THEORETICAL ASPECTS ON PERFORMANCE BOUNDS AND FAULT TOLERANCE IN PARALLEL COMPUTING

Kamilla Klonowska

Blekinge Institute of Technology
Doctoral Dissertation Series No. 2007:18

School of Engineering

Theoretical Aspects on Performance Bounds and Fault Tolerance in Parallel Computing

Kamilla Klonowska

Blekinge Institute of Technology Doctoral Dissertation Series
No 2007:18
ISSN 1653-2090
ISBN 978-91-7295-126-6

Theoretical Aspects on Performance Bounds and Fault Tolerance in Parallel Computing

Kamilla Klonowska

Department of Systems and Software Engineering
School of Engineering
Blekinge Institute of Technology
SWEDEN

Department of Systems and Software Engineering
School of Engineering
Publisher: Blekinge Institute of Technology
Printed by Printfabriken, Karlskrona, Sweden 2007
ISBN 978-91-7295-126-6

In Memory of my Father
Tadeusz Klonowski

This thesis has been submitted to the Faculty of Technology at Blekinge Institute of Technology in partial fulfilment of the requirements for the Degree of Doctor of Philosophy in Computer Systems Engineering.

Contact information:

Kamilla Klonowska
Department of Systems and Software Engineering
School of Engineering, Blekinge Institute of Technology
Box 520
SE-372 25 Ronneby
Sweden
E-mail: Kamilla.Klonowska@bth.se

Abstract

This thesis consists of two parts: performance bounds for scheduling algorithms for parallel programs in multiprocessor systems, and recovery schemes for fault tolerant distributed systems when one or more computers go down.

In the first part we deliver tight bounds on the ratio for the minimal completion time of a parallel program executed in a parallel system in two scenarios. Scenario one, the ratio for minimal completion time when processes can be reallocated compared to when they cannot be reallocated to other processors during their execution time. Scenario two, when a schedule is preemptive, the ratio for the minimal completion time when we use two different numbers of preemptions.

The second part discusses the problem of redistribution of the load among running computers in a parallel system. The goal is to find a redistribution scheme that maintains high performance even when one or more computers go down. Here we deliver four different redistribution algorithms.

In both parts we use theoretical techniques that lead to explicit worst-case programs and scenarios. The correctness is based on mathematical proofs.

Acknowledgments

This Ph.D. thesis may have never been realized without my supervisors, colleauges, friends and family members from here and apart. Thank you.

I especially thank my supervisors, Professor Lars Lundberg and Docent Håkan Lennerstad for their contribution in all the papers, comments, ideas, and encouragement during this work; Lars for his patience and Håkan for a lot of dialogues and discussions about life and mathematics.

I also thank the members of my research group: Charlie Svahnberg for answering on all my questions and his contributions in Golomb-series papers, Dawit Mengistu for a lot of discussions about complexity of life and scheduling, Simon Kågström for helping me understand Assembly and other strange things, Peter Tröger for deep discussions about fault, error and failure, and keeping my family busy during the last, most stressfull weekends, Mia Persson for discussions about NP-complexity, Göran Fries for being my room-neighbour throughout my Ph.D. studies, Bength Aspvall, Håkan Grahn and Magnus Broberg, an ex-member, for his contribution in the first paper.

I would also like to thank my colleagues at the department, especially Linda Ramstedt, Johanna Törnkvist, Lawrence Henesey, Jenny Lundberg, Nina Dzamashvili-Fogelström, Guohua Bai, Maddeleine Pettersson, Monica Nilsson, May-Louise Andersson, Petra Nilsson.

Finally, I want to express my gratitude to my family for their support and encouragement from afar during hard times. Special thanks to Jacek and Wiktor, my Mother, Ciocia Wera, Basia and Ciocia Lodzia, Ciocia Eleonora, Iwonka and Krzysztof for being with me during that time. And my friends: Radek and Gosia Szymanek, Ewa and Jonta Andersson, Jelena Zaicenoka, Magdalena Urbanska, Viola and Tomek Murawscy, Karina Stachowiak and Rafal Lacny.

List of Papers

Papers included in this thesis:

I "Comparing the Optimal Performance of Parallel Architectures"
Kamilla Klonowska, Lars Lundberg, Håkan Lennerstad, Magnus Broberg
The Computer Journal, Vol. 47, No. 5, 2004

II "The Maximum Gain of Increasing the Number of Preemptions in Multiprocessor Scheduling"
Kamilla Klonowska, Lars Lundberg, Håkan Lennerstad
submitted for publication

III "Using Golomb Rulers for Optimal Recovery Schemes in Fault Tolerant Distributed Computing"
Kamilla Klonowska, Lars Lundberg, Håkan Lennerstad
Proceedings of the 17th International Parallel & Distributed Processing Symposium IPDPS 2003, Nice, France, April 2003

IV "Using Modulo Rulers for Optimal Recovery Schemes in Distributed Computing"
Kamilla Klonowska, Lars Lundberg, Håkan Lennerstad, Charlie Svahnberg
Proceedings of the 10th International Symposium PRDC 2004, Papeete, Tahiti, French Polynesia, March 2004

V "Extended Golomb Rulers as the New Recovery Schemes in Distributed Dependable Computing"
Kamilla Klonowska, Lars Lundberg, Håkan Lennerstad, Charlie Svahnberg
Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05), Denver, Colorado, April 2005

VI "Optimal Recovery Schemes in Fault Tolerant Distributed Computing"
Kamilla Klonowska, Lars Lundberg, Håkan Lennerstad, Charlie Svahnberg
Acta Informatica, 41(6), 2005

Publications that are related but not included in this thesis:

VII "Using Optimal Golomb Rulers for Minimizing Collisions in Closed Hashing"
Lars Lundberg, Håkan Lennerstad, Kamilla Klonowska, Göran Gustafsson
Proceedings of Advances in Computer Science - ASIAN 2004, Higher-Level Decision Making, 9th Asian Computing Science Conference, Thailand, December 2004; Lecture Notes in Computer Science, 3321 Springer 2004, ISBN 3-540-24087-X

VIII "Bounding the Minimal Completion Time in High Performance Parallel Processing"
Lars Lundberg, Magnus Broberg, Kamilla Klonowska
International Journal of High Performance Computing and Networking, Vol. 2, No. 1, 2004

IX "Comparing the Optimal Performance of Multiprocessor Architectures"
Lars Lundberg, Kamilla Klonowska, Magnus Broberg, Håkan Lennerstad
Proceedings of the twenty-first IASTED International Multi-Conference Applied Informatics AI 2003, Innsbruck, Austria, February 2003

X "Recovery Schemes for High Availability and High Performance Distributed Real-Time Computing"
Lars Lundberg, Daniel Häggander, Kamilla Klonowska, Charlie Svahnberg
Proceedings of the 17th International Parallel & Distributed Processing Symposium IPDPS 2003, Nice, France, April 2003

XI "Evaluating Heuristic Scheduling Algorithms for High Performance Parallel Processing"
Lars Lundberg, Magnus Broberg, Kamilla Klonowska
Proceedings of the fifth International Symposium on High Performance Computing ISHPC-V 2003, Tokyo, Japan, October 2003

XII "A Method for Bounding the Minimal Completion Time in Multiprocessors"
Magnus Broberg, Lars Lundberg, Kamilla Klonowska
Technical Report, Blekinge Institute of Technology, 2002

XIII "Optimal Performance Comparisons of Massively Parallel Multiprocessors"
Håkan Lennerstad, Lars Lundberg, Kamilla Klonowska
submitted for publication

Contents

Theoretical Aspects on Performance Bounds and Fault Tolerance in Parallel Computing

Paper Section

Introduction

Theoretical Aspects on Performance Bounds and Fault Tolerance in Parallel Computing

1 Introduction

If one single processor does not give the required performance, one improvement alternative may be to execute the application on several processors that work in parallel. To increase the performance in parallel computing, we would like to spread the work between the processors (computers) as evenly as possible. This technique is called load balancing. There are many ways to implement parallel computing with multiple processors. One of them is a loosely coupled parallel system consisting of a number of stand-alone computers, connected by a network, where a process can only be executed by the processor on which it was started. This allocation of processes to processors is called static allocation.

A loosely coupled parallel system is very attractive due to its low cost and the potential improvement in availability and fault-tolerance. If one computer fails, the work on the failed computer can be taken over by another computer. However, one challenging problem in loosely coupled systems is to schedule processes among processors to achieve some performance goals, such as minimizing communication delays and completion time. The completion time of a program is also called the makespan.

Another parallel system is a tightly coupled Symmetric MultiProcessor system (SMP), consisting of multiple similar processors within the same computer, interconnected by a bus or some other fast interconnection network. Here a process may be executed by different processors during different time periods. This allocation of processes to processors is called dynamic allocation. An SMP system offers high performance and efficient load balancing, but does not offer availability. If one processor fails the entire application will usually fail.

One is often interested in improving the performance by reducing the completion time of a parallel program consisting of a number of synchronizing processes (static allocation). There is a conflict between the execution of the tasks and the communication between them. One extreme example is a parallel program that executes on only one processor. This program is not affected by communication/synchronization overhead, but it suffers from serious load imbalance. On the other hand, if the same parallel program is executed on many processors, then the load may be evenly spread among the processors, but the communication cost can be very high.

Finding a scheduling algorithm that minimizes the completion time for a parallel program consisting of a number of processes is one of the classical computer science problems, and it has been shown to be NP-hard (Garey et al., [21]). A number of good heuristic methods have been suggested, but it is difficult to know when to stop the heuristic search for better schedules. Therefore it is important to know the optimal bounds to figure out how close/far the algorithms are from the optimal results.

An important question is how much performance one can gain by allowing dynamic allocation provided that we are able to find (almost) optimal scheduling and allocation algorithms. The answer to this question would provide important input when we want to balance the additional cost and complexity of allowing dynamic allocation against the performance loss of restricting ourself to static allocation. In Paper I we define a function that answers this question for a very wide range of multiprocessors and parallel programs.

Another possibility to increase the performance is to allow preemptions. In a preemptive schedule, a process can be interrupted by other processes and then resumed on the same or on another processor. Scheduling with preemptions is more flexible, but it is not always possible and preemptions can be costly due to overhead for context switching. A preemption can be made at any point in time. Here, the state must be saved before a process is preempted and then restored when it resumes. This means that there is a trade-off - on the one hand one needs preemptions in order to obtain a balanced load, but on the other hand one would like to limit the number of preemptions in order to minimize the overhead costs. The optimal solution to this trade-off problem depends on a number of parameters, such as the cost for preempting and then later restarting a process. One crucial piece of information for making a well informed trade-off decision is the possible performance gain if the number of preemptions is increased, assuming that there are no overhead costs. In Paper II we present a tight upper bound on the maximal gain of increasing the number of preemptions for any parallel program. This means that we compare the minimal makespan for extremal (worst-case) parallel programs consisting of a set of independent jobs on a multiprocessor when allowing different number of preemptions.

In fault tolerant parallel systems, like clusters, the availability is obtained by failover techniques. In its simplest form cluster availability is obtained by having two computers, one active and one stand-by. If the primary computer fails, the secondary simply takes over the work. In order to obtain higher availability, one may want to use more than just two computers [Pfister, 54]. However, it can be very costly to build a large cluster system with many stand-by computers. It is often more attractive to fail over to the computers that already are active in the system, but it is difficult to decide on which computer the work on the failing computer should be executed. This is particularly challenging if this decision has to be made statically before the program starts executing. The goal here is to find a redistribution scheme that maintains high performance even when one or more computers go down. In Papers III - VI we present four different redistribution schemes.

This thesis is divided in two parts. Fig. 1 presents an overview of the thesis as two sets of papers corresponding to Part I and Part II and their intersection of common concepts. In both parts we use theoretical techniques that lead to explicit worst-case programs and scenarios. The correctness is based on mathematical proofs.

In Part I we present performance bounds on the ratio for minimal completion time of a parallel program executed in two scenario. Scenario one, the ratio for minimal completion time when processes can be reallocated compared to when they cannot be reallocated to other processors during their execution time (Paper I). Scenario two,

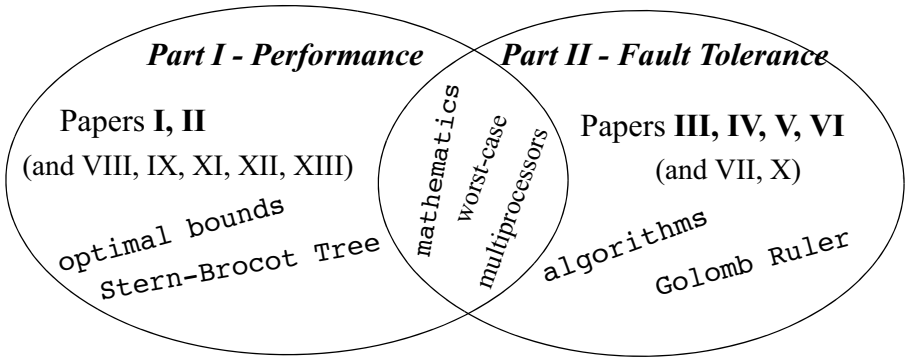


Fig. 1. Overview of the papers

when a schedule is preemptive, the ratio for minimal completion time when we use two different number of preemptions (Paper II). In Section 2 the classification of multiprocessor scheduling and related work on some bounds and complexity are presented. In Part II we discusses the problem of redistribution of the load among running computers in a parallel system if one ore more computers in the system go down. Here we deliver four different redistribution algorithms (Papers III, IV, V and VI). In Section 3 the classification of the faults and related work in fault tolerance system are presented.

1.1 Research Questions

In this thesis three main questions are in focus.

Part I: Performance Bounds in Parallel Computing

1. *Consider two parallel architectures: a Symmetric MultiProcessor (SMP) with dynamic allocation and a distributed system (or cluster) with static allocation.*
How much shorter can the completion time of a parallel program be on the SMP compared to the distributed system provided that we use optimal schedules?

2. *Consider a multiprocessor with identical processors and a parallel program consisting of independent processes, and an optimal schedule with preemptions.*
If the number of preemptions is increased, how large can the gain in completion time of a parallel program be?

Part II: Load balancing in Parallel Computing

3. *Consider a cluster with a number of computers. Consider a worst-case crash scenario, i.e. a scenario when the most unfavorable combinations of computers go down.*
How can the load be evenly and efficiently redistributed among the running computers (using static redistribution)?

1.2 Research Methodology

Since we are interested in worst case/extreme case scenarios in infinite sets, we cannot use empirical method. To find the answers to the research questions, theoretical techniques that lead to explicit worst-case programs are used. The correctness is based on mathematical proofs (Papers I and II). Furthermore, a number of tests on multiprocessor and distributed Sun/Solaris environments have been done to validate and illustrate some of the results in Paper I. In Paper II we take advantage of a number theoretic formula based on the Stern-Brocot tree. Papers III and IV are also based on number theory. In Paper III we present schemes that are based on so called Golomb rulers. Golomb rulers have previously been used in rather different context, e.g. radio astronomy, (placement of antennas), X-ray crystallography, data encryption, geographical mapping and in hash-tables [Lundberg et al., 46]. In Paper IV we extend the results from Paper III by adding a new type of ruler, a "modulo ruler", giving a result that is valid in more cases than Paper III. In Paper V we again extend the Golomb rulers to construct the new recovery scheme (trapezium) which give better performance. In Paper VI we exhaust the cases by calculating the best possible recovery schemes for any number of crashed computers by a branch-and-bound algorithm.

1.3 Research Contribution

The main contributions in this thesis are:

Part I: The optimal bounds:

- an optimal bound on the minimal completion time for all parallel programs with n processes and a synchronization granularity z executed in the multiprocessor system with static allocation with k processors and a communication delay t, compared to the system with dynamic allocation with q processors (see Paper I and Section 4.1.1):

  $$H(n, k, q, t, z) = min_A H(A, n, k, q, t, z) .$$

  Here A denotes an allocation of processes to processors, where a_j processes are allocated to the j:th computer. Hence, the allocation is given by the allocation sequence $(a_1, \ldots, a_k)$, where $\sum_{j=1}^{k} a_j = n$.

  Furthermore, $H(A, n, k, q, t, z) = g(A, n, k, q) + zr(A, n, k, t)$,

  where $g(A, n, k, q) = \left(1 / \binom{n}{q}\right) \sum_{I} \max(i_1, \ldots, i_k) \binom{a_1}{i_1} \cdot \ldots \cdot \binom{a_k}{i_k}$

  and $r(A, n, k, t) = \dfrac{n(n-1) - \sum_{i=1}^{k} a_i(a_i - 1)}{n(n-1)} \cdot t$.

The sum is taken over all decreasing sequences $I = \{i_1, \ldots, i_k\}$ of nonnegative integers such that $\sum_{j=1}^{k} i_j = q$.

- an optimal bound on the minimal completion time of a parallel program executed in the multiprocessor system with m processors comparing the schedules with two different numbers of preemptions, i and j, where $i < j$ (see Paper II and Section 4.1.2):

$$G(m, i, j) = 2\left\lfloor \frac{\min(m, i+j+1)}{\min(m+i+1, 2i+j+2)} \right\rfloor_{\min(m-j,\, i+1)}$$

Here the notation comes from the following definition:

Def: $\left\lfloor \frac{m}{n} \right\rfloor_c = \max\left(\min\left(\frac{m_1}{n_1}, \ldots, \frac{m_c}{n_c}\right)\right)$, where the maximum is taken over all sets of integers $m_1, \ldots, m_c$ and $n_1, \ldots, n_c$ so that $m_1 + \ldots + m_c = m$, $n_1 + \ldots + n_c = n$ for all $m_k > 0$ and $n_k \geq 0$ (Lennerstad and Lundberg, [40]).

Part II: The algorithms:

- Golomb Recovery Scheme (see Paper III and Section 4.2.1)
- Modulo Recovery Scheme (see Paper IV and Section 4.2.2)
- Trapezium Recovery Scheme (see Paper V and Section 4.2.4)
- Optimal Recovery Scheme (see Paper VI and Section 4.2.3)

2 Multiprocessor Scheduling (Part I)

In this section we introduce a classification of scheduling problems and present related work with bounds and complexity for some of them.

A very good taxonomy of task scheduling in distributed computing systems is presented by Casavant and Kuhl in [8] (see Fig. 2). Here, *static* means *off-line*, i.e. when the decision for a process is made before it is executed, while *dynamic* means *on-line*, i.e. when it is unknown when and where the process will execute during its lifetime. In Paper I and II we consider a global optimal static scheduling, when we know the information of the parallel program in advance.

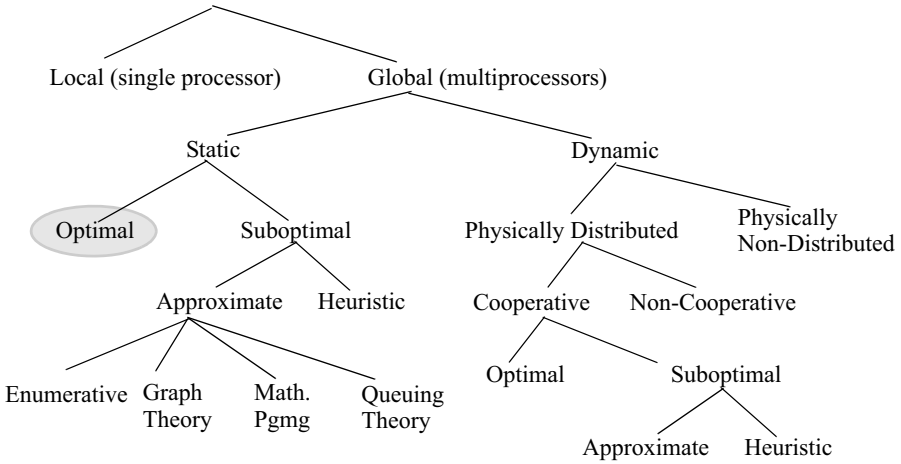


Fig. 2. Task Scheduling Characteristics ([Casavant and Kuhl, 8])

2.1 Classification of scheduling problems

We start with a definition of scheduling developed by Graham, Lawler, Lenstra and Rinnooy Kan [28]:

Consider m machines M_i ($i = 1,...,m$) that have to process n jobs J_j ($j = 1,...,n$).

"A *schedule* is an allocation of one or more time intervals on one or more machines to each job. A schedule is *feasible* if no two time intervals on the same machine overlap, if no two time intervals allocated to the same job overlap, and if, in addition, it meets a number of specific requirements concerning the machine environment and the job characteristics. A schedule is *optimal* if it minimizes a given optimality criterion."

Furthermore, the authors (Graham et al. [28]) proposed a classification of scheduling problems, which has been widely used in the literature, e.g. Lawler et al. [38], Blazewicz et al. [4], Pinedo [55]. The classification shows how many problems in scheduling theory can be if we would look at all combinations. This thesis is limited to one type of machines and one criterion, i.e. a minimal completion time.

A classification of scheduling problem is presented as a three-field classification $\alpha|\beta|\gamma$, where α represents the processor environment, β the job characteristics and γ the optimality criterion as follows:

The first field $\alpha = \alpha_1\alpha_2$ consists of two parameters: $\alpha_1 \in \{\circ, P, Q, R, O, F, J\}$, where $\circ$ (denotes an empty symbol) represents a single processor; *P*: identical processors; *Q*: uniform processors, i.e. the processors with a given speed; *R*: unrelated processors, i.e. the processors with job-dependent speeds; *O*: dedicated processors: open shop sequencing, in which each job has to be processed again on each one of the *m* processors; there are no restrictions regarding the order of each job; *F*: dedicated processors: flow shop sequencing, in which each job has to be processed on each one of the *m* processors with the special order, i.e. first on process one, then on process two, and so on. After completion on one processor, a job joins the queue at the next processor; *J*: dedicated processors: job shop sequencing, each job has its own order to follow; and $\alpha_2 = k$: the number of processors (Pinedo, [55])

In Paper I we describe a scenario of identical machines with different number of processors, i.e. a set *Pk* for the parallel architecture with *k* identical processors and static allocation, and a set *Pq* - with *q* identical processors and dynamic allocation. Then $\alpha_1 = P$. In Paper II we describe a scenario of identical machines with *m* processors, so also here $\alpha_1 = P$.

The second field β describes the job characteristics. Here we describe only some of the possible values. $\beta \in \{\circ, pmtn, prec, tree, res, r_j, p_j = 1\}$, where $\circ$ means that there are no restrictions on the jobs; *pmtn* means that preemptions are allowed; *prec* is precedence relation between the jobs, *tree* is a rooted tree representation; *res* is specified resource constraints; r_j is release dates; $p_j = 1$: each job has a unit processing requirement (occurs only if $\alpha \in \{\circ, P, Q\}$).

In Paper I we do not have any job characteristics, so for both cases (*Pk* and *Pq*) $\beta \in \{\circ\}$. In Paper II we present the scenario with limited number of preemptions, i.e. $\beta \in \{pmtn\}$.

The last field γ describes an optimality criterion. The most commonly chosen are the maximum completion time or makespan ($C_{\max}$), the total completion time ($\sum C_j$), or the maximum lateness $L_{\max}$, where a lateness of a task is its completion time minus deadline [Graham et al., 28].

In Paper I and II we are interesting in minimizing the makespan, i.e. to minimize the maximum completion time, so $\gamma = C_{\max}$.

Using this notation we can represent the problem of minimizing maximum completion time on identical parallel machines allowing preemption as $P|pmtn|C_{\max}$.

2.2 Bounds and Complexity on Multiprocessor Scheduling

The problem of scheduling a parallel program on *m* machines has been shown to be NP-hard (Garey et al., [21]). Many scheduling problems are easier to solve when we allow preemptions, e.g. a problem of minimizing maximum completion time ($C_{\max}$), without preemptions on two identical processors ($P2||C_{\max}$) is NP-hard (Karp [33]), while the problem for preemptive parallel scheduling for more than 2 processors ($P|pmtn|C_{\max}$) is solvable in $O(n)$ time (McNaughton, [50]). However, there is still a practical question of allowing or not allowing preemptions.

In 1966, Graham [26], using List Scheduling (LS) rule, has proved that: $C_{\max}(LS)/C^*_{\max} \le 2 - 1/m$, where *m* is the number of processors and $C^*_{\max}$ denotes the maximum completion time for the optimal schedule. The idea of list scheduling is to make an ordered list of processes by assigning them some priorities. The next job from the list is assigned to the first available machine. One of the most often used algorithms for solving $P||C_{\max}$ problem is the Longest Processing Time (LPT) algorithm, which is a kind of list scheduling. Here, the jobs are arranged in decreasing processing time order and when a machine is available, the largest job is ready to begin processing. The complexity of this algorithm is $O(n \log n)$ and the upper bound is established by Graham ([27]): $C_{\max}(LPT)/C^*_{\max} \le \frac{4}{3} - \frac{1}{3m}$. However, in LPT algorithm, the cost for process reallocation and synchronization is neglected. To achieve better performance, Coffman et al. [12] introduced another approximation algorithm, called Multifit (MF) with makespan $C_{\max}(MF_k)/C^*_{\max} \le 1.22 + 2^{-k}$. This algorithm is based on binpacking techniques, where the jobs are taken in non increasing order and each job is placed into the first processor into which it will fit. This bound was further improved by Friesen to 1.20 [19], Friesen and Langstone [20] to 1.18 and then by Yue [62] to 13/11, which is tight. Hochbaum and Shmoys [29] have developed a polynomial approximation scheme, based on the multifit approach, with the computational complexity $O((n/\varepsilon)^{1/\varepsilon^2})$ for *n* processes.

Lennerstad and Lundberg in [39] established an optimal upper bound on the gain of using two different parallel architecture, with and without migrations of processes, where the number of processors in both systems are equal. In [41] the result is extended for a scenario with different number of processors. Paper I is an extension of those papers, i.e. the cost for communication and synchronization between processors is included (see Paper I and Section 4.1.1).

A problem of minimizing the makespan with preemptions, $P|pmtn|C_{\max}$, can be solved very efficiently with time complexity $O(n)$. The makespan of any preemptive

schedule is at least $C^*_{\max} = \max\left\{\max_j\{p_j\}, \frac{1}{m}\sum_{j=1}^{n} p_j\right\}$ (McNaughton, [50]). However, by the McNaughton rule, no more than $m-1$ preemptions are needed in the unlimited case.

For the scheduling with precedence constraints, i.e. $P|prec|C_{\max}$, Graham's [26] bound from 1966 ($C_{\max}(LS)/C^*_{\max} \le 2 - 1/m$) still holds. If the jobs have unit-length, the problem $P|prec, p_j{=}1|C_{\max}$ is NP-hard (Ullman, [60]). However, if the number of processors is limited to 2, then $P2|prec, p_j{=}1|C_{\max}$ is solvable in $O(n^2)$ time (Coffman & Graham, [13]). Coffman and Garey [11] proved that the least makespan achievable by a non-preemptive schedule is no more than 4/3 the least makespan achievable when preemptions are allowed.

In 1972, Liu [43] conjectured that for any set of tasks and precedence constraints among them, running on two processors, the least makespan achievable by a nonpreemptive schedule is no more than $4/3$ the least makespan achievable by a preemptive schedule. The conjecture was proved in 1993 by Coffman and Garey [11]. Here the authors generalize the results to the numbers 4/3, 3/2, 8/5,... i.e. to the numbers $2k/(k+1)$ for some $k \ge 2$. The number *k* depends on the relative number of preemptions available.

Braun and Schmidt [5] proved 2003 a formula that compares a preemptive schedule with *i* preemptions $C^{ip*}_{\max}$ to a schedule with unlimited number of preemptions $C^{p*}_{\max}$ in the worst case. They generalized the bound 4/3 to the formula $C^{ip*}_{\max}/C^{p*}_{\max} \le 2 - 2/(m/(i+1)+1)$.

In Paper II we extend the results of Braun and Schmidt [5], by comparing a preemptive schedule with *i* preemptions to a schedule with *j* preemptions, where $i < j$ (see Paper II and Section 4.1.2).

3 Load Balancing and Fault Tolerance (Part II)

This section presents related work on redistribution the workload in fault tolerance system and a definition and classification of the faults.

An important issue in a multiprocessor system is how to redistribute the workload in the case of fault of one or more processors. In that case the load should be redistributed to the other processors in the system maintaining maximal load balance.

Load balancing problem can be handle by dynamic policies, where transfer decisions depend on the actual current system state, or by static policies that are generally based on the information of the average behavior of the system. The transfer decisions are then independent of the actual current system state. This makes them less complex than dynamic policies [Kameda et al., 32].

A system is fault tolerance "if its programs can be properly executed despite the occurrence of logic faults" (Avizienis, 1967, [2]). Krishna and Shin [35] define the fault tolerance as "an ability of a system to respond gracefully to an unexpected hardware or software failure". Therefore, many fault-tolerant computer systems mirror all operations, e.g. every operation is performed on two or more duplicate systems in that sense, that if one fails the other can take over its job. This technique is also used in clusters [Pfister, 54]. In [Cristian, 14] fault tolerance for distributed computing is discussed from a wide viewpoint.

Besides hardware failures, the intermittent failures can occur due to software events. In [Vaidya, 61] a "two-level" recovery scheme is presented and evaluated. The recovery scheme has been implemented on a cluster. The authors evaluate the impact of checkpoint latency on the performance of the recovery scheme. For transaction-oriented systems, Gelenbe [22] has proposed "multiple check pointing" approach that is similar to the multi-level recovery scheme presented in [61]. A mathematical model of transaction-oriented system under intermittent failures is proposed in [Gelenbe and Derochete, 24]. Here, the system is assumed to operate in a standard checkpoint-roll-back-recovery scheme. In [Chabridon and Gelenbe, 9] and [Gelenbe and Chabridon, 23] the authors propose several algorithms which can detect tasks failures and restart failed tasks. They analyze the behavior of parallel programs represented by a random task graph in a multiprocessor environment. However, all these algorithms act dynamically.

If a single element of hardware or software fails and brings down the entire computer system we talk about single point of failure (Pfister, [54]). In the thesis we look at more advanced scenario, where an arbitrary number of faults can occur, i.e. we look at the system with no single point of failure.

3.1 Fault Model

A *failure* is "an event that occurs when the delivered service deviates from correct service". "The deviation is called *error*". "The adjudged or hypothesized cause of an error is called a *fault*." (Avizienis et al., [3]) The events can be presented as in Fig. 3.

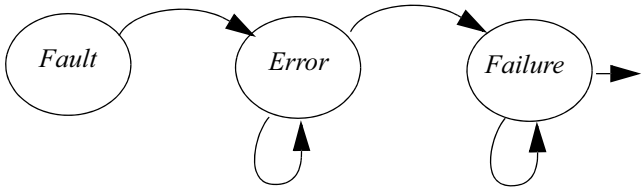


Fig. 3. Representation of a fault, error and failure ([3])

Avizienis et al. [3] presents the elementary fault classes according to eight basic viewpoints: phase of creation or occurrence, system boundaries, phenomenological causes, dimension, objective, intent, capability and persistence, where each of viewpoint consists of two additional viewpoints. If all combinations would be possible, then it would be 256 different combined fault classes ([3]).

A classification of a fault behaviour of its processors or communication controllers is the following:

- crash fault - the processor stops computing/transmission messages;
- omission fault - missed message, a processor loses its content;
- timing fault - the message arrives too late or early;
- fail-stop fault - a process/message is stopped from executing, possibly forever;
- Byzantine fault - a processor may do anything (Cristian, [14]).

From the point of view of occurrences, faults can be classified into:

- transient fault - a fault starts at a particular time, remains in the system for some period and the disappears;
- permanent fault - a fault starts at a particular time and remains in the system until it is repaired;
- intermittent fault - a fault occurs from time to time (Burns and Wellings, [8]).

Our main focus in Papers III-VI is on permanent crash faults.

3.2 Reliability vs. Availability

Computing and communication systems are characterized by fundamental properties: functionality, performance, dependability and security, and cost (Avizienis, [3]). Reliability and availability are two of the five characteristics of dependability. Both are measured by two components: a mean-time-between-failures (MTBF) and a mean-time-to-repair (MTTR).

Reliability (R) is the ability of a computing system to operate without failing, while availability (A) is a readiness for correct service.

Availability is defined as a proportion of time that the system is up (Laprie et al., [37]): $A = \frac{\text{MTBF}}{\text{MTBF} + \text{MTTR}} \approx 1 - \frac{\text{MTTR}}{\text{MTBF}}$, if MTBF is very much greater than MTTR.

A system with 0.999 availability is more reliable than a system with 0.99 availability. Furthermore, a system with 0.99 availability has 1 - 0.99 = 0.01 probability of failure. Hence, a failure is $F = 1 - A$ and a reliability $R = 1/(1 - A)$ (Laprie et al., [37]).

The availability of the system can be measured by a number of 9s, e.g. a system with 0.999 availability is called three 9s and belongs to class 3. Fig. 4 presents a classification of the availability of the system converted to an average down time in a given time period (Pfister [54] and Laprie et al., [37]).

Class / nr of 9s	% Available	Hours / Year	Minutes / Month
2	99,9	87.60	438
3	99,99	8.76	43.8
4	99,999	0.88	4.38
5	99,9999	0.9	0.44
6	99,99999	0.1	0.04

Fig. 4. A classification of availability systems ([54], [37])

The systems of class 4-6 are called high availability systems (Pfister, [54]).

4 Summarizing of the papers

This section summarize the papers involved in the thesis.

4.1 Part I

4.1.1 *Comparing the optimal performance of parallel architectures* **(Paper I)**

This paper extends the results in [Lennerstad and Lundberg, 39] and [Lennerstad and Lundberg, 42], where the synchronization cost was neglected. Here, a bound on the gain of using a system with q processors and run-time process reallocation compared to using a system with k processors, no reallocation and a communication delay t, for a program with n processes and a synchronization granularity z is presented. The main contribution in this paper is that we handle and validate a more realistic computer model, where the communication delay t for each synchronization signal and the granularity z of the program are taken into consideration. We present transformations which enable us to separate execution and synchronization. Analyzing the parts separately and comparing them, we found a formula $H(n, k, q, t, z)$ that produces the minimal completion time: $H(n, k, q, t, z) = min_A H(A, n, k, q, t, z)$. Here A denotes an allocation of processes to processors, where a_j processes are allocated to the j:th computer. Hence, the allocation is given by the allocation sequence $(a_1, \ldots, a_k)$, where $\sum_{j=1}^{k} a_j = n$. Furthermore, the sum is taken over all decreasing sequences $I = \{i_1, \ldots, i_k\}$ of nonnegative integers such that $\sum_{j=1}^{k} i_j = q$.

$H(A, n, k, q, t, z) = g(A, n, k, q) + zr(A, n, k, t)$, where

$$g(A, n, k, q) = \left(1 / \binom{n}{q}\right) \sum_{I} \max(i_1, \ldots, i_k) \binom{a_1}{i_1} \cdot \ldots \cdot \binom{a_k}{i_k} \text{ and}$$

$$r(A, n, k, t) = \frac{n(n-1) - \sum_{i=1}^{k} a_i(a_i - 1)}{n(n-1)} \cdot t .$$

The execution part, represented by $g(A, n, k, q)$, corresponds to the previous results (Lundberg and Lennerstad [39, 42]). The function $g(A, n, k, q)$ is convex in the sense that the value decreases if the load of a worst case program is distributed more evenly among the computers. However, the synchronization part, represented by $r(A, n, k, t)$, is concave. This quantity increases if the load of a worst case program is distributed more evenly. This makes the minimization of the sum H a delicate matter. The type of allocation which is optimal depends strongly on the value of tz. If tz is small, then the

execution dominates, and partitions representing even distributions, uniform partitions, are optimal. If tz is large, then the synchronization dominates, and partitions where all processes are allocated to the same computer are optimal.

Here is an example of the calculation of the execution part of program P calculated first by the formula and then using the vector representation.

Let $n = 3, q = 2, k = 2$. Then $A = \{a_1, a_2; a_1 + a_2 = 3\} = \{(1, 2), (2, 1)\}$ and $I = \{i_1, i_2; i_1 + i_2 = 2\} = \{(1, 1), (2, 0), (0, 2)\}$ and then

$$g(A, 3, 2, 2) = \left(1/\binom{3}{2}\right)\left[1 \cdot \binom{1}{1}\binom{2}{1} + 2 \cdot \binom{1}{2}\binom{2}{0} + 2 \cdot \binom{1}{0}\binom{2}{2}\right] = 4/3 .$$

Using the vector representation of program P with the allocation (1,2) we have:

$(a_1, a_2) = (1, 2)$:		(i_1, i_2):	max of (i_1, i_2):
1 | 1 0		(1, 1)	1
1 | 0 1	⟶	(1, 1) ⟶	1
0 | 1 1		(0, 2)	2

$$g(A, 3, 2, 2) = \sum(\text{max of } (i_1, i_2))/(\text{nr of rows in vector representation}) = 4/3 .$$

4.1.2 *The Maximum Gain of Increasing the Number of Preemptions in Multiprocessor Scheduling* **(Paper II)**

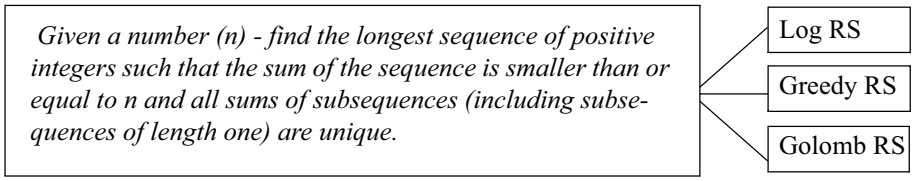
This paper generalize the results by Braun and Schmidt [5]. We present a tight upper bound on the maximal gain of increasing the number of preemptions for any parallel program consisting of a number of independent jobs when using m identical processors. We calculate how large the ratio of the minimal makespans using i and j preemptions respectively, $i < j$, can be. We compare i preemptions with j preemptions in the worst case. We thus allow j from $i + 1$ to $m - 1$, while the problem solved in [5] corresponds to $j = m - 1$. In the case $m \geq i + j + 1$, which does not coincide with $j = m - 1$ unless $i = 0$, we obtain the optimal bound $2(\lfloor j/(i+1) \rfloor + 1)/(\lfloor j/(i+1) \rfloor + 2)$. For example, excluding one preemption $(i = j - 1)$ can never deteriorate the makespan more than a factor 4/3, but may do so. This argument cannot be iterated, since different sets of jobs are worst case, depending on the parameters i and j. In the case $m < i + j + 1$ we present a formula and a fast algorithm based on the Stern-Brocot tree.

4.2 Part II

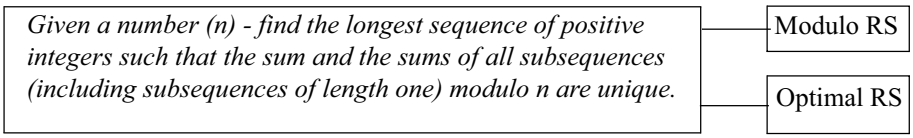
In fault tolerant distributed systems it is difficult to decide on which processor the processes should be executed. When all computers are up and running, we would like the load to be evenly distributed. The load on some processors will, however, increase when one or more processors are down, but also under these conditions we would like to distribute the load as evenly as possible on the remaining processors. The distribution of the load when a computer goes down is decided by the *recovery lists* of the processes running on the faulty processor. The set of all recovery lists is referred to as the *recovery scheme* (RS). Hence the load distribution is completely determined by the recovery scheme for any set of processors that are down.

The computer science problem is reformulated in the following papers into different mathematical problems that produce different static recovery schemes. The corresponding mathematical problems with the corresponding recovery schemes turn out to be the following:

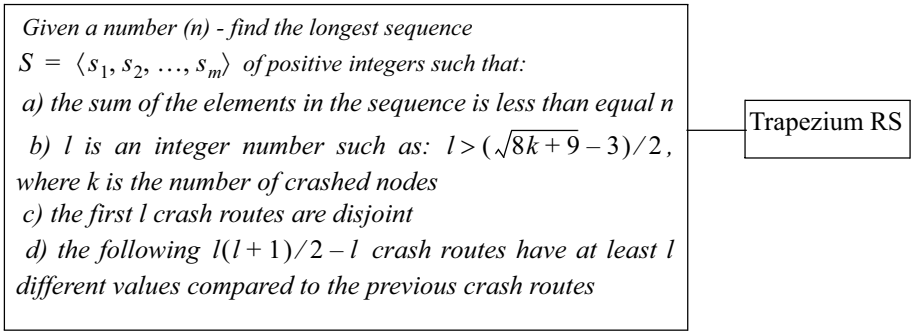
Paper III:



Paper IV and VI



Paper V



4.2.1 *Using Golomb Rulers for Optimal Recovery Schemes in Fault Tolerant Distributed Computing* **(Paper III)**

This paper extends the results presented in Lundberg and Svahnberg [47]. Here we present how the Golomb ruler (a sequence of non-negative integers such that no two distinct pairs of numbers from the set have the same difference) (Fig. 5) is applied to the problem of finding an optimal recovery scheme. Golomb rulers are known for lengths up to 41912 (with the 211 marks). Of these the first 373 (with 23 marks) are known to be optimal.

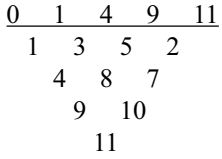


Fig. 5. The triangle presentation of the OGR 11

The Golomb recovery lists are build as followed:

Let G_n be the Golomb ruler with sum $n + 1$ and let $G_n(x)$ be the x:th entry in G_n, e.g. G_{12} = <1,4,9,11> and $G_{12}(1) = 1$, $G_{12}(2) = 4$ and so on. Let then $g_n = x$ be the number of crashed computers with optimal behavior when we have n computers, e.g. $g_{12} = 4$. For intermediate values of k we use the smaller Golomb ruler, and the rest of the recovery list is filled with the remaining numbers up to k-1. For example, by filling with remaining numbers, the ruler G_{12} gives the list {1,4,9,11,2,3,5,6,7,8,10}.

The sequence referred to Golomb recovery schemes is in this case <1,3,5,2> (i.e. the differences between the numbers from the list). All other recovery lists are obtained from this by adding a fixed number to all entries in the modulo sense, i.e. R_i = {(i+1) mod n, (i+2) mod n, (i+3) mod n,…, (i+n-1) mod n}, where R_i is the recovery list for process i. If n = 12 we get: R_0 = <1,3,5,2>, R_1 = <2,4,6,3>, R_2= <3,5,7,4> and so on.

In this paper we present also the greedy algorithm, which is constructed from a sequence with distinct partial sums. It can guarantee optimal behavior until $\lfloor \log_2 n \rfloor$ computers break down, but we can easily calculate it also for large n where no Golomb rules are known.

4.2.2 *Using Modulo Rulers for Optimal Recovery Schemes in Distributed Computing* **(Paper IV)**

Paper IV extends the Golomb recovery scheme. In the formulation which can be handled by the Golomb rulers the wrap-arounds are ignored - i.e. the situations when the total number of "jumps" for a process is larger than the number of computers in the cluster. This problem gives a new mathematical formulation of finding the longest sequence of positive integers such that the sum and the sums of all subsequences (including subsequences of length one) modulo n are unique (for a given n). This

mathematical formulation of the computer science problem gives new more powerful recovery schemes, called *Modulo* schemes, that are optimal for a larger number of crashed computers. Fig. 6 presents an example of a modulo-11 sequence with all modulo differences, that does not exist in the previous case. The recovery lists (and recovery scheme) are constructed in the same way as the Greedy or Golomb recovery lists (and scheme).

0 1 6 3 10
1 5 8 7
6 2 4
3 9
10

Fig. 6. Modulo sequence for n = 11 with all differences

4.2.3 *Extended Golomb Rulers as the New Recovery Schemes in Distributed Dependable Computing* **(Paper V)**

A contribution in this paper is that we have presented new recovery schemes, called trapezium recovery schemes, where the first part of the schemes is based on the known Golomb rulers, (i.e. the crash routes are disjoint) and the second part is constructed in a way, where the following crash routes have at least l unique values compared to previous crash routes. The trapezium recovery schemes guarantee better performance than the Golomb schemes and are simple to calculate. Fig. 7 compares the number of crashes of the trapezium scheme ("trapezium"), with the performance of the scheme using Golomb rulers ("golomb") as a function of the number of nodes in a cluster (n) up to n = 1024.

