Solving Very Large Traveling Salesman Problems by SOM Parallelization on Cluster Architectures

Hannes Schabauer
Department of Statistics and Decision Support Systems
University of Vienna
Universitätsstraße 5, 1010 Vienna, Austria
hannes.schabauer@univie.ac.at

Erich Schikuta, Thomas Weishäupl
Research Lab Computational Technologies & Applications
University of Vienna
Rathausstraße 19, 1010 Vienna, Austria

Abstract

This paper describes how to solve very large Traveling-Salesman Problems heuristically by the parallelization of self-organizing maps on cluster architectures. The used way of parallelizing is a sophisticated Structural Data Parallel approach based on the SPMD model. We distinguish between a non-sophisticated and a sophisticated approach for efficient and simple parallelization of the SOMs.

1 Introduction

The Traveling-Salesman-Problem (TSP) [2] is a classical problem in algorithm theory. The goal of the problem is to find the shortest route of a salesman starting from a given city, visiting a number of other cities and finally arriving at the origin city. The TSP belongs to the class of NP complete problems.

Based on a deterministic approach the world-record-setting traveling salesman solution algorithm is by Applegate, Bixby, Chvatal, and Cook [1], which has solved instances as large as 24,978 cities to optimality.

Trying to solve the course of exponentials parallel implementations of the TSP were realized [4].

However for practicability reasons, specifically for large numbers of cities, also heuristic approaches for solving the TSP are very popular, which try to produce an optimal or close to optimal solution.

Generally neural networks are well suited for solving problems, which are hard to catch in mathematical models. However, the usage and employment of neural networks in such application domains is often dependent on the tractability of the processing costs. The problem domains for the employment of neural networks are increasing and also the problems themselves are getting larger and more complex. This leads to larger networks consisting of huge numbers of nodes and interconnection links, which results in exceeding costs for the network specific operations, as evaluation and training. Especially the cost intensive training phase of a neural network inherits a major drawback, due to the situation that large numbers of patterns (input and/or target values) are fed into the network iteratively.

One solution to this problem is the employment of general purpose parallel or distributed hardware architectures, combined with advanced development tools. In this paper we present a solution to the TSP by using a parallel, SPMD based, implementation of the Kohonen feature map on a cluster architecture.

The paper is structured as follows. In the next section a survey on the parallelization of neural networks is given and the used SPD approach is presented. Section 3 describes how the Traveling-Salesman-Problem can be solved heuristically by self-organizing maps and a mathematical characterization is given. In section 4 the parallelization approaches of the TSP-SOM are presented to be followed for the unsophisticated and sophisticated approach are given. An in-depth run-time analysis is done in section 5 which is followed by the conclusions.

2 Parallelization of Neural Networks

In the literature many approaches for the parallelization of neural networks can be found. These were done on various parallel or distributed hardware architectures, as workstation clusters , large-grain supercomputers or specialized neural network hardware systems. A good survey can be found in [9], which presents a categorization of the various levels of parallel neural network simulation.

Generally two forms of software techniques for the parallelization of neural network systems are distinguished, control parallel and data parallel simulation, which can also be described as decentralized and centralized (regular) control flow.

According to [9] the data parallel programming technique shows centralized control, but decentralized data distribution. Three different forms are described, pipelining, coarse structuring, and fine structuring, according to the grade of granularity of parallelization. Basically all three groups are based on a topological partitioning of the neural network components onto the underlying parallel architecture. That represents a distribution of neural network elements (like neurons, links, etc.) among the available processors of the underlying parallel hardware architecture, like node-per-layer, single-node, or systolic arrays.

We believe that this categorization [9] is too limited. We see a further way to parallelize a neural network, which leads to two different forms of data parallelism, topological and structural data parallelism.

Besides the mapping of logical neural network elements to processing nodes, as in topological data parallelism, it is also feasible to map data structures representing neural network information containers (as weight matrices, error value structures, input vectors, etc.) onto processing elements according to a data parallel scheme.

2.1 Parallelization Approach

Generally, a number of problem decomposition techniques can be used to construct a distributed cluster application. These are pipeline or data flow decomposition, functional decomposition and data-parallel decomposition.

For the underlying work we used the data-parallel decomposition based *Structural Data Parallel* (SDP) approach [7] for the parallel simulation of the SOM. This approach was developed to make parallel neural network simulation as simple as sequential one. It allows users, who are inexperienced in high-performance computing to increase the performance of neural network simulation by parallelization.

