ELSEVIER

Simulation Practice and Theory 4 (1996) 81-96

SIMULATION
PRACTICE THEORY

Simulation of load balancing with cellular automata [1]

R. Hofestädt [a,b,*,2], X. Huang [a,2], D. Beerbohm [a,2]

[a] *Department of Computer Science, University of Koblenz-Landau, Rheinau 1, D-56075 Koblenz, Germany*
[b] *Department of Medical Computer Science and Statistics, University of Leipzig, Liebigstr. 27, D-04103 Leipzig, Germany*

Received 24 August 1994; revised 1 June 1995

Abstract

In this paper the applicability and the speedup potential of a new load balancing strategy for distributed and parallel systems will be investigated by simulations. For this purpose, we will define a cellular automaton system which can produce process migration proposals according to the load states of computing nodes. The new strategy will be tested using our simulation environment CABLE. The simulation results for workstation clusters indicate that this new method possesses good applicability and a significant speedup.

Keywords: Cellular automata; Dynamic load balancing; Massively parallel processing; Workstation cluster

1. Introduction

Massively parallel processing in the case of complex workstation clusters is of increasing interest, because such architectures can usually provide much more attractive cost/performance than large computers [4,8]. Most of the existing workstations are connected with LANs and WANs. To support the distributed programming on a workstation cluster, several development platforms, e.g., PVM, have been provided. However, the usage of workstation clusters, primarily in a single LAN, requires new methods of dynamic load balancing to achieve a good utilization of a complete system [3,5,9,11].

* Corresponding author. Email: ralf@imise.uni-leipzig.de.
[1] This work was supported in part by the Ministry of Education and Science of Rheinland-Pfalz, Germany.
[2] Email: {beerbohm,hofestae,huang}@informatik.uni-koblenz.de.

SSDI 0928-4869(95)00045-3

For such a purpose we will present a new strategy based on the concept of "intelligent tables" [1]. That means, the information necessary for load balancing will be processed in specific data structures which will be realized by specific cellular automata. Under the conditions of a system architecture described in the next section, we will introduce our cellular automaton system which meets these requirements. Moreover, we will present the main characteristics of our simulation environment CABLE (Cellular Automata Based Load Balancing Experiments). CABLE allows users to define arbitrary virtual workstation clusters and to simulate their behaviour under different simulation parameters selected by users. Using CABLE, we will analyse and compare the performance of different virtual workstation clusters without and with regards to load balancing by means of cellular automaton systems.

2. System architecture overview

The regarded system consists of a number of network segments (e.g., a simple bus or ring with a token based protocol) which are connected with a main control and communication unit MCCU (see Fig. 1). Each segment is composed of a certain number of computing nodes. We assume that the computing nodes are homogeneous. In order to realize communication between processes in different segments, the corresponding segments can be switched on by the MCCU which behaves like a bridge/gateway. The MCCU is the main component of the system, to which the segments as well as the console are joined.

The basic component of the MCCU is the mapper, which distributes new applications and performs load balancing dynamically. The mapper is based on a 2-dimensional cellular automaton system, where the number of its cells depends on the configuration of the network segments. Each node is represented by one cell. In Fig. 1, m stands for the number of segments and n for the number of nodes in one segment. According to the state of the cellular automaton system, the mapper generates process migration proposals and sends migration information to the nodes concerned which perform the real process migration by themselves. We assume that the network system provides server routines

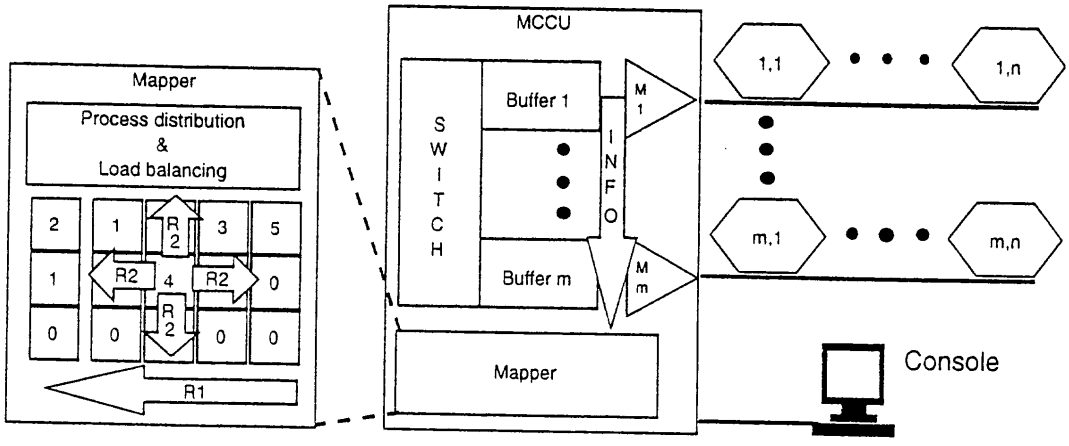

Fig. 1. Architecture overview.

which initialize the workstation cluster and realize the migration of processes.

Within the MCCU there is one monitoring unit for each segment (M_i, $i = 1, \ldots, m$), which inspects the load of the segment and the nodes within the segment, collects their load information and prepares the collected data for the mapper.

3. Monitoring unit

For each segment, the corresponding monitoring unit inspects the segment load from two complexity measures: the communication cost and the number of processes. It records the communication costs of the individual nodes and calculates the average communication rates from the recorded communication costs in a certain period. Furthermore, it combines this data with the number of processes on the corresponding nodes to determine the total load of each individual node and transfers the total load to the mapper periodically by setting the state of the corresponding cell of the cellular automaton.

Each monitoring unit is connected to a network segment physically and provides each node in the segment with one register to record communication costs; let $c(i)$ represent the register for the ith node. The monitoring unit inspects all messages transferred through the segment and can obtain the following information from a message header: message size, sender address and receiver address. If the sender of a message with the size of l bytes is located on the ith node and the receiver on the jth node within the same segment, the half of the message size $(l/2)$ will be added to $c(i)$ and $c(j)$ respectively, so that the whole segment has the communication costs of l bytes. If only the sender or the receiver is located on the kth node within this segment, the whole message size l has to be added to $c(k)$, because such a message concerns two segments and each of them has costs of l bytes. However, it is not necessary to distinguish the costs for sending messages from the costs for receiving messages, because the direction of a message transfer has no meaning to the communication load of a segment.

