Load Balancing
and
Parallel Algorithms II

A Model for Efficient Programming of Dynamic Applications on Distributed Memory Multiprocessors

A. Erzmann, M. Hadeler, C. Müller-Schloer

Institut für Rechnerstrukturen und Betriebssysteme Universität Hannover
Lange Laube 3, D-30159 Hannover, Germany
erzmann@irb.uni-hannover.de

Abstract. We present the TDC programming model which aims to ease the efficient implementation of dynamic applications on distributed memory multiprocessors. This model is based on task descriptors, data objects and capabilities which reside in distinct, globally accessible domains. Dynamic load balancing will be done by the system software and is completely transparent to the user. This often leads to a significant reduction of code complexity. Our prototype of the TDC model on an 128 node nCUBE2 uses a distributed diffusion scheme to balance load dynamically. We have developed a task selection strategy which reduces the load balancing overhead. Measuring and simulation results for a parallel implementation of a block matching algorithm indicate that runtime efficiency close to the optimum can be achieved with the TDC model even for highly parallel systems.

Keywords. Programming Model, Dynamic Load Balancing, Block Matching, Distributed Memory Multiprocessors.

1 Introduction

From the user's point of view two goals are crucial when programming distributed memory multiprocessors: *efficiency* and *ease-of-programming*. Whether both requirements can be met simultaneously depends on the appropriate choice of the *programming model*. The programming model is the description of the virtual parallel architecture as it is seen by the user, i.e. it provides a certain level of abstraction. The better this level fits the application requirements, the easier the programming will be for the user. On the other hand, the implementation of the programming model on a given parallel machine may be the source of substantial efficiency losses if the provided abstraction and the hardware differ too much. A programming model is easy to use if it precisely supports the application needs and on the other hand, may be implemented efficiently if its abstractions closely resemble the underlying hardware. If the problem structure does not naturally map onto the hardware of the parallel machine, then the user has to find a good trade-off between the ease-of-programming and efficiency by carefully choosing the programming model. This choice will be influenced by the application properties and the *target architecture*.

2 The TDC Programming Model

Before the description of the *TDC programming model* (*Tasks, Data Objects and Capabilities*), we define the application class and target multiprocessor architecture:

Application Class

The TDC Programming Model is designed to ease the efficient programming of *dynamic applications*. These applications have at least one *dynamic execution phase* of considerable length with respect to the total execution time. A dynamic execution phase (DP) has the following properties: The work which has to be done during a DP can be split into a certain number of *tasks* which may be created at any time during the DP. All tasks which are existent at a given point of time during the DP can be executed independently and in arbitrary sequence. All information needed to process a task is available as soon as the task is created, i.e. there is no need for direct task interaction. Task dependencies are resolved by delayed creation of the dependent task. The task execution time depends on the input data and cannot be predetermined. Examples for dynamic applications are: ray tracing, volume rendering and block matching algorithms.

Target Architecture

The target architecture is a general purpose distributed memory multiprocessor (MIMD). The nodes of this multiprocessor are linked via an interconnection network. Examples for this class of multiprocessors are: nCUBE 2, Intel iPSC, Intel Paragon.

2.1 Description

The parallelisation of a dynamic application for a distributed memory multiprocessor requires the following steps: (1) The load has to be partitioned into tasks, which can be executed in parallel. (2) The input data has to be partitioned and (3) tasks have to be assigned together with the required parts of input data to the nodes of the parallel machine. Since the task execution times are unpredictable, static assignment of tasks often leads to load imbalance, i.e. low efficiency. To achieve high efficiency it is necessary to balance the load dynamically, which is done by repeated reassignment of tasks and data parts at runtime. Since dynamic load balancing requires complex algorithms especially in highly parallel environments and these algorithms are essentially the same for a wide range of applications, it is desirable to separate dynamic load balancing activities from the application code. The TDC programming model provides the user with the abstractions necessary to program his application easily. Moreover it is designed for efficient implementation on the target architecture.