This approach is based on the *Single-Program-Multiple-Data* (SPMD) programming model and utilizes the highly specialized programming capabilities of high performance languages for the parallelization process.

The development process of the simulation system is reduced to the simple design of a sequential program, which is attributed by data distribution information. Thus the difficult task of physical parallelization is shifted to the programming environment and the compiler of the high performance language. This approach is equally well applicable for simulation on conventional MPP supercomputer and, presented in this paper, on cluster architectures.

The structural data parallel approach is based on four steps, which lead programmers from a sequential description to a parallel and efficient implementation of the neural network:

- Sequential coding of the neural network.
- Data structure identification.
- Distribution schema selection.
- Automated parallel code generation and analysis.

3 Solving the Traveling Salesman Problem by Self-Organizing Maps

3.1 Basic Idea

The Traveling-Salesman-Problem describes the situation of a traveler, who has to visit a number of n cities on his tour, where the start city is also the final city, i. e. he travels a circle. The goal is to find the shortest path.

To find a solution for a large number of cities is due to the exponential number of $n!$ solutions not tractable. Therefore for large numbers a heuristic approach has to be chosen giving a (mostly) near optimal solution. We used self-organizing maps for our approach.

The SOM (also called Kohonen net) is a feed forward net with neurons grouped in two layers (input and Kohonen layer) and connecting links from input neurons only to the neurons of the Kohonen layer. The neurons of the Kohonen layer are coupled with their neighbor neurons in specific possible topologies.

For the solution of the TSP the neurons are assigned random positions. By the unsupervised learning method of the SOM cities attract the neurons differently. The neuron nearest to a city is chosen as winner and is moved closer to the city. The neighboring neurons are moved too. The goal is that finally every city is mapped to a neuron, resulting in the shortest path by the original sequence of the neurons.

3.2 Characteristics of Self-Organizing Maps

Self-Organizing Maps (SOM) [5, p. 85-144] are neural networks with the following properties [3, p. 98-100]:

- Layers: SOMs consist of just one layer of active neurons. In case you call the input as separate layer, it is labeled as single or dual layer neural network. There is no hidden layer.

- Activation function: The activation function is a monotonic decreasing function f with the following properties:

$$\begin{aligned} \mathbb{R}_0^+ &\rightarrow [0,1], \text{whereas} & (1)\\ f(0) &= 1, & (2)\\ \lim_{x\to\infty} f(x) &= 0 & (3) \end{aligned}$$

  Equation (1) means that the set of positive real numbers (the distances) are projected onto the interval 0 to 1 where (2) is the optimum, because the difference is 0, and (3) denotes that an infinite distance results in the value 0.

- Output function: The output function is normally the identity. Sometimes the winner-takes-all-principle is used: The winning vector and sometimes also the second nearest vector are set to the value 1, all other vectors are set to 0.

- Propagation function: The output of each neuron is a distance between the input vector and the weight vector. Each pair of neurons is assigned a non-negative real number (the distance).

$$U \times U \rightarrow \mathbb{R}_0^+ \quad (4)$$

  U is the set of neurons.

- Learn rule: Learning is done by learning-by-competition. It is searched for the neuron, which is most similar to the input vector.

Self-organizing maps define a neighborhood relation between the neurons. SOM are conserving the topology of the input space. The learn method of SOM belongs to the class of unsupervised learning. There is only one layer of active neurons. These neurons are called Kohonen-neurons.

3.3 Grid Structure

The grid structure (map) of this neural network is commonly 1-, 2- or 3-dimensional. Each dimension offers different possibilities of neighborhood functions. 2-dimensional grids are mostly quadratic or hexagonal. Each point in the grid represents a neuron. The weight vectors of the Kohonen-neurons are distributed in a way that they cover the input space as efficient as possible. Within the neighborhood of the winning neuron all neurons are updated according to the neighborhood function during the training phase. The size of the neighborhood is given by the neighborhood radius. The minimum neighborhood radius consists only of one neuron. The maximum neighborhood is equal to the whole grid of neurons. The vector of the winning neuron is moved the furthest. The farer away the neighborhood vectors are, the less they are moved.

The n-dimensional input vector is compared with all weight vectors. The minimum distance to all input vectors is calculated. Afterwards all weight vectors within the neighborhood are updated.

Zell [11, p. 187] recommends 10^4 to 10^6 iterations to receive a good representation of the input space.

4 Parallelization of the TSP-SOM Algorithm

Basis for the TSP-SOM parallelization builds the SDP (Structural Data Parallel Simulation)[7] approach for the simple parallelization of neural network.

