Preprint
UCRL-JC-136867

Comparative Study of Message Passing and Shared Memory Parallel Programming Models in Neural Network Training

J.E. Vitela, U.R. Hanebutte, J.L. Gordillo and L.M. Cortina

This article was submitted to
High Performance Computing 2000
Washington, DC
April 16-20, 2000

U.S. Department of Energy
Lawrence
Livermore
National
Laboratory

December 14, 1999

COMPARATIVE STUDY OF MESSAGE PASSING AND SHARED MEMORY PARALLEL PROGRAMMING MODELS IN NEURAL NETWORK TRAINING *

Javier E. Vitela
Instituto de Ciencias Nucleares
Universidad Nacional Autónoma de México
04510 México D.F., México

Ulf R. Hanebutte
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore CA 94551, USA

José L. Gordillo
Dir. Gral. Serv. Cómputo Académico
Universidad Nacional Autónoma de México
04510 México D.F., México

Lucila M. Cortina
Centro de Ciencias de la Atmósfera
Universidad Nacional Autónoma de México
04510 México D.F., México

Keywords: Radial Basis Neural Networks, Parallel Computing, Message Passing, Shared Memory, MPI, OPenMP, SHMEM, SGI/Cray Origin 2000.

Abstract

It is presented a comparative performance study of a coarse grained parallel neural network training code, implemented in both OpenMP and MPI, standards for shared memory and message passing parallel programming environments, respectively. In addition, these versions of the parallel training code are compared to an implementation utilizing SHMEM the native SGI/CRAY environment for shared memory programming. The multiprocessor platform used is a SGI/Cray Origin 2000 with up to 32 processors. It is shown that in this study, the native CRAY environment outperforms MPI for the entire range of processors used, while OpenMP shows better performance than the other two environments when using more than 19 processors. In this study, the efficiency is always greater than 60% regardless of the parallel programming environment used as well as of the number of processors.

*Work partially supported by CONACYT-3155P and CRAY-UNAM SC010399 projects. The work by Ulf R. Hanebutte was performed under the auspices of the U.S. Department of Energy by the Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

1 Introduction

In the last decade there has been a large research activity in neurocontrollers for a wide range of nonlinear applications. The ability of Artificial Neural Networks (ANN's) of approximating nonlinear mappings, have given the engineering community the possibility of designing nonlinear controllers that cannot be synthesized with traditional control techniques [Vemuri 1992]. However, although different convergence acceleration methods as well as several types of ANN's can be used, the high computing costs of their training procedures is still considered a major obstacle. Parallel computing [Foster 1994] is another alternative to expedite training of ANN's that can be used in addition to those previously mentioned. In this regard two fundamental paradigms for parallel software development exist: the message passing and the shared memory models. However, until recently, a parallel code (independent of the underlying model) developed using the native directives of one particular multiprocessor computer had to be extensively modified to run on a different machine. To overcome these obstacles, portable standards for shared memory and message passing environments are been developed by consortia comprised of vendors and research institutions: OpenMP and MPI, respectively [OpenMP Forum, Dagum et al. 1998, MPI Committe]. In particular shared memory parallel programming models are now emerging as a serious competitive environment

to message passing, due to the appearance of scalable shared memory multiprocessors platforms with embedded hardware support for cache coherence [Culler et al. 1999].

In this work we present a comparative performance study, utilizing a neural network training code, which has been implemented in both OpenMP and MPI. In addition, the OpenMP and MPI versions of the parallel training code are further compared to an implementation utilizing the native SGI/CRAY environment for shared memory programming, SHMEM [Feind 1995]. In what follows we briefly describe the main characteristics of these environments.

- OpenMP is a standard developed for shared memory multiprocessor computers. In these platforms every processor has direct access to the memory of every other processor which allows them to directly load or store any shared address. It is, in essence, a set of compiler directives to express shared-memory parallelism; it allows incremental parallelization of existing sequential codes and can be used for loop-level and for coarse grained parallelism. It allows the programmer the possibility of declaring any pieces of memory as private to each processor which greatly simplifies the development of parallel programs. Further, it may be used by application programmers who want a quick but not very effective parallelization. It is the general consensus among researchers in this field that OpenMP environment is more relevant when used in: codes with large shared databases, which needed to be stored only once per node in OpenMP rather than once per processor in MPI; codes with tasks that actually can benefit from loop-level parallelism in addition to domain-level parallelism; and in MPI-limited facilities.