TDC is based on three separate, globally accessible domains: The *task bag* which contains *task descriptors*, the *data space* which contains *data objects* and the *code space* which contains *capabilities*[1] (Fig. 1).

Worker

The user process, which is written by the programmer, is called the *worker*. It initializes the data objects and inserts task descriptors into the task bag. In order to do work, it removes task descriptors (one after another) from the task bag and executes the task using the specified capability. The task descriptor contains a set of user defined parameters

---

[1] The term capability is not to be confused with an access control mechanism of the same name in the context of data security.

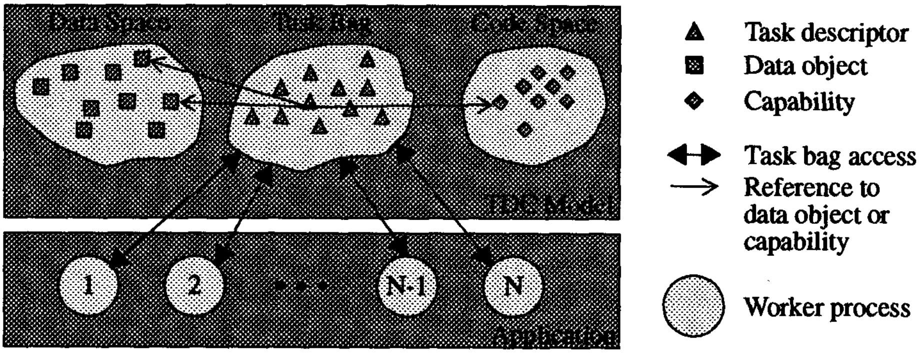

Fig. 1. The TDC programming model.

which specify details of the task execution process. During the task execution period the worker accesses data objects, which are specified in the task descriptor. Typically there will be a single worker per processing node.

Task Descriptor

A task descriptor is a data structure which contains (1) references to an arbitrary number of data objects and a capability and (2) a set of input parameters, which is used by the capability to distinguish tasks from each other. A task descriptor is created and inserted into the task bag by a certain worker as soon as the corresponding task is executable and will be removed by a potentially different worker in order to execute the specified task.

Data Object

Data objects are contiguous pieces of data associated with an globally unique logical identifier. They are initialized by an arbitrary worker before they are accessed and may be read by any number of workers simultaneously. The size and number of data objects is dependent on the application. Data objects reside in the data space.

Capability

A capability is a piece of code which is used by the worker to process a given task. Capabilities must be loaded into the code space prior to task execution. They can be accessed by an arbitrary number of workers simultaneously.

2.2 Related Programming Models and Tools

In this section two related programming models (Message Passing, Linda) and the tool Dynamo for distributed memory multiprocessors are described and compared to TDC. The suitability of these models for programming dynamic applications is discussed.

Message Passing

In the Message Passing programming model, processes communicate and synchronize by explicit exchange of messages. No globally accessible data structures exist and therefore tasks, data objects and code have to be encapsulated inside the worker processes. Dynamic load balancing can be done (1) by the programmer himself, which is often prohibitive because of the involved implementation complexity, or (2) by an underlying process migration system [1]. The latter however requires that the location of processes is transparent to the programmer. Moreover the number of processes per node must be raised artificially in order to make process migration feasible. Thus the benefit of dynamic load balancing is limited due to the additional overhead and the coarse grain size of the balancing entities, i.e. the processes.

Linda

In Linda [2] workers communicate and synchronize via *tuples*, which reside in the globally accessible *tuple space*. Valid operations on tuples are: insert tuple, remove tuple and read tuple. Since tuples are accessed by contents rather than by address these operation require a costly associative matching process. In Linda tuples do not have a semantic meaning, i.e. the system cannot distinguish between task descriptors and data objects. Load balanced execution of dynamic applications is achieved automatically up to a certain degree since workers remove task tuples from tuple space as needed. This works well for shared memory multiprocessors but is likely to produce overhead on distributed memory machines because the communication latency cannot be hidden (tuples reside anywhere in the system and must be searched for and transferred to the worker which accesses them). The system is not able to support load balancing since task descriptors cannot be identified and load balancing costs cannot be calculated.

