Mapping Large-Scale FEM-Graphs to Highly Parallel Computers with Grid-Like Topology by Self-Organization

Marcus Dormanns
Hans-Ulrich Heiss

University of Karlsruhe
Department of Informatics
D-76128 Karlsruhe
Germany
e-mail: {dormanns,heiss}@ira.uka.de

February 9, 1994

Abstract

We consider the problem of mapping large scale FEM graphs for the solution of partial differential equations to highly parallel distributed memory computers. Typically, these programs show a low-dimensional grid-like communication structure.

We argue that conventional domain decomposition methods that are usually employed today are not well suited for future highly parallel computers as they do not take into account the interconnection structure of the parallel computer resulting in a large communication overhead.

Therefore we propose a new mapping heuristic which performs both, partitioning of the solution domain and processor allocation in one integrated step. Our procedure is based on the ability of Kohonen neural networks to exploit topological similarities of an input space and a grid-like structured network to compute a neighborhood preserving mapping between the set of discretization points and the parallel computer.

We report about results of mapping up to 44,000-node FEM graphs to a 4096-processor parallel computer and demonstrate the capability of the proposed scheme for dynamic remapping considering adaptive refinement of the discretization graph.

Keywords : highly parallel computers, Finite Element Method, data partitioning, mapping, Kohonen Neural Networks

Interner Bericht Nr. 5/94
Februar 1994

1 Introduction

While the theoretical computational power of modern parallel computers already exceeds those of classical vector computers considerably, their broader acceptance grows with caution. This is mainly due to a lack of software support, i.e. automatic parallelizing compilers, automatic data partitioning and mapping.

Nowadays, most research laboratories have a parallel computer at their disposal, mainly used to simulate technical and physical systems (i.e. structural mechanics, fluid dynamics, n-body systems, etc.). Most of those parallel computers are MIMD message passing systems but only of limited degree of parallelism (less than 1024). One of the most fundamental feature of parallel computers is scalability, the possibility to configurate parallel computers with as much computational power as is needed just by adding more processors. In the near future also highly parallel systems will be affordable with much more than 1024 processors. This will increase the problems with poor efficiencies due to load unbalance and communication overhead, much bigger as it is already the case with today's affordable small- and medium-size parallel computers with small communication network diameters.

In this paper we consider the partitioning and mapping of a special but typical kind of application, namely solving partial differential equations (PDE) with the finite element method (FEM) on such highly parallel machines.

Today the usual approach for parallelizing such problems is to exploit the knowledge of the shape and dimension of the solution domain to perform some kind of domain partitioning into as many partitions as processors are available in a way that minimizes load unbalance and the amount of data dependencies between different partitions but without taking into account the topology of the parallel computers communication network [7, 10, 11, 12, 18, 19, 21]. In a second, independent step these partitions are mapped onto the processors such that communication is minimized either by employing some kind of processor allocation heuristic [10] or by generating a corresponding partitioning of the parallel computer [7, 11, 12, 18]. Even simpler is just to rely on load balancing that can also lead to good efficiencies for small parallel computers with a high computational load [1, 2]. The communication scheme which is then needed is an *all-to-all broadcast*. This can be done in logarithmic time on a hypercube, not more than is needed for a scalar product, but the amount of data that has to be exchanged is much larger.

However, also if partitioning is performed, because partitioning and mapping are independent of each other, it is possible that the partitioning found is highly inappropriate to establish a good mapping with low communication cost to a highly parallel computer with its specific interconnection topology.

On the other hand, the general approach for the mapping of parallel programs to parallel computers, not only applicable for such geometry based problems, is to model the parallel program as a general graph with the nodes representing parallel tasks and edges between communicating tasks (the so-called *Task Interaction Graph* or short TIG). Analogously the parallel machine is modeled as a graph as well (the so-called *Processor Connection Graph*, or short PCG), with the nodes representing processors and the edges communication links between them. The mapping problem can then be formulated in terms of a graph embedding problem of the TIG into the PCG with optimization goals related to load balancing and communication cost minimization. Several general heuristic techniques for combinatorial optimization are employed to find good embeddings, i.e. Simulated Annealing [3], Mean Field Annealing [4], Genetic Algorithms [5], Evolution Algorithms [15] and graph theoretical

techniques, e.g. [14].

All of these heuristics have the common property that they are computationally very expensive and therefore are only applicable for small and medium size problems and parallel machines. One reason for this is that they are designed for TIGs with arbitrary structures and do not exploit any further knowledge of the structure of the TIG that may be available (like in the case of FEM graphs). Also some of them are not well suited to be executed in parallel or are even inherently sequential (like Simulated Annealing).

In [6, 8] we proposed an embedding heuristic based on Kohonen's Neural Networks and their ability to establish topology conserving mappings between a feature vector space into which the TIG is suitably embedded and a neural network of fixed topology, representing the parallel computer. Although it is a procedure for the general problem instance it exploits coarse similarities between the structure of the TIG and the PCG. In [9] we presented its application to mapping discretized PDE's to small- and medium-size parallel computers using a pre-computation step to reduce the problem size. Using such mappings we solved the resulting systems of linear equations with the well known cg-algorithm (conjugate gradient).

While our previous work dealt with general task interaction graphs, we are now focusing on a particular class of problems that make up the lion's share of current parallel supercomputer applications. Not surprisingly, this specialization leads to much more efficient algorithms. We will show how to incorporate knowledge concerning the special problem structure, namely the fact that the FEM-Graph is of a low-dimensional space (2- or 3-dimensional) and the property that only geometrically nearby nodes need to exchange information, into the procedure based on Kohonen Networks. Using this information results in a procedure that performs partitioning and mapping in one single step and provides very good solutions also for large scale problems and highly parallel machines and can be executed in parallel on the target machine itself.

This paper is organized as follows: We start with a short description of the problems that arise if conventional recursive bisection methods are used to map a FEM graph to a highly parallel computer in Section 2. In Section 3 we formalize the specific problem we address. Section 4 describes the proposed mapping procedure that is based on Kohonen Neural Networks. Section 5 shows the results of the proposed scheme for a non-trivial large-scale realistic example problem and the ability of fast dynamic remapping if the FEM graph changes in time and we conclude with some summarizing remarks in Section 6.

2 Motivation

As already mentioned, the usual approach for mapping discretized PDEs to parallel computers is to perform some kind of solution domain decomposition in conjunction with a corresponding partitioning of the parallel computer (for an overview see e.g. [18]). Afterwards the partitions of the solution domain are mapped to their counterpart partitions of the parallel computer. Optimization goal for that decomposition is the minimization of inter-partition dependencies under the constraint of load balancing, i.e. all partition should include the same number of discretization points (or finite elements). Typically, this is done by recursive bisection: in each step, each partition is bisected independently of the others into two new ones, resulting in 2, 4, 8, 16, ... partitions. At the same time, the parallel machine is also partitioned by recursive bisection into as many compact partitions as there are solution domain partitions. For the usual hypercube- and mesh-connected parallel computers the procedure

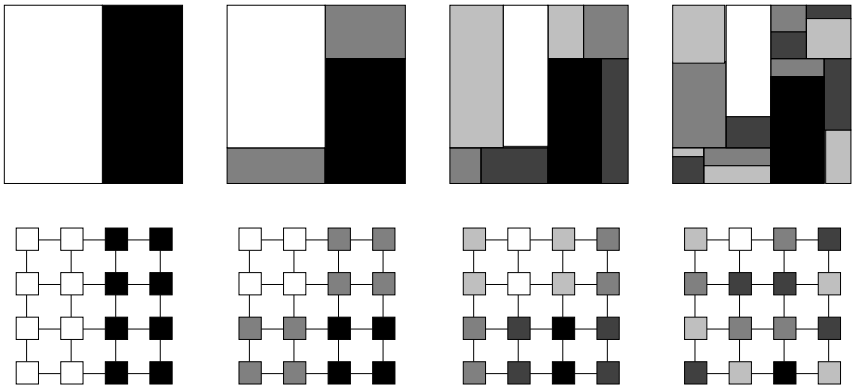


Figure 1: Partitioning of a rectangular solution domain and a mesh-connected parallel computer into 16 partitions by orthogonal recursive bisection. Corresponding partitions are shaded equally.

and the resulting correspondence between solution domain partitions and parallel computer partitions is obvious.

