A Hierarchical Load Balancing Environment for Parallel and Distributed Supercomputer

Rainer Pollak
Institute of Parallel and Distributed High-Performance Systems (IPVR)
University of Stuttgart
Germany

Abstract

This paper presents a scalable hierarchical approach for dynamic load balancing in large parallel and distributed systems, which is not only different from classical centralized and decentralized approaches, but also different from known hierarchical schemes. The present system, which is implemented as a prototype on the shared-nothing supercomputer architecture Intel Paragon XP/S, uses multi-level control for dynamic load balancing as well as for the communication manager. The control tree consists of three different components: the root component, the inner components and the leaf components.
The hierarchical load balancer uses non-preemptive as well as preemptive process migration to balance load between the nodes of the parallel/distributed system. The performance of the system has been evaluated by two existing real world applications: a computational fluid dynamic program and a parallel raytracer.

1 Introduction

As a result of the evolutionary advancements in computation and communication technology, parallel and distributed systems have become more and more popular as supercomputing environments during the last decade, but nevertheless they are still difficult to use. One of the reasons why these systems are so difficult to use is the problem of dynamic load balancing. To take full advantage of a parallel/distributed system the workload should be distributed equally among all available nodes in the system. This can be accomplished by using a static load balancing strategy [5] based on a priori knowledge of the problem structure as well as the runtime behavior of the application. This is a classical optimization problem which can be solved by a large variety of methods such as recursive bisection, simulate annealing, generic algorithms and heuristics. However, for a large class of parallel applications, the runtime behavior is not known in advance. For these problems, the workload is created dynamically during execution and/or the runtime behavior of the processes changes dramatically during the execution time of the application. Such problems need a dynamic load balancing mechanism [8][9][11][14][18] to redistribute the workload among the system nodes.
In general, there are two approaches to dynamic load balancing: It can be handled by either the application [6][20] or by the operating system [10]. While the first approach could take full advantage of the knowledge about the application and its runtime behavior, an operating system approach or a middleware approach [4] is more general and offers the possibility to be fully transparent to the application [7]. In the following, only the second approach and the middleware scheme respectively will be considered.
In order to realize a dynamic load balancer several design decisions have to be made. One of the most important design decisions concerns the questions, should the responsibility for the task of dynamic scheduling physically reside in a single node or should the responsibility be physically distributed among the nodes of the system.
A central dynamic load balancer [17] has several advantages. For example, a central load balancer is easy to realize on the basis of single flow of control, and it also has the potential of yielding optimal performance, because it has a global system view. On the other hand, a central scheme has several disadvantages, which makes this scheme unsuitable for large parallel and distributed systems [1]. One of these disadvantages is, that a centralized algorithm requires the accumulation of global load information which can become a formidable task.
A fully decentralized dynamic load balancer [14][18] is well scalable but has also several disadvantages, which makes this scheme not necessarily suitable for large parallel/distributed systems consisting of hundreds or thousands of nodes [1]. For instance, while using fully distributed schemes, optimal scheduling decisions are difficult or impossible to make [3], because each node in the system operates autonomously with only a local view of the global system state.

In order to combine the best of both worlds, a hierarchical load balancing approach has been chosen. It has the benefits of a fully distributed scheme regarding the parallelism in the leafs of the tree. On the other hand, it also offers the advantage of a centralized approach with regard to the use of aggregate information in the higher level of the load balancing hierarchy and the possibility to react fast to global load changes.
The target machine for the hierarchical load balancer is the parallel supercomputer Intel Paragon XP/S. The Paragon XP/S is a distributed-memory multicomputer. The processing nodes, which are based on the Intel i860 XP RISC processor, are connected by a two-dimensional mesh with a bandwidth of 200 MB/sec. full duplex.
The rest of the paper is organized as follows. In section 2 a brief overview of existing hierarchical load balancing schemes is given and the benefits of preemptive process migration are discussed. In section 3 and 4 the PaLaBer-System (**Pa**rallel **L**o**a**d **B**alanc**er**) is presented. In sections 5 and 6 performance results based on sample applications are presented. Section 7 concludes this paper.

2 Related work and motivation for a new approach

In this section, a brief overview of some existing work in the area of hierarchical load balancing and process migration is given.