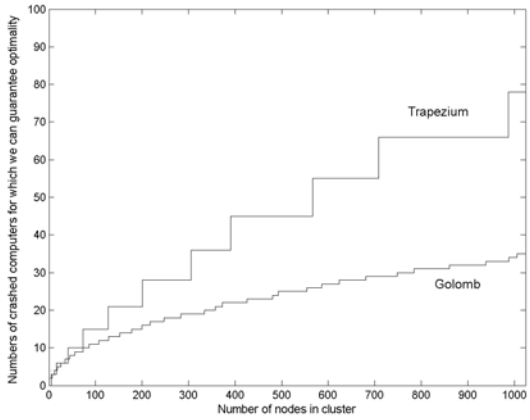


Fig. 7. The difference between the trapezium scheme ("trapezium") and OGRs scheme ("golomb")

The goal of this paper was to find good recovery schemes, that are better than the already known and are easily to calculate also for large n. In Paper V we have found

the best possible recovery schemes for any number of crashed nodes in a cluster. To find such a recovery scheme is a very computational complex task. Due to the complexity of the problem we have only been able to present optimal recovery schemes for a maximum of 21 nodes in a cluster.

4.2.4 *Optimal Recovery Schemes in Fault Tolerant Distributed Computing* **(Paper VI)**

In Paper VI we calculate the best possible recovery schemes for any number of crashed computers. We are giving strict priority to a small number of computers down compared to a large number. That means, that we select the set of recovery schemes with optimal worst case behavior when two computers are down, and among these select recovery schemes that have optimal behavior when three computers are down, and so on. We define the set $R(n, p)$ of recovery schemes that minimizes the maximal load for 1,2,...,p computers down in formula:

$$R(n,p) = \left\{ R \in R(n) | L(n,i,R) = \min_{P \in R(n)} L(n,i,P), i = 1, \ldots, p \right\}, \quad \text{where}$$

$L(n, p, R)$ is a load sequence and defines the worst-case behavior after p crashes when using the recovery scheme R. The optimal load sequence is denoted by *SV*. $MV = \max\left(BV(j), \left\lceil \frac{n}{n-j} \right\rceil\right)$, where BV is a bound vector that contains exactly k entries that equals k for all $k \geq 2$.

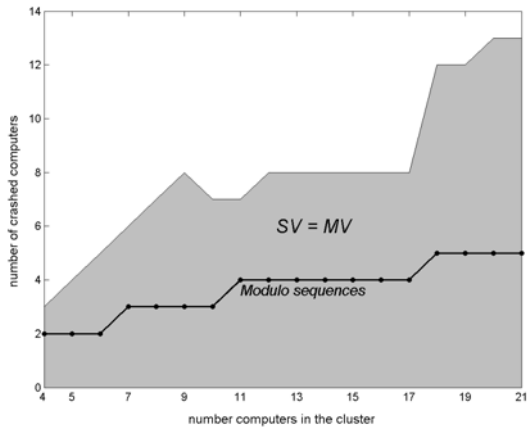


Fig. 8. Comparison of the optimal recovery schemes sequences with the modulo sequences

It is not known if the lower bound *MV* is tight. In this paper we investigate the tightness of the bound *MV* by the optimal load sequence *SV* of $R^*(n)$. We present an algorithm with which we calculate the optimal bound *SV*. In many instances, when we have

a larger number of crashed computers, *SV* do not coincide with *MV*. Fig. 8 shows to which extent *MV* is tight. In the grey area *MV* is tight, since *MV* = *SV* here. For larger values of crashed computers q, *MV* < *SV,* so *MV* is not tight.

5 Future Work

The bounds that are obtained in Part I (papers I and II) are tight within the problem formulation. However, there is a universe of possibilities to find new formulations of both more specific and more general practical situations that has no answer yet, but where the present results may provide a useful foundation for finding answers. Both Paper I and Paper II are in fact extensions of previously studied (and optimally solved) problems. Apart from the results themselves, this is the major contribution to the research domain made by the thesis.

From a mathematical point of view, the results have opened new connections between parallel computer performance and combinatorics. Golomb rulers have found new applications, and new combinatorial problems have arisen and been solved. These problems may be studied and refined as mathematical problems, which then can be interpreted in a computer setting. The new application of the Stern-Brocot tree is one example. I believe that it is possible to find more such connections between computer systems engineering and combinatorics.

6 References

1. Ahmad, I., Kwok, Y.-K., and Wu, M.-Y., *Analysis, Evaluation, and Comparison of Algorithms for Scheduling Task Graphs on Parallel Processors,* in Proceedings of the International Symposium on Parallel Architectures, Algorithms, and Networks, Beijing, China, June 1996, pp. 207-213
2. Avizienis, A., *Design of Fault-Tolerant Computers*, in Proceedings of Fall Joint Computer Conference, AFIPS Conference Proceeding, Vol. 31, Thompson Books, Washington, D.C., 1967, pp. 733-743
3. Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C., *Basic Concepts and Taxonomy of Dependable and Secure Computing*, IEEE Transactions on Dependable and Secure Computing, Vol. 1, No. 1, January-March 2004, pp. 11-33
4. Blazewicz, J., Ecker, K. H., Pesch, E., Schmidt, G., Weglarz, J., *Scheduling Computer and Manufacturing Processes*, Springer Verlag, New York, NY 1996, ISBN 3-540-61496-6
5. Braun, O., Schmidt, G., *Parallel Processor Scheduling with Limited Number of Preemptions*, Siam Journal of Computing, Vol. 32, No. 3, 2003, pp. 671-680
6. Bruno, J., Coffman Jr., E.G., and Sethi, R., *Scheduling Independent Tasks To Reduce Mean Finishing Time*, Communications of the ACM, Vol. 17, No. 7, July 1974, pp. 382-387
7. Burns, A., and Wellings, A., *Real-Time Systems and Programming Languages,* Third Eddition, Pearson, Addison Wesley, ISBN: 0-201-72988-1
8. Casavant, T. L., Kuhl, J. G., *A taxonomy of Scheduling in General-Purpose Distributed Computing Systems,* IEEE Transactions on Software Engineering, Vol. 14, No. 2, February 1988, pp. 141-154
9. Chabridon, S., Gelenbe, E., *Failure Detection Algorithms for a Reliable Execution of Parallel Programs*, in Proceedings of the 14th Symposium on Reliable Distributed Systems, SRDS'14, Bad Neuenahr, Germany, September 1995
10. Coffman Jr., E. G., *Computer and Job-Scheduling Theory,* John Wiley and Sons, Inc., New York, NY, 1976
11. Coffman Jr., E. G., Garey, M. R., *Proof of the 4/3 Conjecture for Preemptive vs. Nonpreemptive Two-Processor Scheduling*, Journal of the ACM, Vol. 40, No. 5, November 1993, pp. 991-1018, ISSN:0004-5411
12. Coffman Jr., E. G., Garey, M. R., Johnson, D. S., *An Application of Bin Packing to Multiprocessor Scheduling*, SIAM J. Computing, 7 (1978), pp. 1-17
13. Coffman Jr., E. G., Graham, R., *Optimal Scheduling for Two-Processor Systems*, Acta Informatica, 1 (1972), pp. 200-213
14. Cristian, F., *Understanding Fault Tolerant Distributed Systems*, Journal of the ACM, Vol. 34, Issue 2, 1991
15. Dollas, A., Rankin, W. T. and McCracken, D., *A New Algorithm for Golomb Ruler Derivation and Proof of the 19 Marker Ruler*, IEEE Trans. Inform. Theory, Vol. 44, No. 1, January 1998, pp. 379-382
16. El-Rewini, H., Ali, H. H., and Lewis, T. G., *Task Scheduling in Multiprocessor Systems,* IEEE Computer Vol. 28, No. 12, December 1995, pp. 27-37

17. El-Rewini, H., Ali, H. H., *Static Scheduling of Conditional Branches in Parallel Programs,* Journal of Parallel and Distributed Computing, Vol. 24, No. 1, January 1995, pp. 41-54
18. Flynn, M.J., *Very High-Speed Computing Systems*, Proceedings of the IEEE, Vol. 54, No. 12, 1966, pp. 1901-1909
19. Friesen, D. K., *Tighter Bounds for the Multifit Processor Scheduling Algorithm,* SIAM Journal of Computing, 13 (1984), pp. 170-181
20. Friesen, D. K., and Langston, M. A., *Evaluation of a MULTIFIT-Based Scheduling Algorithm,* Journal of Algorithms, Vol. 7, Issue 1, March 1986, pp. 35-59
21. Garey, M. R. and Johnson, D. S., *Computers and Intractability - A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, New York, 1979
22. Gelenbe, E., *A Model for Roll-back Recovery With Multiple Checkpoints,* in Proceedings of the 2nd ACM Conference on Software Engineering, October 1976, pp. 251-255
23. Gelenbe, E., Chabridon, S., *Dependable Execution of Distributed Programs*, Elsevier, Simulation Practice and Theory, Vol. 3, No. 1, 1995, pp. 1-16
24. Gelenbe, E., Derochete, D., *Performance of Rollback Recovery Systems under Intermittent Failures*, Communication of the ACM, Vol. 21, No. 6, June 1978, pp. 493-499
25. Gerasoulis, A., Yang, T., *A Comparisons of Clustering Heuristics for Scheduling DAG's on Multiprocessors,* Journal of Parallel and Distributed Computing, 16, 1992, pp. 276-291
26. Graham, R. L., *Bounds for Certain Multiprocessing Anomalies*, Bell System Technical Journal, Vol. 45, No. 9, November 1966, pp.1563-1581
27. Graham, R. L., *Bounds on Multiprocessing Timing Anomalies*, SIAM Journal of Applied Mathematics, Vol. 17, No. 2, 1969, pp. 416-429
28. Graham, R. L., Lawler, E. L., Lenstra, J. K., Rinnooy Kan, A. H. G., *Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey*, Annals of Discrete Mathematics 5, 1979, pp. 287-326
29. Hochbaum, D. S., Shmoys, D. B., *Using Dual Approximation Algorithms for Scheduling Problems Theoretical and Practical Results*, Journal of the ACM (JACM), Vol. 34, Issue 1, January 1987, pp. 144-162
30. Hou, E. S. H., Hong, R., and Ansari, N., *Efficient Multiprocessor Scheduling Based on Genetic Algorithms*, Proceedings of the 16th Annual Conference of the IEEE Industrial Electronics Society - IECON'90, Vol II, Pacific Grove, CA, USA, IEEE, New York, November 1990, pp. 1239-1243
31. Hu, T. C., *Parallel Sequencing and Assembly Line Problems*, Operations Research, Vol. 19, No. 6, November 1961, pp. 841-848
32. Kameda, H., Fathy, E.-Z. S., Ryu, I., Li, J., *A performance Comparison of Dynamic vs. Static Load Balancing Policies in a Mainframe - Personal Computer Network Model,* Information: an International Journal, Vol. 5, No. 4, December 2002, pp. 431-446

33. Karp, R. M., *Reducibility Among Combinatorial Problems*, in R. E. Miller and J. W. Thatcher (editors): Complexity of Computer Computations. New York, 1972, pp. 85–104
34. Khan, A., McCreary, C. L., and Jones, M. S., *A Comparisons of Multiprocessor Scheduling Heuristics,* in Proceedings of the 1994 International Conference on Parallel Processing, CRC Press, Inc., Boca Raton, FL, pp. 243-250
35. Krishna, C. M., Shin, K. G., *Real-Time Systems,* McGraw-Hill International Editions, Computer Science Series, 1997, ISBN 0-07-114243-6
36. Kwok, Y-K., and Ahmad, I., *Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors*, ACM Computing Surveys, Vol. 31, No. 4, December 1999, pp. 406-471
37. Laprie, J.-C., Avizienis, A., Kopetz, H., *Dependability: Basic Concepts and Terminology,* 1992, ISBN:0387822968
38. Lawler, E. G., Lenstra, J. K., Rinnooy Kan A. H. G., Shmoys, D. B., *Sequencing and Scheduling: Algorithms and Complexity,* S.C. Graves et al., Eds., Handbooks in Operations Research and Management Science, Vol. 4, Chapter 9, Elsevier, Amsterdam, 1993, Chapter 9, pp. 445-522
39. Lennerstad, H., Lundberg, L., *An Optimal Execution Time Estimate of Static versus Dynamic Allocation in Multiprocessor Systems*, SIAM Journal of Computing, 24 (4), 1995, pp. 751-764
40. Lennerstad, H., Lundberg, L., *Generalizations of the Floor and Ceiling Functions Using the Stern-Brocot Tree*, Research Report No. 2006:02, Blekinge Institute of Technology, Karlskrona 2006
41. Lennerstad, H., Lundberg, L., *Optimal Combinatorial Functions Comparing Multiprocessor Allocation Performance in Multiprocessor Systems*, SIAM Journal of Computing, 29 (6), 2000, pp. 1816-1838
42. Lennerstad, L., Lundberg, L., *Optimal Scheduling Combinatorics,* Electronic Notes in Discrete Mathematics, Vol. 14, Elsevier, May 2003
43. Liu, C. L., *Optimal Scheduling on Multiprocessor Computing Systems,* in Proceedings of the 13th Annual Symposium on Switching and Automata Theory, IEEE Computer Society, Los Alamitos, CA, 1972, pp. 155-160
44. Lennerstad, H. and Lundberg, L., *An Optimal Execution Time Estimate of Static versus Dynamic Allocation in Multiprocessor Systems*, SIAM Journal of Computing, 24 (4), 1995, pp. 751-764
45. Lundberg, L., *Performance Bounds on Multiprocessor Scheduling Strategies for Chain Structured Programs,* Computer Science Section of BIT, Vol. 33, No. 2, 1993, pp. 190-213
46. Lundberg, L., Lennerstad, H., Klonowska, K., Gustafsson, G., *Using Optimal Golomb Rulers for Minimizing Collisions in Closed Hashing*, Proceedings of Advances in Computer Science - ASIAN 2004, Thailand, December 2004; Lecture Notes in Computer Science, 3321 Springer 2004, pp. 157-168
47. Lundberg, L., and Svahnberg, C., *Optimal Recovery Schemes for High-Availability Cluster and Distributed Computing*, Journal of Parallel and Distributed Computing 61(11), 2001, pp. 1680-1691

48. Markatos, E. P., and LeBlanc, T. B., *Locality-Based Scheduling for Shared Memory Multiprocessors*, Technical Report TR93-0094, ICS-FORTH, Heraklio, Crete, Greece, August 1993, pp. 2-3
49. McCreary, C., Khan, A. A., Thompson, J. J., and McArdle, M. E., *A Comparison of Heuristics for Scheduling DAG's on Multiprocessors,* in Proceedings of the Eighth International Parallel Processing Symposium, April 1994, pp. 446-451
50. McNaughton, R., *Scheduling with Deadlines and Loss Functions*, Management Science, 6, 1959, pp. 1-12
51. Nanda, A. K., DeGroot, D. and Stenger, D. L., *Scheduling Directed Task Graphs on Multiprocessors Using Simulated Annealing*, Proceedings of the IEEE 12th International Conference on Distributed Computing Systems, Yokohama, Japan, IEEE Computer Society, Los Alamitos, June 1992, pp. 20-27
52. Pande, S. S., Agrawal, D. P., and Mauney, J., *A Threshold Scheduling Strategy for Sisal on Distributed Memory Machines*, Journal on Parallel and Distributed Computing, Vol. 21, No. 2, May 1994, pp. 223-236
53. Papadimitriou, C. H., and Yannakakis, M., *Scheduling Interval-Ordered Tasks*, SIAM Journal on Computing, Vol. 8, 1979, pp. 405-409
54. Pfister, G. F., *In Search of Clusters, The Ongoing Battle in Lowly Parallel Computing,* Prentice Hall PTR, 1998, ISBN 0-13-899709-8
55. Pinedo, M., *Scheduling: Theory, Algorithms, and Systems* (2nd Edition), Prentice Hall; 2 edition, 2001, ISBN 0-13-028138-7
56. Shirazi, B., Kavi, K., Hurson, A. R., and Biswas, P., *PARSA: A Parallel Program Scheduling and Assessment Environment,* in Proceedings of International Conference on Parallel Processing, ICPP 1993, August 1993, Vol. 2, pp. 68-72
57. Shirazi, B., Wang, M., *Analysis and Evaluation of Heuristic Methods for Static Task Scheduling,* Journal of Parallel and Distributed Computing Vol. 10, No. 1990, pp. 222-232
58. Soliday, S. W., Homaifar, A., Lebby, G. L., *Genetic Algorithm Approach to the Search for Golomb Rulers*, in Proceedings of the International Conference on Genetic Algorithms, Pittsburg, PA, USA, 1995, pp. 528-535
59. Tang, P., Yew P.-C., and Zhu, C.-Q., *Impact of Self-Scheduling Order on Performance of Multiprocessor Systems,* in Proceedings of the 2nd international conference on Supercomputing, June 1988, St. Malo, France, pp. 593-603
60. Ullman, J. D., *NP-Complete Scheduling Problems,* Journal of Computer and System Sciences, Vol. 10, 1975, pp. 384–393
61. Vaidya, N. H., *Another Two-Level Failure Recovery Scheme: Performance Impact of Checkpoint Placement and Checkpoint Latency*, Technical Report 94-068, Department of Computer Science, Texas A&M University, December 1994
62. Yue, M., *On the Exact Upper Bound for the Multifit Processor Scheduling Algorithm*, Annals of Operations Research, Vol. 24, No. 1, December 1990, pp. 233-259
63. Zapata, O. U. P., Alvarez, P. M., *EDF and RM Multiprocessor Scheduling Algorithms: Survey and Performance Evaluation*, Report No. CINVESTAV-CS-RTG-02. CINVESTAV-IPN, Computer Science Department, CINVESTAV-IPN, Mexico

Paper I

Paper I

Comparing the Optimal Performance of Parallel Architectures

Kamilla Klonowska, Lars Lundberg, Håkan Lennerstad and Magnus Broberg
Proceedings of The Computer Journal, Vol. 47, No. 5, 2004

Abstract

Consider a parallel program with n processes and a synchronization granularity z. Consider also two parallel architectures: an SMP with q processors and run-time reallocation of processes to processors, and a distributed system (or cluster) with k processors and no run-time reallocation. There is an inter-processor communication delay of t time units for the system with no run-time reallocation. In this paper we define a function $H(n, k, q, t, z)$ *such that the minimum completion time for all programs with n processes and a granularity z is at most* $H(n, k, q, t, z)$ *times longer using the system with no reallocation and k processors compared to using the system with q processors and run-time reallocation. We assume optimal allocation and scheduling of processes to processors. The function* $H(n, k, q, t, z)$ *is optimal in the sense that there is at least one program, with n processes and a granularity z, such that the ratio is exactly* $H(n, k, q, t, z)$*. We also validate our results using measurements on distributed and multiprocessor Sun/Solaris environments. The function* $H(n, k, q, t, z)$ *provides important insights regarding the performance implications of the fundamental design decision of whether to allow run-time reallocation of processes or not. These insights can be used when doing the proper cost/benefit trade-offs when designing parallel execution platforms.*

1 Introduction

There are many ways to implement computer systems with multiple processors, e.g. a loosely coupled distributed system consisting of a number of stand-alone computers [1,2], systems with distributed shared memory [3,4,5], or a tightly coupled SMP. In some cases, the multiprocessor is shared by many independent programs. However, one is often interested in improving the performance (i.e. reducing the completion time) of *one* parallel program consisting of a number of synchronizing processes (or threads). When designing multiprocessor systems that support efficient execution of one parallel program, one is faced with the fundamental design decision of whether to allow run-time reallocation of a process from one processor to another or if a process should execute on the same processor during the entire execution. Systems that do not allow run-time reallocation are generally less complex and thus less costly to build. The capacity, and thus the cost, of the communication network is also generally higher in systems that support run-time process reallocation. Allowing run-time process reallocation in multiprocessors with a very large number of processors would be very difficult and costly. The obvious disadvantage of not allowing run-time reallocation is that the load may become unbalanced, since some processors may be very busy while others are idle.

There is consequently no single answer to the question if run-time reallocation should be allowed or not; it is an engineering decision where a lot of factors have to be taken into consideration. Allowing run-time reallocation or not is not directly connected to building a tightly coupled SMP or a distributed system, since run-time reallocation is possible (but more costly) on distributed system and in some SMPs we want to limit, or even eliminate, run-time reallocation in order to improve the hit ratio in the processor caches. One very important factor when considering to allow run-time reallocation or not is the performance gain of allowing run-time reallocation, and thus making the load more balanced. It is obvious that the performance gain of allowing run-time reallocation depends on the quality of the scheduling and allocation policies. Obviously, systems that do not allow run-time reallocation and use a poor scheduling and allocation policy will end up with very poor performance. Finding optimal multiprocessor scheduling and allocation policies is an NP-hard problem [6]. There are, however, a number of good heuristic methods [7,8], and it is thus possible to come close or even very close to the optimal result for many important cases.

One important question is thus how much performance one will gain by allowing run-time reallocation provided that we are able to find (almost) optimal scheduling and allocation algorithms. It is, however, obvious that this gain will depend on the parallel program that we are interested in. For instance, if we have a multiprocessor with k processors there will be no performance gain of allowing run-time reallocation for a program with $4k$ parallel processes that all do the same amount of work (we will place four processes on each processor and end up with a perfectly balanced load). For some other programs, there will, however, be a performance gain by allowing run-time reallocation, even if we use the best possible scheduling and allocation algorithms. The relevant question would thus be how much one can gain at most (for any program) by

allowing run-time reallocation, provided that we use (almost) optimal scheduling and allocation. The answer to this fundamental question would provide important input when we want to balance the additional cost and complexity of allowing run-time reallocation against the performance gain of allowing run-time reallocation. In this paper we define a function that answers this question for a very wide range of multiprocessors and parallel programs.

The paper is organized in the following way. Section 2 presents related previous results. Section 3 describes definitions of dynamic and static allocation and the differences between them. In Sections 4 - 7 we make transformations of the program, and show that the transformations give a worst case program. That is, the program for which the performance gain of using SMPs compared to distributed systems is maximal. The optimal allocation of the processes in the worst-case program and the main result of the paper are presented in Section 7. Section 8 validates the results using distributed and multiprocessor hardware, and Section 9 discusses some practical applications of the result. Finally, in Section 10 we present our conclusions.

2 Previous results

The present report extends the results in [9] and [10]. In those, and all previous, reports the synchronization cost was neglected. Most distributed systems (and other systems that do not allow run-time process reallocation) have a significant cost for inter processor synchronization. By neglecting this cost in the system that does not allow run-time reallocation of processes, we obtained a too short completion time for that architecture. This means that the bound on the maximum gain of using an SMP with run-time reallocation compared to a distributed system (or cluster) with no run-time reallocation may become too low, i.e. the bound in [9] and [10] is actually not valid for most distributed systems. The error caused by neglecting the synchronization cost depends on the synchronization frequency (granularity) of the parallel program, i.e. in order not to neglect the synchronization cost, also the granularity of programs need to be taken into account. The formulas in Section 7 (in the present paper) quantify the error of neglecting the synchronization cost as a function of the real synchronization cost in the distributed system and the granularity of the program. Figure 17 (in the present paper) shows this graphically; based on the results in [9] and [10] we were only able to produce the top graph in Figure 17.

Considering the synchronization cost and program granularity complicates the mathematical problem significantly. Optimal programs can still be characterized, but in this scenario we cannot characterize optimal allocations (which was possible in [9] and [10]). As a result of this, most of the mathematical proof techniques leading to the result in this paper are new. Only the proofs in Section 5 could be reused from previous work, whereas the proofs in sections 4, 6 and 7 are new and original.

In [9] the optimal upper bound on the gain of using distributed systems instead of an SMP concerns the case where the number of computers in the distributed system equals the number of processors in the SMP. The scenario also represents static versus dynamic allocation on the same multiprocessor. In [10] this is extended for cluster

allocation, different number of processors. Also, optimal bounds for programs independent of the number of processes are derived.

In [11], extra information is provided by a test execution of the program. In a situation with this information, the general bounds are not tight [10]. In [11], tight bounds are established.

The basic method presented here has proved useful in other contexts. In [12], the efficiency of cache memories for single processors was studied; tight bounds comparing more flexible with less flexible cache memory organization alternatives were derived. The final part of the argument used in the cache study is similar to that of the present report, while a different set of transformations and arguments is needed to reach the corresponding matrix problem.

In [10,12,13,14] different applications are reported where the mathematical/combinatorial part is strongly related to the corresponding part of this report. The reports [15,16,17] are survey articles.

In previous reports, programs with any synchronization are allowed. However, the cost of synchronization signals is neglected. In the present report we assume a uniform cost t for each synchronization signal. The previous cases are represented by the case $t = 0$.

In [18] an optimal bound is described for general synchronization which includes reallocation costs. That paper concerns parallel programs with a given parallel profile.

Other than the reports [18] and [9,10,11], there are very few general results concerning allocation strategies of parallel programs with arbitrary synchronization. In the paper [19] by R. L. Graham the cost for process reallocation and synchronization is neglected. Here so called list scheduling algorithms are considered. This term is used for dynamic allocation algorithms where, when a processor becomes idle and there are waiting executable processes, one of the executable processes is immediately allocated to the idle processor. It is established in [19] that the makespan for a program allocated with a list scheduling algorithm is never higher than two times the makespan with optimal dynamic allocation.

3 Definitions and main results

3.1 Notation

In the paper we use the following quantities and notations. All will be defined later in this section.

n: the number of processes in the parallel program P ; k: the number of computers in the distributed system, i.e the system without run-time reallocation of processes (Fig. 1a); q: the number of processors in the SMP, i.e. the system with run-time reallocation of processes (Fig. 1b); t: the communication cost between computers in the distributed system (Fig. 2b); z: the granularity of the parallel program P ; $(a_1, \ldots, a_k)$: an allocation sequence, the number of processes allocated to computer x is a_x;

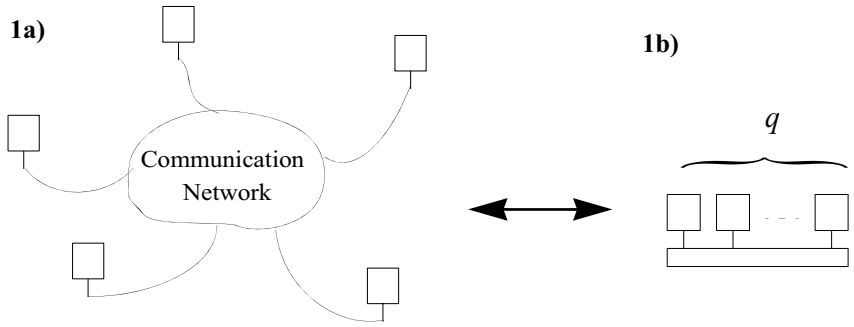


Fig. 1. a) Distributed system - a multiprocessor with *k* computers each containing *one* processor; b) SMP - a multiprocessor with *one* computer containing *q* processors

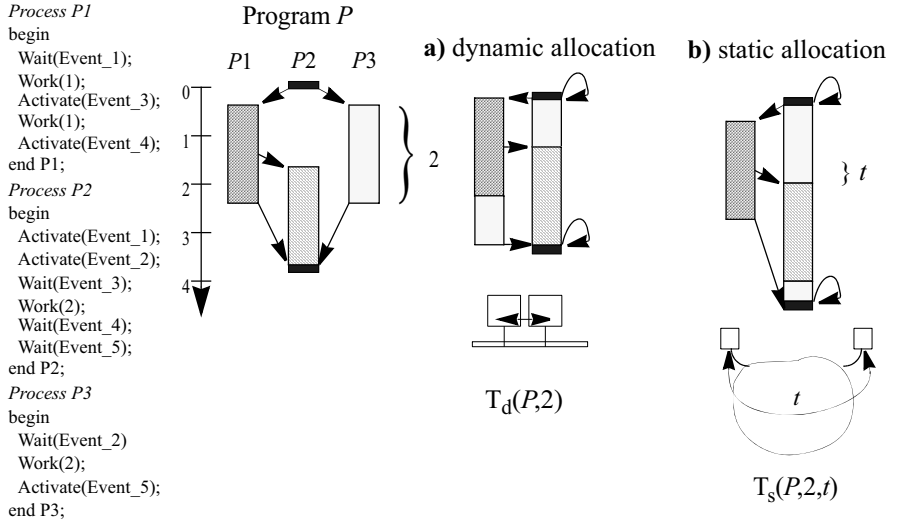


Fig. 2. An optimal dynamic (a) and static (b) allocation of a program *P* executed by two processors

$(y_1, \ldots, y_{k(k-1)/2})$: a synchronization sequence, where y_i is the number of synchronizations between pairs of processes; $T_d(P, q)$: the completion time for the program P with optimal dynamic allocation (Fig. 2a); $T_s(P, k, t)$: the completion time for the program P with optimal static allocation (Fig. 2b); $T_s(P, A, k, t)$: $T_s(P, A)$: the completion time for the program P with a certain static allocation A (Fig. 2b); $g(A, n, k, q)$: the function bounding the completion time of the execution part of a program; $r(A, n, k, t)$: the function bounding the completion time of the synchronization part of a program; $H(n, k, q, t, z)$: the optimal function bounding the total completion time of a program, i.e. the main result in this paper.

3.2 Definitions

This section describes the definitions and shows the differences between dynamic and static allocation.

We consider a parallel program with n processes. Fig. 2 shows a parallel program P consisting of three processes $P1$, $P2$ and $P3$ that are executed by two processors. The execution time of each process equals two. Sequential processing is represented by a procedure Work. Hence, Work(x) denotes sequential processing for x time units. For an uninterrupted piece of execution of a process which does not contain any synchronization we sometimes use the term segment. Consequently, a segment of length x is represented by Work(x).

Processes synchronize with two primitives: Activate and Wait. A process does not become blocked when it executes an Activate; when it executes a Wait, the process becomes blocked until the corresponding Activate has been executed. If the corresponding Activate has been executed before the process reaches the Wait, then the process executing the Wait does not become blocked. In Fig. 2 we see that process $P1$ cannot start its execution before $P2$ has started and activated $P1$ (Activate(Event_1) in $P2$). Then process $P2$ waits for the signal from $P1$ to start its execution. This dependency is represented with the Wait(Event_3) in process $P2$ and the Activate(Event_3) in process $P1$, sometimes called a synchronization signal. A synchronization signal is thus a command in a process which activates another process.

A parallel program has a well-defined start point in terms of a (possible infinitely small) piece of code in one process that must be executed before any other code in any other process is executed. There is also a well-defined end point in terms of a (possible infinitely small) piece of code in the same process that must be executed after all other processes have completed their execution. In Fig. 2, process $P2$ is the process that contains the (infinitely small) piece of code that is executed before any other code is executed, and the (infinitely small) piece of code that is executed after all other code has been executed. This is a very common structure in real parallel programs, where a main process is responsible for starting up the execution and for making sure that all processes are finished before the program terminates.

Dynamic allocation means that a process may be executed by different processors during different time periods. The processes may be transferred between all processors without limitations. An optimal dynamic allocation is an allocation of processes to processors for which the completion time of the parallel program is shorter than or equal to the completion time using any other dynamic allocation. The completion time for an optimal dynamic allocation for program P with a (SMP) multiprocessor with q processors is denoted by $T_d(P, q)$. This quantity includes only execution since in the dynamic case the cost of synchronization signals is zero.

In static allocation a process can only be executed by the processor on which it was created. In this report, contrary to the previous cases, we do not neglect the time for the synchronization in the static case. We denote the communication cost for any synchro-

nization signal by *t*. For static allocation, the completion time is affected by local scheduling. By local scheduling we mean the way processes which are allocated to the same processor are scheduled within their own processor. For a certain static allocation, optimal local scheduling is the local scheduling policy that yields the shortest completion time. An optimal static allocation is an allocation for which the completion time using optimal local scheduling is shorter than or equal to the completion time of any other static allocation, using optimal local scheduling. We denote the completion time with optimal static allocation for the program P, using a distributed system with k computers, and a synchronization cost t, by $T_s(P, k, t)$. This quantity includes both execution and synchronization times. Execution time will often be referred to as work time.

The right-hand side of Fig. 2 shows a graphical representation of P and its optimal dynamic (Fig. 2a; a multiprocessor with one computer containing two processors) and static allocations (Fig. 2b; a multiprocessor with two computers containing one processor each).

In dynamic allocation (Fig. 2a), processes may be transferred between all processors without limitations, e.g. the process $P3$ is executed on two processors. Here, the communication cost equals zero, because the processes execute at the same computer.

The work time of each process in the example shown in Fig 2 equals 2. By adding the work time of all processes in a program disregarding synchronization we obtain the total work time of that program. The number of synchronization signals in program P divided by the total work time for a program is called the granularity of the program, and is denoted by z. In the example of Fig. 2, the granularity z equals $\frac{5}{6}$.

3.3 Main results and outline of the paper

Given the parameters n, k, q, t and z, we will characterize a set of worst case programs called complete programs. This we do by starting with an arbitrary program, and performing successive transformations. In each transformation the program becomes "more worst case" in the sense that the ratio $T_S(P, k, t)/T_d(P, q)$ does not decrease. The final result is the set of complete programs. These programs all have the same $T_S(P, k, t)/T_d(P, q)$. Since we start with an arbitrary program, it follows that all complete programs are worst case, since they have maximal $T_S(P, k, t)/T_d(P, q)$.

An outline of this paper is presented in Fig. 3. We start (Section 4) with some transformations of one program into m copies of this program that allow us to split the program into two parts. The first part consists of all execution, and is described in Section 5. The second part, consisting of synchronization only, is described in Section 6. In Section 7 we combine the results from these parts into a whole program and present a formula for $H(A, n, k, q, t, z) = g(A, n, k, q) + zr(A, n, k, t)$. The first term in this

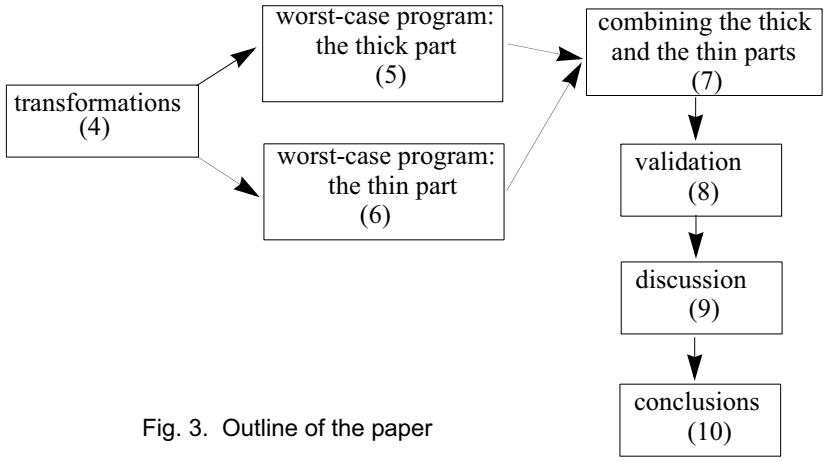


Fig. 3. Outline of the paper

sum comes from Section 5, and the second from Section 6. The formula $H(n, k, q, t, z) = min_A H(A, n, k, q, t, z)$ bounds the completion time for any program P with n processes, granularity z and communication cost t by the following inequality: $T_s(P, k, t) \leq H(n, k, q, t, z) T_d(P, q)$.

Here $T_s(P, k, t) = \min_A T_s(P, k, t, A)$. The function $H(n, k, q, t, z)$ is optimal in the sense that for at least some program P: $T_s(P, k, t) = H(n, k, q, t, z) T_d(P, q)$.

Consequently, for all programs P: $H(n, k, q, t, z) = sup_P \frac{T_s(P, k, t)}{T_d(P, q)}$.

In Section 7 we give a branch-and-bound algorithm to efficiently calculate $H(n, k, q, t, z)$ from $H(A, n, k, q, t, z)$.

4 Transforming program *P* into a new program with a thick and a thin part

In this section we present the techniques that allow us to transform a program P into a new program consisting of two parts, one with all execution (also called the thick part), and the other part with synchronizations only (also called the thin part). In this section we present two lemmas and three theorems, that will prove that it is valid to make these transformations.

4.1 Program *P'* as *m* identical copies of program *P*

In this transformation we construct a program P' by creating m copies of program P. All execution in copy x must be completed before copy $x + 1$ can start. This requires no extra synchronization, since it is obtained by concatenating the processes

with the well defined start- and end-points with each other (see Fig. 4). In Lemma 1, we show that this transformation does not change the ratio of the completion times with optimal static allocation compared to optimal dynamic allocation.

Lemma 1: *The ratio of completion times with optimal static allocation compared to optimal dynamic allocation does not change when we switch from* P *to* P'.

Proof: Having m ($m > 1$) copies of program P, we multiply both quantities $T_s(P, k, t)$ and $T_d(P, q)$ by m:

$$\frac{T_s(P, k, t)}{T_d(P, q)} = \frac{m \cdot T_s(P, k, t)}{m \cdot T_d(P, q)} = \frac{T_s(P', k, t)}{T_d(P', q)}.$$

■

Fig. 4 shows the transformation of the program P into m copies of this program, denote as P'. Program P (left part in the figure) consists of execution and synchronization signals.

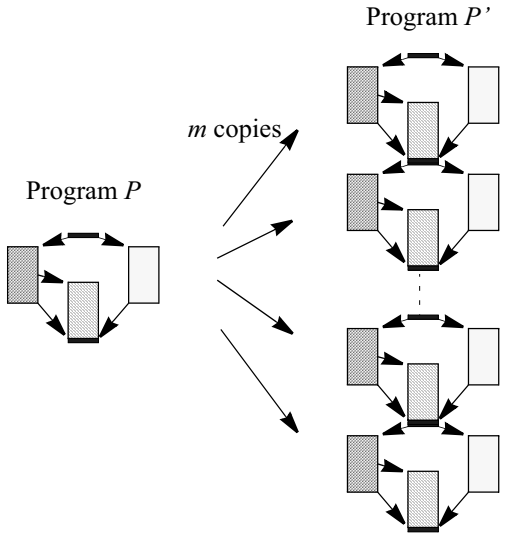


Fig. 4. Transformation P into m copies of P

We will later transform these m copies in two parts: one part with synchronization signals only and one with all execution. Before that, we present another transformation - the prolongation transformation, where we prolong the processes.

4.2 Prolongation of the processes

This transformation is essential for the simplification into the set of worst case programs which we call complete programs. We will see that a convexity property of the transformation (Theorem 1:) plays a central role.

We transform the program P into P' by prolongation of the processes. That is, we prolong each time unit x, $x > 0$ of each process in program P by Δx so that the ratio $\Delta(x/x)$ is the same for all processes. Program P' is then transformed into P'' in the same way. That is, after the transformation each time unit $x + \Delta x$ is prolonged with Δx. Each Work(x) is replaced by Work$(x + \Delta x)$ for P' and by Work$(x + 2\Delta x)$ for P'', where $\Delta(x/x)$ is constant.

This transformation does not affect synchronization.

In the case of dynamic allocation, when the cost of synchronization equals zero, after prolongation we simply have:

$$T_d(P'', q) - T_d(P', q) = T_d(P', q) - T_d(P, q) = \frac{\Delta x}{x} T_d(P, q).$$

The situation for static allocation is different. Since the communication cost in this case is not zero, the differences after prolongation are not always preserved, because we do not prolong the synchronization signals themselves.

Fig. 5 demonstrates the transformation. For the simplicity of notation we denote in this section the static completion time by $T_s(P, A)$ (instead of $T_s(P, A, k, t)$), and denote the length of the program by $L = min_A T_s(P, A)$. The difference between P and P' we denote as ΔL. That means that the length of the program P' equals $L' = L + \Delta L$. Program P'' is created in the same way using the same Δx as in P'. The difference between P' and P'' we denote as $\Delta L'$. Then the length of the program P'' will be $L'' = (L + \Delta L) + \Delta L'$.

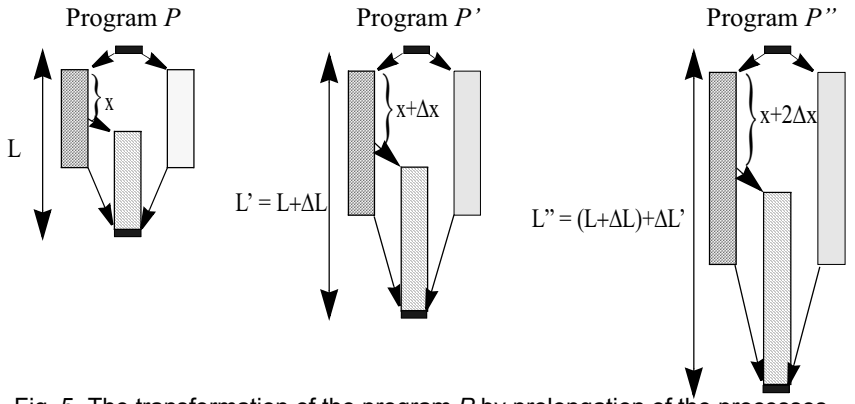


Fig. 5. The transformation of the program P by prolongation of the processes

The convexity property which is formulated in Theorem 2 follows from Theorem 1, and states that $\Delta L \leq \Delta L'$. These theorems are defined later.

In order to discuss the effect of local scheduling separately, we will assume that there is only one process per processor. We will relax this restriction at the end of this section.

First we define some terminology:

By a path of a program we define a connected sequence of segments and synchronization signals. A path starts always at the beginning of the program. By the length of a path we define the time for all execution and synchronization signals that are included in the path. By the longest path we define a path with maximum length. By the critical path we define a longest path with minimum number of synchronization signals. Thus, when we have two longest paths, then the critical path will be the path with fewer arrows. For simplicity we use the term "arrow" as a shorter term for synchronization signal.

Fig. 6 shows a program with four processes on four computers containing four synchronization signals. The right part of the figure shows an outline of two paths. The left path consists of three segments and two arrows, while the right one, that is longer, consists of four segments and three arrows. The longer one is of course the critical path.

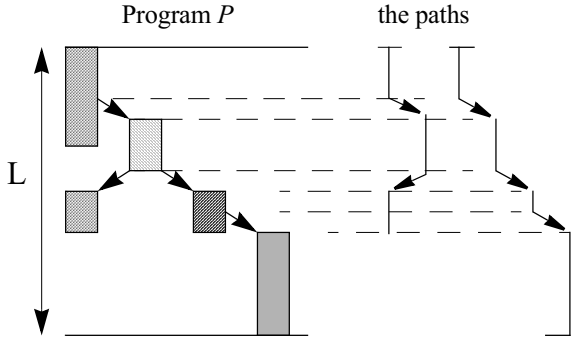


Fig. 6. : The graphical representation of the program *P* and its paths; the critical path is the longer one; arr(*P*) = 3

It is possible that the critical path has no arrows. In that case, the critical path consists of only one process. It is also possible that the critical path changes its way when we go from P to P' or from P' to P'', and in consequence the number of arrows may also change. As discussed earlier we always assume optimal local scheduling.

First we show that for a prolongation, the number of arrows in the critical path cannot increase.

Let arr(P) be the number of arrows in a critical path of the program P, and let arr(P') be the number of arrows in a critical path of the program P' (i.e. P after prolongation).

Lemma 2: $\text{arr}(P) \geq \text{arr}(P')$.

Proof: Suppose that $\text{arr}(P) = m, (m \geq 0)$ and that in the program P there is another path, that consists of more than *m* arrows. When we prolong the processes, the path with more arrows necessarily have less execution. Then this path increases slower than the critical path. Consequently, a path with more arrows cannot be critical for P'. Hence: $\text{arr}(P) \geq \text{arr}(P')$.