Figure 1 shows a small artificial example of a rectangular solution domain and a 16-processor parallel computer. Both are recursively partitioned in four steps resulting in 16 balanced partitions. Corresponding partitions are shaded equally. If we observe the evolution of the black and the white partitions which share some data dependencies we recognize that their counterpart machine partitions drift apart more and more. Once the distance of two partitions that share some data dependencies is more than one in at least one direction (horizontal or vertical) it grows further within at least every second further partitioning step because the machine partition in between is also partitioned further. This results in high-dilation embeddings and high communication cost especially for highly parallel computers.

Although this example is quite artificial, it demonstrates potential problems that arise when applying usual domain decomposition methods that treat all partitions independent of each other and do not take into account the interconnection topology of the parallel computer. Using more powerful recursive bisection methods than that of figure 1 may reduce the problems but cannot remove them. Nevertheless, recursive bisection methods perform well for today's mostly small- and medium-size parallel computers with small network diameters (e.g. hypercubes).

3 Formulation of the Problem

A parallel computer can be modeled as a so-called *Processor Connection Graph* $PCG = (P, E^P)$ with :

P	set of identical processor elements
E^P	set of bidirectional processor connections (links)
$\gamma(p,q)$	$\forall (p,q) \in E^P$ the communication delay (time/data unit)

Using the edge weights (that can be assumed to be unity for a homogeneous interconnection network) a distance metric on the PCG between arbitrary pairs of processors can be defined:

$$d \quad : \quad P \times P \Leftrightarrow\!\!\rightarrow R_0^+ \tag{1}$$

where $d(p,q)$ indicates the length of the shortest (weighted) path between the processors p and q.

Although this definition allows interconnection networks of arbitrary topology we restrict ourselves to 2- or 3-dimensional grid architectures (depending on the dimension of the solution domain) or a virtual embedding of such a grid into the physical topology (e.g. hypercube; for an overview of such embeddings see e.g. [16]). The advantage of such interconnection topologies is that they are relatively cheap to realize because of their small and constant node degree (independent of the machine size, in opposite to hypercubes) and that they are sufficient for the kind of problem we want to attack because of their topological similarity. Also a lot of actual existing parallel computers are built using these interconnection topologies, e.g. Cray T3D, Intel Paragon, MasPar MP1 and MP2 and Parsytec GC.

The problem for which we want to compute an appropriate partitioning and mapping to a parallel computer can be described as follows :

It consists of a 2- or 3-dimensional solution domain modeling a technical or physical system on which we want to solve a certain PDE numerically. This solution domain is discretized in an arbitrary, problem dependent way such that at each discretization point some kind of calculation has to be performed that depends on other discretization points. We make the following assumptions :

> Assumption 1 :
> The solution domain Q is embedded into $[0,1]^k$ ($k = 2$ or 3) depending on the actual dimension of the solution domain itself, not its embedding space (e.g. the roof of a building that is actually a 2-dimensional domain but probably embedded into a 3-dimensional space). If the dimension of the embedding space is larger than that of the solution domain it is assumed to be projected suitably into the corresponding lower-dimensional unity square.

> Assumption 2 :
> The solution domain Q is discretized resulting in a number of discretization points $x_i \in [0,1]^k$, $i = 1, ..., N$, $\Delta Q := \bigcup x_i$. To perform the desired calculations, only information exchange between *geometrically nearby* points is necessary, i.e. points x_i, x_j for which $||x_i \Leftrightarrow x_j||$ is small. The interpretation of *geometrically nearby* depends on the actual problem and the numerical method used, respectively, and the actual local discretization precision .

This last assumption represents the geometrically based communication structure of the problem that would have been hidden if the problem had been explicitly modeled as a general Task Interaction Graph with the discretization points representing the smallest unity that can be regarded as a task and drawing edges between all pairs of tasks that need to exchange information (as it was done in [9]). This is the same assumption that also underlies the conventional geometrically based partition strategies.

Next we define the mapping function :

$$\pi : \Delta Q \Leftrightarrow\!\!\rightarrow P \tag{2}$$

to be the result of a mapping procedure that for each discretization point determines its corresponding processor to which it is mapped. Note that the function π implicitly defines a partitioning of the solution domain.

The measures of quality we consider for the evaluation of our mapping heuristic are quite general. They do not depend on special hardware properties like SIMD vs. MIMD or packet-switching vs. wormhole routing but allow to assess the quality of a mapping for different areas of application due to their diversity. The first two measures refer to load unbalance that can be expressed as the maximum load (here the number of discretization points) of all processors:

$$LU_{max} := \max_{p \in P} |\{x \in \Delta Q | \pi(x) = p\}| \tag{3}$$

from which an upper bound of the overall computation speed can be derived. The second load balance measure is the average percentage load deviation from the load average :

$$LU_{dev} := \frac{1}{|P|} \sum_{p \in P} \left| \frac{|\{x \in \Delta Q | \pi(x) = p\}|}{\bar{L}} \right| \tag{4}$$

with

$$\bar{L} := \frac{|\Delta Q|}{|P|} \tag{5}$$

To measure the communication overhead we define the maximum dilation of the mapping, i.e. the maximum distance of two nodes that need to exchange information under the mapping π :

$$dil_{max} := \max_{\substack{x,\, y \in \Delta Q \\ \text{communicating}}} d(\pi(x), \pi(y)) \tag{6}$$

which can be used as a lower bound of the communication overhead especially on SIMD parallel computers. To measure the total communication overhead the communication costs are defined as the accumulated communication relations of all pairs of nodes that need to exchange information, weighted with the distance under the mapping π :

$$CC := \sum_{\substack{x,\, y \in \Delta Q \\ \text{communicating}}} d(\pi(x), \pi(y)) \tag{7}$$

Because our heuristic is geometry based and does not explicitly minimize a certain objective function we do not need to combine all these measures to one all-embracing function which is considered an art because of the contrary nature of the objectives.

4 Description of the mapping procedure

The Kohonen type of neural network has been developed as an approximate model for the self-organization process of the stimulus response of real, physiological neurons in the cerebral cortex [13, 17].

The Kohonen network can be simplistically explained as a set of *artificial neurons* denoted by P, arranged as a fixed grid (typically 2- or 3-dimensional) which adapt their response in terms of weights to external input from a feature or input vector space in a way that nearby neurons in the neural network respond to similar (nearby) stimuli in the feature vector space. Looking at a weight vector of a neuron as a reference vector or prototype for those stimuli vectors of the feature vector space for which it responds maximally (i.e. those stimuli vectors for which it is the geometrically nearest) the process organizes the weight vectors of the Kohonen network in a way that the Kohonen network can be regarded as a vector quantizer [17]. The induced Voronoi tesselation decomposes the feature vector space into compact partitions. Additionally, the Kohonen network has the property that neighboring partitions of the solution domain correspond to nearby neurons of the neural network. It is this point of view in which we want to explain Kohonen neural networks neglecting all its biology based features.

