Neighbourhood Preserving Load Balancing : A Self-Organizing Approach

Attila Gürsoy and Murat Atun

Computer Engineering Department,
Bilkent University,
Ankara Turkey
{agursoy, atun}@cs.bilkent.edu.tr

Abstract. We describe a static load balancing algorithm based on Kohonen Self-Organizing Maps (SOM) for a class of parallel computations where the communication pattern exhibits spatial locality and we present initial results. The topology preserving mapping achieved by SOM reduces the communication load across processors, however, it does not take load balancing into consideration. We introduce a load balancing mechanism into the SOM algorithm. We also present a preliminary multilevel implementation which resulted in significant execution time improvements. The results are promising to further improve SOM based load balancing for geometric graphs.

1 Introduction

A parallel program runs efficiently when its tasks are assigned to processors in such a way that load of every processor is more or less equal, and at the same time, amount of communication between processors is minimized. In this paper, we discuss a static load balancing heuristic based on Kohonen's self-organizing maps (SOM) [1] for a class of parallel computations where the communication pattern exhibits spatial locality. Many parallel scientific applications including molecular dynamics, fluid dynamics, and others which require solving numerical partial differential equations have this kind of communication pattern. In such applications, the physical problem domain is represented with a collection of nodes of a graph where the interacting nodes are connected with edges. In order to perform these computations on a parallel machine, the tasks (the nodes of the graph) need to be distributed to processor. Balancing the load of processors requires the computational load (total weight of the nodes) to be evenly distributed to processors and at the same time the communication overhead (which corresponds to edges connecting nodes assigned to different processors) to be minimized. In this paper, we are interested in static load balancing problem, that is, the computational load of the tasks can be estimated a priori and the computation graph does not change rapidly during the execution. However, the approach can be extended to dynamic load balancing easily.

The partitioning and mapping of tasks of a parallel program to minimize the execution time is a hard problem. Various approaches and heuristics have been developed to solve this problem. Most approaches are for arbitrary computational graphs such as the heuristic of Kernighan-Lin [2] which is a graph bipartitioning approach with minimal cut costs and the ones based on physical or stochastic optimization such as simulated annealing and neural networks [3]. The computational graphs that we are interested, on the other hand, have spatially localized communication patterns. For example, the computational graph from a molecular dynamics simulation [4] is such that the nodes correspond to particles, and the interactions of a particle is limited to physically close particles. In these cases, it is sometimes desirable to partition the computation graph spatially not only for load balancing purposes but for also other algorithmic or programming purposes. This spatial relation can also be exploited to achieve an efficient partitioning of the graph. The communication overhead can be reduced if the physically nearby and heavily communicating tasks are mapped to the same processor or to the same group of processors. The popular methods are based on decomposing the computation space recursively such as the recursive coordinate bisection method. However this simple scheme fails under certain cases. More advanced schemes include the nearest neighbour mapping heuristic [5] and partitioning by space filling curves [6] which try to exploit the locality of communication of the computation graph. In this work, we will present an algorithm based on Kohonen's SOM to partition such graphs. The idea of SOM algorithm is originated from the organizational structure of human brain and the learning mechanisms of biological neurons. After a training phase, the neurons become organized in such a way that they reflect the topological properties of the input set used in training. This important property of SOM — topology preserving mapping — makes it an ideal candidate for partitioning geometric graphs. We propose an algorithm based on SOM to achieve load balancing. Applying self-organization to partitioning and mapping of parallel programs has been discussed by a few researchers [7], [8], however, our modeling and incorporation of load balancing into SOM is quite different from those work and the experiments showed that our algorithm achieves load balancing more effectively.

The rest of the paper is organized as follows: In Section 2, we give a brief description of the Kohonen maps. Then, we describe a load balancing algorithm based on SOM and present its performance. In Section 4, we present a preliminary multilevel approach to further improve the execution time of the algorithm and discuss future work in the last section.