■

Fig. 7 shows an outline of the transformation with static allocation. The program P consists of two paths: the first with two segments and one synchronization signal, and the second path with three segments and two synchronization signals. The second path is longer and it is a critical path. During the transformation the first path grows faster and in consequence it is the critical path in program P''. Observe, that in program P' both paths are equal. In this case the first path with less synchronization signals is the critical path. The figure presents also: $\text{arr}(P) \geq \text{arr}(P')$ and $\Delta L \leq \Delta L'$.

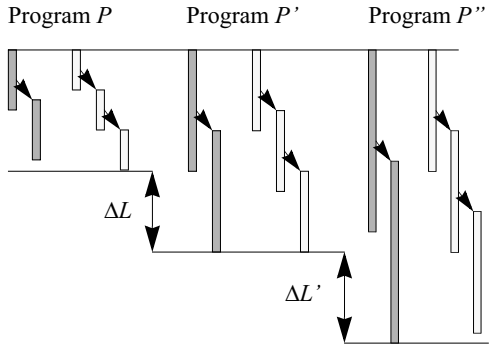


Fig. 7. The prolongation transformation

Theorem 1: $\Delta L \leq \Delta L'$.

Proof: Let $E_1 = y_1 + \text{arr}(E_1) \cdot t$ be the length of path one, where y_1 is the sum of the lengths of the segments (execution) in path one, and $\text{arr}(E_1)$ is the number of arrows where each has communication cost *t*. Let $E_2 = y_2 + arr(E_2) \cdot t$ be the corresponding length for path two. Furthermore, assume that $y_1 > y_2$ and that path two is the critical path, i.e. $E_1 < E_2$.

Let $E_1' = y_1(x + \Delta x)/x + \text{arr}(E_1)t$ and $E_2' = y_2(x + \Delta x)/x + \text{arr}(E_2)t$ be the paths after first prolongation. And let then $E_1'' = y_1(x + 2\Delta x)/x + \text{arr}(E_1)t$ and $E_2' = y_2(x + 2\Delta x)/x + \text{arr}(E_2)t$ be the paths after second prolongation.

We next compare the critical path with any other path during the prolongation. There are three possible alternatives:

(i) The critical path does not change its trajectory, i.e. $E_1 < E_2$, $E_1' < E_2'$ and $E_1'' < E_2''$. Then:

$$\Delta L = E_2' - E_2$$

$$\Delta L' = E_2'' - E_2' = E_2' - E_2 = \Delta L.$$

(ii) The critical path does change its way after first prolongation, i.e. $E_1 < E_2$, $E_1' \geq E_2'$ and $E_1'' > E_2''$. Then $\Delta L < \Delta L'$ because path E_1 grows faster than E_2.

(iii) The critical path does change its way after second prolongation, i.e. $E_1 < E_2$, $E_1' < E_2'$ and $E_1'' > E_2''$. Then, of course, $\Delta L < \Delta L'$ because path E_1 grows faster than E_2.

Consequently, in all cases we have: $\Delta L \leq \Delta L'$.

■

We now take two copies of the program P'. From the previous section we know that the transformation to *m* copies of P guarantees the ratio. According to the prolongation transformation and Theorem 1 we have:

Theorem 2: $2L' \leq L + L''$.

Proof: $2L' = L + \Delta L + L + \Delta L \leq L + \Delta L + L + \Delta L' = L + (L + \Delta L + \Delta L') = L + L''.$

■

This means that the length of two copies of P' (after transformation P) is less than or equal to the sum of the lengths of P and P''. This is a convexity property of the prolongation transformation.

4.3 From four copies into three new programs

First we will discuss the local scheduling problem.

Let programs P, P' and P'' be identical programs, except that the length of each Work in P'' is twice the corresponding Work in P', and the length of each Work in P is zero. Consider an execution of P'' where the allocation A is an optimal local schedule.

Let Q'' be a program where we have merged all processes executing on the same processor into one process. Fig. 8 shows how processes $P2''$ and $P3''$ are merged into process $Q2''$. Let Q' be the program which is identical to Q'' with the exception that the length of each Work time is divided by two. Let Q be the program which is identical to Q'' and Q' except that all Work times are zero.

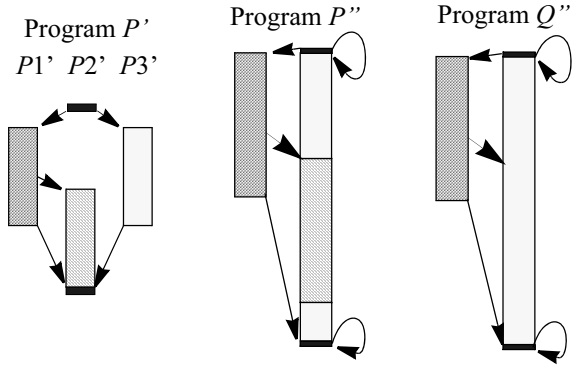


Fig. 8. Transforming *P"* into *Q"*

From Theorem 2, we know that $2T_s(Q', A) \leq T_s(Q, A) + T_s(Q'', A)$. We use the same allocation A for both P'' and Q''. However, since there are less processes in Q'' we ignore the allocation of non-existing processes in Q''. This means that each process in Q'' is allocated to a processor of its own. From the definition of Q'' we know that $T_s(P'', A) = T_s(Q'', A)$. Since the optimal order in which the processes allocated to the same processor (i.e. the optimal local schedule) may not be the same for P' and P'' we know that $T_s(P', A) \leq T_s(Q', A)$.

Consider now a program R, such that the number of processes in R is equal to the number of processes in P, and such that R also has zero Work (i.e. R contains only synchronizations, just like P). The number of synchronizations between processes R_i and R_j in program R is twice the number of synchronizations between processes P_i and P_j in program P. This means that there is always an even number of synchronizations between any pair of processes in R. All synchronizations in R must be executed in sequence (see Fig. 9). We know that it is always possible to form such a sequence, since there is an even number of synchronizations between any pair of processes.

Sequential execution of synchronizations obviously represents the worst case, and local scheduling does not affect the execution time of a sequential program. We thus know that $2T_s(P, A) \leq T_s(R, A)$ and $2T_s(Q, A) \leq T_s(R, A)$.

Consequently:

$$4T_s(P', A) \leq 4T_s(Q', A) \leq 2T_s(Q, A) + 2T_s(Q'', A) \leq T_s(R, A) + 2T_s(P'', A).$$

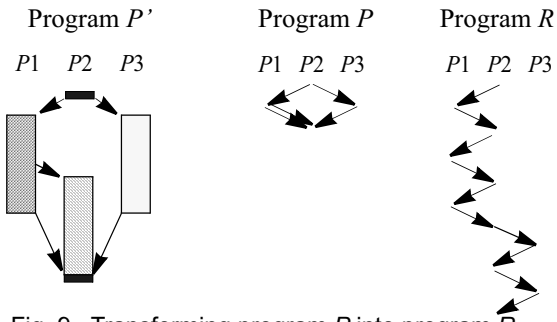


Fig. 9. Transforming program *P* into program *R*

We have proved the following:

Theorem 3: For any allocation A :

$$4T_s(P', A) \le T_s(R, A) + 2T_s(P'', A).$$

This means that the length of four copies of P' using allocation A is less than or equal to the length of the sum of program R and two copies of P'' using the same allocation A .

The prolongation transformation changes the work time but not the number of synchronizations. Hence the prolongation transformation itself does not preserve the granularity. However, in the comparison of Theorem 3, both alternatives have the same granularity.

We will see that Theorem 3 plays an important role in the next section, where we separate the program into two parts.

4.4 Transforming program *P* into a program with a thick and a thin part

In this section we describe how to transform *m* copies of a program P' into a program with one part consisting of synchronization signals only (the thin part), and the other part consisting of all processing (the thick part).

Assume an arbitrary program P' . First we create *m* copies of P' , where $m = 2^x$, for some (large) integer $x \ge 2$ (from Lemma 1 we know that the ratio $T_s(P,k,t)/T_d(P,q)$ does not change). We combine the *m* copies in groups of four and transform each group to two programs P'' and one program R . From Theorem 3 we know that $4T_s(P', A) \le T_s(R, A) + 2T_s(P'', A)$ for any allocation A . The transformation ends up with 2^{x-1} programs P'' and 2^{x-2} programs R . Again we combine 2^{x-1} programs P'' in groups of four and use the same technique. This time the transformation ends up

with 2^{x-2} programs P''' (with twice the execution compared to P'') and 2^{x-3} programs R, which we add to the previous 2^{x-2} programs of this type. We repeat this technique until there are two very thick programs with all execution and $2^{x-2} + 2^{x-3} + \ldots + 2 + 1 = 2^{x-1} - 1$ programs R.

The part consisting of the two very thick programs, with a lot of execution and very little synchronization, is called the thick part. The $2^{x-1} - 1$ copies of R, containing (almost all of) the synchronization, is called the thin part.

Note that by selecting a large enough $m = 2^x$, we can neglect the completion time of the synchronization in the thick part. We will use this fact in the next section. The granularity z is preserved by the transformation.

The transformation for $m = 8$ is illustrated in Fig. 10. Program S'' represents a program consisting of two parts: one thin (three programs R) and one thick (two programs P'''). This transformation allows the completion time for the optimal static allocation to only increase, i.e:

$$mT_s(P', k, t) = T_s(S, k, t) \le T_s(S', k, t) \le T_s(S'', k, t) .$$

But in dynamic allocation, where the cost of synchronization equals zero, the completion time is unaffected during transformation.

Consequently, $\dfrac{mT_s(P', k, t)}{mT_d(P', q)} = \dfrac{T_s(S, k, t)}{T_d(S, q)} \le \dfrac{T_s(S', k, t)}{T_d(S', q)} \le \dfrac{T_s(S'', k, t)}{T_d(S'', q)}$.

From this point onwards, we will discuss the thin part of the program P'' (program sections with synchronization only) and the thick part of the program P'' (program section with the execution time) separately.

5 The thick part

In this section we discuss the thick part, i.e. the program sections where we may neglect the synchronization time. We may thus assume that $t = 0$ when we work with the thick part. The result of this section is a formula for a function $g(A,n,k,q)$, where A is an allocation of processes to processors. This function is one of the two key components we need in order to obtain our final goal: $H(n, k, q, t, z)$.

5.1 Transforming *P* into *Q*

Consider an arbitrary parallel program P with n processes and a multiprocessor with q processors and dynamic allocation. In this transformation we add new synchronizations to P in such a way that $T_d(P, q)$ does not increase. $T_s(P, k, t)$ may, however, increase due to the new synchronizations. The new synchronizations guarantee

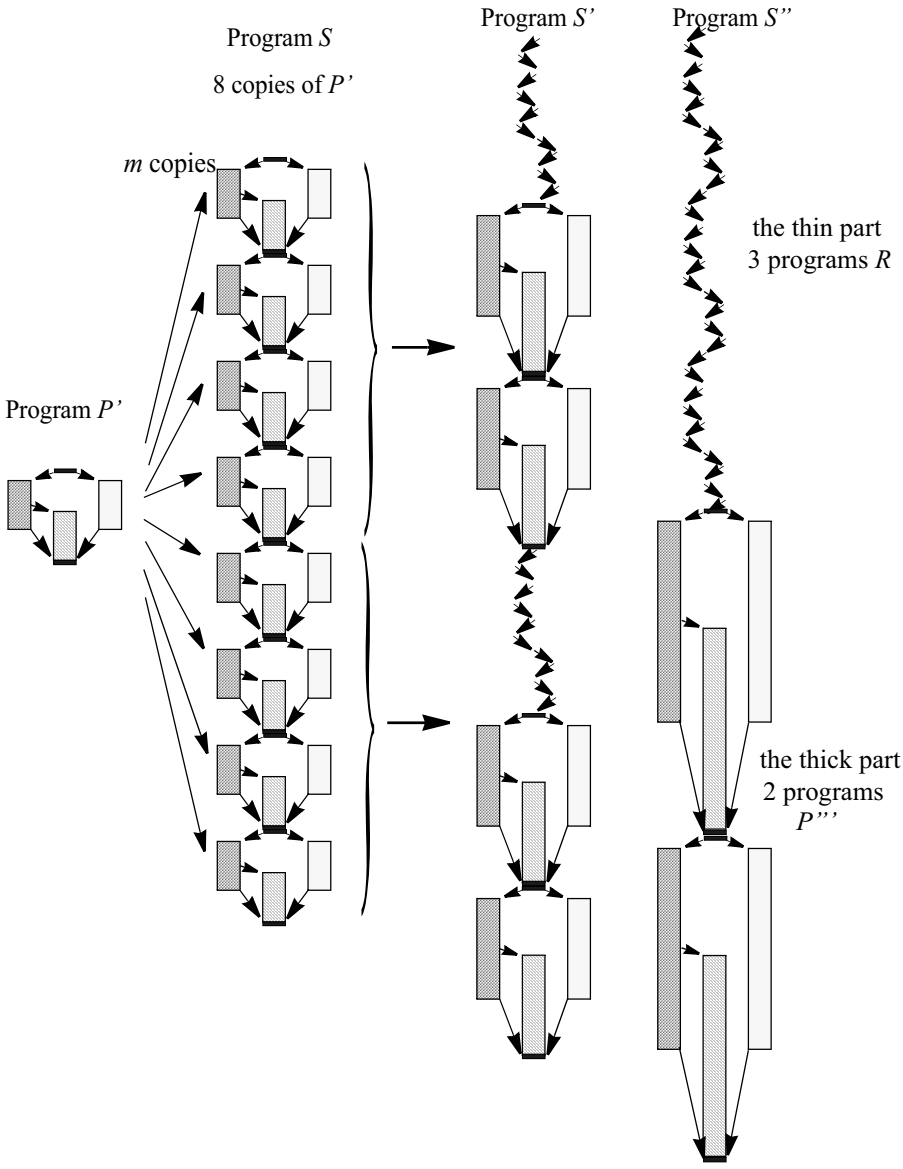


Fig. 10. The transformation of $m = 8$ copies of P' into a new program with a thin and a thick part

that, independent of how processes are allocated and scheduled, there cannot be more than q simultaneously active processes.

The execution time using optimal dynamic allocation is partitioned into m equally sized time slots in such a way that process synchronizations always occur at the end of a time slot.

In order to obtain a new program Q we add new synchronizations at the end of each time slot. These synchronizations guarantee that no processing done in slot *r,* $1 < r \leq m$, using the optimal dynamic allocation, can be done unless all processing in slot $r-1$ has been completed. Fig. 11 shows how a program P is transformed into a new program Q.

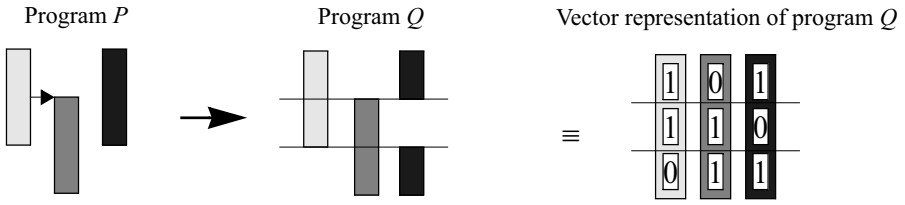


Fig. 11. The transformation of program *P* into program *Q*

The synchronizations in Q form a superset of the synchronization in P, i.e. $T_s(P, k, t) \leq T_s(Q, k, t)$. Consequently, $\frac{T_s(P, k, t)}{T_d(P, q)} \leq \frac{T_s(Q, k, t)}{T_d(Q, q)}$.

We next motivate that the bound is valid under this simplification. Denote for the sake of this argument by $T_{s,t}(Q, k, t)$ the completion time when each new synchronization has completion time *t*. By the argument in the preceding section, if we increase *x*, the set of new synchronization is not changed, but all other synchronization and execution is multiplied. Hence the relative weight of the new synchronizations decrease by *x*, so $\lim_{x \to \infty} \frac{T_{s,t}(Q_x, k, t)}{T_d(Q_x, q)} = \frac{T_s(Q, k, t)}{T_d(Q, q)}$.

By the argument of the present transformation, we have $\frac{T_s(P, k, t)}{T_d(P, q)} \leq \frac{T_{s,t}(Q_x, k, t)}{T_d(Q_x, q)}$ for all *x*. Thus, the inequality $\frac{T_s(P, k, t)}{T_d(P, q)} \leq \frac{T_s(Q, k, t)}{T_d(Q, q)}$ follows.

By this argument we may neglect the completion time for the new synchronization occurring in this transform. This allows us to import results from [10]. If the arbitrary program in the start of the transformations is a complete program, no synchronization is added. Thus, complete programs fulfill this bound.

In order to simplify the following discussion we introduce an equivalent representation of Q, called the vector representation (see Fig. 11). In this representation, each process is represented as a binary vector of length *m*, where *m* is the number of time slots in Q. This means that a parallel program is represented as *n* binary vectors. In some situations we treat these vectors as one binary $m \times n$ matrix, where each column corresponds to a vector and each row to a time slot.

From now on we assume unit time slot length, i.e. $T_d(Q, q) = m$. However, $T_s(Q, k, t) \geq m$, because if the number of active processes exceeds the number of processors on some processor during some time slot, the execution of that time slot will take more than one time unit.

We handle now programs as a binary matrix. Each column in the matrix represents one process, and the rows are independent of each other.

5.2 Transforming *Q* into *Q'*

When transforming Q into a new program Q', we start by creating $n!$ copies of Q. The vectors in each copy are permuted in such a way that each copy corresponds to one of the $n!$ possible permutations of the n vectors. Vector number v $(1 \leq v \leq n)$ in copy number c $(1 \leq c \leq n!)$ is concatenated with vector number v in copy $c + 1$, thus forming a new program Q' with n vectors of length $n! \cdot m$. The execution time from slot $1 + (c - 1)m$ to cm $(1 \leq c \leq n!)$ cannot be less than $T_s(Q, k, t)$. That is, $T_s(Q', k, t)$ cannot be less than $n!T_s(Q, k, t)$. Since, $T_d(Q', q) = n!m$, we know that

$$\frac{T_s(Q, k, t)}{T_d(Q, q)} = \frac{T_s(Q, k, t)}{m} = \frac{n!T_s(Q, k, t)}{n!m} \leq \frac{T_s(Q', k, t)}{T_d(Q', q)}.$$

The n vectors in Q' can be considered as columns in a $n!m \times m$ matrix. Reordering the rows in this matrix affects neither $T_s(Q', k, t)$ nor $T_d(Q', q)$. The rows in Q' can be reordered into $\binom{n}{q}$ equally sized groups of consecutive rows, where all rows in the same group are identical. Each group corresponds to one of the possible permutations of q ones and $n - q$ zeros. Obviously, the ratio $\frac{T_s(Q', k, t)}{T_d(Q', q)}$ is not affected by the fact that each row is duplicated $(n!m)/\binom{n}{q}$ times. This means that program Q' can be collapsed into a program with $\binom{n}{q}$ rows, containing the possible permutations of q ones and $n - q$ zeros.

From now on we will refer to this collapsed program as Q'.

Fig. 12 shows how two complete programs ($Q1$ and $Q2$) are transformed into the program Q'. It is important to note that all complete programs Q result in the same

program Q'. Therefore, we know that $\frac{T_s(P, k, t)}{T_d(P, q)} \leq \frac{T_s(Q', k, t)}{T_d(Q', q)}$. We have now found the worst possible program Q'.

5.3 Allocation properties of the thick part

The completion time is not affected by the identity of the processes allocated to different processors, because if vector v_1 is allocated on processor A and vector v_2 is allocated on processor B, then moving v_1 to B and v_2 to A is equivalent to reordering the rows in Q'. Obviously, reordering the rows does not affect the completion time. Consequently, the completion time of Q' is affected only by the number of vectors allocated to the different processors.

Theorem 4: If there are n_1 vectors allocated on processor A and n_2 vectors on processor B, and $n_1 < n_2$, the completion time cannot increase if we move one vector from processor B to processor A.

Proof: We order the vectors in Q' in such a way that vectors 1 to n_1 are allocated on processor A and vectors $n_1 + 1$ to $n_1 + n_2$ are allocated on processor B. We are now going to prove that moving vector number $n_1 + 1$ to processor A does not increase the completion time.

If vector $n_1 + 1$ has a zero in slot r $\left(1 \leq r \leq \binom{n}{q}\right)$, the contribution to the completion time from row r will not be affected by moving vector $n_1 + 1$ from processor B to processor A. Consequently, we have only to consider rows for which the corresponding slot is one in vector $n_1 + 1$. Moreover, if the number of ones in positions 1 to n_1 is smaller than the number of ones in positions $n_1 + 1$ to $n_1 + n_2$, the contribution to the completion time from row r will not increase.

Q' contains all permutations. Consequently, for each row r such that position $n_1 + 1$ contains a one, and there are at least as many ones in positions 1 to n_1 as in positions $n_1 + 1$ to $n_1 + n_2$, there exists an (n_1,n_2)-permutation r'. An (n_1,n_2)-permutation of row is obtained by switching items i and $n_1 + n_2 + 1 - i$, $(1 \leq i \leq n_1)$ (see Fig. 13).

If the completion time from row r increases from h to $h + 1$ when we move vector $n_1 + 1$ from processor B to processor A, there must be h ones in position 1 to n_1 in row r. In that case we know that symmetry of the (n_1,n_2)-permutation guarantees that there

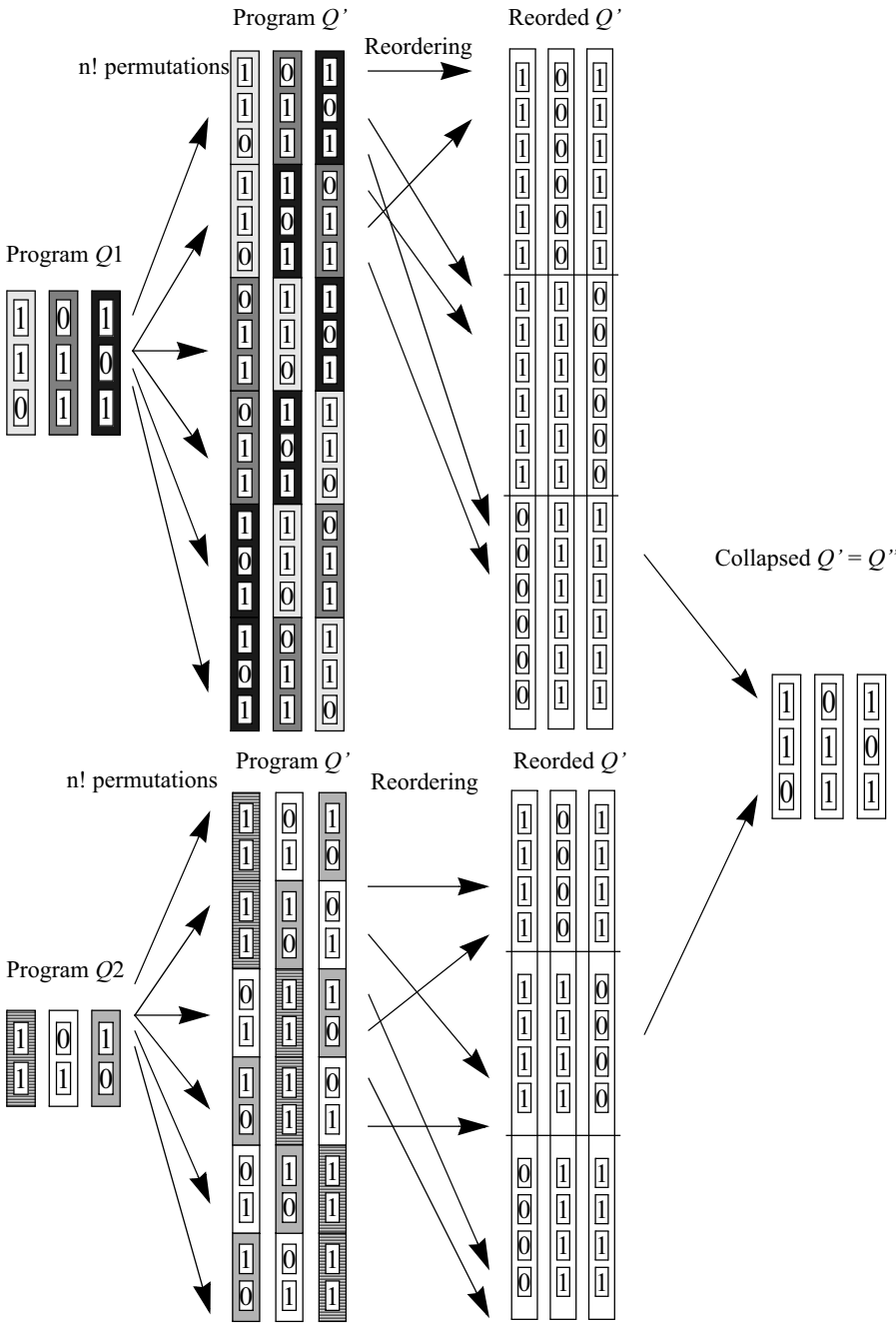


Fig. 12. The transformation of vector representation of program *Q* by permutation of processes into the same program *Q'*

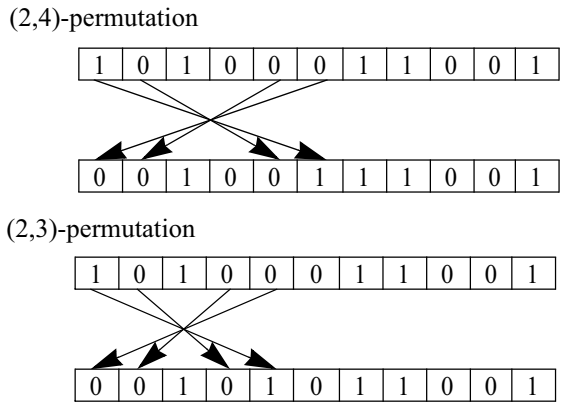


Fig. 13. One (2,4)-permutation and one (2,3)-permutation of the same row

are at least $h + 1$ ones in positions $n_1 + 1$ to $n_1 + n_2$ in row r'. Positions $n_1 + n_2 + 1$ to n are identical in r and r'. Therefore the completion time of r' will decrease with one when moving vector $n_1 + 1$ from processor B to processor A. Consequently, the completion time of Q' cannot increase if we move one vector from processor B to processor A.

■

As a consequence, an allocation of Q' results in shorter completion time the more evenly the processes are spread out on the processors. Also the identity of the processes is of no importance, and we can thus order the n vectors in some arbitrary order. In fact, the minimal completion time is obtained by allocating vector number $c + ik$ to processor c $(1 \le c \le k, 0 \le i \le \lfloor n/k \rfloor)$.

5.4 Calculating the thick part

The most obvious way to calculate the thick part is to generate the $\binom{n}{q} \times n$ matrix containing all the possible permutations of q ones and $n - q$ zeros, and explicitly calculate the completion time for this matrix using allocation described in the previous section. This turns out to be extremely laborious and inefficient, thus making it virtually impossible to calculate the thick part for reasonably large configurations. However, we can use the information that complete programs are worst case to express this calculation in a way which is essentially more efficient (see the following equation).

Equation 1. Consider parallel programs P with n processes, k computers in the distributed system, q processors in the SMP, and an allocation A where a_j processes are allocated to the j:th computer. Hence, the allocation is given by the allocation sequence $(a_1, \ldots, a_k)$, where $\sum_{j=1}^{k} a_j = n$. Then:

$$g(A, n, k, q) = \left(1 / \binom{n}{q}\right) \sum_{I} \max(i_1, \ldots, i_k) \binom{a_1}{i_1} \cdot \ldots \cdot \binom{a_k}{i_k}.$$

Here the sum is taken over all decreasing sequences $I = \{i_1, \ldots, i_k\}$ of nonnegative integers such that $\sum_{j=1}^{k} i_j = q$.

6 The thin part

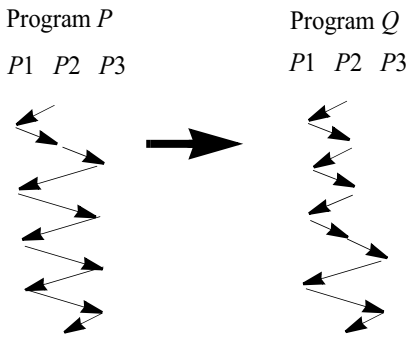
The result of this section is a formula for a function $r(A, n, k, t)$, where A is an allocation of processes to processors. This function is one of the two key components we need in order to obtain our final goal: $H(n, k, q, t, z)$.

From Section 4 we know that the thin part of a program for which the ratio $\frac{T_s(P', k, t)}{T_d(P', q)}$ is maximized consists of a sequence of synchronizations, i.e. program R. From Section 5 we know that the identity of the processes allocated to a certain processor does not affect the execution time of the thick part. For the thick part we have an optimal allocation if we manage to distribute the processes evenly among the processors. In the thin part we would like to allocate processes that communicate frequently to the same processor. These goals are, as we shall see, to a large extent contradictory.

Let $\left(y_1, \ldots, y_{\frac{k(k-1)}{2}}\right)$ be a synchronization sequence of length $\frac{k(k-1)}{2}$ for program with k processes. Entry j in this sequence indicates the number of synchronizations between a pair of processes. The optimal way of binding processes to processors under these conditions maximizes the number of synchronizations within the same processor.

We next want to find the worst case synchronization. Consider a copy of the thin part of the program where we have swapped the communication frequency (the number of synchronizations between processes) such that process j now has the same communication frequency as process i had previously and process i has the same communication frequency as process j had previously. In Fig. 14 the original communication vector for $P3$ is $(6, 2)$, meaning that there are six synchronization between $P3$ and $P1$ and two synchronizations between $P3$ and $P2$. In the copy we have

swapped the communication frequencies of $P2$ and $P3$ and we thus have six synchronizations between $P2$ and $P1$ and two synchronizations between $P2$ and $P3$.



The communication vectors:
Program P: for $P3$: (6,2) and for $P2$: (2)
Program Q: for $P3$: (2,2) and for $P2$: (6)

Fig. 14. Making a copy where we have swapped the communication frequencies of $P2$ and $P3$

It is clear that the minimum execution time of the copy is the same as the minimum execution time of the original version. The mapping of processes to processors that achieves this execution time may, however, not be the same. For instance, if we have two processors with a synchronization sequence $(2, 1)$ (i.e. two processes are allocated on one processor and one process is allocated on another processor) we obtain minimum completion time for the original version when $P1$ and $P3$ are allocated to the same processor, whereas the minimum for the copy is obtained when $P1$ and $P2$ share the same processor.

If we concatenate the original (which we call P) and the copy (which we call Q) we get a new program P' such that $T_s(P', A, k, t) \leq T_s(P, A, k, t) + T_s(Q, A, k, t)$ for any allocation A (all thin programs take zero execution time using a system with one process per processor and no communication delay). By generalizing this argument we obtain the kind of permutation as we had for the thick part (see Fig. 12).

These transformations show that the worst case for the thin part occurs when all synchronization signals are sent from all n processes $n-1$ times - each time to a different process. All possible synchronization signals for n processes equals $n(n-1)$ and all possible synchronization signals for the processes allocated to processor i equals $a_i(a_i-1)$. Because some processes are executed on the same processor, the communication cost for them equals zero. That means that the number of synchronization signals in a worst-case program (regarding communication cost) is equal to $n(n-1) - \sum_{i=1}^{k} a_i(a_i-1)$.

Consequently, the optimal time for the thin part with the communication cost t, and allocation A is:

$$r(A, n, k, t) = \frac{n(n-1) - \sum_{i=1}^{k} a_i(a_i - 1)}{n(n-1)} \cdot t \ .$$

7 Combining the thick and thin parts

Combining the thick part with the thin part is done by adding the function $g(A, n, k, q)$ and $r(A, n, k, t)$, weighted by the granularity, z. Thus, we end up with a function $H(A, n, k, q, t, z) = g(A, n, k, q) + zr(A, n, k, t)$.

Hence, in a program with high z (i.e. high synchronization frequency) the thin path has larger impact than in a program with low z.

From previous results we know that the thick part is optimal when evenly distributed over the processors, whereas the thin part is optimal when all processes reside on the same processor. The algorithm for finding the minimum allocation A is based on the knowledge about the optimal allocation for the thick and thin part respectively.

7.1 Finding the optimal allocation using allocation classes

The basic idea is to create *allocation classes*, and evaluate them. An allocation class consists of allocations and *separated allocations*. Separated allocations are defined as follows:

- A set of processors with a common allocation for both the thick and thin parts. It is described in decreasing order. This means that the number of assigned processes to processor i must be greater than or equal to the number of assigned processes to processor $i + 1$.
- In the allocation for the remaining processors for the thick part, the processes are evenly distributed. The highest number of processes assigned to a processor must be equal to or less than the last number of assigned processes for any processor in the common assignment.
- In the allocation for the remaining processors for the thin part, the processes make use of as few of the remaining processors as possible. The highest number of processes assigned to a processor must be equal or less than the least number of assigned processes for any processor in the common assignment.

Thus a separated allocation is not an allocation, i.e. it does not appear in the minimization of $H(A, n, k, q, t, z)$. It is divided in two parts: one which is common for the thick and thin part, and one where we use optimal allocations for the thick and the thin part separately. If follows that a separated allocation gives better performance than any allocation which has no part identical to the common part of the separated allocation. This observation is fundamental for the branch-and-bound algorithm.

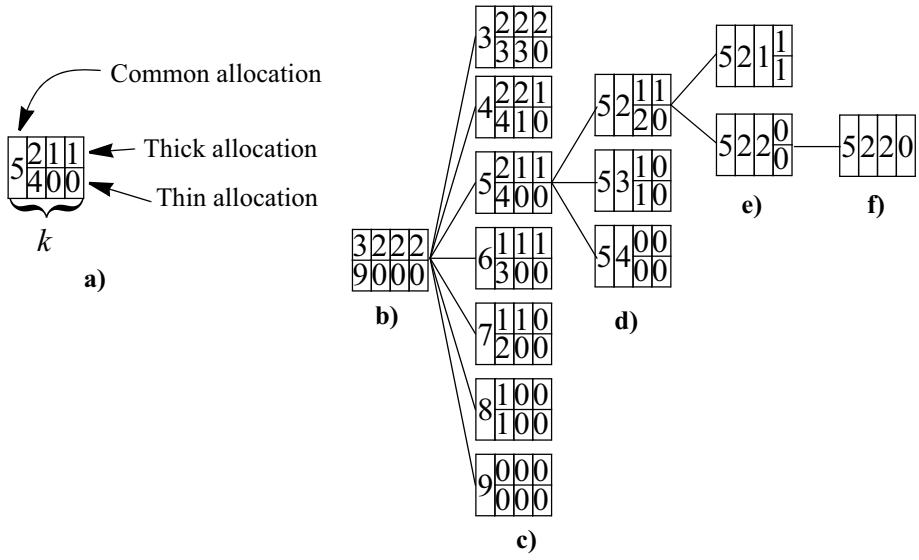


Fig. 15. Allocation classes

An allocation class consists of all allocations which are related to a separated allocation by containing its common part. Thus any separated allocation defines a specific class of allocations.

An example of a separated allocation is shown in Fig. 15. Here the common part consists of one processor, we have $n = 9$ processes and in total $k = 4$ processors. Consider for example Fig 15(a). Here the first part is the common allocation, the upper part is the thick allocation for remaining 3 processors and the lower part is the thin allocation for the remaining 3 processors. In Fig. 15(b) we have the totally separated allocation, with no common allocation at all. Fig. 15(c) shows all possible allocation classes where the size of the common part is one. The result with separated allocations is better than (or equal to) any allocation within that class. We take the minimum value of all the separated allocations in Fig. 15(c) for the over-optimal allocation, and expand the tree from this point. This procedure is described in the next section.

7.2 The branch-and-bound algorithm

The algorithm for finding an optimal allocation is a classical branch-and-bound algorithm [1]. We start by the totally separated allocation (Fig. 15(b)). We move one processor into the common part, and thus create a new subclass of separated allocations (Fig. 15(c)). We next minimize over these separated allocations. Assume that in the example of Fig. 15c the class with 5 processors in the common allocation is the minimum. The new subclass is shown in Fig. 15(d). All separated allocations give a higher (or equal) value than the separated allocation which is parent to the new class.

The classes are organized in a tree structure. The minimum is now calculated over all *leaves* in the tree, including leaves on branches which previously have been expanded. By repeatedly selecting the leaf with minimum value and creating its subclass, we will finally reach a situation where the minimum leaf no longer has any subclass. We have then found an optimal allocation. If we assume that the classes in Fig. 15(e) and (f) gives the minimum values we have reached an optimal allocation.

When calculating a class we use the common allocation concatenated with the thick allocation as input for the function $g(A_{thick})$ and $r(A_{thin}, t)$, where A_{thick} is the common allocation concatenated with the thick allocation and A_{thin} is the common allocation concatenated with the thin allocation. By adding the results (weighted by z) from the two functions, we get the value of that leaf $(g(A_{thick}) + zr(A_{thin}, t))$.

Practical testing has shown that many different types of allocations may be optimal, depending on the values of *n, k, q, t* and *z.* This is in contrast with previous applications presented in [9,10,11], see Section 9.

8 Validation

The results in this paper are based on a theoretical proof leading to a program for which the ratio $T_s(P, k, t)/T_d(P, q)$ is maximal. This part of the result can not be validated since it is impossible to generate all programs and then measure their completion times and take the maximum ratio for all of them. The correctness of this part of the result is thus based on the proofs.

It is, however, possible to see if the program that we have proved to be extremal (worst-case) for the parameters n, q, and z really results in a value close to $H(n,k,q,t,z)$ when executed in a real multiprocessor environment. We have done a number of such tests on multiprocessor and distributed Sun/Solaris environments. We first considered fat programs defined by three parameters: the number of threads (n), the number of active threads in each time slot (q) and the amount of work performed in each time slot (w); w denotes the number of iterations over a vector of length 1024 (each such iteration took approximately 0.1 ms). Using a Sun Enterprise 4000, with 8 processors, these programs are executed using two scheduling algorithms. First we use standard Solaris scheduling, where no threads are bound to processors, and then thread number i is bound to processor (i mod k). This means that if the number of processors (k) is eight and the number of threads (n) is 20, then threads 0, 8 and 16 are bound to processor zero. As discussed in Section 5, we know that this is the optimal way to bind the threads.

The completion time ratio using bound scheduling on k processors compared to unbound scheduling on q processors is denoted $h(n,k,q,w)$, for a program with n threads, q active threads in each slot, and slot length w.

We consider the cases when $q = k$, $1 \le k \le 8$, $k \le n \le 20$ and $w = 100$, 1000 and 10000 (see Fig. 16). These values of $h(n,q,k,w)$ are compared with the theoretical bound $g(n,k,q,t{=}0,z)$ in Fig. 16. The tables show that when $w = 1000$, $n = 18$ and $k = q$

= 8, the completion time using bound scheduling is 1.93 times the completion time using standard unbound Solaris scheduling. The tables also show that $h(n,k,k,w)$ never exceeds $H(n,k,k,t=0,z)$. That is, the measurements indicate that $H(n,k,k,t=0,z)$ is indeed a valid upper bound. When w increases, $h(n,k,k,w)$ comes closer to $H(n,k,k,t=0,z)$, and by selecting a large enough w we can get arbitrarily close to $H(n,k,k,t=0,z)$. Consequently, $H(n,k,k,t=0,z)$ is an optimal upper bound when w is large, i.e. $H(n,k,k,t=0,z)$ is an optimal upper bound when z is small (thus reducing the impact of the thin part).

We also did measurements for thin programs. That is, programs consisting of a chain of threads that synchronize. The test considered a case where n = 8 and the length of the synchronization chain is 100000. The chain was executed on two different environments: a Sun Enterprise 4000, with eight processors (approximated with a synchronization delay of zero in our model) and a case when eight programs are executed on eight identical computers connected with an ethernet. On the SMP we implemented the synchronizations with semaphores and in the distributed systems we sent one byte UDP messages between the computers.

These measurements shown that the ratio between the execution time in a distributed environment and the SMP execution time is 1/200, i.e. 0.5%. In our model we assume that this ratio is zero (since we assume that the SMP execution time of this part is zero). This means that our model has an error of 0.5% for the thin part of the program when comparing these two environments. Since the thin part does not take zero time to execute on an SMP, the bound is still valid but not quite optimal, i.e. there is a difference of 0.5% in this case. If the synchronization time in the distributed system increases (i.e. if t grows), the difference becomes smaller.

We can draw two conclusions from the validations:

- First, the bound seems to be valid for all programs. This means that there are no programs P for which the ratio $T_s(P, k, t)/T_d(P, q)$ exceeds the bound.
- Second, the bound is optimal for coarse grained programs. Consequently, there are coarse grained programs P (i.e. programs with small z) for which the ration $T_s(P, k, t)/T_d(P, q)$ is equal to the bound. The bound is also optimal for systems with very long synchronization latency (i.e. for large t).

Consequently, the model is correct for the fat part (since we can make this part arbitrarily fat), and the difference for the thin part is small for realistic settings.

Our validations are limited to the Sun/Solaris environment, but we believe that these two conclusions will hold for other environments as well.

