PAPER ACCEPTED FOR THE JOURNAL OF PARALLEL ALGORITHMS AND APPLICATIONS, SPECIAL ISSUE ON ALGORITHMS FOR ENHANCED MESH ARCHITECTURES

The Liquid Model Load Balancing Method

Dominik Henrich

Institute for Real-Time Computer Systems and Robotics
University of Karlsruhe, D-76128 Karlsruhe, Germany
e-mail: dhenrich@ira.uka.de

Abstract

Load balancing is one of the central problems that have to be solved in parallel computation. Here, the problem of distributed, dynamic load balancing for massive parallelism is addressed.

A new local method, which realizes a physical analogy to equilibrating liquids in multi-dimensional tori or hypercubes, is presented. It is especially suited for communication mechanisms with low set-up to transfer ratio occurring in tightly-coupled or SIMD systems. By successive shifting single load elements to the direct neighbors, the load is automatically transferred to lightly loaded processors.

Compared to former methods, the proposed Liquid model has two main advantages. First, the task of load sharing is combined with the task of load balancing, where the former has priority. This property is valuable in many applications and important for highly dynamic load distribution. Second, the Liquid model has high efficiency. Asymptotically, it needs $O(D \cdot K \cdot L_{diff})$ *load transfers to reach the balanced state in a D-dimensional torus with K processors per dimension and a maximum initial load difference of* L_{diff}*. The Liquid model clearly outperforms an earlier load balancing approach, the nearest-neighbor-averaging.*

Besides a survey of related research, analytical results within a formal framework are derived. These results are validated by worst-case simulations in one- and two-dimensional tori with up to two thousand processors.

Keywords: distributed computing, parallel algorithms, load sharing, load balancing, rings, tori, hypercubes

1. Introduction

Load balancing is one of the central problems that has to be solved in parallel computation. Load imbalance leads directly to processor idle times and to low exploitation of the potential power of distributed computing. High efficiency can only be achieved if many processors are supplied with work and the computational load is evenly balanced among the processors. This problem can be divided in two distinct tasks, load balancing and load sharing. The easier task of *load sharing* is to supply each processor with at least some load. Thereby, the amount of processor load is of no interest as long as there is load at all. *Load balancing* is the task of equilibrating the load as evenly as possible. As a final goal, every processor should have the same amount of work.[1]

For both of these tasks, many approaches have already been studied. A general taxonomy of load balancing approaches is given in [Casavant88]. Because the number of processors in available parallel computing systems increases quickly, scalable algorithms are required. In centralized load balancing algorithms the scheduler forms an bottleneck. Thus, we concentrate on distributed approaches. Within this category of distributed algorithms, we additionally distinguish between global and local methods. With *global*[2] load balancing one processor may transfer load (or load information) to any other processor in the system. This transfer of the load packages is done by sending them through a routing network. Several global distributed approaches with asynchronous communication are described, for example, in [Kumar91, Schabernack92].

The general drawback of global approaches to load balancing is that, with an increasing number of processors, the global communication of the load will slow down the algorithm. Thus, in the future, only local approaches seem applicable for massively parallel computers. In this paper, we concentrate on *local* distributed load balancing and load sharing methods, where only communication between a processor and its directly connected neighbors is allowed. Thus, the communication via several processors is not admissible.

For the load balancing process, we assume that the total work consists of single *load elements* that represent tasks to be processed. Because in many applications the size of the tasks is not known in advance, all the load elements are assumed to be of the same size. Thus, as a time measure in the load balancing algorithms, the transfer of one load element is appropriate and any compression of load is excluded. The number of communication set-ups transferring several load elements is not sufficient as time measure for tightly-coupled or SIMD systems, because the set-up is fast compared to the communication itself.

For the application process, we assume that the amount of load is increased and reduced dynamically in time by generating new tasks and finishing existing ones. Additionally, the development of load increase and reduction is not predictable, thus, future load distribution

1 The distinction between load sharing and load balancing may seem rather artificial because balanced load is also shared load, but it leads to a better insight of the algorithms and for some applications load sharing is sufficient.

2 This interpretation differs from the one in [Casavant88]. There, *local* load balancing addresses the scheduling and dispatching of processes within a single processor, and *global* load balancing addresses the load distribution over the total multiprocessor system.

cannot be foreseen. Such highly dynamic load distributions are given, for example, in search tree algorithms such as Branch-and-bound, A*, IDA*, etc.

In summary, the application algorithm with dynamic load balancing can be viewed as two interlaced, adversary processes, where the balanced state is unlikely to be reached. As consequence, an explicit termination criterion of the load balancing process can be omitted. Another consequence of dynamic load distribution is that aged load information is nearly worthless. The older the information the more likely has the load configuration changed in the meantime.

Here, we present and evaluate a new local load balancing approach. Former local load balancing approaches are reviewed and discussed in Section 2. Then, the new method is illustrated and a formal framework is set up in Section 3. Using this framework, the properties and the efficiency of the method are derived analytically in Section 4. These analytic results are supported by several simulation results in Section 5.

2. Related Research

In this section, we survey known local load balancing methods. These methods can be divided into three basic approaches: the diffusion method, the dimension exchange, and the nearest-neighbor-averaging. All three approaches have one property in common, the strict locality of control and communication. Nevertheless, each approach uses a different starting point. Finally, we briefly discuss semi-local approaches, which loosen the strict locality property.

2.1. Diffusion Approach

The diffusion method is an iterative algorithm and especially suited for systems with direct communication networks. In every step, a fixed fraction of the load difference between two neighboring processors is exchanged. When these local operations are used, the load distribution converges to the global optimum. The efficiency of the diffusion method depends on a *diffusion parameter* α which determines the size of the transferred load fraction.

For processor i, let the load $L_i \in \mathbb{R}$ and a set of directly neighboring processors $\Delta(i)$ be given. Thereby it is assumed that the load consists of very many and very small load elements, so that a continuous representation is admissible. By the diffusion method, a load $\delta_i(j)$ is transferred from processor i to every neighbor $j \in \Delta(i)$ with

$$\delta_i(j) = \alpha\,(L_i - L_j), \text{ with } \alpha \in (0,1) \qquad (1)$$

With $\delta_i(j) < 0$, the load is transferred in the inverse direction. In this formulation, truncation errors are not considered. Every change of state of the processor load L_i by synchronous load balancing can be described by the following transition equation:

$$L_i(t+1) \ := \ L_i(t) + \sum_{j \in \Delta(i)} \delta_i(j) \tag{2}$$

For a system with P processors and a total load of L that is distributed unevenly over the system, the processor load has to converge to L/P by the diffusion method. In [Cybenko89], this method is analyzed for the first time. Assuming a synchronous communication, necessary and sufficient conditions for the diffusion parameter are given to ensure convergence. Additionally, the optimal parameter for hypercubes is found, which enables the highest convergence rate of load balancing. In [Bertsekas89], the convergence for an asynchronous version of the diffusion method is shown, provided that the communication delay of a link has an upper bound.

Besides the necessary convergence itself, the rate of convergence is important. In [Boillat90], different convergence rates for several network topologies are given. In this analysis, only the number of communication set-ups is considered, but not the amount of transferred data. It is shown that in D-dimensional tori with K_i processing units in dimension i, the load configuration converges asymptotically to a balanced state with $O(D \cdot \max\{K_i\}^2)$ time. In D-dimensional binary hypercubes only $O(D)$ steps are necessary. Additionally, the number of iterations necessary to reach a balanced state depends on the initial load configuration but this fact is not considered by the time measure used. In [Xu93], the optimal diffusion parameter for synchronous load balancing in D-dimensional tori with K processors per dimension is derived.

In [Kumar87] a variant of receiver-initiated diffusion approach named α-splitting is analyzed for unidirectional rings. When an *idle* processor with index $i + 1$ demands processor i for work then the fraction $(1 - \alpha) \cdot L_i$, with $0 < \alpha < 1$, of the total load L_i is transferred to processor $i + 1$. The analysis shows, that for an increasing number P of processing units, an exponential time effort of β^P, with $\beta = 1/(1 - \alpha)$, is necessary asymptotically. The time effort is measured by the number of transfers, independent of the (continuous) amount of information transferred. The discrepancy to the results of the former paragraphs is due to the different initialization of load balancing.