Letting τ be the evaluation period and d the data throughput of a network segment, the communication rate $r(i)$ of the ith node will be calculated from $r(i) = c(i)/(d \times \tau)$. Letting c_1 $(0 < c_1 \leq 1/n)$ and c_2 $(1/n < c_2 < 1)$ be two constants, we define the communication load states $S_c(i)$ of the ith node as follows:

$$
\begin{aligned}
S_c(i) ::= 0 &\iff r(i) = 0 && \text{(no communication)},\\
S_c(i) ::= 1 &\iff 0 < r(i) \leq c_1 && \text{(low communication load)},\\
S_c(i) ::= 2 &\iff c_1 < r(i) \leq c_2 && \text{(normal communication load)},\\
S_c(i) ::= 3 &\iff r(i) > c_2 && \text{(high communication load)}.
\end{aligned}
$$

Besides the communication load, the second load factor is the process load, i.e., the number of processes running on the same node. We assume that on each node there is a server routine which records the number of processes and sends it to the monitoring unit at a certain time.

Letting $p(i)$ be the number of processes on the ith node, and p_1 and p_2 (integer constants) represent the lower and upper limits of process load respectively, we define

Table 1
The total load states of nodes

$S(i)$		$S_c(i)$			
		0	1	2	3
$S_p(i)$	0	0	0	0	0
	1	1	1	2	4
	2	2	2	2	4
	3	3	3	3	5

the process load states $S_p(i)$ of the ith node as follows:

$$S_p(i) ::= 0 \iff p(i) = 0 \quad \text{(idle)},$$
$$S_p(i) ::= 1 \iff 0 < p(i) \leq p_1 \quad \text{(low process load)},$$
$$S_p(i) ::= 2 \iff p_1 < p(i) \leq p_2 \quad \text{(normal process load)},$$
$$S_p(i) ::= 3 \iff p(i) > p_2 \quad \text{(high process load)}.$$

The monitoring unit calculates the abstract total load state $S(i)$ of the ith node by combining $S_c(i)$ with $S_p(i)$ (see Table 1) and sends $S(i)$ to the corresponding cell of the cellular automaton. The meaning of $S(i)$ can be found in Definition 1. In addition to the abstract total load state, the mapper needs some detailed information about the number of processes. So, the monitoring unit has also to send the number of processes to the mapper.

4. Mapper – Cellular automaton system

The load states of nodes in a workstation cluster can be visualized in 3-dimensional graphics (see Fig. 2). If no load balancing algorithm is used in such a system, the load states of different nodes are usually quite diverse (see Fig. 2(a)), i.e., some nodes are overloaded (presented by peaks), while the others are idle or underloaded. In order to achieve a good utilization of the complete system (see Fig. 2(b)), the use of load balancing algorithms is necessary.

Ideally, the load of individual nodes should be balanced globally in the system. In practice, however, this is impossible because maintaining a global consistent load state

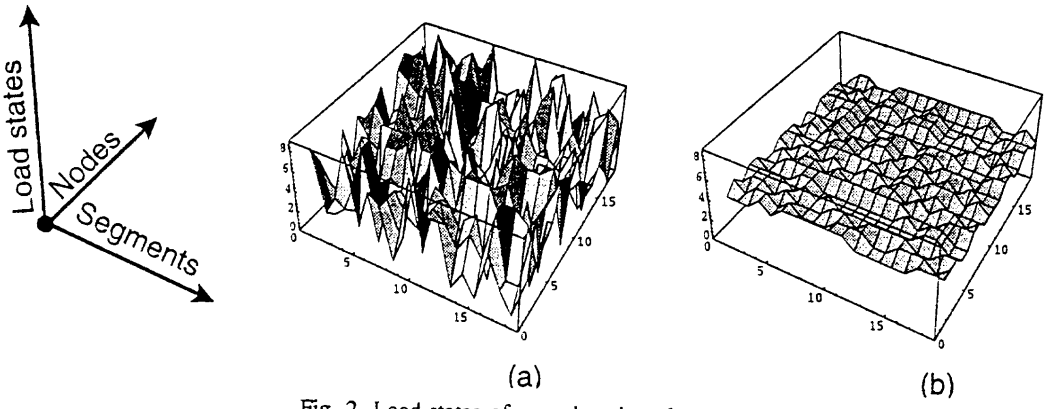

Fig. 2. Load states of a workstation cluster.

of the system is too expensive due to the dynamic behaviour. Therefore, a generally effective solution often aims to balance the load locally in a certain neighbourhood. The theory of cellular automata [10] has been proven to be an elegant method to produce a global transformation by local modifications (in the neighbourhood). Here, the behaviour of a cell will be determined by the application of transformation rules in dependency of the states of cells in the neighbourhood. Our basic idea is to use a cell of a cellular automaton to represent a computing node of a workstation cluster, where the state of the cell stands for the load state of the node. In this way, a global load balancing solution can be produced just regarding local load states.

For this purpose, we have defined a cellular automaton system which consists of a 2-dimensional grid and an additional column vector [1]. The grid represents load states of the workstation cluster and each cell represents the load state of the corresponding node. However, each cell of the column vector represents the abstract load state of the corresponding segment. Therefore, we call the column vector an abstract automaton ($A1$, see Definition 1) and the 2-dimensional grid a load balancing automaton ($A2$, see Definition 2). $A1$ and $A2$ are driven by rule schemes $R1$ and $R2$ respectively. After the monitoring unit has set the automaton system, the mapper will activate $R1$. The states of $A1$ determine the necessity of dynamic load balancing in the system. If the mapper identifies overloaded states in $A1$, $R2$ will be activated to produce a migration proposal. In order to decide whether the migration proposal is useful, the mapper will activate $R1$ once again and compare the new states of $A1$ with the old ones.

Here, we do not intend to introduce the fundamentals of the theory of cellular automata, which can be found in [10]. In the following we define the abstract automaton $A1$ and the load balancing automaton $A2$, and present two representative rule schemes $R1$ and $R2$ which were developed interactively by means of our simulation shell CASS [6]. The individual rules of a rule scheme will be processed successively, until one rule is executed successfully.

Definition 1. The 4-tuple $A1 = \{S, NB, G, c_0\}$ is called an *abstract automaton*, where

- $S = \{0, 1, 2, 3, 4, 5, 6\}$ is a finite set of cell states with 0 = idle, 1 = underload, 2 = normal load, 3 = process overload, 4 = communication overload, 5 = complete overload, 6 = unavailable (cf. Table 1);
- $NB(i)$ corresponds with the ith row of $A2$;
- the global transformation G, $R1$, is based on the local transformation $g \subseteq S^n \times S$;
- c_0 is the initial state of $A1$.

