Scheduling and Migration Strategies for Parallel Applications on Distributed Systems

Von dem Fachbereich für Mathematik und Informatik
der Technischen Universität Carolo-Wilhelmina
zu Braunschweig

von
Bettina Schnor
aus Braunschweig

angenommene Habilitationsschrift
zur Erlangung der Venia legendi
für das Lehrgebiet
Informatik

Braunschweig
14.10.97

Abstract

This work deals with resource management for parallel applications in distributed systems. Its main contribution is to develop a new resource management concept for parallel applications on heterogeneous systems.

Whilst scheduling parallel jobs in multiprocessor systems has been a research topic for a long time, interest in scheduling and load balancing strategies for parallel applications on workstation clusters has recently grown. Scheduling of parallel applications is divided into two steps. First, the application which shall be scheduled next has to be chosen. Second, the nodes which shall be allocated by the application have to be selected. The last step is called *mapping*.

First, we explain the characteristics of parallel applications which are important for resource management. Further, we present different workload models. We give a classification of group scheduling strategies and compare the most popular resource management systems for parallel applications at the moment.

We present a time-sharing algorithm called LST which is based upon Largest-Size scheduling. It differs from the prominent *matrix algorithm*, since it uses no fixed placements of jobs. Instead of this, it makes use of a migration facility if mapping conflicts occur. We investigate the influence of migration costs on the performance of LST. In a simulation, LST performs better than different space-sharing scheduling strategies.

For heterogeneous systems, the Shortest–Expected–Delay–Mapping (SED) is presented. The advantage of using SED is that the heterogeneity of the system is transparent for the programmer. This is done by managing "virtual homogeneous" nodes which makes the development of a parallel application much easier. The concept of mapping states and mapping state diagrams is introduced to investigate the behavior of different SED algorithms.

Further, we present a new dynamic load balancing strategy called Dynamic-SED which is based upon the SED mapping strategy. Dynamic-SED not only looks at the current delay of the machines, but also at the currently free memory and the number of interactive users. It presents a new approach to achieve a co-existence between parallel applications and interactive users.

The presented scheduling and load balancing strategies make use of a migration facility. Since migration is an operation which consumes processor time and network capacity, the overhead caused by migration is one of the main aspects in the investigation of the presented algorithms. Further, a number of 'migration anomalies' are defined and it is checked whether these migration anomalies are true for Dynamic-SED or not.

The presented trace-driven simulation results show that migration is a useful and efficient mechanism to support parallel applications on workstation clusters.

Thanks

Lots of people have supported this work in one way or another. Foremost, I want to thank Professor Dr. Langendörfer for his encouragement and who made this work possible.

Most of this work was performed during my time at the Institute of Operating Systems at the TU Braunschweig under the head of Professor Langendörfer within the research group "Load Balancing and Failure Transparency". I must thank all my colleagues which have influenced this work with their discussions. In particular, I am indebted to Stefan Petri who has designed and implemented the PBEAM migration facility and shown that transparent migration of parallel processes in workstation clusters is feasible.

I also want to thank the students who did their student or master thesis within the research group which contribute to this work. The ones who shared my 'enthusiasm' for group scheduling and load balancing are Marc Gehrke, Sven Kühne, Reinhard Oleyniczak, Susann Sattler, and Stefan Stille.

Further, I want to thank Christopher Ford, Martin Hauner, Holger Schellwatt, and Michael Schmitt for their proofreading.

Braunschweig, December 1996 Bettina Schnor

Contents

Chapter 1

Introduction

The goal of this work is to propose a new approach for resource management support for parallel applications in heterogeneous systems. Resources may be computing nodes, memory, and network capacity. The resource management of computing nodes for parallel applications is divided into two steps. First, the application which shall be scheduled next has to be chosen. Second, the nodes which shall be allocated by the application have to be selected. The last step is called *mapping*.

In this chapter, we give a motivation for our work and describe our "working environment", i.e., we will answer the following questions:

- Which characteristics of parallel applications are important for resource management?
- Which kind of workloads do we expect?
- Which performance metrics are suited for the comparison of scheduling strategies?
- Which types of scheduling strategies for parallel applications exist?
- What is the state-of-the-art?
- What is our approach?

1.1 Motivation

Many numerically intensive "Grand Challenge" applications can be solved using parallel algorithms. Examples of applications are computational models of global climate exchange, unsteady flow around airfoils, galaxy formation, and nuclear explosions.

A parallel application consists of a group of communicating processes which have to be assigned to the processors of the given parallel or distributed system.

A 2-dimensional multi-grid application for example may consist of m^2 processes. Each process calculates the points of one square and after each calculation step it sends its results to the processes which are responsible for the neighboring squares. So the processes are all running in parallel with high communication after each calculation step.

The benefits of parallel job scheduling are as follows:

- Load balancing, since each process is assigned to another processor.
- Speedup through parallel processing.
- Reducing communication costs in the case of fine-grain interactions [2].

Since the mid 80s, the interest in scheduling disciplines for parallel applications increased due to the growing interest in parallel and distributed systems. In the case of distributed systems, the given network topology is of major interest. Hence, the development of scheduling disciplines is hardware driven in the sense that much work was published for hypercubes when hypercubes became popular. Influenced by the transputer topology and the Intel/Paragon machine, mesh architectures became also popular.

When software like Linda [52, 18], PVM [150, 50, 51] and P4 [15] became available, workstation clusters were conquered as a platform for parallel computing. The first task was to build message passing systems which offer a more comfortable interface to the underlying transport layer. But when systems like PVM became more and more popular, the need for efficient resource management increased. A sign that parallel job scheduling is a main issue nowadays was the establishing of a workshop at the IPPS'95 and IPPS'96 about this topic.

This work deals with resource management for parallel applications in workstation clusters. Workstation clusters as they are typical in use in the mid 90s consist of a powerful workstation with at least 32 MByte memory which are connected by an Ethernet.

While the types of machines will change and memory space may be no problem in future, one characteristic will be unchanged: That there are a number of connected machines which differ in speed and which can be used for parallel processing.

1.2 Classification of Scheduling Strategies

The research of scheduling techniques falls into two separate classes, *deterministic* and *dynamic scheduling*.

Deterministic scheduling uses information about the service demands and the size of the applications for constructing a schedule.

Definition: The number of processes which are contained within a parallel application is called the *degree of parallelism* or *size* of the application. The *configurational size* is the number of processes which are created when the job starts.

Deterministic scheduling deals with the classical *mapping problem*. Given parallel jobs $J_1, J_2, ..., J_k$ with service time demands $t_1, t_2, ..., t_k$ and size $n_1, n_2, ..., n_k$, the task is to construct a schedule which minimizes the overall finishing time, i.e., the time when all jobs are finished.

The mapping problem is known to be NP–complete [11]. Deterministic scheduling is used in some batch systems [93, 135].

We use the term *dynamic scheduling* when parallel jobs can be dynamically submitted to the system by the users and nothing is known about the execution time in advance. Here, we are only interested in *group scheduling strategies*.

Definition: A scheduling strategy which schedules all processes of a parallel job simultaneously is called a *group scheduling discipline*.

Dynamical group scheduling strategies are classified into space–sharing and time–sharing disciplines [39]:

Definition: A *space–sharing* discipline is a non–preemptive discipline which partitions the processors among different parallel jobs. A *time–sharing* discipline uses some form of round–robin–scheduling between the parallel jobs.

Time–Sharing disciplines for parallel jobs have first been investigated by Ousterhout who introduced *co-scheduling* in the context of Medusa, an operating system for the Cm* multimicroprocessor [112]. Feitelson and Rudolph have proposed time–sharing scheduling techniques for large distributed systems and have introduced the terminology *gang scheduling* [36].

Definition: A time–sharing strategy is a *gang scheduling* discipline if the processes of parallel job are grouped together into a gang. All the processes in a gang are always scheduled to execute simultaneously on distinct processing elements, using a one-to-one-mapping. All processes in a gang are preempted and rescheduled at the same time (multi-context-switch).

Gang scheduling is necessary to allow efficient execution of communication-intensive applications, combined with reasonable response time and throughput on multiprocessor architectures. The benefits of gang scheduling are discussed in [37] and [143].

Gang scheduling is supported on the Connection Machine CM-5 [153], Intel Paragon [32], the Meiko CS-2, and multiprocessor SGI workstations [8].

Hence, a non–preemptive group scheduling algorithm is a space–sharing algorithm and the preemptive group scheduling algorithms are gang scheduling algorithms.

Space sharing disciplines are divided into fixed, variable, adaptive, and dynamic partitioning schemes [35, 144].

Definition: A *fixed partitioning* scheme divides the processing elements into predifined partitions which are set by the system administrator. A *variable partitioning* scheme sets the number of allocated nodes according to the request of the parallel job. An *adaptive partitioning* scheme sets the number of nodes allocated by an application according to the request and the system load at the time of its arrival. A *dynamic partitioning* scheme is an adaptive partitioning scheme which changes the partition size allocated to jobs at runtime, to reflect changes in job requirements and system load.

Fixed partitioning is used on several parallel computers when for example partitions for batch and interactive workload can be defined [157]. The major disadvantage of fixed partitioning is that parallel jobs have to allocate more jobs as they need and therefore, the *internal processor fragmentation* is high.

The advantage of variable partitioning is that internal fragmentation is avoided. On the other hand, *external processor fragmentation* may occur when the available free processing elements are insufficient to satisfy the request of any submitted job.

Adaptive partitioning schemes reduce the number of allocated nodes when the system load increases. This reduces external fragmentation since jobs tend to make earlier use of the remaining free processing elements. Adaptive partitioning algorithms are investigated in [54, 123, 113].

Dynamic partitioning is the most flexible partitioning scheme and has repeatedly been shown to be superior to other schemes [56, 101]. However, dynamic partitioning requires application-level support. This may be simple in the case of master-worker-style applications, but for Computational Fluid Dynamics (CFD) applications for example there will occur a non-neglectable overhead due to redistribution of data.

Squillante has investigated dynamic partitioning schemes [144]. He states that the benefits of dynamic partitioning depend on the application workload and the reconfiguration costs.

Examples for load balancing facilities which support adaptive or dynamic load balancing are CARMI/WoDi [120] (see section 1.6.4), PARFORM [17] and Piranha [53].

Konura, Moreira, and Naik have implemented dynamic partitioning for CFD applications [78]. They report that only tens of lines had to be modified within the application code. Jobs are redistributed from 8 to 16 nodes or from 16 to 8 nodes for example. The time measured for the redistribution of the data was only a few seconds which is a promising result. Further, they used the measured reconfiguration times as input data for a simulation of dynamic partitioning for different workloads.

The acceptance of dynamic partitioning has still to be proven. Since the user has to change her code, it has to be beneficial for her. This is obvious, if applications can get more resources. But why should an application release resources voluntarily? Here, the individual optimum is in conflict with the social optimum (see section 1.5).

Further, adaptive and dynamic partitioning schemes can be combined with migrating and non-migrating strategies.

Definition: The relocating of an already running process from one processing element to another is called *Migration*. The state of the process is saved into a so-called *checkpoint*. The checkpoint is transfered to the new location where a new process is created from the checkpoint.

The state of a process can be divided into internal and external state [116]. The internal state of a process consists of register contents (including stack pointer and program counter) and contents of the address space (typically text, data, and stack segments). The external state consists of I/O channels (open file descriptors, sockets), signal handlers, timers, parent-child relations, and resource usage statistics.

Migration is motivated by load balancing and failure transparency. Load balancing may be implemented on user level. Then the application decides to migrate when it is informed about substantial load changes. Dynamic load balancing may further be supported on system level by a migration facility which migrates processes when faster machines become available.

The presented classification of partitioning schemes is accepted within the scientific community which is interested in group scheduling. There exists also a common classification of load balancing strategies in distributed system. Applied to parallel applications, this leads to the following characterization:

Definition:

1. *Static load balancing*: On arrival a parallel job is mapped onto a subset of machines and runs there up to completion. The machines may be specified within a configuration file.
2. *Adaptive load balancing*: When a parallel job is assigned to a subset of machines, this subset depends on the current load situation. Further, the number of allocated machines depends on the current load situation.
3. *Dynamic load balancing*: The mapping strategy regards the current load situation like in the case of adaptive load balancing, but reacts dynamically onto load changes.

Hence, dynamic partitioning is an example for a dynamic load balancing strategy.

1.3 Classification of Applications

The classification of parallel applications can be done under different aspects.

The *Basel Algorithm Classification Scheme* (BACS) is an approach to get a classification of parallel applications [14]. The goal is to achieve portability and algorithmic re-usability, i.e., to support the most common parallel computation schemes on the most widely distributed virtual machine models and computation languages.

BACS characterizes the algorithmic behaviour of the application like for example its algorithmic topology, its interaction mechanism (message passing, tuple space, signals, ...), and data distribution.

Since the intention of this work is resource management, we will give a classification of parallel applications which concentrates on the relevant aspects for resource management. These are

- computing demands,
- memory demands,
- communication and synchronization model.

In resource management systems like DQS [30], LoadLeveler [67], CARMI [120] etc. it is common that the user has to specify the amount of memory the application needs at minimum on every host. This is done to avoid the negative effect of swapping when memory becomes scarce.

Parallel applications follow different communication models and synchronization models. We distinguish between applications which follow the *pool-of-tasks* model and *fully distributed* applications. We will define these characteristics in the following.

1.3.1 Pool of Task

In the pool-of-tasks model there is a pool of tasks which can be executed in parallel. There exists a central coordinator process, the so-called master, which distributes the tasks dynamically to the worker processes. This type of application is also often called the *master-worker* model.

A simple example is the calculation of the Mandelbrot set where the master partitions the problem in different rectangulars and distributes them among the worker processes.

Load balancing for master-worker applications can be easily implemented on application level. Since the master always has the control over the application, he can distribute the work according to the current load situation and react easily onto load changes. When new machines become available, the master may start worker processes on these machines and distribute tasks to them, or in the case machines are overloaded, he can stop using them. Hence, dynamic partitioning is easily done for master-worker application, since it is only a special case of the load balancing strategy.

Pruyne and Livny have described a resource management system named CARMI which supports dynamic partitioning [120]. CARMI is used together with WoDi which provides an interface for writing master-worker programs (see section 1.6.4).

1.3.2 Fully Distributed Application

in the case of a *fully distributed application* there exists no special coordinator process.

Examples of fully distributed applications are numerical applications like multigrid applications. Grid applications play an essential role in Scientific Supercomputing. Typical applications come from the field of Computational Fluid Dynamics.

The problem size is partitioned into equal sized meshes, typically in the 2- or 3-dimensional space. The parallel processes solve the problem within several iterations. After each iteration 'neighboring' processes have to exchange intermediate results.

A description of algorithms is given by Meynen and Wriggers [102], implementations of multigrid techniques on mesh architectures are discussed in [155, 4, 102]

Further, we distinguish between *balanced* and *unbalanced* applications. In the case of a balanced application, the computing demands of all processes are nearly the same. In the case of an unbalanced application, the execution demands of the processes may differ.

A parallel program may for example simulate the deformation of a tin. The tin is crushed to something like a disk. When the pressure and tension rises, the computing demand of each step will rise also. But this will happen for almost all regions equally because of the symmetry of the tin. This means that, while the computing demands change during execution, the application will stay almost balanced.

For other input data, i.e., asymmetric objects, the application may behave as if unbalanced. This possible behaviour of parallel applications was the trigger for lots of work on application-level load balancing.

Definition: Let $\mathcal{P} = \{p_1, ..., p_n\}$ be the set of parallel processes which belong to the parallel application J. The relation $\longleftrightarrow^k$ ("exchanges results") on $\mathcal{P}$ is defined as follows:

$$p_i \overset{k}{\longleftrightarrow} p_j :\Longleftrightarrow p_i \text{ and } p_j \text{ exchange intermediate results between the } k\text{-th and } (k+1)\text{-th computation step}.$$

Definition: Let $\mathcal{P} = \{p_1, ..., p_n\}$ be the set of parallel processes which belong to the parallel application J. We say that J follows the *strong synchronization model* if the transitive closure of the relation $\overset{k}{\longleftrightarrow}$ is $\mathcal{P}$ for all k.

In the case of applications which follow the strong synchronization model, an unbalanced load situation may slow down the whole parallel application.

An example of a fully distributed application with strong synchronization is a multigrid application. If we consider for example a 2-dimensional mesh, each node exchanges results with its four neighbors after each iteration. Let us assume that one node A is overloaded and hence it needs more time for an iteration than the other nodes. This will slow down all 4 neighbors A_1, A_2, A_3, A_4, since they cannot carry on their work without the intermediate results from A. The delay of A_1, A_2, A_3, A_4 will also slow down their other 8 neighbors in the next step. Hence, the delay of one node will be propagated like a wave. Finally, the whole application is slowed down by a single node.

Scheduling strategies which address fully distributed applications with strong synchronization are presented in chapter 4.

Another criterion for parallel applications is their *granularity* which gives the mean number of operations between communication. A *fine grain* application has only few operations between communication, and a *coarse grain* application a higher number of operations. Since fine grain applications are more communication-intensive, they need a high bandwidth and are not suited for workstation clusters for example.

1.4 Workload Characteristics

Since the dawn of (computer) time, it has been recognized that performance analysis and modeling of computer systems hinges on using a representative workload (Feitelson/Nitzberg)

A most common technique to test the performance of scheduling disciplines is simulation. For the interpretation of a simulation study, two aspects are important: first, the simulation software has to be correct, and second, the input workload should be realistic.

1.4.1 Workload Models

It is still an open question whether there is any correlation between the size of a parallel job and its service demands.

The *Fixed–Work–Model* assumes that the work done by a job is fixed, and its execution time depends on the degree of parallelism when it is executed. This model is the base for Amdahl's law [5]. In the optimal case, there will be a speed up of k when the application runs on k processors. In this model the runtime is negatively correlated to the job size.

Gustafson [57] has introduced the *Fixed–Time–Model*. Fixed–Time–applications have to be solved in restricted time. An example is weather forecast. Here the problem is sized up by taking the finest grid of measure points which can be solved within the given time and with the given degree of parallelism. Hence, the execution time and degree of parallelism of the application are independent. This leads to an uncorrelated workload.

On a distributed memory computer, the amount of available memory increases with the number of used nodes. Hence, the problem size may be increased to fill the available memory. This gives the *Memory-Bound-Model* [149] where size and service demands of an application are positively correlated.

1.4.2 Variable–Size–Model and Fixed–Size–Model

Beside mean inter-arrival time and mean service demands, the degree of parallelism or so–called size of the parallel application has to be specified.

The user knows the maximum degree of parallelism of her application and normally specifies this value, when she submits the application to the scheduler. If the user wants to allocate exactly as many processors as specified, we call it a *fixed–size–model*.

But most applications, like partial differential equation solvers for example, can have a variable size and only the maximum degree of parallelism is fixed. Here, speedup curve and efficiency curve show the optimal degree of parallelism, which normally depends on the implementation and hardware.

This *variable–size–model* is a generalization of the fixed–size–model, which is included by specifying $minsize = maxsize$. A lower bound for $minsize$ would be one, and an upper bound for $maxsize$ would be the total number of workstations. In our model, the size of an application is fixed when the application is started.

In heterogeneous systems, the variable–size–model may lead to a better performance compared to the fixed-size–model. Consider for example a system of 10 workstations where 4 machines are 2 times faster than the others. In spite of using the 6 slower machines it is of more benefit to run the application on the 4 faster machines. In this case, it is advisable that the user specifies the minimum and maximum degree of parallelism which is suitable for her application. This approach is more flexible and can make better use of system resources.

When the application follows the variable–size–model, the variable, adaptive, or dynamic partitioning scheme can be used. In the case of fixed-size-applications, only variable partitioning is possible.

In [44], the term *rigid job* is used for a parallel application with fixed size, and the term *moldable job* for an application with variable size.

1.4.3 Monitoring Studies

Workload characteristics of uniprocessor machines are meanwhile well–known, but there is still little data available about the workload of parallel systems. In this chapter we summarize the results of monitoring studies.

job size	average runtime	standard deviation	coefficient of variation
1	140.6	736.0	5.2
2	714.2	2422.3	3.4
4	1116.7	4171.5	3.7
8	705.2	2344.3	3.3
16	569.3	1970.9	3.5
32	1305.3	3311.6	2.5
64	1250.8	4155.8	1.8
128	3280.1	4408.1	1.3

Table 1.1: Runtime statistics at NASA Ames Research Center.

Feitelson and Nitzberg present the results of an accounting study on the 128-node iPSC/860 hypercube located at the NASA Ames Research Center [34]. The study was done during the fourth quarter of 1993. Mapping on the iPSC/860 is done in a space-sharing mode where jobs are mapped exclusively onto subcubes. Hence, the partition size, i.e. the size of the parallel applications, is always a power of 2 with 128 nodes at maximum.

Among the parallel jobs, a more-or-less uniform distribution across the possible parallel job sizes is observed. The high number of sequential processes is due to Unix commands which were run by the system administrators mostly to check system functionality.

The percentage of jobs with size 128 was less than the other possible sizes. Further, these biggest possible jobs, which allocate the whole hypercube exclusively, were observed at night or at the weekend. This leads to the assumption that only the actual allocated number of processors was monitored but not the possible maximum parallel job size.

When the job size is set into relation to the consumed processor resources measured in node-seconds, the large jobs, with 32, 64, and 128 nodes, have consumed the most computing time in roughly equal portions. This means that the smaller number of 128-node jobs have consumed more processor resources.

The mean runtime, standard deviation, and coefficient of variation, i.e., the ratio of the standard deviation to the mean, is given in table 1.1. The coefficient of variation is always larger than 1. This can be modeled by hyper-exponentially distributed service demands. Further, it is observed that runtime and job size are positively correlated which is a hint that the memory-bound-model is appropriate.

A study of 23 CFD applications also reports that large jobs tend to run longer and emphasizes the memory-bound-model [106].

Feitelson reports from trace data on a 400-node Paragon at the San Diego Computer Center and on a IBM SP1 at the Argonne National Lab that there is only “a weak tendency for larger jobs to have a higher runtime” [43].

1.5 Performance Metrics

A scheduling discipline can be evaluated from different points of view. The user is interested in a fast response time. Hence, a measure for the *individual optimum* is the *speedup* of the application.

Definition: The *speedup* $S(p)$ is defined as

$$S(p) = \frac{T(1)}{T(p)}$$

where $T(k)$ gives the runtime of the application on k processors.

The *efficiency* $E(p)$ is defined as

$$E(p) = \frac{S(p)}{p} = \frac{T(1)}{p \cdot T(p)}.$$

In general, measurements show a common behaviour of parallel applications. While adding more processing elements increases the speedup, the efficiency drops. This is due to the rising communication overhead.

In most cases, the user will only be interested in speedup and not in the efficiency of its implementation. While nearly the same execution time may be possible with less nodes, the user will typically use as much resources as possible.

For example, Ghosal et al. [54] suppose that the efficiency function of an application is given in advance and use this information in their scheduling decisions. But this is a very strong requirement which would be hard to fulfill in most environments.

Speedup and efficiency are defined for homogeneous systems. In a heterogeneous system it has to be defined of what kind the reference machine is. Usually, $T(1)$ gives the execution time on the fastest machine.

From the point of view of the user, the speedup shall be maximized. But from the point of view of the system administration, an efficient use of the system is more beneficial. i.e., setting the speedup in relation to the costs. Hence, the use of the product of these two metrics as a target function is proposed in [46, 54].

A popular *social metric* for dynamic scheduling is the *mean residence time*. The residence or response time of a sequential job is the sum of waiting and execution time. In the case of a parallel application this is the time from submitting the job until the last process of the application has finished.

In the case of deterministic scheduling, the makespan is used as a measure for the performance. Given parallel jobs $J_1, J_2, ..., J_n$, the makespan is the time from starting the first job up to the completion time of the last job. Another possible criterion is for example the percentage of processor fragmentation.

1.6 Resource Management Systems

A resource management system manages the resources, i.e. processing nodes and memory, of a distributed system. Similarly to the field of classical scheduling, there exist various criteria to decide what is a good resource management strategy. Possible goals are high-throughput and/or a minimum mean residence time and/or preference of special application classes. An important property of a resource management system is fairness, i.e. no application may wait forever for getting computing resources.

A resource management system has four components:

1. **Scheduler:**
   The scheduling component decides, *when* a job is scheduled.
   All scheduling techniques fall into one of two classes: space-sharing or time-sharing strategies.

2. **Mapping Component:**
   The mapping component decides, *where* a job is started, i.e. on which nodes.

3. **Load Information Component:**
   The load information component gathers the current load statistics.

4. **Load Balancing Component:**
   The load balancing component implements the load balancing strategy. Further, a load balancing mechanism is necessary: A remote execution facility in case of adaptive load balancing and a migration mechanism in case of dynamic load balancing.

Some strategies combine the scheduling and mapping strategy. Hence, these systems possess only a scheduling component.

There are several resource management systems in use. We will discuss the most popular ones which support parallel applications. The classification criteria are:

1. Which architecture is supported?

2. Which types of applications are supported?

   Is there special support for the pool-of-tasks or the strong- synchronization-model? Is the fixed-size or variable–size–model supported?

3. Is the scheduling strategy

   (a) space-sharing or time–sharing?
   (b) fixed, variable, adaptive, or dynamic?
   (c) migrating or non-migrating?
   (d) Is the memory demand of the application considered in the scheduling strategy?

4. Load management:

   (a) Which load metric is used?
   (b) Are heterogeneous systems supported?
   (c) Are interactive users considered?

5. Further notes.

1.6.1 NQS/SDSC

The Network Queueing System (NQS) is a batch scheduling facility developed at NASA Ames. NQS was originally designed for throughput-oriented computation with sequential jobs. It has been extended to support parallel applications. NQS is supported by various vendors of parallel machines and is in use among many large UNIX system sites. The POSIX Standard IEEE STd 1003.2d-1994 defines a standard batch and general queuing environment which is based on NQS.

When a job is submitted, its limit on CPU and memory usage has to be specified. Jobs are classified according to these limits in different classes. Jobs that exceed their time limit are terminated.

Here, we describe an extension of NQS for the Intel Paragon parallel computer [32] from the San Diego Supercomputer Center (SDSC) [157]. The SDSC batch scheduler was developed for a 416-node Intel Paragon system at SDSC. The software has been adopted by Intel and is part of the software release for the Paragon machine.

1. SDSC is developed for the Intel Paragon parallel computer. The network architecture of the Paragon is a 2-dimensional mesh. The nodes are managed in different partitions: A service partition for operating system services, such as the file server, and also interactive use, and a compute partition.

2. SDSC uses the fixed–size–model.

3. The scheduling strategy is a priority based space-sharing discipline with variable partitioning.

   (a) While the Intel Paragon architecture offers a gang scheduling mechanism, SDSC favors space-sharing [157]: “However, time-sharing of jobs places a heavy burden on the paging system and is not practical at this time due to I/O band-width performance and disk space limitations.”

   (b) SDSC uses priority based scheduling combined with non-contiguous node allocation.

       NQS/SDSC manages jobs in different queues according to their resource requirements. Jobs have to specify whether they need ”fat” nodes with 32 MBytes or less. Further, they have to specify their computing demands. SDSC distinguishes three classes: Short jobs with a time limit of 1 hour, medium jobs with a time limit of 4 hours, and long jobs with a time limit of 12 hours.

       The queues have different priorities. For example, the queue priority of the long resp. short jobs is 10 resp. 12. Within a queue, the scheduling strategy is simple FIFO.

To avoid starvation of processes in lower priority queues, SDSC uses aging of processes. The job priority is calculated as follows:

$$job\_priority \; = \; queue\_priority + age\_factor \star time,$$

where time is the time in hours the job has been waiting in the queue and age_factor is a constant factor configurable by the system administrator. With an age factor of 0.1, it will take a long job 20 hours waiting in the queue to get the same high priority as a short job.

When a job is chosen for scheduling and enough nodes are available, it will be started. Otherwise, it has to wait. To avoid starvation of bigger jobs, SDSC uses a maximum priority $block\_priority$. When the job has reached the $block\_priority$, scheduling of new jobs will be blocked until there are sufficient free nodes for this job. For example, with $block\_priority = 15$ and an age factor of 0.1, a long process will wait 50 hours in the queue before blocking, i.e., reservation of nodes, will start.

Another crucial point in scheduling parallel applications on distributed memory machines is the mapping strategy. A typical mapping strategy for mesh architectures is mapping onto rectangulars. The Paragon architecture allows non-contiguous node allocation which means that the nodes of an application do not have to be direct neighbors which makes the mapping strategy more flexible and increases the node utilization. SDSC uses a modified 2-D Buddy System [157].