For diffusion methods, various applications are possible. For example, the diffusion was used for branch-and-bound algorithms on Intel iPSC/2 in [Willebeek90] and on Transputer clusters with de-Bruijn and ring topologies in [Lüling92].

The main drawback of diffusion methods comes up in practice. In [Horton93], it is shown that in spite of proven convergence, global imbalance using local balancing operations can arise. This effect is due to the discrete realization of the continuous model. Necessary truncation errors may cause decreasing ramps with slope –1 that are not further equilibrated.

2.2. Dimension Exchange

The dimension exchange is a further local load balancing method. It is a synchronous approach where load balancing takes place successively in a single dimension. See [Willebeek93, Cybenko89, Dragon89]. In [Cybenko89], a dimension exchange for asynchronous multiprocessor systems with hypercube topology is presented, which needs $\log(P)$ steps with P processors. One load transfer includes the communication of multiple load elements. This approach is suited for problems with little dependency between the load elements (see [Fox89]).

Comparing dimension exchange and the diffusion method, each approach is well suited for different communication models. With the diffusion method, simultaneous communication with all direct neighbors is best. For the dimension exchange, one communication at a time is sufficient. In [Cybenko89], it is shown that the dimension exchange outperforms the diffusion method in hypercubes. This holds true for k-ary n-cubes too [Xu93].

2.3. Nearest-neighbor-averaging

The nearest-neighbor-averaging (NNA) is a further, completely local load balancing method. The idea is to change the load of each processor such that it is equal to the mean load of the processor and its neighbors. Regarding the processor i and the set of its direct neighbors $\Delta(i)$, there is a mean load of $\bar{L}_i(t) = (L_i(t) + \sum_{j\in\Delta(i)} L_j(t)) \,/\, (|\Delta(i)| + 1)$ at time t, where $|\Delta(i)|$ stands for the number of neighbors. After one load balancing step, at time $t + 1$, the processor i should have a load of $\bar{L}_i(t)$. As a transition equation for the load change, the following formula holds true:

$$L_i(t+1) := \frac{1}{|\Delta(i)| + 1}\left(L_i(t) + \sum_{j\in\Delta(i)} L_j(t)\right) \tag{3}$$

The NNA can be realized in two different ways. An asynchronous variant is described in [Willebeek93]. When a processor is highly loaded then it transfers a portion of its load to all deficient neighbors. The amount of transferred load is proportional to the difference of the mean load and the load of the neighbor. Let the deficiency of each neighboring processor $j \in \Delta(i)$ for the processor i be given by $h_j = \max\{0, \bar{L}_i - L_j\}$ and the total deficiency by $H_i = \sum_{j\in\Delta(i)} h_j$. Then, the asynchronous NNA performs a load transfer $\delta_i(j)$ from processor i to each of its neighbors $j \in \Delta(i)$ with

$$\delta_i(j) = (\bar{L}_i(t) - L_i)\frac{h_j}{H_i} \tag{4}$$

In the synchronous variant of NNA, the load of every processor is divided into $|\Delta(i)| + 1$ portions of the same size. The processor itself and all of its neighbors receive one load portion. The execution of each load balancing step satisfies the transition equation in (3). Let the processor i have a load L_i and a set of neighbors $\Delta(i)$. Then, by the synchronous NNA, a load transfer δ_i from processor i to each of its neighbors $j \in \Delta(i)$ is performed with[3]

$$\delta_i = \frac{L_i}{|\Delta(i)| + 1} \tag{5}$$

The synchronous NNA is analyzed in [Hong90] for binary hypercubes and in [Qian91] for general hypercubes. There, it is proven that the load balancing method converges and that the variation of the load has an upper bound.

When comparing NNA with the diffusion method in Section 2.1, the NNA can be recognized as a special case of the diffusion method. With a diffusion parameter $\alpha = 1 / (|\Delta(i)| + 1)$ the transition equation of the diffusion method in (2) turns into the one of NNA in (3).

2.4. Semi-local Load Balancing

Besides the strictly local load balancing methods presented above, there are *semi-local* approaches. They have the requirement of locality being more or less loose in common. Nevertheless, these approaches cannot be fully allocated to global load balancing methods, and therefore are only briefly mentioned here.

A hierarchical and topological independent diffusion method for multiprocessor systems is presented in [Horton93]. Local load transfers are controlled based on global load distribution information. On the one hand, neighborhood relations between load elements can be fulfilled. On the other hand, each processor has to be supplied with sufficient load elements to transfer the required amount of load. To reach the equilibrated load state, O(log(P)) transfers with P processors are necessary. Thereby, one transfer includes the communication of several load elements.

The gradient model in [Lin87] is a receiver-initiated, topology-independent load balancing method for multiprocessor systems. A global potential field indicating the proximity of lightly loaded processors is successively approximated. The load packages migrate in the direction of the gradient from highly loaded to lightly loaded processors. Because of the load elements are migrating through several processing units and are not considered as additional load, the gradient model cannot be viewed as a strictly local approach. An application of the gradient model to domains with high, dynamic load changes, e.g., as tree search algorithms, is difficult because the approximation of the potential field ages quickly.

---

[3] The communication effort can be reduced down to the difference between the two opposed load transfers by previous exchange of load information.

2.5 Conclusion

All the presented local load balancing approaches have one disadvantage in common: They assume a continuous amount of processor load. Contrasting with that, for the most applications, a discrete representation of load is more adequate. Additionally, with massively parallel computers, one cannot turn to a continuous representation because the local processor memory is relatively small and, therefore, the number of load elements is limited. A continuous representation simplifies the analysis of the methods but it leads to the load imbalance mentioned in Section 2.1. Therefore, it is important to design load balancing mechanisms, which take this problem into account and, e.g., assume discrete load elements.

Many of the former local load balancing approaches can be reduced to the diffusion approach. Additionally, this is the only approach, which has been analysed to this extend so far. For several topologies, e.g. K-ary D-cubes, the NNA implements the optimal diffusion parameter. Thereby, the time effort grows asymptotically in a quadratic form, for an increasing network diameter. Because NNA represents the best known method analyzed so far, we will use it for comparisons in Section 5.

In the previous investigations of load balancing methods, the number of load transfers, i.e. communication set-ups, has been used as a time measure. For coarse-grained parallel computers, this is an appropriate measure, because the set-up time is huge compared to the transfer time itself. On the other hand, for the fine-grained massively parallel architectures, this ratio is inverse. To determine analytically the time effort on these machines, it is important to consider the amount of transferred information too. This amount is always greater than or equal to the number of transfers, because for communicating one information unit at least one set-up of the link is necessary. Thus, in the rest of the paper, we regard the communicated amount instead of the communication set-ups.

3. The Liquid Model

This section presents a new local load balancing method. The basic idea is illustrated using both a continuous and discrete view point. Then, we develop a formal definition of the model as a basis for our analysis. Finally, an example showing two basic properties is given.

3.1. Illustration

The proposed load balancing method implements a *Liquid model*. For this model, a flat box is filled with a homogeneous liquid. In the balanced state, the liquid has the same height at any place in the box. If one pours additional liquid into the box at an arbitrary location, the liquid equalizes itself such that the height is again the same everywhere. See Figure 1a for a simplified illustration.

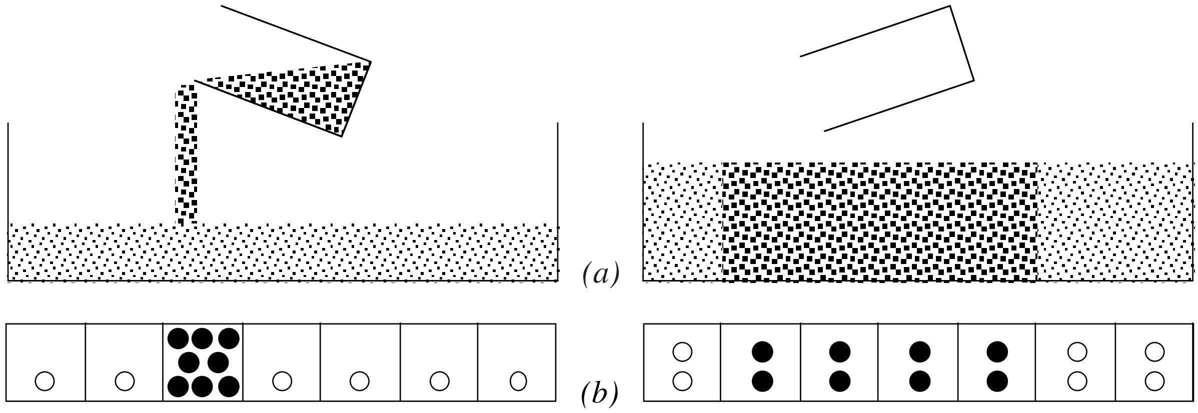


Figure 1: The behaviour of the simplified Liquid model (a) and its discrete equivalent (b) when further liquid resp. load is added

This equilibration happens by locally displacing the superfluous liquid to the neighborhood. By successive displacements, the liquid equalizes itself again. This global effect is achieved by a strictly local mechanism, because none of the additional liquid molecules will "jump" to locations with lacking liquid.