2.1 Hierarchical load balancing

Ahmad, Ghafoor and Fox [2] propose a dual-level hierarchical scheduling approach for assigning parallel computations with dynamic structures to large hypercube multicomputers. The proposed scheme divides the system into multiple regions to achieve load balancing and to reduce processor contention. The first level scheduling is done by the host computer which spreads out the initial workload into these regions. The second level scheduling is done by a set of intermediate processors of these regions which enable the processors of their regions to distribute the dynamically created load optimally at runtime and to communicate with each other with reduced overhead. The class of parallel workload considered in [2] is viewed as a collection of computational modules that can execute asynchronously. Each computational module is an atomic unit of computation which cannot be preempted. A computational unit may, however, communicate with another task at the beginning and/or at the end of its execution. It is shown, based on a detailed simulation study, that the hierarchical approach outperforms a well-known distributed algorithm. However, this hierarchical approach does not support preemptive migration, so this scheme is only suitable for fine-grained units of work of roughly equal size. Furthermore, the assumption about the tree-like application structure makes this approach unusable as a general load balancing mechanism.
Tilborg and Wittie [22] present a multi-level hierarchy to schedule task-forces to processors. A task-force is defined as a collection of cooperating tasks that form a parallel program. The nodes at level-0 (called workers) are available for user tasks. Those at higher levels (called managers) are responsible for maintaining the integrity of the communication subnetwork and for performing resource allocation in a local region. The topmost level of the hierarchy consists of a collection of several "supermanagers". To avoid communication as well as processing overhead, each node in the system can directly exchange control information only one level upwards with its manager and one level downwards with its subnodes. Task forces can enter the system at any level of the hierarchy, worker nodes included. If a task force of size S enters the network at a level where nodes cannot schedule task forces of size S, the task force descriptor is passed up the tree until a suitable manager is reached. The same holds if the task force enters the hierarchy at such a high level that a subnode could schedule the entire task force. In this case, the task force descriptor is passed down until it reaches a level that minimally satisfies its needs.

2.2 Process migration

Most of the earlier studies of process migration do not consider parallel applications consisting of a number of homogeneous or heterogeneous processes.
Krueger and Livny [15] show in their trace-driven simulation study which is based on the data collected by Leland and Ott [16] from over 9.5 million processes executed by a VAX-11/780 under UNIX that, a facility to migrate active processes is often useful. The chosen load distribution algorithm is in its strictest form symmetrically initiated [11]: A node initiates negotiation with probability *SendProb* when becoming overloaded, or with probability *RecvProb* when becoming underloaded. A node states that it is overloaded or underloaded according to the relationship of its local load with two static thresholds. Similarly, identification of a suitable transfer partner also depends on two static thresholds. However, the processes are assumed to execute independently and without intercommunication.
On the other hand, Eager, Lazowska and Zahorjan [12] show, based on their *No Arrivals Model*, that while migratory load sharing can offer modest performance benefits under some fairly extreme conditions, there are no conditions under which migration yields major performance benefits. The *No Ar-*

rival Model they consider in their work allows them to study the performance of a system with the highest possible performance benefits of migration. This model considers a system with an extreme arrival process in which a bulk of processes arrives simultaneously, and then there are no new arrivals until all of these processes have been terminated.
As mentioned above, both studies do not consider tightly coupled, parallel applications, so an uneven load distribution occurs only due to the termination of processes. But in parallel applications an uneven load distribution occurs also due to synchronization between the processes of an application or due to blocking communication calls, which makes the runtime behavior of a parallel application unpredictable. By the way, the only possibility for a dynamic load balancer to support existing, coarse grain, parallel applications without reimplementing them, is to use preemptive process migration.

3 Description of the hierarchical dynamic load balancer

The PaLaBer-System (**Par**allel **L**o**a**d **B**alanc**er**), which is presented in this paper, consists of three separate components: a root component, multiple levels of inner components, and leaf components on each node where application processes are running (see Fig. 1).

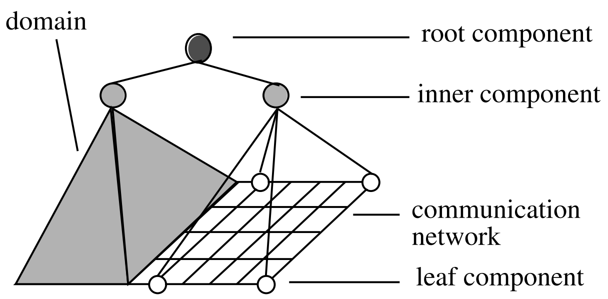


Figure 1: A hierarchically organized dynamic load balancer.