(c) Checkpointing and migration are not supported.

(d) The user specifies the minimum amount of memory each node should have. This is used in determining the job queue of the application.

4. Load management:

   (a) Nodes are dedicated in space-sharing mode to applications.

   (b) NQS/SDSC manages machines which differ for example in their memory resources in different queues. NQS/SDSC is also aware of single and two processor nodes.

   (c) There exists a special partition for interactive jobs.

1.6.2 PVM

A popular tool to implement parallel applications on workstation clusters is PVM (*Parallel Virtual Machine*) [150]. PVM is a message passing system which supports parallel applications. PVM is no batch scheduler. PVM creates a user-specific parallel virtual machine. The user has to specify the hosts which shall belong to her parallel machine in a configuration file.

1. One of the main reasons why PVM became the most popular platform for parallel applications is its availability on a wide range of heterogeneous platforms from workstations up to supercomputers like the Intel Paragon, KSR1, Cray-2, and Thinking Machines CM-5.

2. PVM applications specify a fixed size.

3. (a) PVM lacks resource management. The hosts which are specified in the configuration file belong to the virtual parallel machine. The processes of an application are mapped in a round-robin fashion onto the nodes of its parallel virtual machine.

   If two users start applications in parallel and their configuration files have common entries, then their applications may run concurrently on some machines without knowing from each other.

   (b) PVM uses variable partitioning.

   (c) Checkpointing and migration are not supported.

   (d) Memory demands are not considered.

4. Load management:

   (a) The current load situation on the machines is not considered.

   (b) The heterogeneity of the machines is not considered in the mapping strategy.

   (c) No special actions for interactive users.

Since PVM offers no resource management, different groups have started to add resource management for PVM.

Humphres has implemented a resource management component for PVM [65]. The processes are started on the n fastest machines according to the load index. The used load index is the idle time of the machines which is not very meaningful in heterogeneous systems.

Wilhelms has implemented *PVM+* [161] which uses the idle time and the CPU load average of the machines to estimate the current load situation. PVM+ maps onto the n fastest machines. Interactive users are considered when the job is started. The estimation of load caused by interactive users is based on a user classification. The classification itself and the assignment of users to special classes does not seem very practical. PVM+ does not act on load changes during execution.

Pruyne and Livny have implemented the Condor Application Resource Management Interface (CARMI) for PVM applications (see section 1.6.4).

In [33], a *general resource manager* (GRM) for PVM is proposed. GRM will be distributed with PVM 3.4. GRM uses a load metric which is similar to our delay factor. It differs in the calculation of the speed factors and the load average. Typically for PVM, resource management is done for each application separately. The speed factors are defined to be the weighted mean of integer and floating point performance. The weights are user configurable. This gives the user the opportunity to adapt the speed factor to the demands of its application. On the other hand, this approach is less transparent, since the task to find the optimal values is left to each user. The weights of the load average will be also user configurable which is less meaningful. These are parameters of the load management which are independent from the application.

In [33], four scheduling disciplines are proposed for the GRM:

1. The originally Round-Robin scheduling,

2. using the least loaded machines where only the load average is used as load metric,
3. using the least loaded machines compared by their delay factors with at most one PVM task per host,
4. using the least loaded machines compared by their delay factors (multiple tasks may run on a single host).

The drawback of the first discipline is that it is unaware of the current load situation. The second policy is a ‘random’ policy in heterogeneous systems as explained in chapter 3. The two latter are the only serious ones. Both is common that they insist on the fixed-size-model. Hence, in case of a fully distributed application the slowest node will slowdown the whole application and the policies will not make efficient use of the resources.

Gehrke has implemented *S-PVM* which supports Shortest Expected Delay mapping for PVM application due to the current load situation (see chapter 4) [49].

1.6.3 DQS

The Distributed Queueing System (DQS) from the University of Florida is a scheduling system for sequential and parallel applications on workstation clusters [30].

1. DQS supports different UNIX platforms such as DEC OSF/1, HP-UX, IBM AIX, SGI IRIX, SunOS, and Solaris. DQS supports PVM applications.
2. DQS supports the fixed–size–model.
3. The DQS scheduling is similar to native NQS.
   (a) Queues can be suspended when other queues receive the jobs. Hence, space-sharing is optional.
   (b) DQS provides the user with different queues based on architecture, software availability etc..

       When submitting a job the user has to specify a list or resources like memory demands and software availability (PVM, Mathematica etc.). DQS automatically selects an appropriate queue. Like NQS, the basic feature of DQS is to map the application onto machines which fulfill these requirements.

       Applications are scheduled in FIFO order. There are two possible methods of mapping. The first one is to map jobs according to the queue sequence number. Then the first queue in the list receives the job for execution. The second method is to schedule by weighted load average within a group so that the least busy node is selected to run the job. The actual method used is selected at compilation time.
   (c) Checkpointing and migration are not supported.
   (d) The user specifies the minimum amount of memory each node should have.

4. Load management:
    (a) A CPU load average may be used as a load index.
    (b) There may be several queues on a machine.
    (c) No special care for interactive users.

5. The performance and behavior of DQS depends mainly on its configuration.

   In [135], experiences with an implementation of DQS combined with the concept of "computational equivalents" for heterogeneous systems are described. The algorithm needs the expected runtime of the parallel job as further input parameter and is limited to the fixed–size–workload-model.

1.6.4 CARMI/WoDi

Pruyne and Livny have described a resource management system named CARMI (Condor Application Resource Management Interface) for PVM applications [120]. CARMI is based on the Condor batch system [96].

CARMI is used together with WoDi (Work Distributor) which provides an interface for writing master-worker programs.

1. Like PVM and Condor, CARMI/WODI is available on various UNIX platforms.

2. WoDi supports master-worker-applications and uses the variable–size–model.

3. (a) WoDi supports dynamic partitioning of master-worker-applications.
    (b) WoDi dynamically distributes the tasks which are created by the master to the workers.

        Pruyne and Livny report that in some master-worker-applications, work steps come in groups, and all the results from one group have to be calculated before the next one can be started. This means strong synchronization within a master-worker-application.

        For this reason, WoDi applications can specify the beginning and the end of such a working cycle. When cycles are used, WoDi keeps a record of the computation times of all steps within a cycle. This work step history can be used for further scheduling.

        Based on estimations of the execution times of the work steps WoDi uses a greedy work step distribution algorithm where the steps are ordered according to their estimated run times.
    (c) Checkpointing and migration of master-worker-applications can easily be done on application-level. For arbitrary PVM applications, CoCheck (Consistent Checkpointing) is used [121].
    (d) The user specifies the minimum amount of memory which each node should have.

4. Load management:
    (a) The speed of a machine is determined by one of these three ways: Firstly, a speed factor as a measure of the computing capacity can be used. Secondly, WoDi can use CARMI services. Thirdly, WoDi can execute a benchmark on each new created worker.

        Since benchmarking uses processing capacity, the second approach is recommended. CARMI supports information about speed factors (as calculated by the Dhrystone and LinPack benchmarks) and about interactive users. A load index is not used.
    (b) CARMI supports heterogeneous systems by using speed factors.
    (c) When CARMI is informed by Condor about interactive users on a machine, the machine is not used anymore for the parallel application.

1.6.5 LoadLeveler

The IBM LoadLeveler is a resource management and scheduling system originally developed for the IBM SP machine [67]. Here, we describe IBM LoadLeveler Release 3.0.

1. LoadLeveler runs on every major UNIX platform such as IBM RISC and SP, HP-UX, Solaris, SunOS, SGI IRIX. The supported parallel environments are PVM 3.3 and above or any parallel programming language which uses the LoadLeveler parallel programming interfaces (such as the IBM Parallel Environment Library).
2. LoadLeveler uses the variable–size–model.
3. The scheduling strategy is space-sharing and adaptive.
    (a) The scheduling of parallel applications is space-sharing. Hence, nodes are dedicated to one application.
    (b) The scheduling strategy is adaptive and uses a so-called *temporary reservation*. This means that when a parallel job enters the system, the central manager scans a list of machines with machines which fulfill the job requirements. If sufficient machines are not available, the central manager reserves as many machines as possible. When machines become available, theses machines are added to the list of reserved machines. Finally, when enough machines are reserved, the job is dispatched.

        Since a big job may block the whole system, the amount of time the machines may be reserved is limited. When the limit is exceeded, all reserved machines are released and the parallel job is placed in the "deferred" queue. After some time the job reenters the LoadLeveler queue with its original priority.

        The amount of time for which resources may be reserved for a job and the time a job will remain in the deferred queue are configurable. The default value is 5 minutes reservation time and 5 minutes time out in the deferred queue.

   (c) Checkpointing and migration are only supported for sequential processes. The checkpoint and migration mechanism are based on Condor.

   (d) The user specifies the minimum amount of memory for each node.

4. Load management:

   (a) Nodes are dedicated to applications. Hence, only the state *available* or *not available* is used within the scheduling strategy.

   (b) No special support for heterogeneous systems.

   (c) The use of a resource manager is optional. The resource manager coordinates machine usage between interactive and batch jobs. Machines may become dedicated through interactive jobs.

5. Obviously, the temporary reservation discipline is not starvation free. Big jobs may try to allocate sufficient nodes repeatedly in vain.

   The behavior of the discipline depends mainly on the time parameters for reservation and waiting in the deferred queue. It is the system administrator's task to find the optimum values. There is no mechanism in LoadLeveler to adapt these parameters to the workload by automatically.

   The chosen default values of 5 minutes assume that load changes occur in relatively short time intervals. However, when parallel applications run several hours, this is not a realistic assumption.

1.6.6 Discussion

PVM and CARMI/WoDi differ from the other proposed systems. PVM is mainly a message passing library with minimal support for the administration of the application. There exist several approaches to enhance PVM with resource management features (see for example [65, 161, 33, 49]. CARMI/WoDi is a special resource management system for PVM applications which follow the master-worker-programming model.

In the following, we will compare the remaining resource management systems: NQS/SDSC, DQS, and LoadLeveler.

The only system which supports the variable–size–model is LoadLeveler. All others use the simple fixed–size–model.

The scheduling strategy is mostly based on FIFO-queues where the queues are defined due to resource demands of the application. NQS/SDSC uses queues depending on the runtime approximation of the application. DQS is more flexible and manages queues for different types of applications. For example, there may exist a queue for PVM applications and another queue for applications which need Mathematica.

NQS/SDSC and LoadLeveler use a space-sharing strategy. DQS can be configured to schedule different queues on the same machines.

The main deficits of the systems are in load management. LoadLeveler and NQS/SDSC do not use current load informations in their scheduling and mapping decisions. Only DQS is able to perform adaptive load balancing. None of the systems support checkpointing and migration of parallel applications. Hence, there is no support of dynamic load balancing.

1.7 Summary and our Approach

Group scheduling strategies are divided into two classes: space-sharing and time-sharing disciplines. While space-sharing is the most common technique in most existing systems, the benefits of time-sharing disciplines for multiprocessor machines is repeatedly shown in simulations.

It is still an open question whether time-sharing partitioning schemes may be also beneficial on workstation clusters. The characteristic of time-sharing policies is that the waiting time of a job depends on its service demand. Therefore, we will compare time-sharing and space-sharing disciplines in a simulation to investigate their behavior in workstation clusters (see chapter 2).

Parallel applications differ much in their computation and communication behavior. We have presented two popular types of applications: pool-of-task and fully distributed applications. Load balancing and scheduling for pool-of-task applications is most efficiently done on application-level. Our goal is to support resource management of the second type of the fully distributed application which includes the important class of computation-intensive CFD applications.

Finally, we have described the currently most popular resource management systems for parallel applications. We have discussed these systems under the aspect of scheduling and load management. We conclude that the current support for resource management is only rudimentary. In particular, the systems in use are not suited for workstation clusters where heterogeneity and the presence of interactive users have to be considered in scheduling and load balancing decisions.

Our approach is resource management support on system-level without any modifications at the underlying operating system. The benefits are transparency and portability. The user does not have to take care about load management and load balancing when she writes her application. Since the resource management system is middleware between application and operating system, it is easier to port between different operating systems and in case of operating system changes.

This means that the resource manager runs on top of the given operating system, i.e. an UNIX derivate, and all parallel applications run in competition with “local load” from interactive users or other batch jobs which are all scheduled by the native UNIX scheduler (see section 3.2).

The main contribution of our work is to develop a scheduling and load balancing strategy for heterogeneous systems. We introduce the concept of “virtual homogeneous nodes” to make the heterogeneity of the system transparent for the user. The presented mapping strategy SED uses the variable–size–model to determine an optimum mapping in a heterogeneous environment.

While migration mechanisms in user-space are meanwhile available [116, 117, 19, 146], there is a lack of good dynamic load balancing strategies which make use of them. We present a new dynamic load balancing strategies called Dynamic-SED. Results from a trace-driven simulations show that the migration overhead is neglectable and that migration is an useful mechanism to support parallel applications in workstation clusters.

1.8 Outline

In chapter 2, we present algorithms for dynamic scheduling, both space–sharing and time–sharing policies, and compare their performance under different workload models in a simulation. A new co-scheduling algorithm is presented which differs from the well-known matrix algorithm and makes use of a migration facility. We investigate under which workload parameters the time–sharing discipline may be beneficial. Further, we investigate whether migration is an adequate method for solving mapping conflicts.

In chapter 3, we discuss the special properties of workstation clusters which are important in resource management and introduce the concept of delay factors as a load metric for heterogeneous systems.

In chapter 4, we introduce the Shortest–Expected–Delay–Mapping (SED). SED is a mapping strategy for heterogeneous systems which also supports the variable–size–model. The advantage of using SED is that the heterogeneity of the system is transparent for the programmer. This is done by managing "virtual homogeneous" nodes. This makes the development of a parallel application much easier.

We introduce the concept of mapping state diagrams as a formal description tool to investigate the behaviour of different SED algorithms. In particular, we are interested whether SED may benefit from using migration. Further, we present simulation experiments to test the performance of the algorithms.

In chapter 5, we present a new dynamic load balancing strategy called Dynamic-SED which is based upon the SED mapping strategy. Dynamic-SED not only looks at the current delay of the machines, but also at the currently free memory and the number of interactive users. It presents a new approach to achieve a co-existence between parallel applications and interactive users.

We define a number of 'migration anomalies' and check whether the migration anomalies are true for Dynamic-SED or not.

At the end of each chapter there is a summary and further suggested readings.

1.9 Bibliography

Feitelson gives a detailed survey over scheduling in parallel systems with special care for gang scheduling techniques [35].

Recent developments are described in the proceedings of the IPPS workshops "Job Scheduling Strategies for Parallel Processing" [38, 40, 41, 42].

We do not consider application-level load balancing in this work. This is done for example in [94, 24, 10, 162, 82, 77, 88]. An algorithm for application-level load balancing on d-dimensional meshes is given in [126] which is superior than the common gradient model load balancing methods.

Chapter 2

Dynamic Scheduling Algorithms

In this chapter, we present scheduling disciplines which are are suited for both multiprocessor systems and workstation clusters, since the only assumption is that the nodes are interconnected by a communication bus. The situation differs from parallel computers, like for example on a hypercube or mesh-architecture, where the network topology is regarded in the scheduling and mapping decision.

We compare the presented algorithms for different workloads in a homogeneous system. We choose a homogeneous system to avoid the influence of the mapping strategy on the performance. In case of homogeneous machines a simple 1-to-1-mapping is used which means that on each machine runs at most 1 process. For the presented preemptive discipline, the 1-to-1-mapping is used for every time slice.

Dynamic scheduling disciplines for multiprocessors follow one of two philosophies, either Smallest Number of Processors First (SNPF) [99, 90] or Largest Size First (LS) [92, 130]. Largest–Size–scheduling (LS) schedules parallel jobs which have a larger size, i.e., request a higher number of machines, first.

LS algorithms are first investigated in [130]. Here, we compare the performance of LS with simple FIFO, preemptive LS, SNPF, and with FIFO-V scheduling. The latter adapts the configurational size of the application according to the current system load.

The preemptive algorithm called LST differs from the prominent *matrix algorithm*, since it uses no fixed placement of jobs. Instead of this, it makes use of a migration facility if mapping conflicts occur. We investigate the influence of migration costs on the performance.

2.1 The Scheduling Algorithms

We will investigate two space-sharing scheduling disciplines, LS and SNPF, which represent the two mainstreams on multiprocessor systems. Both scheduling disciplines assume fixed-size applications.

We have introduced LS algorithms in [130]. Here, we give an improved definition of LS. Further, we compare these strategies with the time-sharing discipline LST and FIFO-V. In the simulation, we use more realistic workloads compared with [130] where only results from exponentially distributed service demands are given.

2.1.1 LS Scheduling

Largest size scheduling is a priority based strategy, where the priority of an arriving job is initialized with $maxsize$, i.e., its specified degree of parallelism.

The scheduler gets active when a parallel job arrives or terminates. The run queue is ordered according to the priorities. Larger jobs have a higher priority and are favored. If a new job arrives and there are enough nodes available and no job with $maxprio$ is waiting in the run queue, the job will be scheduled.

Each time the scheduler is active, the priorities of all jobs which cannot be scheduled are increased. Jobs which reach the maximal priority $maxprio$ are scheduled in FIFO order. This saves small jobs from starvation.

If a parallel job terminates, the scheduler looks for the first parallel job in the run queue. If the requirements of the job can be fulfilled, this job is scheduled. This is repeated as long as there are sufficient free nodes.

If there have not been sufficient nodes available for the first job in the queue, we have to distinguish two cases. If the priority is less than $maxprio$, the scheduler tries to map the next job in the run queue. If the job has already reached $maxprio$, the scheduler gets inactive.

This means that the first job now blocks all others, until sufficient nodes get available.

2.1.2 SNPF Scheduling

The Smallest Number of Processors First discipline (SNPF) is the inverse strategy to the LS scheduling. SNPF schedules the job with the smallest size first.

The SNPF policy has been investigated in [99, 90]. The studies state contradictory results. While Majumdar, Eager, and Bunt favour SNPF [99], Leutenegger and Vernon report that "SNPF scheduling discipline performs poorly, even when the number of processes in a job is positively correlated with the total service demands" [90]. They get to this opinion comparing simulation results of SNPF, a gang scheduling algorithm, and an algorithm based upon the dynamic partitioning scheme. The latter performs best in their simulation. Since dynamic partitioning is an unrealistic assumption, it is much more interesting how SNPF performs compared with other variable or adaptive schemes. This is done in our simulation.

2.1.3 FIFO-V

Further, we compare the algorithms with FIFO-V which uses the variable-size-model. FIFO-V is a FIFO scheduling discipline which allocates the minimum of $maxsize$ and idle nodes.

2.1.4 LST Scheduling

We compare LS with its time-sharing modification *Largest Size with Time slicing* (LST). The LST algorithm makes use of a migration facility. Like LS, LST favors large jobs to reduce fragmentation. LST increases the priorities of jobs which cannot be scheduled to save smaller jobs from starvation.

Arriving jobs are ordered in the run queue according to their priorities. Like LS, the priority of a new job is equal to its size. Jobs are scheduled according to their priorities.

At the beginning of a time slice, as much jobs as possible are scheduled. The priorities of jobs which cannot be assigned are increased. If the priority of a job gets equal to $maxprio$, the priority stays constant. Jobs which have the same priority are scheduled in a Round-Robin-fashion.

Since all processors are available at the beginning of a time slice, the first job in the run queue always fits and no job can starve. If the size of the first parallel job is less than the number of processors in the system, there are still free processors and the scheduler assigns the next parallel job which fits.

Here, a *mapping conflict* may occur. If the application has already run on a subset of the machines, some machines of the last placement may already be allocated by another application. In this case, all processes of the parallel application which cannot be scheduled on their last placement are migrated. Since migration causes a substantial overhead, the migrated processes will start delayed and their time slice is reduced.

Example: In a system with 8 nodes, there may run a job J_1 on nodes 1-4 and a job J_2 on nodes 5-6. During the first time slot a job J_3 with size 4 arrives. In the next time slot there will run J_1 again on nodes 1-4 and J_3 on nodes 5-8. If J_1 terminates during this time slot, there will occur a mapping conflict and the two processes of J_2 are migrated from nodes 5 and 6 to nodes 1 and 2.

LST uses *delayed mapping* to reduce mapping conflicts. When a job is scheduled for the first time, any subset of the available nodes is suitable. Hence, nodes are reserved for the job, but it will not yet allocate nodes. This is done to give the chance to another job which requirements can be also fulfilled, but which has already run. Then the job with processor preference is assigned first and the new job uses the remaining nodes.

Example: The system consists again of 8 nodes. There are three jobs waiting in the queue, J_1 and J_2 with size 4 and J_3 with size 2. Then J_1 and J_2 are scheduled first. J_1 will run on nodes 1-4 and J_2 on nodes 5-8. After 2 time slots the priority of J_3 is also 4 and it is the first job in the queue. If we assign nodes 1-2 to job J_3, we have to migrate the processes of J_1 which can also be scheduled. Delayed mapping means that J_1 will be assigned first to nodes 1-4, and then J_3 on 5-6.

If a job terminates and there is still time left in the time slice, the next job in the run queue which fits onto the available machines is scheduled. This *alternate selection* has no effect on the job priorities.

The alternate selection initiates a potentially expensive context switch on the corresponding machines (e.g. some of the processes may have to be swapped in again). Since this overhead cannot be neglected, an alternate selection is only initiated when there is at least 10 % of the time slice left.

	J_1	J_2	J_3	J_4	J_5	J_6
size	6	4	3	4	2	5

Table 2.1: Jobs and their sizes.

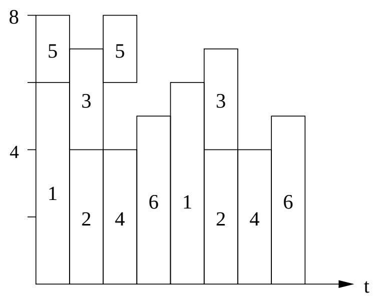


Figure 2.1: Matrix Schedule.

The LST algorithm is different from the *matrix algorithm* used for co-scheduling [112]. The matrix algorithm maps a job on a fixed subset of the processors. LST uses no such fixed mapping and migrates jobs when a mapping conflict occurs.

The matrix algorithm schedules jobs in FIFO order. It is called matrix algorithm, since it manages a (n, m)-matrix where n is the number of time slots and m the number of processors. The entry $m[i, j]$ gives the process identifier of the process which is assigned to node j in time slot i. When a new job arrives, the scheduler looks whether there are sufficient processors available in one of the time slots (First Fit). Otherwise, a new time slot is added, i.e., a new row in the matrix. There is used a Round Robin strategy between the n time slots.

Example: We give an example to illustrate the different behavior of LST and matrix algorithm. The jobs are given in FIFO-order in table 2.1. We assume that the service demands of all jobs are 2 time slots and that there is given a system with 8 nodes.

The schedule for First-Fit-matrix scheduling is given in figure 2.1. The matrix has 4 rows, i.e., 4 time slots are needed to find a placement for every job. In the third time slot an alternate selection occurs, since J_5 also fits.

The corresponding LST schedule is shown in figure 2.2. The job with the largest size is scheduled first. This is J_1. Since there are 2 nodes left, J_5 is also scheduled in the first slot. The priorities of all other jobs are rised. The evolution of priorities and remaining service times is given in table 2.2. Here, the jobs are ordered according to their priorities. The row t_i gives the priorities and remaining service time at the beginning of the i-th time slot. At the beginning of time slot t_1 both J_1 and J_6 have the same priority. Since LST uses Round-Robin between jobs of same priority, J_6 will be scheduled first. Therefore, its priority is marked in boldface.

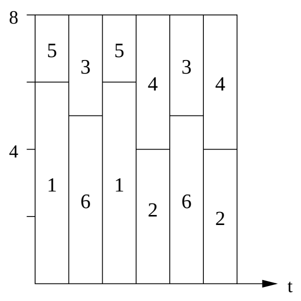


Figure 2.2: LST Schedule.

	J_1	J_6	J_2	J_4	J_3	J_5
t_0	6/2	5/2	4/2	4/2	3/2	2/2
t_1	6/1	**6/2**	5/2	5/2	4/2	2/1
t_2	7/1	6/1	**6/2**	6/2	4/1	3/1
t_3		7/1	**7/2**	7/2	5/1	
t_4		8/1	7/1	7/1	6/1	
t_5			8/1	8/1		

Table 2.2: Jobs priorities under LST.

2.2 Evaluation of Simulation Results

The performance measure of interest is the mean residence time, which is the sum of waiting and execution time. The following figures give the mean residence time as a function of the arrival rate. The corresponding system load is

$$\rho := \frac{\lambda \cdot size}{\mu \cdot n},$$

where λ is the mean arrival rate (Poisson process), μ the expected execution rate, n the number of machines, and $size$ the mean configurational size of the applications. If not stated otherwise, we use $n = 100$.

2.2.1 Approximation of Costs of Migration

Since we want to test the behavior of the LST algorithm in a simulation, we have to estimate the overhead caused by migration.

The costs of migration depend on the number of processes which have to be migrated, the process sizes, the network speed, and the used migration facility [116]. In the simulation, the time for migration is subtracted from the time slice of the migrated parallel job.

We approximate the costs of migration by the measurements with the experimental migration facility ψPBEAM [116]. The time for synchronisation and local checkpointing before migrating a process is about 10 seconds. The checkpointing of several processes can be done in parallel on the different machines. After checkpointing, the processes are migrated to their new locations. The transfer time via TCP for a process of x KB can be approximated as $y = 0.4[s] + 0.0012[s/KB] * x$. This means about 12.7 seconds for the migration of a process of 10MB [116].

The time which is needed for migration is the sum of checkpointing time and transfer time. If we migrate n processes of size 10 MB, the costs of migration are $10 + n \cdot 12.7$ seconds.

2.2.2 Workload Model and Simulation Parameters

The behavior of the LS discipline depends on the workload characteristics. For example, if the sizes of the parallel jobs are nearly constant, all jobs will get nearly the same priorities and LS will behave like FIFO.

Random arrival processes are modeled by a Poisson process. Little is known about realistic distributions of process group sizes and execution times (see section 1.4). Therefore, the investigation of different workloads by varying the distributions is recommended.

In [130], the presented LS and LST algorithms were compared for exponentially distributed service times and exponentially resp. uniform distributed job size.

Studies in high performance computer centers have shown a high variability in service demands [21, 34]. Therefore, hyperexponentially distributed service demands are used. Both workloads follow the fixed-size-model (see section 1.4).

Fixed-time-workload (FT): This workload follows the fixed-time-model, i.e., job size and service demand are uncorrelated. The service demands are hyperexponentially distributed where 50% of the jobs have a mean service of 10 minutes and 50% a service time of 120 minutes (coefficient of variation is 1.56).

Memory-bound-workload (MB): The service demands depend on the job sizes and are hyperexponentially distributed with parameter μ_1 and μ_2. The parameters are given in table 2.3.

size	w_1	μ_1		w_2	μ_2		coeff. of variation
1	.75	10	s	.25	450	s	2.45
2-16	.75	2	min	.25	34	min	2.24
17-32	.75	6	min	.25	60	min	1.97
33-64	.75	12	min	.25	124	min	1.98
65-128	.50	30	min	.50	90	min	1.22

Table 2.3: Memory-bound-workload.

The job size is uniformly distributed between 1 and 100 which is motivated by monitoring studies [34].

We test an “ideal” LST, i.e., there occur no costs for context switching between parallel jobs. In all simulations, if not stated otherwise, the time slice size is 30 time units.

Each process of an application has the same size of 10 MB in the simulations.

FIFO-V uses the variable-size-model. It depends on the application whether size and memory demands are correlated or not. In the presented simulation results we assume that there exists no correlation. Hence, every process has again the same size of 10 MB.

Another model is possible, where the memory demands of the single processes of an parallel application are negative correlated to the number of assigned virtual nodes. But in this case the simulation leads to almost similar results. The reason is that the memory demands of an application are only used to approximate the migration costs. For the discussion of the influence of migration see section 2.2.4.

2.2.3 Comparison of LS, LST, and SNPF

For the comparison of the above strategies, Oleyniczak has implemented a simulation program with which the following figures in this chapter were created [111].

Figures 2.3 and 2.4 show the mean residence time of FIFO, LS, LST, and SNPF. As expected, any other scheduling strategy performs better than FIFO.

The lower figures show the results for FT- and MB-workload with hyperexponentially distributed service demands. Additionally, we give the results with exponentially distributed service times (μ = 120 minutes), to illustrate the influence of the service time distribution[1] (see the upper figures in 2.3 and 2.4).