The discrete equivalent to the above continuous model is shown for the one-dimensional case in Figure 1b. This simplified physical effect can be used for load balancing. There, the geometry of the box is discretized and every interval corresponds to one processing unit. The liquid in the liquid model corresponds to the load in the load balancing process. The quantization of the liquid height is represented by elementary load units. For example, in a tree search algorithm, the nodes of the search tree that still have to be processed are the single load units.

If there is a heavy load at some location and a light load at another location, the load should be transferred from the former to the latter place. Global approaches would detect these locations and transfer the load directly by a communication network. Instead, with the Liquid model, the load is transferred implicitly. While there are processors with light loads, heavily loaded processors shift load elements to some of their neighbors and receive load elements from other neighbors. The lightly loaded processors only receive load elements but do not give any away. By successive shifting, the load is automatically transferred to the processors with low load. The approach taken here targets for both aims, load balancing and load sharing. These aims will not be reached in one step, but asymptotically by several load balancing iterations.

3.2. Formal Model

For a precise description of the liquid model load balancing method and for the formulation of analytical statements about the algorithm, the above presented idea is now formalized. With regard to that, we assume a general cyclic mesh structure (torus) as the communication network of the processing units. This structure is a general topology, which is used, e.g., in Paragon or MasPar computers. The main advantages are that it can easily be scaled in the number of processors and that it is easy to implement.

Given is a D-dimensional, symmetric[4] torus with $P = K^D$ processing elements, for a fixed $K \in \mathbb{N}^+$. The processor P_i has a unique identity i, which results from the (cyclic) coordinates of the torus and is given by the D-dimensional vector $i = (i_1, \ldots, i_D)$. The set of all admissible identifiers of the processing elements is given by the index set

$$I := \{(i_1, \ldots, i_D) \mid i_j \in \{0, \ldots, K-1\} \text{ and } j = 1, \ldots, D\} \tag{6}$$

For simplification the indices are taken (mod K), which means P_i stands for $P_{i \bmod K}$. The access to solely one dimension d is enabled by the vector 1_d that consists of a 1 in the d-th position and in all other positions 0.

The processor system is inspected only at discrete points in time $t \in \mathbb{N}$. Thus, we regard a series of successive system states. At every arbitrary point of time $t \in \mathbb{N}$, the load of processor P_i is denoted by $L_i(t) \in \mathbb{N}$. In the rest of the paper, the unique time parameter can be omitted, and the load states refer to the time t, i.e., L_i stands for $L_i(t)$.

The change of load states is accomplished by shifting elementary load units between neighboring processors. The Boolean function $C_{i,d}(t)$ controls the shift of load elements. If for one processing unit i the condition $C_{i,d}(t)$ holds true, then it shifts one load element to its neighbour in dimension d. The function $C_{i,d}(t)$ represents one of the six conditions C0 to C5 in Table 1. It evaluates the selected condition for the processing element i (respectively for its load L_i) at time t. Thereby, the one-dimensional condition is applied in the direction of dimension d. Thus, the scalar operations within the conditions refer only to the d-th component of the D-dimensional vector. For example $C5_{i,d}(t)$ stands for $L_i > 0 \wedge L_{(i_1,\ldots,i_D)} \geq L_{(i_1,\ldots,i_d+1,\ldots,i_D)}$.

The conditions C0 and C1 use load information L_i of only the processor i. Condition C1 and its extension in C2 guarantee that none of the busy processors become idle due to shifting load elements. Conditions C2 through C5 additionally use load information of the preceding neighbor L_{i-1_d} or succeeding neighbor L_{i+1_d}. By each of these conditions an extra mechanism for load balancing in the frame of the Liquid model is implemented. Because each processing unit uses the same mechanism, the rules C0 to C5 are never working together.

As one load balancing step of the Liquid model, the change of state from time t to time $t+1$ is regarded. Such a load balancing step consists of several substeps. Therefore, the load state L_i of all processors $i \in I$ is changed synchronously in every dimension $d = 1, \ldots, D$. Instead of viewing the explicit transfer of the load, here, we regard the resulting effect. The state change of load depends only on the direct neighbors within one dimension. With that, a definition of load balancing relying only on these shift conditions can be established.

---

[4] A symmetric torus consists of an identical amount of elements (processors) in every dimension. This symmetry is no precondition of the Liquid Model, but it simplifies the description.

	$C_{i,d}$ $\Leftrightarrow$	"Processor P_i shifts one load element to P_{i+1_d}" $\Leftrightarrow$
C0:	$L_i > 0$	"P_i has load elements to be transferred"
C1:	$L_i > 1$	"P_i is not idle after giving away one load element"
C2:	$C1 \vee [(L_i = 1) \wedge (L_{i-1_d} > 1)]$	"P_i is not idle after giving away one load element or receives one load element from P_{i-1_d}"
C3:	$C1 \wedge L_i \geq L_{i+1_d}$	"P_i is not idle after giving away one load element and P_{i+1_d} has not more load"
C4:	$C2 \wedge L_i \geq L_{i+1_d}$	"P_i is not idle after giving away one load element or receives one load element from P_{i-1_d} and P_{i+1_d} has not more load"
C5:	$C0 \wedge L_i \geq L_{i+1_d}$	"P_i has load elements to be transferred and P_{i+1_d} has not more load"

Table 1: Formal and verbal description of different instances (C0 to C5) of shift condition $C_{i,d}$, which indicates dependent on the load L_i whether processor P_i should shift one load element to its neighbour in dimension d

Definition 1: (Liquid model)
A change of state $L_i(t) \rightarrow L_i(t+1)$, with $i \in I$, is called Liquid model load balancing (LM-C, *with condition C in Table 1), if and only if in every dimension d = 1, …, D the following equation is applied successively:*

$$L_i(t+1) := \begin{cases} L_i(t) + 1, & \text{if } C_{i\text{-}1_d,d}(t) \wedge \neg C_{i,d}(t) \\ L_i(t) - 1, & \text{if } \neg C_{i\text{-}1_d,d}(t) \wedge C_{i,d}(t) \\ L_i(t), & \text{otherwise} \end{cases} \qquad \Diamond$$

In Table 2, the mechanism of load transfers depending on the shift condition is illustrated. For each possible evaluation of the shift condition of two neighbouring processors, the resulting load transfers are given. The shift of load elements results by itself in a change of load. This change of load serves in Definition 1 as basis to formulate the Liquid model load balancing method.

By the above definition, the elementary step of iterative load balancing following the Liquid model is stated. Now, the question for the aim and the termination of the load balancing arises. The goal of every load distribution method, when aiming to balance the load configuration, is to obtain the balanced state in as few as possible state changes $t \rightarrow t+1$ starting from any arbitrary load distribution L_i. In the final balanced state, every processor load should have reached about the mean load of the total system. This balanced load configuration can be defined by the maximum difference between two processors as follows.

Processor:	P_{i-1_d}		P_i	
Shift condition:	$C_{i,d}(t)$ = False		$C_{i,d}(t)$ = False	
Shift of load:		–		–
State change:			$L_i(t+1) := L_i(t)$	
Shift condition:	$C_{i,d}(t)$ = False		$C_{i,d}(t)$ = True	
Shift of load:		–		• →
State change:			$L_i(t+1) := L_i(t) - 1$	
Shift condition:	$C_{i,d}(t)$ = True		$C_{i,d}(t)$ = False	
Shift of load:		• →		–
State change:			$L_i(t+1) := L_i(t) + 1$	
Shift condition:	$C_{i,d}(t)$ = True		$C_{i,d}(t)$ = True	
Shift of load:		• →		• →
State change:			$L_i(t+1) := L_i(t)$	

Table 2: Load balancing following the Liquid model for processor P_i with its predecessor P_{i-1_d}. The shift of load elements depends on the Boolean evaluation of the shift condition $C_{i,d}$, which indicates by itself the load change of processor P_i.