9 Discussion

Fig. 17 shows three graphs corresponding to $H(n,k,k,0,10)$, $H(n,k,k,0.0025,10)$, and $H(n,k,k,0.005,10)$ for the range $1 \le n, k \le 50$, i.e. $q = k$ in the graphs. We know that only the product of t and z affects g, i.e. $H(n,k,q,t/2,z) = H(n,k,q,t,z/2)$. The program related parameters (i.e. n and z) can easily be obtained from simple profiling tools. If we have a program with 50 processes and, on average, one synchronization every 400

n

$k=q$	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
2	1.23	1.24	1.25	1.25	1.26	1.26	1.26	1.26	1.26	1.26	1.26	1.25	1.25	1.25	1.24	1.24	1.24	1.24
3	--	1.31	1.32	1.32	1.33	1.33	1.33	1.33	1.33	1.33	1.32	1.32	1.32	1.32	1.31	1.31	1.31	1.31
4	--	--	1.32	1.34	1.34	1.34	1.36	1.36	1.36	1.35	1.35	1.35	1.35	1.34	1.33	1.32	1.32	1.31
5	--	--	--	1.30	1.32	1.32	1.33	1.33	1.33	1.32	1.31	1.31	1.31	1.31	1.30	1.30	1.29	1.29
6	--	--	--	--	1.30	1.32	1.32	1.32	1.31	1.30	1.30	1.30	1.28	1.28	1.27	1.26	1.25	1.25
7	--	--	--	--	--	1.29	1.32	1.31	1.28	1.25	1.23	1.23	1.24	1.23	1.22	1.22	1.22	1.22
8	--	--	--	--	--	--	1.29	1.30	1.27	1.23	1.22	1.22	1.22	1.22	1.23	1.23	1.22	1.22

$h(n,k,q,w = 100)$

n

$k=q$	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
2	1.33	1.33	1.40	1.40	1.43	1.43	1.43	1.44	1.44	1.44	1.44	1.44	1.44	1.44	1.44	1.44	1.43	1.43
3	--	1.50	1.60	1.60	1.69	1.71	1.71	1.73	1.74	1.74	1.75	1.75	1.75	1.74	1.74	1.74	1.73	1.73
4	--	--	1.60	1.73	1.76	1.77	1.85	1.86	1.86	1.87	1.87	1.87	1.86	1.86	1.86	1.86	1.86	1.86
5	--	--	--	1.67	1.81	1.86	1.87	1.87	1.88	1.89	1.89	1.89	1.88	1.88	1.88	1.87	1.87	1.87
6	--	--	--	--	1.72	1.86	1.88	1.88	1.88	1.88	1.90	1.91	1.91	1.91	1.91	1.91	1.91	1.91
7	--	--	--	--	--	1.75	1.87	1.88	1.88	1.88	1.88	1.88	1.90	1.91	1.91	1.91	1.91	1.91
8	--	--	--	--	--	--	1.78	1.84	1.85	1.86	1.86	1.86	1.86	1.86	1.92	1.93	1.92	1.92

$h(n,k,q,w = 1000)$

n

$k=q$	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
2	1.33	1.33	1.40	1.40	1.43	1.43	1.44	1.44	1.45	1.45	1.46	1.46	1.46	1.47	1.47	1.47	1.47	1.47
3	--	1.50	1.60	1.60	1.69	1.71	1.71	1.75	1.75	1.76	1.78	1.79	1.79	1.80	1.80	1.81	1.81	1.82
4	--	--	1.60	1.73	1.77	1.77	1.86	1.89	1.90	1.91	1.93	1.95	1.97	1.97	1.98	1.99	1.99	2.00
5	--	--	--	1.67	1.81	1.86	1.87	1.87	1.94	1.98	2.00	2.00	2.01	2.03	2.04	2.05	2.07	2.07
6	--	--	--	--	1.72	1.86	1.90	1.92	1.92	1.93	2.00	2.04	2.05	2.07	2.08	2.08	2.09	2.09
7	--	--	--	--	--	1.75	1.89	1.93	1.94	1.94	1.95	1.95	2.04	2.06	2.08	2.09	2.09	2.09
8	--	--	--	--	--	--	1.78	1.91	1.94	1.95	1.96	1.96	1.97	1.97	2.06	2.08	2.10	2.10

$h(n,k,q,w = 10000)$

n

$k=q$	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
2	1.33	1.33	1.40	1.40	1.43	1.43	1.44	1.44	1.45	1.45	1.46	1.46	1.47	1.47	1.47	1.47	1.47	1.47
3	1.00	1.50	1.60	1.60	1.69	1.71	1.71	1.75	1.76	1.76	1.78	1.79	1.79	1.80	1.81	1.81	1.82	1.82
4	1.00	1.00	1.60	1.73	1.77	1.77	1.86	1.90	1.91	1.91	1.94	1.96	1.97	1.97	1.99	2.00	2.00	2.00
5	1.00	1.00	1.00	1.67	1.81	1.86	1.87	1.87	1.96	2.00	2.02	2.03	2.03	2.06	2.08	2.09	2.10	2.10
6	1.00	1.00	1.00	1.00	1.72	1.86	1.90	1.92	1.93	1.93	2.01	2.06	2.09	2.10	2.11	2.11	2.14	2.16
7	1.00	1.00	1.00	1.00	1.00	1.75	1.89	1.93	1.95	1.96	1.96	1.96	2.05	2.10	2.13	2.15	2.16	2.16
8	1.00	1.00	1.00	1.00	1.00	1.00	1.78	1.91	1.95	1.97	1.97	1.98	1.98	1.98	2.07	2.12	2.16	2.18

$H(n,k,q,t = 0, z)$, for any z (z has no impact when t is zero)

Fig. 16. Validating the fat part of the program

time units (z = 1/400 = 0.0025) and want to compare the optimal performance gain of using a multiprocessor with 10 processors, run-time reallocation of processes and virtually no communication overhead with a system with 10 processors, no reallocation and a communication delay of 10 time units, we can look at the middle graph in Fig. 17 (tz = 0.025). The value corresponding to n = 50 and $k = q$ = 10 is 2.7. This means that the gain of using the system with reallocation can be at most a factor of 2.7. It is trivial to see that the maximum gain of using the system with no reallocation is zero, i.e. the minimum ratio is one.

It is possible to compare systems with different numbers of processes, e.g. a multiprocessor with 10 processors, run-time reallocation of processes and virtually no communication overhead with a system with 20 processors, no reallocation and a communication delay of 5 time units. The maximum gain of using the smaller system with reallocation for a program with 50 processes and one synchronization every 400 time units is defined by the value $g(50,20,10,5,0.0025) = 2.03$. The maximum gain of using the system with 20 processors compared to the system with 10 processors is trivially a factor of 2. Experiments using real programs, or suites of real programs [20] cannot capture the extreme cases. Consequently, such techniques cannot answer the kind of performance comparisons discussed above.

The parameter t, makes it possible to consider the performance implications of inter-process synchronization. However, we do not consider the implications of inter-process communication. This could lead to an underestimation of the execution time for the distributed system, i.e. the system with no run-time reallocation of processes. In many applications, communication occurs primarily at the synchronization points, and in those cases one can increase the value of t to include also the (average) extra cost for inter process communication. Consequently, t would then include the synchronization overhead and the (average) communication overhead.

We have assumed the reallocation is free on SMPs, i.e. the system with reallocations. This is not completely true since reallocation will increase the number of cache misses. This effect may cause us to underestimate the execution time on the SMP, thus making the bound less tight. The evaluations the previous section support this and show that our bound is not optimally tight (but still valid) for programs with a high synchronization frequency, and thus also a high reallocation frequency.

10 Conclusions

We have presented a bound on the gain of using a system with q processors and run-time process reallocation compared to using a system with k processors, no reallocation and a communication delay t, for a program with n processes and a synchronization granularity z. This bound is denoted $H(n,k,q,t,z)$. Based on our multiprocessor model (defined by k, q and t), the bound is optimal in the sense that there is at least one program with n processes and a granularity z for which the gain is exactly $H(n,k,q,t,z)$. We have also validated our results using multiprocessor and distributed Sun/Solaris systems. The conclusions from the validations are that the bound is valid for all pro-

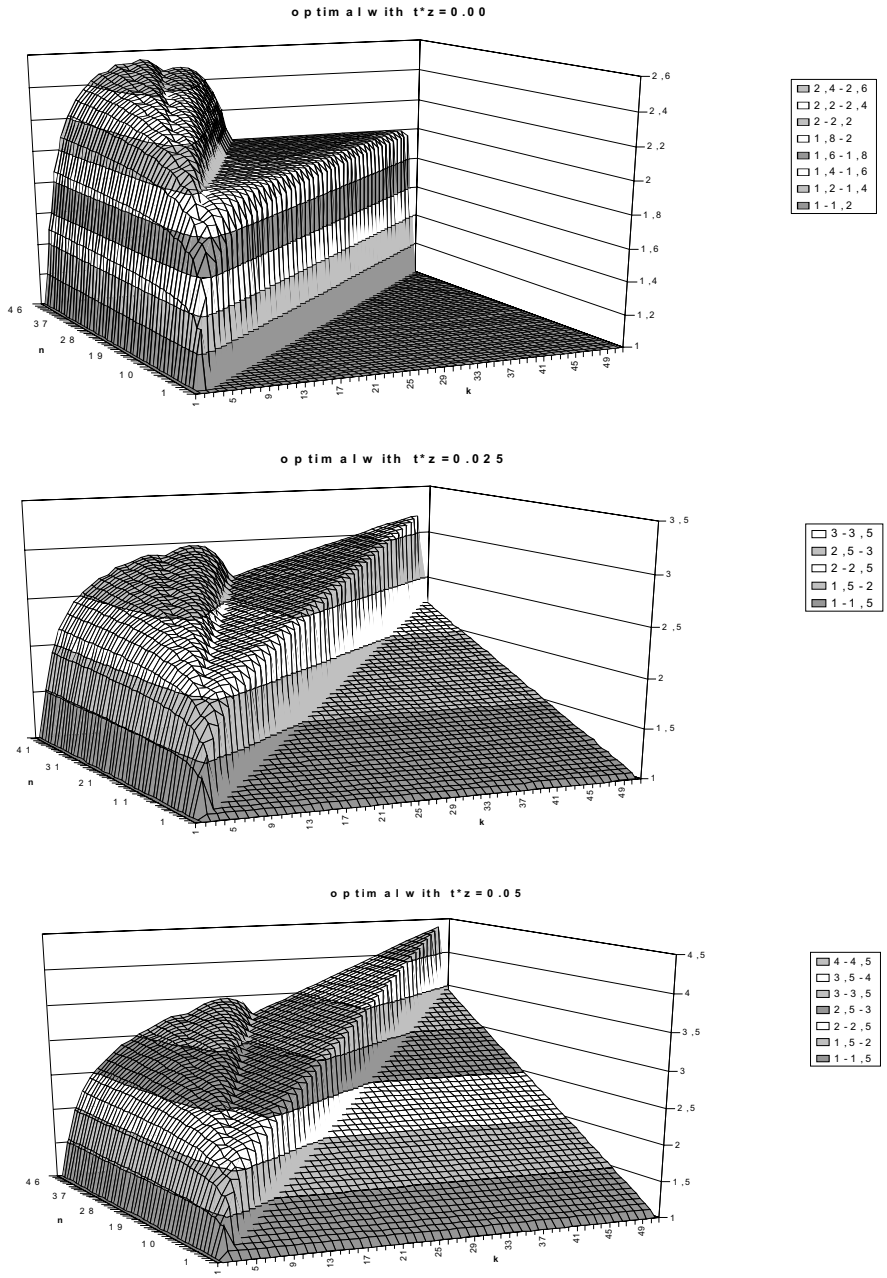


Fig. 17. The function *H(n,k,q,t,z)* for different values of *tz* and the intervals $1 \leq n, k \leq 50$, and $q = k$

grams and architectures, and that the bound is indeed optimal when t is large or z is small.

The function $H(n,k,q,t,z)$ is not trivial, whereas the maximum gain of using k processors, no reallocation and a communication delay t compared to a system with q processors and run-time reallocation is trivial (k/q). By looking at plots of $H(n,k,q,t,z)$ (e.g. plots like the ones in Fig. 17), computer system architects can obtain important insights regarding the consequences of some of their fundamental design decisions. There is a lack of techniques that provide this kind of fundamental insights in the empirical field of computer architecture. The main contribution in this paper compared to previous results is that we have been able to handle and validate a more realistic computer model, where the communication delay t and the granularity z are taken into consideration.

We have presented transformations which enable us to separate execution and synchronization. Analyzing the parts separately and comparing them, we found a formula that produces the minimal completion time. A branch-and-bound algorithm was used to find the optimal result.

The argument leads to the sum $H(A, n, k, q, t, z) = g(A, n, k, q) + zr(A, n, k, t)$. Here the thick part, represented by $g(A, n, k, q)$, corresponds to previous results. The function $g(A, n, k, q)$ is convex in the sense that the value decreases if the load of a worst case program is distributed more evenly among the computers. However, the thin part, represented by $r(A, n, k, t)$, is concave. This quantity increases if the load of a worst case program is distributed more evenly. This makes the minimization of the sum H a delicate matter. The type of allocation which is optimal depends strongly on the value of tz. If tz is small execution dominates, and partitions representing even distributions, uniform partitions, are optimal. If tz is large synchronization dominates, and partitions where all processes are allocated to the same computer are optimal. There is an intermediate domain of values of tz where there are different types of partitions which are optimal.

The worst case programs are the same in all scenarios presented in 9. to 11. including the present one: complete programs. Optimal allocations, however, differ among the applications. In the parallel processing applications, uniform allocations are optimal. In the cache memory application, the optimal allocations contains one set arbitrary size, and the other $k-1$ sets are distributed as evenly as possible. These properties depend on convexity of the load function. In the cache memory application, the load function is not convex.

References

1. Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., and Sunderam, V., *PVM: Parallel Virtual Machine*, MIT press, Boston, 1994
2. MPI Forum, *MPI: A message-passing interface standards*, International Journal of Supercomputer Application, 8 (3/4), 1994, pp. 164-416

3. Hagersten, E. and Koster, M., *WildFire: A Scalable Path for SMPs*, in Proceedings of 5th International Symposium on High Performance Computer Architecture, IEEE Computer Society, Los Alamitos, Orlando, FL, USA, 9-13 January, 1999, pp. 172-181
4. Laudon, J. and Lenoski, D., *The SGI Origin: A ccNUMA Highly Scalable Server*, in Proceedings of 24th Annual International Symposium on Computer Architecture, ACM, Denver, CO, USA, 2-4 June, 1997, pp. 241-251
5. Lovett, T. and Clapp, R., *STiNG: A cc-NUMA Computer System for the Commercial Marketplace*, in Proceedings of 24th Annual International Symposium on Computer Architecture, ACM, Philadelphia, PA, USA, 22-24 May, 1996, pp. 308-317
6. Garey, M.R. and Johnson, D.S., *Computers and Intractability - A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, New York, 1979
7. Hou, E.S.H., Hong, R., and Ansari, N., *Efficient multiprocessor scheduling based on genetic algorithms*, in Proceedings of 16th Annual Conference of the IEEE Industrial Electronics Society - IECON'90, IEEE, Vol II, Pacific Grove, CA, USA, 27-30 November, 1990, pp. 1239-1243
8. Nanda, A.K., DeGroot, D. and Stenger, D.L., *Scheduling directed task graphs on multiprocessors using simulated annealing*, in Proceedings of IEEE 12th International Conference on Distributed Computing Systems, IEEE Computer Society, Yokohama, Japan, 9-12 June, 1992, pp. 20-27
9. Lennerstad, H. and Lundberg, L., *An Optimal Execution Time Estimate of Static versus Dynamic Allocation in Multiprocessor Systems*, SIAM Journal of Computing, 24 (4), 1995, pp. 751-764
10. Lennerstad, H. and Lundberg, L., *Optimal Combinatorial Functions Comparing Multiprocess Allocation Performance in Multiprocessor Systems*, SIAM Journal of Computing, 29 (6), 2000, pp. 1816-1838
11. Lundberg, L. and Lennerstad, H., *Using Recorded Values for Bounding the Minimum Completion Time in Multiprocessors*, IEEE Transactions on Parallel and Distributed Systems, 9 (4), 1998, pp. 346-358
12. Lennerstad, H. and Lundberg, L., *Optimal worst case formulas comparing cache memory associativity*, SIAM Journal of Computing, 30 (3), 2000, pp. 872-905
13. Lundberg, L. and Lennerstad, H., *Bounding the maximum gain of changing the number of memory modules in multiprocessor computers*, Nordic Journal on Computing, 4, 1997, pp. 233-258
14. Lundberg, L. and Lennerstad, H., *Optimal Bound on the Gain of Permitting Dynamical Allocation of Communication Channels in Distributed Processing*, Acta Informatica, 36 (6), 1999, pp. 425-446
15. Lennerstad, H. and Lundberg, L., *Optimal Scheduling Results for Parallel Computing*, SIAM News, 27 (7), 1994
16. Lennerstad, H. and Lundberg, L., *Combinatorics for Multiprocessor Scheduling Optimization and Other Contexts in Computer Architecture*, Proceedings of the Conference of Combinatorics and Computer Science (LNCS 1120), Springer Verlag, Brest, France, 15-18 July, 1995, pp. 341-347

17. Lennerstad, H. and Lundberg, L., *Combinatorial Formulas for Optimal Cache Memory Efficiency*, SIAM News, 29 (6), 1996
18. Lundberg, L., Broberg, M. and Klonowska. K., *Evaluating Heuristic Scheduling Algorithms for High Performance Parallel Processing*, in Proceedings of 5th International Symposium on High Performance Computing (LNCS 2858), Springer-Verlag, Tokyo-Odaiba, Japan, 20-22 October, 2003, pp. 160-173
19. Graham, R.L., *Bounds on Multiprocessing Anomalies,* SIAM Journal of Applied Mathematics, 17 (2), 1969, pp. 416-429
20. Woo, S., Ohara, M., Torrie, E., Singh, J. and Gupta, A., *The SPLASH-2 Programs: Characterization and Methodological Considerations*, in Proceedings of 22nd International Symposium on Computer Architecture, ACM, Santa Margherita Ligure, Italy, 22-24 June, 1995, pp. 24-36

Paper II

Paper II

The Maximum Gain of Increasing the Number of Preemptions in Multiprocessor Scheduling

Kamilla Klonowska, Lars Lundberg, Håkan Lennerstad
Proceedings of The Computer Journal, Vol. 47, No. 5, 2004

Abstract

This paper generalizes the 4/3-conjecture. We consider the optimal makespan $C(P, m, i)$ *of an arbitrary set P of independent jobs scheduled with i preemptions on a multiprocessor with m processors. We optimally compare the makespan for i and j preemptions, where* $i < j$ *, in the worst case, i.e. we calculate a formula for the worst case ratio* $G(m, i, j)$ *defined as* $G(m, i, j) = max\frac{C(P, m, i)}{C(P, m, j)}$*, where the maximum is taken over all sets P of independent jobs.*

Keywords: *parallel processor scheduling, preemptive scheduling, i-preemptive scheduling, worst-case analysis, optimization, Stern-Brocot tree*

1. Introduction and related work

In order to take full advantage of the processing capacity of a multiprocessor it is important to balance the load so that it is evenly spread between the processors. In order to obtain balance load, a job on one processor may have to be preempted and then later restarted on another processor. Each preemption allows the schedule to split a job in two parts which may not run in parallel but on two different processors, this may allow the makespan to decrease. There are no restrictions on the sizes of the two parts - a preemption can be made at any point in time.

Due to cache effects and other forms of overhead, there is a cost for preemptions and then later restarting a job. Consequently, we would like to limit the number of preemptions in order not to suffer too much from such overhead costs. This means that there is a trade-off; on the one hand we need preemptions in order to obtain a balanced load, but on the other hand we would like to limit the number of preemptions in order to minimize the overhead costs. The optimal solution to this trade-off problem depends on a number of parameters, such as the cost for preempting and then later restarting a job. One crucial piece of information for making a well informed trade-off decision is how much we will gain by increasing the number of preemptions, assuming that there are no overhead costs. This will tell us how much we can gain by introducing more preemptions. The system designer can then compare that gain to the increased overhead costs.

Here we are interested in parallel programs consisting of a number of independent jobs. The program has completed its task when all jobs have terminated, and we are thus interested in minimizing the makespan of the parallel program.

It is clear that the performance gain, i.e. the reduction in makespan, of increasing the number of preemptions depends on a number of parameters (even when ignoring the overhead costs). Three obvious parameters are: the number of processors in the multiprocessor, the number of preemptions before the increase and the number of preemptions after the increase. It is also clear that the gain will depend on the parallel program, e.g. there is obviously no gain in increasing the number of preemptions for a program consisting of 5 independent jobs when using a multiprocessor with 5 or more processors.

In this paper we are interested in putting a tight upper bound on the maximal gain of increasing the number of preemptions for any parallel program. This means that we compare the makespan for parallel programs consisting of a set of independent jobs on a multiprocessor when allowing different number of preemptions. We calculate how large the ratio of minimal makespan using i and j preemptions, $i < j$, can be, when using m identical processors. A preemption is at hand if the processing of a job is interrupted and later resumed, possibly on another processor. This ratio has a trivial lower bound that is 1.

This problem and its predecessors has a long story. In 1972, Liu [13] conjectured that for any set of tasks and precedence constraints among them, running on two processors, the least makespan achievable by a nonpreemptive schedule is no more than

$4/3$ the least makespan achievable by a preemptive schedule. The conjecture was proved in 1993 by Coffman and Garey [3]. Here the authors generalize the results to the numbers 4/3, 3/2, 8/5,... i.e. to the numbers $2k/(k+1)$ for some $k \geq 2$. The number k depends on the relative number of preemptions available.

There is also a fundamental bound by Graham from 1969 [7], which is applicable to an arbitrary set of independent jobs. It states that an optimal schedule with no preemptions has at most the double makespan compared to the makespan with an unlimited number of preemptions, when using optimal schedules. This is in accordance with the fact that $2k/(k+1) \to 2$ from below as $k \to \infty$.

Braun and Schmidt proved 2003 a formula that compares a preemptive schedule with i preemptions to a schedule with unlimited number of preemptions in the worst case, using a multiprocessor with m processors [2]. By the McNaughton rule, no more than $m-1$ preemptions are needed in the unlimited case. They generalized the bound 4/3 to the formula $2-2/(m/(i+1)+1)$, which also may be written as $2m/(m+i+1)$.

In the present paper we generalize the results by Braun and Schmidt. We compare i preemptions with j preemptions in the worst case, assuming $i<j$. We thus allow j from $i+1$ to $m-1$, while the problem solved in [2] corresponds to $j = m-1$. In the case $m \geq i+j+1$, which does not coincide with $j = m-1$ unless $i = 0$, we obtain the optimal bound $2(\lfloor j/(i+1) \rfloor + 1)/(\lfloor j/(i+1) \rfloor + 2)$. For example, excluding one preemption $(i = j-1)$ can never deteriorate the makespan more than a factor 4/3, but may do so. This argument cannot be iterated, since different sets of jobs are worst case, depending on the parameters i and j. In the case $m < i+j+1$ we present a formula and an algorithm based on the Stern-Brocot tree.

This paper is organized as follows. In Section 2 we present the problem definition, notation, and the main results. The results are proven in Section 3. Conclusions are given in Section 4.

2. Problem definition, notation and main results

2.1. Problem definition

We consider a multiprocessor with m identical processors and a parallel program P consisting of n independent processes. Of course, P may also be called a set of jobs. The only valid variable of a job is its execution time (also called length). The goal is to minimize the makespan of P, i.e. the makespan with an optimal schedule. The minimal makespan of P on m processors when using i preemptions is denoted $C(P, m, i)$. We consider two scheduling cases: when the program is optimally scheduled on the multiprocessor using at most i preemptions, and similarly with j preemptions, where we assume $i < j$. There is no overhead for process scheduling, context switching or reallo-

cation. Since we consider scheduling with i and j preemptions, we will use notation as i-scheduling and j-scheduling, i-makespan and j-makespan, and so forth.

For a program P we are interested in how large the ratio $C(P, m, i)/C(P, m, j)$ can be. Denote $G(m, i, j) = \max_P C(P, m, i)/C(P, m, j)$, where the maximum is taken over all programs P.

Obviously, the ratio $C(P, m, i)/C(P, m, j)$ is decreasing as a function of i and increasing as a function of j, if the other variables are constant. These properties hold also for $G(m, i, j)$.

2.2. Notation and terminology

A program P' for which $C(P', m, i)/C(P', m, j) = \max_P C(P, m, i)/C(P, m, j)$, where the maximum is taken over all programs P, is called an *extremal program* (or worst case program). A *box schedule* is a schedule where all processors start and stop simultaneously (Fig. 1*a*). Furthermore, we say that processors that share a job, before and after preemption, are *related*. This gives a partition of the processors in sets, where a pair of processors are in the same set if they are related, directly or indirectly via other processors that may form chains of related processors. A set of processors where all processors are related with each other in this way is called a *preemption cluster* (Fig. 1*b*). If it can be scheduled as a box schedule, it has completion time kx/c, where x is the mean value of the completion times for the jobs, k is the number of jobs, and c is the number of processors. A box schedule is possible unless the completion time of one single process is larger than kx/c [14]. Preemption clusters are always considered to be scheduled with a box schedule if possible, since this obviously gives minimal completion time for the cluster. A *critical processor* is the latest processor to complete - it completes at the makespan of the program. A set of jobs on a critical processor may also called critical, as well as a cluster finishing at the global makespan.

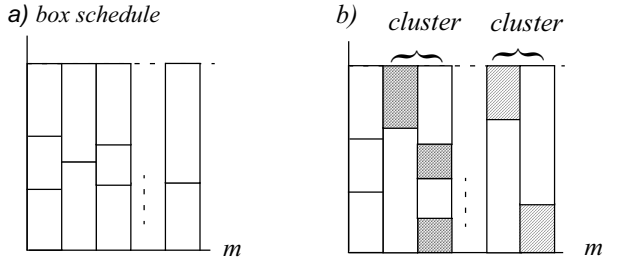


Fig. 1. An example of a box schedule (*a*) and a box schedule with preemption clusters (*b*). The white rectangles represent unpreempted jobs

We denote the number of jobs of a program by n. The McNaughton wrap-around rule [14] provides an optimal schedule for m parallel machines using at most m-1 preemptions. This rule simply specifies that the k:th preempted job, $1 \le k \le m - 1$, starts at the k:th processor and finishes at the $(k + 1)$:st processor. Such a cluster is called a

McNaughton cluster. It follows that for a McNaughton cluster with $i \geq 1$ preemptions, $i+1$ processors are involved, and at least $i+2$ jobs. A *pure McNaughton cluster* has exactly $i+2$ jobs.

An unpreempted job that is scheduled on a processor which has no other jobs will be called a *single job*. A *tower* refers to the jobs scheduled on a processor where all jobs are unpreempted. A tower consisting of k jobs is denoted by k-tower.

2.3. Main result

The main results are the following:

Theorem 1: For all integers $m > 0$ and $0 \leq i < j$, we have

$$G(m,i,j) = 2\left\lfloor \frac{\min(m, i+j+1)}{\min(m+i+1, 2i+j+2)} \right\rfloor_{\min(m-j,\, i+1)}.$$

If $m \geq i+j+1$, the formula can also be written as $G(m,i,j) = 2\dfrac{\lfloor j/(i+1) \rfloor + 1}{\lfloor j/(i+1) \rfloor + 2}$.

The quantity $\left\lfloor \dfrac{m}{n} \right\rfloor_c$ can be defined combinatorially or by the Stern-Brocot tree (see Section 3). It can also be computed by an algorithm that has linear complexity (see Theorem 3), which thus also belongs to the main results. It is a specific rational estimate from below of the ratio m/n, which is in the closed interval $\left(\left\lfloor \dfrac{m}{n} \right\rfloor, \dfrac{m}{n} \right)$.

3. Proofs

3.1. Proof method

We calculate a formula for $G(m,i,j) = \max_P C(P,m,i)/C(P,m,j)$ by characterizing a subset of extremal programs with structure enough to be able to formulate a formula. This leads to a purely mathematical formulation, which is investigated and solved in [12].

We start by showing that there are extremal programs that have a box j-schedule.

Lemma 1: (Prolongation) Consider a program P with an optimal j-schedule. We transform P into a program P' by prolonging some of the jobs so that all processors are constantly busy until $C(P,m,j)$ and so that $C(P,m,j) = C(P',m,j)$.

Proof: This change cannot decrease the i-makespan, but may increase it, while the j-makespan is unchanged.

■

By this lemma we only need to consider programs which have an optimal *i*-schedule which is a box schedule. Unless explicitly stated otherwise, we will normalize the *j*-makespan to 1. From this normalization it certainly follows that the sum of all execution time of the program on the m processors is m. We henceforth consider only such programs.

3.2. An extremal program *P'*

Lemma 2: There is a worst case program *P'* which has $m + i + 1$ jobs, where all jobs in the same *j*-cluster have the same size, and where the two smallest jobs are unpreempted and scheduled to the critical processor. Thus, the critical processor processes a 2-tower.

Proof: As a part of our proof we temporarily introduce a restriction on the way jobs are scheduled to processors when using *i* preemptions. This restriction is defined by an optimal schedule of the same program using *j* preemptions. Without loss of generality we may assume that an optimal schedule using *j* preemptions consists of c preemption clusters. Each such cluster consists of one or more processors - the clusters define specific groups of processors that are relevant also for an *i*-schedule. This means that there is a group of processors associated with each preemption cluster. The temporary restriction on the *i*-schedule that we introduce is that all jobs that are scheduled on a processor in group k ($1 \leq k \leq c$) in this optimal *j*-schedule belong to the same group of processors in an *i*-schedule. Through the coming line of discussion and based on previous results [2] we will obtain the extremal program and an extremal ratio under this restriction. We will then show that the completion time for this program using *i* preemptions is in fact the same whether we apply the restriction or not (by construction we know that the completion time using *j* preemptions is always 1), i.e. the ratio for the extremal program obtained in this way is in fact the same whether we apply the restriction or not. Since a restriction may increase the completion time, but never decrease it, if follows that the extremal program and ratio that we arrive at using the restriction is in fact valid also for the general case without any restriction.

We now start the discussion that will lead us to the extremal program using the restriction. We start by considering an extremal program *P* with the restriction. Due to the restriction, the only level of freedom we have when we decide how to schedule the program is how many of the *i* preemptions that we will assign to each processor group (we assume optimal use of the preemptions in each processor group). Since we are interested in comparing optimal schedules we will of course assign the *i* preemptions to the c processor groups in an optimal way. Using this optimal assignment we consider the processor group which contains the processor with the longest completion time, i.e. the group that limits the completion time of the entire program. Without loss of generality we assume that this is group 1, that group 1 contains m_1 processors, and $j_1 = m_1 - 1$ preemptions when using *j* preemptions. In an *i*-schedule we denote the number of preemptions in group 1 by i_1. Due to the temporary scheduling restrictions

we can consider group 1 in isolation from the other groups. From previous results [2] we know that the extremal (part of) program P that executes on group 1 consists of $n_1 = m_1 + i_1 + 1$ jobs of equal length of m_1/n_1.

We next remark that if we had moved one of the i_k preemptions from any other processor group, group k, say, to processor group 1 (thus obtaining $i_1 + 1$ preemption in group 1), then the completion time of group 1 would decrease (or at least not increase). However, since we are considering an optimal assignment of preemptions to processor groups we know that the completion time of group k will increase so that it becomes longer (or at least as long as) the completion time of group 1 using i_1 preemptions, i.e. group k will then become the critical part of the program. This means that when $i_k > 0$, then the part of program P that runs on group k should be extremal for $i_k - 1$ preemptions. We use this observation below.

We next produce a program *P'* which is identical to program P in group 1. We furthermore consider exactly the same preemption distribution among the groups in *P'* as in *P*. In *P'* we know more about the processes in the other groups than in P because we also assume that group k has $n_k = m_k + i_k$ processes, which all have identical size within each cluster. I.e., we assume that *P'* has the same structure in all groups as in group 1. We next show that from P being extremal it follows that also *P'* is extremal. For this we need to show that the preemption distribution $i_1, \ldots, i_k$ which is optimal for P is optimal also for *P'*. I.e., if the preemptions are distributed differently for *P'*, the completion time cannot be lower.

Consider now that we in *P'* move one of the i_k preemptions from processor group k to processor group 1. Denote the completion time for program P in group k by $c(P, k)$, and the same quantity after a preemption has been moved by $c'(P, k)$. Then we know that $c(P, 1) = c(P', 1)$ (*P'* and P are identical in group 1), $c'(P, k) \le c'(P', k)$ (by [2]), and $c(P, 1) \le c'(P, k)$ (the preemption distribution is optimal for P). Hence, $c(P', 1) \le c'(P', k)$, so the preemption distribution is optimal also for *P'*.

Hence, $n_k = m_k + i_k$ is valid for all processor groups in the extremal program *P'*, except for group 1, where we have $n_1 = m_1 + i_1 + 1$. This means that there are $n = \sum m_k + \sum i_k + 1 = m + i + 1$ jobs in the extremal program. It also means that the completion time of this program using the scheduling restriction is simply the sum of the execution times of the two smallest jobs. If we remove the restriction, we see that we have a program where there must be at least $m + 1$ unpreempted jobs, since we have only i preemptions. Hence, there must be at least one tower with two jobs also without the temporary scheduling restriction (because we only have m processors). It is clear that the smallest such tower consists of the two smallest jobs. This means that the completion time of program *P'* cannot be less than that. As a consequence, we cannot get a smaller completion time for *P*' also when we remove the restriction. Since

removing the restriction may cause the completion time to be lower, it follows that the program *P*' is extremal also without the restriction.

From the way the program is constructed, we then know that we get a completion time of $2(m_1/n_1)$ using the optimal schedule for *i* preemptions, and that the length of the jobs in group 1 (m_1/n_1) is smaller than or equal to the length of the jobs in any other group. The two smallest jobs form a critical tower, which is a processor containing only two jobs, both unpreempted.

The problem of finding an extremal program is thus the problem of finding the program with $m + i + 1$ jobs, where the two smallest jobs are as large as possible, for given values on *m*, *j* and *i*.

■

3.3. The formula for $G(m, i, j)$

We here work with a program *P*' as defined above, in order to calculate a formula. Having m processors and $m + i + 1$ jobs, we can immediately say that $G(m, i, j) \leq 2m/(m + i + 1)$, since $m/(m + i + 1)$ is the mean value of the job sizes (we assume that the *j*-completion time of *P'* is 1), and the two smallest jobs form a critical tower. We also know that all jobs in the same *i*-cluster have the same size. The optimal value of $G(m, i, j)$ is most cases lower than this.

Suppose that an optimal *i*-schedule of *P'* has *c* clusters. In order to find the worst case program, we need to distribute the $m + i + 1$ jobs and the *m* processors on the *c* clusters. This gives *c* ratios $m_1/n_1, \ldots, m_c/n_c$, where $m_1 + \ldots + m_c = m$ and $n_1 + \ldots + n_c = n$. We are interested in the smallest of these ratios: $min(m_1/n_1, \ldots, m_c/n_c)$, which form the critical tower in the *j*-schedule. If we choose a distribution $m_1, \ldots, m_c$ and $n_1, \ldots, n_c$ so that this minimum is maximal, we have a worst case program. Hence,

$$G(m, i, j) = 2max\left(min\left(\frac{m_1}{n_1}, \ldots, \frac{m_c}{n_c}\right)\right),$$

where the maximum is taken over all sets of integers $m_1, \ldots, m_c$ and $n_1, \ldots, n_c$, where all $m_k > 0$ and all $n_k \geq 0$, and the number of clusters *c* remain to be specified. Also, it is essential to find an efficient algorithm to compute this quantity. This was the focus of [12]. The paper describes an algorithm that is $O(max(n,m))$, i.e. it has at most linear complexity in *n* and *m*. Here the following notation is introduced:

Definition 1. $\left\lfloor \frac{m}{n} \right\rfloor_c = max\left(min\left(\frac{m_1}{n_1}, \ldots, \frac{m_c}{n_c}\right)\right)$ *,where the maximum is taken over all sets of integers* $m_1, \ldots, m_c$ *and* $n_1, \ldots, n_c$ *so that* $m_1 + \ldots + m_c = m$, $n_1 + \ldots + n_c = n$ *,all* $m_k > 0$ *and* $n_k \geq 0$.

One motivation for this notation is that it is practical. In the minimum $min(m_1/n_1, \ldots, m_c/n_c)$, where $m_1 + \ldots + m_c = m$ and $n_1 + \ldots + n_c = n$, we are searching for a rational estimate of the ratio *m*/*n* from below. The ratio *m*/*n* is split in *c* ratios, and the larger *c*, the cruder estimate we obtain. The starting value is $c = 1$, where $\left\lfloor \frac{m}{n} \right\rfloor_1 = \frac{m}{n}$, and then $\left\lfloor \frac{m}{n} \right\rfloor_c$ is non-decreasing up to the end value $c = n$, where the estimate is an integer, the floor function: $\left\lfloor \frac{m}{n} \right\rfloor_n = \left\lfloor \frac{m}{n} \right\rfloor$. We do not write the quantity $\left\lfloor \frac{m}{n} \right\rfloor_c$ with a fraction bar, as $\left\lfloor \frac{m}{n} \right\rfloor_c$ since $\left\lfloor \frac{m}{n} \right\rfloor_c \neq \left\lfloor \frac{am}{an} \right\rfloor_c$ in general. For example, $\left\lfloor \frac{2}{5} \right\rfloor_2 = min\left(\frac{1}{3}, \frac{1}{2}\right) = \frac{1}{3}$, while $\left\lfloor \frac{4}{10} \right\rfloor_2 = min\left(\frac{2}{5}, \frac{2}{5}\right) = \frac{2}{5}$.

Returning to the multiprocessor context, we next try to find the value of *c* in $\left\lfloor \frac{m}{n} \right\rfloor_c$ in terms of *m, n, i* and *j*. How many clusters *c* are worst case? Assume that we have a large number of processors, so that this number poses no restriction. Then the jobs in a certain cluster (remember that all jobs in the same cluster have the same size) are as large as possible if they are as few as possible, i.e. if we in the *j*-schedule have $c = i + 1$ pure McNaughton clusters. This means that each cluster has one more job than processors. These clusters have the *j* preemptions distributed among the $i + 1$ clusters. Then the smallest jobs are as large as possible if the clusters are as evenly sized as possible. Hence, some clusters have $\lfloor j/(i+1) \rfloor$ preemptions, and others have $\lceil j/(i+1) \rceil$ preemptions. These $i + 1$ clusters contain $\lfloor j/(i+1) \rfloor + 1$ or $\lceil j/(i+1) \rceil + 1$ processors, respectively. Summing the number of processors in the clusters give in total $j + i + 1$ processors, so we are presently considering the case $m \geq i + j + 1$. Clusters with $\lfloor j/(i+1) \rfloor$ preemptions have the smallest jobs, from which the critical tower in the *i*-schedule is formed. Such a cluster contains $\lfloor j/(i+1) \rfloor + 2$ jobs on $\lfloor j/(i+1) \rfloor + 1$ processors.

Furthermore, the $i + 1$ clusters contain $\lfloor j/(i+1) \rfloor + 2$ or $\lceil j/(i+1) \rceil + 2$ jobs respectively, which give in total $2i + j + 2$ jobs. We have proved a simpler formula in the case $m \geq i + j + 1$:

Lemma 3: If $m \geq i + j + 1$ we have

$$G(m,i,j) = 2\left\lfloor \frac{i+j+1}{2i+j+2} \right\rfloor_{i+1} = 2\frac{\lfloor j/(i+1) \rfloor + 1}{\lfloor j/(i+1) \rfloor + 2}.$$

If $m > i+j+1$, some processors contain unpreempted jobs in both the *i*- and *j*-scheduling.

If m is smaller than $i+j+1$, if $m-j < i+1$, we in the *j*-schedule reduce the number of clusters from $i+1$ to $m-j$ clusters. This allows all preemptions to be used, which is required by an optimal schedule. Since $n = m + i + 1$, the equality $G(m,i,j) = \left\lfloor \frac{m}{n} \right\rfloor_c$ then gives $G(m,i,j) = 2\left\lfloor \frac{m}{m+i+1} \right\rfloor_{m-j}$. We can now summarize the main result in the following theorem by restating Theorem 1 from Section 2.

Theorem 2: For all integers $m > 0$ and $0 \le i < j$, we have $G(m,i,j) = 2\left\lfloor \frac{\min(m, i+j+1)}{\min(m+i+1, 2i+j+2)} \right\rfloor_{\min(m-j,\, i+1)}$. If $m \ge i+j+1$, the formula can also be written as $G(m,i,j) = 2\frac{\lfloor j/(i+1) \rfloor + 1}{\lfloor j/(i+1) \rfloor + 2}$.

The value of $\left\lfloor \frac{m}{n} \right\rfloor_c$ can be calculated by the following algorithm. Here $x \leftarrow y$ signifies assignment of the value of y to the variable x.

Algorithm 1 - calculating the value of $\left\lfloor \frac{m}{n} \right\rfloor_c$.

1. $a \leftarrow m \bmod n$ *and* $b \leftarrow n$.

2. Initialize: $l \leftarrow 0$, $L \leftarrow 1$, $r \leftarrow 1$, $R \leftarrow 1$, $x \leftarrow b - a$, $y \leftarrow a$.

3. If $c > max(x,y)$ *then* $C = \frac{l}{L}$ *and go to 7.*

4. If $x = y$ *then* $C = \frac{l+r}{L+R}$ *and go to 7.*

5. If $x < y$ *do* $l \leftarrow l + r$, $L \leftarrow L + R$, $y \leftarrow y - x$, *and go to 3.*

6. If $x > y$ *do* $r \leftarrow l + r$, $R \leftarrow L + R$, $x \leftarrow x - y$, *and go to 3.*

7. $\left\lfloor \frac{m}{n} \right\rfloor_c = \left\lfloor \frac{m}{n} \right\rfloor + C$.

In the Theorem 3, we import from [12] an algorithm which generates all quantities $\left\lfloor \frac{m}{n} \right\rfloor_c$ for $c = 1,\ldots, n$, as the vector *F*. Suppose that m and n has no common multiple.

In [12] it is proved that $\left\lfloor \frac{m}{n} \right\rfloor_c$ always is an ancestor of m/n. Furthermore, the best decomposition of m/n in two ratios, where the smallest gives $\left\lfloor \frac{m}{n} \right\rfloor_2$, consists of the two ratios that are parents to m/n in the Stern-Brocot tree (see Fig. 2, e.g. $\left\lfloor \frac{2}{5} \right\rfloor_2 = min\left(\frac{1}{3}, \frac{1}{2}\right)$). A triple, where the smallest ratio is $\left\lfloor \frac{m}{n} \right\rfloor_3$, is given from the pair by decomposing the ratio of the two that belongs to the latest generation, e.g. $\left\lfloor \frac{7}{4} \right\rfloor_3 = min\left(\frac{2}{1}, \frac{2}{1}, \frac{5}{5}\right)$. This can be iterated further $c-2$ steps, to derive $\left\lfloor \frac{m}{n} \right\rfloor_c$.

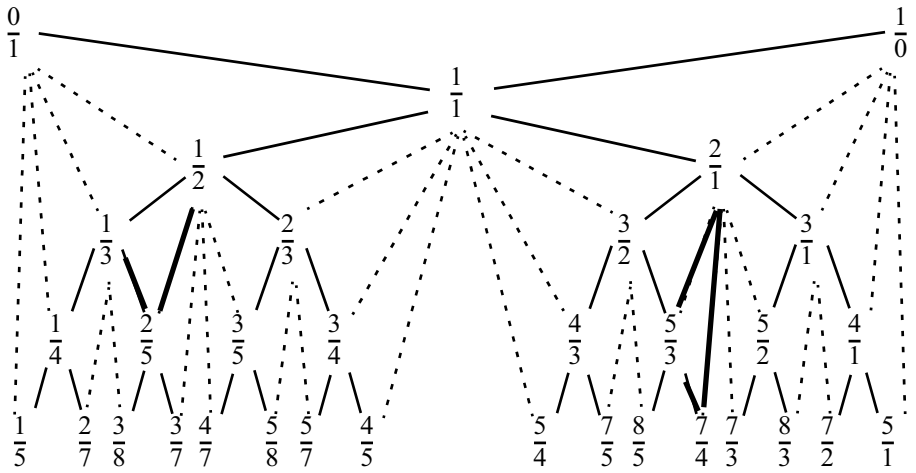


Fig. 2. Stern-Brocot tree with the decomposition of $\left\lfloor \frac{2}{5} \right\rfloor_2 = min\left(\frac{1}{3}, \frac{1}{2}\right)$ and $\left\lfloor \frac{7}{4} \right\rfloor_3 = min\left(\frac{2}{1}, \frac{2}{1}, \frac{5}{5}\right)$

Theorem 3: If $m/n = k$ for some integer k, then $\left\lfloor \frac{m}{n} \right\rfloor_c = k$. Otherwise, the sequence $\left\lfloor \frac{m}{n} \right\rfloor$ is calculated by the following two algorithms.

In the algorithms, the arrow $\leftarrow$ is used in three different ways:

1. If x is a number, $x \leftarrow y$ signifies assignment of the value of y to the variable x.

2. If x is a sequence, A or B, $x \overset{p}{\leftarrow} y$ means insertion of the value y between positions p and $p+1$ in the vector x. The vector x has maximal length n, so entries inserted beyond the length c are ignored. Thus, values to the right of p in the vector x are all juxtapositioned one step to the right, and the length of the vector x is incremented.

3. If x is a sequence F, $x \overset{(i)}{\leftarrow} y$ means appending i copies of the value y after the last position in x. The length of the vector thus increases by i.

Algorithm 2, part 1 - constructing $A\left(\frac{m}{n}\right)$

1. Let $d = GCD(m, n)$. *Assign* $m \leftarrow m/d$ *and* $n \leftarrow n/d$. *Also assign* $m \leftarrow m \bmod n$.

2. Initialize: $A = \left(\frac{0}{1}, \frac{1}{1}\right)$, $B = (0, 1)$, $l \leftarrow 0$, $L \leftarrow 1$, $r \leftarrow 1$, $R \leftarrow 1$, $g = 2$, $p = 1$.