According to the SDP approach a (or number of) data structure(s) has to be identified to distribute it among the available processing nodes by the distribution schema description of a high performance language, as HPF (High Performance Fortran).

The matrix of weights for the neurons is stored in a 2-dimensional array. The first dimension defines the number of the neuron, the second dimension contains the index (1 or 2, which means x- or y-coordinate).

4.1 Unsophisticated Approach

The unsophisticated approach uses a normal Fortran 90 program, and tries to parallelize it right away. It takes the sequential program as base for parallelization. It just tells the system, how the data has to be distributed. It does not use any of the special Fortran 90 capabilities.

Based on the SDP method for parallelizing, the computation of the TSP-SOM program consists of three parts:

1. Get coordinates of cities by random.
2. Get starting weights of Kohonen network by constructing a circle.
3. Train the network.

The training phase, which is the main loop of the program, consists of the following steps:

1. Calculate a vector of distances.
2. Find the winner-neuron.
3. Update matrix of weights.

We applied the unsophisticated approach to get quickly a practicable result. This is straight forward implementing the sequential algorithm with Fortran and distributing the data structure by HPF directives. Out of this the compiler generates the parallel code.

However, simple, but necessary rules of thumbs to write a fast Fortran program are:

- Loops have to be executed in column major order.
- The Fortran 90 capabilities have to be used wherever possible (matrix commands, powerful standard functions).

4.2 Sophisticated Approach

In the sophisticated approach we used the results of the unsophisticated approach and optimized it in several problem areas. This affects changes in the data structure and code for parallel execution. It is important to consider for the processing of the data its alignment in the memory.

However to achieve maximum performance this method is too limited. Therefore we identified the following five issues for specific attention and analysis:

- *Local cache misses*, regarding the array computation along the internal memory alignment.
- *Workload balancing*, considering load balancing between the processors and the alignment of the corresponding elements of different arrays.
- *Latency of communication*, minimizing the occurrence of messages (handshake occurrence).
- *Communication throughput*, exchanging bigger messages to optimize buffering and concerning communication cache size.
- *Locality of data*, considering the distribution scheme of the array data onto the compute nodes respective to the computational work-flow. This results in no computation flow interruptions by communication.

The computational steps are still the same as in the unsophisticated approach, but the implementation is different. So the new, better, program still consists of the same parts of the unsophisticated approach:

1. Calculate a vector of distances.
   The first part can be done in one single Fortran 90 statement. This makes the program shorter and gives Fortran the best possibility to generate a fast parallel code.

2. Find the winner-neuron.
   The code for finding a winner neuron in a distance vector is simply finding the smallest number. For this job two very powerful Fortran functions can be used: $minloc$ (for finding the location) and $minval$ (for finding the value). Fortran is able to parallelize them without request.

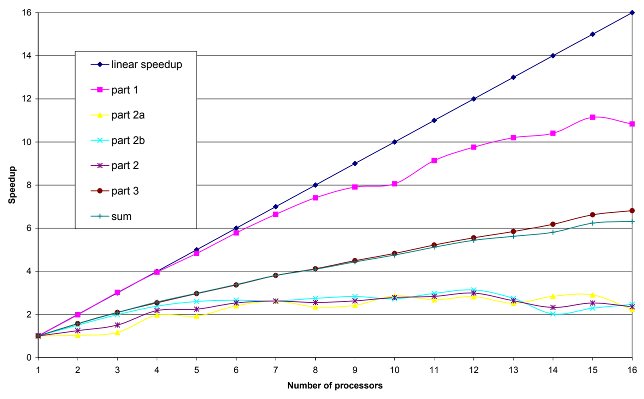

Figure 1. Speedup results

3. Update matrix of weights.
   Now after calculation of the distance vector and detection of the center of excitation, the matrix of weights have to be updated. This could be done by using matrix operations, but it is not very reasonable – because not all weights are updated. Only weights with significant changes are updated.

5 Analysis

The basis for our work was the schroedinger [8] cluster with a Beowulf type clusters architecture [10]. It is the cluster for numerical intensive applications of the University of Vienna. This cluster consists of 192 nodes, each having one Pentium 4, 2.53 GHz and 1 GB DDR RAM, connected by a Gigabit Ethernet.

The TSP algorithm was coded using the PGI pghpf Compiler with MPICH as underlying library.

This presented parallelization approach allows to solve TSP on a SOM basis in the size of hundreds of thousands cities.