The parallel applications enter the system through the root component. The root monitors the system. If the system load falls below a certain threshold, the root tries to start an application of an appropriate size. To start an application, the root passes the process descriptors consisting of binary names, parameters, etc. to its descendants (inner components of the highest level) depending on the load state of the corresponding subtree. The inner components forward these process descriptors to their descendants depending on the load state until the descriptions reach the leaf components. The leaf components work as autonomous units. Depending on the current and the predicted future load state, a leaf component makes the decision to start an application process or to buffer it. A buffered application process is a potential candidate for a non-preemptive migration by sending the process descriptor to an underloaded node.
The main part of the work of the inner components is to mediate over- and underloaded leaf components. If an underloaded leaf component receives the address of an overloaded leaf component, it sends an *AskForWork*-message together with its own load parameters to the overloaded one. If the receiver of the *AskForWork*-message is still overloaded, it will ask an application process to migrate to the underloaded node. To avoid over- or underloading of leaf components, a leaf component performs not more than one migration at a time. For a detailed description of the migration protocol or the selection of an appropriate migration candidate see section 3.4.
In order to keep the inner components as simple as possible, the root also works as a protocol coordinator between the subtrees. Thus, the inner components do not know anything about the root or other subtrees. These components only send messages either to the next higher level of the hierarchy or to the next lower level of the hierarchy.
Before in the next sections a detailed descriptions of the load balancer components is given, a short overview of the collection and exchange of the load state in the PaLaBer-System is presented.

3.1 State information collection and exchange

Whereas gathering the load state is one of the critical tasks in central algorithms, the exchange of state information is critical in a fully distributed algorithm [3]. In a hierarchical load balancing approach, the collection of the load state is distributed among all leaf components and the load state exchange results in a natural way from the tree structure of the system

The PaLaBer-System distinguishes between three load states (see Fig. 2: (a)). A node or subtree is in one of the three load states **underloaded**, **normally loaded** or **overloaded**. In the load state **underloaded**, a node or subtree asks for more work and tries to start new processes until it does not enter the state **overloaded**. In the state **normally loaded**, a node or subtree does not ask for further work, but also tries to start new processes until it does not enter the load state **overloaded**. Finally, a node or subtree in the load state **overloaded** is willing to give up some work - preferably via non-preemptive migration, but if necessary also via preemptive migration.
In order to calculate the load state each leaf component, inner component and the root component

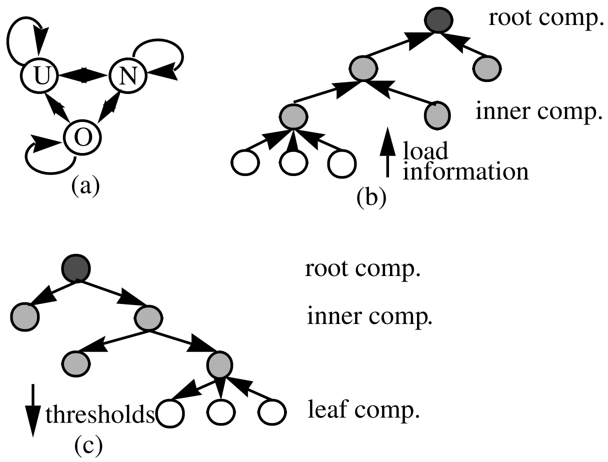


Figure 2: Load state calculation and exchange.

compare their own load values (CPU usage, memory usage, and number of processes) with some globally valid, adaptive thresholds. These thresholds are adaptive because these values depend on the global system state and some fixed constraints. The leaf components collect the load information of their node, calculate the load state and send the new load state to their parent (see Fig. 2: (b)). The parent node (inner component) then calculates the load state of the subtree (average of all descendants) and sends the new load state of the subtree to its parent (inner component or root). The root itself calculates the new load state of the whole system and also the new threshold values. If the new thresholds differ significantly from the previous ones, the root sends the new values down the control tree (see Fig. 2: (c)).

3.2 The root component

The root component is responsible for:

1. the start-up of the load balancing hierarchy,
2. the calculation of adaptive and system-wide valid thresholds,
3. passing the application processes to the lower level of the hierarchy and
4. the coordination of the information/message exchange between the subtrees of the hierarchy.

1.) In the current implementation of the PaLaBer-System, the load balancing hierarchy is static; once created the configuration is fixed for the whole run. Because most of the time critical protocols (migration, location independent communication, collection of load information) are performed at the lowest level of the hierarchy, the only criterion for building the hierarchy is the number of children of an inner component. In principle it would be also possible to make the load balancing hierarchy dynamic, but until now there was no need for it.