2 Self Organizing Maps (SOM)

The Kohonen's self-organizing map is a special type of competitive learning neural network. The basic architecture of Kohonen's map is n neurons that are generally connected in a d-dimensional space, for example a grid, where each neuron is connected with its neighbours. The map has two layers: an input layer and an output layer which consists of neural units. Each neuron in the output layer is connected to every input unit. A weight vector w_i is associated with each neuron i. An input vector, v, which is chosen according to a probability distribution is forwarded to the neuron layer during the competitive phase of the SOM. Every neuron calculates the difference of its weight vector with the input vector v (using a prespecified distance measure, for example, Euclidean distance if the weight and input vectors represent points in space). The neuron with the smallest difference wins the competition and is selected to be the excitation center c:

$$||w_c - v|| = \min_{k=1}^{n} ||w_k - v||$$

After the excitation center is determined, the weight vectors of the winner neuron and its topological neighbours are updated so as to align them towards the input vector. This step corresponds to the cooperative learning phase of the SOM. The learning function is generally formulated as:

$$w_i \longleftarrow w_i + \epsilon * e^{\frac{-d(c,i)}{2\theta^2}} * ||w_i - v||$$

The Kohonen algorithm has two important execution parameters. These are ϵ and θ. ϵ is the learning rate. Generally, it is a small value varying between 1 and 0. It may be any decreasing function with increasing time step or a constant value. θ is the other variable which highly controls the convergence of the Kohonen algorithm. It determines the set of neurons to be updated at each step. Only the neurons within the neighbourhood defined by θ is updated by an amount depending on the distance $d(c,i)$. θ is generally an exponential decreasing function with respect to increasing time step.

3 Load Balancing with SOM

In this section, we will present a SOM based load balance algorithm. For the sake of simplicity, the discussion is limited to two dimensional graphs. We assume that the nodes of the graph might have different computational loads but the communication load per edge is uniform. In addition, we assume that the cost of communicating between any pair of processors is similar (this is a realistic assumption since most recent parallel machines has wormhole routing). However, it is easy to extend to model to cover more complicated cases. As far as the partitioning of a geometric graph is considered, the most important feature of the SOM is that it achieves a topology-representing mapping. Let the unit square $S = [0,1]^2$ be the input space of the self-organizing map. We divide S into p regions called processor regions where $p = p_x \times p_y$ is the number of processors. Every processor P_{ij} has a region, S_{ij}, of coordinates which is a subset of S bounded by $i \times \mathrm{width}_x, j \times \mathrm{width}_y$ and $(i+1) \times \mathrm{width}_x, (j+1) \times \mathrm{width}_y$ where $\mathrm{width}_x = 1/p_x$ and $\mathrm{width}_y = 1/p_y$. Let each node (or task) of the computation graph (that we want to partition and map to processors) correspond to a neuron in the self-organizing map. That is, the computation graph corresponds to the neural layer. A neuron is connected to other neurons if they are connected in the computation graph. We define the weight vectors of the neurons to be the

Algorithm 1 Load Balancing using SOM

for all neurons i **do**
  initialize weight vectors $w_i = (x, y) \in S$ randomly
end for
for all processors i **do**
  calculate load of each processor
end for
set initial and final values of diameter θ_i and θ_f
set initial and final values of learning constant ϵ_i and ϵ_f
for $t = 0$ to t_{max} **do**
  let S_p be the region of the least loaded processor p
  select a random input vector $v = (x, y) \in S_p$
  determine the excitation center c such that for all neurons n
  $||w_c - v|| = \min ||w_n - v||$
  for $d = 0$ to θ **do**
    for all neurons k with distance d from center c **do**
      update weight vectors $w_k \leftarrow w_k + \epsilon e^{\frac{-d}{2\theta^2}} ||w_k - v||$
    end for
  end for
  update diameter $\theta \leftarrow \theta_i (\frac{\theta_f}{\theta_i})^{\frac{t}{t_{max}}}$
  update learning constant $\epsilon \leftarrow \epsilon_i (\frac{\epsilon_f}{\epsilon_i})^{\frac{t}{t_{max}}}$
  update load of each processor
end for

positions on the unit square S. That is, each weight vector, $w = (x, y)$, is a point in S. Now, we define also mapping of a task to a processor as follows: A task t is mapped to a processor P_{ij} if $w_t \in S_{ij}$.