- MPI (Message Passing Interface), also an industrial standard, is a message passing environment which assumes a processor cluster with distributed memory able to work cooperatively. This set of processors runs concurrently copies of a single program and use MPI library calls for sending and receiving messages between processors as well as for tasks synchronization. However, MPI which is intended mainly for coarse grained parallelism, has the disadvantage that the program must be entirely decomposed for parallel execution, and there exist no incremental way to parallelize an application. Nevertheless, in addition to be an industrial standard, another major advantage consist in the fact that it can be used efficiently in a multiprocessor computer or in a cluster of workstations , and furthermore, it can coexists with OpenMP and SHMEM. Hence, it allows the possibility of developing efficient parallel programs able to run in clusters of shared memory multiprocessors computers (i.e. SMP's), using OpenMP within each individual system and MPI whenever inter-SMP communication is required.

- SHMEM, on the other hand, is a SGI/CRAY native set of routines that take advantage of the logically shared memory in systems such as the Origin2000 and the CRAY T3D. A logically shared memory is one which allows any processor unit in a multiprocessor platform to access the memory of any the other processor without the direct involvement of this later unit. It consists of data passing library routines, similar to those used in message passing, designed to maximize bandwidth and minimize data latency, thus minimizing the overall computation overhead of data transfer requests. It is intended exclusively for coarse grained parallelism and although, in contrast with MPI, it contains only a limited number of routines, these are enough for a large number of different applications.

In order to compare these three environments, we limit our study to coarse-grained parallelism which is based on the domain decomposition approach. In this technique, the parallel code goes through essentially the same steps as the sequential code and use a set of parallel directives to perform data transfers and synchronization between processors. The study exploits a neural network training code developed for the control of dynamical systems which uses Radial Basis Neural Networks (RBNN's) are an alternative type of ANN possessing the best representation property [Pogio et al. 1990], have higher convergence speeds than the conventional feedforward multilayer neural networks, and under some conditions in the training set they are also free of local minima. This code makes use of RBNN's composed of Gaussian nodes in the hidden layer and sigmoidal units in the output.

The physical system under consideration is a zero dimensional tokamak fusion reactor model with the design parameters of the ITER-EDA group. It is desired to stabilize this system at subignited nominal operation conditions for a wide range of energy confinement times. The plasma is composed by 50:50 DT, helium ions, a small fraction of high-Z impurities and electrons, and it is assumed that all particles share the same temperature at all times. Heating

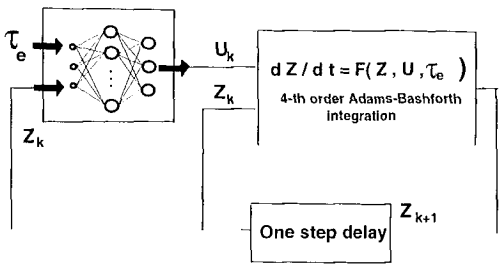

Figure 1: Radial basis neural network-dynamical system configuration.

takes place by the thermalization of the alpha particles produced by fusion; losses are taken into account by bremsstrahlung radiation and by transport mechanisms through the energy confinement time. The stabilization is performed with a feedback control law using a RBNN, in which the input is constituted not only by the current values of the state variables but also by the energy confinement time, as shown in Fig 1. The output of the network, i.e. the control actions include the modulation of the DT refueling rate, the injection of a neutral He-4 beam and an auxiliary heating power constrained to take values between a minimum and a maximum levels. [Vitela et al. 1998, Vitela et al. 1999a].

The study was performed on the SGI/CRAY Origin 2000 multiprocessor platform located at UNAM during dedicated user operation, where access to the entire system by other users was disabled; This system is configured as two independent modules, one with 8 and the other with 32 processors for a total of 40 processors. All processors in the system are 195 MHz MIPS R10000 processors, each of which has a 4 MB secondary cache and 512 MB of memory [UNAM *www* document].

2 Parallel Neural Network Algorithm

In previous works the authors discussed in detail the development of a parallel neural network training code for control of dynamical systems developed with the MPI message passing environment [Hanebutte et al. 1998, Vitela et al. 1999b]. A performance analysis compared three different load distribution schemes: the block, the cyclic, and a predictive bin-packing load assignment. It was concluded that the predictive bin-packing algorithm utilizing an estimate of the work load based on a sliding average over the last five training iterations yields higher speedups and efficiencies than the other two schemes. This is due to the fact that not only the total work load can not be uniformly distributed among the processors but in addition it can not be known a priori. Hence in the present study we shall use only the predictive bin-packing scheme which distribute quasi-optimally the work load estimated using the actual load generated over the last five training iterations.