Dynamo

Dynamo [3] is a library for dynamic load balancing on distributed memory multicomputers based on the PICL message passing primitives. It provides support for managing local task queues and for writing code which dynamically balances these queues. The main differences to TDC are: (1) The underlying programming model neither supports shared data objects nor capabilities. This limits the range of applications which can be programmed using Dynamo. (2) The dynamo implementation does not use a runtime system, i.e. the programmer must explicitly call the load balancer from its application program (see chapter 3 for the TDC implementation). This synchronizes load balancing and task execution and thus limits the range of balancer algorithms which can be used.

2.3 Comparison of Programming Models

As stated in the introduction our main concern is to provide ease-of-programming and high efficiency. Therefore the question which arises is: How much effort is required by the user to write an efficient implementation of a dynamic application using one of the above-described programming models?

The effort to identify and express parallelism is essentially the same: In all cases a suitable partitioning of load and input data has to be found. Using Message Passing, the partitioning generally is contained inside the code, whereas tasks and data objects have to be created explicitly if Linda, Dynamo or TDC are used. The main difference is the way how dynamic load balancing will be achieved: In Message Passing programs, the programmer has to insert dynamic load balancing operations into the application code, thereby increasing the code complexity substantially, whereas dynamic load balancing will be done by the system software if Linda, Dynamo or TDC are used.

A mechanism which we call *passive load balancing* is inherent to the Linda programming model: Each worker, which becomes idle, requests a new task. This task (a Linda tuple) has to be located in the distributed tuple space and must be moved to the node where the worker resides. The same is true for each data tuple which is accessed during task execution time. The worker remains idle until the requested task and data arrive, i.e. the network latency, which usually dominates the transfer time, cannot be hidden. In TDC the semantics of task descriptors is known to the system. It therefore can perform *active load balancing*, i.e. task descriptors, data objects and capabilities can be moved to underloaded nodes in advance. This enables the system to overlap communication and computation, thereby hiding network latency and minimizing idle times.

3 Implementation

The TDC programming model has been implemented[2] on an 128 node nCUBE 2 as runtime environment on top of the VERTEX® node operating system [4]. Our main implementation goals were:

- *High efficiency.* The local parts of the distributed task bag and data space reside in a portion of memory which is shared by the TDC system and the worker. This avoids unnecessary copying of potentially large data objects and task descriptors. The TDC system itself is implemented in a distributed fashion, i.e. there does not exist any central resource that might become a bottleneck in highly parallel systems.
- *Ease-of-use.* The user accesses the TDC system by a well-defined set of C library functions which are linked to the application code. The TDC system will be configured dynamically and is suitable for future integration into the node operating system.

3.1 Overview

The TDC system consists of two system processes per node (see Fig. 2): The *task server processes* control the task bag and perform dynamic load balancing. The *data server processes* send and receive *data objects*.

3.2 User Interface

The worker process accesses the TDC system by a set of C library functions which are described briefly in this section. These library functions hide the

---

[2] Currently it is assumed that each worker stores the entire set of capabilities. The code space is therefore integrated into the worker processes. This restriction does not affect the general applicability of this implementation but may increase memory requirements.

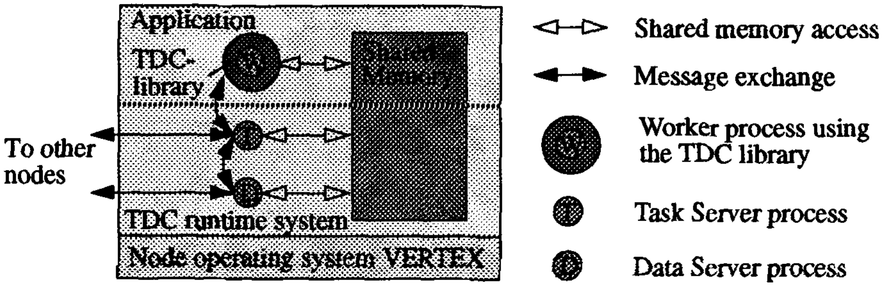

Fig. 2. Overview of TDC implementation on a node of the nCUBE 2 multiprocessor.