The load balancing algorithm, Algorithm1, starts with initializing various parameters. First, all tasks are distributed to processors randomly (that is, weight vectors are initialized to random points in S). During the learning phase of the algorithm, we repetitively chose an input vector from S and present it to the neural layer. If we choose the input vector with uniform probability over S, then, the neural units will try to fill the area S accordingly. If the computation of each task (node) and communication volume (edges) are uniform (equal), then the load balance of this mapping will be near-optimal. However, most computational graphs might have non-uniform computational loads at each node. In order to enforce a load balanced mapping of tasks, we have to select input vectors differently. This can be achieved by selecting inputs from the regions closer to the least loaded processor. This strategy will probably force SOM to shift the tasks towards to least loaded processor and the topology preserving feature of SOM will try to keep communication minimum. A detailed study of how to choose input vector and various alternatives is presented in [9]. It has been experimentally found out that choosing the input vector always in the region of the least loaded processor leads to better results. As we mentioned before, ϵ is the learning rate which is generally an exponential function decreasing with increasing time step. At the initial steps of the algorithm, ϵ is closer to 1, which means learning rate is high. Towards to the end, as ϵ becomes closer to 0, the adjustments do minor changes on weight vectors and so the mapping becomes more stable. In our algorithm we used 0.8 and 0.2 for initial and final values of ϵ. To determine the set of neurons to be updated, we defined θ to be the length of the shortest path (number of edges) to reach from the excitation center. Initially, it has a value of $\theta_i = \sqrt{n}$ and it exponentially decreases to 1. These values are the most common choices used in SOMs. The lines 13-17 in Algorithm1 correspond to this update process. Figure 1 illustrates a partitioning of a graph with 4253 nodes into eight processors using the proposed algorithm.

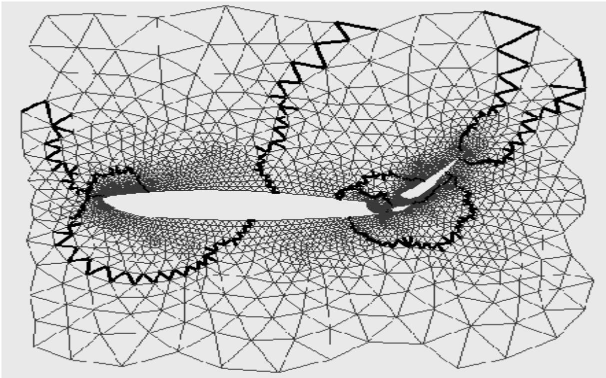

Fig. 1. Partitioning a FEM graph (airfoil): 4253 nodes to 8 processor

3.1 Results

We have tested our algorithm on some known FEM/Grid graphs available from AG-Monien (Parallel Computing Group) Web pages [10]. We compared the performance of our algorithm with the results of algorithm given in Heiss-Dormanns [7]. They reported that their SOM based load balancing algorithm was comparable with other approaches. Particularly, the execution time of their algorithm was on the average larger than mean field annealing based solutions but less than simulated annealing ones for random graphs.

We conducted runs on a set of FEM/Grid graphs and gathered execution times for our algorithms and Heiss-Dormanns' algorithm on a Sun Ultra2 workstation with 167MHZ processor. Table 1 shows the load balance achieved and total execution times. Our approach performed better on all cases. However, as in other stochastic optimization problems, the selection of various parameters such as learning rate plays and important role in the performance. For the runs for the algorithm by Heiss-Dormanns, we used their suggestions for setting various parameters in the algorithm given in their reports.