2.) To balance the load evenly across the whole system, thresholds are inevitable which reflects the load situation of all nodes in the system. The thresholds used by the system are adaptive, because these values are mean values over all load values of the subtrees with some lower bounds to guarantee minimal utilization. To calculate these thresholds a global operation is necessary. Considering the number of messages necessary to perform such a global operation, a tree-like communication structure is a well-known approach (compare Fig. 2 :(b) and (c)).

3.) The root monitors the system. If the system load falls below a certain threshold, the root tries to start an application of an appropriate size. Since the load situation in a large parallel supercomputing system can change rapidly and the load information of the higher level of the load balancing tree is not exact, the root component and the inner components only pass the application description (binary name, ...) down to the leaf components. To distribute the application processes among the subtrees - considering the current load state of the subtrees - the root uses a clustering algorithm to partition the application processes into clusters. Each cluster is then assigned to a subtree as a whole. The present clustering algorithm uses static placement information to assign communicating processes into one cluster which implies the same subtree ($\rightarrow$ locality of communication). On the other hand, the clustering algorithm also attempts to avoid the assignment of more than one process of an application to a node based on the following two observations:

- On the target machine Intel Paragon XP/S the communication between two processes on the same node is much more expensive ($\sim$12 times) than the inter-node communication.
- The processes of a parallel application often terminate within a small time-window. Inside the termination-window the load distribution becomes very unbalanced, which causes a lot of unnecessary load balancing overhead.

In subsection 3.5 the programming interface as well as the interface for the static placement information are described.

3.3 The inner components

An inner component is responsible for:

1. the calculation of the load state of the corresponding subtree (domain),

2. passing the application processes to the lower level of the hierarchy and
3. initiating negotiations between underloaded and overloaded leaf components.

1.) To reduce the computational work of the higher level of the hierarchy, each level of the control tree aggregates the load indicators of its children and passes the aggregate up the tree.

2.) Since the migration protocols are performed at the lowest level of the hierarchy, the inner components pass the application descriptors down the hierarchy without buffering. In order to balance the new processes evenly among the lower level, the load situation of the lower level is taken into account.

3.) Each inner component works as a "central" load balancer for its subtree. The main part of the work of an inner component is the selection of appropriate under- and overloaded leaf components and to initiate a negotiation between them. In a fully distributed load balancing scheme it is very difficult to connect underloaded and overloaded nodes, if the distance between such nodes is too large, because the information range is often limited to the direct neighbors of a node or the load information is distributed at random among the whole parallel/distributed system. In the PaLaBer-System, the inner components try to balance the load in a first step inside their own subtree. If it is not possible to reach a balanced state inside a subtree, the workload is balanced among the subtrees. In the presented system the selection of appropriate under-/overloaded nodes is made by load values/criteria and the communication relations be tween the application processes on the corresponding nodes.

3.4 The leaf components

The leaf components, which run on all nodes of the system are responsible for:

1. capturing the load information and the calculation of the load state for each node,
2. the start-up of the application processes,
3. the contact between the application processes and the load balancing hierarchy and
4. executing preemptive and non-preemptive process migration.

1.) To take full advantage of the possible parallelism of a hierarchical algorithm, the collection of the load information and the calculation of the load state are fully distributed among all leaf components. This reduces the amount of computation as well as communication needed by the higher level of the control tree. Thus, as is shown in section 6, one inner component can easily administrate more than 40 leaf components

2.) Since the load situation in a large parallel/distributed system can change rapidly, the leaf component makes the decision to start or buffer an application process autonomously. In order to do this, the component estimates the future load state based on some assumptions about the runtime behavior of the new application process and also estimates the runtime behavior of the running application processes on this node.

3.) To keep the runtime library of the PaLaBer-System simple and fast, the application processes only know their local leaf component, the remaining part of the multi-level hierarchy is transparent to them.