Roughly speaking, the idea how to use Kohonen neural networks for the mapping problem we consider, is to look at the set of discretization points of the solution domain as the discrete feature vector space and the parallel computer with its fixed communication topology as the neural network. The intuitive idea is that the self-organization process results in a good partitioning as well as mapping of the solution domain, optimizing both those aspects together with mutual consideration. This contrasts to usual domain partitioning methods that do not take into account the interconnection topology of the machine.

Assuming a feature vector space X of dimension k, each neuron p holds a weight vector $w_p \in R^k$. The *response* of a neuron p to an input signal $x \in X \subset R^k$ is inversely proportional to the distance $||x \Leftrightarrow w_p||$. The neuron p^* for which $||x \Leftrightarrow w_p||$ is minimum is called the *excitation center* for that specific input and is the neuron where x is mapped to. The Kohonen algorithm to adjust the weights in a way that a topology conserving mapping of the input space to the neural network is established, is to repeatedly and randomly choose an input vector of the feature vector space, compute the excitation center in the neural network and to adapt the weight vector of this neuron and those in its neighborhood in direction to the input vector. The stepsize for this adaptation is the smaller the larger the distance of the specific neuron to the excitation center is. In this way, the neurons adapt their weight vectors such that a good vector quantization is achieved and neighboring partitions correspond to nearby neurons of the neural network.

So the algorithm is as follows :

0 **Kohonen algorithm**
1 `initialize all weight vectors` $w_p \in P$
2 **For** `t:=1` **to** t_{max} **do**
3 `select` $x \in X$ `randomly`
4 `compute` p^* `with` $||w_{p^*} \Leftrightarrow x|| = \min_p ||w_p \Leftrightarrow x||$
5 $w_p^{t+1} \leftarrow w_p^t + \epsilon(t) h_{pp^*}(t)(x \Leftrightarrow w_p^t)$
6 **end for**

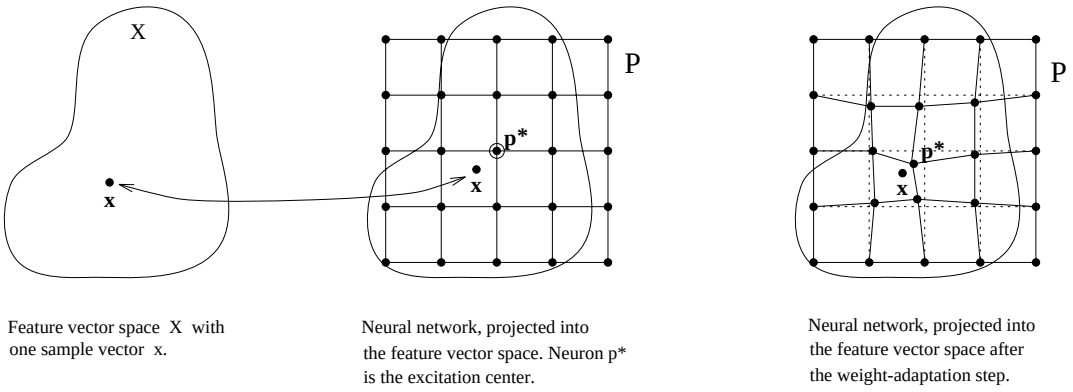


Figure 2: A single weight-adaptation step.

$\epsilon(t)$ is a monotonically decreasing stepsize; $h_{pp^*}(t)$, the so-called *neighborhood function*, is monotonically decreasing in $d(p, p^*)$ and describes the decreasing influence of an adaptation step to a specific neuron p with increasing distance from the excitation center p^*. This function is normally chosen to be Gaussian : $h_{pp^*}(t) := exp(\Leftrightarrow d(p, p*)^2/2\sigma(t)^2)$ where $\sigma(t)$, the *neighborhood size*, also decreases with time such that the network can evolve towards a roughly global order at the beginning of the process and is allowed to fine-tune at the end.

For each feature vector $x \in X$ we can define the mapping function $\pi : X \Leftrightarrow\!\!\rightarrow P$ implicitly by $\pi(x) = p \Leftrightarrow ||x \Leftrightarrow w_p|| = \min_{q \in P} ||x \Leftrightarrow w_q||$, its specific excitation center or in terms of vector quantization its reference neuron. So each discretization point is mapped to that processor that holds the weight vector with the smallest Euclidian distance to it. Using this we can define the *receptive field* $F(p)$ of a neuron p as the set of all feature vectors that are mapped to it:

$$F(p) \quad := \quad \{x \in X | \pi(x) = p\} \tag{8}$$

In the special case of $h_{pp*} = \delta_{pp*}$ (δ denotes the Kronecker-Delta) this algorithm is just the stochastic online version of the well known *k-means clustering* algorithm [17]. In distinction to it, the *soft-max* adaptation rule employed in the Kohonen algorithm enforces the process to consider topology conservation also in the neighborhood defined by the metric $d(p, q)$.

For the adaptation of the self-organization process described above to the mapping problem we assume a k-dimensional solution domain of a PDE discretized in some way, complying with assumptions 1 and 2 and embedded into $[0, 1]^k$ and according to this a parallel computer with a k-dimensional rectangular grid interconnection topology, representing the neural network.

The set of vectors of the discretization points ΔQ is taken as the feature vector space. This is justified, because in accordance with assumption 2, geometrical closeness of discretization points also denotes mutual data dependency. The weight vectors of the neurons, i.e. the processors respectively, are initialized according to the a-priori knowledge concerning the shape of the solution domain (assumption 1). That is: they are regularly and neighborhood preservingly distributed over $[0, 1]^k$, giving a first rough ordering. For example the weight vector of a processor with coordinates (i, j, k) of a parallel computer with a $\{0..n_1 \Leftrightarrow 1\} \times \{0..n_2 \Leftrightarrow 1\} \times \{0..n_3 \Leftrightarrow 1\}$ 3-dimensional interconnection topology is initialized to :

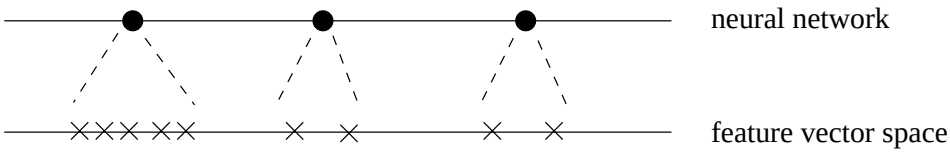


Figure 3: The distribution of processors (nodes of the neural network) in an 1-dimensional example : the load of the three processors (i.e. elements of the feature vector space) is not well balanced as the result of the self-organization process although the vector quantization error is minimum.

$$w_{ijk} := \left(\frac{i+1}{n_1+1}, \frac{j+1}{n_2+1}, \frac{k+1}{n_3+1} \right)^T \tag{9}$$

This ordered initialization allows to start the self-organization process with a small neighborhood-size $\sigma(t=0)$ and prevents it from falling into bad sub-optimal, neighborhood violating solutions, generally a very hard problem. At the end of the process, a compact partition of the solution domain is mapped to each processor, i.e. its receptive field, in a way that neighboring partitions that share data dependencies are mapped to nearby processors.

The self-organization process described so far tends to distribute the processors in a way that minimizes (for $\sigma \to 0$, i.e. at the end of the process) the vector quantization error:

$$E_{VQ} = \sum_{x \in \Delta Q} ||x \Leftrightarrow w_{\pi(x)}||^2 \tag{10}$$

This behavior is not in a good agreement with our objective of load balancing (see figure 3 for an example). Consequentely we need an additional mechanism to explicitly enforce load balancing. In [8] we proposed a mechanism that balances load by increasing or decreasing the size of the receptive fields of the processors (depending on whether they are underloaded or overloaded) at the expense of neighboring processors leading to a diffusion-like load balancing process. Because this process allows only load exchange between neighboring neurons (i.e. processors) and because it considers only local load unbalance, it would take a long time to achieve load balance between far off neurons if the load unbalance is more global and coarse grain. Hence, this mechanism is no longer applicable for highly parallel computers. Here we propose a new one but based on similar ideas.

