IMPLEMENTATIONS OF ASYNCHRONOUS SELF-ORGANIZING MAPS ON OPENMP 
AND MPI PARALLEL COMPUTERS

Jie Hu  
Department of Computer Science  
St Cloud State University  
St Cloud, MN 56377  
jhu@stcloudstate.edu

Liqiang Zhang  
Open Access Technology Int'l Inc.  
Minneapolis, MN  
lisa.zhang@fis-lps.com

Jing Cheng  
Open Access Technology Int'l Inc.  
Minneapolis, MN  
jing.cheng@oati.net

Abstract

In [1], we presented an asynchronous parallel algorithm for self-organizing maps based on a recently defined energy function which leads to a self-organizing map. We generalized the existing stochastic gradient approach to an asynchronous parallel stochastic gradient method for generating a topological map on a distributed computer system (MIMD). We theoretically proved that our algorithm was convergent and the simulations showed our algorithm was effective. In this paper, we implement this algorithm on practical parallel computers with two different types: openMP and MPI, in the Supercomputing Institution at University of Minnesota. By analyzing the experimental results, we demonstrate the convergence, efficiency and speed-up of our algorithm.

1. INTRODUCTION

Self-Organizing Map (SOM), which was presented by Dr. Kohonen in 1989 [2], and a number of existing variants of SOM are prominent unsupervised neural network models. They are useful in a variety of engineering and scientific applications, including speech and pattern recognition, image recognition, signal processing, connectionist natural language processing, high-dimensional data analysis, and image compression. In addition, SOM is also widely used in visualization as a dimension (feature) reduction tool.

However, with the massive advance of the growth of the amount of data, the execution time required to train the SOM map is very demanding. Usually, a classical SOM algorithm needs hundreds of thousands of training steps to reach the convergence. One approach to improve the execution time of SOM is through parallelism on parallel computers.

1.1 Asynchronous Stochastic Algorithm for SOM

SOM can be thought as a neural network that contains two layers of cells – an input layer and a network layer. In the network layer, the cells or nodes $i$ are arranged in lattice type of array. This lattice type of array can be defined as a rectangular, square, hexagonal, or even irregular. In this paper, the lattice is defined as a square. All the cells are associated with a set of weight vectors denoted by $\mu_1, \mu_2, \ldots, \mu_N \in \mathbb{R}^n$, where $N$ is the number of cells, $n$ is the dimension of weight vectors. The input layer is used to hold the data samples that the networks will be learning from. Just like the network layer, the input data is also represented by a set of weight vectors $x=[x_1, x_2, \ldots, x_n]^T \in \mathbb{R}^n$. The dimensions of the weight vectors in both layers could be high dimensional. The number of dimension of both input vector and the network vector should be the same. Following the notation in [3], a sample updated procedure of SOM involves

$$\mu_i(t+1) = \mu_i(t) + \alpha(t)(x - \mu_i(t)), \quad \text{if } i \in N_c$$

(1)

otherwise $\mu_i(t+1) = \mu_i(t)$, where $c$ is the location of the closest neuron to the data $x$, $\alpha(t)$ is the step size which must satisfy additional conditions for stochastic algorithms (see [2], [3]) and $N_c$ is a neighborhood of neuron $c$. The development in [3] works with the same set of neurons and sequence of observations as the SOM described above, but uses an energy function which leads to a self-organizing map. In [1] we generalized this algorithm to an asynchronous parallel stochastic algorithm for SOM on a distributed computer system (MIMD).

To describe the algorithm, we need some definitions. Let’s assume that there is a set of time $T=\{0, 1, 2\ldots\}$ at which one or more components $\mu_i$ of $\mu=(\mu_1, \ldots, \mu_N)$, are updated by some processors of a distributed computer system, where $N$ is the number of neurons. Let a distributed computing system have $N$ processors. At each time $t$, each processor $i$ has a vector $\mu_i(t) \in \mathbb{R}^n$ associated with it. Let
$T'_i$ = Set of times at which $\mu_i$ is updated.

We require that for every $i$ and $t \geq 0$, at least one of the elements of the set $\{t, t+1, \ldots, t+B\}$ belongs to $T'_i$. We further require that information used by a processor be outdated by at most $B$ time units. $B$ is called the asynchronous with these restrictions. For the asynchronous case, the processor updating the vector $\mu_i$ may not have access to the most recent values of $\mu$ that have generated by other processors. So the asynchronous parallel stochastic algorithm for SOM can be described as follows:

$$\mu_i(t+1) = \mu_i(t) + \alpha(t)s_i(t), \forall t \in T'_i, \forall i = 1, N$$

(2)

where $s_i(t)$ is the update direction

$$s_i(t) = \sum_{j=1}^{N} \sigma_{ij} \left( x_k(t) - \mu_i(t) \right)$$

(3)

$$\sigma_{ij} = \frac{g_{ij} \delta_{s_{ij}}(t) \delta_{s_{ij}}(t)}{\sum_{g_{ij} \delta_{s_{ij}}(t) \delta_{s_{ij}}(t)}}$$

(4)

and $s_{ij}(t)$ are times satisfying

$$t - B \leq s_{ij}(t) \leq t, \forall t \in T'_i$$

(5)

and $s_{ij}(t) = t$ for all $i$ and $t \in T'_i$. At all times $t \notin T'_i$, $\mu_i$ is left unchanged, giving

$$s_i(t) = 0, \forall t \notin T'_i$$

(6)

and $g_{ij} = 1$ when neurons $i$ and $j$ are neighbors and zero otherwise, the input data set is $\{X_1, X_2, \ldots, X_M\}$, and

$$p(x_i, \mu_i) = \frac{1}{\sqrt{2\pi \tau}} \exp \left( -\frac{\|x_i - \mu_i\|^2}{\tau} \right)$$

(7)

and $\alpha(t)$ is the step size which must satisfy additional conditions for stochastic algorithms.