Regarding all elements of the ith row of $A2$, the abstract load state of the corresponding segment can be calculated by $R1$. The following is a representative rule scheme $R1$, where $M(i) = \frac{1}{n}\sum_{j=1}^{n} A2(i,j)$ stands for the mean load state of the ith row and $k, o \in \{1, \ldots, n\}$ represent two system specific constants:

$$R1.1: \quad A1(i) ::= 0 \Longleftrightarrow M(i) = 0$$

$$R1.2: \quad A1(i) ::= 1 \Longleftrightarrow M(i) < 1.3$$

$$R1.3: \quad A1(i) ::= 2 \Longleftrightarrow M(i) \leq 2.5$$

$R1.4$: $A1(i) ::= 3 \Longleftrightarrow (2.5 < M(i) \leq 3.5) \wedge (\forall j\ A2(i,j) = 3)$
with $j \in \{j_1, \ldots, j_k\}$ and $j_1, \ldots, j_k \in \{1, \ldots, n\}$

$R1.5$: $A1(i) ::= 4 \Longleftrightarrow (3 < M(i) \leq 4) \wedge (\forall j\ A2(i,j) = 4)$
with $j \in \{j_1, \ldots, j_o\}$ and $j_1, \ldots, j_o \in \{1, \ldots, n\}$

$R1.6$: $A1(i) ::= 5 \Longleftrightarrow M(i) > 2.5$

$R1.7$: $A1(i) ::= 6 \Longleftrightarrow M(i) = 6$

In order to simplify the rule scheme $R2$ so that the efficiency of the load balancing automaton $A2$ can be increased, $A2$ is defined as a homogeneous automaton.

Definition 2. The 4-tuple $A2 = \{S, NB, G, c_0\}$ is called a *load balancing automaton*, where

- $S = \{0, 1, 2, 3, 4, 5, 6\}$ is a finite set of cell states which are defined analogously to the states of $A1$ (see Definition 1);
- NB represents the *Moore* neighbourhood with radius 1, i.e., $h = 8$ (h stands for the number of neighbours);
- The global transformation G, $R2$, is based on the local transformation $g \subseteq S^h \times S$;
- c_0 is the initial state of $A2$.

The state of each cell in $A2$ will be changed by $R2$, depending on the states of the corresponding neighbours. The following presents a representative rule scheme $R2$, which belongs to the class of hybrid-multiple cellular automata, i.e. $R2$ consists of two parts ($R2.1.1$–$R2.1.8$ and $R2.2.1$) representing two rule schemes (also see Section 6). $A2$ is able to switch between the first part of $R2$ and the second part according to the actual load states. In the first part of $R2$ we distinguish between two cases:

(1) If the state of the current cell is 0 or 1, the neighbour with the "highest" state will be identified. If the "highest" state is greater than 1, it will be proposed to move some processes from the identified neighbour to the current node (see $R2.1.1$–$R2.1.3$).

(2) If the state of the current cell is 3, 4 or 5, then the neighbour with the "lowest" state will be identified. If the "lowest" state is 0 or 1, it will be proposed to move some processes from the current node to the identified neighbour (see $R2.1.4$–$R2.1.8$).

In the second part of $R2$ ($R2.2.1$), $A2$ will be activated by the mapper, only if all cell states are 0 or 1.

In the following rules, $p(i,j)$ represents the number of processes on the node corresponding to the cell $A2(i,j)$, and p_1 and p_2 have been introduced in Section 3.

$R2.1.1 : A2(i,j) ::= 1 \Longleftrightarrow A2(i,j) = 0 \wedge p(i,j) = 0 \wedge \exists z \in NB\, z \in \{2,3,4,5\}$

$R2.1.2 : A2(i,j) ::= 1 \Longleftrightarrow A2(i,j) = 1 \wedge p(i,j) < p_1 \wedge \exists z \in NB\, z \in \{2,3,4,5\}$

$R2.1.3 : A2(i,j) ::= 2 \Longleftrightarrow A2(i,j) = 1 \wedge p(i,j) = p_1 \wedge \exists z \in NB\, z \in \{3,4,5\}$

$R2.1.4 : A2(i,j) ::= 3 \Longleftrightarrow A2(i,j) = 3 \wedge p(i,j) > 2p_2 \wedge \exists z \in NB\, z \in \{0,1\}$

$$R2.1.5 : A2(i,j) ::= 2 \Longleftrightarrow A2(i,j) = 3 \wedge p(i,j) > p_2 \wedge \exists z \in NB\, z \in \{0,1\}$$

$$R2.1.6 : A2(i,j) ::= 2 \Longleftrightarrow A2(i,j) = 4 \wedge p_1 \leq p(i,j) \leq p_2 \wedge \exists z \in NB\, z \in \{0,1\}$$

$$R2.1.7 : A2(i,j) ::= 3 \Longleftrightarrow A2(i,j) = 5 \wedge p(i,j) > 2p_2 \wedge \exists z \in NB\, z \in \{0,1\}$$

$$R2.1.8 : A2(i,j) ::= 2 \Longleftrightarrow A2(i,j) = 5 \wedge p(i,j) > p_2 \wedge \exists z \in NB\, z \in \{0,1\}$$

$$R2.2.1 : A2(i,j) ::= 1 \Longleftrightarrow A2(i,j) = 1 \wedge p(i,j) \leq p_1 \wedge \exists z \in NB\, z \in \{0\}$$

Now, we explain how to understand the above rules, using $R2.1.5$ as an example. If the state of $A2(i,j)$ is 3, more than p_2 processes are placed on the node and if there exists a neighbour with the state 0 or 1, then the state of $A2(i,j)$ will be changed to 2. That means, some processes on the node represented by $A2(i,j)$ should emigrate to the neighbouring nodes.

5. Simulation environment CABLE

To discuss the applicability of our method, we developed a simulation environment CABLE, which has been implemented in the objective C programming language under the NeXTSTEP graphic user interface. This implementation is independent from hardware architectures. The source code of CABLE has a size of 1.6 Mbytes and can be easily ported to any systems in which NeXTSTEP and an objective C compiler are available. On a NeXT computer with the M68040 CPU and a memory of 20 Mbytes, a typical simulation of a cluster with 100 computing nodes and 1000 processes needs about 5 minutes.