tdc_init()	Dynamic configuration of the TDC system. The size and maximum number of task descriptors and data objects are specified as well as the logical identifiers of the locally stored data objects. It synchronizes the workers.
tdc_flush()	Delete data objects and capabilities. This may be necessary if multiple dynamic execution phases occur within the same application.
tdc_write()	Create and initialize a data object.
tdc_put()	Insert a task descriptor into the task bag.
tdc_get()	Retrieve a task descriptor from the task bag. This function returns end-of-processing if all tasks have been processed, i.e. the task bag is empty.

Example Usage

Fig. 3 shows an worker program written using the TDC programming model. The function `process()` contains the application code which processes the given task.

```
/* Initialize TDC system */
tdc_init(<application specific parameters>);
/* Initialize data objects */
forall(<local data objects>)tdc_write(<data object>);
/* Put tasks into task bag */
create_tasks();
forall(<local tasks>)     tdc_put(<task descriptor>);
/* Process tasks */
while (tdc_get(&task) != EOP) process(task);
```

Fig. 3. Worker program written using the TDC programming model.

4 Implementation of Block Matching Using the TDC Model

In this section we demonstrate how a real application can be parallelized using the TDC programming model and present a modified receiver initiated diffusion algorithm for dynamic load balancing. This algorithm uses a heuristic approach for the task selection strategy which aims to minimize the load balancing overhead thereby maximizing the overall efficiency.

4.1 The Block Matching Algorithm

In a sequence of pictures successive pictures are likely to be quite similar. Therefore the amount of data necessary to store or transmit image sequences can be reduced significantly if only the difference between two pictures is coded rather than coding each picture separately. The picture which has to be coded will be partitioned into square blocks. For each of these blocks the most similar block is searched for in the previous picture. This is done by calculating the *mean absolute error* for each block and choosing the block with the least error value. This technique is called *block matching* and is used by programs with compress image data according to the MPEG standard [5].

In our implementation we choose each image of the sequence to be a *data object* and the matching of one block to be a *task*. Consequently each task requires two data objects (images) to be processed. Tasks which require an identical set of data objects belong to the same *task class*. The maximum number of task classes which can be stored on a node simultaneously is limited by its memory size.

A static assignment of tasks to nodes would lead to load imbalance mainly for two reasons: The task execution time depends on the contents of the images. This is true since the computation of the mean absolute error is stopped as soon as its value becomes greater than the minimum found so far, i.e. finding a good reference block early will lead to shorter execution time. Moreover due to the limited I/O bandwidth the loading of images takes a variable amount of time and therefore prevents nodes from starting with task execution at the same time.

4.2 Dynamic Load Balancing

The load balancing system which is integrated into the task server processes can be divided into four sections:

- Dynamic load balancing strategy
- Task selection strategy
- Local load estimation and load update policy
- Load imbalance detection and balancer activation

Load Balancing Strategy

A variety of dynamic load balancing strategies have been proposed for highly parallel systems ([6],[7],[8],[9]). The load model assumed here has two important differences:

- Task migration may cause data object migration. Thus the task migration overhead cannot be neglected.
- The limited number of distinct task classes per node influences the task selection.

These restrictions imply the use of a task selection strategy which considers the task migration costs and task class limitations. Therefore the receiver (the underloaded node) must control the task selection and migration process. We have chosen the RID strategy (*receiver initiated diffusion*, [8]) for 3 reasons: (1) RID is a completely asynchronous and distributed approach, (2) task migration is controlled by the receiver and (3) this strategy performs comparable or better than the other strategies ([8],[10]).