4.) Preemptive and non-preemptive process migration are the most time-critical parts of the whole scheme. To get full advantage of the potential parallelism in the leafs of the load balancing tree, the enforcement of the migration is made by the leaf component at the source and destination node. Because the migration of a running process is an expensive task (see section 6.), the selection of an appropriate migration candidate is the most important decision made by the load balancer. Before an overloaded leaf component initializes a migration, it performs several tests, for example:

- The time between two load balancers activities (migration, start/termination of a process) has to be at least t_1 seconds (avoid overreaction).
- An application process is only an appropriate candidate for preemptive migration, if the process has consumed more the t_2 seconds of cpu at the local node (avoid process thrashing).
- The amount of free cpu capacity at the remote node has to be significantly higher than the local amount.
- The working set of the migration candidate has to fit into memory at the remote node (avoid/minimize paging).
- The remaining load at the source node should still keep the node busy, etc..

If a leaf component has the choice between a preemptive and a non-preemptive migration, the component always prefers the cheaper non-preemptive migration.

3.5 The programming interface

It is not reasonable to require a new programming model just for the purpose of load balancing. Instead, it is necessary to support an existing programming model and therefore existing parallel applications. Since the target machine is the multicomputer Paragon XP/S [13] the programming model chosen is based on the NX-library from Intel. This library contains routines for interprocess communication and process management. Consequently, to support an existing parallel application with the PaLaBer-System, it is sufficient to replace the corresponding routines for communication and process administration and to insert a new initialization and termination routine. In Fig. 3, a selection of the implemented routines is shown.

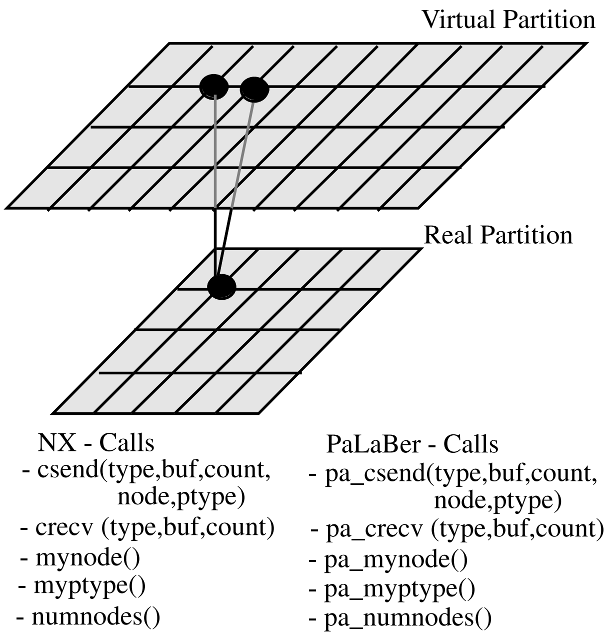


Figure 3: The programming interface.

The main difference between the NX programming model and the PaLaBer programming model is that the programmer can work with a virtual machine, i.e. a machine which is larger than the real machine. This means a programmer can choose a number of nodes which is optimal for the specific algorithm. The mapping of the processes from the virtual machine to the real one is made transparent by the hierarchical load balancer. To start an application in the PaLaBer-System, the user has to allocate a partition of the virtual machine and to place the application processes into the partition. The load balancer uses this process arrangement inside the partition as static placement information for the clustering algorithm mentioned in subsection 3.2

4 Description of the hierarchical communication manager

The communication manager (CM) is responsible for the location independent communication. This component enables the load balancer to migrate processes in a way transparent to the parallel application.

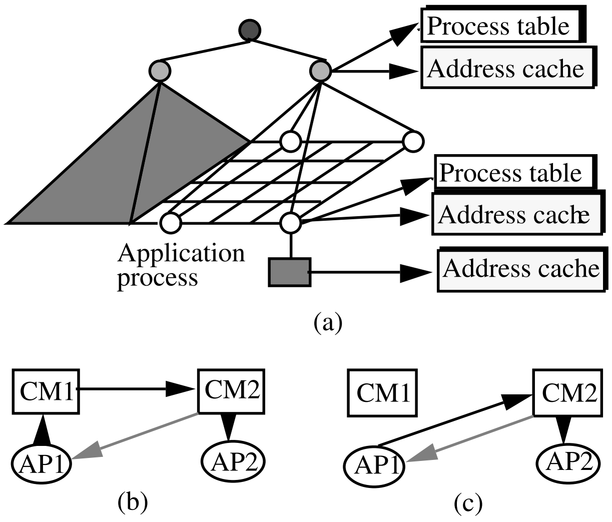


Figure 4: The communication manager.

Analogous to the load balancer, the CM is organized hierarchically (see Fig. 4: (a)), consisting of leaf components, inner components and a root component. The leaf components as well as the inner components have a process table and an address cache. The process table contains entries for all processes located at the node or subtree of this component and is also used by the corresponding load balancer component. The address cache contains the addresses of the last n communication partners of the application processes located at the node or subtree. The root does not have any tables.