The goal of this simulation shell is to demonstrate the achievable speedup and the usage of this new load balancing strategy, considering realistic aspects of workstation clusters. Therefore, a number of system parameters have been defined, which can be interpreted as the user interface of this simulation shell. The main parameters regarded here are: the performance of computing nodes (e.g., MIPS and MFLOPS), the performance of networks (e.g., transfer rate, frame size and token holding time), the size of clusters (i.e., number of segments and that of nodes per segment), the scheduling algorithm, the measurement and evaluation cycle, the cellular automaton system and its states, the simulation mode (i.e., with load balancing, without load balancing, or both), and the applications. These parameters can be defined individually and modified (or selected) interactively. They characterize the simulated workstation clusters, the simulated applications and the used cellular automaton systems.

The simulation shell CABLE allows to define a virtual configuration of a workstation cluster and to simulate its behaviour under different parameters selected by users. CABLE can be divided into three logical levels: the *User interface*, the *Simulation controller* and the *Simulation kernel*, shown in Fig. 3.

The *User interface* consists mainly of the *Parameters* unit, the *Statistics & Visualization* unit and some specific user tools. The parameter unit is responsible for inputting simulation parameters and commands. For some parameters, CABLE provides users with certain values for selection; e.g., in the current version, we have developed eight cellular automaton systems and users can select one of them for a certain simulation.

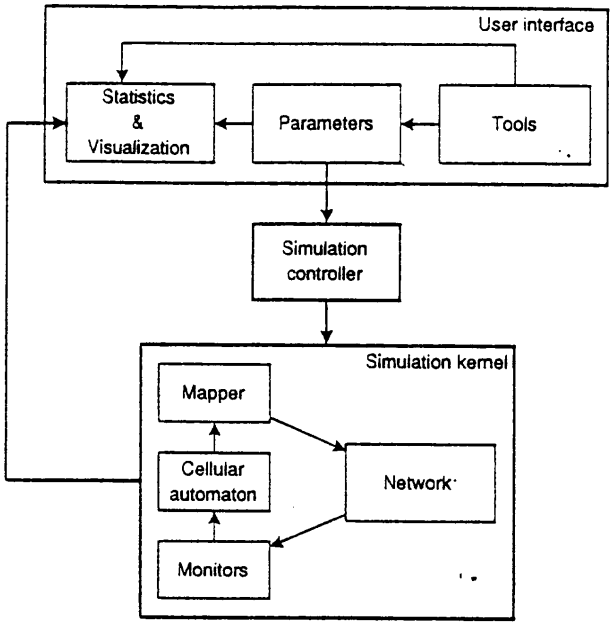

Fig. 3. Logical configuration of CABLE.

However, some other parameters can be defined freely, e.g., users can define any simulation applications by means of a load description language provided by CABLE. Each parameter has a default value which can be changed by users. CABLE provides some specific tools for handling the parameters and visualizing the simulation results.

Based on the parameters from the *User interface*, the *Simulation controller* builds a corresponding simulation environment and controls the execution of the simulation according to a simulation description. A simulation description contains the following information about applications to be simulated:

- the number of applications,
- the start time of each application,
- the number of processes each application has, and
- the contents of each process which consists of computation instructions as well as communications instructions. Computation instructions can be divided into two classes: integer and floating point.

The simulation controller generates the simulation applications at the time specified in the simulation description.

The *Simulation kernel* consists of a virtual workstation cluster (*Network*) and the MCCU (cf. Fig. 1). The behaviour of the cellular automaton can be visualized by means of user tools. The *Mapper* collects simulation results, from which the *User interface* can create different statistics and visualize them. CABLE is able to analyse and compare the performance of a certain virtual workstation cluster using a cellular automaton system with the performance of the same cluster under the same conditions, e.g., the same applications, but without using any cellular automaton system (see Section 6). Fig. 4 shows the main control flow of the *Simulation kernel*, which will be started by the simulation controller, after the simulation controller has finished the simulation preparation, i.e., building the necessary simulation environment. In principle, the kernel

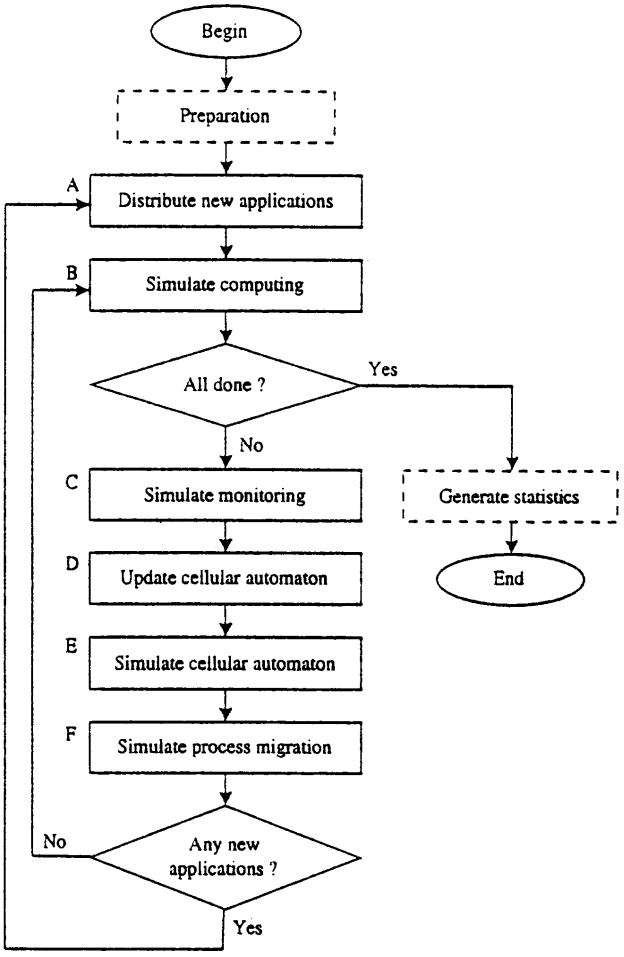

Fig. 4. Main control flow of the simulation kernel.

is a loop of six simulation steps: A–F. However, some steps (e.g., A and F) are not executed in every loop iteration; this is because new applications will be generated only at the time specified (not in each iteration) and the migration of processes is performed only when it is necessary. The decision which steps should be executed is made by the simulation controller. Moreover, the controller must adjust the execution time of each iteration to a constant which is called *simulation cycle*. The simulation cycle is one of the most significant simulation parameters and can directly influence simulation results.