The *balancing domain* consists of the underloaded node itself (receiver) and its direct neighbors. Each time the algorithm is invoked, it balances load locally. Successive local balancing steps lead to globally balanced load [7]. We describe a local balancing step: Let K be the number of direct neighbors per node, l_0 the receiver's load and l_i the load of its i-th neighbor. The average load l_{avg} in the balancing domain is:

$$l_{\text{avg}} = \frac{1}{K+1} \cdot \sum_{i=0}^{K} l_i \qquad \text{(Eq. 1)}$$

The fraction d_i of load which has to be demanded from the i-th neighbor in order to balance load in the local domain is calculated according to the following formulas:

$$h_i = \max(l_i - l_{\text{avg}}, 0) \qquad h_{\text{sum}} = \sum_{i=1}^{K} h_i \qquad d_i = (l_{\text{avg}} - l_0) \cdot \frac{h_i}{h_{\text{sum}}} \qquad \text{(Eq. 2)}$$

Task Selection Strategy

The *load balancing strategy* determines how many tasks have to be migrated within one local balancing step, whereas the *task selection strategy* is used to minimize the load balancing overhead by choosing the set of tasks which causes the least cost. We assume that the task migration cost is primarily caused by the involved data object transfers. Thus it is always profitable to select as many tasks of a given class as possible. We use an iterative, heuristic approach with low computational complexity to get a near optimal task selection instead of doing a full search. The receiver executes the following steps:

1. For all neighbors: Calculate the number of tasks d_i, which have to be demanded from neighbor i according to the RID balancing strategy (Eq. 2).
2. For each neighbor i and each task class j in the balancing domain: Let a_{ij} be the number of tasks of this class which are *available* on neighbor i. Calculate the maximum number of tasks r_{ij} of class j which might be requested from neighbor i:

$$r_{ij} = \min(d_i, a_{ij}) \qquad \text{(Eq. 3)}$$

3. For each task class j: Calculate the total number of tasks which could be requested if class j is selected:

$$R_j = \sum_{i=1}^{K} r_{ij} \qquad \text{(Eq. 4)}$$

Determine the cost C_j due to data object migration, which would be caused by the migration of tasks class j. Assign infinite costs to class j if this class cannot be selected due to the limited number of task classes on the receiving node. Calculate the cost per task c_j assuming that R_j tasks will be transferred:

$$c_j = \frac{C_j}{R_j} \quad \text{(Eq. 5)}$$

4. Determine the task class k with the least cost per task. If no class with finite cost per task can be found then no more tasks can be migrated: Stop here.
5. For each neighbor i: Request r_{ik} tasks of class k and adjust d_i accordingly:

$$d_i \leftarrow d_i - r_{ik} \quad \text{(Eq. 6)}$$

6. If still tasks remain to demand, i.e. any $d_i \neq 0$, continue with step 2, else stop here.

Local Load Estimation and Load Update Policy

We assume that the average task execution time t_{avg} is constant. This time is estimated locally by averaging the measured task execution times of the tasks which have been processed so far. Since t_{avg} is time invariant, we simply use the number of tasks which still have to be processed as load estimate.

As soon as the local load changes significantly, load update messages will be sent to all neighbors in the balancing domain. These messages contain information about the task classes residing on the node and the number of tasks which are currently available for each of these classes.

Balancer Activation

Performing a load balancing step reduces the probability that nodes become idle and on the other hand causes overhead. Thus the frequency of balancer activation and the degree of load imbalance that can be tolerated must be chosen carefully in order to get the best overall efficiency. We use the number of tasks d_{sum}, which would be demanded if the balancer is activated, as a criterion for load imbalance in the balancing domain:

$$d_{\mathrm{sum}} = \sum_{i=1}^{K} d_i = \max(l_{\mathrm{avg}} - l_0, 0) \quad \text{(Eq. 7)}$$

Let t_{latency} be the network latency necessary to load the data objects for a new task class and l_{latency} the number of tasks which can be processed during t_{latency}:

$$l_{\mathrm{latency}} = \frac{t_{\mathrm{latency}}}{t_{\mathrm{avg}}} \quad \text{(Eq. 8)}$$

We distinguish two balancing phases: Phase 1 is active while the local load is greater than l_{latency}, phase 2 otherwise. During phase 1 load imbalance will be evaluated whenever the load drops below the threshold l_{bal}, which will be decreased by the constant activation factor f_a after each imbalance evaluation. Load will be balanced if the detected imbalance is greater than l_{latency}. During phase 2 load imbalance will be evaluated whenever the node recognizes a load change in the balancing domain. Load will be balanced each time an imbalance of at least one task is detected.