The communication manager works as follows: When application process AP1 sends the first message to application process AP2, it sends the message to its local communication manager CM1 (part of the leaf component) (see Fig. 4: (b)). The local CM looks up its tables for the address of the responsible CM of AP2 and forwards the message to this CM component. CM2 then forwards the message to AP2 and sends an asynchronous acknowledgment to AP1. AP1 uses this acknowledgment to update the local address cache of its communication partners. This acknowledgment is also necessary to preserve the message order. Any further message from AP1 to AP2 is sent directly to CM2 without the local communication manager of AP1 (see Fig. 4: (c)). The whole communication protocol is handled by the communication manager and therefore transparent for the application.

Description	Number of messages
Process does not know address of the receiver	3 synchronous msg. + 1 asyncronous msg.
Process knows address of the receiver	2 synchronous msg. + 1 asynchronous msg.
Search protocol[1]	3+1 (normal msg.) +2n (send search msg. up and down to all inner components) +2m-2 (forward msg. to all subtrees)

Table 1: Number of system messages for one application message.

If a local CM cannot find the address of the receiver of a message, it buffers the message and sends a search message up the tree (see Table 1). When the local CM receives the address, it forwards the message as described above. The CM located at the root component is responsible for the coordination of search messages between subtrees.

The reason why the application processes also have an address cache of their communication partner is that this information also is available for an application process after migration.

5 Applications

So far two parallel applications have been used to evaluate the performance and behavior of the load balancer. In the following two subsections a brief overview of these applications is given.

5.1 Lattice-Boltzmann-Hydrodynamics

The first application (called Lattice [19]) is a SPMD program implementing the Lattice-Boltzmann-Method to solve a simplified kinetic equation on a two-dimensional regular grid. The aim is the simulation of fluid flow, especially convective flow. Convective flow is the movement of fluids in a gravity field caused by temperature and, therefore, density difference between different regions of the flow field. The calculation domain is partitioned along one dimension and therefore the communication structure of the SPMD application is a ring (see Fig. 5).

[1] Typical values for n are 1 or 2 and typical values for m are 1, ... , 9.

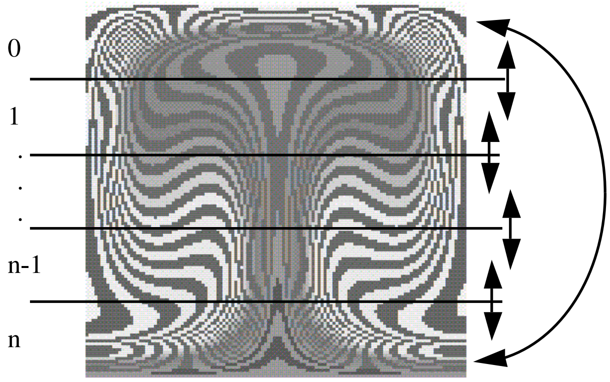


Figure 5: Partitioning and communication structure of the lattice application.

The application uses two grids with different resolutions to calculate the convective flow, the fine grid contains four times more nodes than the coarse one. Depending on the results of the previous iterations of the calculation, each process changes autonomously between the two grids. Only the two processes which perform the calculation on the partitions *0* and *n* have to work exclusively on the fine grid. Based on this adaptive runtime behavior, this application is a suitable candidate for dynamic load balancing. Furthermore, this program is an example for a tightly coupled parallel application, which means that during each iteration several messages between adjacent partitions are exchanged. The number of partitions or processes has to be a power of 2.

5.2 Parallel raytracer

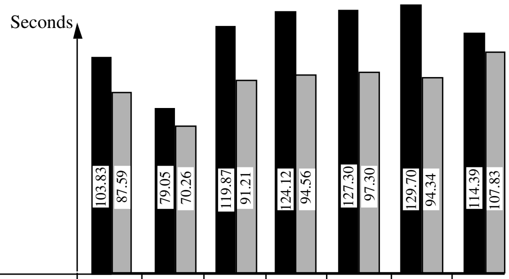
The second application is a parallel raytracer (called NoSpoort [21]). Based on the fact that visualization and animation have become more and more important during the last years, a parallel raytracer is also an interesting application for parallel and distributed systems. The raytracer is parallelized based on the Boss-/Worker-Paradigm. The application contains an application integrated dynamic load balancing mechanism. This mechanism however could be easily turned off. Each process (Worker as well as Boss) receives one partition of an equal number of rows of the picture. The assignment of partitions to processes is made by the Boss process. After each process has finished its calculation, the result is written into an output file. The time each process requires for the calculation depends on the sub-scene and varies widely between the partitions. In case the application's own dynamic load balancing mechanism is turned on, the execution works as follows: After a Worker has finished its calculations, the process asks the Boss for more work. In case the Boss has not finished its computation, the Boss sends its remaining computations to the calling Worker. Otherwise, the Boss tries to reschedule the work among the Workers.