For the exponentially distributed service demands, there are less or no benefits from time-sharing. Under exponentially distributed MB-workload LST performs worse than the space-sharing strategies. But under hyperexponentially distributed workload, it performs substantially better, in particularly under medium and high load.

When we compare the upper and lower figures, the benefit of a time-sharing discipline under workloads with high varying service demands is impressive.

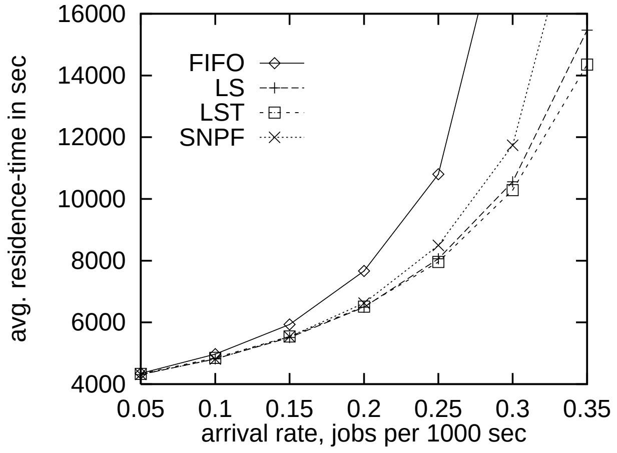
In figure 2.5 the speedups of LS, SNPF, and LST against FIFO are shown. For the FT-workload, the two space-sharing strategies behave almost the same. Under a medium loaded system speedups up to 40 % are achieved. For high load the speedup rises up to 60-80 %. The speedup of LST is substantially better than the ones of the space-sharing policies.

For MB-workload, SNPF performs almost as good as LST. SNPF corresponds to SJF, since it minimazes the waiting times under MB-workload. While SNPF should be the favorite policy for MB-workload, the benefits compared with LS are not so spectacular. The reason is that LS schedules larger jobs much earlier than SNPF, but favors simultaneously small jobs, since they can make use of idle processors.

2.2.4 Discussion of LST

Since the performance of LST depends on several parameters, we have varied these parameters to investigate their influence. The parameters are the size of the time slice, the minimum percentage

[1]All other figures show results from simulation experiments with hyperexponentially distributed service demands.



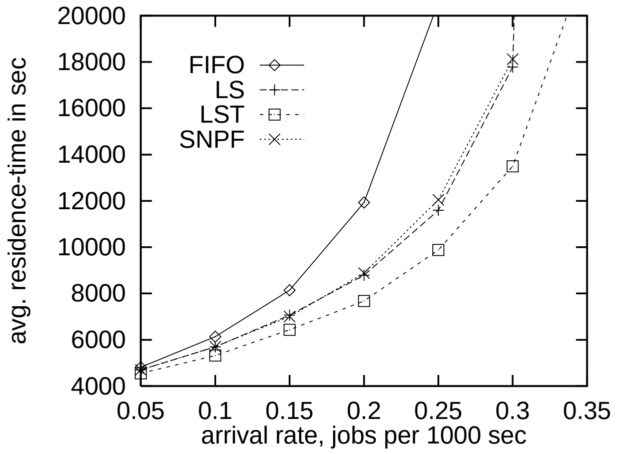
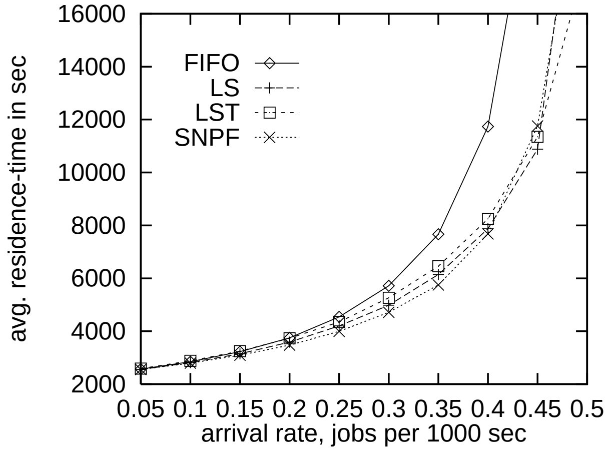


Figure 2.3: Mean Residence Time under Fixed-time-workload.

of time slice which is used for an alternate selection, and the costs of migration.

We compare the LST algorithm with LST_0 which is an "ideal" LST algorithm where no migration costs occur, with LST_NA which uses no alternate selection, and with LST_NM which uses no migration. When no migration is used, there may occur fragmentation while jobs are



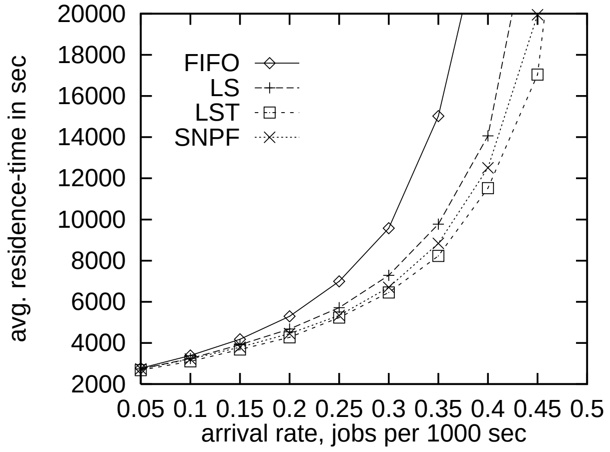
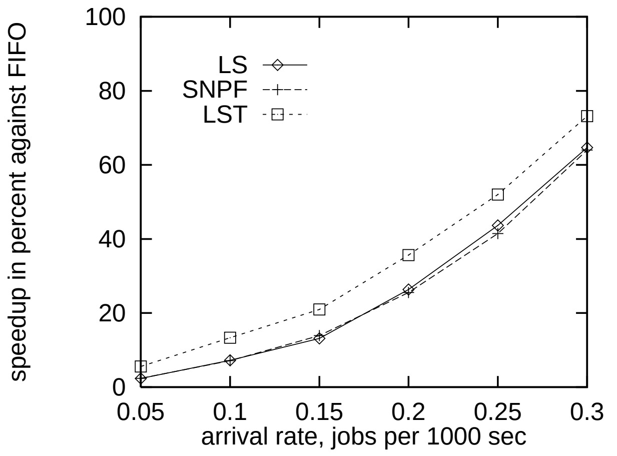


Figure 2.4: Mean Residence Time under Memory-bound-workload.

waiting in the run queue due to mapping conflicts.

Figure 2.6 shows the mean residence time of the different algorithms. While LST_NA performs worse than the other algorithms, all other LST algorithms perform very similar under both workloads. This means that the preemptive LS algorithms do not benefit from migration.



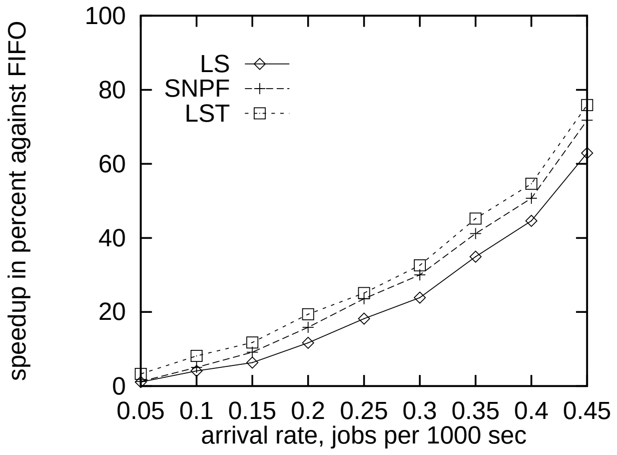
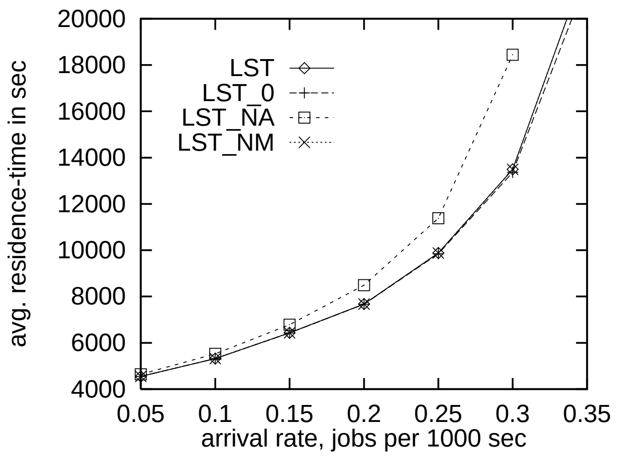


Figure 2.5: Speedup against FIFO under Fixed-time-workload (top) and Memory-bound-workload (bottom).

To investigate whether migration has so less influence on the performance, the percentages of migrated processes are shown in figure 2.7. Under both workloads, the percentage of migrated processes increases under higher load, but only less than two percent of all processes are migrated



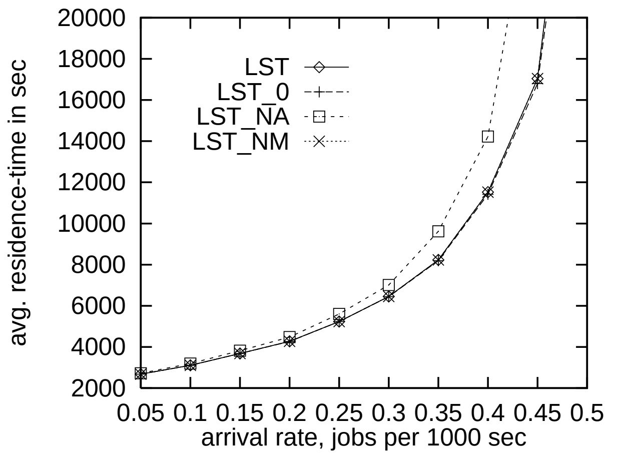


Figure 2.6: Mean residence time under Fixed-time-workload (top) and Memory-bound-workload (bottom) for different LST Algorithms.

at all. The mean time between two migration events is also shown in figure 2.7. A migration event occurs when the scheduler determines that migration is necessary due to mapping conflicts. Even under high load the mean time between two migration events is more than 10 hours! The

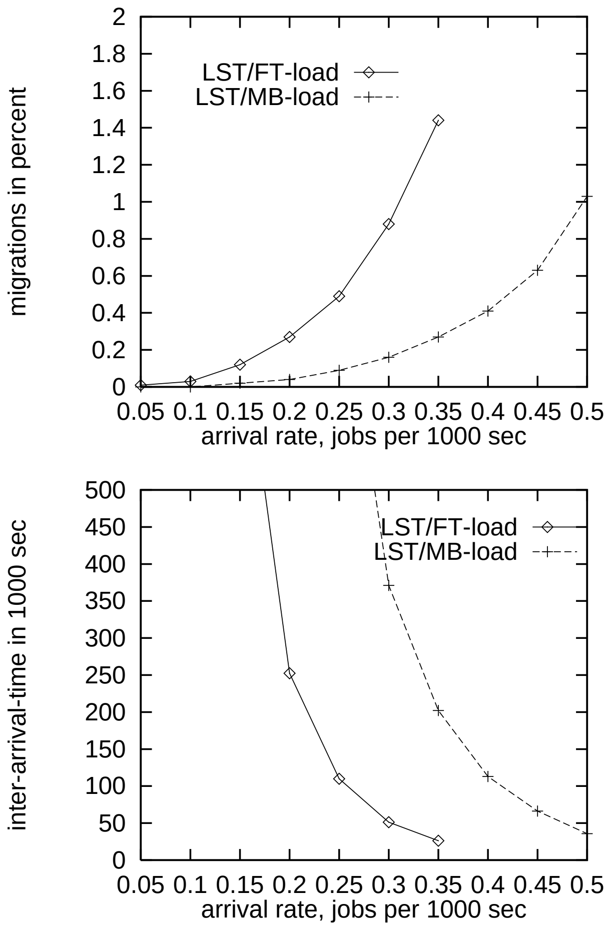


Figure 2.7: Percentage of migrated processes (top) and mean time between migration events (bottom).

mean number of migrated processes per migration event is about 20 for both workloads.

This shows that mapping conflicts are a rare event under the considered workloads and the performance of the LST algorithm cannot be improved substantially by a faster migration facility.

The higher influence on the performance of LST, has the alternate selection which uses rests of time slices. This makes the performance behaviour of LST nearly independent from the choice of the time slice size under exponentially distributed service demands [130].

Figure 2.8 shows the mean residence time for different time slice values under FT-workload. The upper figure shows the results for LST, the lower the results for LST_NA. The same values for MB-workload are given in figure 2.9.

Without alternate selection the performance of LST_NA depends much on the time slice size. In particularly for the FT-workload, the smaller sizes show a better performance, since less processor time is wasted.

The performance of LST stays stable for the different slot sizes. Only the smallest time slice value of 5 minutes shows a lower performance under medium and higher load.

2.2.5 Benefits of Variable-Size-Model

Here, we compare the LS algorithms with FIFO-V which uses the variable-size-model. If there are nodes available, FIFO-V allocates

$$size = min\,\{maxsize, \text{number of idle processors}\}$$

nodes. Hence, the $minsize$ of an application is assumed to be 1.

Figure 2.10 shows the mean residence time of FIFO-V compared with the algorithms discussed above.

For both workloads, FIFO-V performs worse than the other disciplines under low load, even worse than FIFO. Under medium load it performs better than the other space-sharing strategies, and under high load it performs even better than LST.

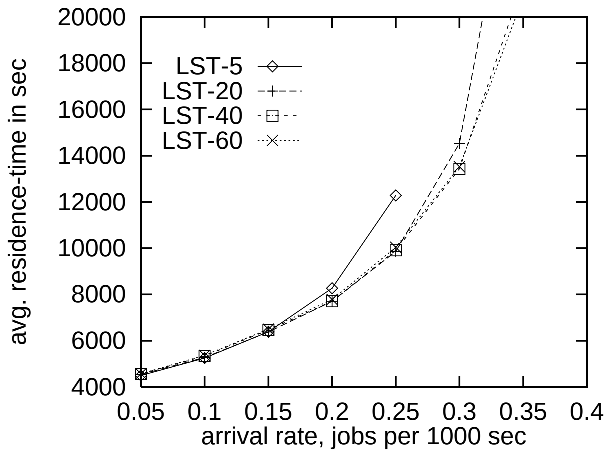
When the application is fixed on a size, there will be the case that processors are idle while jobs are waiting in the run queue. Hence, all algorithms which have to schedule fixed-size-workload suffer under increasing waiting time when the system load increases.

In the simulation, we assume an optimal speedup on all $maxsize$ processors. Hence, an application will be n times faster on n processors compared with sequential execution. The larger jobs are the candidates to reduce their size. Hence, the runtime of the larger applications will be increased under FIFO-V. In particularly, these are exactly the jobs with high service demands in the MB-workload. Hence, the benefits of the variable-size-model are less obvious this workload. For the FT-workload, there can be seen only performance benefits in the comparison with the space-sharing policies.

2.3 Summary

We have presented both space-sharing and time-sharing LS strategies. Further, we have compared scheduling disciplines which are based on the fixed-size-model and variable-size-model.

The presented gang scheduling algorithm LST differs from the matrix algorithm in using migration if necessary instead of fixed mappings. We have investigated this algorithm with dif-



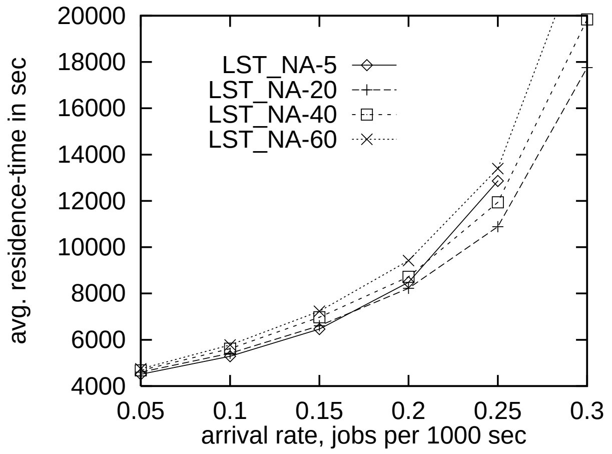
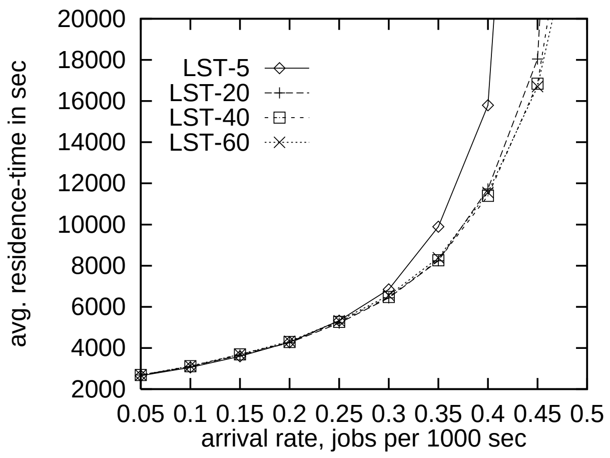


Figure 2.8: Influence of time slice values under Fixed-time-workload.

ferent parameter values for the size of the time slice and with/without alternate selection. From our simulation results, we conclude:

- The workload characteristics and the system load are most important for choosing an efficient scheduling strategy.



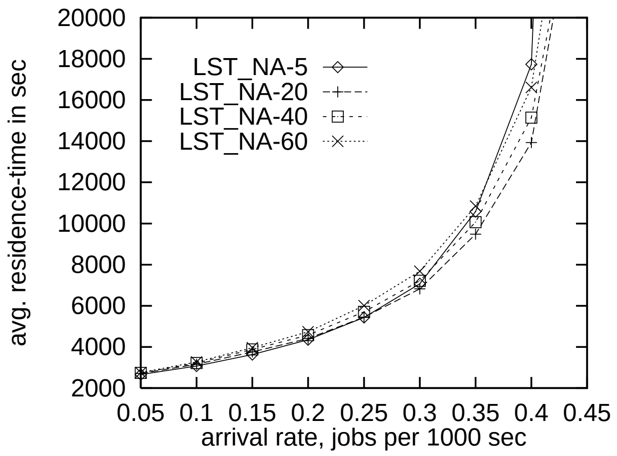
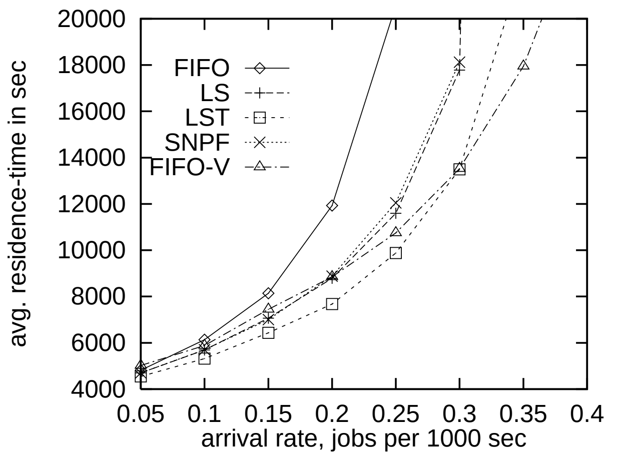


Figure 2.9: Influence of time slice values under Memory-bound-workload.

- Simple scheduling strategies like LS and SNPF can make effcient use of the resources compared with FIFO.
- Under FT-workload the space-sharing strategies LS and SNPF behave similar. We conclude that processor fragmentation is not as dangerous as assumed. Both strategies are



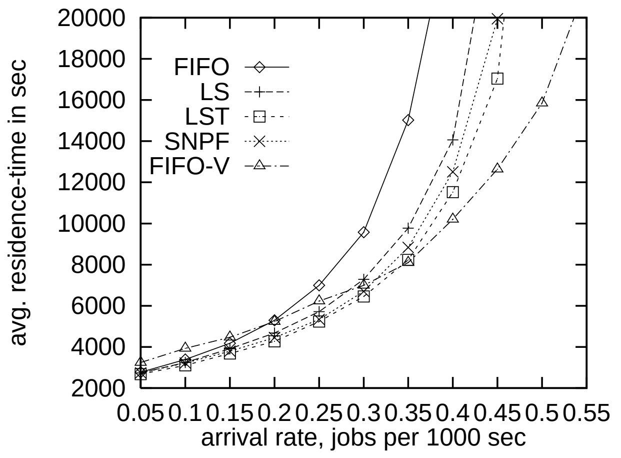


Figure 2.10: Mean residence time under Fixed-time-workload (top) and Memory-bound-workload (bottom).

outperformed by LST.

- Under MB-workload the space-sharing strategy SNPF behaves almost as good as LST.

- The benefits of a time-sharing discipline under workloads with high varying service demands is impressive.
- Mapping conflicts are seldom under LST (less than 2 % of the processes have been migrated). Hence, migration is an adequate method for solving mapping conflicts. The migration overhead is neglectable.
- Alternate selection is necessary to make the performance of LST independent from the choice of the time slice.
- Under medium and high system load, the performance can be improved by supporting the variable-size-model.

2.4 Bibliography

The implementation of the simulation program is described in detail in [111].

An algorithm called LDLP (Largest Dimension Longest Processing time), similar to LS, is investigated in [129]. LDLP needs runtime approximations for its calculation of the job priorities. It is shown that the LDLP scheduling policy is suitable for a wide range of processor networks which own a decomposability property. Examples of decomposable graphs are decomposable Cayley graphs such as the n-cube and the n-star.

Krueger et al. present a dynamic group scheduling algorithm called Scan-algorithm for hypercubes [79]. Scan is a non-preemptive algorithm which clusters jobs of equal size. Their simulation results show that Scan improves the mean residence time compared to algorithms that do not cluster equal-sized jobs.

Feitelson presents simulation results for different “packing schemes” for the matrix algorithm [43]. The problem to find sufficient processors for a new application is similar to the allocation problem in memory management. In the simulation, First-Fit and Best-Fit perform similar, while a mapping scheme based upon buddy systems (Buddy-Matrix-Algorithm) performs best. The latter is not astonishing, since it reduces processor fragmentation.

Further, Feitelson investigates an algorithm which uses migration instead of fixed placements. This algorithm is similar to our LST algorithm presented in [130]. In the simulation, the migration based algorithm performs as good as the buddy allocation scheme. Feitelson states that this result may be misleading, since the overhead for migration is ignored in the simulation. From our experiences with LST, we conclude that the migration overhead is not a relevant factor. Hence, LST will outperform traditional packing schemes used in co-scheduling.

Setia [136, 137] compares a scheduling strategy similar to LST with the Buddy-Matrix-Algorithm. He uses a different calculation scheme for job priorities:

$$priority := (jobsize \times approx.\, ofruntime)^{-1}.$$

In a trace-driven simulation which also regards the cost of migrations, this migration-based scheduling strategy performs better than the Buddy-Matrix-Algorithm.

Time-Sharing on a 2-dimensional mesh architecture is investigated by Rotzoll [124]. She presents simulation results which show that under the time-sharing policy the mean residence time and waiting time are positively correlated. She reports that the mean residence time was in no case higher compared with a FIFO space-sharing strategy. Further, for higher load the mean residence times may be improved by time-sharing, but Rotzoll gives no figures for this.

A space-sharing strategy for a 2-dimensional mesh architecture which clusters jobs according to their size (similar to the Scan strategy for hypercubes) is presented in [103]. The authors compare their algorithm for different parameters and show the influence of the parameters on the performance.

In [143] demand-based co-scheduling for multiprocessors is proposed. Sobalvarro and Weihl use informations about which processes are communicating in order to coschedule only these. The authors state that their approach is more flexible than the traditional co-scheduling. Obviously, the result depends on the workload characteristics. If we assume that the applications are fully distributed (see section 1.3) demand-based co-scheduling is equal to co-scheduling.

Chapter 3

Resource Management in Workstation Clusters

Since workstations are often dedicated to a special user, they run most of the time idle. A number of recent research activities have tried to exploit the computing power of such environments [17, 150]. Failure transparency and support for group scheduling are areas of many current research activities [116, 19, 146, 64].

The benefits of workstation clusters are their

- availability,
- price/performance ratio, and
- scalability.

Further, workstation clusters are easy to rejuvenate. While parallel computers are out-of-date after only 3-6 years and lots of efforts have to be done for a new investment, a single workstations is easily replaced. If money is available, more machines can be added which means in most cases that more powerful machines are available. Hence, a workstation cluster is comparable with an organism which cells are periodically regenerated.

In figure[1] 3.1 the load situation of cluster of about 150 workstations at the Technical University of Braunschweig is shown. The three curves show the total number of ready-to-operate machines, the number of active machines, i.e., with running jobs, and the number of machines which are used by interactive users. The load change between day and night time is obvious. The load on almost all machines increases from Friday on. This is due to a parallel application which was started on Good Friday to run over the easter weekend.

Further, machines which are shown to be 'active' are not necessarily highly loaded. There may be also a lot of idle times on these machines which could have been used for long running applications.

The importance of workstation clusters for parallel computing is, for example, demonstrated by the fact that the Gordon Bell Prize in the price/performance category for significant achievements of supercomputers to scientific and engineering problems has been won by a parallel

[1]This figure was provided by D.J. Schmidt of the Computing Center of the TU Braunschweig.

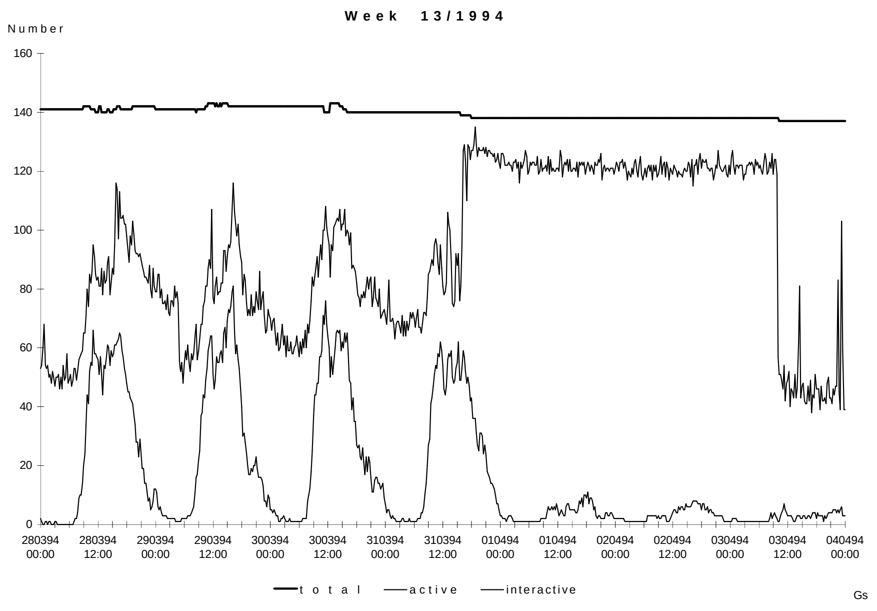


Figure 3.1: Load Situation in a Workstation Cluster at the TU Braunschweig.

program on a workstation cluster in the recent years. In 1995, Panayotis Skordos of the Massachusetts Institute of Technology was recognized for his modelling of air flow in flute pipes [73].

In the following section we report the results from monitoring studies which have investigated the load situation on workstations. Next, we discuss the differences between workstation clusters and parallel computers which are important in scheduling.

Since every non-trivial load balancing strategy needs load informations for its decisions, we have to define a suitable load metric for workstation environments. This is done in the third part of this chapter. We will introduce delay factors as a load metric for heterogeneous systems and present a heuristic approach to calculate them.

3.1 Load Characteristics of Workstations

Monitoring studies of the load situation on workstations are given in [16, 89, 26, 71].

All studies consistently report that the majority of all UNIX processes have a run time of few seconds. Cabrera has obtained the following figures [16]:

- 70 - 80 % of all processes consume less than 0.5 s cpu time,
- 78 - 95 % of all processes consume less than 1 s cpu time,
- only 2 % of all processes run longer than 16 s.

Ju et al. have found that about 80 % of all processes consume less than 0.5 s cpu time [71].

Leland and Ott state similar results [89]. They analyzed the behaviour of 9.5 million UNIX processes during normal operation. They found the following approximation for the probability distribution F of the amount of CPU time used by an arbitrary process:

$$(1 - F(x)) \approx r \cdot x^{-c},\ 1.05 < c < 1.25.$$

Further, Leland and Ott have investigated the joint distribution of the amount of CPU time used and the number of disk accesses made. They report that the result confirms the common "folk theorem" that there exist three types of processes: CPU intensive applications ("CPU hogs"), I/O intensive applications ("disk hogs"), and "ordinary" processes which use relatively little CPU or I/O with no compelling relation between use of either. Processes which are CPU and I/O intensive are rare. Only 10 of the observed processes belong to both classes.