6. Simulation results

We have developed and tested different types of cellular automaton systems. One type of cellular automata transfers processes from overloaded nodes to underloaded nodes, which is called an active cellular automaton. In contrast to this, in a passive cellular automaton underloaded nodes request processes from overloaded nodes. Complex cellular automaton systems combine active and passive strategies, i.e., hybrid cellular automata. We have also simulated hybrid cellular automata with two rule schemes, namely hybrid-

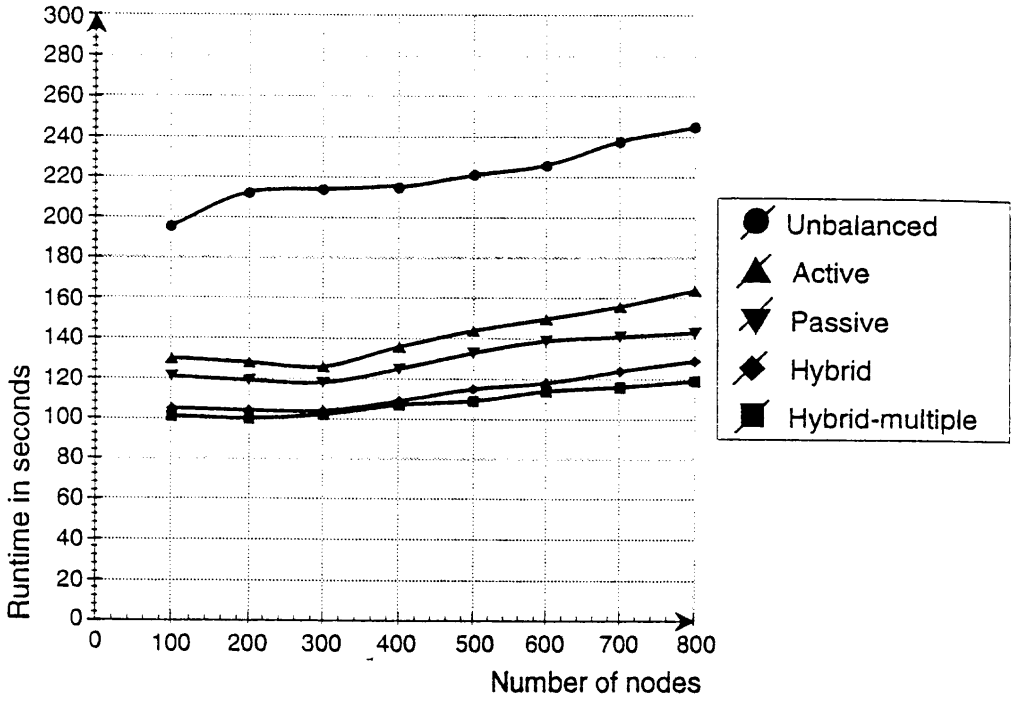

Fig. 5. Speedup comparison among different cellular automaton systems.

multiple ones (see Section 4). Concerning the performance speedup and the scalability of the system, the hybrid-multiple cellular automata have shown the best results in comparison with other automaton types (see Fig. 5). The regarded simulations are based on applications of different communication classes (low, middle or high) and different process sizes (from 15 Kbytes to 150 Kbytes). The size of simulated workstation clusters is scalable, e.g., in Fig. 5 the cluster size ranges from 100 to 800 nodes. In order to compare the speedup of different cellular automaton systems under the same condition, in Fig. 5 the simulated processes are statically distributed at the beginning of each simulation according to the same sequence of random numbers. Moreover, the number of processes in different simulations is proportional to the size of the corresponding workstation cluster, i.e., 800 processes are simulated on a workstation cluster with 100 nodes, while simulations for a workstation cluster with 800 nodes need 6400 processes. In this way, all of the regarded simulations have the same initial system load.

Fig. 6 shows another speedup comparison from the viewpoint of the same number of processes on different sizes of workstation clusters. At the beginning of each simulation, 1800 processes are distributed randomly on the corresponding cluster. Here the simulations are based on a hybrid-multiple cellular automaton. The achieved speedup changes between 2.09 (with 300 nodes) and 1.94 (with 600 nodes), shown in Fig. 6(b). Generally, the speedup can be influenced by two main factors: the basic simulation overhead and the effectiveness of the cellular automaton system. On the one hand, the smaller the cluster, the greater the simulation overhead of each node, because the basic simulation overhead of the system is constant. On the other hand, the greater the cluster, the fewer overloaded nodes in the complete system and the less the cellular automaton system can work effectively. Therefore, in these simulations the best speedup has been achieved

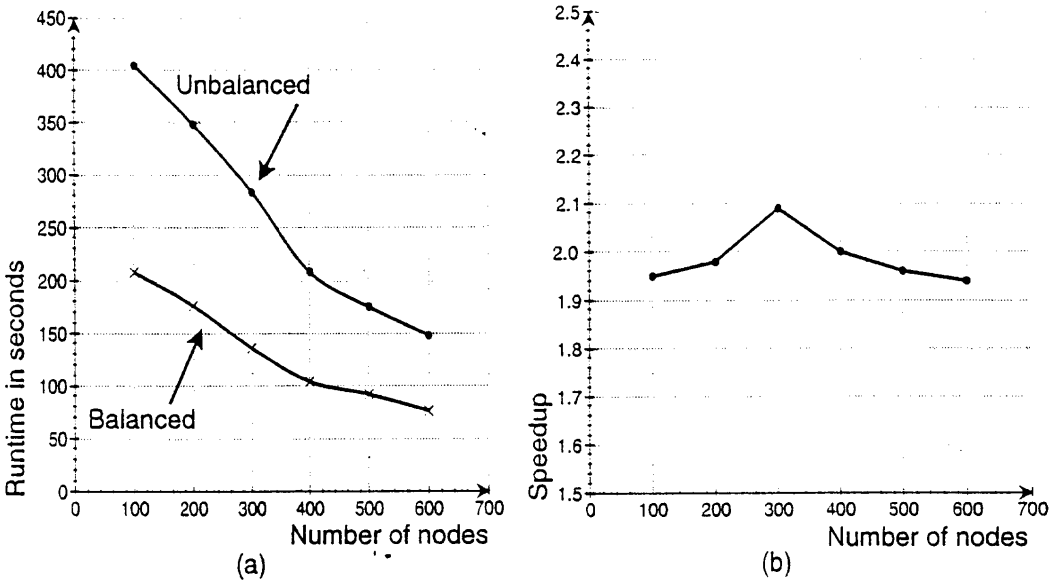

Fig. 6. Speedup comparison between balanced and unbalanced simulations.

with a cluster of 300 nodes. The optimal size of the cluster changes with the simulation parameters.