The results of the test runs, specifically for a TSP-SOM of size 100000 cities and 500000 neurons, are given in Figure 1. A speedup behavior half of the perfect linear speedup can be easily seen.

In the figure the execution cost of the different program parts are shown separately together with the sum of the overall execution time. For comparison reason also the perfect linear speedup is shown.

The program consists, according to section 4.2, of the following parts:

Startup Phase: Before execution the main loop, construct a polygon of neurons, which approximates a circle. It is also possible to start with random values or even with $(0, 0)$ for each neuron. But a circle (polygon) is known to converge faster towards a good path.

Part 1: Calculate the Euclidean distances between input vector and all neurons. This vector of distances is needed to allow the usage of $minloc$, $minval$. These 2 standard functions are used to find a minimum in a vector or a matrix (and not a distance, so this step is needed). The calculation is done in one Fortran 90 statement, which uses a vector.

Part 2a: Search for minimum value. HPF parallelizes $minloc$ and $minval$ automatically. The quality of parallel execution varies with the size of the vector.

Part 2b: Search for first location of the minimum value.

Part 3: Training of the self-organizing map. The training changes the weights of the neurons from the polygonal order, step by step, to a representation of a shortest path for the TSP. Specifically here are some important things to note:

- Not the whole vector is updated – only neurons, which have a significant change in value. The neighborhood radius shrinks after each iteration. This is necessary to receive stability along time.
- The percentage of updated elements was measured. This is done to avoid a huge number of idle iterations. In addition to this we check, when the last update of the matrix was done.

Part 4: Path construction by the following steps:

1. For all cities find the nearest neuron.
2. Start traversing the resulting path from city 1, this is by definition the starting city; all neurons, which are not nearest to any city, are dropped.
3. Traverse through the chain of neurons.
4. We now have the path, beginning at city 1, containing all other cities and returning to city 1 in the end.

In order to justify our approach algorithmically we compared our solution to other TSP programs using the well known TSPLIB [6]. Our test instances were in EUC_2D coordinates. The best result in finding a shortest path was achieved with the test instance $berlin52$. As our SOM-TSP algorithm is a heuristic approach, the resulting path is obviously not always the same and not always the best. However the documented best path length of 7542 (integer distances) for a symmetric traveling salesman problem was detected with our implementation after approximately 50-500 iterations.

6 Conclusion

We present in the paper a newly developed sophisticated approach for the solution of the well-known Traveling-Salesman-Problem heuristically by self-organizing maps, applying the SDP approach. The results show a very nice speedup behavior of our approach and makes the solution of very large TSPs viable, up to hundreds of thousands cities.

Acknowledgment

The work described in this paper was supported by the Special Research Program SFB F011 AURORA of the Austrian Science Fund.

References

[1] D. Applegate, R. Bixby, V. Chvátal, and W. Cook. Finding cuts in the tsp (a preliminary report). Technical Report DIMACS 95-05, Rutgers University, Piscataway NJ, 1995.

[2] D. Applegate, R. Bixby, V. Chvátal, and W. Cook. On the solution of traveling salesman problems. *Doc.Math.J.DMV Extra Volume ICM III*, pages 645–656, 1998.

[3] C. Borgelt, F. Klawonn, and R. Kruse. *Neuro-Fuzzy-Systeme*. Vieweg Verlagsgesellschaft, 2002.

[4] T. Christof and G. Reinelt. *Parallel Programming and Applications*, chapter Parallel cutting plane generation for the TSP, pages 163–169. IOS Press, 1995.

[5] T. Kohonen. *Self-Organizing Maps*. Springer Series in Information Sciences. Springer, third edition, 2001.

[6] G. Reinelt. TSPLIB — A Traveling Salesman Problem Library. *ORSA Journal on Computing*, 3(4):376–384, 1991.

[7] E. Schikuta, T. Fuerle, and H. Wanek. Structural Data Parallel Simulation of Neural Networks. *System Research and Info. Systems*, 9:149–172, 2000.

[8] Schroedinger, cluster at university of vienna.

[9] N. B. Serbedzija. Simulating Artificial Neural Networks on Parallel Architectures. *IEEE Computer*, 29(3):56–63, 1996.

[10] T. Sterling, D. Becker, D. Savarese, J. Dorband, U. Ranawake, and C. Packer;. BEOWULF: A Parallel Workstation for Scientific Computation. In *International Conference on Parallel Processing*, 1995.

[11] A. Zell. *Simulation neuronaler Netze*. Oldenbourg, 1994.