3. Do $A \overset{p}{\leftarrow} \frac{l+r}{L+R}$, $B \overset{p}{\leftarrow} g$ *and* $g \leftarrow g+1$, *and go to the appropriate case 4a, 4b or 4c.*

4a. If $\frac{m}{n} = \frac{l+r}{L+R}$, *exit the iteration and go to 5.*

4b. If $\frac{m}{n} < \frac{l+r}{L+R}$, *do* $r \leftarrow l+r$ *and* $R \leftarrow L+R$ *and go to 3.*

4c. If $\frac{m}{n} > \frac{l+r}{L+R}$, *do* $l \leftarrow l+r$, $L \leftarrow L+R$ *and* $p \leftarrow p+1$ *and go to 3.*

5. $p \leftarrow p+1$, $g \leftarrow g-1$ *and Exit.*

Algorithm 2, part 2 - searching $A\left(\frac{m}{n}\right)$

1. The algorithm starts with values defined by Algorithm 1, i.e: p, g from 5, d from 1 and A and B from 3. Furthermore, initialize as fallows: F is an empty sequence. Then $x \leftarrow p-1$, $F \overset{(1)}{\leftarrow} \frac{m}{n}$ *and* $u \leftarrow 1$, $v \leftarrow 1$, $q \leftarrow g$.

2. $q \leftarrow q-1$. *Pick i so that* $B(i) = q$.

3a. If $q = 0$, $F \overset{(u-v)}{\leftarrow} A(0)$ *and exit the iteration, i.e. go to 4.*

3b. If $i < p$, $F \overset{(u)}{\leftarrow} A(x)$, $x \leftarrow x-1$, $v \leftarrow v+u$. *Go to 2.*

3c. If $i > p$, $u \leftarrow v+u$. *Go to 2.*

4. If $d > 1$, *copy each value in F into d copies without changing the order. This increases the length of the vector by a factor of d.*

5. Add $\left\lfloor \frac{m}{n} \right\rfloor$ *to all values in F.*

6. $\left\lfloor \frac{m}{n} \right\rfloor = F$, *where* $\left\lfloor \frac{m}{n} \right\rfloor_x = F(x)$.

The complexity of the algorithm is O(n).

The algorithm 2 consists of two separate parts. First we construct the ancestor sequence $A\left(\frac{m}{n}\right)$ and the corresponding generation information $B\left(\frac{m}{n}\right)$. In the second algorithm we work backwards in the ancestor sequence to construct the appropriate *c*-values and the sequence $\left\lfloor \frac{m}{n} \right\rfloor_c$, i.e. the values of $\left\lfloor \frac{m}{n} \right\rfloor_c$ for all $c = 1,..., n$.

We end the section by a brief description of the Stern-Brocot tree (Fig. 2). The first part of the algorithm constructs this tree, and the second part searches backwards in its branches. This tree is thoroughly presented in [8]. It can iteratively be constructed by the "mediant addition": $\frac{a}{b} \oplus \frac{c}{d} = \frac{a+c}{b+d}$. Starting from the ratio sequence $\frac{0}{1}, \frac{1}{0}$ and producing mediants in intermediant spaces, we get $\frac{0}{1}, \frac{1}{1}, \frac{1}{0}$, followed by $\frac{0}{1}, \frac{1}{2}, \frac{1}{1}, \frac{2}{1}, \frac{1}{0}$ and $\frac{0}{1}, \frac{1}{3}, \frac{1}{2}, \frac{2}{3}, \frac{1}{1}, \frac{3}{2}, \frac{2}{1}, \frac{3}{1}, \frac{1}{0}$, and so on, in each step adding a new generation of ratios. Note that the size order is preserved: all sequences are increasing. The tree is formed by joining related ratios by edges, and allowing each generation to form a certain level in the tree. It is proven in [8] that all non-negative ratios occur exactly once in the tree, and occur always in lowest terms. We have above established the relevance of the quantity $\left\lfloor \frac{m}{n} \right\rfloor_c$ for the present multiprocessor scheduling problem. In [12], the relevance of the Stern-Brocot tree for the quantity $\left\lfloor \frac{m}{n} \right\rfloor_c$ is discovered, and it is proven that the above algorithm gives $\left\lfloor \frac{m}{n} \right\rfloor_c$.

4. Conclusions

Preempting a job and restarting it on another processor is rather inexpensive in some multiprocessor systems, such as shared memory multiprocessors. It can, however, be very costly and time consuming on other multiprocessor and distributed systems, and in those cases we want to limit the number of preemptions. Avoiding all kind of preemption may, however, result in an unbalanced load in the multiprocessor, even when using the best possible nonpreemptive schedule. Consequently, there is a trade-off between minimizing the number of preemptions (and thus the amount of overhead)

and keeping the load reasonable balanced between the processors (or computers) in the system.

Our results make it possible to exactly quantify one main element in the maximal performance gain by increasing the number of preemptions. Estimates of the overhead costs need to be added. The results are easily computable also for massively parallel multiprocessors, i.e. if m is very large.

5. Discussion

Our results generalize the results by Braun and Schmidt [2]. They look at the maximum performance gain of introducing $m - i - 1$ new preemptions in a schedule with i preemptions running on a system with m processors (computers). We provided a general formula that calculates the maximum gain of introducing an additional j preemptions in a schedule with i preemptions using m processors. For instance, the maximum gain if we introduce one additional preemption in a schedule with one preemption using m processors can be computed.

Figures below (Fig. 3 and Fig. 4) present the function G with i- versus j-preemptive makespan of the worst-case program for $m = 200$ respective $m = 400$. If $i \geq j$ the value of G trivially equals one. This can be seen as the low flat area in the pictures. For $m \geq i + j + 1$ we can represent the formula as a formula with the floor functions (see Lemma 3), so therefore the value of G is constant for all j with $k(i+1) \leq j \leq k(2(i+1)-1)$, $k \in N$. This explains some of the triangle shaped plateaus in the pictures. For example, for $m = 200$ and $i = 50$ the value of $G = 1.333$ for $50 \leq j \leq 98$. The same value G is for $i = 10$ and $10 \leq j \leq 18$. That means that the gain of increasing the number of preemptions from 50 to 55, or from 50 to 98, or even from 10 to 18 is the same, assuming that there are no overhead costs. For $m < i + j$ we have the Stern-Brocot Tree formula: $G(m, i, j) = 2\left\lfloor \frac{m}{m+i+1} \right\rfloor_{m-j}$. For $j = m - 1$ and $i = 0, \ldots, m - 2$ the curve represents the results of Braun and Schmidt [2].

.

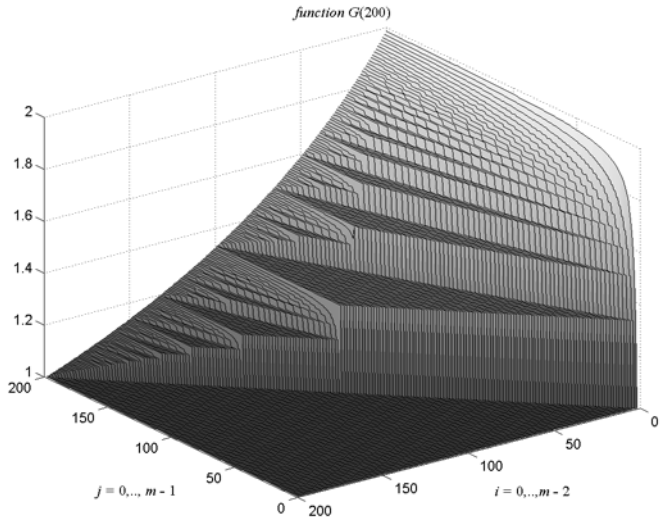


Fig. 3. Function *G* for *m* = 200

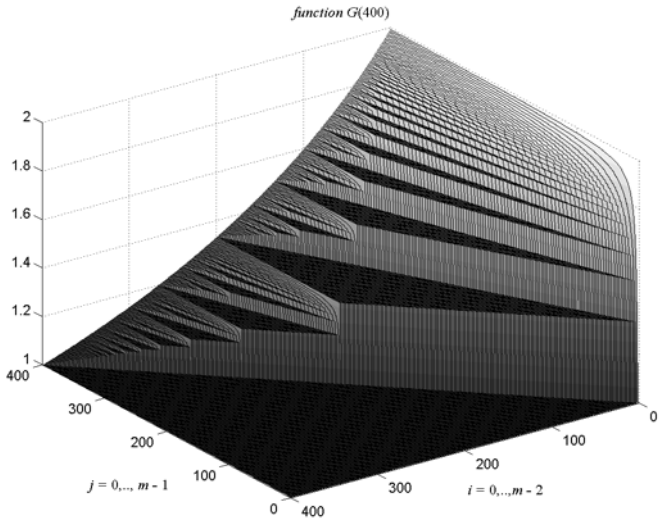


Fig. 4. Function *G* for *m* = 400

6. References

1. Blazewicz, J., Ecker, K. H., Pesch, E., Schmidt, G., Weglarz, J., *Scheduling Computer and Manufacturing Processes*, Springer Verlag, New York, NY 1996, (II ed.) 2001
2. Braun, O., Schmidt, G., *Parallel Processor Scheduling with Limited Number of Preemptions*, Siam Journal of Computing, Vol. 32, No. 3, 2003, pp. 671-680
3. Coffman Jr., E. G., Garey, M. R., *Proof of the 4/3 Conjecture for Preemptive vs. Nonpreemptive Two-Processor Scheduling*, Journal of the ACM, Vol. 40, No. 5, November 1993, pp. 991-1018, ISSN:0004-5411
4. Dobrin, R., Fohler, G., *Reducing the Number of Preemptions in Fixed Priority Scheduling*, in Proceedings of the 16th Euromicro International Conference on Real-Time Systems, ECRTS 04, Catania, Sicily, Italy, June 2004
5. Garey, M. R. and Johnson, D. S., *Computers and Intractability - A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, New York, 1979
6. Graham, R. L., *Bounds for Certain Multiprocessing Anomalies*, Bell System Technical Journal, Vol. 45, No. 9, November 1966, pp.1563-1581
7. Graham, R. L., *Bounds on Multiprocessing Timing Anomalies*, SIAM Journal of Applied Mathematics, Vol. 17, No. 2, 1969, pp. 416-429
8. Graham, R. L., Knuth, D., Patashnik, O., *Concrete Mathematics*, Addison-Wesley, ISBN 0-201-14236-8, 1994
9. Karger, D., Stein, C., Wein, J., *Scheduling Algorithms*, in Handbook of Algorithms and Theory of Computation, M. J. Atallah, editor. CRC Press, 1997
10. Lawler, E. G., Lenstra, J. K., Rinnooy Kan A. H. G., Shmoys, D. B., *Sequencing and Scheduling: Algorithms and Complexity,* S.C. Graves et al., Eds., Handbooks in Operations Research and Management Science, Vol. 4, Chapter 9, Elsevier, Amsterdam, 1993, Chapter 9, pp. 445-522
11. Lennerstad, L., Lundberg, L., *Optimal Scheduling Combinatorics,* Electronic Notes in Discrete Mathematics, Vol. 14, Elsevier, May 2003
12. Lennerstad, H., Lundberg, L., *Generalizations of the Floor and Ceiling Functions Using the Stern-Brocot Tree*, Research Report No. 2006:02, Blekinge Institute of Technology, Karlskrona 2006
13. Liu, C. L., *Optimal Scheduling on Multiprocessor Computing Systems,* in Proceedings of the 13th Annual Symposium on Switching and Automata Theory, IEEE Computer Society, Los Alamitos, CA, 1972, pp. 155-160
14. McNaughton, R., *Scheduling with Deadlines and Loss Functions*. Management Science, 6, 1959, pp. 1-12
15. Pinedo, M., *Scheduling: Theory, Algorithms and Systems*, (2nd Edition), Prentice Hall; 2 edition, 2001, ISBN 0-13-028138-7

Paper III

Paper III

Using Golomb Rulers for Optimal Recovery Schemes in Fault Tolerant Distributed Computing

Kamilla Klonowska, Lars Lundberg, Håkan Lennerstad
Proceedings of of the 17th International Parallel & Distributed Processing Symposium
IPDPS 2003, Nice, France, April 2003

Abstract

Clusters and distributed systems offer fault tolerance and high performance through load sharing. When all computers are up and running, we would like the load to be evenly distributed among the computers. When one or more computers break down the load on these computers must be redistributed to other computers in the cluster. The redistribution is determined by the recovery scheme. The recovery scheme should keep the load as evenly distributed as possible even when the most unfavorable combinations of computers break down, i.e. we want to optimize the worst-case behavior. In this paper we define recovery schemes, which are optimal for a number of important cases. We also show that the problem of finding optimal recovery schemes corresponds to the mathematical problem called Golomb Rulers. These provide optimal recovery schemes for up to 373 computers in the cluster.

1 Introduction

One way of obtaining high availability and fault tolerance is to execute an application on a cluster or distributed system. There is a primary computer that executes the application under normal conditions and a secondary computer that takes over when the primary computer breaks down. There may also be a third computer that takes over when the primary and secondary computers are both down, and so on. The order in which the computers are used is referred to as the *recovery order*, given by a *recovery list.* A lot of cluster vendors support this kind of error recovery, e.g. Sun Cluster [14] MC/ServiceGuard (HP) [9], TruCluster (DEC) [15], HACMP (IBM) [1], and MSCS (Microsoft) [10,16].

An advantage of using clusters, besides fault tolerance, is load sharing between the computers. When all computers are up and running, we would like the load to be evenly distributed. The load on some computers will, however, increase when one or more computers are down, but also under these conditions we would like to distribute the load as evenly as possible on the remaining computers.

The distribution of the load when a computer goes down is decided by the recovery orders of the processes running on the faulty computer. The set of all recovery orders is referred to as the *recovery scheme*, i.e. the load distribution in case of one or more faults is determined by the recovery scheme. The problem of finding optimal (or even reasonably good) recovery schemes has not been studied before by other researchers. In the previous paper [8] we have defined recovery schemes that are optimal for some cases. In this paper we present new recovery schemes that are optimal for a significantly larger number of crashed computers. Some of the schemes are based on so-called Golomb rulers, which have been used in radio astronomy.

2 Problem definition

We consider a cluster with n identical computers. There is one process on each computer. The work is evenly split between these n processes. There is a recovery list associated with each process. This list determines where the process should be restarted if the current computer breaks down. Fig. 1 shows such a system for $n = 4$. We assume that processes are moved back as soon as a computer comes back up again. In most cluster systems this can be configured by the user [9,14,15], i.e. in some cases one may not want automatic relocation of processes when a faulty computer comes back up again. The left side of the figure shows the system under normal conditions. In this case, there is one process on each computer. The recovery lists are also shown; one list for each process. The set of all recovery lists is referred to as the recovery scheme. The right side of Fig. 1 shows the scenario when computer zero breaks down. The recovery list for process zero shows that it should be restarted on computer one when computer zero breaks down. If computer one also breaks down, process zero will be restarted on computer two, which is the second computer in the recovery list. The first computer in the recovery list of process one is computer zero. However, since computer zero is down, process one will be restarted on computer three. Consequently, if computers

zero and one are down, there are two processes on computer two (processes zero and two) and two processes on computer three (processes one and three).

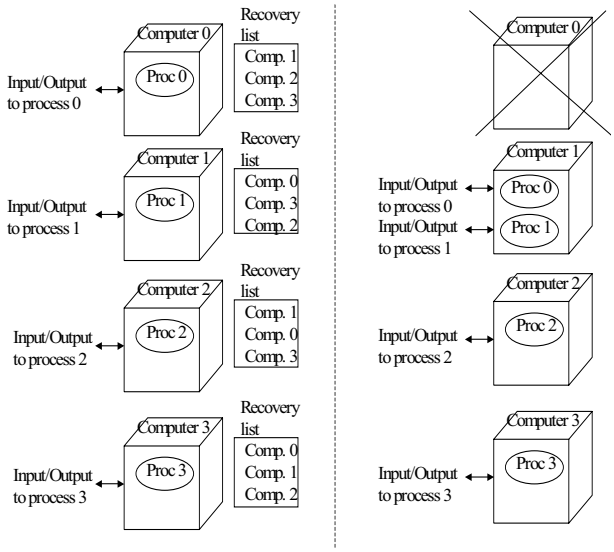


Fig. 1. An application executing on a cluster with four computers

If computers zero and one break down the maximum load on each of the remaining computers is twice the normal load. This is a good result, since the load is as evenly distributed as possible. However, if computers zero and two break down, there are three processes on computer one (processes zero, one and two), i.e. the maximum load on the most heavily loaded computer is three times the normal load. Consequently, for the recovery scheme in Fig. 1, the combination of computers zero and two being down is more unfavorable than the combination of computers zero and one being down.

Our results are also valid when there are *n* external systems feeding data into the cluster, e.g. one telecommunication switching center feeding data into each computer in the cluster. If a computer breaks down, the switching center must send its data to some other computer in the cluster, i.e. there has to be a "recovery list" associated with each switching center. The fail-over order can alternatively be handled by recovery lists at the communication protocol level, e.g. IP takeover [12]. In that case, redirecting the communication to another computer is transparent to the switching center.

Many cluster vendors offer not only the user defined recovery schemes considered here, but also dynamic load balancing schemes in case of a computer going down. The unpredictable worst-case behavior and relatively long switch-over delays of such dynamic schemes is, however, unattractive in many (real-time) applications. Also, external systems cannot use dynamic load balancing schemes when they need to select a new node when the primary destination for their output is not responding. Consequently, the results presented here are very relevant for systems where each node has

its own network address. The large shared nothing clusters must know where the job will go if the current computer crashes. The reason for this is that we make sure that all necessary data are copied to the computer on which the job will be restarted in the event of a crash, and for large clusters we cannot copy all data to every node in the cluster. This means that we must use static schemes for large shared nothing clusters. Finding good recovery schemes is thus a fundamental problem in distributed and cluster systems.

We assume that the work performed by each of the n computers must be moved as one atomic unit. Examples are systems where all the work performed by a computer is generated from one external system or when all the work is performed by one process, or systems where the external communication is handled by IP takeover [12] (in this case all external traffic to one network address is rerouted as one atomic unit).

The previous results are summarized in Section 3. In Section 4 we improve the previous results by defining greedy recovery schemes that are optimal also when a significantly larger number of computers breaks down. The optimal recovery scheme derived from Golomb rulers is presented in Section 5. The conclusions are presented in Section 6.

3 The general lower bound B

We start by introducing some notations. Consider a cluster with n computers. Let $L(n,x,\{c_0,\ldots,c_{x-1}\},RS)$ ($n > x$) denote the load on the most heavily loaded computer when computers $c_0,\ldots,c_{x-1}$ are down and when using a recovery scheme RS. $L(4,2,\{0,1\},RS) = 2$ for the example in Fig. 1, but $L(4,2,\{0,2\},RS) = 3$. Let $L(n,x,RS) = max\ L(n,x,\{c_0,\ldots,c_{x-1}\},RS)$ for all vectors $\{c_0,\ldots,c_{x-1}\}$, i.e. for all combinations of x computers being down. Consequently, $L(n,x,RS)$ defines the worst-case behavior. For the recovery scheme in Fig. 1, $L(4,2,RS) = 3$. The recovery scheme should distribute the load as evenly as possible for any number of failing computers x. Small values of x are (hopefully) more common than large values of x. We denote the $\{L(n,1,RS),\ldots,L(n,n\text{-}1,RS)\}$ as $V(L(n,RS))$ Note that VL is a vector of length n-1 since we disregard from the case when all n computers have crashed.

We say that $V(L(n,RS))$ is *consecutively smaller than* $V(L(n,RS'))$ if the entries are identical up to an index y, and for this index the value of $V(L(n,RS))$ is smaller than $V(L(n,RS'))$. Hence, for some $y < n$ we have

1. $L(n,y,RS) < L(n,y,RS')$ *and*
2. $L(n,z,RS) = L(n,z,RS')$ *for all* $z < y$.

If $y = 1$, it is enough that $L(n,y,RS) < L(n,y,RS')$.

If $V(L(n,y,RS))$ is consecutively smaller than $V(L(n,y,RS'))$, we consider RS as a better recovery scheme than RS', e.g. for $V = V(L(n,y,RS)) = \{2,2,3,3,3,4,4\}$ and $V' = V(L(n,y,RS')) = \{2,3,3,3,3,3,3\}$ the V is consecutively smaller than V'. The behavior when few computers are down is thus strongly prioritized compared to when many computers are down.

The converse to "$V(L(n,y,RS))$ is consecutively smaller than $V(L(n,y,RS'))$" is "$V(L(n,y,RS'))$ is *consecutively larger than or equal* to $V(L(n,y,RS))$". Then either $V(L(n,y,RS))$ and $V(L(n,y,RS'))$ are identical, or $V(L(n,y,RS))$ is consecutively smaller than $V(L(n,y,RS'))$.

Let $VL = min\ V(L(n,RS))$, where minimum is taken over all recovery schemes *RS*.

We have previously defined a lower bound B on VL, i.e. $B \leq VL$ (B is a vector of length n-1), and for $n \leq 11$, we defined a recovery scheme *RS* such that $V(L(n,RS)) = B$ [8]. We refer to such recovery schemes as *optimal recovery schemes*. We have also defined recovery schemes that are optimal when at most $\lfloor \log_2 n \rfloor$ computers break down.

Next we define *bound vectors (BV)*: A BV of length l=2+3+4+...+k has the following sequence: <2,2,3,3,3,4,4,4,4,5,5,5,5,5,…,k,k,…,k> (the vector ends with k entries with value k). For instance, a BV of length 2+3+4+5=14 looks like this: <2,2,3,3,3,4,4,4,4,5,5,5,5,5>. We can simply calculate the n:th entry in the sequence by the formula $\lfloor \sqrt{2(n+1)} + 1/2 \rfloor$.

We now extend the definition of bound vectors to include vectors of any length by taking an arbitrary BV and truncating it to the designated length, e.g. a BV of length 12 looks like this: <2,2,3,3,3,4,4,4,4,5,5>.

Theorem 1: $VL(i) \geq \lceil n/(n-i) \rceil$ ($VL(i)$ is entry number i in VL).

Proof: If i computers are down, there are i processes which must be allocated to the remaining n-i computers. The best one can hope for is obviously to obtain a load of $\lceil n/(n-i) \rceil$ processes on the most heavily loaded computer.

■

Theorem 2: BV is consecutively smaller than VL, i.e., BV and VL are identical up to an index y, and $BV(y) < VL(y)$.

Proof: Because of Theorem 1 we know that $BV(n-1) < n \leq VL(n-1)$ if $n > 3$. Hence BV and VL cannot be identical.

We use proof by contradiction. Assume that VL is consecutively smaller than BV. In that case there is a y such that $BV(x) = VL(x)$ ($x < y$) and $BV(y) > VL(y)$.

Obviously, $VL(y\text{-}1) \leq VL(y)$. BV only increases at entries $x_k = k(k\text{-}1)/2$, for k = 2, 3, 4,…, i.e. $x_2 = 1$, $x_3 = 3$, $x_4 = 6$, …. Consequently, the difference between VL and BV must occur at an entry x_k, i.e. $y = x_k$. This means that, $VL(x_k) < BV(x_k) = k$ for some x_k.

If $k = 2$ ($x_k = 1$), one computer is down, and obviously $VL(1) = 2$, i.e. $VL(1) = BV(1)$. Since $VL(2) = 2$ we know that no two recovery lists start with the same computer. (If two lists corresponding to processes c_1 and c_2 start with the same computer c_3, there will be three processes on computer c_3 if computers c_1 and c_2 break down.)

If $k = 3$ ($x_k = 3$), three computers are down. Since all recovery lists start with different computers, there must be at least one pair of recovery lists l_1 (corresponding to pro-

cess c_1) and l_2 (corresponding to process c_2) such that computer c_3 is first in l_1 and c_3 is the second alternative in l_2. If c_1, c_2 and the first alternative in l_2 break down, there will be three processes on c_3. Consequently, $VL(3) = 3$. Since $VL(5) = 3$, we know that no two recovery lists have the same computer as the second alternative. If two lists corresponding to c_1 and c_2 have the same computer c_3 as the second alternative, then there will be four processes on computer c_3 if computers c_1, c_2, the first alternative in l_1 and l_2 and the computer with c_3 as the first alternative break down.

If $k = 4$ ($x_k = 6$), six computers are down. Since all recovery lists start with different computers and have different computers as the second alternative, there must be at least one triplet of recovery lists l_1, l_2 and l_3 such that computer c_4 is first in l_1 and the second alternative in l_2 and the third alternative in l_3. If computers c_1, c_2, c_3, the first alternative in l_2, the first and the second alternative in l_3 break down, there will be four processes on c_4. Consequently, $VL(6) = 4$. Since $BV(9) = 4$, we know that no two recovery lists have the same computer as the third alternative. If two lists corresponding to c_1 and c_2 have the same computer c_3 as the third alternative, there will be four processes on computer c_3 if computers c_1, c_2, the first alternative in l_1 and l_2 and the computer with c_3 as the first alternative break down.

This procedure can be repeated for all k. From this we can conclude that there is no k such that, $VL(x_k) < k$, where $x_k = 2+3+4+\ldots+k-(k-1)$. Consequently, BV is consecutively smaller than VL.

■

Based on theorems 1 and 2, we define $B(i) = \max(BV(i), \lceil n/(n-i) \rceil)$.

4. Greedy Recovery Schemes

An intuitive recovery scheme (RS) is $R_i = \{(i+1) \bmod n, (i+2) \bmod n, (i+3) \bmod n, \ldots, (i+n-1) \bmod n\}$, where R_i is the recovery list for process i. If $n = 4$ we get: $R_0 = <1,2,3>$, $R_1 = <2,3,0>$, $R_2 = <3,0,1>$, and $R_3 = <0,1,2>$. The intuitive scheme is not optimal for $n = 4$ (in that case $B(2) = 2$ but $L(4,2,\{0,1\},RS) = 3$). For $n = 8$ we know that $B(5) = 3$ but $L(8,5,\{0,1,2,3,4\},RS) = 6$. Consequently, the worst-case behavior of the intuitive scheme is twice as bad as an optimal scheme for $n = 8$. We have previously shown that when at most $x = \lfloor \log_2 n \rfloor$ computers break down $VL(i) = B(i)$ ($i \leq x$) [8]. If we (for some integer $y > 1$) have $x = \sum_{i=2}^{y} i \leq \lfloor \log_2 n \rfloor$, the difference between the intuitive scheme and an optimal scheme can be $\sum_{i=1}^{y} i / y$ (i.e. $VL(x) = y$) but the load

on the most heavily loaded computer in the intuitive scheme may be as high as $x + 1 = \sum_{i=2}^{y} i + 1 = \sum_{i=1}^{y} i$.

This occurs when computers 0, 1, 2, …, x-1 are down, e.g. for $y = 3$ and thus $x = 2+3 = 5$ we get a difference of (1+2+3)/3 = 2 when computers 0,1,2,3,4 are down.

We will now define a recovery scheme called *the greedy recovery scheme* that can guarantee optimal behavior also when significantly more than $\lfloor \log_2 n \rfloor$ computers break down. We will show that this recovery scheme is optimal for a number of important cases.

We start by defining R_0, i.e. the recovery list for process zero (**N** = {1,2,3,…}): $r(x) = \min\{j \in \mathbf{N}\}$, such that for all a_1, a_2, b_1, b_2 that fulfill the conditions $((1 \le a_1 \le b_1 \le x)$ and $(1 \le a_2 \le b_2 \le x)$ and $((a_1 \ne a_2)$ or $(b_1 \ne b_2)))$ so that we get $\sum_{l=a_1}^{b_1} r(l) \ne \sum_{l=a_2}^{b_2} r(l)$.

If $\sum_{l=1}^{x} r(l) < n$, then $R_0(x) = \sum_{l=1}^{x} r(l)$, else

$R_0(x) = \min\{j \in N - \{R_0(1), \ldots R_0(x-1)\}\}$.

We obtain all other recovery lists from R_0, by using $R_i(x) = (R_0(x) + i) \bmod n$.

In the definition of R_0 we use the *step length* vector *r*. The important property in the greedy recovery scheme is that all sums of subsequences $\sum_{l=a}^{b} r(l)$, $(1 \le a \le b \le x)$ are unique, i.e. $\sum_{l=a_1}^{b_1} r(l) \ne \sum_{l=a_2}^{b_2} r(l)$ when $(1 \le a_1 \le b_1 \le x)$ and $(1 \le a_2 \le b_2 \le x)$ and $((a_1 \ne a_2)$ or $(b_1 \ne b_2))$.

From the definition above we see that for large values of n (i.e. $n > 289$) the first 16 elements in R_0 for the greedy recovery scheme are: 1, 3, 7, 12, 20, 30, 44, 65, 80, 96, 122, 147, 181, 203, 251, 289. For $n = 16$, R_0= <1, 3, 7, 12, 2, 4, 5, 6, 8, 9, 10, 11, 13, 14, 15>.

We now define a new vector of length x. This vector is called the *reduced step length vector* (*r'*) and it consists of the first x entries in the step length vector r, where $x = \max(i)$, such that $R_0(i) < n$. From the definition of the greedy recovery scheme we know that all sums of subsequences in r' are unique. Table 1 illustrates this for $n = 97$, resulting in a reduced step length vector of length 10 ($10 = x = \max(i)$, such that $R_0(i) < n = 97$).

Theorem 3: The greedy recovery scheme is optimal as long as x computers or less have crashed, where $x = \max(i)$, such that $R_0(i) < n$.

Entry no. (i)	1	2	3	4	5	6	7	8	9	10
R_0	1	3	7	12	20	30	44	65	80	96
Reduced step length vector r'	1	2	4	5	8	10	14	21	15	16
Note that there are only unique values in the triangular matrix below										
Sum of subsequences of length 1 starting in position i	1	2	4	5	8	10	14	21	15	16
Sum of subsequences of length 2 starting in position i	3	6	9	13	18	24	35	36	31	
Sum of subsequences of length 3 starting in position i	7	11	17	23	32	45	50	52		
Sum of subsequences of length 4 starting in position i	12	19	27	37	53	60	66			
Sum of subsequences of length 5 starting in position i	20	29	41	58	68	76				
Sum of subsequences of length 6 starting in position i	30	43	62	73	84					
Sum of subsequences of length 7 starting in position i	44	64	77	89						
Sum of subsequences of length 8 starting in position i	65	79	93							
Sum of subsequences of length 9 starting in position i	80	95								
Sum of subsequences of length 10 starting in position i	96									

Table 1. All sums of subsequences of the reduced step length vector for $n = 97$

Proof: Let y ($0 \le y < n$) be the heaviest loaded computer when x computers have crashed, where $x = \max(i)$, such that $R_0(i) < n$.

When x computers have crashed, process z ($0 \le z < n$) will in the i:th step end up on computer $z + \sum_{j=1}^{i} r(j) \mod n$ ($1 \le i \le x$). This means that a process that ends up on computer y after i steps was originally allocated to computer $\left(y - \sum_{j=1}^{i} r'(j) + n\right) \mod n$, and this process has passed computers $\left(y - \sum_{j=2}^{i} r'(j) + n\right) \mod n$, $\left(y - \sum_{j=3}^{i} r'(j) + n\right) \mod n$, …, $\left(y - \sum_{j=i}^{i} r'(j) + n\right) \mod n$ before it reached computer y. The lists below show the x possible sequences of computers that need to be down in order for an extra process to end up on computer y, i.e. if computer $(y\text{-}r'(1)+n) \mod n$ is down one extra process will end up on computer y, if computers $(y\text{-}r'(1)\text{-}r'(2)+n) \mod n$ and $(y\text{-}r'(2)+n) \mod n$ are down another extra process will end up on computer y, and so on.

1. $\left(y - \sum_{j=1}^{1} r'(j) + n\right) \mod n$
2. $\left(y - \sum_{j=1}^{2} r'(j) + n\right) \mod n, \left(y - \sum_{j=2}^{2} r'(j) + n\right) \mod n$
3. $\left(y - \sum_{j=1}^{3} r'(j) + n\right) \mod n, \left(y - \sum_{j=2}^{3} r'(j) + n\right) \mod n, \left(y - \sum_{j=3}^{3} r'(j) + n\right) \mod n$

…

$x.\left(y - \sum_{j=1}^{x} r'(j) + n\right) \bmod n, \left(y - \sum_{j=2}^{x} r'(j) + n\right) \bmod n, ..., \left(y - \sum_{j=x}^{x} r'(j) + n\right) \bmod n.$

From the definition of r we know that all sums of subsequences $\sum_{l=a}^{b} r(l)$, $(1 \le a \le b \le x)$ are unique. This means that no computer is included in more than one of the lists. Thus, the worst-case is clearly when the computers in the shortest lists have crashed, e.g. for $x = 3$ the worst-case occurs when computer: $\left(y - \sum_{j=1}^{1} r'(j) + n\right) \bmod n$ have crashed, and then when computers $\left(y - \sum_{j=1}^{2} r'(j) + n\right) \bmod n$, $\left(y - \sum_{j=2}^{2} r'(j) + n\right) \bmod n$ have crashed. Similarly for x=6 the worst case occurs when computer: $\left(y - \sum_{j=1}^{1} r'(j) + n\right) \bmod n$ have crashed, and then when computers $\left(y - \sum_{j=1}^{2} r'(j) + n\right) \bmod n$, $\left(y - \sum_{j=2}^{2} r'(j) + n\right) \bmod n$ have crashed, and then when computers $\left(y - \sum_{j=1}^{3} r'(j) + n\right) \bmod n$, $\left(y - \sum_{j=2}^{3} r'(j) + n\right) \bmod n$, $\left(y - \sum_{j=3}^{3} r'(j) + n\right) \bmod n$ have crashed. By comparing this with the proof of Theorem 2 we see that the first x entries in B and $V(L(n,RS))$ are identical (RS = greedy recovery scheme, $x = \max(i)$ such that $R_0(i) < n$). The theorem follows.

■

To recapitulate our problem formulation: *Given a certain number of computers (n) we want to find a recovery scheme that can guarantee optimal worst-case load distribution when at most x computers are down, and we are interested in the schemes that have as large x as possible.*

By looking at the proof of Theorem 3, we see that the problem can be reformulated as: *Given a number (n) we want to find the longest sequence of positive integers such that the sum of the sequence is smaller than or equal to n and such that all sums of subsequences (including subsequences of length one) are unique.*

This problem is similar to the well-known problem described by Golomb in 1977 as the *spanning ruler* [2]. The spanning ruler with *n* marks and length *L* measures every distance from 1 to *L,* as a distance between two marks on the ruler, in *at most one way.* The interesting combinatorial problem is to determine the *shortest spanning ruler* for each *n*. A dual problem is to determine the *longest covering ruler* with *n* marks and length *L,* which measures every distance from 1 to *L,* as a distance between two marks on the ruler, in *at least one way.* Both of these problems have long histories in the com-

binatorial literature [6]. Gardner [4] termed these objects as “Golomb rulers”, which have been widely adopted. The Golomb rulers are used in a variety of areas such as radio astronomy (placement of antennas), X-ray crystallography, and myriads of other fields such as data encryption and geographical mapping.

5 Golomb Rulers

The Golomb ruler (the spanning ruler) is a sequence of non-negative integers such that no two distinct pairs of numbers from the set have the same difference. These numbers are called *marks* and correspond to positions on a linear scale. The difference between the values of any two marks is called the *distance*. The shortest Golomb rulers for a given number of marks are called the Optimal Golomb Rulers (OGRs) [3]. The search for OGRs becomes more difficult as the number of marks increases. It is known as an NP-complete problem [13]. The optimality for 19 marks was proved computationally by Dollas et al.[4]. OGRs for 20, 21, 22 and 23 marks were produced by a worldwide-distributed effort on the Internet [7, 18]. The problem for a large number of marks is still open. The known OGRs are presented in a table in Appendix A.

An example of the representation of OGR with four marks is shown in Fig. 2. It is possible to measure the distances: 1, 2, 3, 4 as 1+3, 5, 7 as 5+2, 8 as 3+5, 9 as 1+3+5, 10 as 3+5+2 and 11 as 1+3+5+2, but we cannot measure the distance 6.

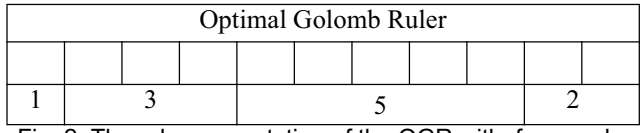


Fig. 2. The ruler presentation of the OGR with four marks

The Golomb rulers can also be presented as a triangle, where each number represents the difference between a specific pair of numbers. Fig. 3 shows an example with four marks. The first order (over the line) represents the marks along a line such that each pair of marks measures a unique linear distance. In the second row are the distances measured between marks placed two apart on the ruler. A ruler with m marks has $m - 1$ first order differences, $m - 2$ second order differences, and so on, up to a single $m - 1$ order difference. Thus, by a k order difference we mean a difference of marks that are k positions from each other.

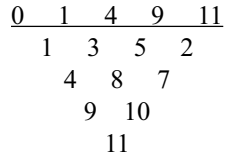


Fig. 3. The triangle presentation of the OGR

Golomb rulers are usually characterized by their absolute distances, rather than differences. The difference sequence is the sequence of first order differences (the first row under the line in the diagram above). The above ruler would be 1-4-9-11 as absolute distances (sometimes this is written as 0-1-4-9-11, but the leading zero is often dropped) and 1-3-5-2 as differences. In this paper we use the differences notation for Golomb recovery schemes. However, we use absolute distances notation for Golomb recovery lists. In Appendix A we present a table with 23 OGRs. For example, the currently best-known 23-mark ruler characterized by the differences is the following: 0-3-4-10-44-5-25-8-15-45-12-28-1-26-9-11-31-39-13-19-2-16-6.

We reformulate the problem in the present paper: *Given a number (n) we want to find the longest sequence of positive integers such that the sum of the sequence is smaller than or equal to n and such that all sums of subsequences (including subsequences of length one) are unique.* Since this problem is equivalent to the problem of finding Golomb rulers, we can use the results about Golomb rulers. However, optimal sequences for large numbers are still not known.

A recovery list is obtained by adding the values of the sequences – it is the sequence of partial sums. The first part of the recovery lists consists of the Golomb rulers, i.e. the first x entries for the largest x such that the sum of the optimal sequence of length x is smaller than n. The remaining part of the recovery list is filled with the remaining numbers (computers) up to n-1. Let G_n be the Golomb ruler with sum $n + 1$ and let $G_n(x)$ be the x:th entry in G_n, e.g. $G_{12} = <1,4,9,11>$ and $G_{12}(1) = 1$, $G_{12}(2) = 4$ and so on. Let then $g_n = x$ be the number of crashed computers with optimal behavior when we have n computers, e.g. $g_{12} = 4$. For intermediate values of k we use the smaller Golomb ruler, and the rest of the recovery list is filled with the remaining numbers up to k-1. For example, by filling with remaining numbers, the ruler G_{12} gives the list {1,4,9,11,2,3,5,6,7,8,10} and G_{18} gives the list {1,4,10,12,17,2,3,5,6,7,8,9,11,13,14, 15,16}. Thus, for $n = 14$ we use G_{12} to obtain the list {1,4,9,11,2,3,5,6,7,8,10,12,13}.

Table 2 shows that the difference, in terms of the number of crashed computers for which we can guarantee an optimal load distribution, is at most one for $n = 35$, e.g. using recovery schemes that are based on the optimal Golomb rulers sequence we are able to guarantee optimal load distribution for 6 crashed computers in the interval $26 = n = 30$, whereas we can only guarantee optimal load distribution for 5 crashed computers for this interval when using the greedy approach. For the interval $31 = n = 34$ we are able to guarantee optimal load distribution for up to 6 crashed computers for both the recovery scheme based on the optimal Golomb sequence as well as for the recovery scheme based on the greedy sequence.

In Fig. 4 we compare the performance of the scheme using Golomb rulers (golomb) with the performance of the greedy recovery scheme (greedy) and the performance of the (old) recovery scheme (org) as a function of the number of computers. The performance is defined as the number of crashes that we can handle while still guaranteeing an optimal load distribution. Fig. 4 shows that the behavior of the Golomb rulers scheme is better for large values of n, e.g., for $n = 100$ the Golomb ruler guarantees optimal behavior even if 11 computers break down, while the greedy scheme guaran-

Length of sequence x	Optimal Golomb Sequence	Sum of optimal sequence	Greedy Sequence	Sum of greedy sequence
1	1	1	1	1
2	1,2	3	1,2	3
3	1,3,2	6	1,2,4	7
4	1,3,5,2	11	1,2,4,5	12
5	1,3,6,5,2	17	1,2,4,5,8	20
6	1,3,6,8,5,2	25	1,2,4,5,8,10	30
7	1,3,5,6,7,10,2	34	1,2,4,5,8,10,14	44
8	1,4,7,13,2,8,6,3	44	1,2,4,5,8,10,14,21	65
9	1,5,4,13,3,8,7,12,2	55	1,2,4,5,8,10,14,21,15	80
10	1,3,9,15,5,14,7,10,6,2	72	1,2,4,5,8,10,14,21,15,16	96
11	2,4,18,5,11,3,12,13,7,1,9	85	1,2,4,5,8,10,14,21,15,16,26	122

Table 2. Comparing the optimal sequences with the greedy sequences

tees optimal behavior if 10 computers break down and the old version can only guarantee optimal behavior as long as at most 6 computers break down. For $n = 1000$ we can guarantee optimal behavior for up to 34 crashed computers using the Golomb rulers scheme, 26 crashed computers using the greedy scheme, whereas the old scheme only guarantees optimal behavior up to 9 crashed computers. For $n = 373$ we can guarantee optimal behavior for up to 22 crashed computers using the Golomb rulers scheme. For larger n it is not known whether the corresponding Golomb rulers are optimal or not. The curve absolute represents Perfect Golomb Rulers, when no distances are omitted (see definition of “Perfect Golomb Rulers “ in [4]). No scheme can pass this border, and we do not know how close one can actually come to this upper bound.

6. Conclusion

In many cluster and distributed systems, the designer must provide a recovery scheme. Such schemes define how the workload should be redistributed when one or more computers break down. The goal is to keep the load as evenly distributed as possible, even when the most unfavorable combinations of computers break down, i.e. we want to optimize the worst-case behavior which is particularly important in real-time systems.

We consider n identical computers, which under normal conditions execute one process each. All processes perform the same amount of work. Recovery schemes that guarantee optimal worst-case load distribution, when x computers have crashed are referred to as optimal recovery schemes for the values n and x. A contribution in this paper is that we have shown that the problem of finding optimal recovery schemes for a system with n computers corresponds to the mathematical problem of finding the longest sequence of positive integers such that the sum is smaller than n and the sums

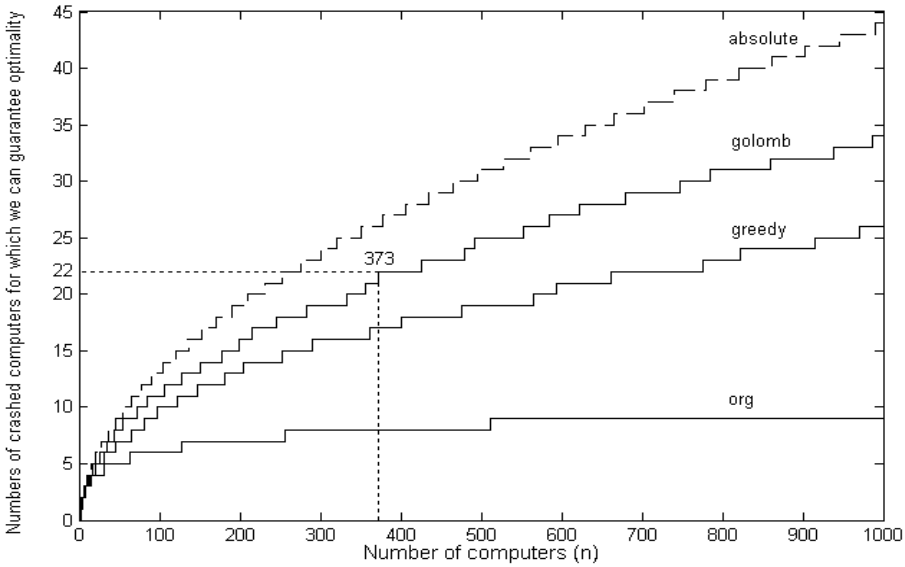


Fig. 4. The “performance” difference between the OGRs scheme ('golomb'), greedy version of the recovery scheme (‘greedy’) and the old version of the recovery scheme (‘org’)

of all subsequences are unique, called the Golomb rulers. It is hard to define an efficient algorithm that finds the longest sequence with these properties.