6.1. Simulation with static distribution of applications

In this section, the result of a specific simulation will be shown. At the beginning, all of three applications with 1200 processes were distributed randomly on a workstation cluster consisting of 200 nodes (10 segments and 20 nodes per segment).

Fig. 7 presents a comparison of an unbalanced simulation (without using any load

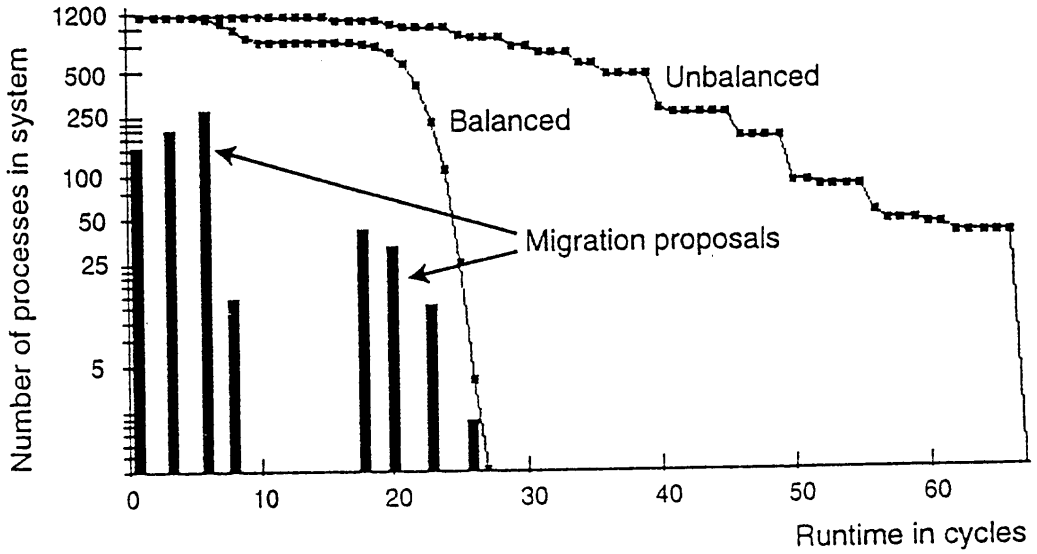

Fig. 7. Runtime comparison of balanced and unbalanced simulations.

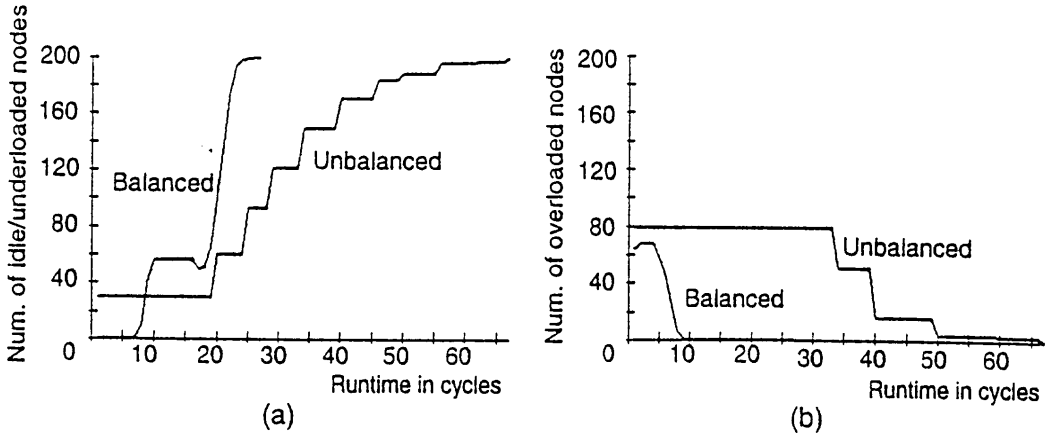

Fig. 8. Comparison of idle/underloaded and overloaded nodes.

balancing strategies) and a balanced one. The histogram in Fig. 7 shows the number of processes proposed by the cellular automaton $A2$ to migrate (see Section 4).

Because of the random distribution of new applications, the system is unbalanced at the beginning of the simulation. Therefore, in the case of the balanced simulation, $A2$ produces migration proposals to balance the system load. After the system load is balanced according to the proposals, $A2$ will be rarely activated. From the 17th cycle, a lot of processes terminate and the system load becomes unbalanced. Therefore, $A2$ will be activated again. In this example, the balanced simulation needs only 27 cycles, while the unbalanced simulation needs 69 cycles. That means, a speedup of 2.56 has been achieved here.

Fig. 8 compares the number of idle/underloaded and overloaded nodes in the unbalanced and balanced simulations. Fig. 8(a) shows that almost all nodes are used at the beginning of the balanced simulation. At about the 8th cycle, the termination of some processes results in approximately 60 nodes becoming idle/underloaded. In contrast to this, about 30 nodes are already idle/underloaded from the beginning of the unbalanced simulation. Fig. 8(b) shows that the system will be balanced after the 7th cycle. In the unbalanced simulation, approximately 80 nodes are overloaded until the first half of the simulation and there are always some overloaded nodes in the system.

6.2. *Simulation with dynamic distribution of applications*

Generally, new applications arrive randomly and independently. This can be considered as a Markov process. Therefore, it has to be possible to distribute new applications dynamically during the simulation time. Furthermore, the applicability and the efficiency in the case of dynamic application distribution are two of the most important criteria against which a method of dynamic load balancing should be evaluated.