When we want to do load balancing, we need a load management, a scheduling and an export/migration component [87]. The load management component gathers and distributes the current load information. The scheduler is responsible for the placement decisions, i.e., on which side the job shall be executed. The export/migration component executes the scheduling decisions and exports and starts the job on another machine. In case the selected process is already running, it has to be migrated which causes a higher overhead.

Since the export or migration of a job adds additional overhead, this is only worthwile for long running applications, i.e., applications which run at least several minutes.

In some distributed operating systems such as Amoeba, load balancing is done per process. A `run server` selects the machine when a job has to be created [151]. The `run server` checks whether there is a machine of the specified architecture available (SPARC, VAX, etc.) and whether the machine has enough memory. Among the possible candidates, the fastest machine is chosen. The used load metric is similar to the one proposed in section 3.3.2. The speed of the machines is approximated by MIPS values.

Since most of the processes created on workstations have a lifetime of only a few seconds, this approach seems not reasonable. Instead, we use a process model with 3 classes:

1. Short jobs $<$ n seconds are executed locally,
2. jobs $<$ 15 minutes are initially placed by the load balancing system,
3. jobs $>$ 15 minutes are initially placed by the load balancing system and periodically checkpointed for failure transparency. These processes may be migrated for load balancing reasons.

Parallel applications typically belong to the third class of computational-intensive processes. They have to be scheduled with special care.

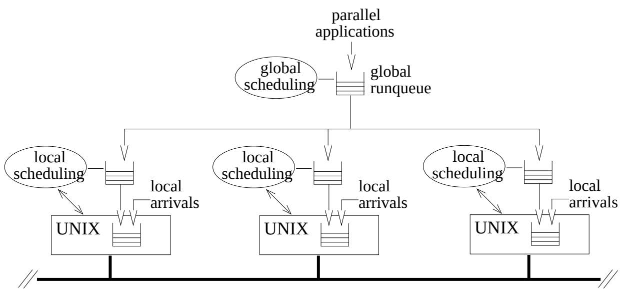


Figure 3.2: System Model [134].

3.2 The Challenges of Workstation Clusters

Scheduling on workstation clusters is quite different from scheduling on parallel computers. From the operating system view, each workstation is an autonomous system. Each machine has its own scheduler, most commonly a UNIX time-sharing scheduler. A portable approach has to consider this situation. Hence, we assume that the implementation of parallel job scheduling has to be done in user space on top of the kernel without any modification at the existing operating system. This leads to the 2-level system model shown in figure 3.2 [134].

The first level is the UNIX time-sharing system. Above this, there is a scheduling facility, which schedules only the parallel applications which are submitted to the system by a special user command. The global run queue for arriving parallel jobs is managed by a global scheduler, which is responsible for mapping the parallel jobs onto the machines.

3.2.1 Network Topology

Scalable parallel computers are based on special network topologies such as mesh or hypercube architectures [154]. On these machines, an application is mapped onto processors which are topologically close, or more formally, the application allocates a subgraph with minimal diameter [79, 91, 103, 129].

In figure 3.3 the network topology of a 4-cube and a workstation cluster is presented. Workstation clusters typically have a simpler network topology where the machines are connected by a broadcast medium like the Ethernet. Thus the topology does not have much influence on the mapping decisions.

The situation gets more complex, if some workstations of the cluster are connected by different networks, i.e. Ethernet, Fast Ethernet, and ATM.

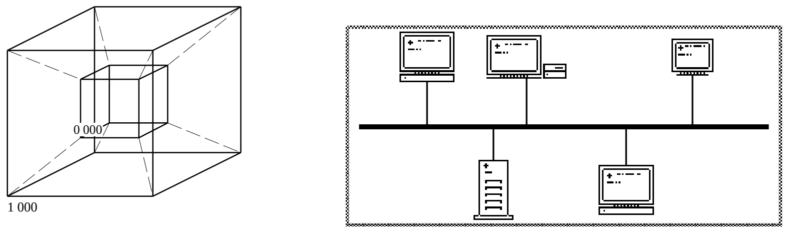


Figure 3.3: Network Topology of 4-cube and Workstation Cluster.

3.2.2 Time-Sharing Disciplines

While gang scheduling is assumed to be an efficient scheduling strategy on multiprocessor machines [112, 90, 37], it is not yet clear whether gang scheduling can be useful on networks of workstations.

In a workstation cluster, there is no synchronization hardware for context switching between parallel applications and no global time at all. This makes the implementation of time-sharing disciplines more difficult.

On a workstation cluster, time slices for scheduling parallel applications can only be implemented on top of the local scheduler and the synchronization for context switching between parallel jobs can only be done by message passing.

Switching can be done in the following way: When the global scheduler wants to initiate a parallel job switch, it sends a message to all machines that the running process has to be suspended and the next process in the local run queue has to be scheduled. When an application is suspended, all messages that are “on the way” have to be saved. This means that a checkpoint of the application has to be made. Therefore, additional synchronization overhead occurs.

In [64], first experiences with gang-scheduling on a workstation cluster are reported. The authors assume “that every workstation in the cluster is dedicated for use as a computation server”. This assumption does not seem valid in most computer departments. The implemented runtime environment is tested with two applications which are not representative for parallel applications: a program which calculates Fibonacci numbers and a program which forks threads to the next processor sequentially. The authors give results from measurements with time slice size 0.2, 0.4, 0.6, 0.8, and 1 second which are not promising even when they conclude that “gang-scheduling on workstation clusters can be practical.” Results with a more realistic application should be more interesting.

Further, a preemptive discipline may increase the danger of *swapping*, since the memory requirements of all running processes may exceed the available local memory. The job switch may effect that a process has to be swapped in again. Since this may happen for several processes of a parallel application on different machines simultaneously, network traffic will rise and the time for the parallel job switch increases considerably.

The question is, whether there are any good reasons to use a time-sharing policy on workstation clusters. Time-sharing disciplines are favoured to guarantee short response times, in particular for interactive users. Since each workstation has its own UNIX scheduler, short response times for interactive users are already supported by the UNIX time-sharing system.

Another reason why time-sharing may be beneficial is the heterogeneity of the workload, i.e., workload with a high coefficient of variation of job service demand. The characteristic of time-sharing policies is that the waiting time of a job depends on its service demand. Our experiments with a time-sharing policy in chapter 2 have shown the limited benefits under medium and high load. Under MB-workload the space-sharing strategy SNPF behaves almost as good as LST.

Since the benefits of co-scheduling on workstation clusters seem questionable, we will exclude them in our approach.

3.2.3 Heterogeneity

The most challenging property of workstation clusters is their *heterogeneity*. Heterogeneity includes different architectures which are not binary compatible, different memory sizes, and different speed characteristics [134].

If a parallel application is mapped onto a subset of the machines consisting of slower and faster machines, the slower machines may slow down the whole application, and the overall performance of the system is reduced (see section 1.3).

3.2.4 Motivation for Migration

There are three reasons why a migration facility is necessary for resource management in workstation clusters [134].

Firstly, applications may profit from migration to faster hosts.

Secondly, there may be *interactive users* who want to use their machines exclusively. Condor, as one well-known example, uses the idle times of workstations for compute-bound sequential jobs [96]. If the user of a workstation returns, Condor suspends the application and observes the user behaviour. If the user is still active after several minutes, Condor migrates the application to another idle host.

Thirdly, the failure probability in a workstation cluster is much higher compared to parallel computers. Rebooting a machine may be originated by other users, due to software errors, or by the system operator for administration reasons. Migration can improve fault tolerance, by evacuating hosts prior to regular shutdowns.

We conclude that the scheduler has to be active in case of the following *migration events*:

- shutdown of a machine,
- interactive user arrivals,
- substantial load changes on a machine which would slow down the parallel job,
- substantially faster machines become available.

3.3 The Load Management Component

In this section we motivate the definition of *delay factors* as a load metric suitable for heterogeneous systems and discuss several approaches to calculate this metric [132, 133].

3.3.1 Definition of Delay Factors

In a homogeneous system the current speed of a machine depends only on the current load, which can be measured as the number of jobs in the run queue. But in a heterogeneous system, we have further to consider the different server capacities and configurations. A load metric suitable in a heterogeneous system is a *delay factor* which gives the expected delay of an application on a machine normalized relative to, for example, the slowest machine in the system.

Let $runtime(M_i, load_i)$ be the runtime of an application on machine M_i under load $load_i$. The delay factor of an application on machine M_i is defined as

$$delay(M_i) := \frac{runtime(M_i, load_i)}{runtime(M_{slow}, 0)},$$

where M_{slow} denotes the machine architecture with the lowest processing capacity in the system. For example, a delay of machine M_i equal to 0.5 means that the application will run two times faster on machine M_i compared to the idle slowest machine.

We define the *speed factor* α_i of machine M_i to be the quotient of the runtime on machine M_i and the runtime on the slowest machine of the system when there is no other load in the system

$$\alpha_i := \frac{runtime(M_i, 0)}{runtime(M_{slow}, 0)} .$$

When the machine M_i is idle, we get a $delay(M_i) = \alpha_i$.

However, the definition of the delay factors is of no use for computing the delay factor. While $runtime(M_{slow}, 0)$ is the result of one benchmark for each application, $runtime(M_i, load_i)$ has to be calculated for every application, every machine M_i, and every possible load $load_i$, which is impossible.

Instead, we use another approach where the delay factors are only estimated from the current load. If there are already n processes running on machine M_i and we assume a time-sharing system, the delay will be approximately $delay(i) = \alpha_i \cdot (1 + n)$, neglecting the overhead for time-sharing and, possibly, swapping. This motivates our redefinition of the delay factor

$$delay(M_i) := \alpha_i \, (1 + load_i),$$

where $load_i$ is an approximation for the current load. This definition regards that even if the load is zero on all machines, the machine with the best speed factor is most attractive. The question is, how to calculate α_i and $load_i$?

3.3.2 Calculation of Speed factors

A common way to calculate speed factors is benchmarking. But the experiences with real applications show that speed factors depend on the measured applications. This makes the use of a common speed factor difficult.

We have measured the speed characteristics of applications which are popular in our computer department due to our system accounting, and of some popular benchmarks.

The speed factors for a LaTeX application, a C-compiler run (gcc), a program which calculates Mandelbrot sets (mdst), the PovRay raytracer, and the dhrystone benchmark are shown in table 3.1. Additionally, the relative MIPS values are given. The speed factor of the Sun SLC is 1, since this is the slowest architecture in our system.

	LaTeX	gcc	mdst	PovRay	dhrystone	MIPS
Sun SLC	1.00	1.00	1.00	1.00	1.00	1.00
Sun IPC	0.83	0.75	0.55	0.73	0.84	0.79
Sun ELC	0.70	0.69	0.34	0.52	0.56	0.53
Sun Classic	0.72	0.56	0.36	0.46	0.36	–
Sun SS2	0.53	0.45	0.26	0.42	0.46	0.44

Table 3.1: Speed factors of different applications.

The measurements confirm that the speed characteristic of a machine also depends on the measured application. The reason is that the applications differ in the rate of floating point and integer arithmetic, and in their I/O-behaviour.

While the ranking of the machines stays stable for the different applications (the only exception is Sun Classic which is slower than the Sun ELC for some applications and faster for some others), the absolute values differ significantly. If we have an application running on an idle Sun ELC, the expected delay on this machine will be $0.7 \cdot (1 + 1) = 1.4$ according to the speed factor of the LaTeX application. This looks worse than the expected delay on an idle Sun SLC. But if we use instead the speed factor which is measured for the Mandelbrot set application, the expected delay will be 0.68 which looks better than the idle SLC. Hence, the load balancing decision depends mainly on the used speed factors.

There are two solutions for this problem. The load management can use one speed factor for each machine which is calculated as a mean value from several different benchmarks. This approach is transparent for the user, but may lead to non-optimal placement decisions of the load balancing facility.

Alternatively, the user may give hints how much I/O traffic will occur within her application and whether the application does mainly floating point or integer arithmetic. Then the load management component can choose the suitable speed factor due to these characteristics.

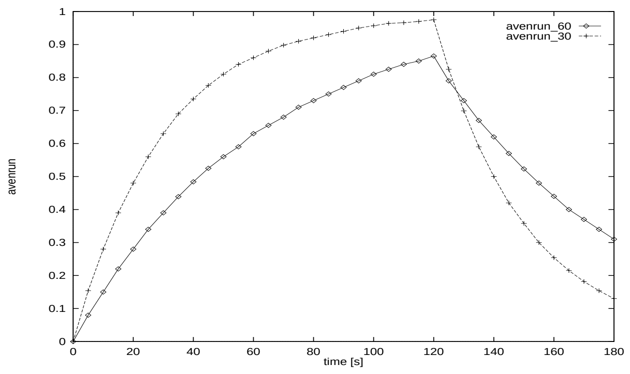


Figure 3.4: Comparison of 1-minute-average and 30-seconds-average.

3.3.3 Load Parameters for Homogeneous Systems

Most UNIX kernels, i.e. HP-UX, Irix, SunOS, Solaris, and FreeBSD, gather load statistics periodically every 5 seconds. This information is available through the `vmstat` or `sar` command [107] which report statistics about the CPU run queue, the virtual memory, disk, context switches, and CPU activity. Altogether, `vmstat` reports about 22 load parameters.

The reported values of the run queue are non-averaged snapshot values. A process which has been running for the first 4 seconds and is then interrupted because of I/O will not be counted in the run queue length.

To get more reliable load informations, load averages are in common use which smooth the gathered load parameters exponentially. These load averages, called *avenrun*, are calculated as

$$\textit{avenrun}_b(t) = e^{-\frac{a}{b}}\ \textit{avenrun}_b(t-1)\ +\ (1 - e^{-\frac{a}{b}})\ \cdot load,$$

where $load$ is the sum of the length of the CPU run queue (including the running process) and the number of processes which are waiting for I/O, a is the smoothing period, and b is the smoothing interval. Since UNIX updates its load statistics every 5 seconds, a smoothing period $a = 5$ seconds is used. UNIX provides *avenrun* values for the smoothing intervals $b = 1, 5$ and 15 minutes.

This means that in case of the 1-minute-average the old load value is weighted with 0.92 and the new $load$ is weighted with 0.08. Hence, the load average reacts very slowly on load changes. Figure 3.4 shows the value of the 1-minute-average when a process arrives in an idle system and runs for 2 minutes. The curve for a 30-second-average is also shown for comparison. The *avenrun* metric adapts very slowly to load changes. When the job is already running for 60 seconds, the value of *avenrun_60* is still about 0.6 instead of 1.

Comparison of Load Metrics in Homogeneous Environments

There already exist comparisons of *avenrun* with other load indices which show that *avenrun* is not the best choice for a load metric even in homogeneous environments [45, 84].

Kunz [84] tested six one-dimensional workload descriptors based on parameters like number of processes in the run-queue, CPU-time, or 1-minute load average within a load balancing environment on a network of homogeneous UNIX workstations. The results show that all examined descriptors lower the mean response time of processes and that the best single workload descriptor is the number of processes in the run queue, and the worst is the 1-minute load average. Further, combining the two best single workload descriptors, the number of processes in the run queue and the system call rate, leads to no improvement over the scheduler versions using a one-dimensional workload descriptor.

Similar results are presented by Ferrari and Zhou [45]. The authors have also tested a wide range of load indices within a load balancing environment. The results also indicate that load indices which are more up to date than the UNIX 1-min average improve the performance of the load balancing facility.

The index which is found to be among the best is the sum of the CPU– and disk– queue lengths, and the amount of processes in page wait, averaged over a 4 second period. Ferrari and Zhou implemented their own system statistic inside the kernel to gather such current information. The length of the used system queues was sampled every 10 ms by the clock interrupt routine and used to compute the one-second average. These variables were managed by the kernel.

It has to be mentioned that these results depend on the used workload. When the inter arrival time of jobs is in the magnitude of some minutes, *avenrun* is capable to adapt to load changes in time. Hence, the time constraints given by the expected workload determine whether *avenrun* adapts to load changes fast enough. The major problem is that the results cannot be generalized for heterogeneous systems.

3.3.4 Load Parameters for Heterogeneous Systems

Ferrari and Zhou report that the metric which equals the total number of processes ready to run and execute, being paged and swapped, and doing file I/O performs best. The *avenrun* uses also the number of processes in the run queue and disk queue. But the parameter disk queue is misleading in heterogeneous systems.

An example which illustrates the situation for configurational heterogeneous machines is shown in table 3.2.

We used an artificial workload which consumes a large amount of system–time and consists of a process which repeatedly opens a file, writes a byte into it, closes it, reopens the same file, reads a byte from it and closes it again. This load is characterized by a lot of I/O operations. We call it $sys\,n$, where n gives the number of started processes. These processes run concurrently to the measured application. It is obvious that this type of load is in no way realistic, but enables us to examine the behaviour of certain applications under various background loads quite easily.

The Sun SS2 was configured as a file server in the first test case (all file accesses of the background load were local), and in the second as a diskless client (the file accesses were remote via NFS). The file accesses of the gcc application were via NFS in both cases.

The measured benchmark which runs concurrently to the artificial workload is the compilation run. We give the delay of this benchmark compared to the execution time on the idle SS2. The results in table 3.2 show that the delay factors for the gcc application are quite similar by contrast to the gathered load data. Hence, the disk queue as load metric is especially misleading in this case.

	SS2 (file server)		SS2 (diskless)	
load	disk run queue	delay gcc	disk run queue	delay gcc
sys1	0.92	1.22	0.03	1.26
sys2	1.75	1.28	0.08	1.37
sys3	2.76	1.31	0.87	1.50

Table 3.2: Disk queue and delay.

Further, we tested the smoothed run queue parameter as load parameter for delay factors in heterogeneous environments. Again, we used an artificial workload as background load. The background load, called $CPUn$, consists of n processes which are always runnable.

In table 3.3 the delays of the gcc compiler and the LaTeX application, which run concurrently to the workload $CPUn$, are shown. Here, the reported delay is normalized to the local execution on the idle machines. The standard deviations of the measurements were between 0 and 3 %.

The CPU run queue lengths are gathered before the benchmarks are started. We make two observations:

1. The delay of the applications is correlated with the CPU run queue lengths for all architectures.
2. The more CPU-bound application LaTeX suffers more under the load $CPUn$ than the C-compiler. The delay of the LaTeX application under $CPU3$ is about 3.5 on the SS2, where the delay of the gcc application under the same load is only 1.87. This effect can be observed on all machines. The LaTeX application spends about 137.4 seconds in execution and 2.0 seconds in the system, while the gcc compiler spends about 124 seconds in execution and 21.8 seconds in the system. Since I/O interrupts are handled with a higher priority, the gcc application is not delayed by the $CPUn$ load as much as the number n should let expect.

If the delay factors of n machines have to be compared, it is not necessary to compute the exact values, only the order is important. Since the ranking induced by the CPU run queue lengths and the ranking induced by the real observed delays is the same for both applications, the CPU run queue seems to be a good parameter to specify the load for all the considered architectures.

Therefore, we use the following simple delay factor[2]

$$\text{delay} := \alpha \cdot (1 + \text{CPU run queue}).$$

[2]This definition was first used in our experimental YALB (Yet Another Load Balancing System [147]).

Type	load	CPU run queue	LaTeX delay	gcc delay
SS2	none	0.04	–	–
	cpu1	1.12	2.03	1.28
	cpu2	2.17	2.66	1.60
	cpu3	3.13	3.50	1.87
ELC	none	0.19	–	–
	cpu1	0.98	1.90	1.41
	cpu2	2.13	2.73	1.77
	cpu3	3.01	3.65	2.13
IPC	none	0.11	–	–
	cpu1	1.00	1.92	1.51
	cpu2	2.01	2.79	2.03
	cpu3	2.96	3.66	2.43
SLC	none	0.02	–	–
	cpu1	0.99	1.89	1.49
	cpu2	1.97	2.74	1.99
	cpu3	2.96	3.70	2.49

Table 3.3: Evaluation of CPU run queue

The quality of this delay factor is evaluated in [133, 132]. The metric is tested in a trace-driven simulation of a load balancing system. The load balancing system determines the fastest available machine in the system based upon the current delay values of the machines.

The results are evaluated by calculating the *slowdown* which is the difference between the runtime on the chosen machine and the possible optimal placement. In the simulation the slowdown of compute-intensive applications is between 10-20 % which is a promising result. Only under medium and high load, i.e., when there are no idle machines any more, the slowdown values increase over 50 %.

3.4 Summary

Scheduling parallel applications on a workstation cluster has to be done on the top of the local time-sharing scheduler. This leads to a 2-level architecture of the scheduling system.

We have discussed the problems of co-scheduling on workstation clusters. We conclude that co-scheduling is not feasible in these environments.

We have explained why a migration facility is necessary for resource management in workstation clusters.

The most challenging property in scheduling is the heterogeneity of workstation clusters and the presence of interactive users.

We have introduced delay factors as a load metric for heterogeneous systems and presented a heuristic approach to calculate them which behaves well for compute-intensive applications [133, 132].

3.5 Bibliography

Parts of this chapter are published in [133, 132, 134].

In [133, 132], the presented run queue based delay factor is compared with a load metric which is calculated by neural networks. Whilst the first is easy to calculate and performs satisfactory in heterogeneous systems, the neural network approach yields better results, especially under medium and high load.

Chapter 4

Mapping Strategies for Heterogeneous Systems

In this chapter, we present mapping algorithms for heterogeneous systems which are an adaption of the *Shortest–Expected–Delay* mapping (SED) proposed for sequential processes. SED in homogeneous systems is known as “join the shortest queue”. This is the optimal strategy in the case of sequential processes in homogeneous systems [163, 134].

In a heterogeneous system, the runtime of an arriving job depends not only on the number of jobs in the run queue, but also on the server capacity. When a sequential job arrives, the SED strategy assigns the job to the node with the shortest expected delay based on the current delay factors. Studies of scheduling strategies for heterogeneous systems show that SED outperforms other strategies like for example Bernoulli Splitting [108].

SED supports the variable–size–model in a heterogeneous environment. The advantage of using SED is that the heterogeneity of the system is tranparent for the programmer.

Saphir et al. state that “currently most applications at NASA Ames are statically load balanced, assuming that each processor is equally fast so that the work should be divided evenly among them” [127]. It is easier for the developer of an parallel application to assume that all nodes are equally fast and to do not care for heterogeneity.

The proposed SED mapping for parallel applications searches for “virtually homogeneous” nodes. Therefore, the heterogeneity of the system is transparent. This makes the development of a parallel application much easier.

We present preemptive and non-preemptive SED strategies. The preemptive disciplines are no gang scheduling disciplines, since gang scheduling seems to be not feasible in workstation clusters (see section 3.2).

We introduce the concept of mapping state diagrams to investigate the behaviour of different SED algorithms. In particular, we are interested whether SED may benefit from using migration. Further, we did simulation experiments to test the performance of the algorithms.

Mapping state diagrams have shown to be an useful description tool for investigating the behaviour of the algorithms and to analyse the simulation results.

SED1 scheduling, the concept of mapping state diagrams, and simulation results with the Proof-workload for SED1 are published in [134]. Here, we give an improved definition of mapping state diagrams.

4.1 SED Strategy

The characteristics of SED are

- the user has to specify the maximum and minimum degree of parallelism ($minsize$ and $maxsize$),
- the global run queue is managed in FIFO order,
- migration, if faster machines become available.

Definition of Delay Classes

SED maps a parallel application onto *virtually homogeneous nodes*, i.e., onto machines with the same current delay.

The *delay factor* of a machine M_i at time t is given as

$$d_i(t) := \alpha_i(1 + \text{load}_i(t)),$$

where $\text{load}_i(t)$ is the number of runnable processes on machine i (see chapter 3).

Since we suppose that the applications follow the *strong synchronization model*, i.e. every process of a parallel job tends to synchronize with other processes of the parallel application, the slower machines will slowdown the faster ones.

Definition: The *current delay class* D_k of a parallel job P_k which is assigned to machines M_j, with current delay factors $d_j,\ j = 1, 2, ..., n_k$ is defined as

$$D_k := max\,\{d_j \mid j = 1, 2, ..., n_k\}.$$

We claim that the highest delay of a process is limited by the α_{max} value of the slowest architecture, i.e., SED maps at most one process onto a machine of the slowest architecture.

Let $\alpha_k,\ k = 1, ..., K$ be the different speed factors of the machines. The speed factors are ordered ($\alpha_1 = 1$, $\alpha_K = \alpha_{max}$). We introduce a δ-set, which consists of the representatives of the delay classes:

$$\delta\text{-set} := \{j \mid j \in \mathbf{N} \wedge\ 1 \leq j \leq \alpha_{max}\}\ .$$

There are $maxclass := \mid \delta\text{-set} \mid$ delay classes. The representative delay factor of delay class i is denoted by δ_i, the lower bound by $\delta_{i_{min}}$, and the upper bound by $\delta_{i_{max}}$. The first delay class is $[1, 1.5)$. For all other classes $[\delta_{i_{min}}, \delta_{i_{max}})$, we define

$$\begin{aligned}\delta_{i_{min}} &= i - 0.5,\\ \delta_{i_{max}} &= i + 0.5.\end{aligned}$$

In our example, we have $\alpha_1 = 1,\ \alpha_{max} = 4$ and δ-set $= \{1, 2, 3, 4\}$. The corresponding delay classes are shown in table 4.1.

When the load index of a machine changes significantly, the machine will also change its delay class.

δ_i	$\delta_{i_{min}}$	$\delta_{i_{max}}$
1	1	1.5
2	1.5	2.5
3	2.5	3.5
4	3.5	4.5

Table 4.1: Example of delay classes.

The SED Mapping Algorithm

The components a_i of the *availability vector* $(a_1, a_2, \ldots, a_{maxclass})$ give the number of currently available nodes in class i.

The SED algorithm compares different possible mappings by the resulting expected delay. SED maps an application onto the machines which are currently in delay class m, if

$$\frac{\delta_m}{a_{m_p}} = min\left\{\frac{\delta_i}{a_{i_p}}\middle| i \in \{1, ..., maxclass\} \ \wedge \ a_{i_p} \geq minsize\right\},$$

where $a_{i_p} = min\{a_i, maxsize\}$. If there is more than one delay class which fulfills this equation, we choose the fastest one. Hence, the application is started with the smallest degree of parallelism which gives the best speed-up under the current load situation.

This is done to reduce communication costs. For example, if we map a 2-dimensional multigrid application on 16 nodes instead of mapping them onto 4 faster nodes, we reduce the number of operations per node to a quarter. At the same time, the communication costs are only halved for inner nodes. Since data have to be exchanged between neighbouring nodes, the total communication costs on 4 nodes for each iteration are $4 \cdot 2 \cdot a = 8a$, where a is the length of one square. If the application runs on 16 nodes, the total communication costs will be $24 \cdot 2 \cdot \frac{a}{2} = 24a$.

The chosen delay class m is called the *expected delay class* of the application (EDC). The *expected delay time* of the application (EDT) is $\frac{\delta_m}{a_{m_p}}$.

The Availability Vector

We will compare two SED algorithms, called SED1 and SED2. The algorithms differ in the calculation of the availability vector which leads to different mapping decisions.

SED1 uses an availability vector which is a generalization of the 1-to-1-mapping used in homogeneous systems. There, a parallel application is spread onto the given processors by mapping 1 process onto 1 machine. In delay class i are all machines with current delay less or equal δ_i.

To determine the components of the availability vector, we have to consider the current delay factors $d_i(t)$ of the machines and their *slow–down threshold* $s_i(t)$ which is defined as follows. This threshold gives the maximum delay which can be tolerated on a machine M_i without slowing down one of the assigned applications. We define

$$s_i(t) := min\{\delta_k \mid \text{ a process of application } P_k \text{ runs on } M_i \text{ with EDC} = k\}.$$

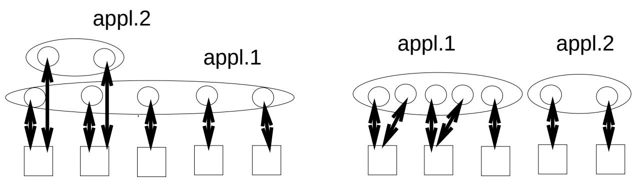


Figure 4.1: Different Mappings of SED1 (left) and SED2 (right).

When the system is idle, $s_i(t) = \alpha_{max}$.