We have previously obtained recovery schemes that are optimal when $x \leq \lfloor \log_2 n \rfloor$ [8]. However, in this paper we have defined a greedy recovery scheme that is optimal when a significantly larger number of computers are down, and we have presented a new recovery scheme, the Golomb rulers scheme, which is optimal for an even larger number of crashed computers. For large n the difference in terms of the number of crashed computers for which we can guarantee optimality can be significant. For instance, for $n = 1000$ the Golomb rulers scheme guarantees optimality for almost four times as many crashed computers compared to the old scheme, i.e. 34 crashed computers vs. 9 crashed computers. Compared to the greedy recovery scheme it is only 1.3 times better (34 crashed computers vs. 26 crashed computers). (Fig. 4)

The advantage with the greedy algorithm compared to the Golomb rulers is that we can easily calculate a sequence with distinct partial sums also for large n where no Golomb rules are known. Golomb rulers are known for lengths up to 41912 (with the 211 marks) [17,18,19]. Of these the first 373 (with 23 marks) are known to be optimal. Our recovery schemes can be immediately used in commercial cluster systems, e.g. when defining the list in Sun Cluster using the *scconf* command. The results can also be used when a number of external systems, e.g. telecommunication switching centers, send data to different nodes in a distributed system (or a cluster where the nodes have individual network addresses). In that case, the recovery lists are either implemented as alternative destinations in the external systems or at the communication protocol level, e.g. IP takeover [12].

7. References

1. Ahrens, G., Chandra, A., Kanthanathan, M., Cox, D., *Evaluating HACMP/6000: A Clustering Solution for High Availability Distributed Systems*, in Proceedings of the Workshop in Fault-Tolerant Parallel and Distributed Systems, IEEE, College Station, TX, USA, 12-14 Juni, 1994, pp. 2-9
2. Bloom, G. S., Golomb, S. W., *Applications of Numbered, Undirected Graphs,* Proceedings of the IEEE, Vol. 65, No. 4, April, 1977, pp. 562-571
3. Dimitromanolakis, A., *Analysis of the Golomb Ruler and the Sidon Set Problems, and Determination of large, near-optimal Golomb Rulers,* Department of Electronic and Computer Engineering Technical University of Crete, June, 2002
4. Dollas, A., Rankin, W. T., McCracken, D., *A new Algorithm for Golomb Ruler Derivation and Proof of the 19 Marker Ruler*, IEEE Transactions on Information Theory, Vol. 44, No. 1, January 1998, pp. 379-382
5. Gardner, M., *Wheels, Life and Other Mathematical Amusements,* W.H.Freeman and Co., New York, 1983, Chapter 15, pp.152-165
6. Golomb, S. W., Taylor, H., *Cyclic Projective Planes, Perfect Circular Rulers, and Good Spanning Rulers, Sequences And Their Applications – SETA'01*, Proceedings of Seta01, Bergen, Norway, May 2001, pp. 166-180
7. Hayes, B., *Computing Science: Collective Wisdom,* American Scientist, Vol. 98, No. 2, March-April, 1998, pp. 118-122
8. Lundberg, L., Svahnberg, C., *Optimal Recovery Schemes for High-Availability Cluster and Distributed Computing*, Journal of Parallel and Distributed Computing 61(11), 2001, pp. 1680-1691
9. *Managing MC/ServiceGuard for HP-UX 11.0*, http://docs.hp.com/hpux/ha/index.html
10. *Comparing MSCS to other products,* http://www.microsoft.com/ntserver/ProductInfo/Comparisons/comparisons/CompareMSCS.asp
11. Mullender, S., *Distributed Systems (Second Edition),* Addison-Wesley, 1993
12. Pfister, G. F., *In Search of Clusters*, Prentice-Hall, 1998
13. Soliday, S. W., Homaifar, A., Lebby, G. L., *Genetic Algorithm Approach to the Search for Golomb Rulers,* International Conference on Genetic Algorithms, Pittsburg, PA, USA, 1995, pp. 528-535
14. Sun Cluster 2.1 System Administration Guide, Sun Microsystems, 1998.
15. TruCluster, Systems Administration Guide, Digital Equipment Corporation, http://www.unix.digital.com/faqs/publications/cluster_doc
16. Vogels, W., Dumitriu, D., Agrawal, A., Chia, T., and Guo, K., *Scalability of the Microsoft Cluster Service*, Department of Computer Science, Cornell University, http://www.cs.cornell.edu/rdc/mscs/nt98
17. http://www.cuug.ab.ca:8001/~millerl/g3-records.html
18. http://www.distributed.net/ogr/index.html
19. http://www.research.ibm.com/people/s/shearer/grtab.html

8. Appendix A: Optimal Sequences

In Table 2 we present the Optimal Golomb Rulers for $n = 22$ (known as $n = 23$ if we count zero). The quantity m is the number of inequivalent rulers of length of the sequence. It's interesting to see that for $n = 4$ and $n = 10$ we have two OGRs, for $n = 5$ we have four OGRs, and for $n = 6$ we have five OGRs [3,4,6,17,19]

Length of sequence (n)	Sum of sequence	m	Sequence of Marks
1	1	1	1
2	3	1	1,2
3	6	1	1,3,2
4	11	2	1,3,5,2 3,1,5,2
5	17	4	1,3,6,2,5 1,3,6,5,2 1,7,3,2,4 1,7,4,2,3
6	25	5	1,3,6,8,5,2 1,6,4,9,3,2 2,1,7,6,5,4 2,4,3,5,10,1 3,1,8,6,5,2
7	34	1	1,3,5,6,7,10,2
8	44	1	1,4,7,13,2,8,6,3
9	55	1	1,5,4,13,3,8,7,12,2
10	72	2	1,3,9,15,5,14,7,10,6,2 1,8,10,5,7,21,4,2,11,3
11	85	1	2,4,18,5,11,3,12,13,7,1,9
12	106	1	2,3,20,12,6,16,11,15,4,9,1,7
13	127	1	4,2,14,15,17,7,18,1,8,3,10,23,5
14	151	1	4,16,10,27,2,3,14,24,11,12,13,8,1,6
15	177	1	1,3,7,15,6,24,12,8,39,2,17,16,13,5,9
16	199	1	5,2,10,35,4,11,13,1,19,22,16,21,6,3,23,8
17	216	1	2,8,12,31,3,26,1,6,9,32,18,5,14,21,4,13,11
18	246	1	1,5,19,7,40,28,8,12,10,23,16,18,3,14,27,2,9,4
19	283	1	1,7,3,57,9,17,22,5,35,2,21,15,14,4,16,12,13,6,24
20	333	1	2,22,32,21,5,1,12,34,15,35,7,9,60,10,20,8,3,14,19,4
21	356	1	1,8,5,29,27,36,16,2,4,31,20,25,19,30,10,7,21,39,11,12,3
22	372	1	3,4,10,44,5,25,8,15,45,12,28,1,26,9,11,31,39,13,19,2,16,6

Table 3. The known Optimal Golomb Rulers

Paper IV

Paper IV

Using Modulo Rulers for Optimal Recovery Schemes in Distributed Computing

Kamilla Klonowska, Lars Lundberg, Håkan Lennerstad, Charlie Svahnberg
Proceedings of the 10th International Symposium PRDC 2004, Papeete, Tahiti, French Polynesia, March 2004

Abstract

Clusters and distributed systems offer fault tolerance and high performance through load sharing. When all computers are up and running, we would like the load to be evenly distributed among the computers. When one or more computers break down the load on these computers must be redistributed to other computers in the cluster. The redistribution is determined by the recovery scheme. The recovery scheme should keep the load as evenly distributed as possible even when the most unfavorable combinations of computers break down, i.e. we want to optimize the worst-case behavior. In this paper we define recovery schemes, which are optimal for a larger number of computers down than in previous results. We also show that the problem of finding optimal recovery schemes for a cluster with n computers corresponds to the mathematical problem of finding the longest sequence of positive integers for which the sum of the sequence and the sums of all subsequences modulo n are unique.

Keywords: fault tolerance, high performance computing, recovery schemes, Golomb rulers, modulo sequence

1 Introduction

One way of obtaining high availability and fault tolerance is to execute an application on a cluster or distributed system. There is a primary computer that executes the application under normal conditions and a secondary computer that takes over when the primary computer breaks down. There may also be a third computer that takes over when the primary and secondary computers are both down, and so on. The order in which the computers are used is referred to as the *recovery order*, given by a *recovery list.* Most cluster vendors support this kind of error recovery, e.g. the nodelist in Sun Cluster [9], the priority list in MC/ServiceGuard (HP) [3], the placement policy in TruCluster (DEC) [2], cascading resource group in HACMP (IBM) [4], and node preference list in Windows Server 2003 clusters (Microsoft, earlier called MSCS) [7].

An advantage of using clusters, besides fault tolerance, is load sharing between the computers. When all computers are up and running, we would like the load to be evenly distributed. The load on some computers will, however, increase when one or more computers are down, but also under these conditions we would like to distribute the load as evenly as possible on the remaining computers.

The distribution of the load when a computer goes down is decided by the recovery orders/lists of the processes running on the faulty computer. The set of all recovery orders/lists is referred to as the *recovery scheme*. Hence the load distribution is completely determined by the recovery scheme regardless of the number of computers that are down. We consider recovery schemes consisting of recovery lists of the same structure in the sense that one list is obtained by adding 1 to the entries in the previous list, modulo the number of computers (n). This can be viewed as having the computers connected in a ring, and a certain recovery scheme corresponds to a certain set of jumps between computers. The jumps are the same wherever we start. We use the term "wrap-around" when the total sum of jumps for a process exceeds the number of computers. Then a process has passed the initial computer in the ring configuration.

The problem of finding optimal (or even reasonably good) recovery schemes has not been studied before [6]. The outline of this problem is presented in Figure 1. The first mathematical problem formulation does not include "wrap-arounds". These recovery schemes are presented in details in [5] and [6]. "Log" algorithm is an optimal recovery scheme where at most $\lfloor \log_2 n \rfloor$ computers break down in a cluster with n computers [6]. Here "optimal" means that the maximal number of processes on the same computer after k crasches is $BV(k)$. The function $BV(k)$ provides a lower bound for any recovery scheme (see [6] and Section 3). "Greedy" and "Golomb" schemes (presented in [5]) are optimal for a larger number of computers than "Log". The Golomb schemes give optimality for a larger number of computers down than the Greedy scheme. Both consider the formulation where "wrap-around" is not taken into account.

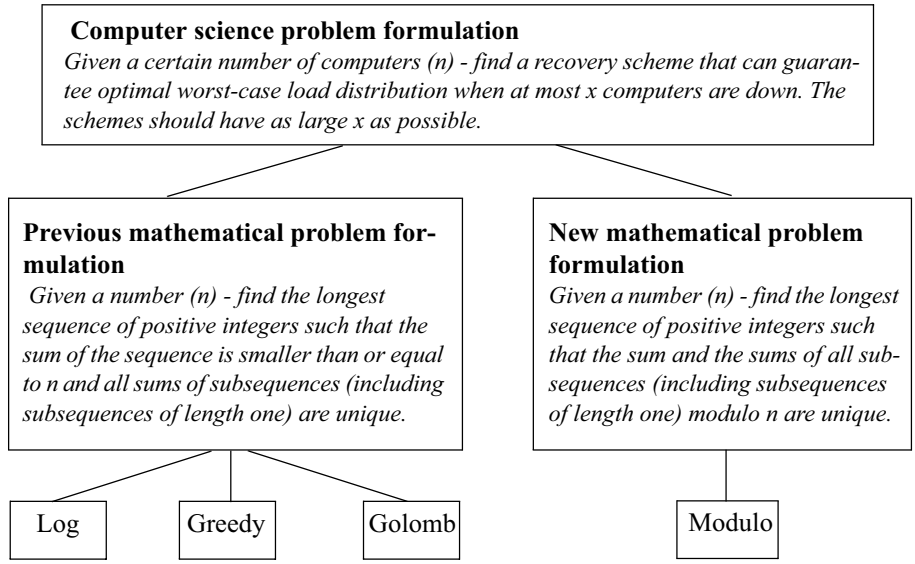


Fig. 1. Outline of the problem formulation

In this paper we optimize also when wrap-around occurs. This is a sharper mathematical formulation of the computer science problem, and gives new recovery schemes, called modulo schemes, that are optimal for a larger number of computers down in the original computer science problem. I.e. these results represent state-of-the-art in the field.

2. Problem definition

We consider a cluster with n identical computers with one process on each computer. The work is evenly split between these n processes. There is a recovery list associated with each process. This list determines where the process should be restarted if the current computer breaks down. The set of all recovery lists is referred to as the recovery scheme. Figure 2 shows such a system for $n = 4$. We assume that processes are moved back as soon as a computer comes back up again. In most cluster systems this can be configured by the user [3,9,10], i.e. in some cases one may not want automatic relocation of processes when a faulty computer comes back up again. The left side of the figure shows the system under normal conditions. In this case, there is one process on each computer. The recovery lists are also shown; one list for each process. The set of all recovery lists is referred to as the recovery scheme. The right side of Figure 2 shows the scenario when computer zero breaks down. The recovery list for process zero shows that it should be restarted on computer one when computer zero breaks down. If computer one also breaks down, process zero will be restarted on computer two, which is the second computer in the recovery list. The first computer in the

recovery list for process one is computer zero. However, since computer zero is down, process one will be restarted on computer three. Consequently, if computers zero and one are down, there are two processes on computer two (processes zero and two) and two processes on computer three (processes one and three). If computers zero and one break down the maximum load on each of the remaining computers is twice the normal load. This is a good result, since the load is as evenly distributed as possible. However, if computers zero and two break down, there are three processes on computer one (processes zero, one and two), i.e. the maximum load on the most heavily loaded computer is three times the normal load. Consequently, for the recovery scheme in Figure 2, the combination of computers zero and two being down is more unfavorable than the combination of computers zero and one being down. We are interesting in the worst-case behavior.

Our results are also valid when there are n external systems feeding data into the cluster, e.g. one telecommunication switching center feeding data into each computer in the cluster. If a computer breaks down, the switching center must send its data to some other computer in the cluster, i.e. there has to be a "recovery list" associated with each switching center. The fail-over order can alternatively be handled by recovery lists at the communication protocol level, e.g. IP takeover [8]. In that case, redirecting the communication to another computer is transparent to the switching center.

Many cluster vendors offer not only the user defined recovery schemes considered here, but also dynamic load balancing schemes in case of a computer going down. The unpredictable worst-case behavior and relatively long switch-over delays of such

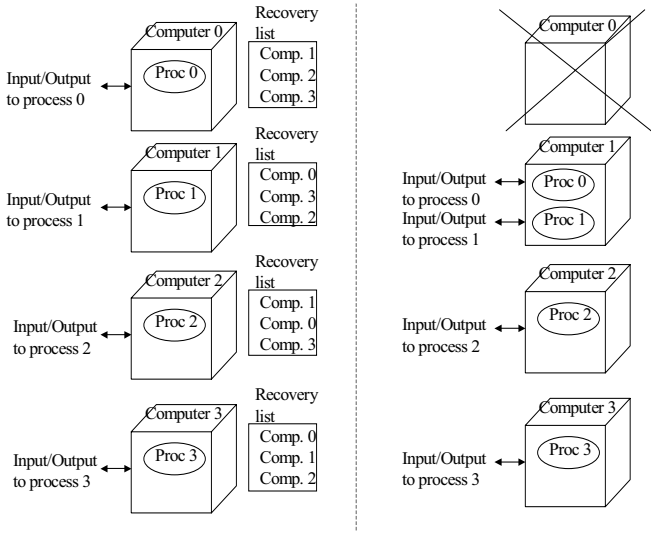


Fig. 2. An application executing on a cluster with four computers

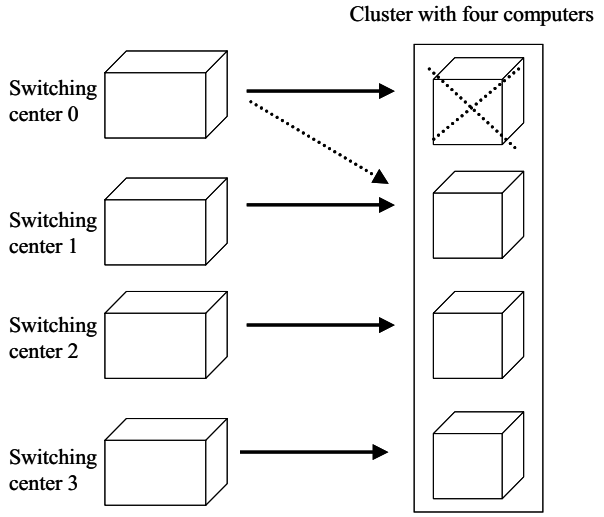


Fig. 3. Load distribution in a telecommunication system when one computer in the cluster crashes

dynamic schemes is, however, unattractive in many (real-time) applications. Also, external systems cannot use dynamic load balancing schemes when they need to select a new node when the primary destination for their output is not responding (see Figure 3). Consequently, the results presented here are very relevant for systems where each node has its own network address. The large shared nothing clusters must know where the job will go if the current computer crashes. The reason for this is that we make sure that all necessary data are copied to the computer on which the job will be restarted in the event of a crash, and for large clusters we cannot copy all data to every node in the cluster. This means that we must use static schemes for large shared nothing clusters. Finding good recovery schemes is thus a fundamental problem in distributed and cluster systems.

We assume that the work performed by each of the n computers must be moved as one atomic unit. Examples are systems where all the work performed by a computer is generated from one external system or when all the work is performed by one process, or systems where the external communication is handled by IP takeover [8] (in this case all external traffic to one network address is rerouted as one atomic unit).

The case when there are a number of processes on each computer, and these processes can be redistributed independently of each other is discussed in Appendix A.

The results are based upon general results which are summarized in Section 3. In Section 4 we present previously used recovery schemes, the greedy and Golomb recovery schemes. The modulo-m schemes are defined in Section 5 and compared with the Golomb schemes in Section 6. Section 7 contains conclusions of the use of modulo rulers.

3. Previous work

We start by introducing some notations. Consider a cluster with n computers.

Let $L(n,x,\{c_0,\ldots,c_{x-1}\},RS)$ $(n > x)$ denote the load on the most heavily loaded computer when computers $c_0,\ldots,c_{x-1}$ are down and when using a recovery scheme RS.

Let $L(n,x,RS) = max\ L(n,x,\{c_0,\ldots,c_{x-1}\},RS)$ for all vectors $\{c_0,\ldots,c_{x-1}\}$, i.e. for all combinations of x computers being down. Consequently, $L(n,x,RS)$ defines the worst-case behavior. The recovery scheme should distribute the load as evenly as possible for any number of failing computers x. Small values of x are (hopefully) more common than large values of x.

Let $V(L(n,RS))$ represent $\{L(n,1,RS),\ldots,L(n,n\text{-}1,RS)\}$. Note that VL is a vector of length n-1 since we disregard from the case when all n computers have crashed.

We say that $V(L(n,RS))$ is *consecutively smaller than* $V(L(n,RS'))$ if the entries are identical up to an index y, and for this index the value of $V(L(n,RS))$ is smaller than $V(L(n,RS'))$. Hence, for some $y < n$ we have

1. $L(n,y,RS) < L(n,y,RS')$ and
2. $L(n,z,RS) = L(n,z,RS')$ for all $z < y$.

If $y = 1$, it is enough that $L(n,y,RS) < L(n,y,RS')$.

If $V(L(n,y,RS))$ is consecutively smaller than $V(L(n,y,RS'))$, we consider RS as a better recovery scheme than RS', e.g. for $V = V(L(n,y,RS)) = \{2,2,3,3,3,4,4\}$ and $V' = V(L(n,y,RS')) = \{2,3,3,3,3,3,3\}$ the V is consecutively smaller than V'. The behavior when few computers are down is thus strongly prioritized compared to when many computers are down.

The converse to "$V(L(n,y,RS))$ is consecutively smaller than $V(L(n,y,RS'))$" is "$V(L(n,y,RS'))$ is *consecutively larger than or equal* to $V(L(n,y,RS))$". Then either $V(L(n,y,RS))$ and $V(L(n,y,RS'))$ are identical, or $V(L(n,y,RS))$ is consecutively smaller than $V(L(n,y,RS'))$.

Let $VL = min\ V(L(n,RS))$, where minimum is taken over all recovery schemes RS.

In [6] we defined a lower bound B on VL, i.e. $B \leq VL$ (B is a vector of length n-1) in a way that the i:th entry in B is: $B(i) = max(BV(i), \lceil n/(n-i) \rceil)$. BV is the *bound vectors* of length $l = 2+3+4+\ldots+k$, defined as: $\{2,2,3,3,3,4,4,4,4,5,5,5,5,5,\ldots,k,k,\ldots,k\}$, where the n:th entry in the sequence is equal to $\lfloor \sqrt{2(n+1)} + 1/2 \rfloor$.

We presented also recovery schemes RS for $n \leq 11$ such that $V(L(n,RS)) = B$. They are refered as optimal recovery schemes when at most $\lfloor log_2 n \rfloor$ computers break down.

In [6] we presented new recovery schemes - improved recovery schemes, called also greedy recovery schemes, which are optimal for a larger number of important cases. They are consecutively smaller than the schemes described in [6]. We also showed that the problem of finding optimal recovery schemes corresponds to the

mathematical problem of finding sequences of integers with minimal sum and for which all sums of subsequences are unique.

In [5] another class of recovery schemes were presented: the Golomb schemes. They are consecutively smaller than the greedy schemes. The advantage with the greedy algorithm compared to the Golomb schemes is that we can easily calculate a sequence with distinct partial sums also for large *n,* where no Golomb sequences, also called rulers, are known. The disadvantage with Golomb schemes is that the sequences are known only for lengths up to 41912 (with the 211 marks), which means that for a cluster with 41912 computers the work will be distributed evenly if 212 computers break down. Only the first 373 Golomb rulers (with 23 marks) are known to be optimal [11,12,13].

4. Greedy and Golomb Recovery Schemes

We will now describe Greedy and Golomb recovery scheme. These recovery schemes are optimal for a larger number of important cases, but the results do not cover load balancing when wrap-around occurs.

We start by defining R_0, i.e. the recovery list for process zero: $r(x) = \min\{j \in \mathbf{N}_+\}$, such that for all a_1, a_2, b_1, b_2 that fulfill the conditions ($1 \le a_1 \le b_1 \le x$) and ($1 \le a_2 \le b_2 \le x$) and (($a_1 \ne a_2$) or ($b_1 \ne b_2$)) we get $\sum_{l=a_1}^{b_1} r(l) \ne \sum_{l=a_2}^{b_2} r(l)$.

If $\sum_{l=1}^{x} r(l) < n$, then $R_0(x) = \sum_{l=1}^{x} r(l)$, else $R_0(x) = \min\{j \in \mathbf{N}_+ - \{R_0(1),...,R_0(x-1)\}\}$. ($n$ is a number of identical computers in a cluster)

All other recovery lists are obtained from R_0, by using $R_i(x) = (R_0(x) + i) \bmod n$.

In the definition of R_0 we use the *step length vector r.* The important property in the greedy recovery scheme is that all sums of subsequences $\sum_{l=a}^{b} r(l)$, ($1 \le a \le b \le x$) are unique, i.e. $\sum_{l=a_1}^{b_1} r(l) \ne \sum_{l=a_2}^{b_2} r(l)$ when ($1 \le a_1 \le b_1 \le x$) and ($1 \le a_2 \le b_2 \le x$) and (($a_1 \ne a_2$) or ($b_1 \ne b_2$)).

From the definition above we see that for large values of n (i.e. $n > 289$) the first 16 elements in R_0 for the greedy recovery scheme are: 1, 3, 7, 12, 20, 30, 44, 65, 80, 96, 122, 147, 181, 203, 251, 289. For $n = 16$, R_0= {1, 3, 7, 12, 2, 4, 5, 6, 8, 9, 10, 11, 13, 14, 15}.

In [6] we have proved that the greedy recovery scheme is optimal as long as x computers or less have crashed, where $x = \max(i)$, such that $R_0(i) < m$.

A special case of the greedy recovery scheme is a sequence called the Golomb ruler with n marks, which is defined as a sequence of positive integers $0 = d_1 < d_2 < \ldots < d_n$ such that all $\binom{n}{2}$ differences $d_j - d_i$, with $i < j$ are distinct. The shortest Golomb rulers for a given number of marks are called Optimal Golomb Rulers (OGRs).

A *Golomb recovery scheme* for a cluster with j computers is obtained from the Golomb recovery list, which contains of two parts. The first part consists of an optimal Golomb ruler with the sum less or equals j and the second part is filled with the remaining numbers up to j-1.

Let $GR_{0,j}$ be the Golomb recovery list for process zero in a cluster with j computers. It contains an optimal Golomb ruler with the sum less than or equals j and the remaining numbers up to j-1. For instance for processor number zero in a cluster with 12 computers we have: $GR_{0,12}$={1,4,9,11,2,3,5,6,7,8,10}. All other recovery lists are obtained from $GR_{0,j}$, by using $GR_{i,j}(x) = (GR_{0,j}(x) + i) mod (j + 1)$ for all $i \leq j$.

The special case of the OGR is a perfect Golomb ruler, which exists where there are exact $\binom{n}{2}$ differences (the consecutive integers from 1 to $\binom{n}{2}$, occurring in any order. That means, that there is no "gap" in the triangle representation. Golomb et al. [1] have proved that there are no perfect Golomb rulers when $n > 4$.

In the formulation which can be handled by Golomb rulers wrap-arounds are ignored – i.e. the situations when the total number of jumps for a process is larger than the number of computers in the cluster. The following formulation takes this into account: *Given a number (n) we want to find the longest sequence of positive integers such that the sum and the sums of all subsequences (including subsequences of length one) modulo n are unique.*

5. Modulo Recovery Scheme

A modulo-n sequence is a sequence of positive integers $\{a_n\}$ for $n \geq 0$, where no two distinct pairs $\{a_i, a_j\}$, $\{a_k, a_l\}$ for $i > j$ and $k > l$ of the numbers from the sequence have the same difference modulo n, for all $n \geq l(l + 1)/2 + 1$, where l is the length of the sequence. For example, for $l = 4$, {1,6,3,10}, {2,1,6,9} and {3,7,9,8} are examples of sequences modulo-11. Figure 4 shows an example of a modulo-11 sequence with all modulo differences. (the first zero is not counted to the length of the sequence)

A modulo sequence is used as the first values of a modulo recovery list for computer number zero in a cluster with n computers. The rest of the recovery list is filled with the remaining numbers up to n-1.

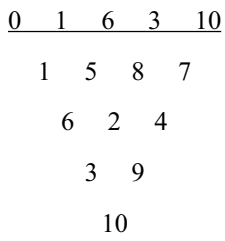


Fig. 4. Modulo sequence for $l = 5$ and $n = 11$ with all differences

Let $MR_{0,n}$ be the modulo recovery list for process zero in a cluster with n computers numbered from zero to n-1. Then for $n = 11$ (and thus $l = 4$) we have: $MR_{0,11}$ ={1,6,3,10,2,4,5,7,8,9}, e.g. the first part in $MR_{0,11}$ is a modulo-11 sequence for length 4 and the rest of $MR_{0,11}$ is the remaining numbers up to n-1. That means that if the computer number zero breaks down, then the process should be restarted first on computer one, followed by computer six, followed by computer three and so on. The jumps from the crashed computers are equal to the differences from the modulo sequence.

The modulo recovery scheme is obtained by the modulo recovery list for process i in a cluster with n computers: $MR_{i,n} = \{(i + MR_{0,n}(1)) \bmod n, (i+ MR_{0,n}(2)) \bmod n, (i + MR_{0,n}(3)) \bmod n,\ldots,(i + MR_{0,n}(n\text{-}1)) \bmod n\}$, where $MR_{0,n}(i)$ is the i:th entry in the $MR_{0,n}$. For instance, for $n = 4$ we have: $MR_{0,n} = \{1,3,2\}$, $MR_{1,n} = \{2,0,3\}$, $MR_{2,n} = \{3,1,0\}$, and $MR_{3,n} = \{0,2,1\}$.

The number of computers in the cluster determines which different modulo scheme to use. For instance, for the clusters with 11, 12, 13, 14, 15, 16, 17 (z) computers we use modulo schemes with length 4 modulo 11. The recovery lists for particulary computers are build as follows: $MR_{i,z} = \{(i + MR_{0,n}(1)) \bmod z, (i+ MR_{0,n}(2)) \bmod z, (i + MR_{0,n}(3)) \bmod z,\ldots,(i + MR_{0,n}(n\text{-}1)) \bmod z\}$, where $MR_{0,n}(i)$ is the i:th entry in the $MR_{0,n}$. For instance, for $z = 5$ we have: $MR_{0,z} = \{1,3,2\}$, $MR_{1,z} = \{2,4,3\}$, $MR_{2,z} = \{3,0,4\}$, $MR_{3,z} = \{4,1,0\}$, and $MR_{3,z} = \{0,2,1\}$.

Figure 5 presents an example of recovery list in a cluster with 11 computers when computer zero has crashed.

Theorem 1: The modulo recovery scheme is optimal as long as x computers or less have crashed, where $x = \max(i)$, such that $MR_{0,n}(i) < n$.

In the following proof, all numbers modulo n are any of the positive integers 0, 1,..., n - 1, denoting computers enumerated in this way.

Proof: Let y ($0 \le y < n$) be the heaviest loaded computer when x computers have crashed, where $x = \max(i)$, such that $M_{0,n}(i) < n$.

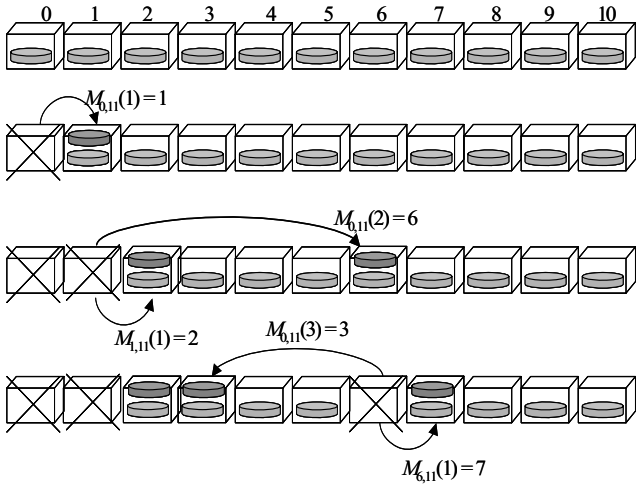


Fig. 5. Modulo-11 recovery list for process zero

When x computers have crashed, process z ($0 \le z < n$) will in the i:th step end up on computer $\left(z + \sum_{j=1}^{i} r(j)\right) \bmod n$ ($1 \le i \le x$). By $r(j)$ we denote the steps from the crashed computers, which are the differences in the modulo sequence. This means that a process that ends up on computer y after i steps was originally allocated to computer $\left(y - \sum_{j=1}^{i} r(j)\right) \bmod n$; and this process has passed computers $\left(y - \sum_{j=2}^{i} r(j)\right) \bmod n$, $\left(y - \sum_{j=3}^{i} r(j)\right) \bmod n$, …, $\left(y - \sum_{j=i}^{i} r(j)\right) \bmod n$ before it reached computer y. The lists below show the x possible sequences of computers that need to be down in order for an extra process to end up on computer y, i.e. if computer (y-r(1)) mod n is down one extra process will end up on computer y, if computers (y-r(1)-r(2)) mod n and (y-r(2)) mod n are down another extra process will end up on computer y, and so on.

1. $\left(y - \sum_{i=1}^{1} r(i)\right) \bmod n$
2. $\left(y - \sum_{i=1}^{2} r(i)\right) \bmod n$, $\left(y - \sum_{i=2}^{2} r(i)\right) \bmod n$
3. $\left(y - \sum_{i=1}^{3} r(i)\right) \bmod n$, $\left(y - \sum_{i=2}^{3} r(i)\right) \bmod n$, $\left(y - \sum_{i=3}^{3} r(i)\right) \bmod n$

…

x. $\left(y - \sum_{i=1}^{x} r(i)\right) \bmod n$, $\left(y - \sum_{i=2}^{x} r(i)\right) \bmod n$, ..., $\left(y - \sum_{i=x}^{x} r(i)\right) \bmod n$.

From the definition of *modulo sequence* we know that all sums of subsequences modulo n are unique. This means that no computer is included in more than one of the lists. Thus, the worst-case is clearly when the computers in the shortest lists have crashed, e.g. for $x = 3$ the worst-case occurs when computer: $\left(y - \sum_{i=1}^{1} r(i)\right) \bmod n$ have crashed, and then when computers $\left(y - \sum_{i=1}^{2} r(i)\right) \bmod n$, $\left(y - \sum_{i=2}^{2} r(i)\right) \bmod n$ have crashed. Similarly for $x = 6$ the worst case occurs when computer: $\left(y - \sum_{i=1}^{1} r(i)\right) \bmod n$ have crashed, and then when computers $\left(y - \sum_{i=1}^{2} r(i)\right) \bmod n$, $\left(y - \sum_{i=2}^{2} r(i)\right) \bmod n$ have crashed, and then when computers $\left(y - \sum_{i=1}^{3} r(i)\right) \bmod n$, $\left(y - \sum_{i=2}^{3} r(i)\right) \bmod n$, $\left(y - \sum_{i=3}^{3} r(i)\right) \bmod n$ have crashed.

We see that no computer is included into more than one of the x first lists. Based on the previous results we know that the scheme is optimal as long as no computer is included in two such lists. The theorem follows.

■

6. Golomb schemes vs. modulo schemes

A modulo sequence can be defined as a special case of a Golomb ruler with l marks $d_1, d_2, \ldots, d_l$, with the property that all $\binom{l}{2}$ differences $(d_j - d_i) \bmod n$ are distinct, for $i < j$, and that all differences $(d_j - d_i) \bmod n$ are non-zero. Since there are $\binom{l}{2}$ differences, we necessarily have $n \geq \binom{l}{2} + 1$. A modulo sequence with exact $\binom{l}{2}$ distinct differences is called a perfect modulo ruler. Exhaustive searches indicate that there are no perfect modulo rulers when $l > 5$.

In Table 1 modulo and Golomb sequences are compared. The sequences are equal for small clusters. We are able to guarantee optimal load balancing for a larger number of crashed computers if we use modulo sequences than Golomb sequences, which means that Golomb schemes are consecutively larger than modulo schemes, and are staying close to the BV vector for more crasches. For example, in the case of 76 com-

Length of sequence *x*	Modulo Sequence	m	Optimal Golomb Ruler	n
1	1	2	1	2
2	1,3	4	1,3	4
3	1,4,6	7	1,4,6	7
4	1,6,3,10	11	1,4,9,11	12
5	1,3,7,17,12	18	1,4,10,12,17	18
6	1,3,23,7,17,12	24	1,4,10,18,23,25	26
7	1,5,7,18,27,30,15	31	1,4,9,15,22,32,34	35
8	1,4,30,15,38,17,22,10	40	1,5,12,25,27,35,41,44	45
9	1,4,48,33,9,50,25,39,19	51	1,6,10,23,26,34,41,53,55	56
10	1,5,49,58,15,18,39,41,47,12	62	1,4,13,28,33,47,54,64,70,72	73
11	1,5,43,34,55,65,71,14,41,73,58	76	2,6,24,29,40,43,55,68,75,76,85	86
12	1,6,78,47,20,24,45,74,57,17,8,87	92	2,5,25,37,43,59,70,85,89,98,99,106	107

Table 1. Comparing modulo sequences with optimal Golomb sequences
m = number of computers in a cluster that we can guarantee optimality using scheme modulo *m*;
n = number of computers in a cluster that we can guarantee optimality using Golomb scheme;

puters in a cluster, modulo sequences guarantee optimal behavior in the case of 11 crashes, while Golomb rulers only guarantee optimality for 10 crashes.

We remark that so called circular rulers are sometimes mentioned in the literature. These are not identical to modulo rulers, since circular rulers consider the differences $(d_j - d_i) \bmod n$ for all i and j. Here there are in total n^2 differences which are required to be distinct.

Consequently, Golomb schemes are not the best schemes, since modulo schemes are proved to be better. If we consider all differences of a Golomb sequence, there are unavoidable gaps if $n > 4$ (no perfect rulers). Modulo sequences have fewer gaps, which allows optimal behavior for fewer computers. Optimal behavior is defined in Section 3.

On Figure 6 we compare the performance of the modulo scheme (modulo) with the performance of the scheme using Golomb rulers (golomb) and the performance of the greedy recovery scheme (greedy) as a function of the number of computers. The performance is defined as the number of crashes that we can handle while still guaranteeing an optimal load distribution. All of the schemes have the same behavior up to seven computers in a cluster. For larger number of computers the behavior of the modulo scheme is better than Golomb or greedy schemes, e.g., for $n = 100$ the modulo scheme guarantees optimal behavior even if 12 computers break down, while the Golomb scheme guarantees optimal behavior if 11 computers break down and the greedy scheme guarantees optimal behavior if 10 computers break down.

7. Conclusions

In many cluster and distributed systems, the designer must provide a recovery scheme. Such schemes define how the workload should be redistributed when one or more computers break down. The goal is to keep the load as evenly distributed as possible, even when the most unfavorable combinations of computers break down, i.e. we want to optimize the worst-case behavior which is particularly important in real-time systems.

We consider n identical computers, which under normal conditions execute one process each. All processes perform the same amount of work. Recovery schemes that guarantee optimal worst-case load distribution, when x computers have crashed are referred to as optimal recovery schemes for the values n and x. A contribution in this paper is that we have shown that the problem of finding optimal recovery schemes for a system with n computers corresponds to the mathematical problem of finding the longest sequence of positive integers such that the sum and the sums of all subsequences (including subsequences of length one) modulo n are unique. No efficient algorithm that finds the longest sequence with these properties is known.

We have previously obtained recovery schemes that are optimal when a larger number of computers are down, but they do not cover load balancing when wrap-around occurs [6].

With modulo sequences we minimize the maximum load also in the event of wrap-arounds. Modulo sequences allow optimal behavior for a larger number of crashed computers than Golomb and greedy sequences. For example, in the case of 92-107 computers in a cluster, modulo-m sequences guarantee optimal behavior in the case of

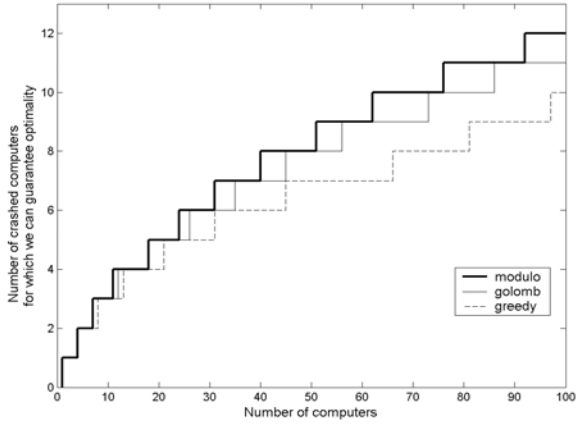


Fig. 6. Performance difference between the Golomb, greedy and modulo scheme

12 crashes, while Golomb rulers only guarantee optimality for 11 crashes, and greedy scheme for 10 crashes.

The advantage with the greedy algorithm compared to other schemes is that we can easily calculate a sequence with distinct partial sums also for large *n* where no Golomb and modulo sequences are known. Golomb rulers are known for lengths up to 41912 (with the 211 marks) [11,12,13]. Of these the first 373 (with 23 marks) are known to be optimal while modulo rulers are known only for 13 marks. Modulo sequences are known only up to 92 computers in a cluster.

Our recovery schemes can be immediately used in commercial cluster systems, e.g. when defining the list in Sun Cluster using the *scconf* command. The results can also be used when a number of external systems, e.g. telecommunication switching centers, send data to different nodes in a distributed system (or a cluster where the nodes have individual network addresses). In that case, the recovery lists are either implemented as alternative destinations in the external systems or at the communication protocol level, e.g. IP takeover [8].

8. References

1. Golomb, S. W., and Taylor, H., *Cyclic Projective Planes, Perfect Circular Rulers, and Good Spanning Rulers, Sequences And Their Applications – SETA'01,* Proceedings of Seta01, Bergen, Norway, May 2001, pp. 166-180
2. Hewlett-Packard Company, *TruCluster Server - Cluster Highly Available Applications*, Hewlett-Packard Company, September 2002
3. Hewlett-Packard, *Managing MC / ServiceGuard*, Hewlett-Packard, March 2002
4. IBM, *HACMP, Concepts and Facilities Guide*, IBM, July 2002
5. Klonowska, K., Lundberg, L. and Lennerstad, H., *Using Golomb Rulers for Optimal Recovery Schemes in Fault Tolerant Distributed Computing,* in Proceedings of the 17th International Parallel & Distributed Processing Symposium IPDPS 2003, Nice, France, April 2003, pp. 213
6. Lundberg, L., and Svahnberg, C., *Optimal Recovery Schemes for High-Availability Cluster and Distributed Computing*, Journal of Parallel and Distributed Computing 61(11), 2001, pp. 1680-1691
7. Microsoft Corporation, *Server Clusters: Architecture Overview for Windows Server 2003*, Microsoft Corporation, March 2003
8. G.F. Pfister, *In Search of Clusters*, Prentice-Hall, 1998
9. Sun Microsystems, *Sun Cluster 3.0 Data Services Installation and Configuration Guide*, Sun Microsystems, 2000
10. TruCluster, Systems Administration Guide, Digital Equipment Corporation, http://www.unix.digital.com/faqs/publications/cluster_doc
11. http://www.cuug.ab.ca:8001/~millerl/g3-records.html
12. http://www.distributed.net/ogr/index.html
13. http://www.research.ibm.com/people/s/shearer/grtab.html

Appendix A: Non-Atomic Loads

Hitherto, we have considered the case when all work performed by a computer must be moved as one atomic unit. Obviously, there could be more than one process on each computer and these processes may in some cases be redistributed independently of each other, and in that case there is one recovery list for each process. In this appendix we consider the case where there are p processes on each computer. We assume that the workload is evenly split between the processes.

Previous studies show that, if there is a large number of processes on each computer, the problem of finding optimal recovery schemes becomes less important [6]. The reason for this is that the intuitive solutions become better. Consequently, the importance of obtaining optimal recovery schemes is largest when p is small compared to n. In the main part of this paper, we have obtained results for $p = 1$. The techniques used for obtaining those results are also useful when we consider $p > 1$.

We now define bound vectors of type p. The bound vectors discussed in Section 3 were of type one, i.e. $p = 1$. A bound vector of type p is obtained in the following way (the first entry in the bound vector is entry 1):

1. $t = p$; $i = 1$; $j = 1$; $r = 1$; $v = 1$
2. If $r = 1$ then $t = t+1$; $r = j$; $v = v+1$ else $r = r$-1
3. Let entry i in the Bound vector of type p have the value t/p
4. $i = i+1$
5. If $v = p$ and $r = 1$ then $j = j+1$; $v = 0$
6. Go to 2 until the bound vector has the desired length.

A bound vector of type two looks like this (note that the load on each computer is normalized to one when all computers are up and running): {3/2, 4/2, 4/2, 5/2, 5/2, 6/2, 6/2, 6/2, 7/2, 7/2, 7/2, 8/2, 8/2, 8/2, 8/2, 9/2, 9/2, 9/2, 9/2, 10/2,…

A bound vector of type three looks like this: {4/3, 5/3, 6/3, 6/3, 7/3, 7/3, 8/3, 8/3, 9/3, 9/3, 9/3, 10/3, 10/3, 10/3, 11/3, 11/3, 11/3, 12/3…

A bound vector of type four looks like this: {5/4, 6/4, 7/4, 8/4, 8/4, 9/4, 9/4, 10/4, 10/4, 11/4, 11/4, 12/4, 12/4, 12/4, 13/4,…

We now extend the definition of VL to cover not only the case when $p = 1$, but also the case when $p > 1$. Consequently, $VL(x)$ $(1 \le x < n)$ denotes the maximum number of processes divided with p (in order to normalize the original load to one) on the most heavily loaded computer when using an optimal recovery scheme and when x computers are down, not only when $p = 1$ but also when $p > 1$.