2. IMPLEMENTATIONS OF THE ALGORITHM BY OPENMP AND MPI

The implementations are preferred in a two-dimensional grid. For simplicity, we have several assumptions during the testing. First, the weight of both input data samples and neurons are randomly initialized as two-dimensional, non-negative floating-point numbers uniformly distributed over the range of $[0.0, 1.0]$. Second, the grid that is used to arrange the neurons is defined as a square or rectangle region. To be more specific, this region is located around the origin of a two-dimensional coordinate system and extended from 0 to 1 in both dimensions.

2.1 OpenMP Implementations

OpenMP is a programming interface for developing portable programs of shared memory systems and distributed shared memory systems [4]. It started in 1997 and was developed by the OpenMP Forum, a group of hardware and software vendors and academic research institutions. Most major hardware vendors of shared memory systems support OpenMP, including Compaq, Hewlett-Packard, Intel, IBM, Kuck & Associates (KAI), Silicon Graphics, Inc. (SGI), and Sun. The core of OpenMP API is a set of specification standard for compiler directives, library routines, and environment variables that can be used to specify shared-memory parallelism in Fortran and C/C++ programs [4].

The algorithm is implemented by using openMP on the 96-processor Silicon Graphics, Inc (SGI) Origin 3800s and the 224 IBM Power4 processors pSeries 690 (Regatta) at the University of Minnesota Supercomputing Institution (MSI).

2.1.1 Demonstration of the convergence

Since the algorithm has been proved to be convergent theoretically, the graphic results should demonstrate a well-formed grid after certain test runs, which indicates the neurons near to each other in physical locations also have their corresponding input data close (similar) to each other in the input space. In Figure 1, the evolution from a random start of a map of 10 × 10 nodes to a well-formed order map is displayed. The test is executed under threads. As the training proceeds, the grid structure, in which the neurons are organized, becomes more and more visible.
2.1.2 Execution time and Speed-up

Figure 2 shows two test runs of a 7x7 and 8x8 network executed on Origin up to 14 threads with the same parameters and data samples as in Figure 1.

![Figure 1. Demonstration of the convergence of a 10\times10 network with 500 data samples (T = 0.2 and A = 3)](image1)

![Figure 2. Execution time for a 7x7 network and a 8x8 network with the same parameters and data samples as in Figure 1.)](image2)

![Figure 3. Speedup of the results in Figure 2.](image3)

The speed-up is measured by the following definition

\[ S = \frac{T_1}{T_p} \]

where \( T_p \) denotes the execution time of the parallel program solving a given problem, when executing on \( p \) processors \( (p \geq 1) \), and \( T_1 \) denotes the execution time of the sequential program. Figure 3 demonstrates the speed-ups of the results in Figure 2. We notice that the execution time keep decreasing and the speed-ups keep increasing as we increase the number of thread from 1 to 8. At this point, the program is scalable. When the program is executed with 9 and more threads, the execution time tends to increase and the speedup is decrease. This shows the program is experiencing a thread overhead. Thread overhead happens when the size of the data needed to be processed is relatively small and too many threads have been requested from the system to execute the program than necessary. The system spends extra time to fork those threads that are not actually needed by the executions. Therefore it is not always the case that the more thread we request, the less execution time we will achieve. To overcome the overhead, we need to make the size of the parallel work construction significant enough.

Figure 4 is the execution times of network from 6x6 to 20x20 executed with thread up to 16 tested on IBM Regatta machine. Figure 5 is the corresponding speed-up results.

![Figure 4. The performance chart for 6x6 to 20x20 networks executed with threads up to 16.](image4)
2.2. MPI Implementations

Message-Passing Interface (MPI) is a widely used message-passing library including a collection of routines for facilitating communication among the processors in a distributed computing environment. It is a message passing model and not a compiler specification. MPI was designed to permit the development of parallel software libraries and to provide access to advanced parallel hardware by a group of computer industry major vendors such as IBM, Cray, Intel, Convex, etc in early of 1990s. Now it matures as programming paradigm.

The asynchronous parallel SOM algorithm is implemented by using MPI on IBM Power 4 system, which is a group of several shared-memory nodes with Power4 processors. There are 312 Power4 processors available to users of the system. To compare the results, we used the same data samples and parameters for all the tests.

2.2.1 Demonstration of the convergence

Figure 6 displays the evolution from a random start to a well-formed feature map of 6x6 nodes with 50000 input samples. The execution times are also shown in each graph.

Figure 7 shows four final convergent graphs of the asynchronous parallel SOM implementations with four different networks of 6x6, 7x7, 8x8, and 10x10. From this figure, we can see that the featured convergent SOM map is becoming better formed as the number of network nodes increases.

2.2.2 Execution time and Speed-up

To analyze the performance, three different tests were performed. They are sequential, synchronous, and asynchronous parallel implementations. The execution time and speed-up are displayed in Figure 8 and 9.
From Figure 9, we notice that the speedups keep increasing as the network size changes. This is because parallelism achieves more efficiency when working on more complex or bigger problems. It also tells us that asynchronous implementation has better performance than synchronous one since synchronous implementation experienced some synchronization overhead.

3. CONCLUSIONS

In this paper, the asynchronous parallel SOM algorithm presented in [1] is analyzed by implementations on practical parallel computers with two different types: openMP and MPI. The testing results show that the algorithm is convergent and efficient.

We are grateful for resources from the University of Minnesota Supercomputing Institute.

4. REFERENCES