6 Performance results

Before presenting the measurement results, a short description of the migration routine used by the PaLaBer-System is given.
To migrate processes, a remote-fork routine is used. This call generates a new process image on the destination node (approximately 440 ms[2]) but no memory pages are copied. A memory page is copied only if a page is referenced by the migrated process at the destination node. The time needed for a remote page fault is between 5 and 7 ms. The advantage of this scheme is that only the working-set is transferred from the source to the destination node. The size of the working-set for the parallel raytracer and the simulation program is approximately 300 pages. This implies that a migration needs between 1.9 and 2.5 seconds, if the whole working-set is transferred during the remaining execution of the process.
The main goal of the PaLaBer-System is to improve the throughput of a parallel/distributed supercomputing system and therefore to reduce the computational costs. So far the improvements were measured with 17 different configurations (and over 3500 runs) consisting of combinations of the two applications. In all configurations where the NoSpoort application is involved the application integrated load balancing mechanism of the raytracer is activated, therefore the runtime behavior of this application changes rapidly and unpredictably. In Fig. 6, the runtime results of a subset of seven configurations with and without the dynamic load balancer of the PaLaBer-System are shown. In all these configurations the hierarchical load balancer achieved a significant improvement.

Fig. 6 also shows, that the hierarchical load balancing scheme is highly scalable. Configuration 6 shows that one inner component can administrate up to 48 leaf components without becoming a bottleneck. In configuration 7 it is shown that the load transfer between the subtrees of the control hierarchy works fast enough to achieve significant runtime improvements also for larger systems. In this context it is worth mentioning that these good results are achieved in spite of the fact that the used process migration routine is not as fast as possible and the number of invocations of the migration routine is high. In Table 2, the average number of migration for the seven configurations is shown. The number of migrations required depends strongly on the number of leaf components (compare configurations 3, 4, 5, 6, 7) and the application behavior (compare configuration 2 and 3). Whereas in configuration 2 the applications (NoSpoort) change their runtime behavior rapidly and therefore cause a large number of migrations, the runtime behavior of the application mix in configuration 3 is much more balanced.

[2] These times are valid for Release 1.2 of the operating system OSF1/AD.



Conf. No.	1	2	3	4	5	6	7
Number of inner components	1	1	1	1	1	1	2
Number of leaf components	24	18	18	24	36	48	66
Number of Lattice (#Processes)	2x(16)		1x(16)	1x(16)	2x(16)	3x(16)	4x(16)
Number of NoSpoort (#Processes)		2x(16)	1x(8)	2x(8)	2x(8)	2x(8)	2x(18)

without dynamic load balancing   with dynamic load balancing

Figure 6: Measurements for different application configurations.

Configuation	1	2	3	4
Number of migrations	23	20	14	20

Configuation	5	6	7
Number of migrations	34	46	83

Table 2: Number of migrations.

7 Conclusions

In this paper, a new approach for dynamic load balancing in large parallel/distributed systems is proposed based on a multi-level control hierarchy. As it was shown in section 6, this scheme is highly scalable and improves the throughput of the system significantly because of the parallelism in the lower levels of the hierarchy and the possibility to aggregate information in the higher level of the control tree.
It also shows that preemptive process migration is a suitable load transfer mechanism for operating system integrated load balancing schemes as long as the granularity of the workload is not too fine compared to the time needed for a migration.
This paper also shows that it is possible to efficiently support existing parallel applications through an operating system integrated dynamic load balancing mechanism with only minor changes in the application code. This aspect is particularly important for the acceptance of a dynamic load balancing environment by the users, because most users/code owners are not willing to reimplement their applications in order to benefit from a dynamic load balancer scheme.

References

[1] I. Ahmad, A. Ghafoor, Semi-Distributed Load Balancing For Massively Parallel Multicomputer Systems, IEEE Transactions on Software Engineering, Vol. 17, No. 10, October 1991, pp. 987-1004.