Definition 2: (Balancing)
A load configuration L_i, with $i \in I$, in a D-dimensional torus is called balanced*, if and only if for all $i \in I$ the following condition holds true:*

$$|L_i - L_j| \leq D, \text{ for all } i, j \in I \qquad \Diamond$$

The above definition serves as a formal termination criterion of the load balancing method, which is necessary for the analysis in Section 4. As long as the condition for a balanced load configuration is not fulfilled, a successive application of single balancing steps is necessary. As a load shifting operation, a procedure Transfer_Load(P_i, P_{i+1_d}) is assumed, which transfers one load unit from processor P_i to processor P_{i+1_d}. With that, we obtain the following imperative formulation of the load balancing method of Definition 1.

Algorithm 1: (Liquid model load balancing)

> **while** ($|L_i - \overline{L}| > 1$) **do**
>   **for** $d = 1, \ldots, D$ **do**
>   **for** all processors P_i, $i \in I$, **do in parallel**
>   **if** $C_{i,d}$ **then**
>   Transfer_Load(P_i, P_{i+1_d});
>   /* $L_i := L_i - 1$ and $L_{i+1_d} := L_{i+1_d} + 1$ */
>   **end**;
>   **end**;
> **end;**

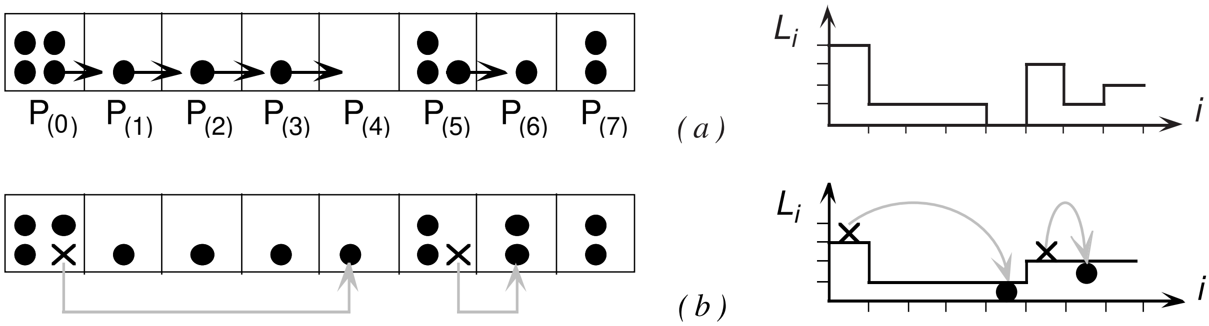


Figure 2: Example of a single load balancing step by the Liquid model with shift condition C5 in a ring of eight processors $P_{(1)}$ through $P_{(8)}$ (black arrows: real load transfer, grey arrows: "virtually" transferred load). The load configurations are depicted as processors with load elements on the left and as function graph of the load L_i depending on the processor P_i on the right.

When integrating the Liquid model load balancing method in a given application algorithm, the formal termination criterion is not checked. To do so, global information about the system state had to be computed, which reduces the scalability of the algorithm. Without termination criterion, the load balancing mechanism will not terminate by itself. On the other hand, this property is not necessary, because the application algorithm and the load balancing proceed concurrently. Thus, the termination of the load balancing mechanism is guaranteed by the application.

3.3. Example

An example for one single Liquid model step in a ring topology is given in Figure 2. Thereby, every processing unit shifts one load element to its neighbor on the right iff condition C5 holds, i.e., if a processor has load elements and its load is greater than or equal to the load of its neighbour. Additionally, the processors shift the load elements in the same direction that is given by the indices (here, to the right). In (a), the initial configuration is shown and the future shifts are indicated by arrows. In (b), the resulting configuration after the shift operation is given. the arrows indicate the "virtually" transferred load elements. (A formal definition of the virtual load transfer is given in Section 4.)

Two effects of the Liquid model can be seen from this example. In the left processor group, $P_{(0)}$ through $P_{(4)}$, a global transfer by local shifts is performed. Additionally, in the right processor group, $P_{(5)}$ through $P_{(7)}$, the load is balanced.

There are two main differences between the Liquid model and the former load balancing methods in Section 2. First, with the former methods, none of these global effects are possible by shifting load elements locally. This is because if the load of three neighbors is already balanced, then no load elements will pass this triplet in one step. Therefore, this balanced triplet forms a burden for a (virtual) load transfer. Second, the former methods achieve load sharing only as a side effect of balancing the load. This contrasts with the Liquid model, where load is shared among the processors in the first place, and after that, load balancing takes place. We will investigate the second effect in greater detail in the following section.

4. Analysis

In this section, we use the framework of the last section to achieve three basic statements about the Liquid model. The statements refer to global effects by local operations, to the convergence to an equilibrated load distribution, and to the high efficiency of the algorithm. In the following, both statements are derived formally only for condition C5, because it results in the most efficient variant (see Section 5). The other conditions C3 through C4 can be treated in the similar way.

One representative of global effects has already been illustrated for LM-C5 in Figure 2.[5] The virtual load transfers occur in the shift direction (here, to the right). The precondition for that effect is a series of processors that have exactly one elementary load, some processors to the right being idle. Another general representative of global effects is the virtual load transfer in the inverse shift direction. Thereby, the load of some processor is reduced by one element and of another processor, with a smaller index, increased. The load of the intermediate processors remains unchanged. The latter effect is explained more precisely in the following definition (see also examples in Figure 3).

Definition 3: (Virtual load transfer)
Given are two processors P_x and P_y, with $x, y \in I$ and within a dimension d, i.e., there exists a $k \in \mathrm{N}^+$ with $y = x + k \cdot 1_d$. A change of a given load configuration $L_i(t)$, with $i \in I$, is called a virtual load transfer *between processors P_x and P_y, if $L_x(t+1) = L_x(t) + 1$ and $L_y(t+1) = L_y(t) - 1$ under the condition that, for all $i = x+1_d, \ldots, y-1_d$ $L_i(t+1) = L_i(t)$ holds.* ◊

Besides global effects by the load balancing method, global structures can be observed in the load configuration itself. Because the load balancing method considers multiple dimensions one by one, only one-dimensional structures are of interest. A typical structure is given by the strongly monotone ascent of the processor load in the shift direction. Such load ascents with maximum length are called ramps in the following definition (see also examples in Figure 3).

5 An detailed example is given in Table 3.

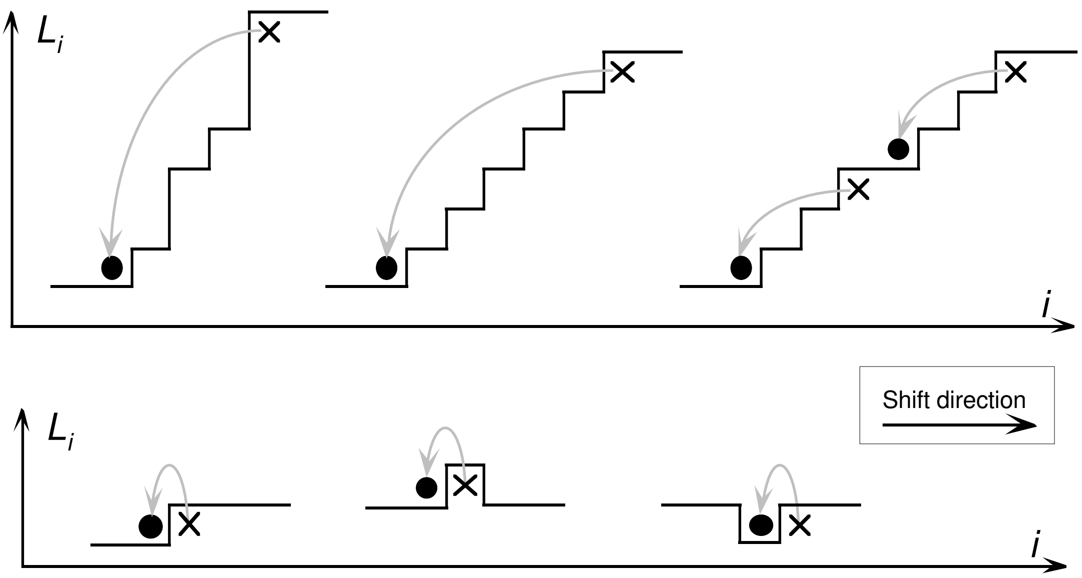


Figure 3: Different shapes of ramps in one-dimensional load configurations and the resulting virtual load transfers by LM-C5 in a one-dimensional ring.