The availability vector is calculated as follows. The components are initialized with zero. If $d_i \leq s_i$, then $a_k := a_k + 1$ for all classes $k \geq d_i$. This is checked for every machine M_i. In our example, when all machines are idle, the availability vector is $(5, 5, 5, 30)$.

The SED2 algorithm counts a machine with current delay 1 as k "virtual nodes" with delay k. For all delay classes $i = 1, ..., maxclass$ and machines $M_j,\ j = 1, ..., n_k$,

$$a_{i_j} = max\,\{k \mid \alpha_j \cdot (k + \text{load}_j) \leq min\,\{\delta_{i_{max}}, s_j\}\,\}$$

is calculated. The a_{i_j} are summed for all machines and give the component a_i of the availability vector. In our example system, this results in $(5, 10, 15, 45)$, when the system is idle.

SED1 and SED2 result in different mapping situations. For example, in a system with 5 machines may be two faster machines with $\alpha_1 = \alpha_2 = 1$ and 3 slower machines with $\alpha_3 = \alpha_4 = \alpha_5 = 2$. Figure 4.1 shows the mapping situation when there are two applications in the run queue, both with $minsize = 1$ and $maxsize = 5$. While SED1 spreads application 1 over all 5 machines, SED2 maps the application compactly onto 3 of the 5 machines.

The values of $d_i,\ s_i$ and $(a_1, a_2, ..., a_{max})$ are updated whenever a parallel job is assigned or terminates.

Termination of a Parallel Job

When a job terminates, the scheduler updates the expected delays and checks whether jobs are waiting in the run queue and may be mapped now. If the run queue is empty, the scheduler checks whether a running job can be migrated to machines of a faster delay class.

For example, when application 1 on the left side of figure 4.1 terminates, application 2 runs alone on the two fast machines with a new expected delay of 1. This scheduling event is called *upgrading*.

Since the application is always running and contained in $load_j$, we cannot calculate D_k as above, but use

$$D_k = max\,\{\alpha_j(1 + \text{load}_j(t) - 1) \mid P_k \text{ is ass. to } M_j\}\,.$$

When a job is upgraded, the thresholds s_i of all machines have to be recalculated. This also effects the availability vector.

4.2 Mapping State Diagrams

We introduce mapping state diagrams to characterize the behavior of the SED algorithm. Mapping state diagrams are distinct from state diagrams which describe the system behavior in terms of number of jobs in the system.

Definition: A *mapping state* is a tuple $(m_1, m_2, ..., m_{maxclass})$ with one component m_i for each delay class. Let ν_i denote the maximal number of virtual nodes in delay class i when the system is idle. The component m_i is a ν_i-tuple, where each entry $m_{i_j},\ j = 1, 2, ..., \nu_i$ gives the number of applications with size j which allocate virtual nodes of delay class i.

Like in state diagrams, state transitions occur when an application arrives or terminates. The possible state transitions are visualized in a mapping state diagram. A *mapping state diagram* is a directed graph whose nodes are the possible mapping states. There is an edge from node A to node B when a state transition exists from A to B.

The advantages of mapping state diagrams compared with normal state diagrams are:

- There are less states since different run queue lengths are not considered. This makes the mapping state diagram more manageable.
- Since the mapping diagram gives information about the current location of the application, migrations are observed as special state transitions.
- Since mapping state diagrams consider virtual nodes, processes which run pseudoparallel on the same machine (due to the local time-sharing system) are also included in the model.

When we consider workloads where the user specifies the maximum degree of parallelism of her application equal to the maximum number of machines or maximum number of virtual nodes, the number of possible mapping states decrease enormously, since a single application tends to fill almost the whole system.

The only information about the workload which we need for constructing the mapping state diagram are the $minsize$ and $maxsize$ parameters. In the following we show the mapping state diagram for different systems in the case of applications which specify $maxsize = 30$ and $minsize = 1$.

When there is only one job running in a delay class, we note only the size of the running job, i.e., a mapping state $(5, 0, 0, 25)$ means that one application is running on 5 machines of delay class 1 and another application is running on 25 machines in delay class 4.

4.2.1 SED1-Mapping State Diagram of System 1

Example system 1 consists of 5 fast machines ($\alpha = 1$) and 25 slower machines with $\alpha = 4$. The mapping state diagram of SED1 for system 1 is shown in figure 4.2.

The mapping diagram shows that SED1 in system 1 has the following characteristics under the given load:

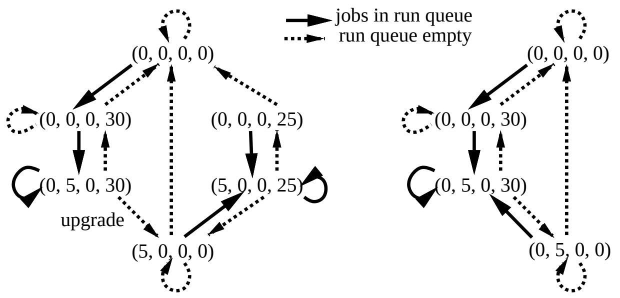


Figure 4.2: Mapping State Diagram of System 1: SED1 (left) and SED1-NU (right) [134].

- At most 2 jobs are running in parallel,
- jobs may run in time-sharing mode (mapping state (0, 5, 0, 30)),
- upgrading may occur, and
- there will be no migration at all.

We will compare SED1 with SED1-NU which uses no upgrading (see fig. 4.2).

4.2.2 SED2-Mapping State Diagram of System 1

When we use SED2 instead of SED1, the availability vector is $(5, 10, 15, 45)$ in case system 1 is idle. SED2 maps the first arriving application also on 30 virtual nodes of delay class 4. But SED2 uses the fast machines first. There will be 4 processes running on each of the 5 fastest machines and only 10 of the slower machines are used. The availability vector is now $(0, 0, 0, 15)$.

The next arriving job will allocate the remaining 15 machines. Now, there are 2 jobs running on machines of delay class 4. We denote the corresponding mapping state as $(0, 0, 0, 30 + 15)$. The complete mapping state diagram is shown in figure 4.3. While no upgrading occurs, SED2 makes use of migration. The mapping state diagram of SED2-NM which does not use migration is also shown in figure 4.3.

4.2.3 SED1- and SED2-Mapping State Diagram of System 2

In system 2 there are 20 machines with $\alpha = 1$ and 10 machines with $\alpha = 4$. Here, SED divides the system into two homogeneous subsystems: One consists of the 20 fast and one of the 10 slower machines. Since SED1 and SED2 behave the same in homogeneous systems, the mapping state diagrams are exactly the same (see figure 4.4).

The mapping diagram shows that SED has the following characteristics in system 2.

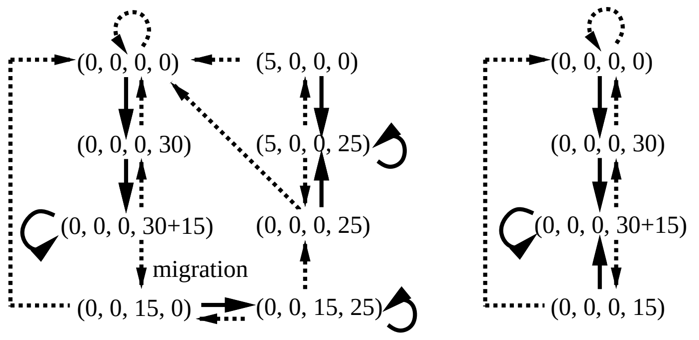


Figure 4.3: Mapping State Diagram of System 1: SED2 (left) and SED2-NM (right).

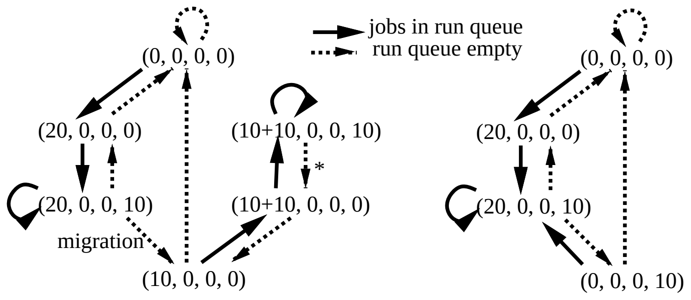


Figure 4.4: Mapping State Diagram of System 2: SED2 (left) and SED2-NM (right) [134].

- At most 3 jobs are running in parallel (state $(10 + 10, 0, 0, 10)$),
- 1-to-1-mapping between processes and machines,
- space-sharing between applications,
- there will be no upgrading at all, and
- migration may occur.

We will compare SED2 with SED2-NM that uses no migration. The Mapping state diagram of SED2-NM is similar to the left side of figure 4.4.

4.3 Evaluation of Simulation Results

Oleyniczak has implemented a simulation program [111] for the presented scheduling strategies and has created the figures shown in this chapter. The simulation software uses parts from simuLan [114].

The approximation of migration costs is the same as in case of the homogeneous system (see section 2.2).

4.3.1 Definition of Workloads

In the simulations, we investigate the algorithms under four different workloads. Three follow the variable–size–model and the fourth the fixed-size–model. The given mean value of service times are the execution times when the application would run sequentially on one of the fastest machine in the system ($\alpha = 1$).

Proof-Workload: In the first workload we use the variable–size–model with $maxsize = 30$ and $minsize = 1$. Since $maxsize$ is equal to the number of machines in the systems, the parallel jobs will act greedily and allocate as many nodes as possible. To get a better understanding of the behavior of the algorithms under different hardware characteristics and to test the correctness of the simulation software, we use an artificial workload with constant service demands (100 min).

Exp-Workload: In the Exp-workload we use an exponentially distributed service demands with a mean service time 120 minutes.

Hyperexp-Workload: In the Hyperexp-workload the service demands are hyperexponentially distributed where 50 % of the jobs have a mean service time of 10 minutes and 50 % of the jobs have a mean service time of 120 minutes. Hyperexponentially distributed service demands are motivated by several monitoring studies in parallel computer centers where service demands with coefficient of variation between 1.3 and 3.7 have been observed (see for example [34]).

FS-Workload: Here, the service demands are hyperexponentially distributed like in the Hyperexp-workload. The FS-workload follows the fixed-size–model where the size is uniformly distributed between 1 and 30.

We compare the performance of the SED algorithms in the two example systems. The coefficient of variation of the generated random numbers for the service demands is about 0.9 in case of the Exp-workload resp. 1.6 for the Hyperexp-workload.

4.3.2 Performance Results for the Proof-Workload

We use this workload for the discussion of the upgrading technique and of migration of SED1.

The results of our simulations are shown in figure 4.5. The mean residence time, the mean waiting time, and the mean computing time of the jobs are given as a parameter of the job arrival rate. Further, the percentage of upgraded resp. migrated processes is shown.

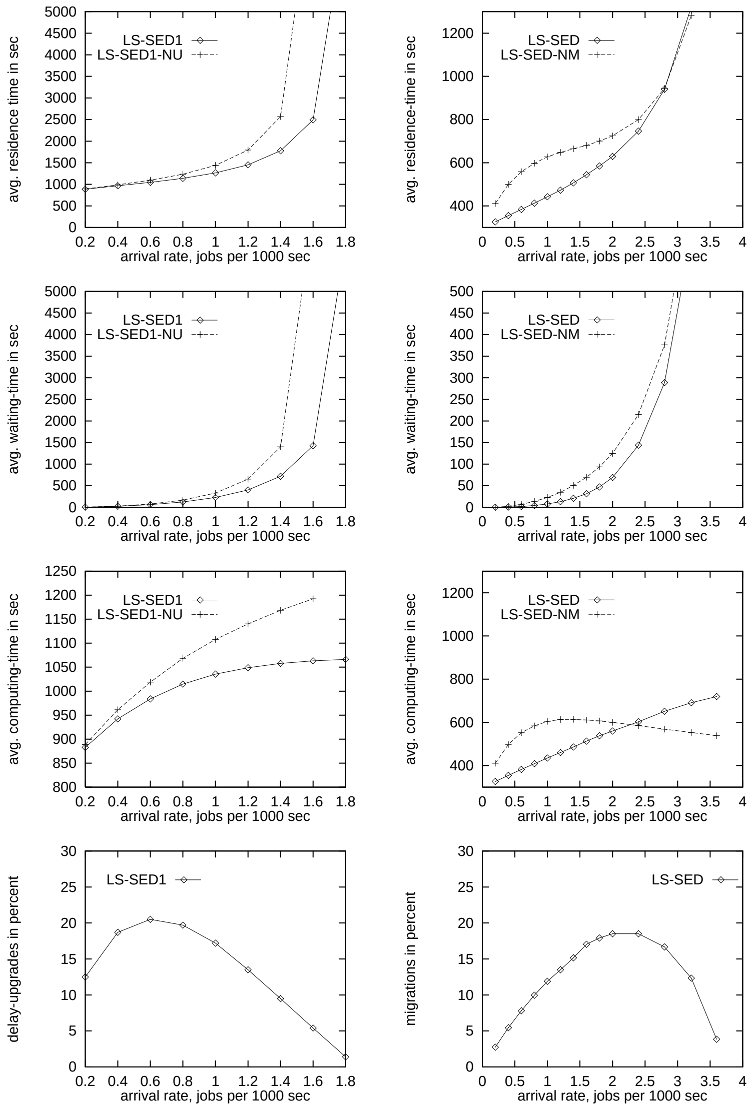


Figure 4.5: Simulation results with System 1 (left) and System 2 (right) under Proof-workload.

Influence of Upgrading

For system 1, upgrading reduces both waiting and computing time. The percentage of upgrades decreases with increasing system load. This occurs, since upgrading is done only when the run queue is empty.

Nevertheless, upgrading leads to a better performance even under high load (see figure 4.6). The reason can be found by observing the corresponding mapping state diagram.

In case of SED1-NU the corresponding mapping state for higher load is $(0, 5, 0, 30)$. The *asymptotic computing time* is

$$c_{limes} = \left(x \cdot \tfrac{2}{5} + y \cdot \tfrac{4}{30}\right) \cdot t,$$

where t is the mean service time demand of the jobs ($t = 100min$), and the weights x and y are the solution of the equations $x \cdot \frac{2}{5} = y \cdot \frac{4}{30}$ and $x + y = 1$. In this case there will be 3 jobs running with delay $\frac{4}{30}$ while one job runs with delay $\frac{2}{5}$. This gives $c_{limes} = \frac{2}{10} \cdot t = 1200s$ (see fig. 4.5).

In case of SED1 the system may change between the states $(5, 0, 0, 25)$ and $(0, 5, 0, 30)$. The asymptotic computing time in state $(5, 0, 0, 25)$ is

$$c_{limes} = \left(x \cdot \frac{1}{5} + y \cdot \frac{4}{25}\right) \cdot t,$$

with weights $x = \frac{4}{9}$ and $y = \frac{5}{9}$. The resulting asymptotic computing time is about 1060 s which is the upper boundary of the SED1 curve in figure 4.5.

Discussion of Migration

The mapping state analysis has shown that under the given workload migration occurs only in system 2. Further, SED1 and SED2 behave the same in system 2.

The simulation shows the benefits of migration under low and medium load in system 2 (see fig. 4.5). Under low load most jobs will be mapped onto the 20 fast machines and migration rarely occurs. The percentage of migrated processes increases with increasing load. Under higher load the run queue will seldom be empty and the probability of migration decreases again.

Figure 4.5 shows that migration reduces the mean waiting time. SED1 also reduces the computing time up to an arrival rate of about 2.3 jobs per 1000 seconds. Here, the jobs benefit from the shorter runtime on the faster machines after migration. But for higher load, SED1-NM leads to the smaller mean computing time. The reason is that the two strategies result in different asymptotic computing times.

Without migration most of the time the system will be in state $(20, 0, 0, 10)$ under high load. The corresponding computing time is $\frac{1}{9}\left(8 \cdot \frac{1}{20} + 1 \cdot \frac{4}{10}\right) = 533, \bar{3}$ seconds. When SED1 is used the system may be in mapping state $(20, 0, 0, 10)$ or $(10{+}10, 0, 0, 10)$. The latter has a computing time of $\frac{1}{9}\left(4 \cdot \frac{1}{10} + 4 \cdot \frac{1}{10} + 1 \cdot \frac{4}{10}\right) = 800$ seconds which increases the mean computing time of SED1 compared with SED1-NM.

The different behavior of upgrading and migration is summarized in figure 4.6 where the speed-up of SED1 against SED1-NU in system 1 resp. SED1-NM in system 2 is shown. The speed-up of SED1 against SED1-NU is defined as

$$\text{speedup (SED1)} = \frac{\text{res. time of SED1-NU} - \text{res. time of SED1}}{\text{residence time of SED1}},$$

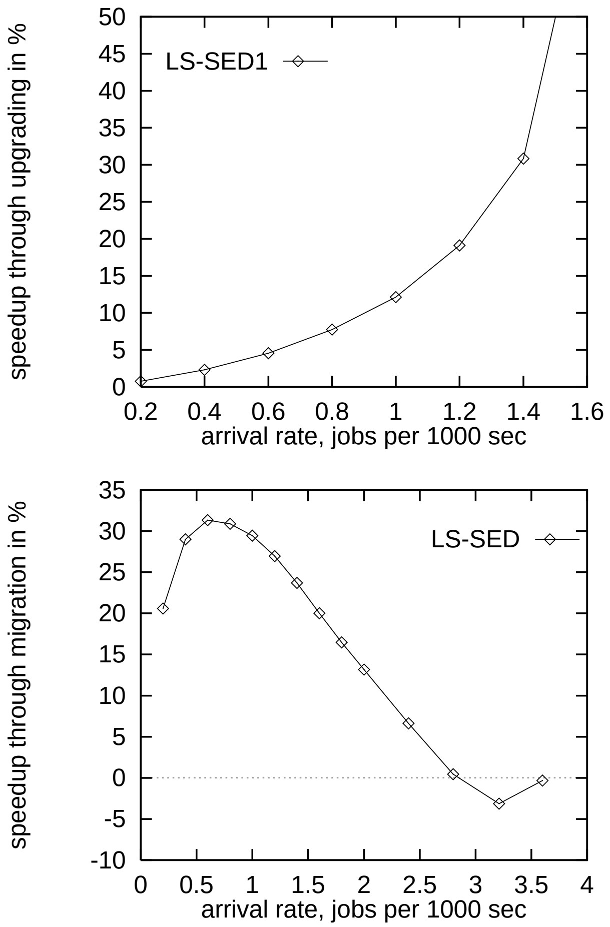


Figure 4.6: Speed-Up of SED1 by upgrading in system 1 (top) and by migration in system 2 (bottom) under the Proof-workload.

and similar for SED1-NM. While the benefits of upgrading increase under higher load, SED1 with migration shows substantial benefits under low and medium load. It should be clear that the possible speed-up depends on the hardware characteristics of the system.

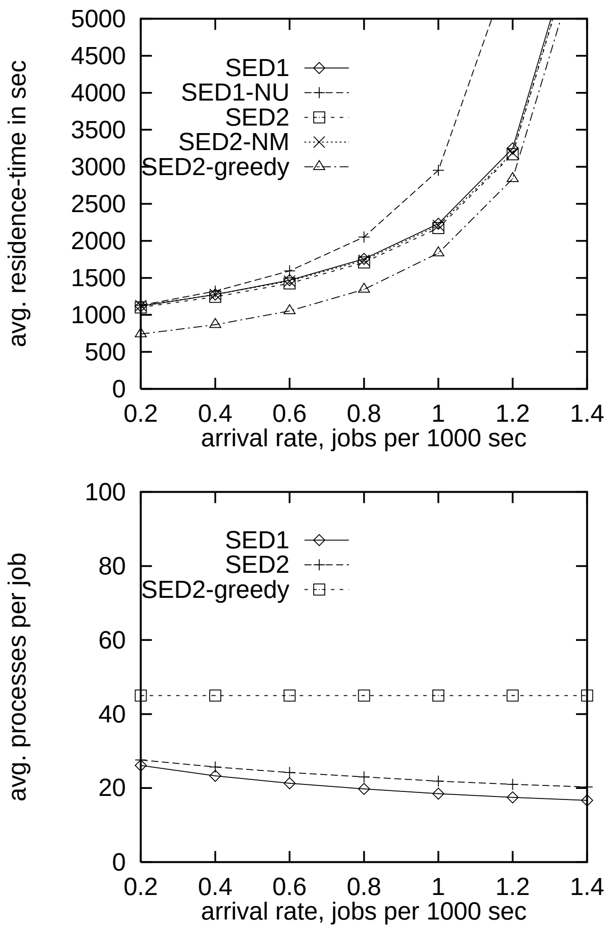


Figure 4.7: Mean residence time (top) and job size (bottom) for system 1 for the Exp-workload.

4.3.3 Performance Results for the Exp-Workload

The mean residence times for the different algorithms in system 1 and system 2 are shown in figure 4.7 and 4.8. Since SED1 and SED2 behave the same for system 2 under the given workload, only curves labeled SED are shown (see figure 4.8).

In system 1, the simulation results for SED1 and SED2 show a very similar behavior. The

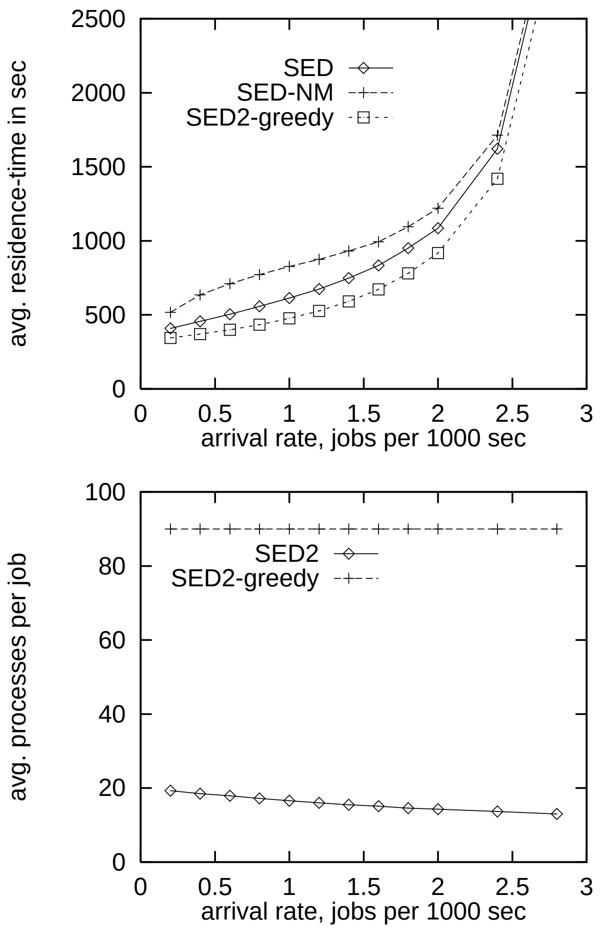


Figure 4.8: Mean residence time (top) and job size (bottom) for system 2 for the Exp-workload.

best performance is achieved by SED2 in both systems. Migration and upgrading improves the performance of SED.

Influence of the Expected Job Size

Both SED1 and SED2 are compared for $maxsize = 30$ which is equal to the number of machines in the system. We also tested the SED strategies with $maxsize = 100$.

For SED2, the maximum number of virtual nodes in system 1 is 45, and 90 in system 2. Hence, SED2 maps a job onto all virtual nodes and becomes a simple FIFO strategy. The simulation results for SED2 are labeled as SED2-greedy in figure 4.7 and 4.8. The results for SED1 stay the same, since the maximal number of virtual nodes is equal to the number of machines. The figures show that SED2 makes efficient use of the resources with this workload in both systems, even under high load.

Since SED2 tends to map more processes onto a machine than SED1 under this greedy workload, this may result in a lack of memory resources and the machines may start swapping. So, in a real implementation of the SED algorithms, the available memory resources have to be considered.

Influence of Migration

The benefits of migration are shown in figure 4.9, where the speedup of SED2 against SED2-NM is plotted. While migration has only less benefits in system 1 (less than 5 % speedup), the speedup achieved in system 2 increases up to nearly 30 %. The benefits of migration decrease under higher load.

The reason of the different behavior of SED in the two systems can be found by observing the corresponding mapping state diagrams (see figure 4.3 and refmap2). While in system 1 SED migrates processes from machines of delay class 4 to machines of delay class 3, the processes can be migrated to machines of delay class 1 in case of system 2. This results in an overall better speedup.

Since the possible speedup depends on the hardware characteristics of the system, a change of the α values will result in higher or lower speedups.

Migration is an operation which consumes processor time and network capacity. Since the network is highly loaded during the transfer time of a process, migration events are “visible” to all users.

The number of migrations and the time between the migration events depend on the job arrival rate and the job service demands (see figure 4.9). The smallest observed interarrival time is observed in system 2 (1919.0 sec for arrival-rate 2.8). This seems acceptable for users in a workstation cluster. Further, when the mean service demands of the jobs increase, we expect that the mean interarrival times will also increase.

Since migration occurs only when the global runqueue is empty, migration events should decrease under higher load. This is not true for system 2. Here the mean time between migration events stay almost constant for medium and higher load. The mapping state diagram (see fig. 4.4) shows that even under high load migration may occur (state transition $(10 + 10, 0, 0, 10) \rightarrow (10 + 10, 0, 0, 0)$).

Here, we investigate migration events which occur when faster machines get available. If more than one process is migrated, we call this a *migration bulk*. The length of a bulk is the

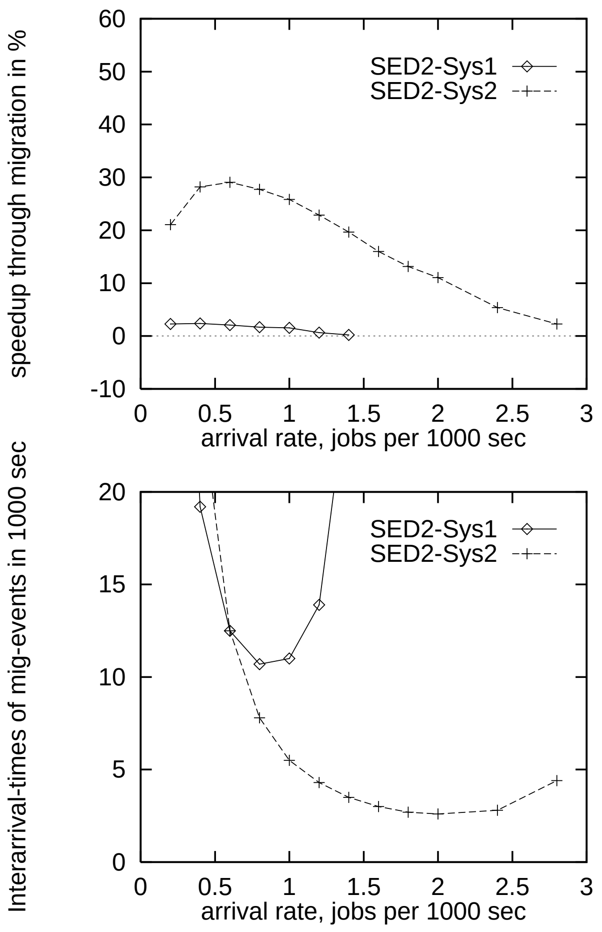


Figure 4.9: Speedup of SED2 through migration (top), and mean time between migration events (bottom) for the Exp-workload.

number of migrated processes and gives information about the costs. The mean costs of migration in system 1 are higher than in system 2, since SED2 migrates 15 processes per migration event in the case of system 1, and 10 processes in the case of system 2 (see fig. 4.3 and 4.4). But overall, since the migration costs are several seconds for 10 MB processes, they are neglectable

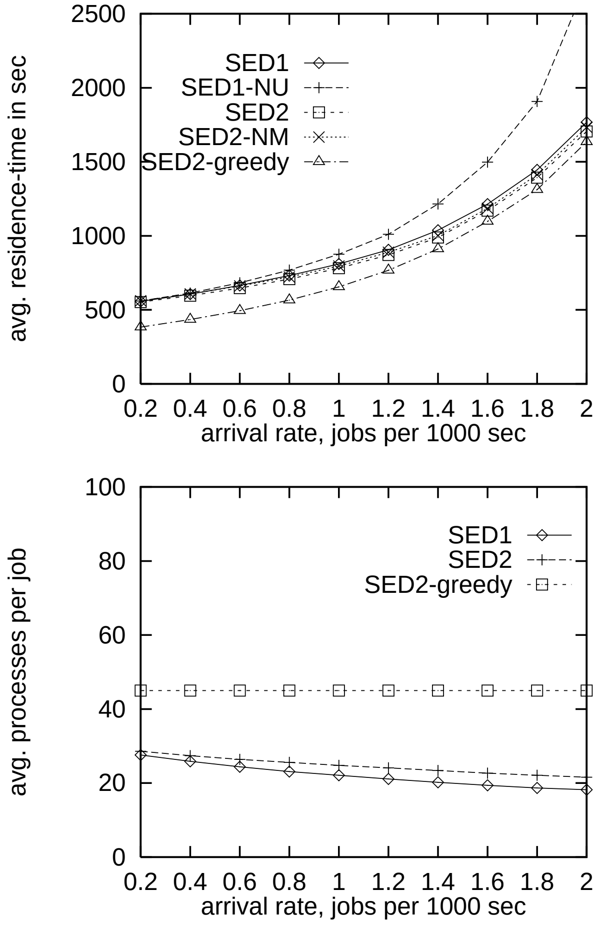


Figure 4.10: Mean residence time (top) and job size (bottom) for system 1 for the Hyperexp-workload.

compared with the service demands of the jobs.