During phase 1 both load imbalance will be determined less frequently and a higher load imbalance will be tolerated compared to phase 2. Thereby unnecessary balancing overhead is avoided unless the local load drops below the critical threshold l_{latency} and on the other hand, the probability that nodes become idle is kept low.

Balancer Activation Algorithm

```
$l_{\text{bal}} = f_a \cdot l$
while (not end of balancing)
    if ($l_{\text{bal}} > l_{\text{latency}}$)                    /* Phase 1 */
        if ($l_0 \le l_{\text{bal}}$)
            evaluate $d_{sum}$
            if ($d_{sum} > l_{\text{latency}}$) balance load
            $l_{\text{bal}} = f_a \cdot l_{\text{bal}}$
        endif
    else                                                          /* Phase2 */
        if (load in the domain has changed)
            evaluate $d_{sum}$
            if ($d_{sum} \ge 1$) balance load
        endif
    endif
endwhile
```

4.3 Results

In this section the results we have obtained using our TDC implementation on an 128 node nCUBE 2 are presented and compared to simulation results. We have used the block matching algorithm to process the *flowergarden* video sequence as example application. Each picture of this sequence contains 720*576 pixels and is divided into 1620 square blocks. Each node has to compute the motion vectors for one quarter of an image (405 blocks) on average.

Simulation

We have used *event-driven simulation* based on traces of program execution to determine which efficiency could be optimally achieved, if (1) the balancer has global knowledge about the system load, i.e. it can get tasks from any node in the system, (2) dynamic load balancing activities, except the transfer of data objects, do not cause overhead and (3) the network latency is zero. Consequently, only the idle time at the end of the processing phase and the overhead for data object transfer is considered.

In Fig. 4 the efficiency for the block matching algorithm and a constant average load per node is depicted for a varying number of nodes. Additionally the overhead and the

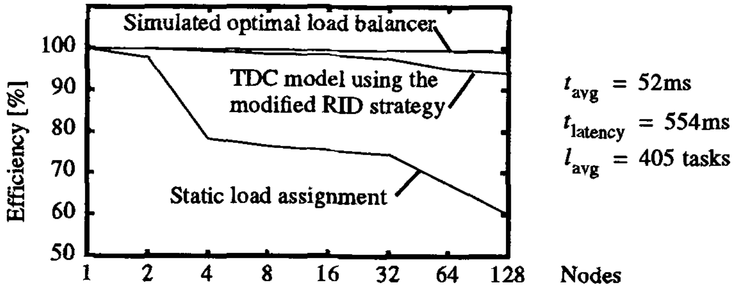

Fig. 4. Measured versus simulated efficiency for the Block Matching Application.

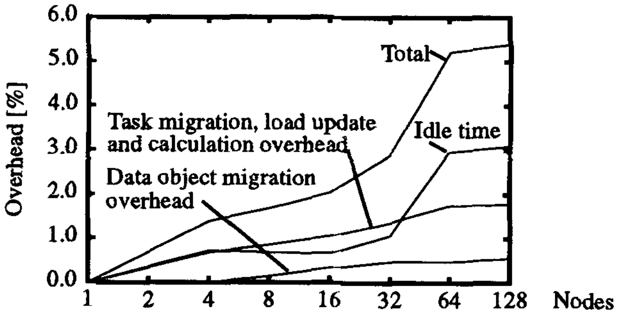

Fig. 5. Idle times and overhead for the RID dynamic load balancer.