As discussed in previous works the ANN's training is based in the minimization of an error $\mathcal{E}$, which is produced by independent contributors, i.e. $\mathcal{E} = \sum_{m=1}^{M} \mathcal{E}_m$, generated by the individual trial trajectories generated by the the algorithm, using the ANN-dynamical system configuration shown in Fig. 1. It is important to point out that in the present work besides using a RBNN instead of a standard feedforward ANN, we also generate the dynamical system trajectories using a 4-th order Adams Bashforth integration scheme instead of the simple Euler method used previously. The parallelization strategy Is based on the fact that the gradient of the total error $\nabla\mathcal{E}$, which is needed in the numerical search for the minimum, is the sum of the gradients of the individual errors of each of the M independent trajectories,

$$\nabla\mathcal{E} = \sum_{m=1}^{M} \nabla\mathcal{E}_m \quad . \tag{1}$$

Thus, assuming that we have a set of P processors, labeled $p = 0, 1, \ldots P-1$, which are available to contribute to the calculation of $\nabla\mathcal{E}$, given a load balance scheme, the following parallel algorithm is devised,

† In OpenMP only one process exist during Steps 1 - 3 below

- Step 1: Task assigned to processors $p = 0$ (also called *root*): Randomly select the initial values of the set of weights that specify the NN, and use MPI (SHMEM) library calls to broadcast these values to all the other $P-1$ processors.
- Step 2: Task assigned to processor $p = 0$: Divide an admissible region of the phase-space in a number M of cells.
- Step 3: Task assigned to processor $p = 0$: Select M initial states by random sampling each one of the cells in phase space. Use MPI(SHMEM) library calls to broadcast these states to all the other $P-1$ processors.

† OpenMP creates a parallel region with P processors to perform tasks in Steps 4 - 6.

- Step 4: According to a given load distribution scheme each of the processors is assigned a subset of the M initial states. For each initial state the corresponding trajectories are generated. A record of the maximum number of time steps in each trajectory is made and will be used by the load balance scheme at the beginning of this Step in future iterations, as will be discussed further below.

- Step 5: Using dynamic backpropagation [Piché 1994] each of the P processors calculates and stores, the gradient of the individual error $\mathcal{E}_m$ associated with the subset of the M trajectories that were assigned to the processor by the load distribution scheme. Each process determines the number of trajectories which satisfy the convergence criterion,

- Step 6: A test for convergence in each of their corresponding subsets is performed by each processor and the results are added up and sent, using MPI (SHMEM) reduction calls, to processor $p = 0$, which determines whether or not global convergence has been achieved. If the global convergence criteria is satisfied one should stop; otherwise proceed to the next step. In OpenMP this procedure is automatically done because the corresponding variable is declared a *reduction* variable and thus at the end of the parallel region the partial values are added up.

† In OpenMP Step 7 below is not necessary since the corresponding variable was defined as shared.

- Step 7: Using the MPI (SHMEM) global sum operation, the partial gradient of the errors $\nabla\mathcal{E}_m$, associated to each individual cell, are share by all P processors.

† OpenMP closes the parallel region and the master process proceeds with Steps 8-9.

- Step 8: Each processor proceeds to determine, by adding these components in the same ordered sequence, the gradient of the total error $\nabla\mathcal{E} = \sum_{m=1}^{M} \nabla\mathcal{E}_m$ avoiding thus the differences due to roundoff errors, occurring when no care is taken in the order in which the individual terms are added. This ensures us that the results will be independent of the number P of processors involved in the computation since the sum of floating point numbers is not an associative operation.

- Step 9: All processors individually use $\nabla\mathcal{E}$ to determine the new conjugate gradients direction, and

† OpenMP opens once more a parallel region with P processors to perform together Step 10 below.

- Step 10: The processors use the conjugate direction and search for the minimum along this direction, in a cooperative fashion.

† OpenMP closes the new parallel region and the master process continues with the rest of the steps.

- Step 9: Updating of the weights is done in each processor.

- Step 10: Repeat steps 3-9 until the training of the NN is successfully completed, i.e. when the entire set of M trajectories, each of which starts from a different cell, reaches the target $\mathbf{z}_t$ within the error range ϵ.

In Fig.2 we show the flow diagram of the parallel code described above. The locations where communications are required in the message passing environments but not in the shared memory models are pointed out.

3 Performance analysis

For the problem we consider here, the phase space was divided into 108 cells. These cells cover with 27 three-dimensional cubes the allowable perturbation region around the nominal operating values of the electron density, the fraction of helium ash and the plasma temperature defining the state of the thermonuclear system we are concerned here; in addition the range of interest of the energy confinement time was divided in 4 intervals, from where 4×27 different transient trajectories were generated. Thus the granularity of the problem is restricted to a maximum of 108 processors. In this particular problem the code converges in 151 iterations; however for the purposes of keeping the computing time in this study within reasonable values, we restricted the number of iterations to only 30.