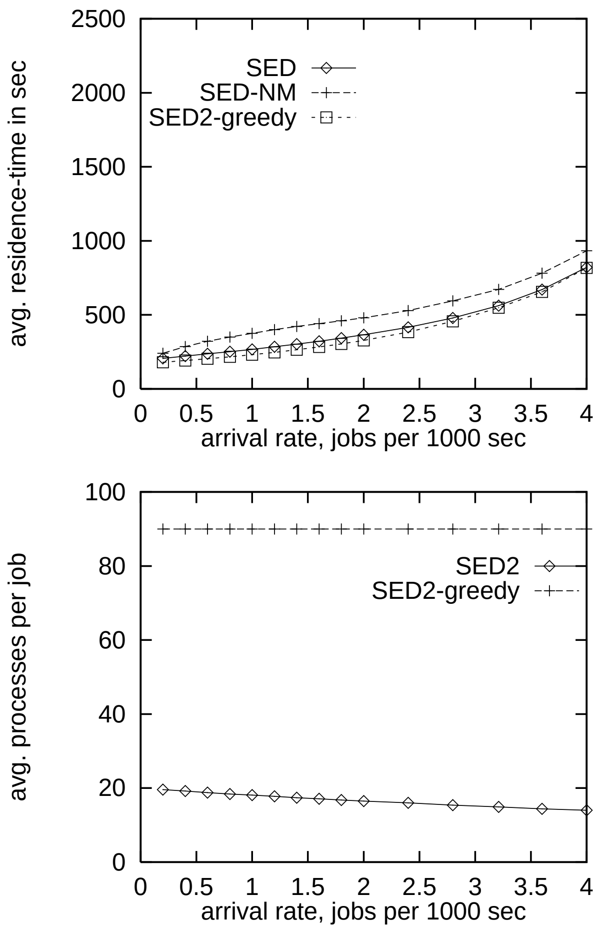


Figure 4.11: Mean residence time (top) and job size (bottom) for system 2 for the Hyperexp-workload.

4.3.4 Performance Results for the Hyperexp-Workload

The Hyperexp-workload differs from the Exp-workload only in the distribution of the service demands. The coefficient of variation of this workload is 1.6, and it is about 0.9 for the Exp-workload.

The mean residence times for the different algorithms in system 1 and system 2 are shown in figure 4.10 and 4.11. Since much more jobs with a shorter service demand arrive, the mean residence times are smaller compared to the Exp-workload. However, we are not interested in the absolute performance values, but in the behavior of the algorithms compared with each other.

Comparing the figures for Exp-workload 4.7 and 4.8 with the figures 4.10 and 4.11 for Hyperexp-workload, we see that the different disciplines have a similar behavior under both workloads. Again, SED2 achieves the best performance in both systems.

The benefits of migration are shown in figure 4.12, where the speedup of SED2 against SED2-NM is plotted. In system 2, a maximum speedup of 30 % is achieved by migration and a minimum speedup of 10 % even under high load. This differs from the results under Exp-workload where the benefits of migration decreased for higher load. If we do not use migration, each job which is mapped on the slower machines runs there up to completion. But if migration is possible, the long running jobs which are mapped onto the slower machines migrate to the faster ones when they become available. The faster machines may be occupied by a long or short running job. If it is also a long running application, the probability for migration is less, but on the other side it is not neccessary for the overall performance. In case it is a short running application, the probability of migration increases. Hence, under the Hyperexp-workload a higher percentage of applications will be served by the faster machines due to migration.

The smallest interarrival time is again observed in system 2 (1346.8 sec for arrival-rate 3.6). This seems acceptable for users in a workstation cluster.

Since the possible speedup depends on the hardware characteristics of the system, a change of the α values will result in bigger or smaller speedups.

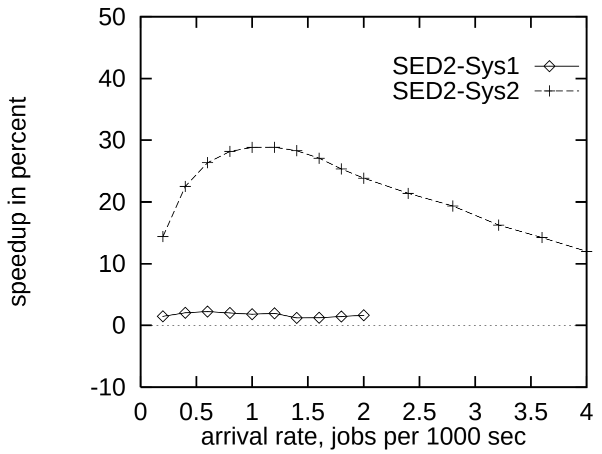
4.3.5 Performance Results for FS-Workload

Figure 4.13 shows that SED2 performs better than SED1 in both systems. Since the number of available nodes is much higher under SED2, this discipline is much more flexible to handle the fixed-size–workload.

In contrast to the results under variable–size–workload, upgrading makes the performance of SED1 worse. When an application is upgraded, this means it uses more virtually nodes, since it will use some machines exclusively after upgrading. This means the number of available nodes is reduced by upgrading. This makes SED1 less flexible for the FS-workload.

The performance of SED2 is again improved by migration. For the FS-workload, the speedup of SED2 against SED2 without migration increases up to about 20 % in both systems. The speedup curves for both systems are shown in figure 4.14 (labeled Fix-SED2-Sys1/2) together with the speedup curves under the variable–size–workload (labeled Var-SED2-Sys1/2). While there are less benefits of migration for the variable–size–workload in system 1, the situation is quite different for the FS-workload. In systems 2, there are performance benefits observed under both types of workload. While under the variable–size–workload the benefits of migration decrease under higher load, the speedup by migration increases under higher load in the case of FS-workload.

The number of migration events is much higher under FS-workload (see figure 4.14). Since applications with sizes between 1 and 30 arrive, the probability that a smaller job may migrate to



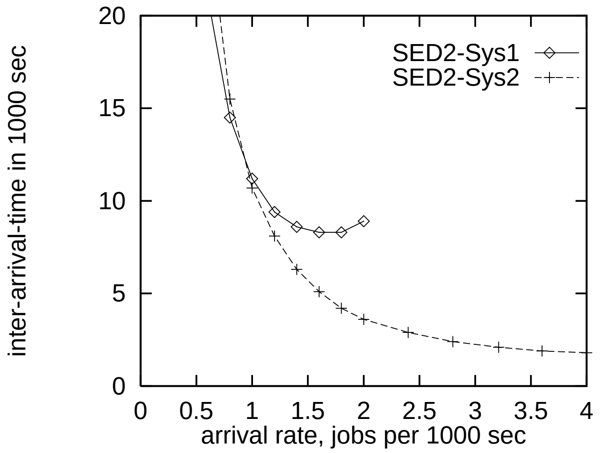
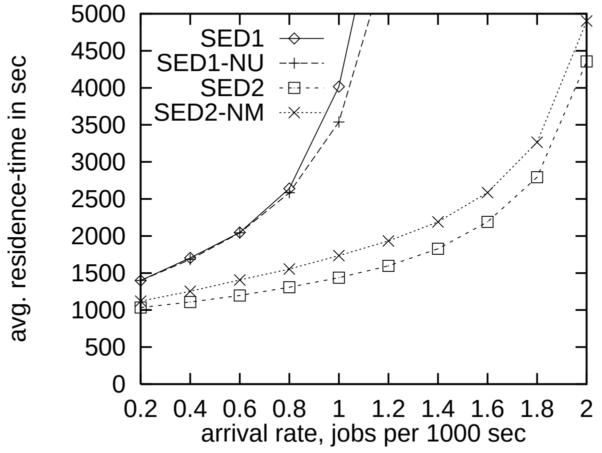


Figure 4.12: Speedup of SED2 through migration (top), and mean time between migration events (bottom) for the Hyperexp-workload.

the faster machines is much higher.



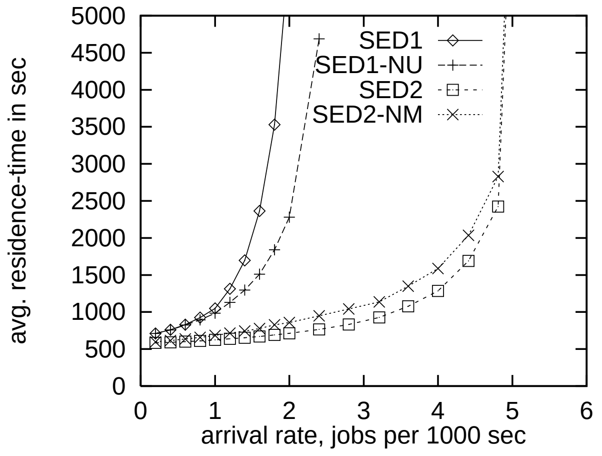
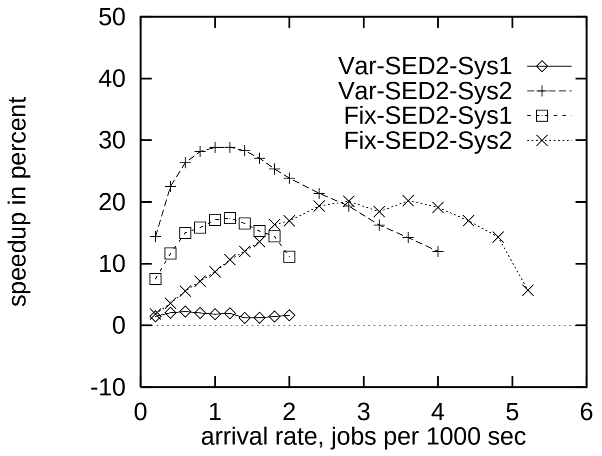


Figure 4.13: Mean residence time for system 1 (top) and system 2 (bottom) for FS-workload.

4.3.6 Influence of Hardware Characteristics

We expect that the speedup which is achieved by migration will be higher when we replace machines by faster ones.

Therefore, we tested SED for systems with different hardware characteristics. The systems are given in table 4.2. These are the example systems 1 and 2 as before, and system 2' which



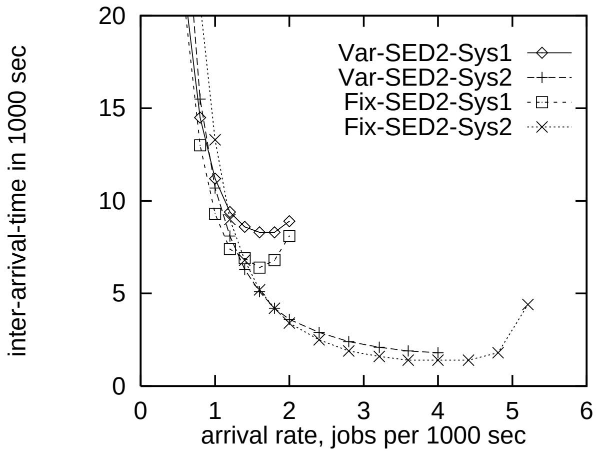


Figure 4.14: Speedup of SED2 through migration (top), and mean time between migration events (bottom) for FS-workload.

consists of 20 machines with $\alpha_1 = 1$ and 10 machines with $\alpha_2 = 10$. This means that 20 machines are replaced by faster ones compared with system 2.

The mapping state of system 2’ is the same like system 2 (see fig. 4.4). In system 3, most of the jobs are executed on the faster machines with speed factor 1 resp. 4. The 5 slowest machines

System	$\alpha_1 = 1$	$\alpha_2 = 4$	$\alpha_3 = 10$
1	5	25	-
2	20	10	-
2'	20	-	10
3	5	20	5

Table 4.2: Systems with different hardware characteristics.

will only be used, if the others are occupied.

The different speedups and the mean time between migration events are shown in figure 4.15. These are results from the simulation with the Hyperexp-workload.

The 'individual speedup' of a job achieved by migration is

$$\frac{EDT_{old}}{EDT_{new}},$$

where EDT_{old} is the expected delay time before, and EDT_{new} the expected delay time after migration. Since the size of the application is unchanged during migration, this is equal to

$$\frac{EDC_{old}}{EDC_{new}},$$

the quotient of the corresponding expected delay classes.

The individual speedup of processes in system 2 is $\frac{4}{1} = 4$, in the case of system 2' it is $\frac{10}{1} = 10$. This means that for the individual process it is much more beneficial to migrate in system 2'. Figure 4.15 shows that this corresponds also to a higher social speedup as expected.

The same is true for system 3 where a higher individual speedup results in a higher social speedup compared with system 1 and 2.

4.4 Summary

We have presented and compared SED mapping strategies for parallel applications on heterogeneous systems. SED make the heterogeneity of the system transparent for the user.

We introduced the concept of mapping state diagrams to characterize the behavior of the algorithms. Further, we tested their performance in a simulation.

The simulations have shown some of the dependencies between the system configuration, workload characteristics, and scheduling performance.

The presented simulation results show the benefits of the upgrading technique used by SED under the variable–size–workload. Further, process migration leads to an improved performance. The overhead which is caused by migration is expected to be tolerable, since the observed numbers of migrations which are neccessary to improve the performance is very low.

The presented SED1 and SED2 policies differ in their calculation of the availability vector. SED2 makes better use of the resources and shows a better performance in the simulated systems for variable–size and fixed-size workloads.

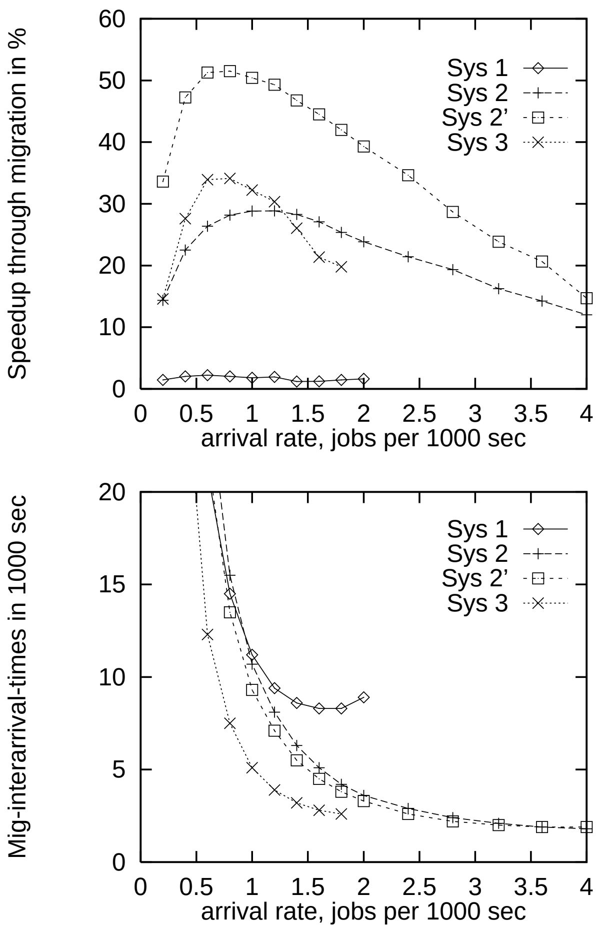


Figure 4.15: Speedup of SED2 through migration (top), and mean time between migration events (bottom) for Hyperexp-workload in different systems.

4.5 Bibliography

An implementation of SED2 scheduling for PVM applications is described in [49].

A time-sharing scheduling strategy which shall be implemented in the context of MPVM [19] is presented by Al-Saqabi, Otto and Walpole [3]. They use the concept of ‘virtual processors’ which is similar to our ‘virtual nodes” to make effcient use of heterogeneous machines but the algorithm is based upon the fixed-size-model. The algorithm calculates the placement of a job in three steps (Minimum Turn Around Time Algorithm, Compression Algorithm, and Expansion Algorithm). The authors do not investigate whether a preemptive policy is beneficial on workstation clusters.

Chapter 5

Dynamic-SED

In this chapter, we extend SED2 to Dynamic-SED to make it operational in ‘real’ workstation environments. Dynamic-SED regards not only the current delay of the machines, but also the currently free memory and the number of interactive users. It presents a new approach to achieve a co-existence between parallel applications and interactive users.

The presented location and selection rules are part of Marc Gehrke’s master thesis [49]. Gehrke has implemented SED for PVM applications and carried out the presented trace-driven simulation. Further, we define different ‘migration anomalies’ and check whether these anomalies are true for Dynamic-SED or not.

Several studies about migration strategies have been published (see for example [31, 81, 58, 164]). They all have in common that they investigate migration policies in homogeneous systems with no special care of parallel applications. Further, they all use a fuzzy-like classification of lowly, medium, and highly loaded machines, and they are not aware of interactive users.

While it has been argued that migration is not always beneficial for load balancing in homogeneous systems [31], the situation is different in heterogeneous systems where applications may profit from migration to faster hosts.

5.1 Motivation and Definitions

Up to now, we tested the performance of the SED algorithms under “closed conditions”, i.e., the only load in the system was due to parallel jobs. As long as all jobs are submitted to the global scheduler and no local ‘background’ load occurs the scheduler will take care that the delays of the applications will be guaranteed [1]. But when jobs arrive which are not mapped by the global scheduler like jobs from interactive users, applications may get out of their delays.

Workstations are user dedicated machines and only their idle times should be used for long running and parallel applications. The question is how both, parallel application and interactive user, can coexist. In this chapter, we present Dynamic-SED, where reservation and migration rules are added to the SED discipline.

[1] See the definition of the slowdown-threshold.

The SED discipline maps a parallel job onto machines which are currently in the same delay class. When the user of a machine gets active again, the local load on the machine may rise. This means that the application is slowed down by the user.

On the other side, the user is handicapped by the parallel application which consumes processor and memory resources. Even in the case that the user does not need a lot of CPU capacity when starting a text editor or an Web browser for example, he will need some MBytes of free memory for a pleasant operation. The user has different opportunities to act in this situation:

- He does a `remote login` on another lower loaded host which benefits the parallel application.
- He is frustrated and does a log off which also benefits the parallel application.
- He can urge the system administrator to stop the application.
- He ignores the slowdown and keeps on working.

We will favour the last possibility in our scheduling discipline. The scheduling discipline shall try to fulfill two principles [49]:

Principle 1: The user who wants to work at a workstation takes precedence over the parallel jobs.

Principle 2: The expected delay of the application shall be constant or decrease.

The only solution to fulfill these principles in the presence of interactive users is to migrate a parallel application when the resources get rare. Principle 2 states further that the target node should be within the delay class of the application.

In the next sections, we will present the details of Dynamic-SED that tries to fulfill these principles. An evaluation of an experiment with this discipline in a real workstation cluster is presented in section 5.5.

First, the system has to recognize interactive users [49].

Definition: An *interactive user* is a user who is logged on and regularly submits jobs to the system.

Some systems like Condor [96] observe whether there is someone active on the console or on the mice device of the machine. But in a distributed environment people are used to work remote on different machines, in special when there are some faster machines in the system available. In this case the console and the mice device will be meaningless.

Further, it is possible to check periodically whether a user is active. In a UNIX environment, the output of the `top`-command displays information about processes. The raw cpu percentage is used to rank the processes. Since `top` itself is a resource intensive operation, its use is rejected in the design.

The notice whether there is any user logged on, we use the `who`-command which lists the login name, terminal name, and login time for each current user. These informations are read from the `/etc/utmp` file.

In the description of the migration rules, we use the following definition [49]:

Definition: Each entry in the `/etc/utmp` file defines an *interactive user*.

The benefit of this definition is that the system can periodically check whether there are entries in the `/etc/utmp` file and identify interactive users.

The proposed strategy acts preventively and reserves resources for the interactive user. When these reserved resources are not enough to guarantee the claimed principles, migration of a process is necessary. The reservation, selection, and location policy is described in the following sections.

5.2 Resource Reservation

Since interactive users shall not be disturbed by the parallel applications, we reserve cpu and memory resources. Therefore, we introduce two parameters, $load_{reserv}$ and mem_{reserv}, which give the percentage of cpu time and the amount of memory which shall be reserved for interactive processes.

Since swapping will slowdown the performance, machines which seem to be memory critical should no longer be available in the delay classes. The parameter mem_{min} gives the lower bound for free memory. If the free memory drops under this bound, the probability of swapping is increased.

The components a_{i_j} for machine j with current load $load_j$ and current free memory $memory_j$ are calculated as follows [49]:

$$\overline{\text{load}}_j = \begin{cases} \text{load}_j & \text{if there are no interactive users} \\ \text{load}_j + load_{reserv} & \text{if there are interactive users} \end{cases} \tag{5.1}$$

$$mem_j = \begin{cases} memory_j & \text{if there are no interactive users} \\ \\ memory_j - mem_{reserv} & \text{if there are interactive users} \end{cases} \tag{5.2}$$

$$a_{i_j} = \begin{cases} max\{k \mid \alpha_j \cdot (k + \overline{\text{load}}_j) \le min\{\delta_i, s_j\}\} & \text{if } mem_j > mem_{min} \\ 0 & \text{otherwise} \end{cases} \tag{5.3}$$

Dynamic-SED reserves some amount of resources for interactive users. This amount is chosen to be independent from the number of current users. Since interactive users tend to do other things like ‘thinking’, they will not start processes all the time. Hence, the ‘reserved resources’ may be shared between the different users.

5.3 Migration Rules

The introduced resource reservation is suitable to protect the users who are currently working. Future users who will log on later cannot be regarded. The only possibility to react on load changes like additional interactive users is migration. The benefits of migration for load balancing are twofold. First, the new interactive user is no longer handicapped by the parallel application, and second, the parallel application can gain performance.

Migration strategies consist of a *selection* and a *location* policy.

Definition: The selection policy determines the migration candidate, i.e., it determines which process on which machine has to be migrated. The location policy chooses the destination host for a migration candidate.

Several studies about migration strategies have been published (see for example [31, 164]). They all have in common that they investigate migration policies in homogeneous systems with no special care of parallel applications and that they are not aware of interactive users.

Here, we add selection and location rules for parallel applications to the SED algorithm (see paragraph 4).

5.3.1 The Selection Policy

In previous papers, the selection policy is always host-oriented. In the first step, the load balancing system checks whether there are overloaded hosts. If there are, the system will select the migration candidate. In [164] for example, policies are compared which select the most computation-intensive process, which select the job which has been running for the longest time so far, and combination of these strategies.

The selection policy for SED is process-oriented. Since SED tries to guarantee an expected delay, the process becomes a migration candidate when the current delay gets significantly worse than the expected one. Further, a process may get a migration candidate if its host has less free memory and swapping may occur. Finally, a high number of interactive users may cause a process to become a migration candidate [49].

Definition: A process is called a *migration candidate* if at least one of the following properties holds:

1. The process was started with expected delay δ_i and the current delay of its host j is d_j with $d_j - \alpha_j > \delta_i^{max}$.
2. The process is running on host j with $memory_j < mem_{min}$.
3. The process is running on host j where the number of interactive users exceed the maximum number which are permissible on host j.

The current delay d_j of machine j includes the already running process. Hence, we have to subtract this process from the current load to calculate the current expected delay:

$$\alpha_j(1 + (load_j - 1)) = d_j - \alpha_j > \delta_i^{max}.$$

We assume in this definition that the process is computation-intensive and causes a load of approximately 1.

The reservation rule reserves resources for interactive users, but it is assumed that only few users are active at once. Hence, to many users rises the probability that at most one user will be handicapped. The maximum number of permissible users is dependent on the architecture. A machine with less main memory has a smaller maximum number than a fast machine with a lot of memory.

Migration itself is a resource consuming process [116, 49]. The state of the process has to be saved. Then, the process has to be transmitted over the network to the new destination host. During the transmission phase, almost all of the net capacity will be used for migration and all other processes which want to make use of the network, like the file server for example, are delayed. This will indeed worry all users. Hence, migration has to be used carefully.

Therefore, the global scheduler checks all of its processes periodically whether they are migration candidates or not. When it was noticed to be a migration candidate for several times, the process will be selected.

The Selection Algorithm

The selection policy uses the following parameters [49]:

δT:	time between two checks,
c:	counter for positive checks, i.e., detecting that the process is a migration candidate (initially zero),
c^1_{max}:	upper bound of positive checks without action if there is no interactive user,
c^2_{max}:	upper bound of positive checks without action if there is an interactive user.

The counter c is used to avoid unnecessary migrations. Two different upper bounds are used to make the policy more sensitive in the presence of interactive users (choosing $c^1_{max} > c^2_{max}$). The algorithm works as follows:

1. Each process is periodically checked (δT seconds) and its counter is updated:

$$c = \begin{cases} c+1 & \text{if the process is a migration candidate} \\ c-1 & \text{if the process is no migration candidate} \end{cases} \tag{5.4}$$

2. If $c = c^1_{max}$ (no interactive users) resp. if $c = c^2_{max}$ (interactive users), the process is selected for migration.

5.3.2 The Location Policy

After selecting a process for migration, the destination host has to be determined. This is done by the location policy.

The main location rule for SED is very simple.

Main Location Rule: When the process was started in delay class i, then any machine which belongs currently to delay class i will be an adequate destination host.

Only when the delay class is empty, the situation gets more difficulty. Then, the location policy depends on whether there are interactive users on the origin host or not. If there are interactive users on the origin host, the need for migration is much higher than without interactive users.

When the delay class is empty, the process has to accept a slowdown. The set of possible destination hosts P is the set of hosts without interactive users and enough memory. Otherwise, we would migrate also the problem.

$$P := \{k \mid \text{there are no interactive users on host } k \text{ and } memory_k > mem_{min}\} \tag{5.5}$$

Interactive Users on the Origin Host

In this case the process has to be migrated for the benefit of the interactive user [49].

Let j be the origin host. The destination host is the host with the lowest current delay which has at least as much memory as the origin host. Since p is already running on j, we have to recalculate the available memory. Let $memory\_usage(p)$ be the memory currently used by p, then

$$memory'_j := memory_j + memory\_usage(p),$$

and

$$d_{dest} = \min\{d_k \mid k \in P \wedge memory'_j \leq memory_k \wedge d_k < d^{max}\}. \tag{5.6}$$

The upper bound d^{max} limits the number of processes on a host. Otherwise, single hosts without interactive users could be hopelessly overloaded.

If a destination host still cannot be found, the memory condition is dropped:

$$d_{dest} = \min\{d_k \mid k \in P \wedge d_k < d^{max}\}. \tag{5.7}$$

If still no destination host can be found, the user has to accept the situation.

No Interactive Users on the Origin Host

In this case, the process has dropped out of its delay class and/or memory is scarce. After migration the delay of the process should be substantially improved [49].

Now, the origin host j belongs to P. Since the process is already running on j, the delay of p on j is

$$d'_j := d_j - \alpha_j.$$

We claim that the delay of the destination host should be at least better than one α_j.

$$d_{dest} = \min\{d_k \mid k \in P \wedge memory'_j \leq memory_k \wedge d'_j - d_k > \alpha_j\}. \tag{5.8}$$

5.4 Dynamic-SED is Ping–Pong–Free

Introducing migration rules into a resource management system keeps the danger that processes are migrated around like nomads.

Gehrke reports that during the simulation no *Ping-Pong-effect* was observed. He describes Ping-Pong-situations as follows [49]:

- A process p is migrated from host A to host B.
- Now the load on host A decreases and the situation is fine on host A.
- Since p is added to the load on host B, processes on B may become migration candidates.
- Since the load situation is normalized on A, location A may look better than B and p is migrated back.

A solution for this problem is to use a migration counter. Each time a process is migrated, its counter is increased. When the counter has reached a threshold which gives the maximum number of allowed migrations, the process will not be migrated any more. This solution is simple and recommended within load balancing strategies (see for example [59]).