[2] I. Ahmad, A. Ghafoor, G. Fox, Hierarchical Scheduling of Dynamic Parallel Computations on Hypercube Multicomputers, Journal of Parallel and Distributed Computing 20, 1994, pp. 317-329.

[3] A. Barak, A. Shiloh, A Distributed Load-balancing Policy for a Multicomputer, Software-Practice and Experience, Vol. 15, No. 9, September 1985, pp. 901-913.

[4] W. Becker, R. Pollak, Efficiency of Server Task Queueing for Dynamic Load Balancing, Technical Report No. 1994/9, University of Stuttgart, Institute of Parallel and Distributed High-Performance Systems (IPVR), 1994.

[5] S.H. Bokhari, On the Mapping Problem, IEEE Transactions on Computers, Vol. C-30, No. 3, March 1981, pp. 207-214.

[6] C. Cap, V. Strumpen, The PARFORM - A High Performance Platform for Parallel Computing in a Distributed Workstation Environment, Technical Report, Institut of Computer Science, University of Zürich, June 1992.

[7] J.Casas, R. Konuru, S. W. Otto, R. Prouty, J. Walpole, Adaptive Load Migration System for PVM, Proceedings Supercomputing '94, Nov. 1994, Washington D.C., pp. 390-399.

[8] T. Casavant, J. Kuhl, A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems, IEEE Transactions on Software Engineering, Vol. 14, No. 2, February 1988, pp. 141-154.

[9] G. Cybenko, Dynamic Load Balancing for Distributed Memory Multiprocessors, Journal of Parallel and Distributed Computing 7,(1989) 279-301.

[10] F. Douglis, J. Ousterhout, Transparent Process Migration: Design Alternatives and the Sprite Implementation, Software-Practice and Experience, Vol. 21, No. 8, 1991, pp. 757-785.

[11] D.L. Eager, E.D. Lasowska, J. Zahorjan, A Comparison of Receiver-Initiated and Sender-Initiated Adaptive Load Sharing, Performance Evaluation, Vol. 6, March 1986, pp. 53-68.

[12] D.L. Eager, E.D. Lazowska, J. Zahorjan, The Limited Performance Benefits of Migrating Active Processes for Load Sharing, ACM SIGMETRICS, Performance Evaluation Review, May 1988, pp. 63-72.

[13] Intel Corporation, ParagonTM User's Guide, Order Number 312489-003, 1994.

[14] F. Lin, R. Keller, The Gradient Model Load Balancing Method, IEEE Transactions on Software Engineering, Vol. 13, No. 1, January 1987, pp. 32-38.

[15] P. Krueger, M. Livny, A Comparison of Preemptive and Non-Preemptive Load Distribution, The 8th International Conference on Distributed Computing Systems, June 1988, pp. 123-130.

[16] W.E. Leland, T.J. Ott, Load-balancing Heuristics and Process Behavior, ACM SIGMETRICS Performance Evaluation Review, Vol. 14, No. 1, May 1986, pp. 54-69.

[17] H. Lin, C. Raghavendra, A Dynamic Load-Balancing Policy With a Central Job Dispatcher (LBC), IEEE Transactions on Software Engineering, Vol. 18, No. 2, February 1992, pp. 148-158.

[18] R. Lüling, B. Monien, F. Ramme, Load Balancing in Large Networks: A Comparative Study, Proceedings of the 3rd. IEEE Symposium on Parallel and Distributed Processing, Dallas 1991

[19] G. Pätzold, Parallel Lattice-Boltzmann Simulation of Three-Dimensional Convective Flow, Proc. of the Conference on Parallel Computational Fluid Dynamics 1993, Paris, France, 10.-12 May 1993.

[20] A. Reuter, U. Geuder, M. Härdner, B. Wörner, R. Zink, GRIDS User's Guide, Version 0.1, Report No. 4/93, University of Stuttgart, Institute of Parallel and Distributed High-Performance System (IPVR), 1993.

[21] J.P. Richter, Parallelisierung des Raytracing Verfahrens, Studienarbeit im Fach Informatik, Lehrstuhl für Informatik VII, Universität Erlangen, 1992 (in german).

[22] A.M. van Tilborg, A.D. Wittie, Wave Scheduling - Decentralized Scheduling of Task Forces in Multicomputers, IEEE Transactions on Computers, Vol. C-33, No. 9, September 1984.