Table 1. Load balance and execution time results for FEM/Grid graphs

		Communication Cost(x1000)		Load Imbalance (%)		Execution Time (secs)	
Graph	Processor Mesh	Heiss-Dorm. Alg.	Our Alg.	Heiss-Dorm. Alg.	Our Alg.	Heiss-Dorm. Alg.	Our Alg.
3elt	4X4	2.16	1.11	22.71	0.45	604.19	135.69
3elt	4X8	1.25	1.66	15.03	1.47	372.69	133.08
Airfoil	4X4	1.76	1.04	24.15	0.57	498.66	118.67
Airfoil	4X8	0.90	1.56	10.04	0.82	300.22	115.98
Bcspwr10	4X4	0.48	0.76	21.96	0.33	555.96	162.65
Bcspwr10	4X8	0.74	1.09	51.14	2.44	862.85	159.21
Crack	4X4	2.23	1.51	5.47	0.21	1986.90	290.65
Crack	4X8	3.67	2.37	26.88	0.42	810.02	281.52
Jagmesh	4X4	0.50	0.39	12.25	0.85	16.87	18.26
Jagmesh	4X8	0.92	0.62	32.19	4.84	29.27	18.40
NASA4704	4X4	136.02	10.27	60.77	0.91	843.27	214.34
NASA4704	4X8	255.31	14.90	88.21	3.63	1182.2	211.09

4 Improvement with Multilevel Approach

In order to improve the execution time performance of the load balancing algorithm, we modified it to do the partitioning in a multilevel way. Since physically nearby nodes get assigned to the same processor (most likely), it will be beneficial if we could cluster a group of nodes into a super node and run the load balancing on this coarser graph, then unfold the super nodes and refine the partitioning. This is very similar to multilevel graph partitioning algorithms which have been used very successfully to improve the execution time [11]. In multilevel graph partitioning, the initial graph is coarsened, to get smaller graphs where a node in the coarsened graph represents a set of nodes in the original graph. When the graph is coarsened to a reasonable size, an expensive but powerful partitioner performs an initial partitioning of the coarsened graph. Then, the graph is uncoarsened, that is unfolding the collapsed nodes, and mapping of unfolded nodes is handled (refinement phase). In our implementation, we have used the heavy-edge-matching (HEM) scheme for the coarsening phase as described in [11]. HEM scheme selects nodes in random. If a node has not matched yet, then the node is matched with a maximum weight neighbour node. This algorithm is applied iteratively, each time obtaining a coarser graph. For initial partitioning of the coarsest graph and refinement phases, we have used our SOM algorithm without any change. The performance results of a preliminary multilevel implementation of our algorithm for the FEM/Grid graphs is presented in Table 2. The results show that multilevel implementation reduces the execution time significantly.

Table 2. Execution results of SOM and MSOM: n-initials is the number of nodes in the initial graph, n-final is the number of nodes in the coarsest graph, and Levels is the number of coarsening levels

Graph	Processors	n-initial	Levels	n-final	Load Imbalance		Execution Time	
					SOM	MSOM	SOM	MSOM
Whitaker	4x4	9800	9	89	0.08	0.73	216.18	42.29
Whitaker	4x8	9800	9	89	0.57	1.55	212.16	60.57
Jagmesh	4x4	936	5	65	0.85	1.42	18.26	3.86
Jagmesh	4x8	936	5	65	4.84	5.98	18.40	5.70
3elt	4x4	4720	8	71	0.45	1.69	135.69	31.99
3elt	4x8	4720	8	71	1.47	1.69	133.08	33.95
Airfol	4x4	4253	8	63	0.57	1.07	118.67	28.58
Airfol	4x8	4253	8	63	0.82	1.82	115.98	35.16
NASA704	4x4	4704	7	97	0.91	4.98	214.34	61.31
NASA704	4x8	4704	7	97	3.63	8.16	211.09	101.28
Big	4x4	4704	10	96	0.10	2.25	521.28	78.50
Big	4x8	4704	10	96	4.37	2.66	492.30	142.69

5 Related Work

Heiss-Dormanns used computation graph as input space and processors as output space (the opposite of our algorithm). A load balancing correction, activated once per a predetermined number of steps, changes the receptive field of processor nodes according to their load. Changing the magnitude of a receptive field corresponds to transferring the loads between these receptive fields. The results show that our approach handles load balancing better and has better execution time performance.