Definition 4: (Ramps)
The load configuration between two processors P_x and P_y, with $x, y \in I$ and within dimension d, forms a ramp *if the following four conditions hold true:*

(1) *there exists a $k \in \mathrm{N}^+$ with $y = x + k \cdot 1_d$ and*
(2) *for all $i = x, x+1_d, \ldots, y-1_d$ holds $L_i < L_{i+1_d}$ and*
(3) *$L_{x-1_d} \geq L_x$ and*
(4) *$L_{y+1_d} \leq L_y$* ◊

In Figure 3, a series of examples for ramps in one-dimensional load configurations is depicted. There, processor load L_i is plotted against the processor index i. The shift direction is to the right. In the upper row, the slope of the ramps decreases from the left to the right example. In the lower row, special cases with smallest-possible ramps in a load plateau are shown (left: jump, mid: maximum, right: minimum). Additional arrows in the figure indicate the virtual load transfers. The relationship between ramps and virtual load transfers is stated in the following theorem.[6]

[6] In this and the following theorems, a load configuration is assumed, where all processors have at least one load element ($L_i > 0$, with $i \in I$), i.e., the goal of load sharing has already been reached. This assumption is not critical, because all processors are supplied with load elements by the Liquid model in a very short time (in $O(D \cdot K)$ steps).

Theorem 1: (Global effects)
Let $L_i(t) > 0$, with $i \in I$. A virtual load transfer between processors P_x and P_y, with $x, y \in I$, can be observed after one load balancing step by LM-C5 if the load configuration between these two processors forms a ramp.

Proof: The condition (1) of Definition 4 guarantees the order of the processors P_x and P_y, which is required by Definition 3. From conditions (2) and (3) and Definition 1 follows that P_x receives an additional load element from P_{x-1_d} but does not shift any to P_{x+1_d}, i.e., after one load balancing step, processor P_x has one more element and $L_x(t+1) = L_x(t) + 1$ holds. From conditions (4) and (2), it follows analogously that P_y shifts one load element to P_{y+1_d} but does not receive any from P_{y-1_d}, i.e., after one load balancing step, processor P_y has one element less and $L_y(t+1) = L_y(t) - 1$ holds. For all processors P_j with $j = x+1_d, \ldots, y-1_d$, it follows from condition (2) with $i = j$, that P_j does not give away any load, and from condition (2) with $i = j-1_d$, that P_j does not receive any load. With that, the load state of processor P_j is unchanged and $L_j(t+1) = L_j(t)$ holds after one load balancing step. Now all the requirements of Definition 3 are fulfilled. ◊

A basic precondition for the efficiency of the load balancing method is that the quality of the load configuration does not worsen by applying load balancing. With LM-C5, this conservative behavior can be guaranteed. The reason is that the maximum or the minimum of the load is not increased or decreased, respectively. The following lemma proves this statement.

Lemma 1: (Conservativity)
After the application of one load balancing step by LM-C5 to a load configuration $L_i(t)$, with $i \in I$ and $L_{max}(t) = max\{L_i(t) \mid i \in I\}$ with $L_{min}(t) = min\{L_i(t) \mid i \in I\}$ respectively, it holds:

$$L_{max}(t+1) \le L_{max}(t) \quad \text{and} \quad L_{min}(t+1) \ge L_{min}(t).$$

Proof. Initially, we consider only a partial step in dimension d. By the last two cases of the state transition by LM in Definition 1, the load state of a processor cannot be increased. The first state change $L_i(t+1) := L_i(t) + 1$ is performed if $C_{i,d}(t) \wedge \neg C_{i+1_d,d}(t)$ holds true. With C5 as a condition, this is equivalent to $L_{i-1_d}(t) \ge L_i(t) \wedge L_i(t) < L_{i+1_d}(t)$. Thus, processor P_i has no more load than its neighbor P_{i+1_d} after one partial load balance step in dimension d. With minimal load the argumentation is analogous. Because the partial steps in all dimensions are executed successively, their combination to one full load balancing step does not increase or decrease any maxima or minima, respectively. ◊

Besides the conservativity referring to the load configuration, the load balancing method additionally has to improve the distribution of the load. For iterative methods, this requires an improvement within a fixed number of iterations. Otherwise, the efficiency cannot be guaranteed. For LM-C5, this improvement can be observed at all left maximum positions or right minimum positions in load plateaux or valleys, respectively. At these positions, the behavior is symmetrical, thus we regard only a left, global maximum position. The application of a load balancing step can have two alternative consequences. Either the value in the maximum position is reduced (see upper row in Figure 3), (because no other extreme positions can arise spontaneously, the total number of maximum positions has been decreased), or the maximum position has moved for one processor in the inverse shift direction (see lower row of Figure 3). In the following lemma, the two alternatives of improvement are derived.

Lemma 2: (Improvement)

Given is a non-balanced load configuration $L_i > 0$*, with* $i \in I$ *and* $L_{max} = max\{L_i \mid i \in I\}\}$*. After a load balancing step using LM-C5, every maximum position* $m \in I$ *at the left end of a plateau (*$L_m(t) = L_{max}(t)$ *and* $L_{m-1_d}(t) < L_{max}(t)$*, for one d) is either:*

(1) reduced in its value for one load unit:
$L_m(t+1) = L_{max}(t) - 1$ *and* $L_{m-1_d}(t+1) < L_{max}(t)$ *or*

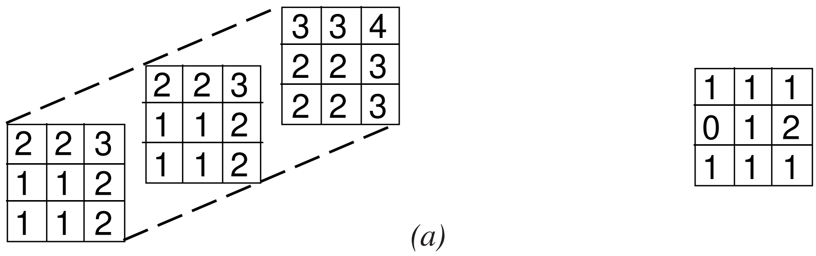
(2) moved left for one processor element:
$L_m(t+1) = L_{max}(t) - 1$ *and* $L_{m-1_d}(t+1) = L_{max}(t)$

Proof. Because the load configuration $L_i(t)$ is not balanced, there always exist a dimension d and an index $m \in I$ with $L_m = L_{\max}$ and $L_{m-1_d} < L_m$. With that, P_m forms the upper (right) end of a ramp of Definition 4. Let P_n, be the corresponding lower (left) end. Thus, $L_m - L_n$ indicates the load difference of the ramp.

Case 1. If $L_{m-2\cdot 1_d} < L_{m-1_d}$ or $L_{m-1_d} + 1 < L_m$ holds, then the load difference is greater than one. According to Theorem 1, one load element has been transferred virtually from P_m to P_n after one load balancing step. Because $L_m(t+1) = L_m(t) - 1$ holds, L_m is no longer as high as the previous maximum height, i.e., $L_m(t+1) < L_{\max}(t)$. Because the load difference is grater than one, no other global maximum at the lower end of the ramp can arise and, thus, condition (1) is satisfied.