The idea is to overlay the Kohonen algorithm with a drift-like process that guides the processors towards regions of the solution domain where processors hold a very heavy load, indicating that the processor density in such regions is not sufficient in comparison with the discretization fineness. This has to be done very carefully so that the global topological order established by the Kohonen process is not violated too much.

During the Kohonen process we periodically compute the complete mapping function π to determine the actual load of each processor. To save computational resources this can be done incrementally because it can be assumed due to the smooth behavior of the Kohonen process that the distance of the destination processors for each discretization point from one load-determination step to the next is very small. Each processor is then able to compute a local

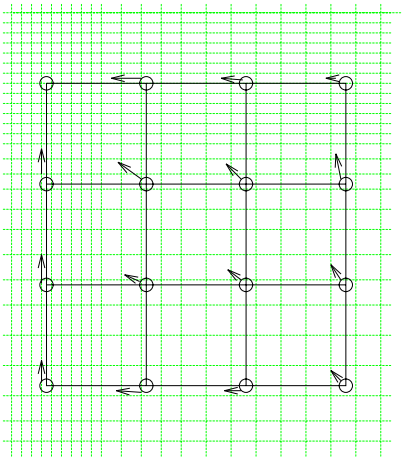


Figure 4: For each processor a small arrow shows the direction and the magnitude of it's load gradient. Processors adapt their weight vectors according to these load gradients to enforce load balancing.

load gradient and to adapt its weight vector a little bit in the direction to the neighboring processors' weight vectors according to the load gradient (see figure 4 for an illustrative example). According to the moves of the weight vectors, the receptive fields change in a way that load is balanced in a local surrounding depending on the number of nearby processors that are taken into account for the calculation of the gradient vector. In order to establish a roughly global load balance at the begining of the iterative process and a locally fine-tuned load balance at the end, being in accordance with the mode of operation of the Kohonen process, the region taken into account for the load gradient computation decreases in time. Considering a larger surrounding at the beginning does not only speed up the process but also results in a somewhat smoother gradient surface (see figure 5a). To ensure that a rough global topological ordering has been established before the self-organization process is overlayed with the additional load balancing process and probably disturbed by it, load balancing starts not until 1/3 of all iterations of the Kohonen process are already performed.

To bound the magnitude of the load gradient (otherwise resulting in too large, topology violating moves of weight vectors) and to ensure a smoother load gradient surface, the load gradient is not computed using the actual load values of the processors. Instead, each processor in the surrounding taken into account for a load gradient calculation contributes to it with a value monotonic in its actual load situation but bounded with respect to the average load (see also figure 5b) :

$$bounded\_load(p) := \begin{cases} \Leftrightarrow 3 & , \quad load(p) < 0.4 \cdot \bar{L} \\ \Leftrightarrow 2 & , \quad 0.4 \cdot \bar{L} \leq load(p) < 0.6 \cdot \bar{L} \\ \Leftrightarrow 1 & , \quad 0.6 \cdot \bar{L} \leq load(p) < 0.8 \cdot \bar{L} \\ 0 & , \quad 0.8 \cdot \bar{L} \leq load(p) \leq 1.2 \cdot \bar{L} \\ 1 & , \quad 1.4 \cdot \bar{L} \geq load(p) > 1.2 \cdot \bar{L} \\ 2 & , \quad 1.6 \cdot \bar{L} \geq load(p) > 1.4 \cdot \bar{L} \\ 3 & , \quad load(p) > 1.6 \cdot \bar{L} \end{cases} \tag{11}$$

The specific boundaries depend on the ratio of the number of discretization points to the

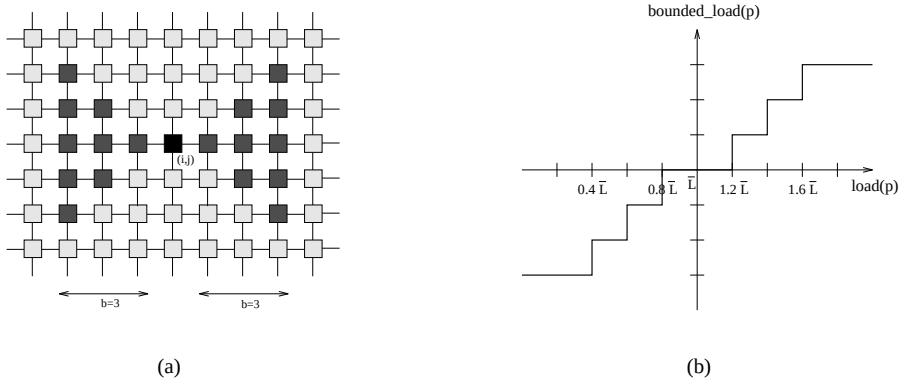


Figure 5: (a) All processors whose load state contribute to the horizontal component of the load gradient for a specific processor are shaded more darkly. This region decreases to only the left and right adjacent processor at the end of the process. (b) *bounded_load* - values that contribute to the load gradient calculation with respect to the average load ($\bar{L}$).

number of processors. For a higher total load the boundaries can be more restrictive because the load that is to be distributed evenly occurs in a finer granularity.

For example, the horizontal component of the load gradient for a specific processor p of a parallel computer with a 2-dimensional interconnection network with coordinates $p = (i, j)$ is computed according to :

$$grad\_load_x(i,j) := \sum_{k=-b}^{+b} \sum_{l=-|k|+1}^{|k|-1} bounded\_load(i+k, j+l) \cdot sgn(k) \tag{12}$$

where $b = b(t)$ denotes the size of the surrounding taken into account. The resulting algorithm looks like :

```
0 Load Balancing Process
1 Do every 150 iteration steps of the Kohonen Process :
2      Do every 15th call :
3           update the mapping function π and each processor's load
4           compute load gradient vector for each processor
5      EndDo
6      adapt each processor's weight vector according to :
7      w_p ← w_p + 1/(100·b²) · grad_load(p)
8 EndDo
```

5 Demonstration of an application

To demonstrate the capability of the proposed mapping heuristic and to give some more insight into it's mode of operation we report about some possible applications on a large-scale, non-trivial sequence of FEM-graphs.

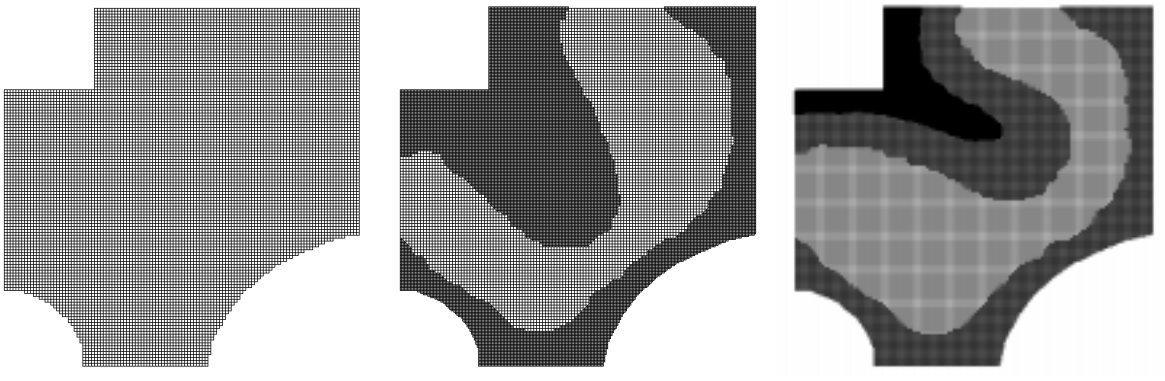


Figure 6: A sequence of hypothetical FEM-graphs, resulting from adaptive refinement. The first, most coarsest consists of 12,199 discretization points, the second of 31,682 and the most finest of 44,663.