In a workstation cluster, migration is not only used for load balancing, but for the benefit of interactive users. Further, jobs with a runtime of several days or weeks will probably migrate more often than a short job of some hours. Hence, the threshold should give a maximum number of migrations per hour for example.

In the following, we will show that the migration rules of Dynamic-SED avoid Ping-Pong situations without using a threshold. We introduce further migration properties and check whether these properties are true for Dynamic-SED.

Definition: We call a system of migration rules *ping-pong-free* if for every process p which is migrated from host A under current load situation $load_A$ to host B under current load situation $load_B$ one of the following conditions holds:

(1) p is no migration candidate on host B under load $load_B$.

(2) p is a migration candidate on host B under load $load_B$, but no better destination host is available, i.e., p remains on B.

Theorem: The migration rules of Dynamic-SED are ping-pong-free.

Proof: If the main location rule was applied, process p will be no migration candidate and (1) holds.

Otherwise, the delay of the new destination host i_{dest} is higher than the delay class of process p and it becomes a migration candidate. In this case, i_{dest} was chosen from P due to rule 5.6, 5.7, or 5.8. Hence, there is no interactive user on i_{dest} and a new destination host $dest$ is determined due to 5.8:

$$P' = \{k \mid k \in P \wedge memory'_{i_{dest}} \leq memory_k \wedge d'_{i_{dest}} - d_k > \alpha_{i_{dest}}\}. \quad (5.9)$$

We have to show that no better destination host is available, i.e., $i_{dest} = dest$ which is true if $P' \subset P$.

Let i_{orig} be the origin host from p.

Interactive users on i_{orig}:

Since P contains only hosts which have no interactive users, p will never be migrated back to i_{orig}.

When i_{dest} was chosen due to 5.6, there is $memory_{i_{orig}} \leq memory_{i_{dest}}$. This means that the memory and the delay requirement in 5.9 shrinks the set of possible machines, and there is $P' \subset P$.

When i_{dest} was chosen due to 5.7, it is the fastest host regardless of the memory. Hence, again there is $P' \subset P$.

No interactive users on i_{orig}:

Then, i_{dest} was chosen due to 5.8. There is $memory'_{i_{orig}} \leq memory'_{i_{dest}}$ and $d'_{i_{orig}} - d_{i_{dest}} > \alpha_{i_{orig}}$.

Hence

$$d'_{i_{orig}} - d_k > \alpha_{i_{orig}} + d_{i_{dest}} - d_k > \alpha_{i_{orig}} + \alpha_{i_{dest}} > \alpha_{i_{dest}},$$

and again we have $P' \subset P$. $\Diamond$

After migration the system should run stable and the next migration should happen only after load changes which are not related to the migration itself.

A stronger property than ping-pong-free is migration-stable:

Definition: We call a system of migration rules *migration-stable* if after migration of a process p from host A under current load situation $load_A$ to host B under current load situation $load_B$, will cause no migration with origin B.

If a system of migration rules is migration-stable, no process on host B becomes a migration candidate, or, if a process on B may become a migration candidate, there will be no better destination host available.

Theorem: The migration rules of Dynamic-SED are not migration-stable.

Example: Let p be a process running on host A without interactive users. Dynamic-SED may migrate process p from A to B because of to high delay on A.

Hence, there are no interactive users on B.

Let q be a process on B with expected delay 1 and

$$memory\_usage(p) = memory\_usage(q) + \delta, \ \delta > 0. \quad (5.10)$$

Then q drops out of its delay class and gets a migration candidate. The destination host will be calculated due to 5.8:

$$d_{dest} = \min\{d_k \mid k \in P \wedge memory'_B \leq memory_k \wedge d'_B - d_k > \alpha_B\}, \qquad (5.11)$$

where $memory'_B$ is the free memory on B without q. Since

$$\begin{aligned} memory'_B &= memory_B + memory\_usage(q) - memory\_usage(p) \\ &= memory_B - \delta, \end{aligned}$$

the memory condition is less rigid and a host C which was not a suitable destination for the bigger process p may become now a suitable destination host for the smaller process q. Hence, q is migrated to C.

A situation which should be avoided is circular migrations of processes.

Definition: We say that a system of migration rules is *circular-free* if a migration of a process p from host A_0 to host A_1 will cause no migrations from A_i to host $A_{i+1},\ i = 1, ..., n-1$ with $A_n = A_0$.

Theorem: The migration rules of Dynamic-SED are circular-free.

Proof: Let us assume that Dynamic-SED is not circular-free. Then there exists a chain of hosts $A_i,\ i = 0, ..., n$ with $A_0 = A_n$ and A_{i+1} is destination of a process from A_i.

We will show the contradiction for three hosts A, B, C. The argumentation is analogous for longer chains. Let process p migrate from A to B, process q from B to C, and process r from C to A.

When A is allowed to be a migration destination, there are no interactive users on A. Hence, all destinations are determined due to rule 5.8.

Since p is migrated from A to B, there is

$$d'_A - d_B > \alpha_A > 0,$$

where d'_A is the delay on A without p.

C is destination for process q. Hence,

$$d''_B - d_C > \alpha_B > 0,$$

where d''_B is the delay on B without q. Since meanwhile p is running on B, there is $d''_B = d_B$ and we have

$$d_B - d_C > \alpha_B > 0.$$

From the migration of r it follows that

$$d'''_C - \tilde{d}_A > \alpha_C > 0,$$

where d'''_C is the delay on C without r but meanwhile with q running on C. Hence, $d'''_C = d_C$. The current delay $\tilde{d}_A$ of A is the delay of A without p which is equal to d'_A.

Therefore,

$$d_C - d'_A > \alpha_C > 0.$$

This gives a strong monotone decreasing chain of delays:

$$d'_A > d_B > d_C > d'_A,$$

which is a contradiction. ◊

It follows that processes will never be swapped between two hosts, i.e., a migration from A to B will never cause a migration of another process from B to A.

5.5 A Trace-Driven Simulation

We present the results of a trace-driven simulation to valuate the performance of the Dynamic-SED algorithm, i.e., SED with additionally reservation and migration rules as introduced in section 5.2 and 5.3. A detailed description can be found in [49].

The interesting questions are:

- Can the delay of a parallel application be guaranteed within a tolerance interval during execution?
- How many migrations per hour occur? This gives an approximation of the migration costs and whether the migration overhead is tolerable.

Dynamic-SED has to be tested now under changing load situations and in the presence of interactive users. Therefore, we need informations about load caused by interactive users who work concurrently to the parallel applications on the machines.

In this case, a trace-driven simulation is recommended where the input parameters for current load, free memory, and number of interactive users is received from monitoring a real workstation cluster.

During the simulation the current delay d_j is calculated from the trace data and used within the Dynamic-SED scheduling.

5.5.1 Model of the Test System

The test model is based upon the following assumptions [49]:

- The processes of a parallel application are computation-intensive.

  In this case a parallel process will appear most of the time in the run queue. The current delay d_j of machine j is then calculated as

$$d_j \quad = \quad d_j^{real} + \alpha_j \cdot n_j, \tag{5.12}$$

where d_j^{real} is the real monitored delay of machine j and n_j is the number of processes which are mapped onto this machine by the global scheduler during the simulation.

- The memory resources of processes which are mapped by the global scheduler during the simulation are 0.

  The reason for this assumption is the inaccurate load statistics of UNIX systems. The common interface for load informations on UNIX platforms is the `vmstat` command (see section 3). The available memory statistic gives only a hint about the free memory and no correct informations. This is due to caching strategies which are in common use in current file systems. Pages which are belonging to the cache are subtracted from the free memory, while these pages are available.

  Therefore, it would be misleading to calculate with memory resource demands.

- The action of the user in the presence of parallel jobs is 'ignore it'.

  The sampled trace data give the load situation in the test environment **without** parallel applications. Hence, the presence of interactive users is also monitored **without** parallel applications and the action of the user in the presence of parallel applications cannot be simulated.

  Therefore, we assume that the user keeps on working as long as he did due to the trace data[2].

5.5.2 Test System and Parameters

To test Dynamic-SED, we need informations about the load situation, free memory and the number of interactive users over a period of time for several machines. The following simulation uses trace data which were gathered on 28 Sun workstation in the Institute of Operating Systems and Computer Networks at the Technical University of Braunschweig. The machines and their characteristics are listed in table 5.1. The column labeled i_{max} gives the maximal number of interactive users monitored during the sampling of the data.

The parameters of Dynamic-SED are chosen as follows for the given test environment [49]:

Reservation Parameters

$\boldsymbol{mem_{min} = 500}$ **kBytes:** This bound was delivered from practical experiences with SunOS.

$\boldsymbol{mem_{reserv} = 1}$ **MByte:** If mem_{reserv} is chosen to high, it means that the machine may get unavailable, while the interactive users may not make use of the resources. On the other hand, programs used by interactive users like `X`, `emacs`, or `netscape` tend to use several MBytes.

[2]On the other hand, it is a common observation that user tend to flee from a machine when they recognize that the load on the machine gets higher.

name	type	α	i_{max}	memory	disk
achill	IPC	$3,36$	2	20 MB	yes
amalthea	ELC	$2,72$	3	16 MB	no
ares	SS 5	1	8	64 MB	yes
athena	SLC	4	1	8 MB	no
atlas	SLC	4	1	16 MB	no
bacchus	ELC	$2,72$	3	16 MB	no
bal	SLC	4	1	16 MB	no
bayes	SS 2	2	3	32 MB	yes
eos	IPC	$3,36$	2	24 MB	yes
hektor	IPC	$3,36$	2	24 MB	yes
helena	SLC	4	1	16 MB	no
helios	SS 2	2	4	40 MB	yes
io	SLC	4	1	8 MB	no
isis	SLC	4	1	16 MB	no
janus	ELC	$2,72$	3	16 MB	no
kastor	SS 20	1	10	96 MB	yes
logic	IPC	3.36	2	36 MB	yes
moloch	SLC	4	1	16 MB	no
nemesis	IPC	$3,36$	2	12 MB	yes
neptun	SLC	4	1	16 MB	no
neuro	SLC	4	1	16 MB	no
pandora	SLC	4	1	8 MB	no
possi	SLC	4	1	16 MB	no
ra	SS 2	2	1	48 MB	yes
ran	SLC	4	1	16 MB	no
sol	SS 10	1	5	64 MB	yes
thor	SLC	4	1	16 MB	no
venus	SLC	4	1	16 MB	no

Table 5.1: Test Environment [49].

load_reserv = 0.5: Since interactive processes need only less computing power, $load\_reserv$ should be less than 1. Due to the local UNIX scheduling and their short runtime, they will be scheduled with an higher priority than the long running parallel application.

Migration Parameters

$\delta T = 3$ min: The scheduler will check every three minutes, whether there is any migration candidate.

$z^2_{max} = 4$: When the process is a migration candidate over a time period of at least 9 minutes and an user is active, the process will be selected for migration.

$\boldsymbol{z^1_{max} = 6}$**:** If no user is active, but the process is a migration candidate over a time period of 15 minutes, it will be selected for migration.

$\boldsymbol{d^{max} = 12}$**:** This bound is to avoid overloading of a single host.

Simulation Parameters

One important workload parameter is the size of the parallel application. This depend on the specified $minsize$ and $maxsize$ and on the currently available nodes. If the application is started during night time, when the system is low loaded, the configurational size of the application will be higher. Therefore, two different experiments were simulated. In the **night experiment**, the parallel application was started at 0:00 h in the night and monitored over 4 days. In the **day experiment**, the parallel application was started at 4:00 pm in the afternoon and monitored over 4 days.

Both experiments were simulated for three parallel applications which differ in their sizes:

1. The first application A_{greedy} has

$$\begin{aligned} minsize &= 1, \\ maxsize &= a_{maxclass}. \end{aligned}$$

   Here, $maxsize$ is equal to the maximum number of available virtual nodes.

2. The second application A_{70} has

$$minsize = maxsize = 0.7 \cdot a_{maxclass}.$$

   Since there is $minsize = maxsize$, the application will allocate exactly 70 % of the available resources.

3. The third application A_{50} has

$$minsize = maxsize = 0.5 \cdot a_{maxclass}.$$

   A_{50} will allocate exactly 50 % of the available resources.

5.5.3 Results of the Night Experiment

At the start time of the application, the availability vector is $(3, 6, 10, 26)$. This leads to the following configurational sizes:

1. A_{greedy} has $minsize = 1$ and $maxsize = 36$. Hence, the configurational size is 26 with expected delay time $EDT = \frac{4}{26} \approx 0.15$.
2. A_{70} has $minsize = maxsize = 18$. Hence, the configurational size is 18 with expected delay time $EDT = \frac{4}{18} \approx 0.22$.
3. A_{50} has $minsize = maxsize = 13$. Hence, the configurational size is 13 with expected delay time $EDT = \frac{4}{13} \approx 0.31$.

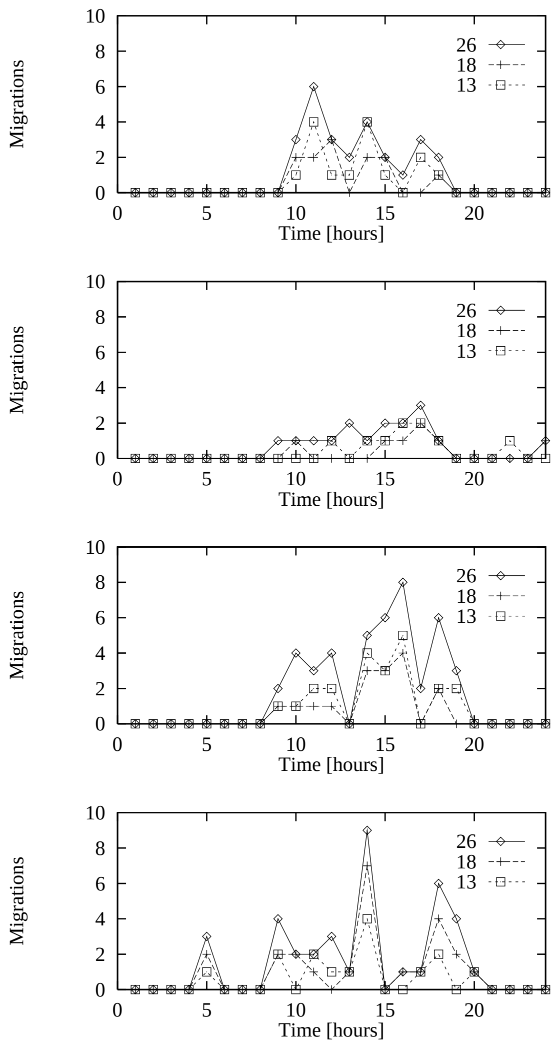


Figure 5.1: Number of Migrations over 4 days [49].

Migration Overhead

Figure 5.1 shows the number of migrations per hour for all 4 days. The curves for the different applications are labeled by the corresponding job size.

As expected, the number of migrations correlates to the number of interactive users (see figure 5.2). Migrations occur from about 10 h a.m. up to 7 h p.m. which corresponds to the working time of the staff and the students at the institute.

The observed maximum number of migrations per hour is 9 for A_{greedy}, 7 for A_{70}, and 5 for A_{50}. The mean number of migrations per hour over all 4 days is 1.27 for A_{greedy}, 0.61 for A_{70}, and 0.63 for A_{50}. This means that twice as much processes of the big application are migrated compared with the smaller applications. The number of migrations for A_{70} and A_{50} are almost the same.

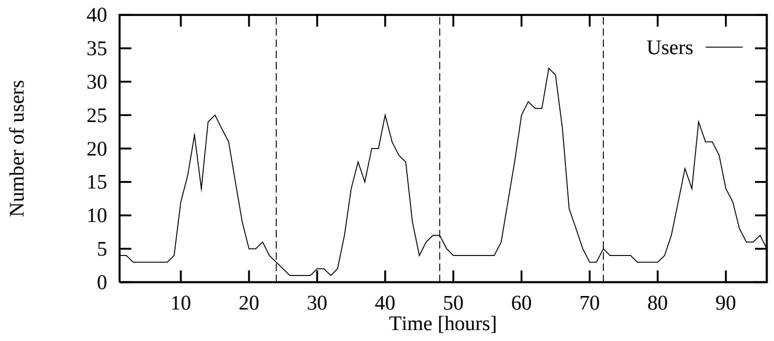


Figure 5.2: Number of interactive users over 4 days [49].

A_{greedy} allocates all available nodes including the slowest machines. If a user logs on one of these machines, migration will necessarily occur due to our migration rules. The smaller applications A_{50} and A_{70} do not occupy all machines. Hence, not every new interactive user will cause a migration.

The number of migrations is further correlated with the load situation on the fast machines [49]. If the load on the fast machines increases, processes will be migrated to other probably slower nodes. If the system load decreases again, the processes will migrate back to the faster nodes. This is visible in the migration peaks at about 6 h p.m..

Slowdown of the Applications

Figure 5.3 shows the current delay classes of the applications over the observed 4 days. The current delay class of the application was defined to be the maximum delay which one of the processes of the application currently has.

While all 3 applications were started within the same delay class, the current delay class of A_{greedy} is almost significantly higher. This shows that the application is much more often slow downed.

Since A_{greedy} allocates all machines, there is no possible destination host for migration when the interactive users start working.

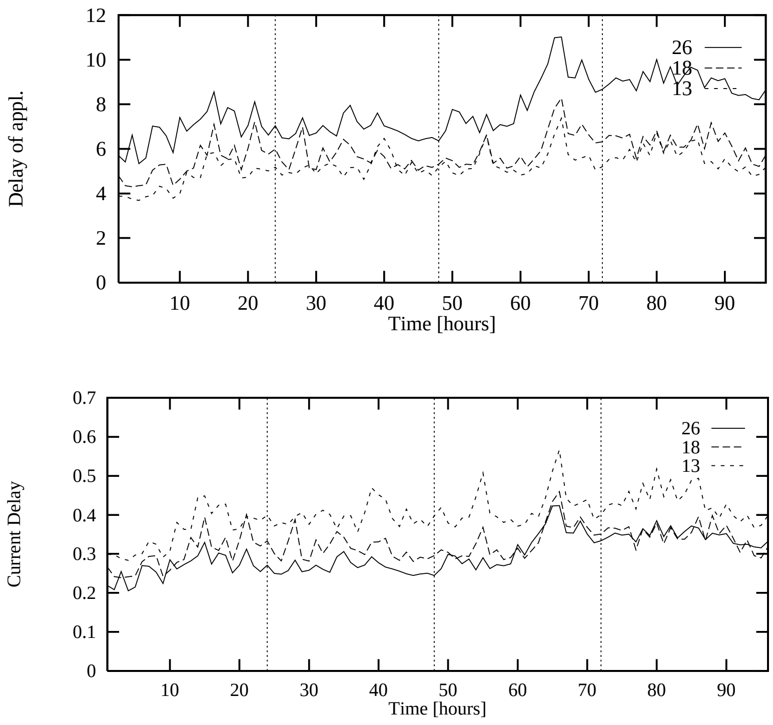


Figure 5.3: The current delay class of the applications over 4 days (top) and the expected delay time of the applications over 4 days (bottom) [49].

The delays of the smaller applications could be hold most of the time between 4 and 6. There are two reasons for this behaviour. Firstly, it is less probably that the smaller applications suffer under overloaded hosts. Secondly, if an overload situation occurs, it will be more probably that a destination host can be found.

In figure 5.3, the delay time of the applications is shown, i.e., the current delay divided by the size of the application. The observed delay time of A_{greedy} is almost the best. Since the smallest

appl.	size	mean delay class	mean delay time	EDT	slowdown
A_{greedy}	26	8.2	$\frac{8.2}{26} \approx 0.32$	0.15	113 %
A_{70}	18	6.04	$\frac{6.04}{18} \approx 0.36$	0.22	64 %
A_{50}	13	5.43	$\frac{5.43}{13} \approx 0.42$	0.31	35 %

Table 5.2: Different delays of the applications [49].

delay time offers the shortest runtime, the greedy approach pays off for the application.

Table 5.2 shows the mean delay class, the mean delay time, the originally expected delay time (ED), and the slowdown of the applications. While the mean delay time of A_{greedy} is 0.32, the mean delay time of A_{70} is 0.36. This means that A_{70} is almost as fast as the greedy application.

When we compare EDT and observed mean time delay, the slowdown of the greedy application is with 113 % the highest. This shows again that the application suffers significantly under high system load.

The slowdown of the smallest application is with 35 % a very good result which is achieved by migration. While A_{70} was migrated as often as A_{50}, the slowdown is with 64 % higher.

5.5.4 Results of the Day Experiment

The second measurement was started at 4 h p.m.. At this time the availability vector was $(0, 5, 9, 15)$ due to the loaded system. This leads to the following configurational sizes:

1. A_{greedy} has $minsize = 1$ and $maxsize = 25$. Hence, the configurational size is 15 with expected delay time $EDT = \frac{4}{15} \approx 0.27$.

2. A_{70} has $minsize = maxsize = 10$. Hence, the configurational size is 10 with expected delay time $EDT = \frac{4}{10} \approx 0.40$.

3. A_{50} has $minsize = maxsize = 7$. Hence, the configurational size is 7 with expected delay time $EDT = \frac{3}{7} \approx 0.43$.

A_{50} could be started in delay class 3 this time. The size of A_{greedy} is comparable with A_{70} of the night experiment. Again, the greedy application has the smallest expected delay.

The maximum number of migrations per hour is only 3 for A_{greedy} and 2 for the other ones. Since the applications are started when the system is already in its highest load situation, less resources are used and further, less migrations are necessary to guarantee a co-existence between parallel applications and interactive users.

Table 5.3 shows the observed delays of the applications. The observed delays are closer to the expected delays. Since the sizes of the applications are much smaller, it was easier to hold the delay during the execution.

appl.	size	mean delay class	mean delay time	EDT	slowdown
A_{greedy}	15	5.26	$\frac{5.26}{15} \approx 0.35$	0.27	30 %
A_{70}	10	4.23	$\frac{4.23}{10} \approx 0.42$	0.40	5 %
A_{50}	7	3.52	$\frac{3.52}{7} \approx 0.50$	0.43	16 %

Table 5.3: Different delays of the applications for the day experiment [49].

5.6 Summary

The experiments with Dynamic-SED have shown that migration is a useful tool for load balancing in a workstation cluster. We conclude that in the given test environment:

1. The migration rules add a tolerable overhead to the system, since the maximum number of migrations per hour is moderate.
2. The delay of greedy applications which allocate all in the idle system available nodes will increase significantly during execution.
3. When the application uses only about 70 % of the in the idle system available nodes, the runtime of the application will be approximately as good as the greedy application, but less migrations resp. migration wishes will occur.
4. When the applications are started at day time, the number of migrations are very low. This means that the reservation rules of Dynamic-SED have proven to guarantee a coexistence between parallel applications and interactive users.

Obviously, we have to distinct between *individual* and *social optimum*. The individual optimum was achieved by the greedy application in both experiments. But at the same time we observe the highest migration rate and the highest number of migration wishes without any available destination host.

When the greedy application allocates all possible nodes, there will be almost no possibility to migrate when interactive users start working. This means that for long running applications which will run over days, it has to be forbidden to allocate all available nodes.

For the observed environment, Gehrke has given a rule of thumb [49] which we state in the following way:

> **Rule of Thumb:** The social optimum for long running applications will be achieved when 30 % of the available nodes are reserved for interactive users.

The measure for "available nodes" is the number of nodes $a_{maxclass}$ in the slowest delay class in the idle system. The maximum number of nodes which may be allocated by parallel applications is

$$P_{max} = 70\% \cdot a_{maxclass}.$$

If the availability vector of the idle system is for example $(1, 2, 3, 10)$, P_{max} will be 7. Hence, the maximum size of an applications will be 7. If for example an application is started on the fastest host, the availability vector will be $(0, 0, 0, 6)$. Let Q denote the number of already allocated nodes. Then $Q = 4$ and a second application can be started with size at most $P_{max} - Q = 3$.

The trace-driven simulation of Dynamic-SED in a workstation cluster was done for only one application at a time. Additionally, it it interesting how Dynamic-SED behaves when several parallel applications are running. Therefore, we have formulated different migration properties. While the migration rules of Dynamic-SED are ping-pong-free and circular-free, they are not migration stable.

5.7 Bibliography

A survey of migration strategies and migration systems is given in [115].

Systems which support migration of parallel applications are ‹PBEAM [116, 117], MPVM [19], CoCheck [146]. While MPVM and CoCheck support only PVM applications, ‹PBEAM is not restricted to any programming model.

Migration strategies for load balancing are studied in [31, 81, 58, 164]. These studies investigate migration policies in homogeneous systems with no special care of parallel applications and interactive users.

In [59, 60], a loadbalancing algorithm based upon the gradient model is proposed. The algorithm considers communication costs in its decisions, but no memory demands. Further, it does not take care of interactive users. Simulation results are presented for a 2-dimensional mesh and Hypercubes.

Chapter 6

Conclusions

First, we have shown that the behavior of the presented scheduling algorithms depends much on the characteristics of the workload and that simple scheduling disciplines like LS or SNPF result in high performance benefits compared to FIFO.

Under medium and high system load, the performance can be improved, if the application uses the variable-size-model. While a wide range of applications such as CFD applications follow this model, there are less resource management systems available which support this opportunity.

One of the main contributions of this work is to investigate whether migration is an useful tool for scheduling and load balancing in workstation clusters. Both, the presented time-sharing discipline LST and the SED mapping policies make use of migration.

LST makes use of migration, if a mapping conflict occurs. The simulation results show that mapping conflicts are seldom under LST (less than 2 % of the processes have been migrated) and the migration overhead is neglectable.

The SED mapping discipline migrates processes when faster machines become available. The Dynamic-SED strategy adds further migration rules to support dynamic load balancing.

The simulations have shown that these are useful rules which lead to an improved performance. The overhead which is caused by migration is expected to be tolerable, since the observed numbers of migrations are very low.

Mapping state diagrams have shown to be an useful description tool for investigating the behaviour of the algorithms and to analyse the simulation results.

The trace-driven simulation of Dynamic-SED in a workstation cluster was done for only one application at a time. Additionally, it it interesting how Dynamic-SED behaves when several parallel applications are running. Therefore, we have formulated migration properties. While the migration rules of Dynamic-SED are ping-pong-free and circular-free, they are not migration stable.

References

[1] S.B. Akers and B. Krishnamurthy. A group theoretic model for symmetric interconnection networks. In *Proceedings of the Intern. Conference Parallel Processing*, pages 216–223, 1986.

[2] S. G. Akl and K. Quiu. A novel routing scheme on the star and pancake network and its applications. *Parallel Computing*, 19:95–101, 1993.

[3] K. Al-Saqabi, S.W. Otto, and J. Walpole. Gang scheduling in heterogeneous distributed systems. Technical report, Departement of Computer Science and Engineering, Oregon Graduate Institute of Science and Technology, 1994.

[4] M. Alef. Concepts for efficient multigrid implementation on suprenum-like architectures. *Parallel Computing*, 17:1–16, 1991.

[5] G. Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings*, volume 30, pages 483–485, Atlantic City NJ, 1967.

[6] H. Angstl. Design und Implementierung einer Scheduling-Komponente für das Codine Batch-Queuing-Systen. Master's thesis, Fachhochschule Regensburg, 1995.

[7] A. Barak, S. Guday, and R. G. Wheeler. *The MOSIX Distributed Operating System*. Springer–Verlag, 1993.

[8] J.M. Barton and N. Bitar. A scalable multi-discipline, multi-processor scheduling framework for IRIX. In Feitelson and Rudolph [40], pages 45–69.

[9] D. L. Black. Scheduling Support for Concurrency and Parallelism in the Mach Operating System. *IEEE Computer*, 23(5):35–43, 1990.

[10] J. E. Boillat. Load balancing and Poisson equation in a graph. *Concurrency: Practice and Experience*, 2(4):289–311, 1990.

[11] S. H. Bokhari. On the mapping problem. *IEEE Transactions on Computing*, C-30(3):207–214, March 1981.

[12] T. Bönniger, R. Esser, and D. Krekel. CM-5E, KSR2, Paragon XP/S: A comparative description of massively parallel computers. *Journal of Parallel Computing*, 21(2):199–232, 1995.

[13] Jim Browne, Jack Dongarra, Alan H. Karp, Ken Kennedy, and Dave Kuck. Gordon Bell Prize 1988. *IEEE Software*, 6(3):78–85, May 1989.

[14] H. Burkhart, C.F. Korn, S. Gutzwiller, P. Ohnacker, and S. Waser. BACS: Basel Algorithm Classification Scheme. Technical Report 93-3, University of Basel, March 1993.