For this purpose, we have defined a load description language to specify the simulation parameters of applications, such as the size, the type and the start time. The following is the result of a specific simulation in which five applications are started at the 0th, 5th, 10th, 15th and 26th cycle respectively, on the same virtual workstation cluster as that in the last section. The second application which is started at the 5th cycle has 400

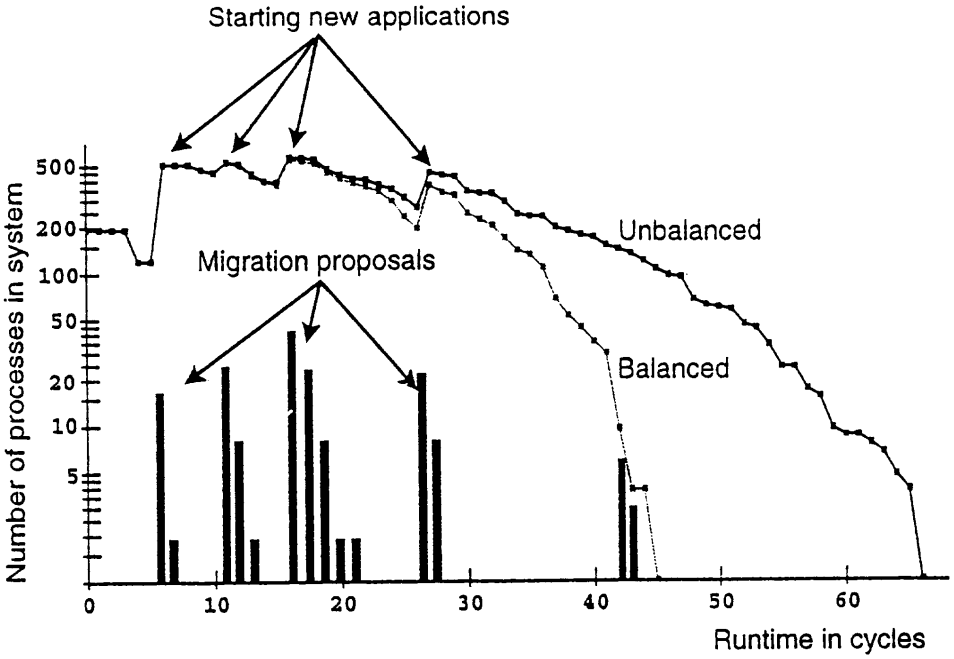

Fig. 9. Runtime comparison of balanced and unbalanced simulations.

processes, and each of the other four applications has 200 processes.

Similarly to Fig. 7, the runtime of the unbalanced simulation and the balanced one is compared in Fig. 9. Because only 200 processes are distributed at the beginning of the simulation, there are almost no overloaded nodes at this moment and, therefore, the cellular automaton makes no process migration proposal. After starting the second application at the 5th cycle, a few nodes become overloaded and the automaton becomes effective. This effect can be observed more clearly after the 10th and 15th cycle when the third and forth applications are started. Between the 28th and 42nd cycle, because no more application is started and the system load is well balanced, the automaton will be rarely activated. However, from the 42nd cycle, a lot of processes terminate and the system load becomes relatively unbalanced again. In this case, the second rule scheme of the hybrid-multiple automaton begins to work.

In this simulation, the speedup of the first four applications can not be seen in Fig. 9 directly. But the last application started at the 26th cycle finishes at the 45th cycle with load balancing and at the 66th cycle without load balancing respectively, i.e., a speedup of 2.1 has been achieved.

Fig. 10 compares the number of normally loaded and overloaded nodes in the unbalanced and balanced simulations. As explained above, during the first 5 cycles, there are only 200 processes in the system consisting of 200 nodes. Therefore, the most of nodes are underloaded or idle; in other words, there are only a few normally loaded nodes and hardly any overloaded ones. In Fig. 10(a), it can be observed that the number of normally loaded nodes is much higher during the main phase of the balanced simulation than in the unbalanced one. From the 28th cycle, the balanced curve goes monotonously and sharply down and ends at the 45th cycle, while the unbalanced one fluctuates with a slowly falling tendency and ends at the 66th cycle. The reason is that,

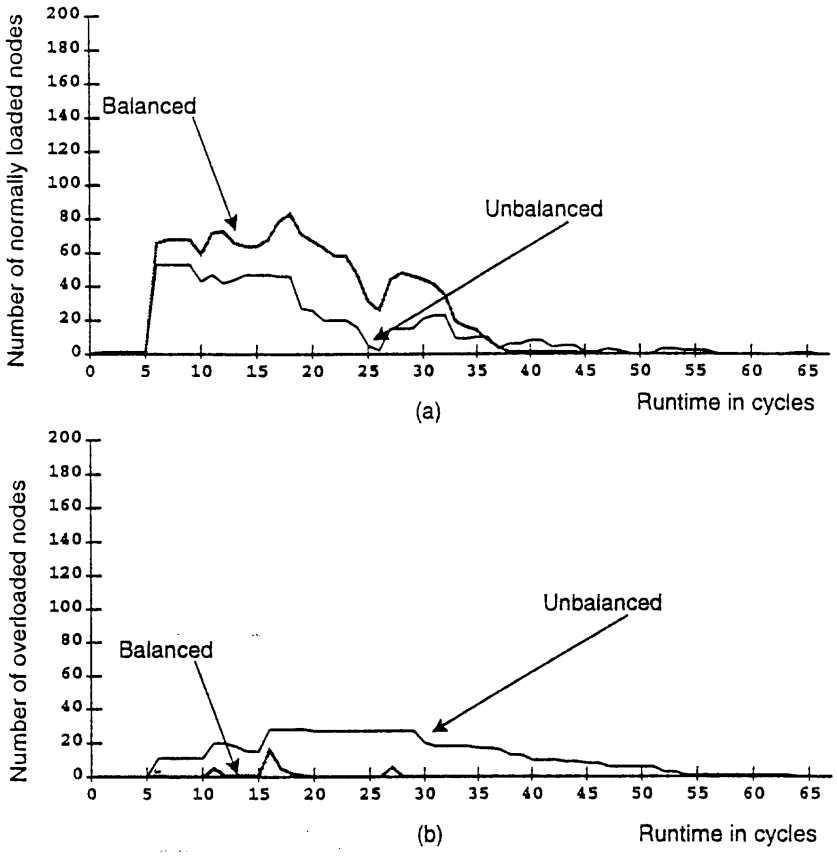

Fig. 10. Comparison of normally loaded and overloaded nodes.

from the 28th cycle, the termination of processes changes some nodes from overload to normal load and results, sometimes, in increasing the number of normally loaded nodes in the complete system.

Fig. 10(b) shows that, only when new applications are started at the 5th, 10th, 15th and 26th cycle respectively, there are some overloaded nodes in the balanced simulation and the automaton can balance the load in the system in a few cycles. In contrast to this, in the unbalanced simulation there exist overloaded nodes during the most of the simulation time; especially from the 16th to 28th cycle, about 30 nodes are overloaded.

