A NEW LOAD-BALANCING ALGORITHM USING SELF-ORGANIZING MAPS

Murat Atun and Attila Gürsoy
Computer Engineering & Information Sciences, Bilkent University, Ankara, Turkey
{atun,agursoy}@cs.bilkent.edu.tr

Abstract In order to have an optimal performance during an execution of a parallel program the load of tasks should be distributed as evenly as possible while highly communicating tasks are placed closely. We describe a new implementation of Kohonen Self-Organising Map for static load balancing problem and examine variations of the algorithm. We also provide results of performance studies for different types of task graphs.
Keywords: neural networks, self-organising maps, Kohonen, partitioning, load balancing

1. Introduction

The increase of multi-processor systems with accordance to demand to solve large problems rapidly has brought many advantages with also many problems. One common question became "how to assign the parallel program tasks to processors of multi-processor system?'' A parallel program runs most efficiently when 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 minimised. Such an assignment is defined to be optimal if the total execution time of a parallel program on the multi-processor system is minimal. This is a difficult problem, and many heuristics have been developed to solve this problem.

In this paper, we discuss a static load balancing heuristic based on Kohonen's self-organising maps (SOM) [6] for a class of parallel programs where the communication between tasks are localised, that is, the communication pattern has spatial clustering. Many real life parallel applications have this kind of communication patterns from molecular dynamics to finite element analysis. Applying Kohonen SOM to this problem is advantageous since SOMs preserve neighbourhood relations. We will show that the neighbourhood preservation of SOM satisfies the communication requirement, however, the algorithm should be forced to distribute the loads balanced. In this work, we have incorporated load balancing into the SOM algorithm.

The rest of the paper is organised as follows: In Section 2, we give a model for processor and task representations. In Section 3, we briefly introduce the Kohonen algorithm. In Section 4, a static load-balancing algorithm using SOMs with several alternative approaches is discussed. Section 5 presents the performance results and other similar approaches using SOMs for mapping are discussed in Section 6.

2. A Model for Representation of Processors and Tasks

In order to find an efficient mapping, processors and tasks must be represented by suitable models that exploit the features of the architecture of the multi-computer and properties of parallel program tasks. Processor Connection Graph (PCG) and Task Interaction Graph (TIG) are models commonly used for this purpose. A PCG, is a weighted undirected graph $G(V,E)$ where each $v \in V$ represents a processor and each $(u,v) \in E$ represents the physical link between processors u and v. In order to handle different architectures, a weight is assigned to each vertex indicating the speed of that processor and to each edge representing the bandwidth of the communication link.

In our task model, we assume that parallel tasks can be thought of as a sequence of computation and communication phases. The load of a task, then, is the average computation time between successive communication phases. A TIG representing a collection of such tasks is a weighted undirected graph *G(V,E)* such that $\forall v \in V$, represents a task and each $(u,v) \in E$ represents the data exchange between tasks *u* and *v*. Similarly, a weight is assigned to each vertex indicating the load of that task and a weight to each edge *(u,v)* representing the communication load between tasks *u* and *v*. Given a TIG $G^T(V^T,E^T)$ and PCG $G^P(V^P,E^P)$, we want to find a mapping $M\colon V^T \rightarrow V^P$ such that

$$\text{Communication Cost} = \sum_{(i,j) \in E^T} \alpha(i,j) * d(\rho(i), \rho(j)) \qquad (1)$$

and load imbalance are minimised. In the above formula, $\alpha(x,y)$ is the weight of edge between vertices *x* and *y* in TIG, $\rho(x)$ is the processor that task *x* is assigned in PCG and *d(x,y)* is the Euclidean distance between vertices *x* and *y* in PCG.

Applying Kohonen SOM algorithm to this problem has many advantages. Since the algorithm preserves neighbourhood relations, it naturally solves the communication cost problem. That is since Kohonen algorithm tries to place neighbouring neurons closer this is exactly what a mapping should satisfy (placing neighbouring tasks closer).