[15] R. Butler and E. Lusk. Monitors, Messages and Clusters: The P4 Parallel Programming System. *Parallel Computing*, 20:547–564, 1994.

[16] L. Cabrera. The influence of workload on load balancing strategies. In *Proceedings of the 1986 USENIX Summer Technical Conference*, pages 446–458, Atlanta, 1986.

[17] C. H. Cap and V. Strumpen. Efficient parallel computing in distributed workstation environments. *Parallel Computing*, 19(11):1221–1234, 1993.

[18] N. Carriero and D. Gelernter. Linda in context. *Communication of the ACM*, 32(4):444–458, 1989.

[19] Jeremy Casas, Dan L. Clark, Ravi Konuru, Steve W. Otto, Robert M. Prouty, and Jonathan Walpole. MPVM: A migration transparent version of PVM. *Computing Systems*, 8(2):171–216, 1995.

[20] G. I. Chen and T. H. Lai. Scheduling jobs on partitionable hypercubes. *Journal of Parallel and Distributed Computing*, 12:74–78, 1991.

[21] S. Chiang, R. Mansharamani, and M. Vernon. Use of application characteristics and limited preemption for run–to–completion parallel processor scheduling policies. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurements and Modelling of Computer Systems*, pages 33–44, 1994.

[22] P.-J. Chuang and N.-F. Tzeng. An efficient submesh allocation strategy for mesh computer systems. In *Proc. of the 1991 International Conference on Distributed Computer Systems*, May 1991.

[23] F.G. Coffman, M.R. Garey, and D.S. Johnson. Approximation algorithms for bin-packing - an updated survey. In G. Ausiello and Serafini [48], pages 49–106.

[24] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *Journal of Parallel and Distributed Computing*, 7:279–301, 1989.

[25] K. Day and A. Tripathi. Arrangement graphs: A class of generalized star graphs. *Information Processing Letters*, 42(5):235–241, 1992.

[26] M. V. Devarakonda and R. K. Iyer. Predictability of process resource usage: A measurement-based study on UNIX. *IEEE Transactions on Software Engineering*, 15(12):1579–1586, December 1989.

[27] Jack Dongarra, Alan H. Karp, and Ken Kennedy. Gordon Bell Awards 1987. *IEEE Software*, 5(3):108–112, May 1988.

[28] Jack Dongarra, Alan H. Karp, Ken Kennedy, and Dave Kuck. Gordon Bell Prize 1989. *IEEE Software*, 7(3):100–110, May 1990.

[29] Jack Dongarra, Alan H. Karp, Ken Miura, and Host Simon. Gordon Bell Prize 1990. *IEEE Software*, 8(3):92–102, May 1991.

[30] D.W. Duke, T.P. Green, and J.L. Pasko. Research towards a heterogeneous networked computing cluster: The Distributed Queueing Cluster, Version 3.0. Technical report, Florida State University, 1994. ftp.scri.fsu.edu:/pub/DQS.

[31] D. L. Eager, E. D. Lazowska, and J. Zahorjan. The limited performance benefits of migrating active processes for load sharing. *Performance Evaluation*, 6(1):63–72, 1988.

[32] R. Esser and R. Knecht. Intel Paragon XP/S - Architecture and Software Environment. In *Supercomputer '93*, Mannheim, 1993.

[33] G.E. Fagg, K.S. London, and J.J. Dongarra. Taskers and general resource Managers: PVM Supporting DCE Process Managemnet. In A. Bode, J. Dongarra, T. Ludwig, and V. Sunderam, editors, *Parallel Virtual Machines - EuroPVM'96*, pages 180–187. Springer, October 1996.

[34] Feitelson and Nitzberg. Job Characteristics of a Production parallel scientific workload on the NASA Ames iPSC/860. In *Lecture Notes in Computer Science 949*, pages 337–360. Springer, 1995.

[35] D. G. Feitelson. A Survey of Scheduling in Multiprogrammed Parallel Systems. Technical Report RC 19790 (87657), IBM T. J. Watson Research Center, February 1995. http://www.cs.huji.ac.il/˜feit.

[36] D. G. Feitelson and L. Rudolph. Distributed hierarchical control for parallel processing. *IEEE Computer*, 23(5):65–77, 1990.

[37] D. G. Feitelson and L. Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing*, 16:306–318, 1992.

[38] D. G. Feitelson and L. Rudolph, editors. *Job Scheduling Strategies for Parallel Processing*. Proceedings of the IPPS '95 Workshop, Lecture Notes in Computer Science 949. Springer, Santa Barbara, April 1995.

[39] D. G. Feitelson and L. Rudolph. Parallel job scheduling: Issues and approaches. In *Lecture Notes in Computer Science 949*, pages 1–18. Springer, 1995.

[40] D. G. Feitelson and L. Rudolph, editors. *Job Scheduling Strategies for Parallel Processing*, Proceedings of the IPPS ’96 Workshop, Lecture Notes in Computer Science 1162, Santa Barbara, April 1996. Springer.

[41] D. G. Feitelson and L. Rudolph, editors. *Job Scheduling Strategies for Parallel Processing*, Proceedings of the IPPS ’97 Workshop, Lecture Notes in Computer Science 1291, Geneva, April 1997.

[42] D. G. Feitelson and L. Rudolph, editors. *Job Scheduling Strategies for Parallel Processing*, Proceedings of the IPPS ’98 Workshop, Lecture Notes in Computer Science 1459. Springer, April 1998.

[43] D.G. Feitelson. Packing Schemes for Gang Scheduling. In Feitelson and Rudolph [40], pages 89–110.

[44] D.G. Feitelson, L. Rudolph, U. Schwiegelshohn, K.G. Sevcik, and P. Wong. Theory and Practice in Parallel Job Scheduling. In D. G. Feitelson and L. Rudolph, editors, *Proceedings of the IPPS ’97 Workshop Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Computer Science*, pages 1–25, Geneva, April 1997. Springer.

[45] D. Ferrari and S. Zhou. An empirical investigation of load indices for load balancing applications. In *Performance ’87*, pages 515–528. Elsevier Science Publishers, 1988.

[46] H.P. Flatt and K. Kennedy. Performance of Parallel Computers. *Parallel Computing*, 12(1):1–20, 1989.

[47] Ian Foster, J. Geisler, W. Nickless, W. Smith, and S. Tuecke. Software Infrastructure for the I-WAY High-Performance Distributed Computing Experiment. In *Proceeding 5th IEEE Symp. on High Performance Distributed Computing*. IEEE Computer Society Press, 1996. http://www.mcs.anl.gov/globus.

[48] M. Lucertini G. Ausiello and P. Serafini, editors. *Algorithm Design for Computer Systems Design*. Springer, 1984.

[49] Marc Gehrke. Ressourcemanagement für PVM–Anwendungen. Master’s thesis, Institut für Betriebssysteme und Rechnerverbund, TU Braunschweig, 1996.

[50] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. PVM3 Users Guide and Reference Manual. Technical Report TM-12187, Oak Ridge National Laboratory, September 1994.

[51] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine – A Users’ Guide and Tutorial for Networked Parallel Computing*. The MIT Press, Cambridge, Massachusetts, 1994.

[52] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[53] D. Gelernter and D. Kaminsky. Supercomputing out of recycled garbage: Preliminary Experiences with Piranha. In *Proc. of the ACM International Conference on Supercomputing*, July 1992.

[54] D. Ghosal, G. Serazzi, and S. K. Tripathi. The processor working set and its use in scheduling multiprocessor systems. *IEEE Transactions on Software Engineering*, 17(5):443–453, 1991.

[55] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM J. Appl. Math.*, 17(2):416–429, 1969.

[56] A. Gupta, A. Tucker, and S. Urushibara. The impact of operating system scheduling policies and synchronization methods on the performance of parallel applications. In *Proceedings of the 1991 ACM SIGMETRICS Conference on Measurements and Modeling of Computer Systems*, pages 120–132, 1991.

[57] J. Gustafson. Reevaluating Amdahl's Law. *Communications of the ACM*, 31(5):532–533, 1988.

[58] Mor Harchol-Balter and Allen B. Downey. Exploiting lifetime distributions for dynamic load balancing. In *Proceedings of 15th ACM Symposium on Operating Systems Principles Poster Session (SIGOPS'95)*, page 236, December 1995.

[59] H.-U. Heiß. Dynamic Decentralized Load Balancing: The Particles Approach. In *Proceedings of the International Symposium on Computer and Information Sciences VIII*, Istanbul, 1993.

[60] H.-U. Heiß. *Prozessorzuteilung in Parallelrechnern*. BI Wissenschaftsverlag, 1994.

[61] R.L. Hendersen. Job Scheduling Under the Portable Batch System. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 279–294, Santa Barbara, April 1995. Proceedings of the IPPS '95 Workshop, Lecture Notes in Computer Science 949, Springer. LNCS 949.

[62] D. Henrich. *Lastverteilung für Branch-and-bound auf eng-gekoppelten Parallelrechnern*. Dissertation, Universität Karlsruhe, 1994.

[63] D. Henrich. Local load balancing according to a simple liquid model. In *PARS Workshop*, PARS Mitteilungen, Stuttgart, 1995.

[64] A. Hori, H. Tezuka, Y. Ishikawa, N. Soda, H. Konaka, and M. Maeda. Implementation of Gang-Scheduling on Workstation Cluster. In Feitelson and Rudolph [40], pages 126–139.

[65] Christopher Wade Humphres. A load balancing extension for the PVM software system. Master's thesis, University of Alabama, Alabama, 1995.

[66] Kai Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmibility*. McGraw-Hill, 1993.

[67] IBM Corporation. *Using and Administering LoadLeveler – Release 3.0*, 4 edition, August 1996. Document Number SC23-3989-00.

[68] IEEE, editor. *Proceedings of the 6th International Conference on Distributed Computer Systems*, Washington, 1986.

[69] IEEE, editor. *Proceedings of the 8th International Conference on Distributed Computer Systems*, Washington, D.C., 1988.

[70] C. Jacqmot, E. Milgrom, W. Joossen, and Y. Berbers. Unix and Load-Balancing: A Survey. In *Proceedings of the EUUG '89*, pages 1–15, April 1989.

[71] J. Ju, G. Xu, and K. Yang. An Intelligent Dynamic Load Balancer for Workstation Clusters. *Operating System Review*, 29(1):7–16, 1995.

[72] Jiubin Ju and Yong Wang. Scheduling PVM tasks. *Operating System Review*, 30(3):22–31, July 1996.

[73] Alan H. Karp, Michael Heath, and Al Geist. 1995 Gordon Bell Prize Winners. *IEEE Computer*, 29(1):79–85, January 1996.

[74] Alan H. Karp, Michael Heath, Don Heller, and Horst Simon. 1994 Gordon Bell Prize Winners. *IEEE Computer*, 28(1):68–74, January 1995.

[75] Alan H. Karp, Don Heller, and Host Simon. 1993 Gordon Bell Prize Winners. *IEEE Computer*, 27(1):69–75, January 1994.

[76] Alan H. Karp, Ken Miura, and Host Simon. Gordon Bell Prize 1992. *IEEE Computer*, 26(1):77–82, January 1993.

[77] G. A. Kohring. Dynamic load balancing for parallelized particle simulations on MIMD computers. *Parallel Computing*, 21:683–693, 1995.

[78] R.B. Konura, J.E. Moreira, and V.K. Naik. Application-Assisted Dynamic Scheduling on Large-Scale Multi-Computer Systems. In Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *Parallel Processing, Volume II of the Proceedings of the Second International Euro-Par Conference (Euro-Par'96)*, volume 1124 of *Lecture Notes in Computer Science*, pages 621–630. ENS Lyon, Springer, August 1996.

[79] P. Krueger, T.-H. Lai, and V. A. Radiya. Processor allocation vs. job scheduling on hypercube computers. In *Proc. of the 1991 International Conference on Distributed Computer Systems*, pages 394–401, May 1991.

[80] P. Krueger and M. Livny. The diverse objectives of distributed scheduling policies. In *Proceedings of the 7th International Conference on Distributed Computer Systems*, pages 242–249, September 1987. Berlin, West Germany.

[81] P. Krueger and M. Livny. A comparison of preemptive and non-preemptive load distributing. In *Proceedings of the eighth International Conference on Distributed Computer Systems*, pages 123–130, 1988.

[82] N. Kuck, M. Middendorf, and H. Schmeck. Generic branch-and-bound on a network of transputers. In R. Grebe et al., editors, *Transputer Applications and Systems*, pages 521–535. IOS Press, 1993.

[83] S. Kühne. Gruppen-Scheduling. Studienarbeit, 1993. Institut für Betriebssysteme und Rechnerverbund, TU Braunschweig.

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

[85] S. Lakshmivarahan, J.-S. Jwo, and S.K. Dhall. Symmetry in interconnection networks based on Cayley graphs of permutation groups: A survey. *Parallel Computing*, 19:361–407, 1993.

[86] H. Langendörfer, editor. *Praxisorientierte Parallelverarbeitung – Beiträge zum 3. Workshop über Wissenschaftliches Rechnen*, Braunschweig, October 1994. Hanser.

[87] H. Langendörfer and B. Schnor. *Verteilte Systeme*. Hanser, München, 1994.

[88] T. Lauer. *Adaptive dynamische Lastbalancierung*. PhD thesis, Max Planck Institut für Informatik Saarbrücken, 1995.

[89] W. E. Leland and T. J. Ott. Load-balancing heuristics and process behavior. In *Proceedings of Performance '86 and ACM Sigmetrics 1986*, pages 54–69, 1986.

[90] S. T. Leutenegger and M. K. Vernon. The performance of multiprogrammed multiprocessor scheduling policies. In *Proc. of the 1990 ACM SIGMETRICS Conference on Measurements & Modeling of Computer Systems*, pages 226–236, May 1990.

[91] K. Li and K. H. Cheng. Job scheduling in partitionable mesh connected systems. In *Proceedings of the 1989 International Conference on Parallel Processing*, pages 65–72, 1989.

[92] K. Li and K. H. Cheng. Job Scheduling in a Partitionable Mesh using a two-dimensional Buddy System Partitioning Scheme. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):413–422, October 1991.

[93] D.A. Lifka. The ANL/IBM SP Scheduling System. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 295–303, Santa Barbara, April 1995. Proceedings of the IPPS ’95 Workshop, Lecture Notes in Computer Science 949, Springer. LNCS 949.

[94] F. C. Lin and R. M. Keller. The gradient model load balancing method. *IEEE Transactions on Software Engineering*, 13(1):32–38, 1987.

[95] M. Litzkow and M. Solomon. Supporting checkpointing and process migration outside the UNIX kernel. In *USENIX Conference Proceedings*, pages 283–290, San Francisco, CA, Winter 1992. USENIX.

[96] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor - A hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Systems*, pages 104–111, Los Alamitos, Calif., 1988. IEEE CS Press.

[97] Martin Loitz. Lmake – Entwurf und Implementierung eines parallelen Make-Programms. Master’s thesis, Institut für Betriebssysteme und Rechnerverbund, TU Braunschweig, 1993.

[98] D. Long, J. Caroll, and C. Park. A Study of the Reliability of Internet Sites. In *Proceedings of the 10th Symposium on Reliable Distributed System*, pages 177–186, 1991.

[99] S. Majumdar, D.L. Eager, and R.B. Bunt. Scheduling in Multiprogrammed Parallel Systems. In *Proceedings of the 1988 ACM SIGMETRICS Conference on Measurements and Modelling of Computer Systems*, pages 104–113, May 1988.

[100] Herrmann G. Matthies and Josef Schüle, editors. *Paralleles und Verteiltes Rechnen – Beiträge zum 4. Workshop über Wissenschaftliches Rechnen*, Aachen, October 1996. TU Braunschweig, Shaker.

[101] C. McCann, R. Vaswani, and J. Zahorjan. A dynamic processor allocation policy for multiprogrammed shared–memory multiprocessors. *ACM Transactions on Computer Systems*, 11(2):146–178, May 1993.

[102] S. Meynen and P. Wriggers. Parallele Algorithmen zur Lösung von nichtlinearen Problemen in der Festkörpermechanik. In Matthies and Schüle [100], pages 129–138.

[103] D. Min and M. W. Mutka. Effects of job size irregularity on the dynamic resource scheduling of a 2-D mesh multicomputer. In *5th International PARLE Conference*, pages 476–487, June 1993.

[104] M. W. Mutka. Estimating capacity for sharing in a privately owned workstation environment. *IEEE Transactions on Software Engineering*, 18(4):319–328, April 1992.

[105] M. W. Mutka and M. Livny. Scheduling remote processing capacity in a workstation-processor bank network. In *Proceedings of the 7th International Conference on Distributed Computer Systems*, pages 2–9, 1987.

[106] V.K. Naik, S.K. Setia, and M.S. Squillante. Performance analysis of job scheduling policies in parallel supercomputer environments. In *Supercomputer'93*, 1993.

[107] E. Nemeth, G. Snyder, S. Seebass, and T. R. Hein. *UNIX System Administration Handbook, Sec. Edition*. Prentice Hall, Englewood Cliffs, 1995.

[108] L. Ni and K. Hwang. Optimal load balancing strategies for a multiple processor system. In *1981 International Conference on Parallel Processing*, pages 362–367, August 1981.

[109] L. Ni and K. Hwang. Optimal load balancing in a multiple processor system with many job classes. *IEEE Transactions on Software Engineering*, SE-11(5):491–496, 1985.

[110] L. M. Ni, C. Xu, and T. B. Gendreau. A distributed drafting algorithm for load balancing. *IEEE Transactions on Software Engineering*, 11(10):1153–1161, October 1985.

[111] Reinhard Oleyniczak. Simulation von Gruppenschedulingverfahren für heterogene Plattformen. Master's thesis, Institut für Betriebssysteme und Rechnerverbund, TU Braunschweig, 1997.

[112] J. K. Ousterhout. Scheduling techniques for concurrent systems. In *3rd International Conference on Distributed Computing Systems*, pages 22–30, Miami, 1982.

[113] E. W. Parsons and K. C. Sevcik. Processor scheduling for high–variability service time distributions. In *Lecture Notes in Computer Science 949*, pages 127–145. Springer, 1995.

[114] S. Petri. simuLan- Ein Werkzeug zur Simulation lokaler Netze. Master's thesis, Institut für Betriebssysteme und Rechnerverbund, TU Braunschweig, 1991.

[115] S. Petri. *Lastausgleich und Fehlertoleranz in Workstation-Clustern*. PhD thesis, Institut für Betriebssysteme und Rechnerverbund, TU Braunschweig, 1996. In Preparation.

[116] S. Petri and H. Langendörfer. Load Balancing and Fault Tolerance in Workstation Clusters – Migrating Groups of Communicating Processes. *Operating Systems Review*, 29(4):25–36, October 1995.

[117] S. Petri, B. Schnor, H. Langendörfer, and J. Steinborn. Consistent Global Checkpoints for Distributed Applications on Clusters of Unix Workstations. In Matthies and Schüle [100], pages 77–86.

[118] Stefan Petri, Bettina Schnor, Matthias Becker, Bernd Hinrichs, Titus Tscharntke, and Horst Langendörfer. Evaluation of Multicast Methods to Maintain a Global Name Space for Transparent Process Migration in Workstation Clusters. In M. Zitterbart, editor, *Kommunikation in Verteilten Systemen*, Informatik aktuell, pages 224–234, Braunschweig, February 1997. GI/ITG Fachtagung KIVS'97, Springer.

[119] R. Popescu-Zeletin, G. LeLann, and K.H. Kim, editors. *Proceedings of the 7th International Conference on Distributed Computer Systems*, Berlin, 1987. IEEE.

[120] J. Pruyne and M. Livny. Parallel Processing on Dynamic Resources with CARMI. In *Lecture Notes in Computer Science 949*, pages 259–279. Springer, 1995.

[121] J. Pruyne and M. Livny. Managing Checkpoints for Parallel Programs. In D.G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing, IPPS'96 Workshop)*, volume 1162 of *Lecture Notes in Computer Science*, pages 140–154. Springer, April 1996.

[122] S. Ranka, J.-C. Wang, and N. Yeh. Embedding meshes on the star graph. *Journal of Parallel and Distributed Computing*, 19(2):131–135, October 1993.

[123] E. Rosti, E. Smirni, G. Serazzi, and L.W. Dowdy. Analysis of non-work-conserving processor partitioning policies. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 165–181, Santa Barbara, April 1995. Proceedings of the IPPS '95 Workshop, Springer. LNCS 949.

[124] B. Rotzoll. Dynamischer Multiprogrammbetrieb von Parallelrechnern. In Wedekind, editor, *Verteilte Systeme*, pages 229–243. BI, 1994.

[125] Y. Rouskov and P.K. Srimani. Fault diameter of star graphs. *Information Processing Letters*, 48(5):243–252, December 1993.

[126] P. Sanders and Th. Worsch. Konvergente lokale Lastverteilungsverfahren und ihre Modellierung durch Zellularautomaten. In *PARS-Mitteilungen Nr. 14*, pages 285–291. GI FG 3.1.2, 1995.

[127] W. Saphir, L.A. Tanner, and B. Traversat. Job Management Requirements for NAS Parallel Systems and Clusters. In *Lecture Notes in Computer Science 949*, pages 319–336. Springer, 1995.

[128] S. Sattler. Leistungsanalyse von Algorithmen für Gruppen-Scheduling. Studienarbeit, 1995. Institut für Betriebssysteme und Rechnerverbund, TU Braunschweig.

[129] B. Schnor. Architectures and algorithms for group scheduling. In *Proceedings of the International Conference on Parallel Computing Technologies 1993*, pages 225–233, Obninsk, 1993.

[130] B. Schnor. Dynamic scheduling of parallel applications. In *Lecture Notes in Computer Science 964, Third International Conference on Parallel Computing Technologies*, pages 109–116, St. Petersburg, 1995.

[131] B. Schnor, H. Langendörfer, and S. Petri. Einsatz neuronaler Netze zur Lastbalancierung in Workstationclustern. In Langendörfer [86], pages 154–165.

[132] B. Schnor, S. Petri, and H. Langendörfer. A Detailed Comparison of Traditional and Neural Network Based Approaches for Load Metrics on Heterogeneous Platforms. In Matthies and Schüle [100], pages 93–106.

[133] B. Schnor, S. Petri, and H. Langendörfer. Load Management for Load Balancing on Heterogeneous Platforms: A Comparison of Traditional and Neural Network Based Approaches. In Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *Parallel Processing, Volume II of the Proceedings of the Second International Euro-Par Conference (Euro-Par'96)*, volume 1124 of *Lecture Notes in Computer Science*, pages 615–620. ENS Lyon, Springer, August 1996.

[134] B. Schnor, S. Petri, R. Oleyniczak, and H. Langendörfer. Scheduling of Parallel Applications on Heterogeneous Workstation Clusters. In Koukou Yetongnon and Salim Hariri, editors, *Proceedings of the ISCA 9th International Conference on Parallel and Distributed Computing Systems*, volume 1, pages 330–337, Dijon, September 1996. ISCA.

[135] J. Schüle and A. Brandes. Practical Aspects Using a Distributed Queuing System. In Clemens H. Cap, editor, *Workstations und ihre Anwendungen, Proceedings der Fachtagung SIWORK'96*, pages 91–102. vdf Hochschulverlag AG an der ETH Zürich, May 1996.

[136] S. K. Setia. Trace-driven analysis of migration-based gang scheduling policies for parallel computers. In H. Dietz, editor, *Proceedings of the 1997 International Conference on Parallel Processing*, pages 489–492, August 1997.

[137] S. K. Setia. Trace-driven analysis of migration-based gang scheduling policies for parallel computers. Technical Report TR97-03, Computer Science Departement, George Mason Universit, 1997.

[138] S. K. Setia, M. S. Squillante, and S. K. Tripathi. Processor scheduling on multiprogrammed, distributed memory parallel computers. *Performance Evaluation Review*, 21(1):158–170, June 1993.

[139] S. K. Setia and S. K. Tripathi. A comparative analysis of static processor partitioning policies for parallel computers. In *Proceedings of the International Workshop on Modeling and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 283–286, 1993.

[140] K. C. Sevcik. Application scheduling and processor allocation in multiprogrammed parallel processing systems. *Performance Evaluation*, 19:107–140, 1994.

[141] J.-P. Sheu, W.-H. Liaw, and T.-S. Chen. A broadcasting algorithm in star graph interconnection networks. *Information Processing Letters*, 48(5):237–242, December 1993.

[142] J. Skovira, W. Chan, H. Zhou, and D. Lifka. The EASY - LoadLeveler API Project. In Feitelson and Rudolph [40], pages 41–47.

[143] P. G. Sobalvarro and W. E. Weihl. Demand-based coscheduling of parallel jobs on multiprogrammed multiprocessors. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 106–126, Santa Barbara, April 1995. Proceedings of the IPPS ’95 Workshop, Springer. LNCS 949.

[144] M.S. Squillante. On the benefits and limitations of dynamic partitioning in parallel computer systems. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 219–238, Santa Barbara, April 1995. Proceedings of the IPPS ’95 Workshop, Springer. LNCS 949.

[145] Susanne Steiner. Priority based location policies applied to heterogeneous server speeds. Technical report, Institut für Betriebssysteme und Rechnerverbund, Universität Hildesheim, 1995.

[146] Georg Stellner. Resource Management and Checkpointing for PVM. In *Proceedings of the Second European PVM User Group Meeting*, Lyon, 1995.

[147] S. Stille. Lastbalancierung in verteilten Systemen. Master’s thesis, Institut für Betriebssysteme und Rechnerverbund, TU Braunschweig, 1993.

[148] Tony T.Y. Suen and Johnny S.K. Wong. Efficient Task Migration Algorithm for Distributed Systems. *IEEE Transactions on Parallel and Distributed Systems*, 3(4):488–499, July 1992.

[149] X.-H. Sun and L.M. Ni. Scalable problems and memory-bounded speedup. *Journal of Parallel and Distributed Computing*, 19(1):27–37, 1993.

[150] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, 1990.

[151] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 1992.

[152] Marvin M. Theimer and Keith A. Lantz. Finding Idle Machines in a Workstation-based Distributed System. In *Proceedings of the 8th International Conference on Distributed Computer Systems*, pages 112–122, San Jose, CA, June 1988.

[153] Thinking Machined Corp. *Connection Machine CM-5 Technical Summary*, November 1992.

[154] A. Trew and G. Wilson (Eds.). *Past, Present, Parallel: A Survey of Available Parallel Computing Systems*. Springer, 1991.

[155] U. Trottenberg and K. Solchenbach. Parallele Algorithmen und ihre Abbildung auf parallele Rechnerarchitekturen. *Informationstechnik*, 30(2):71–82, 1988.

[156] A. M. van Tilborg and L. D. Wittie. Wave scheduling - distributed allocation of task forces in network computers. In *Proceedings of the 2nd International Conference on Distributed Computing Systems*, pages 337–347, Paris, 1981.

[157] M. Wan, R. Moore, G. Kremenek, and K. Steube. A Batch Scheduler for the Intel Paragon with a non-contiguous Node Allocation Algorithm. In D.G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing, IPPS'96 Workshop)*, volume 1162 of *Lecture Notes in Computer Science*, pages 48–64. Springer, April 1996.

[158] Q. Wang and K. H. Cheng. List scheduling of parallel tasks. *Information Processing Letters*, 37:291–297, May 1991.

[159] A. Weinrib and S. Shenker. Greed is not enough: Adaptive load sharing in large heterogeneous systems. In *Proc. InfoCom*, pages 986–994, 1988.

[160] Jon B. Weissmann and Andrew S. Grimshaw. A Federated Model for Scheduling in Wide-Area Systems. In *Proceeding 5th IEEE Symp. on High Performance Distributed Computing*, pages 542–550. IEEE Computer Society Press, 1996.

[161] Gerhard Wilhelms. *Dynamische adaptive Lastverteilung für PVM mittels unscharfer Benutzerprofile - PVM+*. PhD thesis, Universität Augsburg, Oktober 1994.

[162] M. H. Willebeek-LeMair and A. P. Reeves. Strategies for dynamic load balancing on highly parallel computers. *Transactions on Parallel and Distributed Systems*, 4(9), 1993.

[163] W. Winston. Optimality of the shortest line discipline. *Journal of Applied Probability*, 14:181–189, 1977.

[164] W. Zhu and P. Socko. Migration Impact on Load Balancing - An Experience on Amoeba. In *Proc. of the Fifth International Symposium on High Performance Distributed Computing*, pages 531–540, August 1996.

[165] Y. Zhu. Efficient processor allocation strategies for mesh-connected parallel computers. *Journal of Parallel and Distributed Computing*, 16:328–337, 1992.

Index