We performed a preliminary theoretical study in which the load balance associated with each trajectory was estimated as proportional to the number of time steps contained in it. Since this number can only be known after the trajectory was actually generated, we used a sliding average bin packing load distribution scheme using the last five iterations of the algorithm. Thus, the assumed linear relationship between

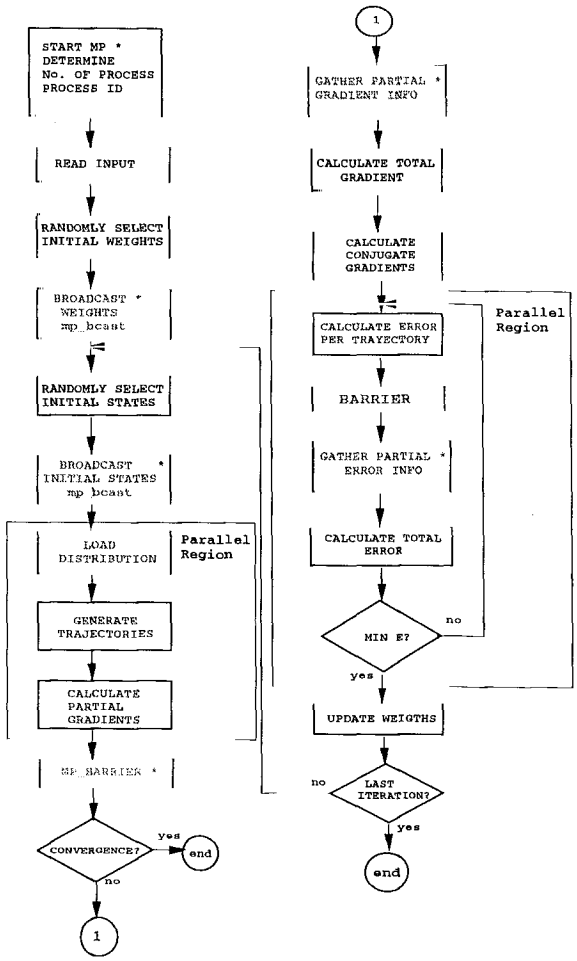

Figure 2: Flow diagram of the parallel code showing locations with special communication requirements; the * show communications not required in shared memory.

the work load and the number of time steps in the trajectories provide us with a reference framework, although communications and computations requirements like the integration scheme used to generate the trajectories introduce nonlinear effects which impede these theoretical predictions. The complete 151 iterations, required 5876 seconds for the entire run, and the fraction of the time spent in the intrinsically sequential part of the code is approximately 0.2% . The high percentage of the total time doing parallelizable work show that the parallel code described in the previous section may significantly reduce the training time of the RBNN.

The performance of the code was measured by the speed-up and the parallel efficiency. The speed-up is defined as the ratio between the execution time required by one processor and the time required by p processors; the parallel efficiency on the other hand

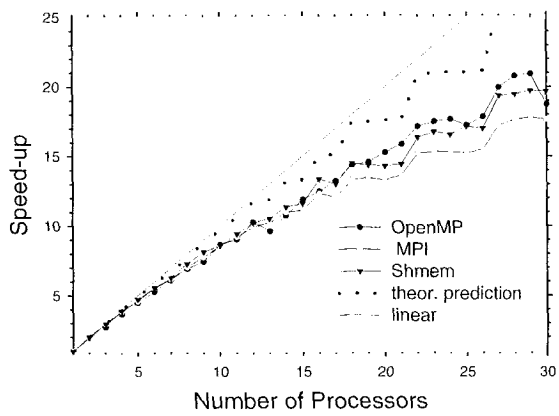

Figure 3: Speed-up's obtained with the three parallel environments and the estimated prediction as discussed in the text.

measure the fraction of the total execution time of all processors which is spent performing computations.

In Figs. 3 and 4 we show the speed-up and the efficiency of the entire code obtained by timing the first 30 iterations as function of the number of processors; the figures include also the theoretical estimation discussed above. It is observed that SHMEM, a shared memory data passing environment, clearly outperform MPI in the entire range of processors, furthermore it shows a similar or a slightly better performance than OpenMP when using 18 processors or less and is outperformed by OpenMP afterwards. MPI on the other hand shows a similar behavior to OpenMP for small number of processors but it is outperformed when using more than 14 processors. The three programming environments yield efficiencies higher than 60%, for the entire range of processors available.