Based on p and n we now define B in the following way: $B(i)$ = max (entry i in the bound vector of type p, $\lceil pn(n-i) \rceil / p$).

When $p = 2$ and $n = 12$ we get: B = {3/2, 4/2, 4/2, 5/2, 5/2, 6/2, 6/2, 6/2, 7/2, 12/2, 24/2}.

For $p \le n$, we have previously defined a recovery scheme - the p-process recovery scheme - that is optimal as long as at most $\lfloor \log_2 \lfloor n/p \rfloor \rfloor$ computers break down [6]. We

call it *p*-process because we consider the case where there are *p* processes on each computer when all computers are up and running.

We now define *the modulo p-process recovery scheme*. The scheme is defined when $p \leq n$ (remember that optimal recovery schemes are most important when *p* is small compared to *n*). We will show that this recovery scheme is optimal also when significantly more than $\lfloor \log_2 \lfloor n/p \rfloor \rfloor$ computers break down. In fact, the results presented in Section 5 correspond to the case when $p = 1$.

Let M_0 be the modulo scheme for some *n* and $M_0(k)$ be *k* entry in M_0. Let then $M_{i,j}$ denotes the modulo recovery list for process *j* ($0 \leq j < p$) on computer *i* ($0 \leq i < n$). We define $M_{i,j}$ based on M_0, i.e. the recovery scheme for process zero in the single-process recovery scheme defined in Section 5.

If $M_0(k) + j\lfloor n/p \rfloor < n$, then $M_{i,j}(k) = (M_0(k) + i + j\lfloor n/p \rfloor) \bmod n$, else $M_{i,j}(k) = (M_0(k) + i + j\lfloor n/p \rfloor + 1) \bmod n$, where $M_{i,j}(k)$ denotes integer number *k* in the lists for process *j* on computer *i*.

The table below shows the recovery lists when $n = 11$ and $p = 2$.

$M_{0,0}$	1,6,3,10
$M_{0,1}$	6,0,8,4
$M_{1,0}$	2,7,4,0
$M_{1,1}$	7,1,9,5
$M_{2,0}$	3,8,5,1
$M_{2,1}$	8,2,10,6
$M_{3,0}$	4,9,6,2
$M_{3,1}$	9,3,0,7
$M_{4,0}$	5,10,7,3
$M_{4,1}$	10,4,1,8

Theorem 2: Let MRS_n denote the modulo *p*-process recovery scheme for *n* computers with *p* processes on each computer, and let *B* be a bound vector of type *p*. In that case the first entry in $V(L(n,RS_n))$ is equal to $B(1)$, the second entry in $V(L(n,RS_n))$ is equal to $B(2)$, the third entry in $V(L(n,RS_n))$ is equal to $B(3)$, …, the *x*:th entry in $V(L(n,RS_n))$ is equal to $B(x)$, where $x = \max(i)$, such that $R_0(i) < \lfloor n/p \rfloor$.

Proof: The proof is similar to the proof of Theorem 1. As in that proof, all numbers modulo *n* are any of the positive integers 0, 1,..., *n* - 1.

Let *y* ($0 \leq y < n$) be the heaviest loaded computer when *x* computers have crashed, where $x = \max(i)$, such that $R_0(i) < \lfloor n/p \rfloor$. The lists below show the *x*p* possible sequences of computers that need to be down in order for an extra process to end up on computer *y*.

1.1 $\left(y - \sum_{i=1}^{1} r(i)\right) \bmod n$

1.2 $\left(y - \sum_{i=1}^{2} r(i)\right) \bmod n, \left(y - \sum_{i=2}^{2} r(i)\right) \bmod n$

…

$1.x \left(y - \sum_{i=1}^{x} r(i)\right) \bmod n, \left(y - \sum_{i=2}^{x} r(i)\right) \bmod n, \ldots, \left(y - \sum_{i=x}^{x} r(i)\right) \bmod n$

$2.1 \left(y - \sum_{i=1}^{1} r(i) - \lfloor n/p \rfloor\right) \bmod n$

$2.2 \left(y - \sum_{i=1}^{2} r(i) - \lfloor n/p \rfloor\right) \bmod n, \left(y - \sum_{i=2}^{2} r(i) - \lfloor n/p \rfloor\right) \bmod n$

…

$2.x \left(y - \sum_{i=1}^{x} r(i) - \lfloor n/p \rfloor\right) \bmod n, \left(y - \sum_{i=2}^{x} r(i) - \lfloor n/p \rfloor\right) \bmod n, \ldots,$

$\left(y - \sum_{i=x}^{x} r(i) - \lfloor n/p \rfloor\right) \bmod n$

…

$p.1 \left(y - \sum_{i=1}^{1} r(i) - (p-1)\lfloor n/p \rfloor\right) \bmod n$

$p.2 \left(y - \sum_{i=1}^{2} r(i) - (p-1)\lfloor n/p \rfloor\right) \bmod n, \left(y - \sum_{i=2}^{2} r(i) - (p-1)\lfloor n/p \rfloor\right) \bmod n$

…

$p.x \left(y - \sum_{i=1}^{x} r(i) - (p-1)\lfloor n/p \rfloor\right) \bmod n, \left(y - \sum_{i=2}^{x} r(i) - (p-1)\lfloor n/p \rfloor\right) \bmod n, \ldots,$

$\left(y - \sum_{i=x}^{x} r(i) - (p-1)\lfloor n/p \rfloor\right) \bmod n.$

Since $x = \max(i)$, such that $R_0(i) < \lfloor n/p \rfloor$, we see that no computer is included into more than one of the lists one to x. By looking at the proofs of Theorem 4 and Theorem 3, we see that the bound is optimal as long as no computer is included in two such lists. The theorem follows.

■

Paper V

Paper V

Extended Golomb Rulers as the New Recovery Schemes in Distributed Dependable Computing

Kamilla Klonowska, Lars Lundberg, Håkan Lennerstad, Charlie Svahnberg
Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05), Denver, Colorado, April 2005

Abstract

Clusters and distributed systems offer fault tolerance and high performance through load sharing. When all computers are up and running, we would like the load to be evenly distributed among the computers. When one or more computers break down the load on these computers must be redistributed to other computers in the cluster.

The redistribution is determined by the recovery scheme. The recovery scheme should keep the load as evenly distributed as possible even when the most unfavorable combinations of computers break down, i.e. we want to optimize the worst-case behavior.

We have previously defined recovery schemes that are optimal for some limited cases. In this paper we find a new recovery schemes that are based on so called Golomb rulers. They are optimal for a much larger number of cases than the previous results.

Keywords: high performance computing, clusters, fault tolerance, recovery scheme, Optimal Golomb Rulers, load balance.

1 Introduction

Load balancing and availability are important in online distributed dependable systems. Krishna and Shin define fault tolerance as “an ability of a system to respond gracefully to an unexpected hardware or software failure” [13]. Many fault-tolerant computer systems mirror all operations, e.g. every operation is performed on two or more duplicate systems so that if one fails the other can take over its job. In [2] fault tolerance for distributed computing is discussed from a wide viewpoint. An advantage of using clusters, besides fault tolerance, is load sharing between the computers [15].

One can handle this problem dynamically, where transfer decisions depend on the actual current system state. A problem with dynamic policies is that they are rather unpredictable. The other way is to use static policies that are generally based on the information of the average behavior of the system. Here, the transfer decisions are independent of the actual current system state. That makes them less complex and more predictable than dynamic policies [7].

The problem studied in this paper is: *how to evenly distribute the load among the running computers, when one or more computers in a cluster go down?* We consider static redistribution of work.

There is a primary computer that executes the application under normal conditions and a secondary computer that takes over when the primary computer breaks down. There may also be a third computer that takes over when the primary and secondary computers are both down, and so on. When all computers are up and running, we would like the load to be evenly distributed. The load on some computers will, however, increase when one or more computers are down, but also under these conditions we would like to distribute the load as evenly as possible on the remaining computers. The distribution of the load when a computer goes down is decided by the *recovery lists* of the processes running on the faulty computer. The set of all recovery lists is referred to as the *recovery scheme*. Hence the load distribution is completely determined by the recovery scheme regardless of the number of computers that are down.

Most cluster vendors support this kind of error recovery, e.g. the nodelist in Sun Cluster [17], the priority list in MC/ServiceGuard (HP) [5], the placement policy in TruCluster (DEC) [4], cascading resource group in HACMP (IBM) [6], and the node preference list in Windows Server 2003 clusters (Microsoft, earlier called MSCS) [14].

The problem of finding optimal (or even reasonably good) recovery schemes has been studied in [8,9,10]. In this paper we calculate the recovery schemes, that guarantee load balance for larger number of computers in a cluster and are simply to calculate. The work considers the worst case scenario which is very relevant in real-time systems.

The problem formulation is presented in Section 2. Our previous research on this problem is described in Section 3. In sections 4 and 5 we present our main results. The conclusions are presented in Section 6.

2. Problem formulation

We assume that the work performed by each of the n computers must be moved as one atomic unit. This is a very common scenario including cases when:

- the work performed by a computer is generated from one external system (in this case the recovery schemes are implemented in the external systems as second, third, fourth,... alternative destinations (alternative cluster nodes) in case the primary, secondary,... destination goes down);
- there is one network address for each computer and we use IP takeover (or similar techniques);
- all the work performed by a computer is done by one process or a group of related processes that share local resources and thus must be moved as one unit.

In [11,12] we discuss the case when there are a number of independent processes on each computer. The results presented here can be easily generalized to the case with a number of independent processes on each computer using the same technique as in the previous papers [11,12].

Consider a cluster with n identical nodes, which under normal conditions execute one process each. All processes perform the same amount of work. We will find a recovery scheme for the cluster that determines where the process should be restarted if its current node goes down. The recovery scheme should keep the load balanced in a cluster when even the most unfavorable combination of nodes go down. The recovery scheme should also be possible to calculate for large n in polynomial time.

A recovery scheme R consists of n recovery lists - one list per process. Let S be a sequence of distances, i.e. a sequence of positive integers $S = \langle s_1, s_2, \ldots, s_{n-1} \rangle$, where $\sum_{j=1}^{i} s_j$ (i=1,...,n-1) is the distance between the crashed node and the node on which the process should be restarted in the i:th step. Let then R_0 be a recovery list for process 0 (executing on computer 0). The recovery list R_0 we construct from the sequence S: $R_0 = \langle 0, s_1, s_1 + s_2, s_1 + s_2 + s_3, \ldots, s_1 + \ldots + s_{n-1} \rangle$.

(we come back to this in Section 4.4)

When one node in a cluster is down, the load on the most loaded node in a cluster (let us call it node Z) will contain two jobs: its own and the job from the node $Z - s_1$.

When two nodes in a cluster are down, there are two possibilities. If the two first numbers in the sequence are equal $(s_1 = s_2)$, the load on Z can contain three jobs: its own, from node $Z - s_1$ and from node $Z - (s_1 + s_2)$. This situation is presented on Fig. 1a). The nodes $Z - s_1$ and $Z - (s_1 + s_2)$ are the crashed nodes. The second possibility is when two first numbers in a sequence are different $(s_1 \neq s_2)$. In that case, Z can contain only two jobs: its own and from the node $Z - s_1$ or from $Z - (s_1 + s_2)$. This situation is presented on Fig. 1b).

Consequently, the worst-case load on Z is less when $s_1 \neq s_2$ than when $s_1 = s_2$.

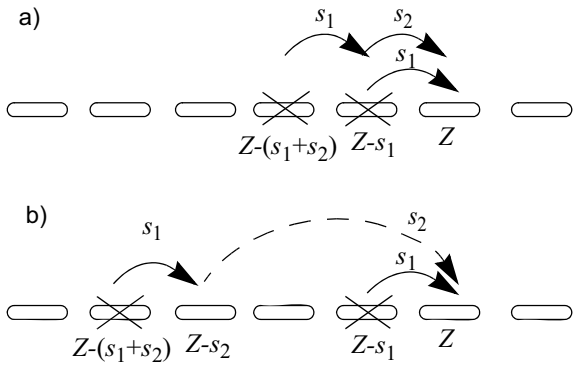


Fig. 1. The scenario with three crashed nodes a) when $s_1 = s_2$ b) when $s_1 \neq s_2$

Fig. 1b) also shows the situation when three nodes in a cluster are down and $s_1 \neq s_2$. Then the load on Z can contain three jobs: its own and from nodes $Z - s_1$ and $Z - (s_1 + s_2)$. Note, that in this case node $Z - s_2$ is also a crashed node.

The worst-case for three crashed nodes is when the first three numbers in a sequence are equal, i.e. $s_1 = s_2 = s_3$. In that case, Z will contain four jobs: it's own and from nodes $Z - s_1$, $Z - (s_1 + s_2)$ and $Z - (s_1 + s_2 + s_3)$.

We are interesting in finding a recovery scheme, where, in the worst-case scenario, the load on the node Z is minimum.

3. Previous research

In [12] the authors initiate the problem of finding a recovery scheme that can guarantee optimal worst-case load distribution when at most k computers are down. The schemes should have as large k as possible. The authors present and prove the *Log* algorithm, that generates the recovery schemes that guarantee optimality where at most $\lfloor log_2 n \rfloor$ computers go down. *Optimal* means that the maximal number of processes on the same computer after k crashes is $MV(k)$, where the function $MV(k)$ provides a lower bound for any static recovery scheme. The function MV is described later in Section 4.1.

Another algorithm, called *Greedy*, is presented in [11]. This algorithm generates the recovery schemes that give optimality for a larger number of cases than the *Log* algorithm, (i.e. Greedy guarantees optimality also when more than $\lfloor log_2 n \rfloor$ computers go down.). The Greedy algorithm is based on the mathematical problem of finding the sequence of positive integers such that all sum of subsequences are unique and minimal. It is simple to calculate the Greedy algorithm even for large n. The recovery list for process zero, executing on computer number zero, consists of two parts. The first

part is a sequence from the Greedy algorithm, and the second part is filled with the remaining numbers. The other lists are obtain from the first list by adding one in the modulo sense (as described in Section 2 in the present paper). The recovery scheme consists of all recovery lists.

A special case of the Greedy algorithm is an algorithm called *Golomb*, described in [9]. The name of this algorithm comes from the Golomb ruler (called also the spanning ruler), which is a sequence of non-negative integers such that no two distinct pairs of numbers from the set have the same difference. These numbers are called *marks* and correspond to positions on a linear scale. The difference between the values of any two marks is called the *distance*. The shortest Golomb rulers for a given number of marks are called the Optimal Golomb Rulers (OGRs) [3]. The search for Optimal Golomb Rulers becomes more difficult as the number of marks increases. It is known as an NP-complete problem [16]. The problem of finding OGRs for a large number of marks is still open. An example of the representation of OGR with four marks is shown in Fig. 2. It is possible to measure the distances: 1; 2; 3; 4 as 1+3; 5; 7 as 5+2; 8 as 3+5; 9 as 1+3+5; 10 as 3+5+2 and 11 as 1+3+5+2, but we cannot measure the distance 6.

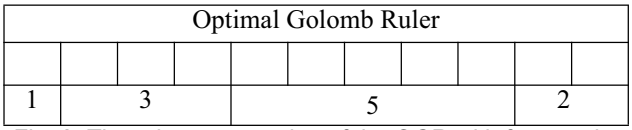


Fig. 2. The ruler presentation of the OGR with four marks

The Golomb rulers can also be presented as a triangle, where each number represents the difference between a specific pair of the numbers. Fig. 3 shows an example with four marks (same example as in Fig. 2).

0 1 4 9 11
1 3 5 2
4 8 7
9 10
11

Fig. 3. The triangle presentation of the OGR

In the Golomb rulers the wrap-arounds are ignored, i.e. the situations when the total number of "jumps" for a process is larger than the number of computers in the cluster. (a "jump" is a distance between a crashed node and a node on which the process should be restarted). Including the wrap-arounds gives a new mathematical formulation of finding the longest sequence of positive integers such that the sum and the sums of all subsequences (including subsequences of length one) *modulo n* are unique (for a given *n*). This mathematical formulation of the computer science problem gives new more powerful recovery schemes, called *Modulo* schemes, that are optimal for a larger number of crashed computers than the Golomb schemes [10].

All these algorithms (Log, Greedy, Golomb and Modulo) guarantee optimality for a certain number of crashed computers in a cluster. In [8] we calculate the best possible

recovery schemes for any number of crashed computers. Due to the computational complexity of the problem, finding such a recovery scheme is a very complex task. We are only able to present the optimal recovery scheme for a maximum of 21 computers in a cluster ($n \leq 21$).

Here we present new recovery schemes that are based on the Optimal Golomb Rulers but guarantee optimal behavior for a much larger number of crashed nodes.

4. Recovery schemes

4.1. The lower bound *MV*

The function $MV(k)$ is a lower bound for any recovery scheme: $MV(k) = \max\left(BV(k), \left\lceil \frac{n}{n-k} \right\rceil\right) \leq L(n, k, R)$ for all $k = 1, 2, \ldots, p$ [12]. Here $L(n,k,R)$ is the load on the most loaded computer in the cluster with n computers using the recovery scheme R, when the most unfavorable combination of k computers are down. The vector BV is increasing and contains exactly k entries that equals k for $k \geq 2$. The j:th entry in the BV equals $\lfloor \sqrt{2(j+1)} + 1/2 \rfloor$. Hence, $BV = \langle 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, \ldots \rangle$. The numbers increase by one at the $\frac{l(l+1)}{2}$ entry for $l \geq 1$, i.e. $BV\left(\frac{l(l+1)}{2}\right) = l + 1$.

Definition 1. We say that a recovery scheme is ***optimal*** *for q crashed nodes if it is tight up to q with the bound vector MV, i.e.* $L(n, k, R) = MV(k)$ *for* $k = 1, \ldots, q$.

For two crashed nodes in a cluster $MV(2) = 2$. As long as $s_1 \neq s_2$ the recovery scheme is optimal for two crashed nodes. When $s_1 = s_2$ the recovery scheme is not optimal.

4.2. The sequence *S*

Definition 2. By a ***crash route*** C_i *we define a sequence:* $<(s_1+s_2+\ldots+s_i), (s_2+\ldots+s_i), \ldots, s_i>$. *The j:th entre in the sequence* C_i *we denote by* $C_{i,j}$.

Then there are the following crash routes:
crash route C_1: s_1,
crash route C_2: (s_1+s_2), s_2,
crash route C_3: $(s_1+s_2+s_3)$, (s_2+s_3), s_3, i.e. $C_{3,2} = (s_2+s_3)$.

The problem of finding a sequence that is optimal up to k crashed nodes we formulate as follows:

Find a sequence $S = \langle s_1, s_2, \ldots, s_m \rangle$ of positive integers such that:

1. the sum of the elements in the sequence is less than equal n, i.e. $\sum_{i=1}^{k} s_i \leq n$

2. $l = \left\lfloor \frac{\sqrt{8k+9}-1}{2} \right\rfloor$ *(we come back to this in Section 5)*

3. the first l crash routes are disjoint, i.e. $\bigcap_{k=1}^{l} C_k = \varnothing$, *(i.e. the number of unique values equals:* $\left| \bigcup_{k=1}^{l} C_k \right| = \frac{l(l+1)}{2}$*)*

4. the following $\frac{l(l+1)}{2} - l$ *crash routes have at least l unique values compared to the previous crash routes, i.e.* $\left| C_y \cap \bigcup_{k=1}^{y-1} C_k \right| \leq y - l$ for $l < y < \frac{l(l+1)}{2}$.

4.3. Example of the sequence

Let $n = 19$ and $S = <1,3,2,5,2,5>$. We present the sequence and its crash routes as a triangle (see Fig. 4). The bold numbers are all different. The first three crash routes (C_1={1}; C_2={4,3}; C_3={6,5,2}) are disjoint. In the following crash routes there are at least 3 unique values. It is simple to calculate that for $l = 3$ the number of crashed nodes is smaller than 9. Thus this sequence should be optimal up to at least 8 crashed nodes.

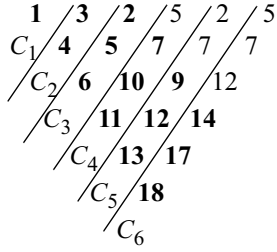


Fig. 4. Triangle representation of the sequence

Fig. 5 presents an outline when using the sequence <1,3,2,5,2,5>. The load on Z increases by one when all nodes from the crash route are down. In this example the load on Z increases when $Z - C_{1,1} = Z - 1$ is down, the nodes from the crash route C_2, i.e. $Z - C_{2,1} = Z - 4$ and $Z - C_{2,2} = Z - 3$, the nodes from the crash route C_3, i.e. $Z - C_{3,1} = Z - 6$, $Z - C_{3,2} = Z - 5$, $Z - C_{3,3} = Z - 2$, and the nodes from the crash route C_4. After nine crashed nodes the load on Z increases by four (four crash routes). If in the crash route C_4 only two nodes would crash, then the load on Z would increase only by three. From the bound vector we know that $MV(9) = 4$, so the scheme

will be not optimal for nine nodes in our example. However, $MV(8) = 4$ and that the scheme guarantee optimality for 8 crashed nodes.

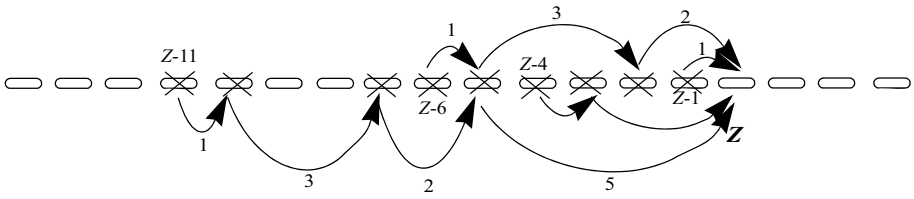


Fig. 5. Example of using a sequence <1,3,2,5,2,5>. Here the load on *Z* after 9 crashes equals 5. N.B. crash routes C_3 and C_4 share computer *Z*-5

Theorem 1: A recovery scheme constructed in a way when the first C_x crash routes for $1 \le x \le l$ are disjoint and the following C_y crash routes for $l < y < \frac{l(l+1)}{2}$ have at least l unique values compared to previous crash routes (i.e. $\left| C_y \cap \bigcup_{k=1}^{y-1} C_k \right| \le y - l$) guarantees optimality when at most $\frac{l(l+1)}{2} + l - 1$ nodes crash.

Proof: First we will show that if the first C_x crash routes for $1 \le x \le l$ are disjoint then the recovery scheme guarantees optimality as long as $\frac{l(l+1)}{2}$ nodes from these crash routes have crashed. Let Z ($0 \le Z < n$) be the heaviest loaded node when x nodes have crashed.

When x nodes have crashed, process z ($0 \le z < n$) will in the i:th step end up on a node $z + \sum_{j=1}^{i} s_j$ ($1 \le i \le x$). It means that a process that ends up on node Z after i steps was originally allocated to a node $Z - \sum_{j=1}^{i} s_j$ and this process has passed the following nodes $Z - \sum_{j=2}^{i} s_j$, $Z - \sum_{j=3}^{i} s_j$, …, $Z - \sum_{j=i}^{i} s_j$ before it reached node Z. It means that the process passed all the nodes $Z - C_{i,j}$, for all $j = 1, \ldots, i$, where $C_{i,j}$ is a crash route for the i:th entry in the sequence S and the j:th entry in the crash route.

The lists below show the x possible sequences of the nodes that need to be down in order for an extra process to end up on the node Z, i.e. if node $Z - s_1$ is down one extra process will end up on node Z, if nodes $Z - (s_1 + s_2)$ and $Z - s_2$ are down (and $s_1 \neq s_2$) another extra process will end up on node Z, and so on.

1. $Z - s_1 = Z - C_{1,1}$

2. $Z - (s_1 + s_2)$, $Z - s_2 = Z - C_{2,1}, Z - C_{2,2}$

3. $Z - (s_1 + s_2 + s_3)$, $Z - (s_2 + s_3)$, $Z - s_3 = Z - C_{3,1}, Z - C_{3,2}, Z - C_{3,3}$

…

x. $Z - \sum_{j=1}^{x} s_j$, $Z - \sum_{j=2}^{x} s_j$,..., $Z - \sum_{j=x}^{x} s_j = Z - C_{x,1}, \ldots, Z - C_{x,x}$.

The first l crash routes are disjoint. It means that no node is included in more than one of the lists. Thus, the worst case is clearly when the nodes in the shortest lists have crashed, e.g. for $k = 3$ the worst case occurs when the node from the crash route $Z - C_1$ has crashed, and then when the nodes from the crash route $Z - C_2$ have crashed. It means that after three crashes the load on the node Z has three jobs. The lower bound $MV(3) = 3$, so the optimality follows. Similarly for $k = 6$ the worst case occurs when the nodes from $Z - C_1$, $Z - C_2$ and $Z - C_3$ have crashed. It means that six nodes have crashed and the work on the node Z is four. The lower bound $MV(6) = 4$ and the optimality follows.

The number of different nodes in the crash routes is equal to $1 + 2 + 3 + \ldots + l = l(l+1)/2$ and consequently for $k = l(l+1)/2$ the worst case occurs when the nodes from $Z - C_1$, $Z - C_2$,..., $Z - C_l$ have crashed. Because $MV\left(\frac{l(l+1)}{2}\right) = l + 1$, the optimality follows.

The crash route C_{l+1} has l unique values compared to the previous crash routes, i.e. it has one value that is included in one of the previous crash routes. This means, that the process from node $Z - C_{l+1}$ will pass l new nodes to end up on the node Z and one node that is already down.

When only $l - 1$ nodes crash from the crash route C_{l+1}, then the process from node $Z - C_{l+1}$ do not end up on the node Z and the load on Z will be still $l + 1$. The optimality is guaranteed because $MV\left(\frac{l(l+1)}{2} + l - 1\right) = l + 1$.

Analogically, $l - 1$ nodes from the crash route C_{l+2} can crash to guarantee optimality for $\frac{l(l+1)}{2} + l - 1$ crash nodes, because C_{l+2} has l unique values compared to the previous crash routes and two values that are included in the previous crash routes.

It needs $\frac{l(l+1)}{2} - l$ crash routes with l unique values compared to the previous crash

routes to still guarantee optimality for $\frac{l(l+1)}{2} + l - 1$ crash nodes.

Thus the recovery scheme when the first l crash routes are disjoint and the following $\frac{l(l+1)}{2} - l$ crash routes have at least l unique values compared to previous crash routes guarantees optimality when at most $\frac{l(l+1)}{2} + l - 1$ nodes crash.

■

4.4. Regular recovery scheme

In this paper we mainly consider *regular* recovery schemes, when all lists have the same structure, based on one recovery list (R_0). The recovery list R_0 we construct from the sequence of distances *S*:

$R_0 = \langle 0, s_1, s_1 + s_2, s_1 + s_2 + s_3, \ldots, s_1 + \ldots + s_{n-1} \rangle$.

Then the recovery schemes are constructed as fallows:

$R_i = \{(r_0 + i) \bmod n, \ldots, (r_n + i) \bmod n\}$ for $0 \le i < n$, where $r_i = s_1 + \ldots + s_i$ is the *i*:th position in the vector R_0.

For example, for $n = 6$ we have following regular recovery scheme given by the vector: $R_0 = \langle 0, 1, 3, 5 \rangle$:

R_0: 1, 3, 5
R_1: 2, 4, 0
R_2: 3, 5, 1
R_3: 4, 0, 2
R_4: 5, 1, 3
R_5: 0, 2, 4

where R_i is a recovery list for computer number *i*. Here, the sequence *S* = <1,2,2>.

The first zero in a vector R_0 remains a computer's number. For a clarity when describing recovery lists, we omit the computer's number.

The set of regular recovery schemes with length *n* (*n* computers) is denoted by *R*(*n*). There are clearly $(n-1)!$ different such sequences with *n* computers, so $|R(n)| = (n-1)!$.

4.5. Example of the recovery scheme based on the sequence

Let *n* = 19 and *S* = <1,3,2,5,2,5>. Then a recovery list based on this sequence is: R_0 = <0,1,4,6,11,13,18>. This recovery scheme is optimal for 8 crashed nodes.

Fig. 6 shows a triangle with a recovery list and all differences between the numbers in the list. All the bold numbers are different.

0	1	4	6	11	13	18
1	**3**	**2**	5	2	5	
4	**5**	**7**	7	7		
6	**10**	**9**	12			
11	**12**	**14**				
13	**17**					
18						

Fig. 6. The difference triangle

We see, that the first three crash routes are disjoint. (C_1={1}; C_2={4,3}; C_3={6,5,2}) and in the rest of the crash routes there are at least l = 3 unique values. Thus, this sequence is optimal up to at least $\frac{l(l+1)}{2} + l - 1 = 8$ crashed nodes.

The bold numbers in a triangle build a kind of trapezium. Therefore the new schemes constructed in this way we will call the *trapezium* schemes.

5. Trapezium recovery schemes vs. Golomb recovery schemes

The first l crash routes in the trapezium recovery scheme are called the *Golomb rulers* [1,3]. The Golomb recovery scheme is a regular recovery scheme. For n nodes in a cluster we build a recovery list R_0 by using the known Golomb ruler with the sum less or equal to n, and the rest of the recovery list is filled with the remaining numbers up to n-1, e.g. for n = 12 we have the list <1,4,9,11,2,3,5,6,7,8,10>.

The trapezium recovery schemes are based on the Golomb rulers and filled with the numbers that construct the new crash routes. The trapezium recovery schemes give the better optimality than the Golomb schemes.

In Table 1 the load balancing of the trapezium and Golomb schemes are compared. The optimal load balancing is guaranteed for a larger number of crashed nodes if we use the trapezium sequences than the Golomb sequences, which means that trapezium schemes are staying close to the *MV* vector for more crashes. For example, in the case of 127 computers in a cluster, the trapezium sequences guarantee optimal behavior in the case of 21 crashes, while Golomb rulers only guarantee optimality for 13 crashes.

Table 1 shows some examples of the sequences. For intermediate values of k we use either the smaller sequence or we can simply find a new trapezium sequence using the nearest Golomb ruler and build new crash routes. For example, if the number of nodes in a cluster is 15 then using the sequence with the sum 5 the optimality is guaranteed for 4 crashes. Examples of the trapezium sequences are presented in Table 2.

number of nodes in a cluster = sum of the trapezium sequences	max number of crashes using trapezium sequences	number of nodes in a cluster = sum of the Optimal Golomb Rulers	max number of crashes using Optimal Golomb Rulers
5	3	6	3
16	6	17	5
41	10	44	8
73	15	72	10
127	21	127	13
201	28	199	16
305	36	283	19
390	45	372	22

Table 1. Comparing the trapezium sequences with the optimal Golomb sequences

Figure 7 compares the performance of the trapezium scheme ("trapezium"), with the performance of the scheme using Golomb rulers ("golomb") as a function of the number of nodes in a cluster (n) up to n = 1024. The performance is defined as the number of crashes that we can handle while still guaranteeing an optimal load distribution when the most unfavorable combination of computers crash. There are no big differences between the schemes for small n. For larger n the behavior of the trapezium scheme is much better than Golomb schemes, e.g., for n = 100 the trapezium scheme guarantees optimal behavior when 15 nodes break down, while the Golomb scheme guarantees optimal behavior if 10 computers break down. The larger steps in the trapezium curve are because k and l are depend on each other, i.e.

$\frac{(l-1)l}{2} + l - 1 \le k \le \frac{l(l+1)}{2} + l - 1$ (see also Section 4.2).

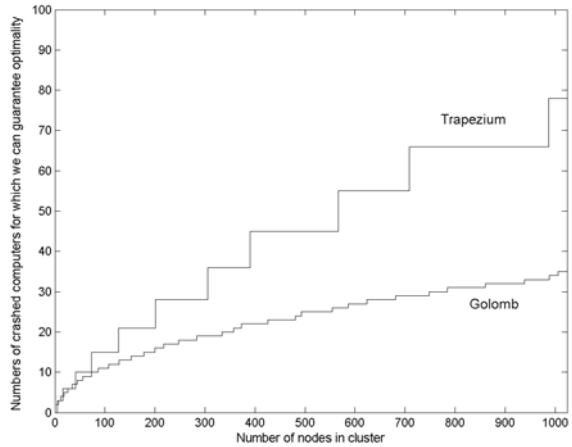


Fig. 7. The "performance" difference between the trapezium scheme ("trapezium") and OGRs scheme ('golomb')

The sum of the sequence	Examples of the trapezium sequences
5	**1,2**,2
16	**1,3,2**,6,1,3
41	**1,3,5,2**,6,6,6,3,3,6
73	**1,3,6,2,5**,13,1,9,2,3,8,5,6,3,6
127	**1,3,6,8,5,2**,20,6,5,1,10,12,2,4,3,5,11,5,10,1,7
201	**1,3,5,6,7,10,2**,24,3,11,2,20,2,2,8,6,2,19,2,6,6,2,13,10,2,14,6,7
305	**1,4,7,13,2,8,6,3**,28,14,3,1,17,3,18,10,2,7,9,3,18,9,7,9,3,9,18,3,8,8,4,13,2,14,3,18
390	**1,5,4,13,3,8,7,12,2**,36,1,5,4,13,3,8,7,12,2,36,1,5,4,13,3,8,7,12,2,36,1,5,4,13,3,8,7, 12,2,36,1,5,4,13,3
566	**1,3,9,15,5,14,7,10,6,2**,40,22,8,3,19,15,7,4,26,3,6,3,12,26,2,1,10,14,11,10,10,11,18, 2,4,14,17,4,9,15,20,1,10,6,7,15,23,8,8,15,1,21,1,10,2
708	**2,4,18,5,11,3,12,13,7,1,9**,48,2,4,18,5,11,3,12,13,7,1,9,48,2,4,18,5,11,3,12,13,7,1,9,4 8,2,4,18,5,11,3,12,13,7,1,9,48,2,4,18,5,11,3,12,13,7,1,9,48,2,4,18,5,11,3
987	**2,3,20,12,6,16,11,15,4,9,1,7**,69,2,24,5,19,1,24,9,31,2,7,1,16,18,16,5,16,3,30,9,7,1,1 4,44,3,10,17,7,11,6,12,13,6,21,21,1,28,1,20,4,17,7,1,5,19,18,16,5,8,13,30,10,6,26, 10,11,9,16,29,1,10,7,25,2,19,7
1175	**4,2,14,15,17,7,18,1,8,3,10,23,5**,56,4,2,14,15,17,7,18,1,8,3,10,23,5,56,4,2,14,15,17, 7,18,1,8,3,10,23,5,56,4,2,14,15,17,7,18,1,8,3,10,23,5,56,4,2,14,15,17,7,18,1,8,3,10, 23,5,56,4,2,14,15,17,7,18,1,8,3,10,23,5,56,4,2,14,15,17,7,18
1559	**4,16,10,27,2,3,14,24,11,12,13,8,1,6**,95,3,45,5,13,18,18,18,17,19,6,13,6,15,31,4,13,1 5,2,23,34,10,7,14,12,18,10,6,28,11,6,14,33,20,2,9,12,22,9,25,9,1,5,35,15,8,20,19,1,1 6,21,16,11,31,9,6,19,12,2,30,17,15,23,2,3,14,14,9,8,29,1,17,32,8,2,33,5,23,13,9,6, 21,18,22,8,23,5,14,18,7,22

Table 2. Examples of the trapezium sequences

6. Discussion and conclusions

In many cluster and distributed systems, the designer must provide a recovery scheme, which define how the workload should be redistributed when one or more computers break down. The goal is to keep the load as evenly distributed as possible, even when the most unfavorable combinations of computers break down.

We consider a cluster with n identical nodes, which under normal conditions execute one process each. All processes perform the same amount of work. Recovery schemes that guarantee optimal worst-case load distribution, when k computers have crashed are referred to as optimal recovery schemes for the values n and k.

The recovery schemes presented here can also be used in clusters with a number of independent processes on each computer. This is discussed in [11] and [12].

A contribution in this paper is that we have presented new recovery schemes, called trapezium recovery schemes, where the first part of the schemes is based on the known Golomb rulers, (i.e. the crash routes are disjoint) and the second part is constructed in a way, where the following crash routes have at least l unique values compared to previous crash routes. The Golomb recovery schemes are presented in [9]. The trapezium recovery schemes guarantee better performance than the Golomb schemes and are

simple to calculate. For small number of nodes in a cluster there is no big difference between these schemes. But for a larger number of nodes the behavior of the trapezium recovery scheme is much better than the Golomb schemes (which are the current state-of-the-art), e.g., for n = 201 the trapezium scheme guarantees optimal worst-case behavior for 28 crashes, while the Golomb scheme can only guarantee this for 16 crashes.

The goal of this paper was to find good recovery schemes, that are better than the already known and are easily to calculate also for large n. In previous work [8] we have found the best possible recovery schemes for any number of crashed nodes in a cluster. To find such a recovery scheme is a very computationally complex task. Due to the complexity of the problem we have only been able to present optimal recovery schemes for a maximum of 21 nodes in a cluster.

The trapezium recovery schemes can be immediately used in commercial cluster systems, e.g. when defining the list in Sun Cluster using the *scconf* command. The results can also be used when a number of external systems, e.g. telecommunication switching centers, send data to different nodes in a distributed system (or a cluster where the nodes have individual network addresses). In that case, the recovery lists are either implemented as alternative destinations in the external systems or at the communication protocol level, e.g. IP takeover [15].

7. References

1. Bloom, G. S., and Golomb, S. W., *Applications of Numbered, Undirected Graphs,* in Proceedings of the IEEE, Vol. 65, No. 4, April, 1977, pp. 562-571C
2. Cristian, F., *Understanding Fault Tolerant Distributed Systems*, Journal of the ACM, Vol 34, Issue 2, 1991
3. Dimitromanolakis, A., *Analysis of the Golomb Ruler and the Sidon Set Problems, and Determination of large, near-optimal Golomb Rulers,* Dept. of Electronic and Computer Engineering Technical University of Crete, June, 2002
4. Hewlett-Packard Company, *TruCluster Server - Cluster Highly Available Applications*, Hewlett-Packard Company, September 2002
5. Hewlett-Packard, *Managing MC / ServiceGuard*, Hewlett-Packard, March 2002
6. IBM, *HACMP, Concepts and Facilities Guide*, IBM, July 2002
7. Kameda, H., Fathy, E.-Z. S., Ryu, I., Li, J., *A performance Comparison of Dynamic vs. Static Load Balancing Policies in a Mainframe - Personal Computer Network Model,* Information: an International Journal, Vol. 5, No. 4, December 2002, pp. 431-446
8. Klonowska, K., Lennerstad, H., Lundberg, L., Svahnberg, C., *Optimal Recovery Schemes in Fault Tolerant Distributed Computing,* Acta Informatica, 41(6), 2005
9. Klonowska, K., Lundberg, L., and Lennerstad, H., *Using Golomb Rulers for Optimal Recovery Schemes in Fault Tolerant Distributed Computing,* in Proceedings of the 17th International Parallel & Distributed Processing Symposium IPDPS 2003, Nice, France, April 2003

10.Klonowska, K., Lundberg, L., Lennerstad, H., Svahnberg, C., *Using Modulo Rulers for Optimal Recovery Schemes in Distributed Computing*, in Proceedings of the 10th International Symposium PRDC 2004, Papeete, Tahiti, French Polynesia, March 2004
11.Lundberg, L., Häggander, D., Klonowska, K., and Svahnberg, C., *Recovery Schemes for High Availability and High Performance Distributed Real-Time Computing,* in Proceedings of the 17th International Parallel & Distributed Processing Symposium IPDPS 2003, Nice, France, April 2003
12.Lundberg, L., and Svahnberg, C., *Optimal Recovery Schemes for High-Availability Cluster and Distributed Computing*, *Journal of Parallel and Distributed Computing* 61(11), 2001, pp. 1680-1691
13.Krishna, C. M., Shin, K. G., *Real-Time Systems,* McGraw-Hill International Editions, Computer Science Series, 1997, ISBN 0-07-114243-6
14.Microsoft Corporation, *Server Clusters: Architecture Overview for Windows Server 2003*, Microsoft Corporation, March 2003
15.Pfister, G. F., *In Search of Clusters, The Ongoing Battle in Lowly Parallel Computing,* Prentice Hall PTR, 1998, ISBN 0-13-899709-8
16.Soliday, S. W., Homaifar, A., Lebby, G. L., *Genetic Algorithm Approach to the Search for Golomb Rulers,*in Proceedings of the International Conference on Genetic Algorithms, Pittsburg, PA, USA, pp. 528-535, 1995
17.Sun Microsystems, *Sun Cluster 3.0 Data Services Installation and Configuration Guide*, Sun Microsystems, 2000

Paper VI

Paper VI

Optimal Recovery Schemes in Fault Tolerant Distributed Computing

Kamilla Klonowska, Lars Lundberg, Håkan Lennerstad, Charlie Svahnberg
Acta Informatica, 41(6), 2005

Abstract

Clusters and distributed systems offer fault tolerance and high performance through load sharing. When all n computers are up and running, we would like the load to be evenly distributed among the computers. When one or more computers break down, the load on these computers must be redistributed to other computers in the system. The redistribution is determined by the recovery scheme. The recovery scheme is governed by a sequence of integers modulo n. Each sequence guarantees minimal load on the computer that has maximal load even when the most unfavorable combinations of computers go down. We calculate the best possible such recovery schemes for any number of crashed computers by an exhaustive search, where brute force testing is avoided by a mathematical reformulation of the problem and a branch-and-bound algorithm. The search nevertheless has a high complexity. Optimal sequences, and thus a corresponding optimal bound, are presented for a maximum of twenty one computers in the distributed system or cluster.

Keywords: distributed systems, recovery scheme, failure, load balance, integer sequences

1 Introduction

One way of obtaining high availability and fault tolerance is to execute an application on a cluster or distributed system. There is a primary computer that executes the application under normal conditions and a secondary computer that takes over when the primary computer breaks down. There may also be a third computer that takes over when the primary and secondary computers are both down, and so on. Another advantage of using distributed systems or clusters, besides fault tolerance, is load sharing between the computers. When all computers are up and running, we would like the load to be evenly distributed. The load on some computers will, however, increase when one or more computers are down, but also under these conditions we would like to distribute the load as evenly as possible on the remaining computers. The distribution of the load when a computer goes down is decided by the *recovery lists* of the processes running on the faulty computer. The set of all recovery lists is referred to as the *recovery scheme*. Hence the load distribution is completely determined by the recovery scheme for any set of computers that are down. The problem of finding optimal (or even reasonably good) recovery schemes has been studied in [16,17]. In the present paper we calculate the best possible such recovery scheme for all n.

Most cluster vendors support this kind of error recovery, e.g. the nodelist in Sun Cluster [27], the priority list in MC/ServiceGuard (HP) [12], the placement policy in TruCluster (DEC) [11], cascading resource group in HACMP (IBM) [14], and node preference list in Windows Server 2003 clusters (Microsoft, earlier called MSCS) [22].

In [5] fault tolerance for distributed computing is discussed from a wide viewpoint. Load balancing and availability are specially important in fault tolerant distributed systems, where it is difficult to predict on which computer the processes should be executed. This problem is NP-complete for the larger number of computers [30].

Krishna and Shin define the fault tolerance as an ability of a system to respond gracefully to an unexpected hardware or software failure [18]. Therefore, many fault-tolerant computer systems mirror all operations, e.g. every operation is performed on two or more duplicate systems in that sense, that if one fails the other can take over its job. This technique is also used in the clusters [23].