Figure 6 shows a sequence of three FEM graphs, discretizing an irregular 2-dimensional domain with varying local precision, each resulting from adaptive refinement of the preceding graph as they emerge if the neccessary discretization precision is not known in advance or changes in time. These graphs are mapped to a 4096 processor machine (64x64 grid). The implementation and timing-measurement was done on a MasPar MP1 SIMD parallel computer, although this is not the type of machine we adress at first.

The first experiment we would like to report is the mapping of the FEM graph with the highest precision to the 4096 processor machine without consideration of the preceding coarser graphs. The Kohonen algorithm ran for $t_{max} = 600,000$ time steps incorporating the load balancing process in the last $400,000$ time steps. Figure 7 and 8 show a sequence of figures demonstrating the evolution of the mapping during the self-organization process. Figure 7 shows the grid of the communication network of the parallel computer projected into $[0,1]^2$ according to the coordinates of the weight vectors of the processors. It can be seen that this projection approximates the shape and the discretization precision of the FEM graph better and better with progressing time in a topology conserving way. This leads to an embedding of the FEM graph into the parallel computer with small overall communication cost and small dilation. Figure 8 shows the corresponding sequence of the 64x64 processor array shaded corresponding to the computational load of the individual processors (black for high load, white for low load). As can be seen the load distribution converges to an almost uniform density (gray level) as desired. The white areas correspond to processors with no computational load at all. In the very early stages of the process this occurs because they are mapped to regions of $[0,1]^2$ outside the solution domain. But during the self-organization process they move towards the solution domain in a topology preserving way, leading to a decrease of the number of idle processors and to a better load balance. At the end of the process, the tremendous global load unbalance of the initial mapping has disappeared leaving only a few processors with a noticeably higher load than their neighbors. This remaining unbalance could be further removed by a simple local balancing step to be performed only on these few processors without affecting neither communication cost nor maximum dilation significantly.

While the quality of the values for load unbalance, maximum load and maximum dilation are easy to assess, the quality of the values for communication cost remain quite abstract. To

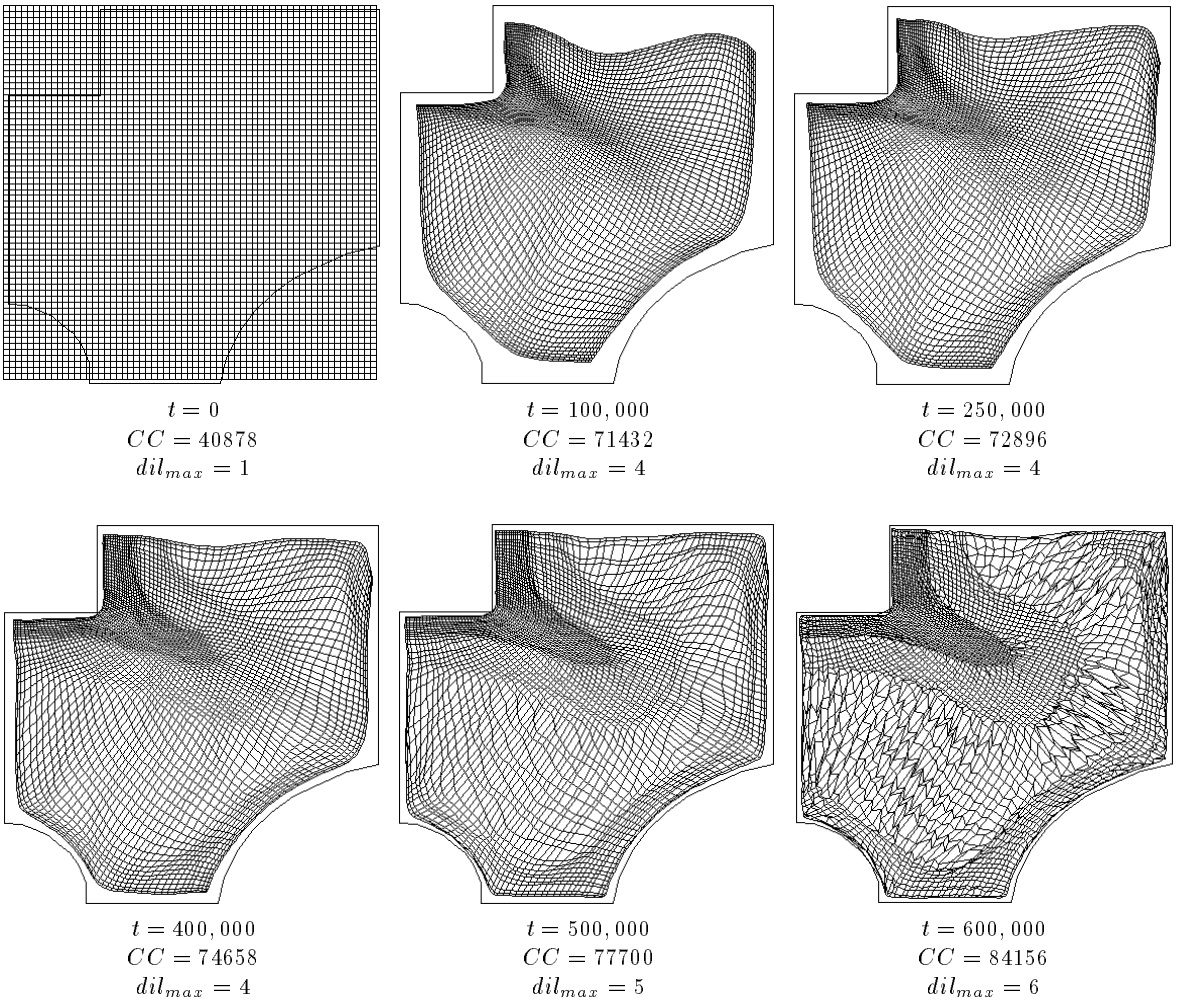
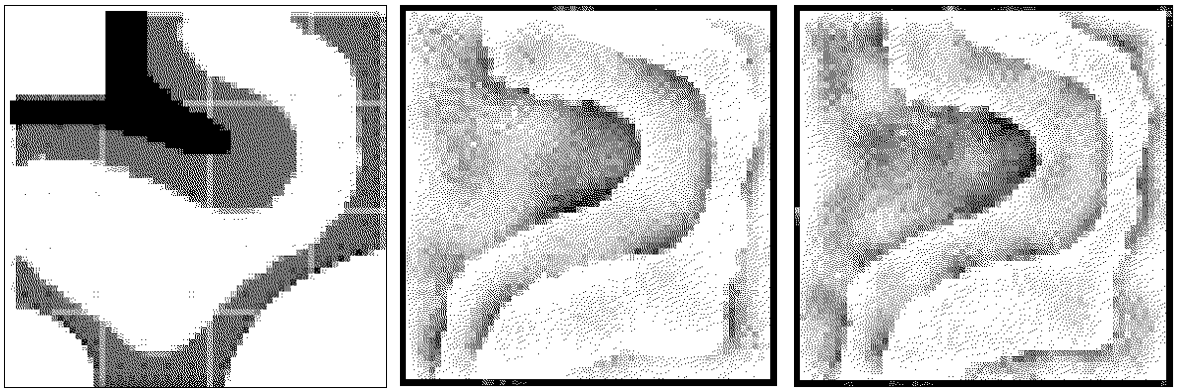