3. Self Organising Maps (SOM)

The Kohonen Network is a special type of neural network with unsupervised training. Its main motivation is training a set of neurons with a set of inputs for enough number of times. The stopping criteria is generally a predetermined number of steps or the result of an error function whose value is expected to become less than a predetermined threshold value. During training the neurons organise themselves in such a way that the final interaction pattern of neurons reflect the properties of input vectors. The basic architecture of Kohonen's self-organising map is *n* neurons, which are generally connected in a d-dimensional space. The map has two layers, where the neurons act as output layer. Each neuron in the output layer is connected to every input unit. A weight vector W_i is associated with each neuron *i*. These weight vectors store the learned exemplar patterns. An input vector *I*, which is chosen according to a probability distribution is forwarded to the neuron layer. Every neuron calculates the difference of its weight vector with this input. The one with the smallest difference wins the competition and is selected to be the excitation centre *c*:

$$||W_c - I|| = Min \, ||W_k - I|| \; (\text{for } k=1 \text{ to } n) \qquad (2)$$

After the excitation center is determined, the weight vectors of neurons are updated according to a learning function by which the neurons organise themselves. The learning function of Kohonen algorithm is generally formulated as:

$$W_i^{t+1} = W_i^t + \varepsilon * h(c,i) * (I^t - W_i^t) \qquad (3)$$

In formula 3, W_i^t and W_i^{t+1} are the weight vectors of neuron *i* at time *t* and *t+1* respectively. I^t is the selected input vector at time *t* and *h(c,p)* is the neighbourhood function.

The neighbourhood function determines interaction between neurons and weight vector update ratios. It is mainly based on distance of neurons from excitation centre. It has a maximum value 1, for the excitation center (distance=0) and decreases with increasing distance from the excitation center. The most important characteristic of this neighbourhood function is that it is a Gaussian function and is generally formulated as:

$$h(c,i) = e^{(-d(c,i)/(2*\theta*\theta))} \quad (4)$$

In the upper formula, $d(c,i)$ is the Euclidean distance in the output space between the neuron i and excitation center c.

Kohonen algorithm has two important execution parameters which are ε and θ. ε is the learning rate at time t and it is generally a small value varying between 1 and 0. θ is the other function, which determines the neurons to be updated at each step. Both of them are generally exponential decreasing functions with increasing time step.

Kohonen algorithm simply maps the output space to input space. That is with chosen input and output spaces, the algorithm iteratively organises the output space such that the neurons of output space reflect the properties of the input space at the end of the execution. So, it finds a topological mapping that preserves neighbourhood [2][3].

4. Solving the Mapping Problem with Kohonen Neural Network

In this section, we will show how we can use self-organising maps to solve the load-balancing problem. Let the unit square $S=[0,1]^2$ be the input space of the self-organising map. We divide S into p regions called processor regions as shown in Figure 1, where $p = p_x*p_y$ is the number of processors. Every processor P_{ij} has a region of coordinates which is a subset of S bounded by $i*width_x$, $j*width_y$ and $(i+1)*width_x,(j+1)*width_y$ where $width_x=1/p_x$ and $width_y=1/p_y$. Each task corresponds to a neuron in the self-organising map. The weight vectors of these neurons are selected to be positions on the unit square S. That is, each weight vector, $W=(x,y)\in S$, is a point in S. A task, i, is mapped to a processor P_{ij} if the weight of the task i, W_i, is in the region of P_{ij}. Figure 1 shows a 4-processor space where tasks are distributed over the S.

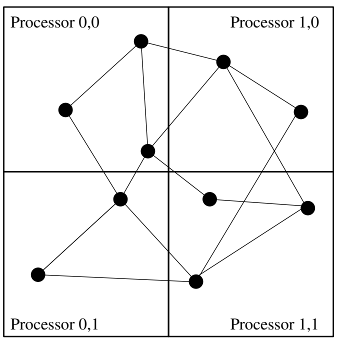

Figure 1: Input and Output Spaces

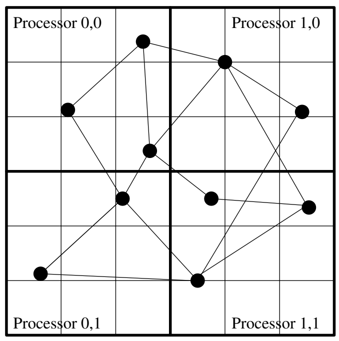

Figure2: Processor Regions and Subregions

The algorithm distributes the tasks to processors randomly at the beginning. Then, at each time step, weights of the tasks evolve to cover the range of input values (processor space) as follows:

1. Initialise all weight vectors, W_i randomly (i.e., distribute tasks randomly over S).
2. For t = 1 to t_{max} :
   - Select an input, $I^t=(x,y)\in S$. SOM algorithm would choose any input vector with equal probability. This is where we force the algorithm to do load balancing. We choose I^t such that it is close to a lightly loaded processor area.

This enforces the SOM algorithm to bring more tasks to the lightly loaded processor. How exactly choose I^t has an important impact on the performance of the algorithm. Various alternatives are discussed in Section 4.2.