Case 2: Let $L_{m-2\cdot 1_d} \geq L_{m-1_d}$ and $L_{m-1_d}+1 = L_m$. The direct neighboring processor P_{m-1_d} forms the left end of a ramp and the load difference of the ramp is equal to one. According to Theorem 1, the maximum position P_m is virtually transferred to P_{m-1_d} with $L_m(t+1) = L_m(t) - 1$. Because of the small load difference, a new global maximum position arises at P_{m-1_d} after the load transfer, i.e., $L_{m-1_d}(t+1) = L_{\max}(t)$. The former maximum position of P_m has moved left to P_{m-1_d} and condition (2) is satisfied. ◊

In Case 2 of the lemma, the extreme value positions cannot be reduced because there is a load difference of only one element in the ramp. E.g., if a ring of processors with a load ramp of height one is part of a two-dimensional torus, then a orthogonal ring may have such a ramp, too. Thus, the number of total load difference in the torus is increased by one. If we regard higher-dimensional topologies then the total load difference, which will not necessarily be



(a) *(b)*

Figure 4: In (a) an example of a balanced load configuration in a 3-dimensional torus, with total load difference of 3 elements is given. In (b) a balanced 2-dimensional torus with an unbalanced ring is shown.

reduced by LM-C5, increases with the number of dimensions. This is because the extreme positions may cycle on different rings. An example of this situation is given in Figure 4a.

On the other hand, if the total load difference in a D-dimensional torus is greater than D, the difference will be reduced by LM-C5. This fact is shown in the next theorem. Before that, the relation of balanced torus and their embedded rings is pointed out.

Lemma 3: (Balance)
Given is a D-dimensional torus. If all existing rings are balanced then the torus is balanced, too.

Proof. Let all rings in the D-dimensional torus be balanced (according to Definition 2) and let $x, y \in I$ be two arbitrary processors. Then, each path from x to y leads via at most D pairwise orthogonal rings. Each ring contains a load difference of at most one load element because they are balanced. Thus, the load difference between processor x and y sum up to at most D load elements. This argument holds for all processor pairs x, y and, therefore, the torus is balanced. ◊

The reverse implication does not hold because two extreme positions x and y in a D-dimensional torus, with $1 < |L_x - L_y| \le D$, may belong to the same ring. See Figure 4b for a counterexample. Together with the in Lemma 2 derived possibilities for improvement, this lemma is used to show that an overall balancing of the disturbed load is reached.

Theorem 2: (Convergence)
A unbalanced load configuration $L_i > 0$, with $i \in I$ in a D-dimensional torus, will converge to a balanced configuration when LM-C5 is applied.

Proof. As long as the torus is not balanced, there exist at least one unbalanced ring with load difference greater than two elements (Lemma 2). Applying Lemma 3 to this ring, then, in every load balancing shift, either a global extreme position is reduced, which is the trivial case, or is is moved, which we will regard further. Without loss of generality, we can asume that the extreme position is a maximum. Because this ring has a load difference greater than two

elements, there exist a corresponding (local) minimum position, which will move in the inverse direction. Thus, the distance of the extreme positions is reduced regarding to the dimension of the ring. (Here, the distance in the ring is measured in the move direction, i.e. the extreme positions may diverge at first.)
After a shift in another dimension these two extreme posisions may not belong to the same ring anymore. Still, there exist another unbalanced ring in which the distance is further reduced according to the above arguments. This process repeats until the maximum is reduced due to Case 1 of Lemma 2 and there is no unbalanced ring in the torus anymore. Hence, the load configuration of the torus is balanced. ◊

With that, the following upper bound for the necessary time effort of the LM-C5 can be set-up.

Theorem 3: (Efficiency)
For balancing an unbalanced load configuration $L_i > 0$, with $i \in I$ and $L_{diff} = max\{|L_i - L_j|, i, j \in I\}$, in a symmetrical, D-dimensional torus with $P = K^D$ processors, $K \in \mathbb{N}$, a maximum time effort T (measured in shifts) by LM-C5, is necessary of:

$$T = O(D \cdot K \cdot L_{diff})$$

Proof. It is sufficient to prove, that the global maximum is reduced for at least one load element after $O(D{\cdot}K)$ load balancing steps, i.e., $L_{max}(t + O(D{\cdot}K)) < L_{max}(t)$ holds, with $L_{\max} = \max\{L_i \mid i \in I\}$. For that, the set of global maximum positions is regarded. Let $m \in I$ with $L_m = L_{max}$ be such a position.
Case 1. Let ($L_{m-1_d} < L_{max}-1$) or ($L_{m-1_d} = L_{\max}-1 \wedge L_{m-2\cdot1_d} < L_{m-1}$) for one dimension d. According to Lemma 2 (Case1), the maximum position m is removed by one load balancing step.
Case 2. Let ($L_{m-1_d} = L_{max}-1 \wedge L_{m-2\cdot1_d} \geq L_{\max}-1$). According to Lemma 2 (Case2), the maximum position m moves left onto $m - 1_d$ in dimension d by one load balancing step. The Manhattan distance of the maximum position to the nearest position n with $L_n \leq L_{max} - 2$ is at most $D{\cdot}K$ in a torus with unidirectional links. According to Theorem 2, this distance will be reduced by moving both extreme positions in inverse directions. Thus, at most $D{\cdot}K/2$ steps are necessary until the maximum position is equilibrated.
Case 3. Let $L_{m-1_d} = L_{max}$. The position m will not move before all additional maximum positions lying directly to the left of m have been moved and $L_{m-1_d} < L_{max}$ holds true. In the worst case, $K - 2$ positions have to move. After that, for position m, Case 1 or 2 is appropriate.
In all cases, at most $O(D{\cdot}K)$ load balancing steps are necessary to remove the maximum position m. Because this position is representative for all existing maximum positions and all

positions are processed in parallel, $L_{max}(t + O(D \cdot K)) < L_{max}(t)$ holds. With the maximum load difference L_{diff}, the theorem is proven. ◊

The last theorem shows that local load balancing can be efficient. Contrasting with NNA in Section 2, the Liquid model shows only linear time effort in the single parameters using a tori as interconnection network.[7] The NNA as special case of the diffusion approaches needs quadratic time depending on the maximum number of processing units per dimension. Comparing LM-C5 with the dimension exchange in hypercubes, the LM-C5 has the same asymptotical time effort, which has a logarithmic form.

When examining the analytical results of LM-C5 and other load balancing methods, please note that two different time measures have been applied. The linear time effort of LM-C5 is measured in the amount of communicated data. The time effort of NNA or of the dimension exchange is measured in the number of communication set-ups. The data amount is always greater or equal to the number of set-ups, because for transferring a single data unit at least one set-up of the connection is necessary. Thus, the comparison of the two different time measures is justified.

5. Simulation results

The application algorithm with dynamic load balancing can be viewed as two interlaced, adversary processes. The application algorithm disturbs the load distribution by increasing or reducing the load in an unpredictable way. On the other hand, the load balancing process tries to re-equilibrate the load by transferring load elements. Investigating solely the balancing process apart from the application algorithm makes the effects more clearly recognizable. Because the nearest-neighbour-averaging method (NNA) implements the optimal diffusion parameter of the diffusion approaches in several topologies, we use this method for comparison.

We compared NNA and LM with different shifting conditions in ring simulations of size P. As the worst case scenario, one processor holds the total load $L_{sum} = c \cdot P$, where $c > 0$ is an arbitrary integer, and the remaining processors are idle. Thus, in the perfectly balanced system state, every processor holds c load elements. Each balancing method is executed synchronously until the balanced state is reached. For NNA, a synchronous variation of the method in Section 2.3 is used. If a non-integer amount of load should be shifted, then the amount is rounded asymmetrically. When transferring to the right and to the left, it is rounded upwards and

[7] For the common topologies, either the number of processors K per dimension or the dimension D itself grows with the topology size, but not both simultaneously.

T:	LM-C5:								NNA:							
0	16	0	0	0	0	0	0	0	16	0	0	0	0	0	0	0
1	15	1	0	0	0	0	0	0								
2	14	1	1	0	0	0	0	0								
3	13	1	1	1	0	0	0	0								
4	12	1	1	1	1	0	0	0								
5	11	1	1	1	1	1	0	0								
6	10	1	1	1	1	1	1	0								
7	9	1	1	1	1	1	1	1								
8	8	1	1	1	1	1	1	2								
9	7	1	1	1	1	1	2	2								
10	6	1	1	1	1	2	1	3								
11	5	1	1	1	2	1	2	3	5	6	0	0	0	0	0	5
12	4	1	1	2	1	2	2	3								
13	3	1	2	1	2	1	3	3								
14	3	2	1	2	1	2	2	3	5	4	2	0	0	0	1	4
15	3	2	2	1	2	1	3	2								
16	2	2	2	2	1	2	2	3	4	4	2	1	0	0	2	3
17	2	2	2	2	2	1	3	2	4	3	2	1	1	0	2	3
18	2	2	2	2	2	2	2	2	3	3	2	2	0	1	2	3
19									3	2	3	1	1	1	2	3
20									2	3	2	2	1	1	2	3
21									3	2	3	1	2	1	2	2
22									2	3	2	2	1	2	2	2
23									2	2	3	1	2	2	2	2
24									2	2	2	2	2	2	2	2

Table 3: A worst case example depicted over time T for the Liquid model (LM-C5) and nearest-neighbour-averaging (NNA) with eight processors in a ring topology.

downwards respectively. This insures that the perfectly balanced system state is actually reached and not only "ramps" turn up.