idle times which occur if dynamic load balancing is used are shown in Fig. 5. The efficiency decreases as the number of nodes increases primarily for 3 reasons:

- *Statistic properties of the load.* The probability, that at least one node exhibits an execution time close to the maximum increases with the number of nodes. This leads to an increased fraction of idle time, since this node determines the execution time.
- *Number of task classes.* For the considered application the number of task classes is proportional to the number of processing nodes. As the number of task classes becomes higher, task migration becomes more restricted due to the limited number of task classes per node and the overhead caused by data object migration increases.
- *Network diameter.* The higher the network diameter, the more steps might be necessary to migrate tasks from overloaded to underloaded nodes thereby increasing the load balancing overhead and the fraction of idle time.

Fig. 4 shows that the modified RID algorithm is able to keep the efficiency close to the optimum even when the number of processing nodes is high. There is an increasing demand for dynamic load balancing in highly parallel systems since load imbalance tends to increase with the number of nodes for dynamic applications.The overhead for data object migration, which is controlled by the task selection algorithm, is kept consistently low. Our simulations have shown, that although the balancer can only select tasks in the local balancing domain, at most twice the number of data object migrations have been performed compared to the simulated balancer.

4.4 Conclusion

We have introduced the TDC programming model for distributed memory multiprocessors. TDC eases the implementation of dynamic applications since dynamic load balancing is now performed by the system rather than the programmer. The semantics of task descriptors, data objects and capabilities as well as the logical linkage between them is visible to the TDC system. This knowledge is used for active load balancing, i.e. load and the required code and data will be transferred by the system to underloaded nodes in advance thereby hiding network latency and hence reducing idle times.

We have implemented the TDC model on an 128 node nCUBE 2 and used this prototype to write a parallel version of the block matching algorithm. Our measurements and simulation results indicate that the receiver initiated diffusion scheme in conjunction with our task selection strategy leads to a runtime efficiency close to the optimum even for highly parallel systems. Further research is required to proof the suitability of this load balancing system for a broader range of dynamic applications.

References

[1] Ludwig, T.: *Lastverwaltungsverfahren für Mehrprozessorsysteme mit verteiltem Speicher*, Dissertation, Institut für Informatik, TU München, 1993

[2] Gelernter, D.; Ahuja, S.; Carriero, N.: *Linda and Friends*, Computer, Vol. 19, No. 8, Aug. 1986, pp 26-34

[3] Tärnvik, E.: *Dynamo - a portable tool for dynamic load balancing on distributed memory multicomputers*, Concurrency: Pratice and Experience, Vol. 6, No. 8, Dec. 1994

[4] nCUBE Cooperation: *nCUBE 2 Programmer's Guide*, PN 102294, 1992

[5] ISO/IEC 11172-2, *Information technology - Coding of moving pictures and associated audio for digital storage media at up to about 1.5 MBit/s - Part 2: Video*, Annex D.6.2, pp 78-85 Motion estimation and compensation

[6] Lin, F.; Keller, R.: *The Gradient Model Load Balancing Method*, IEEE Transactions on Software Engineering, Vol. SE-13, No. 1, Jan. 1987

[7] Cybenko, G.: *Dynamic Load Balancing for Distributed Memory Multiprocessors*, J. Parallel and Distributed Computing, Vol. 7, pp 279-301, October 1989

[8] Willebeek-LeMair, M.H.; Reeves, A.P.: *Strategies for Dynamic Load Balancing on Highly Parallel Computers*, IEEE Transactions on Parallel and Distributed Systems, Vol. 4, No. 9, Sep. 1993

[9] Gerogiannis, D.; Orphanoudakis, S.C.: *Load Balancing Requirements in Parallel Implementations of Image Feature Extraction Tasks*, IEEE Transactions on Parallel and Distributed Systems, Vol. 4, No. 9, Sep. 1993

[10] Erzmann, A.; Müller-Schloer, C.: *Zur Beurteilung dynamischer Lastausgleichsverfahren*, PARS Mitteilungen, Nr. 13, November 1994