- Determine excitation center c such that $||W_c - I^t|| = Min\,||W_k - I^t||$ *(for k=1 to n)* The input vector represents a coordinate information and it lies within the region of a processor area as mentioned in the above part. In order to determine the closest task to the input, all the vertices of TIG should be scanned which is a very high-cost operation. In order to eliminate this drawback, we applied a range searching method, which is also known as “grid method”. We divided every processor region into grid connected subregions (Figure 2). For a given input vector, in order to find the excitation center, beginning from the subregion that owns the input vector, all surrounding subregions are searched iteratively by which the excitation center is found much more quickly.
- Update weights with $W_i^{t+1} = W_i^t + \varepsilon * h(c,i) * (I^t - W_i^t)$ where $||I^t - W_i^t||$ is the Euclidian distance between I^t and W_i^t.

  At the initial steps of the algorithm ε is closer to 1, which means learning rate is high. At the latest steps, 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 the below formula for ε with initial and final values as 0.8 and 0.2 respectively.

  $$\varepsilon = epsInitial * (epsFinal/epsInitial)^{(currentstep/totalsteps)} \qquad (5)$$

  For neighbourhood function we used a Gaussian model. Many functions can be used which produces nearly the same result. But in [5] it is proved that, within examined neighbourhood functions, Gaussian gives the best result from the point of metastable states.

  $$\theta = thetaInitial * (thetaFinal/thetaInitial)^{(currentstep/totalsteps)} \qquad (6)$$

In Figure 3 we present a sample execution of our algorithm for a simple grid of 100 vertices. Initially all the tasks are randomly distributed over processor regions. At step 3200 the graph began to expand to fit the whole input space of processors and in the final figure (10000) the vertices all most fit the processor area in a balanced way.

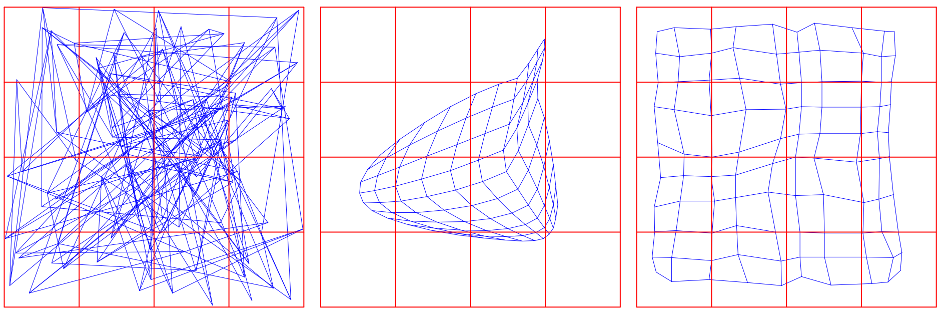

Figure 3: Execution states at step 0, 3200 and 10000

4.1. Input Selection and Incorporating Load Balance

As we mentioned SOM satisfies one requirement of mapping — preserving neighbourhood. But there is the need to explicitly force the algorithm to take care of load imbalance. Generally, uniformly distributed input selection models are used in SOM algorithms. This type of selection forces the SOM algorithm to distribute equal number of tasks to processors without taking into account task loads [4]. But such a distribution is not a desired criterion for load balancing. So SOM algorithm should be forced to distribute the loads of the tasks to processors in a balanced way. This can be achieved by selecting inputs from the regions closer to least loaded processor. By this way, such an input will probably force the algorithm to shift the tasks towards to least loaded processor which means assignment of more tasks to that processor, thus minimising the load imbalance. During input selection, least loaded processor information can be directly used and an input within the region of that processor may be chosen or such an information can be used to determine another processor. In either case the possibility for the selection of least loaded processor should be high. A Gaussian distribution function identifying processor selection probabilities can be used as an indirect method, still keeping in mind that least loaded processor has the maximum probability. Such a function usually choose least loaded processor or another region closer to it with high probabilities, which will satisfy our requirements.

4.2. Processor Selection Strategies

We have several alternatives, for choosing a point nearby the least loaded processor. We can choose a point inside the least loaded one, or in a neighbouring processor area. We have used the following alternative selection methods to determine the processor area:

- PRO-DLL: For this case, the input is directly selected from the region of least loaded processor.
- PRO-GASD: For this case, first, the least loaded processor is determined. Then, the number of GASD levels are determined within the processor grid where each GASD level corresponds to a different distance to the least loaded processor as shown in Figure 5a. Afterwards, the processors, which are in the same GASD level, are determined and a probability value is assigned to each level according to Gaussian distribution function, where the least loaded processor has the maximum probability. In the final step, one of the processors in the determined GASD level is selected with uniform probability.
- PRO-GASR: This is similar to PRO-GASD. The only difference is the definition of the GASR level as shown in Figure 5b.