In another SOM based work, Meyer [8] identified the computation graph with the output space and processors with input space (called inverse mapping in their paper). Load balancing was handled by defining a new distance metric to be used in learning function. According to this new metric the shortest distance between any two vertices in the output space is formed by the vertices of least loaded ones of all other paths. It is reported that SOM based algorithm performs better than simulated annealing approaches.

6 Conclusion

We describe a static load balancing algorithm based on Self-Organizing Maps (SOM) for a class of parallel computations where the communication pattern exhibits spatial locality. It is sometimes desirable to partition the computation graph spatially not only for load balancing purposes but for also other algorithmic or programming purposes. This spatial relation can also be exploited to achieve an efficient partitioning of the graph. The communication overhead can be reduced if the physically nearby and heavily communicating tasks are mapped to the same processor or to the same group of processors. The important property of SOM — topology preserving mapping — makes it an interesting approach for such partitioning. We represented tasks (nodes of the computation graph) as neurons and processors as the input space. We enforced load balancing by choosing input vectors from the region of least loaded processor. Also, a preliminary multilevel implementation is discussed which has improved the execution time significantly. The results are very promising (it produced better results than the other self-organized approaches). As future work, we plan to work on improving both the performance of the current implementation and also develop new multilevel coarsening and refinement approaches for SOM based partitioning.

Acknowledgement
We thank H.Heiss and M. Dormanns for providing us the source code of their implementation.

References

1. Kohonen, T.: The Self-Organizing Map *Proc. of the IEEE, Vol.78, No.9, September*, 1990, pp.1464-1480
2. Kernighan, B.W., Lin., S.: An Efficient Heuristic for Partitioning Graphs, Bell Syst. J., 49, 1970, pp. 291-307
3. Bultan T., Aykanat C.: A New Mapping Heuristic Based on Mean Field Annealing, J. Parallel Distrib. Comput., 1995, vol 16, pp. 452-469
4. Nelson, M., et al.: NAMD: A Parallel Object-Oriented Molecular Dynamics Program, Intn. Journal of Supercomputing Applications and High Performance Computing, Volume 10, No.4, 1996., pp.251-268
5. Sadayappan, P., Ercal., F.: Nearest-Neighbour Mapping of Finite Element Graphs onto Processor Meshes, IEEE Trans. on Computers, Vol. C-36, No 12, 1987, pp. 1408-1424
6. Pilkington, J.R, Baden, S.B.: Dynamic Partitioning of Non-Uniform Structured Workloads with Spacefilling Curves, IEEE Trans. on Parallel and Distributed Sys. 1997, Volume 7, pp. 288-300
7. Heiss, H., Dormanns, M.: Task Assignment by Self-Organizing Maps *Interner Bericht Nr.17/93, Mai 1993* Universität Karlsruhe, Fakultät für Informatik
8. Quittek, J.W., Optimizing Parallel Program Execution by Self-Organizing Maps, Journal of Artificial Neural Networks, Vol.2, No.4, 1995, pp.365-380
9. Atun, M.: A New Load Balancing Heuristic Using Self-Organizing Maps, M.Sc Thesis, Computer Eng. Dept., Bilkent University, Ankara, Turkey, 1999
10. University of Paderborn, AG-Monien Home Page (Parallel Computing Group), http://www.uni-paderborn.de/fachbereich/AG/monien.
11. Karypis., G, Kumar. V.: A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs, TR 95-035, Department of Computer Science, University of Minesota, 1995.