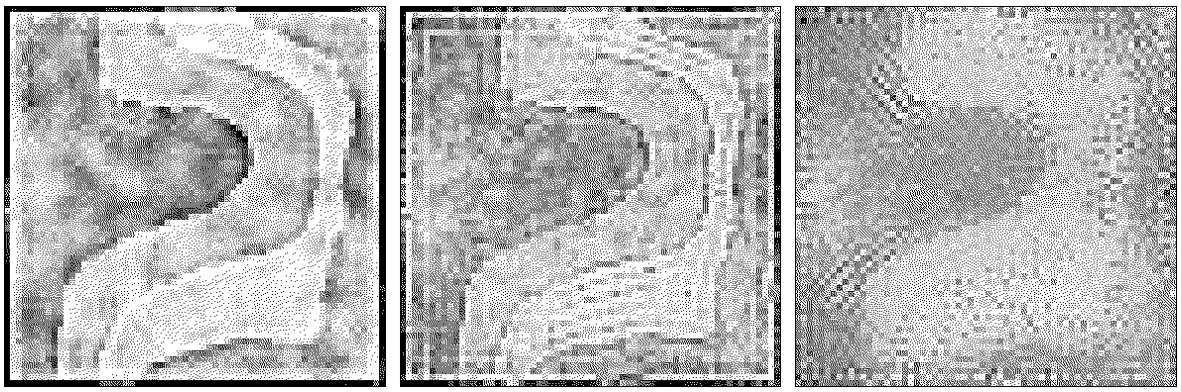
Figure 7: Snapshot sequence of intermediate mappings during the Kohonen process. The Processor Connection Graph is projected into the solution domain whose boundary is drawn only. Discretization points are mapped to the geometrically nearest processor. (Process parameters were $t_{max} = 600,000$, $\sigma(t) = 6 \Leftrightarrow 5.75\frac{t}{t_{max}}$, $\epsilon(t) = 0.06 \Leftrightarrow 0.02\frac{t}{t_{max}}$, $b(t) = \lfloor 5 \Leftrightarrow 4\frac{t}{t_{max}} \rfloor$)



$t = 0$
$LU_{max} = 75$
$LU_{dev} = 0.93$

$t = 100,000$
$LU_{max} = 404$
$LU_{dev} = 0.60$

$t = 250,000$
$LU_{max} = 218$
$LU_{dev} = 0.49$



$t = 400,000$
$LU_{max} = 110$
$LU_{dev} = 0.35$

$t = 500,000$
$LU_{max} = 48$
$LU_{dev} = 0.25$

$t = 600,000$
$LU_{max} = 22$
$LU_{dev} = 0.17$

Figure 8: Snapshot sequence of intermediate load states of the processor array during the Kohonen process. Black benotes heavy load, white a very low load ($\bar{L} = 10.90$).

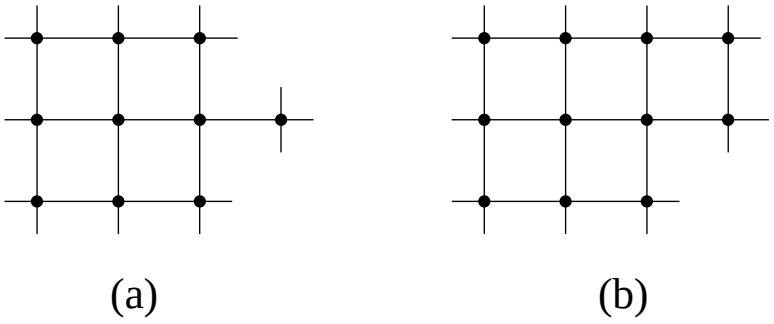


Figure 9: Optimum compact domain partitions consisting of 10 (a) and 11 (b) points, assuming a rectangular grid.

get an impression how this value can be assessed we perform the following rough estimate :

The example FEM graph consists of 44,663 discretization points. Assuming a rectangular grid discretization, inner points have 4 neighbors, boundary points only 3 (a few of them only 2). For the FEM graph given, this results in an average number of data dependencies of 3.96 per point. On a 4096-node processor the load average is 44,663/4096=10.9, so with an optimum balanced load, most processors (3703) are assigned 11 points, the rest (393) are assigned 10 points. Assuming optimum domain partitions (minimum data dependencies) each partition (regardless whether it has 10 or 11 points) would have 14 edges to other partitions if all points had 4 neighbors (figure 9). Since the average number of neighbors is actually 3.96 we obtain $14 \times (3.96/4) = 13.86$ data dependencies for each partition or 13.86 communication relations crossing processor boundaries, respectively. If, ideally, only nearest neighbor communication were necessary, the communication cost would be $CC = 4096 \times 13.86 = 56,770$. This is a lower bound which is theoretically unreachable because due to the goal of load balancing, dilations greater than 1 are often unavoidable. Also the assumption of optimum compact partitions is unrealistic.

Compared to our results, partitioning and mapping using the orthogonal recursive bisection method leads to communication costs that are about 10% higher (25% for a 16,384 processor parallel computer) and maximum dilations that are 3 times larger (4 times for a 16,384-processor parallel computer). To confirm the predicted problems of recursive bisection methods that served as a motivation for this work, figure 10 shows the embedding of the most finest FEM graph into a 256-processor parallel computer, computed with orthogonal recursive bisection and with the algorithm proposed here. The predicted large dilations ($dil_{max} = 6$) develop exactly at the boundaries of the different bisection steps. Additionally these dilations lead to an unintentional edge congestion resulting in further communication overhead which is not considered in our quality measures. So even if dilation is not an issue (since the interconnection network has small global propagation delays), there might occur some hot spot communication links, that could degrade the overall performance. As can be seen, the Kohonen based mapping shows smaller dilations ($dil_{max} = 4$) and more evenly distributed communication edges avoiding congestion.

The next experiment demonstrates the dynamic remapping capabilities of the procedure. Assuming that a mapping for one of the more coarser FEM graphs has already been computed we show how to exploit this mapping to find a good mapping for the succeeding FEM graph within less time than starting from nothing. Such situations are very typical when the discretization precision that is necessary to solve a certain PDE within a given precision varies

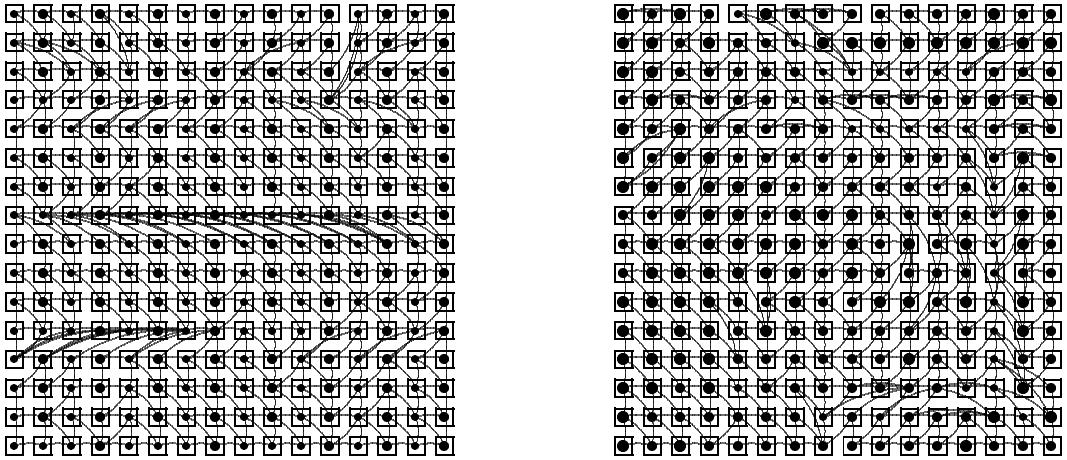


Figure 10: Mapping of the 44,663-node FEM graph to a 256-processor parallel computer using orthogonal recursive bisection (left) and the algorithm proposed here (right). Different sizes of the black circles in the boxes (i.e. processors) represent different computational loads. An edge is drawn between all pairs of processors that need to perform communication under the computed mapping.

from region to region and is not known in advance (this corresponds to a transition from the first FEM graph of figure 6 to the second one) or if the necessary local discretization precision changes from one time step to the next in non-stationary circumstances (this corresponds especially to the transition from the second FEM graph to the third).