The determination of the processor region that will produce the input does not exactly determines the input. Since we have another degree of freedom “subregions”, their effect should also be analysed.

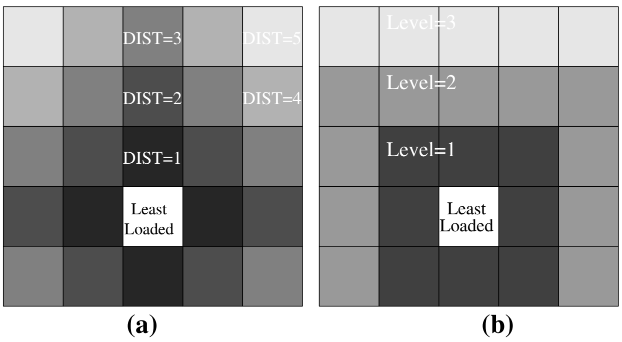

Figure 5: Gaussian Distribution with Distance (a) and Region (b) Information

4.3. Subregion Selection Strategies

We applied the same functions for subregion selection. But since least loaded selection is not as important as in processor selection for subregions, we also test two other alternative methods “Uniform” and “Center” totally making 5 with “Least Loaded”, ”Gaussian with Distance” and “Gaussian with Region”. For these last three we applied the same idea that we used for processor selection. Additionally for ”Center” directly the subregion in the center of processor region is selected and for “Uniform” a uniformly distributed random number is used to determine the subregion. We test each subregion selection strategy for each processor selection method and the results of these tests are given in Section 5.

5. Performance Results

In order to determine the most efficient strategy for load balancing we used test graphs with different properties, which are listed in Table 1. For each graph, we test each 3 processor and each 5 subregion selection strategy with 16 and 32 processors. Each execution was repeated 3 times and average values are calculated. We fixed the number of steps in the range of 5000 and 20000, changing according to the properties of TIGs. During the analysis we used the below formula for load imbalance ratio determination.

$$\textit{Load Imbalance} = ((\textit{Max Load} - \textit{Avg Load})*100)/\textit{Avg Load} \qquad (7)$$

	# Vertices	# Edges	Graph Type
Gr150	22500	44404	Regular Grid
GrAbk	5300	8271	Non-regular
Biplane	21701	42038	4-connected Planar
Jagmesh	936	2664	Non-regular FEM
Whitaker	9800	28989	Regular FEM

Table 1: Properties of Test Graphs

In Table 2, we give percentage load imbalance values related to the executions for 32 processors. For the sake of simplicity, the table includes results for 3 test graphs Gr150, Jagmesh and Whitaker where the behaviour of the program for others is same. As can be seen, the values on PRO-DLL column outperform the ones on the other two columns for each graph and subregion selection strategy.

Graph Type	Subregion Selection Strategy	PRO-DLL	PRO-GASD	PRO-GASR
	Uniform	0.739	29.392	34.697
	Center	2.079	55.050	58.601
Gr150	Least Loaded	4.631	33.999	42.735
	Gaussian with Distance	0.394	46.458	40.638
	Gaussian with Region	1.171	44.743	32.161
	Uniform	7.331	27.023	38.989
	Center	4.376	61.643	45.026
Jagmesh	Least Loaded	4.840	38.761	55.343
	Gaussian with Distance	5.716	30.025	37.625
	Gaussian with Region	4.399	40.287	41.900
	Uniform	1.752	37.452	44.359
	Center	23.521	78.324	82.107
Whitaker	Least Loaded	4.570	46.992	59.354
	Gaussian with Distance	3.223	49.501	65.595
	Gaussian with Region	2.990	50.243	49.951

Table 2: Load imbalance values for 3 test graphs for 32 processors

In Table 3 we give the results for all subregion selection strategies for PRO-DLL method. As can be seen from the table, the method "Center" generally produce over average results for both 16 and 32 processors, which are not acceptable. So we directly eliminate it. The method "Least Loaded" seems to perform well for 16 processors but the increase on the number of processors highly effects the performance of this method making it nearly worst or second worst. The same thing also holds for "Uniform" especially for graph types 3 and 4, and for "Gaussian with Distance" especially for test graphs 2 and 3. For 16 processors the values found by "Gaussian with Region" are generally not acceptable when compared with others. But for 32 processors the values produced are generally second best and all values lie below *8%*. But the more important is that the values found by this method are stable. That is changing test graphs and numbers of processors do not affect the performance of this method heavily, which is best observed for test graph 3.