As time unit *T*, the number of shifts executed synchronously is used. Because NNA generally needs more than one shift per logical load balancing step (one NNA iteration), the maximum number of load transfers per single load balancing step is summed. In the LM, this corresponds to exactly one shift. This time measure is reasonable, especially if the set-up time for communication is short compared with the transfer time itself. This case holds especially for the tightly-coupled systems regarded here.

In Table 3, a worst case example for NNA and LM is given. In a ring structure with eight processors, initially only one processor holds 16 load units. For both methods, only the logical elementary steps are recorded. Please note that multiple steps are necessary for one averaging with NNA. Before NNA has averaged the first two processors, LM could already reach the load sharing goal after the seventh step. After the 18-th step, LM has reached the load balancing goal. NNA still needs 6 further steps for balancing. Altogether, in this example, LM is about 25 % faster than NNA.

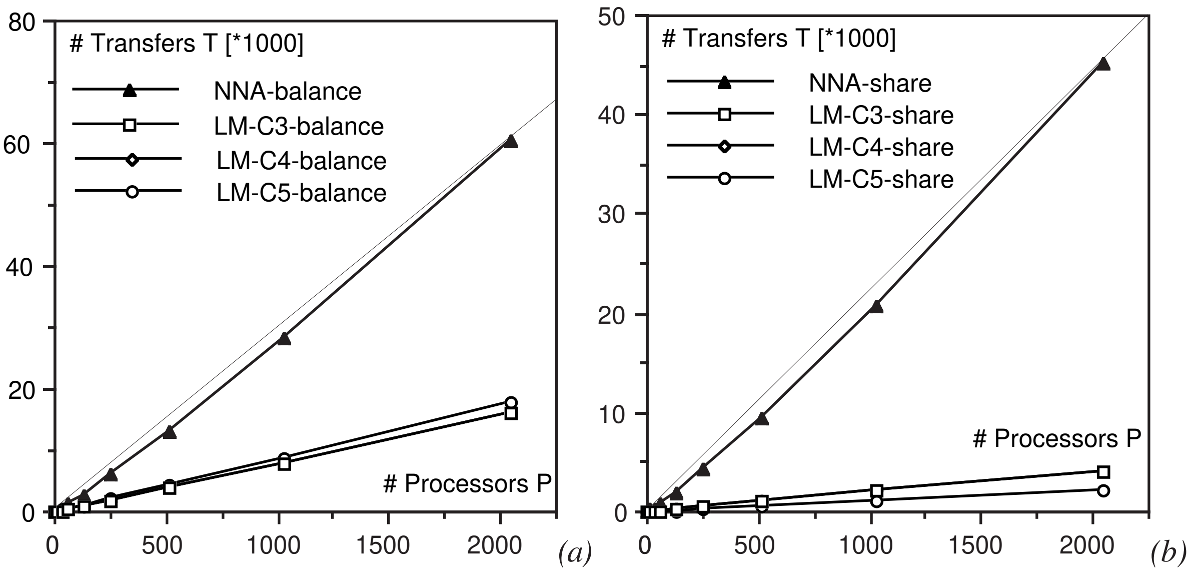


*Figure 5: Simulation results of load balancing (a) and load sharing (b) by the nearest-neighbour-averaging (NNA) and the Liquid model (LM) for synchronous processing in a ring for the worst case (*c = *5)*

The general validity of the run-time proportions in the above example can be shown by further simulations. In Figure 5, the simulation results for different numbers of processors are given. The results show an almost linear increase in time with P for all balancing methods. When numerically fitting analytical functions to the results, quadratic components with small coefficients can be recognized only for NNA. For load balancing, NNA needs about four times longer than LM within our range of processor numbers. For load sharing, the difference is worse – NNA is 23 times slower than LM.

With LM, the load elements are always shifted, unless the load difference to the successor is negative (C3 through C5). Therefore, load sharing has priority compared to load balancing. In the worst case (see above), only when all processors are supplied with a load the load balancing phase begins. The time effort of LM for load sharing amounts to $T = P - 1$, measured in necessary shifts. This explains the huge difference between LM and NNA for load balancing.

Additionally, simulation results not shown here indicate a linear increase in time with the size L_{sum} of initial load.

In Figure 6 the runtime behaviour of NNA is further investigated. Depending on the initial load $L_{sum} = c{\cdot}P$ of the first processor, the normalized number of NNA iterations is depicted. For NNA, one iteration mostly includes several shifts. The data points of each initial load have been normalized so that they match at value One for 512 processors. If iterations instead of shifts are depicted then NNA shows not such an extreme behaviour, because the multiple shifts per iteration smear the quadratic form. Altogether, the quadratic portion of the

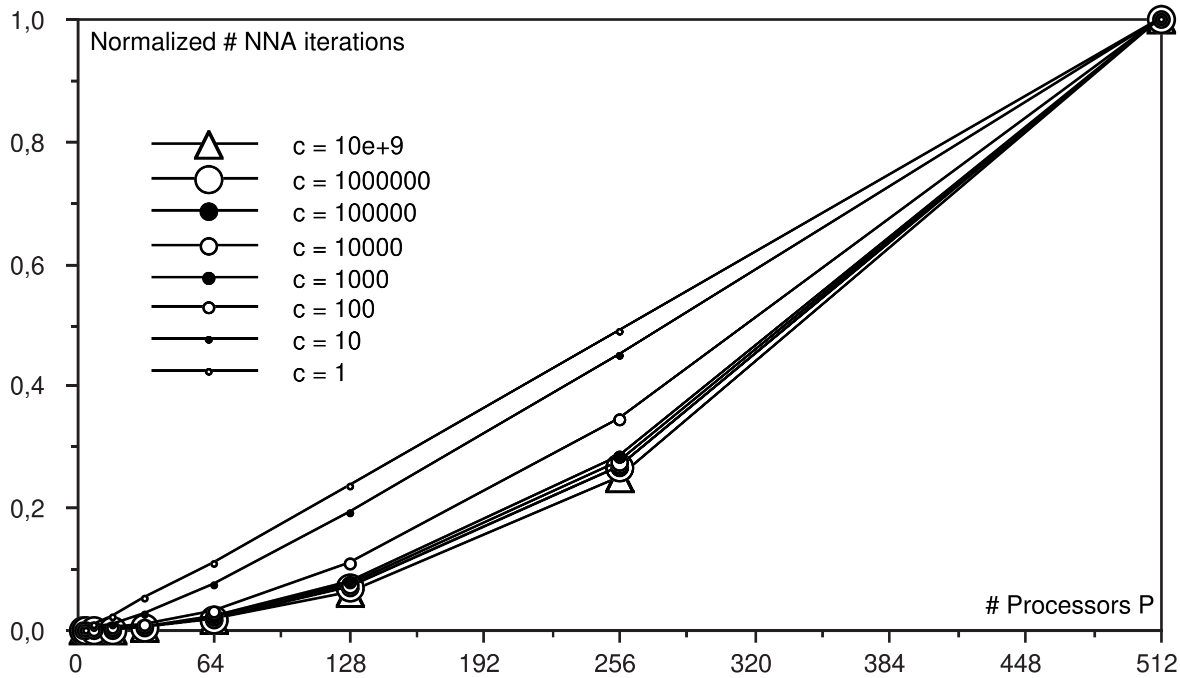


Figure 6: Dependency of the nearest-neighbour-averaging (NNA) execution time of the initial load difference $L_{sum} = c \cdot P$ plotted by the number of iterations normalized to one for up to 512 processors.

runtime increases with the initial load of the first processor (the total number of load elements, respectively). The borderline case is formed by continuous load.