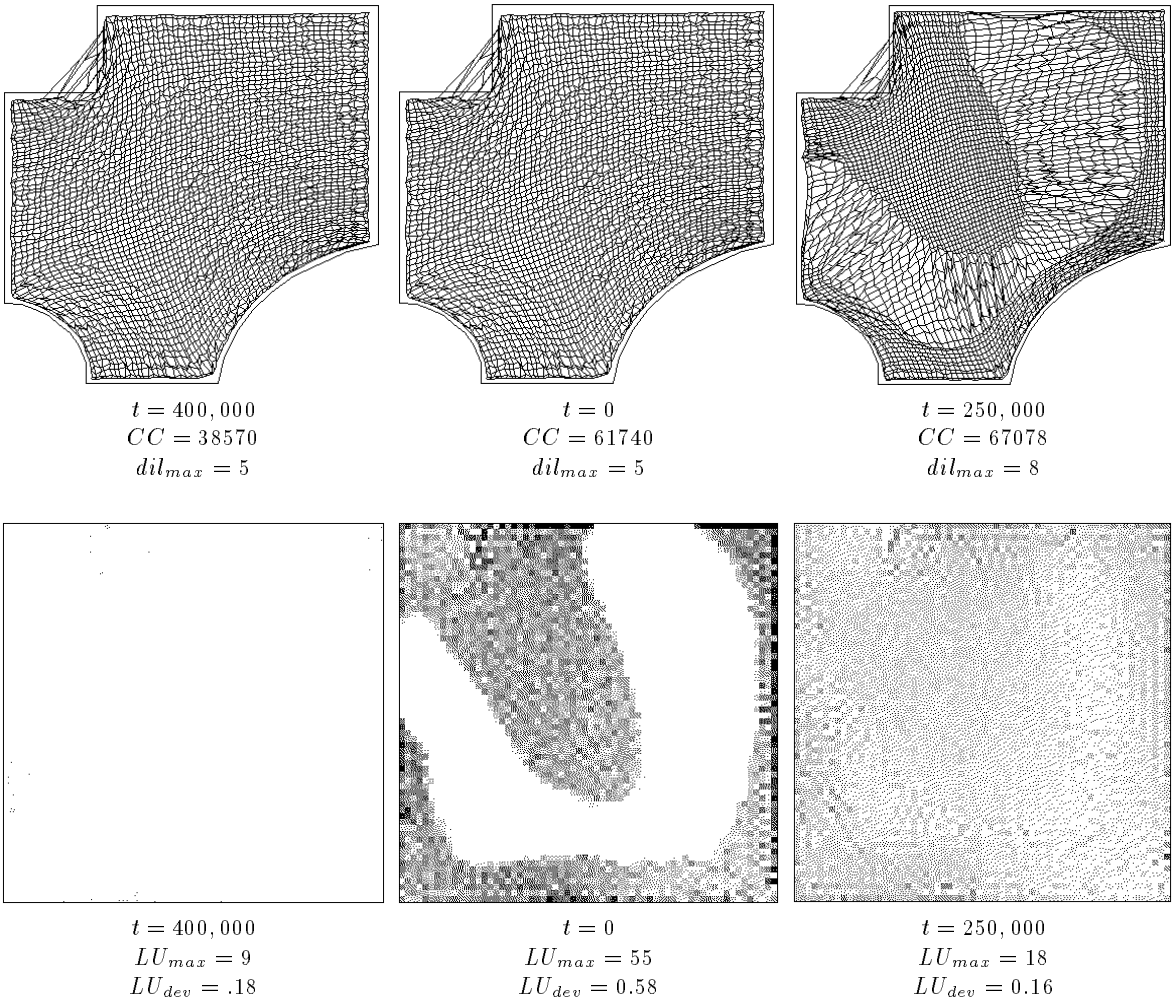
To compute such successive mappings we take the last mapping (more precisely its computed weight vectors) as an approximation for the mapping that is to be computed. The similarities of such successive mappings allow us to start with a more narrow neighborhood function $h(p, p^*)$ assuming that a rough ordering already exists and we need only to perform a fraction of the iteration steps that would have been necessary if starting from nothing. Doing so we can save about 2/3 of computation time. Figure 11 shows the transition from the first graph to the second. Note the good correspondence to the results of mapping the second FEM graph with starting from nothing although the differences between the first and the second FEM graph are quite drastic.

The run time needed to compute mappings with the proposed heuristic is e.g. for the third FEM graph about 16 minutes on 4096 processors of a MasPar MP1 with a theoretically peak performance of 300 32-bit MFLOPS.

6 Conclusions

We presented a new mapping heuristic specialized for application to large-scale grid-like problems that exhibit very local communication patterns. These properties are typical for numerical methods for solving partial differential equations, e.g. with Finite Element or Finite Difference Methods.

As the presented results demonstrate it is possible to perform data partitioning (or domain partitioning) and processor allocation at the same time under consideration of each other within a reasonable amount of time. This gives the potential of better mappings which is particularly important for highly parallel computers with larger communication network



$t = 400,000$
$CC = 38570$
$dil_{max} = 5$

$t = 0$
$CC = 61740$
$dil_{max} = 5$

$t = 250,000$
$CC = 67078$
$dil_{max} = 8$

$t = 400,000$
$LU_{max} = 9$
$LU_{dev} = .18$

$t = 0$
$LU_{max} = 55$
$LU_{dev} = 0.58$

$t = 250,000$
$LU_{max} = 18$
$LU_{dev} = 0.16$

Figure 11: Dynamic remapping when changing from FEM graph 1 to FEM graph 2. The first column shows the mapping of graph 1 from that the computation started. The middle column shows the resulting mapping if the weight vectors of the preceding mapping are used to map the new graph, the last column shows the result of the mapping process (result of mapping the second FEM graph with starting from nothing using $600,000$ iteration steps are: $CC = 68150$, $dil_{max} = 8$, $LU_{max} = 15$, $LU_{dev} = 0.16$).

diameters. More precisely, not only the amount of data dependencies between interacting partitions is minimized but this is done with respect to the mapping problem resulting in lower communication cost finally. This is possible because special properties of the communication structure are exploited, namely the low-dimensionality, the locality and topological similarities of the communication structure and the target parallel computer.

The proposed mapping heuristic can be parallelized itself very efficiently and is also capable of dynamic remapping which is necessary if the communication structure changes in time. The necessity for dynamic remapping is often mentioned but only very rarely demonstrated for other mapping heuristics.

References

[1] Basermann, A. : *Conjugate Gradients Parallelized on the Hypercube.* Internal Report KFA-ZAM-IB-9309 Research Center Juelich 1993 (also to be published in: Int. Journal of Modern Physics C).

