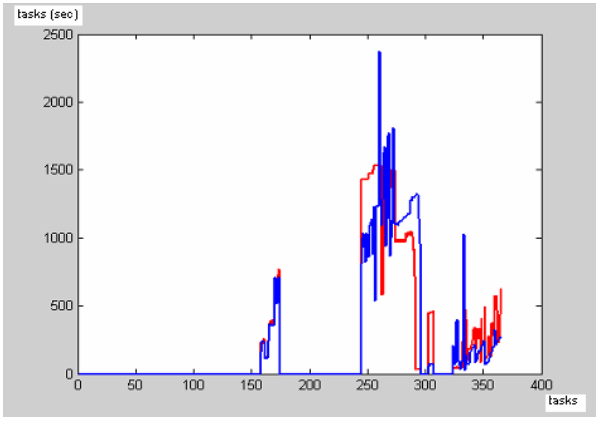

Figure 4. Execution time prediction (Elman NN) on "Regatta"

error. The backpropagation training algorithm aims to reduce absolute mean-squared-error. During training, examples on which the model makes higher absolute error are given greater

weight, even though this error may be small in relative terms as a percentage of the target value. Table 2 summarizes all prediction accuracy results.

All experiments described above were implemented on "Regatta" – 16-processor IBM eServer pSeries 690 [5].

Proposed approach (using Elman NN) was tested on another multiprocessor system – 10-processor cluster "Hill" (IBM eServer pSeries 360) – and we got the results (Fig. 5) which were much better than results we got using Regatta's flow. Explanation lies in task flow differences on Regatta and Hill: Hill is not so widely used as Regatta and its flow is more homogeneous.

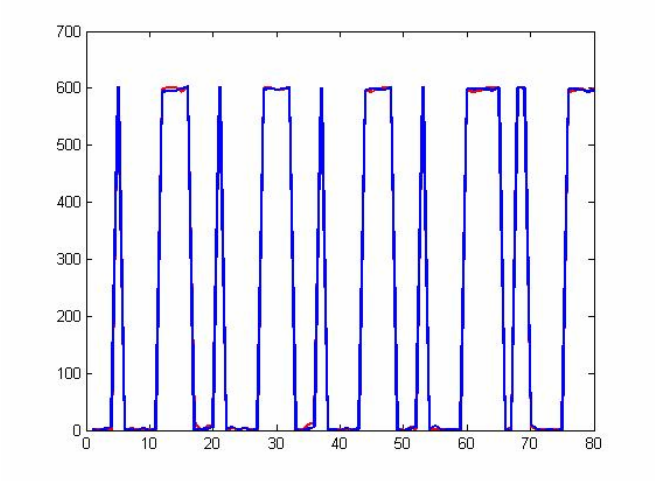

Figure 5. Execution time prediction (Elman NN) on "Hill"

V. CONCLUSIONS AND FUTURE WORK

It is planned to go further with neural network method of job characteristic prediction. Improving data processing methods possibly will lead to more accurate results. Using this approach new scheduling algorithms can be developed.

One more way of further development is to apply these predicting mechanisms on other multiprocessor platforms (not only Regatta and Hill) and especially it would be interesting to try this approach on Problem Solving Environments, where we have similar tasks and only input data may vary.

REFERENCES

[1] Kwiatkovski J. Evaluation of Parallel Programs by Measurement of Its Granularity // Proceedings of the International Conference on Parallel Processing and Applied Mathematics, 2001. P. 145 – 153.

[2] Crovella Mark E., Tomas J. LeBlanc. Parallel Performance Prediction Using Lost Cycles Analysis // Proceedings of Supercomputing'94, 1994. P. 600-610.

[3] Ipek, B.R. de Supinski, M. Schultz, S.A. McKee "An Approach to Performance Prediction for Parallel Applications" // Euro-Par 2005 Parallel Processing, Volume 3648, 2005, p. 196-205, ISBN 978-3-540-28700-1, 2005

[4] D.R. Gonchar "Heuristic algorithm of N tasks distribution to M processors" // Proceedings of "Methods and tools of information processing" conference, MSU, 2005, ISBN 5_89407-230_1, p. 537-540

[5] V.V. Balashov, A.P. Kapitonova, V.A. Kostenko, R.L. Smelyanskiy, N.V. Yuschenko "Method for estimating platform-optimized application execution time based on its high-level language source code" // Proceedings of 1st international conference "Digital signal processing and its applications", Volume IV, p. 203-220

[6] IBM eServer pSeries 690 (Regatta), http://www.regatta.cmc.msu.ru/