Graph	# of processors	Uniform	Center	Least Loaded	Gaussian with Distance	Gaussian with Region
	16	0,146	0,524	0,384	0,684	0,148
Gr150	32	0,740	2,080	4,631	0,394	1,172
	16	0,777	6,088	2,208	0,700	3,616
GrAbk	32	5,000	16,162	15,503	13,391	8,275
	16	0,446	20,352	7,261	0,525	1,164
Biplane	32	34,311	18,976	40,583	17,632	2,413
	16	1,947	1,768	1,673	2,473	2,026
Jagmesh	32	7,332	4,376	4,841	5,717	4,399
	16	0,691	1,155	0,614	0,783	1,630
Whitaker	32	1,752	23,522	4,571	3,224	2,991

Table 3: Load Imbalance values handled for PRO-DLL

Choosing the best subregion selection method is not as straightforward as processor selection. But it seems "Gaussian with region" strategy performs best especially for 32 processors. The values are not best all the time, but on the average this method produces stable results. For all graphs and strategies, the maximum load imbalance value produced by this method is *8%*, while all other methods produce at least one value over *15%*. As a result PRO-DLL combined with "Gaussian with region" subregion selection strategy distributes loads more balanced than the other combinations.

6. Related Work

Up to now, many algorithms/heuristics designed for load balancing or graph partitioning. But only a few ones were related to Kohonen's algorithm. To the best of our knowledge the latest implementations are the works done by Meyer and Heiss-Dormanns, In this section we will examine Heiss-Dormanns's and J.Meyer's (or Quittek's) implementations.

Heiss-Dormanns [1] used TIG as input space and PCG as output space. During the execution, randomly a task is chosen and the weight vectors of PCG nodes representing the processors are updated. They proved that SOM algorithm minimises their cost function but they claim that it should be forced to take care of load distribution. For this purpose they implemented an external load balancing routine which is activated once per a determined number of steps after a globally ordered map is handled. This load balancing routine mainly changes the receptive fields of PCG nodes according to their load where changing the magnitude of a receptive field corresponds to transferring the loads between these receptive fields thus making tasks assigned to another processor(s).

Unlike Heiss-Dormanns, Meyer [2][3] identified the TIG with the output space and PCG with input space. She called her mapping inverse mapping with respect to Heiss-Dormanns. She proposed 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. By this way, she forced the Kohonen algorithm not to make closer the heavy tasks. Later she defined a function using this metric which calculates the sum of the weights of edges and the sum of weights of vertices on such path. She associated a parameter with each sum (edge and vertex) which are α and $1 - \alpha$ and addressed her problem as finding the best value of α since increasing values of α makes the communication cost more important while decreasing values will yield a perfect distribution of load without taking communication cost into account.

The algorithm we propose is very different from Heiss-Dormanns's approach and initial results show that our algorithm outperforms the implementations given below. The performance results comparing our algorithm will be presented elsewhere.

7. Conclusion

We presented a new implementation of Kohonen's self-organising map for static load balancing problem. Applying SOM algorithm to this problem is advantageous since it preserves neighbourhood relations. However, the algorithm should be forced to distribute the loads balanced. In this work, we have incorporated load balancing into the SOM algorithm. We represented tasks as neurons and processor area as input space. We analysed different input selection strategies for this algorithm and determined that choosing the input always from least loaded processor area performs better than the other approaches. The algorithm produced better results compared to other self-organising approaches used for load balancing.

Reference:

[1] H. Heiss, M. Dormanns, Task Assignment by Self-Organizing Maps, Interner Bericht Nr.17/93, Mai 1993, Universität Karlsruhe, Fakultät für Informatik

[2] Jürgen Quittek, Optimising Parallel Program Execution by Self-Organising Maps

[3] Jürgen Meyer, Self-Organising Processes, Buchberger, J. Volkert, Parallel Processing: CONPAR 94 – VAPP VI, Lecture Notes in Computer Science 854, Springer-Verlag, Berlin, 1994, pp.842-853

[4] E. Erwin, K. Obermayer, K. Schulten (1992) Self-Organising Maps: Convergence Properties and Energy Functions. Biological Cybernetics

[5] E. Erwin, K. Obermayer, K. Schulten (1992) Self-Organising Maps: Stationary States, Metastability and Convergence Rate. Biological Cybernetics

[6] T. Kohonen, The Self-Organising Map, Proceedings of the IEEE, Vol.78, No.9, September, 1990, pp.1464-1480