4 Conclusions

A comparative study of shared memory and message passing environments was performed using MPI, OpenMP and Shmem in a coarse grained parallel RBNN training code for control of dynamical systems, for a number of processors ranging between 1 to 30 of a SGI/CRAY Origin 2000 system. Since the actual work load is not known a priori the code utilize an estimate of the work load that is based on a sliding average over the last five training iterations. An almost monotonic increase in the speed-up for the range between 1 to 30 processors for the three programming environments is observed. As expected

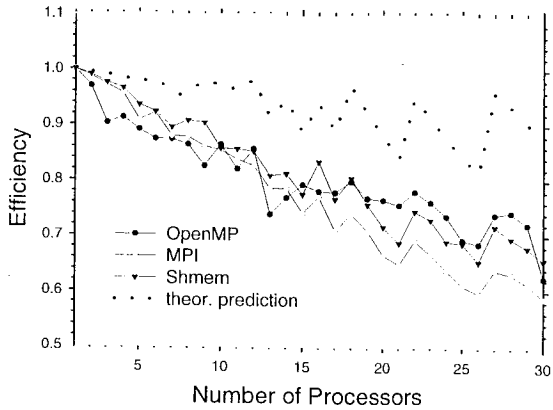

Figure 4: Behavior of the efficiencies obtained with the three parallel environments used in this study showing also the estimated prediction.

the native shared memory directives for parallel programming of the SGI/CRAY platforms outperforms the industrial standard MPI although it shows a similar behavior when a relative small number of processors is used. OpenMP on the other hand outperforms MPI clearly when using 16 or more processors, while demonstrating a comparable performance for smaller number of processors. SHMEM shows a better performance up to the 19 processors case, while being slightly outperformed afterwards. Finally, it must be pointed out that in this specific application OpenMP outperforms MPI on the SGI Origin even so a domain/task based parallelization model is applied here. A comparison of the presented results with those of a code version which includes loop-level parallelism is the subject of forthcoming work.

Acknowledgments

The authors gratefully acknowledge the help provided for the dedicated time tests by the Departamento de Supercómputo at the National University of México in México City.

References

Culler D.E., Singh J.P., and Gupta A., *Parallel Computer Arquitecture.* Morgan Kaufmann Publ. San Francisco CA 1999.

Dagum, L. and Menon, R., *OpenMP: An Industry-Standard API for Shared-Memory Programming.* IEEE Computational Science and Engineering, January-March (1998).

Feind K. , *Shared memory Access (SHMEM) Routines.* Cray User Group Spring 1995 Conference, March 13-17, Denver CO (1995). http://reality.sgi.com/kaf_craypark

Foster, I., *Designing and Building Parallel Programs.* Addison Wesley, Massachussets 1994.

Hanebutte, U.R., Vitela, J.E. and Gordillo, J.L., *A Parallel Neural Network Training Algorithm for Control of Discrete Dynamical Systems..* Proc. High Performance Computing HPC'98 (A. Tentner Ed.) pp.81-87 (1998). Boston MA April 5-9, 1998.

MPI Committe, *The Message Passing Interface (MPI) Standard.* http://www.mcs.anl.gov/mpi

OpenMP Forum, *http://www.openmp.org.*

Piché, S.W., *Steepest Descent Algorithms for Neural Networks Controllers and Filters.* IEEE Trans. on Neural Networks, Vol.5 No.2, 198 (1994).

Pogio, T. and Girosi, F., *Networks for Approximation and Learning.* Proc. IEEE, Vol 78, No. 9, pp. 1481-1497 (1990)

UNAM *www* document, Department of Supercomputing DGSCA, *SGI/Cray-Origin 2000 at UNAM.* http://www.super.unam.mx

Vemuri, V.R. (Ed.), *Artificial Neural Networks: Concepts and Control Applications.* IEEE Computer Society Press, Los Alamitos, CA, 1992.

Vitela, J.E. and Martinell, J.J., *Stabilization of Burn Conditions in a Thermonuclear Reactor using Artificial Neural Networks* . Plasma Phys. Contr. Fusion, Vol. 40 pp. 295-318 (1998).

Vitela, J.E. and Martinell, J.J., *Stabilization of Fusion Reactor Burn Conditions with Radial Basis Neural Networks* . 26-th EPS Conference on Contr. Fusion and Plasma Phys., Vol. 23J p. 1273 (1999). 14-18 June 1999, Maastricht, The Netherlands.

Vitela, J.E., Hanebutte U.R. and Gordillo, J.L., *Performance Analysis of Parallel Neural Network Training Code for Control of Discrete Dynamical Systems..* Inter. Journal Computer Research, 1999 (in press).