[2] Barragy, E.; Carey, G.F.; van de Geijn, R. : *Performance and Scalability of Finite Element Analysis for Distributed Parallel Computation.* Technical Report CNA 254, University of Texas, Austin, 1993 (also to be published in : J. of Parallel and Distr. Processing)

[3] Bollinger, S. W.; Midkiff, S. F. : *Heuristic Technique for Processor and Link Assignment in Multicomputers.* IEEE TOC Vol. 40,3 (March 1991), pp. 325-333.

[4] Bultan, T.; Aykanat, C. : *A New Mapping Heuristic based on Mean Field Annealing.* J. of Parallel and Distributed Computing 16 (1992), pp.292-305.

[5] Chockalingam, T.; Arankumar, S. : *A randomized heuristic for the mapping problem: The genetic approach.* Parallel Computing 18 (1992), pp. 1157-1165.

[6] Dormanns, M.; Heiss, H.-U. : *Topology Conserving Graph Mapping by Self-Organization: A Solution to the Processor Assignment Problem.* In: Albrecht, R.F. et.al. (eds) Artificial Neural Networks and Genetic Algorithms, Conf. Proc. (Innsbruck 14.-16. April 1993), Springer Wien, pp. 198-205.

[7] Fox, G. C. : *A Graphical Approach for Load Balancing and Sparse Matrix Vector Multiplication on the Hypercube.* In: Numerical ALgorithms for Modern Parallel Computers, M. Schultz (ed.), Springer Verlag, Berlin 1988.

[8] Heiss, H.-U.; Dormanns, M. : *Task Assignment by Self-Organizing Maps.* Internal Report 17/93 (April 1993), University of Karlsruhe.

[9] Heiss, H.-U.; Dormanns, M. : *Mapping Large Parallel Simulation Programs To Multicomputer Systems.* High Performance Computing Conference 94, Part of the SCS 1994 Simulation Multiconference, La Jolla, California, April 11-15 1994.

[10] De Keyser, J.; Roose, D. : *Adaptive Irregular Multiple Grids on a Distributed Memory Multiprocessor.* In Bode (ed.): Proc. of the 2nd European Distr. Memory Conf. 1991.

[11] De Keyser, J.; Roose, D. : Partitioning and Mapping Adaptive Multigrid Hirarchies on Distributed Memory Computers. Report TW 166, Feb. 1992, Dept. of Computer Science, K. U. Leuven, Belgium.

[12] De Keyser, J.; Roose, D. : *Load balancing data parallel programs on distributed memory computers.* Parallel Computing 19 (1993), pp. 1199-1219.

[13] Kohonen, T. : *Self-Organization and Associative Memory.* 3rd edition, Springer Verlag, Berlin 1989.

[14] Lo, V. M. : *Heuristic Algorithms for Task Assignment in Distributed Systems.* IEEE TOC Vol 37, No. 11 (Nov. 1988), pp. 1384-1397.

[15] Mühlenbein, H.; Gorges-Schleuter, M.; Krämer, O. : *New solutions to the mapping problem of parallel systems : The evolution approach.* Parallel Computing 4 (1987), pp. 269-279.

[16] Monien, B.; Sudborough H. : *Embedding one Interconnection Network in Another.* Computing Suppl. 7 (1990), pp. 257-282.

[17] Ritter, H.; Martinetz, T.; Schulten, K. : *Neural Computation and Self-Organizing Maps.* 2nd ed. Addison Wesley, 1991.

[18] Roose, D.; Van Driessche, R. : *Distributed Memory Parallel Computers and Computational Fluid Dynamics.* Report TW 186, March 1993, Dept. of Comp. Science, K.U. Leuven, Belgium.

[19] Sadayappan, P.; Ercal, F.; Ramanujan, J. : *Cluster partitioning approaches to mapping parallel programs onto a hypercube.* Parallel Computing 13 (1990), pp. 1-16.

[20] Savage, J.E.; Wloka, M.G. : *Mob - A Parallel Heuristic For Graph-Embedding.* Technical Report CS-93-01, January 1993, Brown University.

[21] Williams, R.D. : *Performance of dynamic load balancing algorithms for unstructured mesh calculations.* Concurrency: Practice and Experience, Vol. 3(5), pp. 457-481, Oct. 1991.

Appendix

Time complexity and implementation issues

As already mentioned, there is a one-to-one correspondence between the Kohonen neural network and the parallel computer with its interconnection network for which a mapping is to be computed. Consequently, a parallel implementation presents itself. We will not give a closed-form expression for the asymptotic parallel time complexity of the proposed algorithm because this depends also on hardware and architectural specific properties of the parallel computer. Instead we discuss each component of the algorithm separatly, giving also some implementation hints.

The first point to be discussed is the number of iteration steps of the Kohonen algorithm that are required. This number depends exclusively on the complexity of the shape of the solution domain of a certain PDE and the discretization precision variations along the solution domain and not on the actual number of discetization points or on the machine size. This astonishing statement needs some more explanation. On the one hand, when applying the procedure to a smaller machine with less processors, the corresponding weight vectors move more rapidly through the feature vector space to approximate its shape and discretization density because an individual neuron will be more often the excitation center or at least near to it. On the other hand we choose the neighborhood size $\sigma(t)$ larger when applying this procedure to a larger machine with more processors. So the total amount of weight vector adaptation in one iteration step accumulated over all neurons that take part (with the specific step-size dependent on the neighborhood function $h(p, p^*)$) scales with the machine size. Furthermore, increasing the number of discretization points without modifying the shape and the complexity of the discretization precision variations of the solution domain (this means scaling the discretization precision equally everywhere) does not change the task for the geometrically based mapping procedure that is to approximate the shape of the given solution domain and the density of the discretization points, although this discretization precision is then given more detailly.

Every single adaptation step contains the distribution of the choosen discretization point to all processors, a minimum-operation over all processors and some calculations that can be performed just locally in parallel. The cost for these communication operations depend highly on the specific parallel computer. Some of them are able to perform such operations very fast (especially most SIMD machines, like the MasPar MP1) or provide special hardware to support such operations (like the CM-5). To further reduce this communication overhead it is possible to modify the algorithm such that these communication operations can be grouped together over some iterations, reducing the communication overhead. Therefore, not just one but a whole set of discretization points that are choosen as stimuli vectors are distributed to the processors together as one package. Then, each processor computes locally the distance of its weight vector to all the stimuli vectors, neglegting changes of the weight vector that would have happened in the meantime considering the original algorithm. The excitation centers of all these stimuli vectors can be computed simultaneously and all weight vector adaptation steps also altogether. This modification is usually called *epoch learning* and has generally no negative influence on the result of the process.

The discussion of the load balancing process is somewhat more difficult. The main and most expensive operation is the calculation of the mapping function π which is needed to compute each processor's load state. Because this process starts not before a rough ordering

has already been established, it can be assumed that the mapping does not change very fast any more so it is sufficient to update the mapping and the load state only from time to time (the actual frequency we used in our experiments is every 2250 iterations of the Kohonen process). Because of the smooth behaviour of the Kohonen process it is reasonable to assume only local changes resulting in an almost local communication pattern for this parallel mapping calculation. Although, up to this end the mapping function has to be computed for each discretization point. But following the same argumentation as before, it is sufficient for load balance purposes to consider nearby discretization points only in clusters, taking their center of gravity as position in the solution domain and weighted with respect to the number of discretization points included for load calculation. As long as only the density variations of the discretization points is described sufficiently precisely this heuristic does not change results very much. The clustering can be done in a pre-computation step. An upper bound for the size of the clusters is given by the complexity of discretization precision variations.

All together, the parallel time complexity of the whole procedure is almost independent of the problem size. It is influenced by the machine size only in how fast broadcast and combine communication operations can be performed. Instead it depends on the complexity of the shape of the solution domain of a certain PDE and the complexity of discretization precision variations, which is a highly desirable feature considering the typically very large problem sizes.