In Figure 7, the simulation results of LM in a squared, two-dimensional torus are depicted for an increasing number of processors. In (a), the worst case with $L_{sum} = c \cdot P$ load elements on only one processors is assumed ($c = 5$) as initial load distribution. For the last three conditions (C3 through C5), identical numbers of iterations are necessary to reach the balanced state. In (b), the initial load was uniformly ranomized among 0 and 100 load elements. As above, the results show an only linear time dependency of LM on the number of processors.

To show the efficiency of the LM, NP-hard scheduling problems are used as an application domain in [Henrich94, Henrich95]. For the experiments, we used the MasPar SIMD machine MP-1 with 16384 processors arranged in a two-dimensional torus. Altogether, 20 problem instances with 10^6 up to 10^8 expanded nodes of the search tree were solved. The different shift conditions mentioned in Table 1 show very different behavior in the experimental results. All three conditions performing only the load sharing task (C0 through C2) are average. In contrast, the LM using any shift condition including load balancing (C3 through C5) are very efficient. All of them outperform the NNA method.

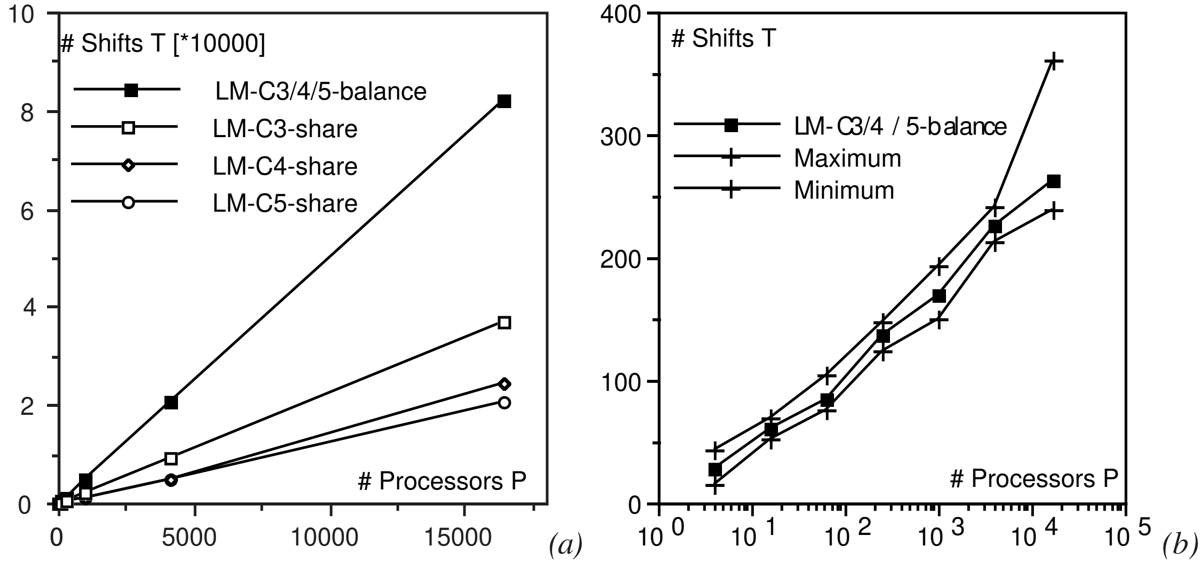


Figure 7: Simulation results of load balancing and sharing by LM in a 2-dimensional torus for the worst case (a) and for a uniform random distribution (b).

6. Summary

The realization of the Liquid model leads to a series of scalable and efficient dynamic load balancing techniques (LM-C3 through LM-C5). This is ensured by the strong locality of the algorithm as well as by exploiting the feature of tightly-coupled processing units. As it has been proven for multi-dimensional tori and simulated for the ring and the 2-dimensional torus, it is expected to be suitable and efficient for various other topologies.

Besides the simplicity of the presented algorithm, the main advantage lies in the combination of load sharing and load balancing. The most simple version, following both the sharing and the balancing task (C5), yields the best overall run time. The Liquid model gives high priority to the sharing task before balancing is performed. Especially for algorithms with highly dynamic load distribution, as e.g., tree search techniques, this prioritization demonstrates to be efficient. But this property is useful in many more applications.

The Liquid model has outperformed a former local load balancing approach, nearest-neighbor-averaging (NNA). None of the above two properties are reached by the NNA algorithm. It does not have such global effects when executing only a single local operation, and NNA performs load sharing only as a consequence of balancing.

Acknowledgement

This research work was supported by the Deutsche Forschungsgemeinschaft (DFG) with a stipend by the "Graduiertenkolleg" of the Computer Science Department, University of Karlsruhe. The work was performed at the Institute for Real-Time Computer Systems and Robotics, Prof. Dr.-Ing. U. Rembold and Prof. Dr.-Ing. R. Dillmann. I want to thank Alf-Christian Achilles and Peter Sanders for proofreading an earlier version of this paper and the anonymous reviewers for their comments.

References

[Bertsekas89] Bertsekas D. P., Tsitsiklis J. N., 1989, "Parallel and Distributed Computation: Numerical methods", Prentice-Hall, Englewood Cliffs, NJ, pp 519-526.

[Boillat90] Boillat J. B., 1990, "Load balancing and poisson equation in a graph", Concurrency: Practice and Experience, vol 2, no 4, pp 289-313.

[Casavant88] Casavant T. L., Kuhl J. G., 1988, "A taxonomy of scheduling in general-purpose distributed computing systems", IEEE Transactions on Software Engineering, vol 14, no 2, pp 141-154.

[Cybenko89] Cybenko G., 1989, "Load balancing for distributed memory multiprocessors", Jour. Parallel Distributed Comput., vol 7, pp 279-301.

[Dragon89] Dragon K. M., Gustafson J. L., 1989, "A low-cost hypercube load balance algorithm", Proc. 4th Conf. Hypercubes, Concurrent Comput. and Appl., pp 583-590.

[Fox89] Fox G. C., 1989, "Parallel computing comes of age: Supercomputer level parallel computations at Caltech", Concurrency Practice Exper., vol 3, no 5, pp 457-481.

[Henrich94] Henrich D., 1994, "Local load balancing for data parallel branch-and-bound", Int. Conf. Massively Parallel Processing, June 21-23, Delft, The Netherlands.

[Henrich95] Henrich D., 1995, "Lastverteilung für feinkörnig parallelisiertes Branch-and-bound", Dissertation, Universität Karlsruhe.

[Hong88] Hong J.-W., Tan X.-N., Chen M., 1988, "From local to global: An analyzis of nerarest neighbor balancing on hypercube", Proc. 1988 ACM Symp. on SIGMETRICS, pp 73-82.

[Horton93] Horton G., 1993, "A multi-level diffusion method for dynamic laod balancing", Parallel Computing, vol 19, pp 209-218.

[Kumar91] Kumar V., Ananth G. Y., Rao V. N., 1991, "Scalable Load balancing techniques for parallel computers", Technical Report 91-55, Department of Computer Science, University of Minnesota.

[Lin87] Lin F. C. H., Keller R. M., 1987, "The gradient model load balancing method", IEEE Transactions on Software Engineering, vol 13, no 1, pp 32-38.

[Lüling92] Lüling R., Monien B., "Load balancing for distributed branch-and-bound algorithm", Proc. 6th Int. Parallel Processing Symp., pp 543-548.

[Qian91] Qian X.-S., Yang Q., 1991, "Load baalncing on generalized hypercube and mesh multi-processors with LAL", Proc. 11th Int. conf. on Distributed Computing Systems, pp 402-409.

[Schabernack92] Schabernack, 1992, "Lastausgleichsverfahren in verteilten Systemen - Überblick und Klassifikation", Informationstechnik it, vol 34, no 5, pp 280-295.

[Willebeek90] Willebeek-LeMair M., Reeves A. P., 1990, "Local vs. global strategies for dynamic load balancing", Proc. Int. Conf. on Parallel Processing, vol 1, pp 569-570.

[Willebeek93] Willebeek-LeMair M. H., Reeves A., 1993, "Strategies for dynamic load balancing on highly parallel computers", IEEE Trans. on Parallel and Distributed Systems, vol 4, no 9.

[Xu93] Xu C. Z., Lau F. C. M., 1993, "Optimal parameters for load balancing using the diffusion method in k-ary n-cube networks", Information Processing Letters, vol 47, pp 181-187.