7. Conclusions and extensions

In this paper we have presented a new idea for dynamic load balancing. Our approach is based on the application of "intelligent tables" which are realized by cellular automaton systems [1]. Therefore, we developed different simulation shells: CASS [6] is a specific simulation shell to develop complex cellular automaton systems and CABLE simulates the behaviour of a workstation cluster with respect to dynamic load balancing by cellular automaton systems.

Our strategy treats the load balancing task as a hybrid system. Calculation, storage and evaluation of the load states are performed centrally (by the mapper) and the

real process migration is done decentrally (by nodes). Such a combination can gain the advantages of centralized and decentralized strategies with the goal of minimizing communication overhead in the whole network. Our simulation results show that load balancing strategies based on the theory of cellular automata can increase the system performance significantly (a speedup from 1.75 to 2.70). This speedup is better than that of some related approaches [7,11].

Kunz [7] provide an implementation of a task scheduler based on the concept of a stochastic learning automaton on a network of five Sun workstations. About 900 artificial executable tasks were created and executed on the five workstations to evaluate the performance of his task scheduler with different workload descriptions (e.g., number of tasks in the run queue, size of the free available memory, rate of CPU context switches, rate of system calls, etc.). The workload descriptions characterize the load at each host and determine whether a newly created task should be executed locally or remotely. Without trying to balance the load, the mean response time of the whole system was 31.215 seconds. By using the number of tasks in the run queue as workload descriptor, the mean response time was shortened to 13.576 seconds, i.e., a speedup of 2.3. The speedup of the experiments in [7] ranges from 1.74 to 2.3.

In [11], Willebeek-LeMair and Reeves present five different strategies: SID, RID, HBM, GM and DEM. The *Sender (Receiver) Initiated Diffusion* (SID/RID) strategies are asynchronous schemes which only use near-neighbour information. The *Hierarchical Balancing Method* (HBM) organizes the system into a hierarchy of subsystems within which balancing is performed independently and ascends from the lowest level to the highest level. The *Gradient Model* (GM) employs a gradient map of the proximities of underloaded processors in the system to guide the migration of tasks between overloaded and underloaded processors. Finally, the *Dimension Exchange Method* (DEM) requires a synchronization phase prior to load balancing and then balances iteratively. All five approaches have been implemented on an Intel iPSC/2 hypercube. Experiment results for artificial loads on a 32 node iPSC/2 with a granularity of 100 tasks per node have shown a speedup from 1.28 to 1.52. The RID strategy is considered the best one among them.

Moreover, the main advantages of our strategy are the scalability, extensibility and portability, because it is independent from any hardware architecture. The simulation is just based on a virtual hardware configuration which is characterized by the performance of the network and that of the computing nodes. However, the disadvantage of a centralized load balancing strategy is the overhead to collect load state information and to perform process migration. Therefore, a high performance network, e.g., FDDI, whose time characteristics are strongly predictable, is necessary. In our system architecture (see Section 2), due to FDDI networks the computation effort needed by the monitoring unit for calculating the communication and process load state is so little that it can be ignored. The main computation effort is needed by sending migration proposals as well as by the real migration of processes. To overcome this disadvantage, we have extended our approach [2], in which a distributed cellular automaton system is defined. Regarding massively parallel systems, a distributed cellular automaton system can be realized by implementing the individual cells directly on the corresponding nodes. The neighbourhood can be determined by the actual network configuration.

Besides this extension, we just intend to implement our approach on a real network by means of standard networking tools, e.g., *etherfind* and *PVM*. Therefore, we will examine several different computing platforms for possible adaptations of our load balancing strategy. Furthermore, we are extending our simulation shell CASS so that it can be integrated with CABLE.

Because there is no trade-off between the hardware costs and the speedup factor, we are preparing to realize our method by a specific hardware chip in order to exploit the speedup potential. This work will be carried out in cooperation with the GMD (the German National Research Center for Computer Science).

References

[1] D. Beerbohm, S. Bresgen, R. Hofestädt and X. Huang, Dynamic load balancing based on the theory of cellular automata, in: L. Dekker, W. Smit and J.C. Zuidervaart, eds., *Proceedings of International Conference on Massively Parallel Processing*, Delft, The Netherlands (North-Holland, Amsterdam, 1994) 235–242.

[2] D. Beerbohm, S. Bresgen, R. Hofestädt and X. Huang, Automatengesteuertes skalierbares Verfahren zum Lastausgleich, in: R. Flieger and R. Grebe, eds., *Proceedings of 6th Transputer-Anwender-Treffen*, Aachen, Germany (IOS, Amsterdam, 1994) 100–108.

[3] F. Bonomi and A. Kumar, Adaptive optimal load balancing in a nonhomogeneous multiserver system with a central job scheduler, *IEEE Trans. Comput.* **39** (10) (1990) 1232–1250.

[4] C.H. Cap, Massive parallelism with workstation clusters — Challenge or nonsense, in: *Proceedings of the High Performance Computing and Networking Conference HPCN94 Europe*, München, Germany, Lecture Notes in Computer Science, Vol. 797 (Springer, Berlin, 1994) 42–52.

[5] D.L. Eager, E.D. Lazowska and J. Zahorjan, Adaptive load sharing in homogeneous distributed systems, *IEEE Trans. Software Engineering* **12** (5) (1986) 662–675.

[6] R. Hofestädt and U. Hörkens, CASS: A programming environment for the simulation of nondeterministic cellular automata, *SAMS* **14** (1) (1994) 1–14.

[7] T. Kunz, The influence of different workload descriptions on a heuristic load balancing scheme, *IEEE Trans. Software Engineering* **17** (7) (1991) 725–730.

[8] H. Nakanishi, V. Rego and V. Sunderam, Superconcurrent simulation of polymer chains on heterogeneous networks, in: *Proceedings of Supercomputer'92*, Minneapolis (IEEE Press, New York, 1992) 561–569.

[9] N.G. Shivaratri, P. Krueger and M. Singhal, Load distributing for locally distributed systems, *IEEE Comput.* **25** (12) (1992) 33–44.

[10] R. Vollmar, *Algorithmen in Zellularautomaten* (Teubner, Stuttgart, 1979).

[11] M.H. Willebeek-LeMair and A.P. Reeves, Strategies for dynamic load balancing on highly parallel computers, *IEEE Trans. Parallel and Distributed Systems* **4** (9) (1993) 979–993.