One can handle this problem dynamically, where transfer decisions depend on the actual current system state. The other way is to use static policies that are generally based on the information of the average behavior of the system. Here, the transfer decisions are independent of the actual current system state. That makes them less complex and more predictable than dynamic policies [15].

Besides hardware failures, the intermittent failures can occur due to software events. In [29] a “two-level” recovery scheme is presented and evaluated. The recovery scheme has been implemented on a cluster. The authors evaluate the impact of checkpoint latency on the performance of the recovery scheme. For transaction-oriented systems, in [7], Gelenbe has proposed “multiple check pointing” approach that is similar to the multi-level recovery scheme presented in [29]. A mathematical model of transaction-oriented system under intermittent failures is proposed in [9]. Here, the system is assumed to operate in a standard checkpoint-rollback-recovery scheme. In

[3] and [8] the authors propose several algorithms which can detect tasks failures and restart failed tasks. They analyze the behavior of parallel programs represented by a random task graph in a multiprocessor environment. However, all these algorithms act dynamically.

Our results are also valid when there are n external systems feeding data into the cluster, e.g., one telecommunication switching center feeding data into each of the n computer in the cluster. If a computer breaks down, the switching center must send its data to some other computer in the cluster; i.e., there has to be a recovery list associated with each switching center. The fail-over order can alternatively be handled at the communication protocol level, e.g. IP takeover [24]. In that case, redirecting the communication to another computer is transparent to the switching center.

Many cluster vendors offer not only the user defined recovery lists considered here, but also dynamic load balancing schemes when a node goes down. The run-time overhead and unpredictable worst-case behavior of such dynamic schemes may, however, be unattractive in many applications. More importantly, external systems do generally not have access to the internal state of the cluster, and can thus not use dynamic load balancing schemes when they need to select a new node when the primary destination for their output is not responding.

The problem area and formulation are presented in Sections 2 and 4. Our previous research on this problem is described in Section 3. In Section 5 we present main results. The tightness of the lower bound presented in [16,17] is investigated in Section 6. In the same section we describe the algorithm of finding optimal sequences. The conclusions are presented in Section 7.

2 Problem area

In this section we present a model explaining terminology used in this paper.

In Appendix A we discuss the case when there are a number of independent processes on each computer. However, until then we assume that the work performed by each of the n computers must be moved as one atomic unit. This is a very common scenario including cases when:

- the work performed by a computer is generated from one external system;
- there is one network address for each computer and we use IP takeover (or similar techniques);
- all the work performed by a computer is done by one process or a group of related processes that share local resources and thus must be moved as one unit.

Some computer networks, such as hypercubes [1,10], restricts the communication paths in the system, so that a message from computer A to computer B may in some cases have to be routed through a computer C (or D,...). However, many bus-based systems, e.g. systems using an ethernet [13], allow direct communication between any pair of nodes in the cluster [26]. We assume that it is possible to restart a job on any computer in the cluster (or redirect a communication channel in the case of external systems feeding data to the cluster).

We consider a fully connected distributed system or a cluster with n identical computers. During normal operation there is one process on each computer. The work is evenly split between these n processes. Each process has a *recovery list*, which determines at which computer the process is to be restarted if the current computer breaks down. If that computer is down, the process is transferred to the next computer in the list, and so on. We assume that processes are moved back as soon as a computer comes back up again. In most cluster systems this can be configured by the user [12,27,28], i.e. in some cases one may not want automatic relocation of processes when a faulty computer comes back up again.

The process at computer i, $0 \le i \le n-1$, is transferred to computer $x_{i,1}$ if computer i goes down. If also computer $x_{i,1}$ goes down, or is down before computer i goes down, the same process is transferred to computer $x_{i,2}$ and so on. Generally, if all the computers $x_{i,1}, x_{i,2}, \ldots, x_{i,j-1}$ are down, and computer $x_{i,j}$ is up, the process that was initially at computer i is at computer $x_{i,j}$. Figure 1 shows an example of relocation of the process from computer 0. The computers 0, 1 and 6 are down. Consequently, the process from computer 0 is on the computer $x_{0,3} = 3$.

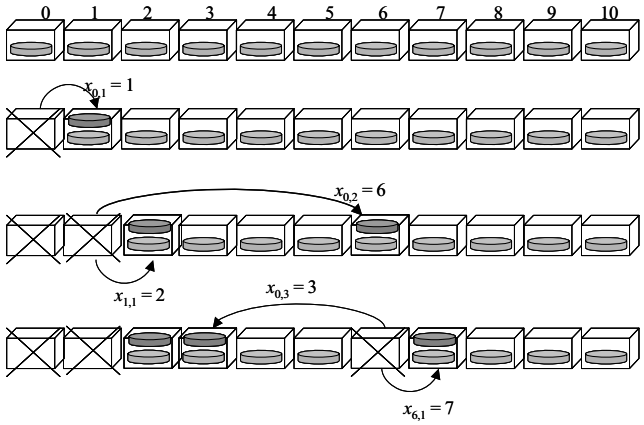


Fig. 1. A transfer of a process from computer 0

We call the sequence $x_{i,1}, x_{i,2}, \ldots, x_{i,j-1}$ a *recovery list* for computer i to $x_{i,j}$. In our example the chain for computer 0 starts with <1,6,3,...>.

A set of n recovery lists is called a *recovery scheme*. A general recovery scheme R has thus the following appearance:

$0 : x_{0,1}, x_{0,2}, \ldots, x_{0,n-1}$

$1 : x_{1,1}, x_{1,2}, \ldots, x_{1,n-1}$

...

n-1 : $x_{n-1,1}, x_{n-1,2}, \ldots, x_{n-1,n-1}$

In order to avoid directing a process to a computer that we know has crashed, all entries in a recovery list need to be distinct.

We mainly consider *regular* recovery schemes. Then all lists have the same structure, given by a vector $R = \langle 0, R_1, R_2, \ldots, R_{n-1} \rangle$, as follows:

$0 : \langle R_1, R_2, \ldots, R_{n-1} \rangle$

$1 : \langle R_1 + 1, R_2 + 1, \ldots, R_{n-1} + 1 \rangle$

...

n-1 : $\langle R_1 + n - 1, R_2 + n - 1, \ldots, R_{n-1} + n - 1 \rangle$ where all numbers are calculated mod n.

We denote the first computer by $R_0 = 0$ and all computers $x_{0,i}$ by R_i for $i = 1, \ldots, n-1$. Note that a process initially at computer 0 is at computer R_j after j crashes.

For example, for $n = 6$ we have following regular recovery schemes given by the vector $R = \langle 0, 1, 3, 5, 4, 2 \rangle$:

0: 1, 3, 5, 4, 2
1: 2, 4, 0, 5, 3
2: 3, 5, 1, 0, 4
3: 4, 0, 2, 1, 5
4: 5, 1, 3, 2, 0
5: 0, 2, 4, 3, 1.

The set of regular recovery schemes with length n (n computers) is denoted by $R(n)$. There are clearly $(n-1)!$ different such sequences with n computers, so $|R(n)| = (n-1)!$.

3 Previous research

In previous work we have presented algorithms that give recovery schemes which are optimal for a number of crashed computers.

In [20] the problem of finding a recovery scheme that can guarantee optimal worst-case load distribution when at most x computers are down is presented for the first time. The schemes should have as large x as possible. We present and prove the *Log* algorithm, which generates the recovery schemes that guarantee optimality where at most $\lfloor \log_2 n \rfloor$ computers go down. Here *optimal* means that the maximal number of processes on the same computer after k crashes is $BV(k)$, where the function $BV(k)$ provides a lower bound for any static recovery scheme. BV is by definition increasing and contains exactly k entries that equals k for all $k \geq 2$. The j:th entry in the vector BV equals $\lfloor \sqrt{2(j+1)} + 1/2 \rfloor$. Hence,

$BV(p) = \langle 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 5, 6, \ldots \rangle$.

In [16] we present two other algorithms, *Greedy* algorithm and *the Golomb* scheme. These algorithms generate the recovery schemes that give better optimality than *Log* algorithm, (i.e. better load balancing). The Greedy algorithm is based on the mathematical problem of finding the sequence of positive integers such that all sum of subsequences are unique and minimal. It is easy to calculate the Greedy algorithm even for large *n.* The recovery list for computer number zero consists of two parts. The first part is a sequence from the Greedy algorithm, and the second part is filled with the remaining numbers. The other lists are obtain from the first list by adding one in the modulo sense. The recovery scheme is then a scheme consisting of all recovery lists.

The Golomb scheme is a special case of the Greedy algorithm. The name of this algorithm comes from the Golomb ruler [2,6]. The mathematical formulation has been changed into looking after the longest sequence of positive integers such that the sum of the sequence is smaller than or equal to n and all sums of subsequences (including subsequences of length one) are unique, where n is the number of computers in a cluster. The Golomb recovery lists are formulated such as the Greedy recovery lists: for k computers in a cluster we build a recovery list for computer zero by using the known Golomb ruler with the sum less or equal to k, and the rest of the recovery list is filled with the remaining numbers up to k-1, e.g. for k=12 we have the list <1,4,9,11,2,3,5,6,7,8,10>. All other recovery lists are obtain from the first one by adding one in the modulo sense.

The Golomb schemes give optimality for a larger number of computers down than the Greedy scheme. However, finding (and proving) optimal Golomb schemes becomes exponentially more difficult as the number of computers (n) increases [32,33]. Therefore, for large n we can easily calculate a sequence with distinct partial sums with the Greedy algorithm, where no Golomb sequences are known.

In [17] we optimize the problem by taking into account the *wrap-around* scenario: process being sent backwards or passing by the initial computer. This corresponds to the new mathematical problem of finding the longest sequence of positive integers for which the sum of all subsequences are unique modulo n. This mathematical formulation of the computer science problem gives new more powerful recovery schemes, called *modulo schemes*, that are optimal for a larger number of crashed computers in the original computer science problem. I.e. these results represent state-of-the-art in the field. The modulo recovery scheme is formulated such as the Greedy and Golomb recovery schemes.

Characteristics and discussion of some examples of schemes are presented in Section 5.4.

4 Problem formulation

In this section we first define a load on the most heavily loaded computer after p crashes. Then we are looking after optimal recovery schemes by choosing the schemes that minimize this load.

4.1 Worst case load

Let $L(n, p, J, R)$ $(n > p)$ denote the load on the most heavily loaded computer when p computers $J = \{j_1, \ldots, j_p\}$ are down, and when using the recovery scheme R.

Let $L(n, p, R) = \max_J L(n, p, J, R)$ for all vectors $J = \{j_1, \ldots, j_p\}$, i.e. for all combinations of p computers being down. $L(n, p, R)$ defines the worst-case behavior after p crashes. We often regard $L(n, p, R)$ as a sequence in the variable p and call it a *load sequence*.

For example, if $p = 3$ we may have: $L(n, \{1, 2, 3\}, R) = \langle 2, 2, 3 \rangle$, for $(n > p)$ and the recovery scheme R. It means, that if one computer is down, the load on the most loaded computer is equal to 2, when two computers are down, then the load is also equal to 2, but when three computers are down, then the load on the most loaded computer equals 3 (in the worst-case scenario).

4.2 Consecutive load balancing

When load balancing, we are giving strict priority to a small number of computers down compared to a large number. We call this *consecutive load balancing*, which we next formulate mathematically.

We define the set $R(n, p)$ of recovery schemes that minimizes the maximal load for $1, 2, \ldots, p$ computers down in formula:

$$R(n, p) = \left\{ R \in R(n) | L(n, i, R) = \min_{P \in R(n)} L(n, i, P), i = 1, \ldots, p \right\}.$$

Since $L(n, 1, R) = 2$ for any scheme R, we have $R(n, 1) = R(n)$, i.e. all schemes minimize the maximal load when only one computer is down. Then the maximal load on the most loaded computer equals two. When two computers are down, the load on the most loaded computer may be two or three. Because we are looking after minimum, the schemes that give load three are canceled, and $R(n, 2)$ contains all recovery schemes where this load is two.

Now, *among the schemes in* $R(n, 2)$ we choose the schemes that minimize the load on the most loaded computer when three computers are down, to form the set $R(n, 3)$. Thus by definition: $R(n, 3) \subset R(n, 2)$.

Thus, the set $R(n, 3)$ contains the recovery schemes that minimize the load where we give priority to the performance when two computers are down over three computers down. This is consecutive load balancing, which we next define in general.

In general we form $R(n, p + 1)$ from $R(n, p)$ by choosing the recovery sequences in $R(n, p)$ which has minimal $L(n, p + 1, R)$. Hence, $R(n, p)$ is non-empty for all

$p = 1, \ldots, n-1$, and we have the inclusion $R(n, p+1) \subset R(n, p)$ for all $p = 1, \ldots, n-1$.

Note that there might exist recovery schemes $R \notin R(n, p)$ so that $L(n, p, R) < L(n, p, R')$ for all $R' \in R(n, p)$. Then $L(n, i, R) > L(n, i, R')$ for some $R' \in R(n, p)$ and $i < p$. This is a consequence of the consecutive load balancing.

The sequence $L(n, 1, R), L(n, 2, R), \ldots, L(n, n-1, R)$ is identical for all $R \in R(n, n-1)$. This optimal load sequence is denoted by *SV*.

The table below shows some examples of the sequences modulo 6. There are 5! possible sequences.

the sequences	$p = 1$	$p = 2$	$p = 3$	$p = 4$	$p = 5$
$R1 = \{0,1,3,5,4,2\}$	2	2	3	3	6
$R2 = \{0,1,3,5,2,4\}$	2	2	3	4	6
$R3 = \{0,3,1,4,2,5\}$	2	2	4	4	6
$R4 = \{0,2,4,1,5,3\}$	2	3	3	3	6
$R5 = \{0,2,4,1,3,5\}$	2	3	3	4	6

Table 1. Examples of the sequences mod 6

These sequences are accepted when one computer is down (belongs to $R(6,1)$). For two crashed computers the sequences $R4$ and $R5$ are excluded, the remaining belong to $R(6,2)$. For three crashes $R3$ is excluded, and for four crashes $R2$ is excluded. In consequence, the optimal sequence in this example is $R1$. If we look only on the sequences $R3$ and $R4$ then we have that $R4 \notin R(6, 4)$ in spite of the fact that $L(6, 4, R4) < L(6, 4, R3)$, because for $p = 2$ we have $L(6, 2, R4) > L(6, 2, R3)$.

4.3 Optimal recovery schemes

The sequences in $R(n, p)$ are sometimes referred to as p-optimal sequences.

Definition 1. The set $R^*(n) = R(n, n-1)$ *is the set of* ***consecutively optimal*** *recovery schemes.*

The main aim of this paper is to describe the sets $R^*(n)$ for all n and its load sequence *SV*. By comparison with the previous work in this field as described initially in Section 3, such a description would exhaust this problem formulation.

Let $MV = \max\left(BV(j), \left\lceil \frac{n}{n-j} \right\rceil\right)$. For all $R \in R(n)$ it is proven in [20] that $MV \le L(n, j, R)$ for all $j = 1, 2, \ldots, p$.

It is not known if the lower bound *MV* is tight. In this paper we investigate the tightness of the bound *MV* by the optimal load sequence *SV* of $R^*(n)$. We present an algo-

rithm with which we calculate the optimal bound *SV*. In many instances, when we have a larger number of crashed computers, *SV* do not coincide with *MV*.

5 Main results

In this section we present a method to reach the maximum load on the worst-case behavior after p crashes.

5.1 Computer chains

Calculation of optimal sequences becomes feasible by the following formulation of the problem in chains and unions of chains. To study the load balancing, we switch perspective to the point of view of a computer y that receives processes from several crashing computers.

A *chain* is a subsequence $x_{i,1}, x_{i,2}, \ldots, x_{i,j}$ of a recovery list $L_i(y)$ that end at computer y, i.e. $y = x_{i,j}$. It thus represents the path that the i:th process (the process on the i:th computer) takes to end up at computer y. The variable j denotes the number of crash steps to y. If $y = x_{i,j}$, we say that process i has *distance* j to computer y. Since the entries $x_{i,j}$ are distinct for each fixed i in each regular recovery list, there exists exactly one process with distance j, for all $1 \le j < n$. We may then order the chains by increasing distance to y:

$L_1 = \{y_{1,1}\}$,

$L_2 = \{y_{2,1}, y_{2,2}\}$,

...

$L_{n-1} = \{y_{n-1,1}, y_{n-1,2}, \ldots, y_{n-1,n-1}\}$.

In this section all recovery schemes and chains are counted in modulo sense.

There can not be two chains of the same length, since that would imply that the processes were originated at the same computer.

We summarize: Each chain represents a way for a process to be transferred to the computer y by subsequent computer crashes. Each entry in the chains represents a computer that goes down.

A regular recovery list gives the chains:

$\{y - R_1\} \pmod{n}$,

$\{y - R_2, y - (R_2 - R_1)\} \pmod{n}$,

$\{y - R_3, y - (R_3 - R_1), y - (R_3 - R_2)\} \pmod{n}$,

...

$\{y - R_{n-1}, y - (R_{n-1} - R_1), \ldots, y - (R_{n-1} - R_{n-2})\} \pmod{n}$.

We may alternatively describe a regular recovery scheme $R = \{0, R_1, \ldots, R_{n-1}\}$ by the differences $r = \{r_1, r_2, \ldots, r_{n-1}\}$. We define

$r_1 = R_1 \pmod n$,

$r_2 = R_2 - R_1 \pmod n$,

$r_3 = R_3 - R_2 \pmod n$,

...

$r_{n-1} = R_{n-2} - R_{n-1} \pmod n$.

and thus

$R_1 = r_1 \pmod n$,

$R_2 = r_1 + r_2 \pmod n$,

...

$R_{n-1} = r_1 + r_2 + \ldots + r_{n-1} \pmod n$.

The regular recovery schemes are then given by:

0: $\{r_1, r_1 + r_2, r_1 + r_2 + r_3, \ldots, r_1 + \ldots + r_{n-1}\} \pmod n$,

1: $\{r_1 + 1, r_1 + r_2 + 1, \ldots, r_1 + \ldots + r_{n-1} + 1\} \pmod n$,

...

n-1: $\{r_1 + n - 1, \ldots, r_1 + \ldots + r_{n-1} + n - 1\} \pmod n$.

The chains of computers ending at y are as follows:

$\{y - r_1\} \pmod n$,

$\{y - (r_1 + r_2), y - r_2\} \pmod n$,

$\{y - (r_1 + r_2 + r_3), y - (r_2 + r_3), y - r_3\} \pmod n$,

...

$\{y - (r_1 + \ldots + r_{n-1}), \ldots, y - (r_{n-2} + r_{n-1}), y - r_{n-1}\} \pmod n$.

The properties we are interested in are more easily studied in the following version:

$C_1 = \{r_1\} \pmod n$,

$C_2 = \{r_1 + r_2, r_2\} \pmod n$,

$C_3 = \{r_1 + r_2 + r_3, r_2 + r_3, r_3\} \pmod n$,

...

$C_{n-1} = \{r_1 + \ldots + r_{n-1}, \ldots, r_{n-2} + r_{n-1}, r_{n-1}\} \pmod n$.

5.2 An example of a sequence

For $n = 7$ a regular recovery list for computer 0 is: $\{R_0, R_1, R_2, \ldots, R_6\} = \{0, 1, 4, 6, 2, 3, 5\}$, described by differences (mod 7) r: $\{r_1, r_2, \ldots, r_{n-1}\} = \{1, 3, 2, 3, 1, 2\}$.

The chains of computers ending at y are then as follows: $\{y-1\} \bmod 7$, $\{y-4, y-3\} \bmod 7$, $\{y-6, y-5, y-2\} \bmod 7$, and so on. The studied sequences are: $C_1 = \{1\}$, $C_2 = \{4, 3\}$, $C_3 = \{6, 5, 2\}$ and so on. Hence, we present the differences diagonally from down to up in a triangle on Fig. 2.

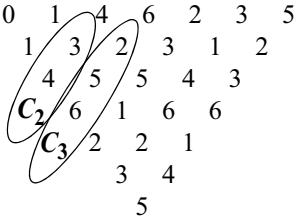


Fig. 2. The difference triangle for $n = 7$.

5.3 Normal sequences

The requirement that the entries in $R = \{0, R_1, \ldots, R_{n-1}\}$ are distinct transforms for the differences $r = \{r_1, r_2, \ldots, r_{n-1}\}$ into the requirement that all consecutive partials sums are non-zero (mod n):

Definition 2. A sequence $r = \{r_1, r_2, \ldots, r_{n-1}\}$ *is* ***normal*** *if* $\sum_{i=a}^{b} r_i \neq 0 (\bmod n)$ *for all* $1 \le a \le b \le n-1$.

Lemma 1: R has distinct entries $\Leftrightarrow$ r is normal.

Proof: We have $R_j = R_k$, $0 \le k \le j \le n-1$, if and only if $\sum_{i=1}^{j} r_i = \sum_{i=1}^{k} r_i (\bmod n)$, which is true if and only if $\sum_{i=k+1}^{j} r_i = 0 (\bmod n)$. The lemma is proved.

■

Hence, if any consecutive partial sum is n, then $R_i = R_j$ for some $i \neq j$. Note also that the modulus is one more than the length of the difference sequence. For example sequences of 1:s only are normal, $r = \{1, 1, 1, \ldots\}$, corresponding to $R = \{0, 1, 2, \ldots\}$. As we have noted before, there are $(n-1)!$ different normal sequences of length $n-1$. Only normal sequences give useful recovery schemes.

5.4 Examples: Golomb, greedy and modulo sequences

As we will see below, for optimality in terms of load balancing, we will need that as many as possible of the partial sums $\sum_{i=a}^{b} r_i$ are not only non-zero, but also distinct. A *Golomb sequence* is a sequence where all consecutive partial sums are distinct, and, preferably, the total sum is minimal (see [16]). Golomb sequences are however not constructed for the modular scenario which we consider in this paper. Furthermore, in this scenario the minimal sum is irrelevant. Golomb sequences are presented here since they are important in a neighboring scenario, and as examples of recovery schemes.

The first Golomb sequences are

$g^3 = \{1, 2\}$

$g^4 = \{1, 3, 2\}$

$g^5 = \{1, 3, 5, 2\}$

$g^6 = \{1, 3, 6, 5, 2\}$

$g^7 = \{1, 3, 6, 8, 5, 2\}$

with corresponding sum sequences:

$G^3 = \{0, 1, 3\}$

$G^4 = \{0, 1, 4, 6\}$

$G^5 = \{0, 1, 4, 9, 11\}$

$G^6 = \{0, 1, 4, 10, 15, 17\}$

$G^7 = \{0, 1, 4, 10, 18, 23, 25\}$,

taken modulo the length gives the sequences

$\{0, 1, 0\} \pmod 3$

$\{0, 1, 0, 2\} \pmod 4$

$\{0, 1, 4, 4, 1\} \pmod 5$

$\{0, 1, 4, 4, 3, 5\} \pmod 6$

$\{0, 1, 4, 3, 4, 2, 4\} \pmod 7$.

Hence, none of these Golomb sequences are normal (the sum sequences do not have distinct entries). As remarked before, Golomb sequences are not constructed for the modular scenario of this paper. However these sequences can be completed into nor-

mal sequences by adding computers and thus increasing the length and the modulo, for example as follows:

C^3 = {0, 1, 3, 2} mod 4

C^4 = {0, 1, 4, 6, 2, 3, 5} mod 7

C^5 = {0, 1, 4, 3, 5, 2} mod 6

C^6 = {0, 1, 4, 10, 3, 5, 2, 6, 7, 8, 9, 11} mod 12

C^7 = {0, 1, 4, 10, 2, 7, 9, 3, 5, 6, 8, 11, 12, 13, 14, 15} mod 16.

We mention another sequence that has distinct partial sums {1, 2, 4, 8, 16, 32,...} - the Log sequence. This sequence is not very dense. The last entry is large, hence it needs to be completed by many entries possibly giving a large number of non-distinct partial sums. Therefore we often want to minimize the size of the last entry.

Another construction of a sequence with distinct partial sums is the greedy sequence (see [19]). This is rapid to generate and is more dense than the Log sequence. Also for these sequences we apply the procedure of filling the sequence with remaining computers. We start by defining G_0, (i.e. the recovery list for process zero): $g(x) = \min\{j \in N_+\}$, such that for all a_1, a_2, b_1, b_2 that fulfill the conditions

$(1 \le a_1 \le b_1 \le x)$, $(1 \le a_2 \le b_2 \le x)$ and $((a_1 \ne a_2)$ or $(b_1 \ne b_2))$

we get $\sum_{l=a_1}^{b_1} g(l) \ne \sum_{l=a_2}^{b_2} g(l)$.

If $\sum_{l=1}^{x} g(l) < n$, then $G_0(x) = \sum_{l=1}^{x} g(l)$, else

$G_0(x) = \min\{j \in \mathrm{N}_+ - \{G_0((1), \ldots, G_0(x-1))\}\}$.

From the definition above we see that for large values of n (i.e. $n > 289$) the first 16 elements in G_0 for the greedy recovery scheme are:

1,3,7,12,20,30,44,65,80,96,122,147,181,203,251,289.

Then for $n = 16$, G_0= {1,3,7,12,2,4,5,6, 8,9,10,11,13,14,15}.

The modulo sequences ([17]) are, like Golomb sequences, different for different number of computers - they do not build upon the same initial values. The sum of elements is insignificant in this kind of sequence.

Examples of the first modulo sequences are:

m^3 = {1, 2} mod 3

m^4 = {1, 4, 6} mod 7

m^5 = {1, 6, 3, 10} mod 11

m^6 = {1, 3, 7, 17, 12} mod 18

m^7 = {1, 3, 23, 7, 17, 12} mod 24.

5.5 Load balancing by chains

As discussed before, each sequence $(r_1, \ldots, r_{n-1})$ defines $n-1$ chains: $C_i = \{r_1 + \ldots + r_i, r_2 + \ldots + r_i, \ldots, r_i\}$, $i = 1, \ldots, n-1$.

Let us consider the most loaded computer. If we, for example, pick three chains C_{i_1}, C_{i_2}, and C_{i_3}, it will require $\left|C_{i_1} \cup C_{i_2} \cup C_{i_3}\right|$ crashes to obtain the processes $y_{i_1,1}$, $y_{i_2,1}$ and $y_{i_3,1}$ on computer y (see Section 5.1), by crashing all computers in the lists C_{i_1}, C_{i_2}, and C_{i_3}.

Obviously, if the lists are not disjoint, so that $\left|C_{i_1} \cup C_{i_2} \cup C_{i_3}\right|$ is not a disjoint union, there are some computers whose crash give multiple effect in transferring processes to *y,* and the number of crashes needed is smaller. Thus, after $\left|C_{i_1} \cup C_{i_2} \cup C_{i_3}\right|$ crashes we obtain $1 + \left|\{i_1, i_2, i_3\}\right| = 4$ processes on *y.*

In general we have the following:

Theorem 1: Let $I = \{i_1, \ldots, i_s\}$ (for $s > 0$) be a certain set of processes which all are transferred to computer *y* if all computers in $\bigcup_{i \in I} C_i$ go down. For a recovery scheme *R* and if *p* crashes occur, we have $L(n, p, R) = 1 + \max_i \left\{ i = |I|; \forall I; \left|\bigcup_{i \in I} C_i\right| \le p \right\}$.

Proof: In general, the set $\bigcup_{i \in I} C_i$ corresponds to the case that $\left|\bigcup_{i \in I} C_i\right| = p$ computers go down. In order to compute the worst case we have to investigate all unions $\bigcup_{i \in I} C_i$ with $\left|\bigcup_{i \in I} C_i\right| \le p$. Then we have $1 + s$ processes on the same computer.

■

We next rephrase the problem slightly for the sake of the algorithm.

There are in total $2^{n-1} - 1$ possible unions of the sets $C_1, \ldots, C_{n-1}$. We denote the unions by $U_j = \bigcup_{i \in I} C_i$, where $j = 1, \ldots, 2^{n-1}$. As described, each union U_j has two properties which are interesting for our purposes: the number of sets s_j, and the number of distinct elements p_j. The number s_j is sometimes called the *depth* of the union

U_j, and the number p_j is called the *size* of U_j. We may reformulate the definition $L(n, p, R) = 1 + \max_i \left\{ i = |I|;\ \forall I;\ \left| \bigcup_{i \in I} C_i \right| \le p \right\}$ in terms of size and number of sets:

$L(n, p, R) = \{\max(1 + s_j);\ \forall U_j;\ p_j \le p\}$.

The $2^{n-1} - 1$ unions $\bigcup_{i \in I} C_i$ are generated from $C_1, \ldots, C_{n-1}$ by successively adding sets to previously generated unions. We can arrange the $(n-1)!$ sequences $(0, R_1, \ldots, R_{n-1})$ together with all possible subsequences of shorter length in a rooted tree, where two sequences are related if they are identical except that the last entry is missing in one of them. Hence, $(0, R_1, \ldots, R_k)$ and $(0, P_1, \ldots, P_l)$ are related if $R_i = P_i$ for all $i = 1, \ldots, \min(k, l)$ and $|k - l| = 1$. The empty sequence is the root in the tree, and all recovery schemes are the leaves. We call this tree the *sequence tree* (see Fig. 3).

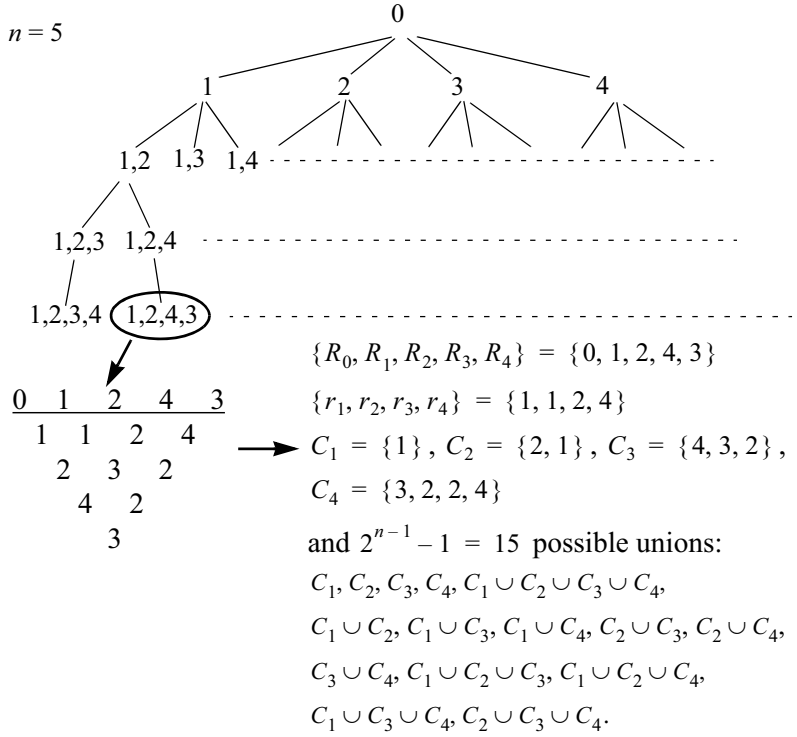


Fig. 3. Example of a sequence tree for $n = 5$

In principle, we thus need to investigate all $2^{n-1} - 1$ possible unions for each of the $(n-1)!$ sequences. However, there are several ways to exclude large sets of sequences and unions. The following lemma gives a condition with low complexity that categorizes a large class of sequences as non-optimal. The lemma first requires a definition.

Definition 3. We say that BV is tight up to q if there is a recovery scheme R so that $L(n, i, R) = BV(i)$ *for* $i = 1, \ldots, q$.

Lemma 2: Suppose that *BV* is tight up to *q*. Then all partial sums of *r* up to $BV(q)$ need to be distinct if *R* is *n*-optimal recovery scheme. In other words, the numbers $\sum_{i=a}^{b} r_i$ for all *a* and *b*, $1 \le a \le b \le BV(q)$, are distinct.

Proof: Suppose the contradiction, i.e. that two partial sums are equal, say at row *i* and *j*. Assume that $i \ge j$. Then we would after $i(i+1)/2 - 1$ crashes obtain $i+1$ processes on the computer with heaviest load. This contradicts the optimality of *R*.

■

This means that we can start with the previously calculated modular sequences (see [17]). We say that such sequences satisfy the *modular condition*. If a modular sequence has distinct subsequences up to *m* numbers in the sequence, these *m* initial numbers can be used as the start of the sequence, and there remains $(n-m-1)!$ sequences to investigate for optimality.

6 Tightness of *MV*

We here investigate for which values of *n* and *q* the bound *MV* is tight. The tight bound is called *SV*. The tightness of *MV* may follow from a previously generated sequence. Thus, if we have one recovery scheme that follows *MV* up to *q*, we can discard all recovery schemes where not all partial sums up to *q* are distinct.

The following algorithm finds all consecutively optimal recovery schemes. If there is a previous sequence which has *p*-optimality, we immediately can cancel the investigation of a sequence which does not have *p*-optimality. Then also an entire branch of the sequence tree, the branch which has identical initial sequence, may be removed. This is a consequence of the consecutive load balancing.

Furthermore, by Lemma 2 and the modular condition, a large class of non-optimal sequences may be removed. If a sequence is found to be non-optimal, the search continues at a neighboring branch.

6.1 Algorithm

In the algorithm, the sequences A and B belong to the set of all possible permutations of integer numbers from 1 to n that fulfills *the modulo condition* (see Lemma 2).

Let $p_d(A) = \min \{p_j : \forall U_j\}$ for all unions U_j with depth d (the depth is the number of sets in the union).

1. Let A be a sequence that satisfies the modulo condition and calculate $p_d(A) < p_d(B)$ *for all* $d = 1,..., n - 1$*;*
2. **while** *(the set of remaining sequences satisfying the modulo condition is non-empty)* **do begin**
3. Let B be a sequence that satisfies the modulo conditions that not yet has been considered;
4. Let $d = 1$*;*
5. **while** $d < n$ **do begin**
a) **if** $p_d(A) < p_d(B)$ **then begin**
break and goto *2)*; // *B* is canceled
end;
b) **if** $p_d(A) = p_d(B)$ **then begin**
$d = d + 1$; // check the next depth
goto *5)*
if $p_d(A) > p_d(B)$ **then begin**
A = B; // *A* is canceled
calculate $p_d(A) < p_d(B)$ **for all** $d = 1,..., n - 1$;
break and goto *2)*;
end;
end;
end;
6. $SV = \langle p_1(A), p_2(A), ..., p_{n-1}(A) \rangle$*, where SV is the optimal sequence vector.*

The figures below (Fig. 4, 5 and 6) show the relation between the bound vector *MV* and the optimal vector *SV* for n equal to 10, 15 and 20. For a lower number of crashes, the bound vector *MV* is tight, but not for a larger number. For example, for $n = 10$ *MV* is tight up to seven crashes (Fig. 4). For eight crashes the difference between *MV* and *SV* is one. This means that it does not exist a recovery scheme for ten computers with behavior as *MV* for any number of crashes up to eight crashes.

In a distributed system with 15 computers (Fig. 5) the bound *MV* is tight up to eight crashes. Surprisingly, *MV* is not tight for nine crashes, but it is tight again for ten crashes.

For 20 computers in a distributed system the bound is tight up to thirteen crashes (Fig. 6). The bound *MV* is trivially tight when $n - 1$ computers break down ($d = n - 1$). Then all processes are at one computer.

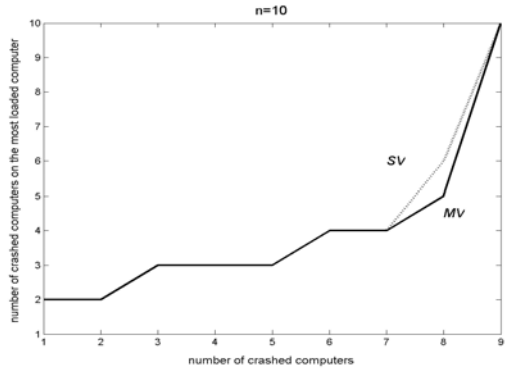


Fig. 4. *SV* vector versus *MV* vector for $n = 10$

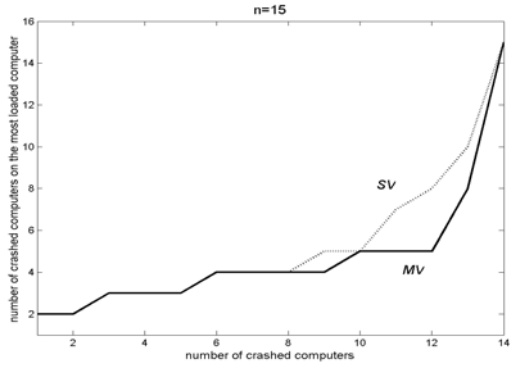


Fig. 5. *SV* vector versus *MV* vector for $n = 15$

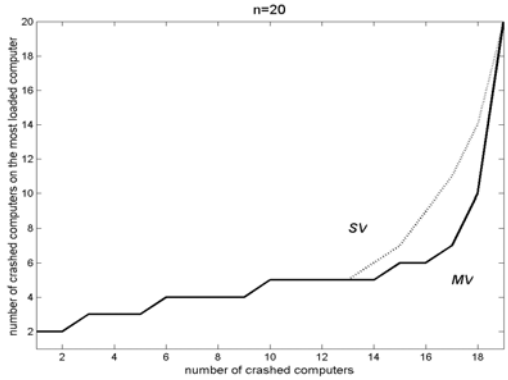


Fig. 6. *SV* vector versus *MV* vector for $n = 20$

To show the differences between *MV* and *SV* we first present a graph of bound vectors *MV* (see Fig. 7) for distributed systems consisting of up to 21 computers. The fig-

ure presents the load on the most loaded computer when q computers in a distributed system go down.

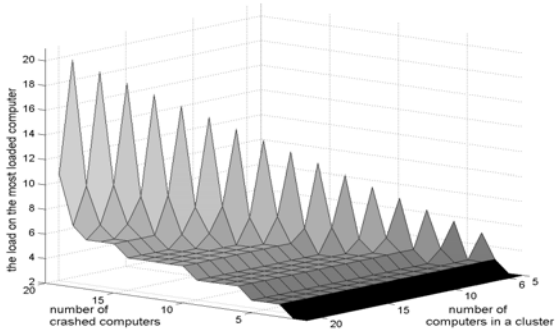


Fig. 7. Bound vectors *MV* up to 21 computers in a distributed system

In Fig. 8 we present the minimal load on the most loaded computer for various number of computers that break down, i.e. the vectors *SV*.

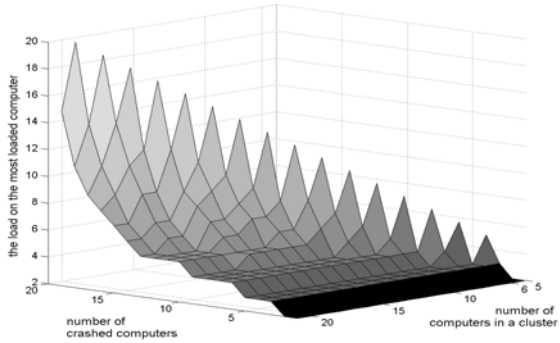


Fig. 8. Vectors *SV* up to 21 computers in a distributed system

Fig. 9 shows the performance difference between *SV* and *MV* for all n up to 21. For $n \leq 9$ the bound *MV* is tight for any number of crashes (we consider a maximum of n - 1 crashes). The max number of crashed computers where *SV* coincide with *MV* is presented in Tab. 2. The table shows also examples of the optimal sequences.

Fig. 10 shows to which extent *MV* is tight. In the grey area *MV* is tight, since *MV* = *SV* here. For larger values of crashed computers q, *MV* < *SV,* so *MV* is not tight. This area is the same as the area in Fig. 8 where the difference between *MV* and *SV* is zero. It follows that the area cannot be further extended. The line on the figure is a result of the best modulo sequences presented in [17].

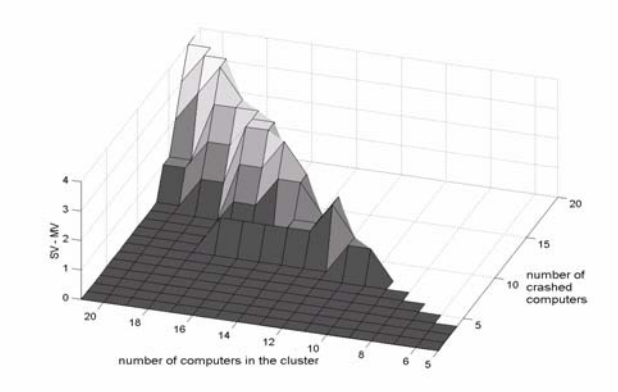


Fig. 9. Performance difference between *SV* and *MV* for *n* from 1 to 20

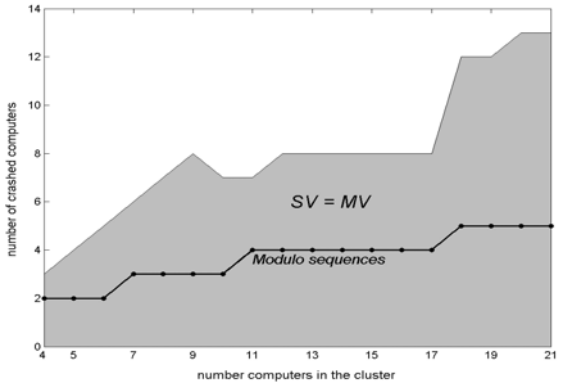


Fig. 10. Comparison of the *optimal recovery schemes* sequences with the *modulo sequences*

7 Discussion and conclusions

In many distributed systems and clusters, the designer must provide a recovery scheme. Such schemes define how the work load should be redistributed when one or more computers break down. The goal is to keep the load as evenly distributed as possible, even when the most unfavorable combinations of computers break down, i.e. we want to optimize the worst-case behavior. This is particularly important in real-time systems.

Hardware fault tolerance techniques can be divided into fault detection and fault tolerance [24]. Different techniques are used to examine fault detection and there is a broad body of research in that area (e.g. [4,21,25,31]). The fault detection problem is outside the scope of this paper.

We consider *n* identical computers that execute one process each. All processes perform the same amount of work. Recovery schemes that guarantee optimal worst-case

number of nodes	max number of crashed nodes where *SV* coincide with *MV*	Examples of optimal sequences
4	3	0,1,3,2
5	4	0,1,3,2,4
6	5	0,1,3,5,4,2
7	6	0,1,4,6,2,3,5
8	7	0,1,3,7,5,6,2,4
9	8	0,1,3,7,5,2,8,6,4
10	7	0,1,5,3,2,9,7,8,4,6
11	7	0,1,6,3,10,2,4,5,7,8,9
12	8	0,1,4,9,11,6,3,2,5,7,8,10
13	8	0,2,7,6,3,10,5,1,4,8,9,11,12
14	8	0,6,1,5,3,11,2,4,7,8,9,10,11,13
15	8	0,1,6,8,4,11,2,3,5,7,9,10,12,13,14
16	8	0,1,6,8,4,11,2,3,7,5,9,10,12,13,14,15
17	8	0,1,6,8,4,13,2,3,7,5,9,10,11,12,14,15,16
18	11	0,2,11,8,3,7,5,1,4,6,9,10,12,13,14,15,16,17
19	12	0,1,8,12,6,3,11,2,4,5,7,9,10,13,14,15,16,17,18
20	13	0,1,8,13,4,10,7,6,2,3,5,12,11,9,16,14,15,17,18,19
21	13	0,1,9,11,7,14,3,2,5,6,8,10,12,13,15,16,17,18,19,20

Table 2. Optimal sequences

load distribution, when x computers have crashed, are referred to as optimal recovery schemes for the values n and x. In this paper we present an algorithm that gives optimal recovery schemes for any number of crashes. The complexity of the algorithm is exponential.

Optimal recovery schemes are computed for up to 21 computers. We also compare the results to the previously presented lower bound *MV* [20]. The results show that for up to nine computers, $SV = MV$ for any number of computers down, so in this interval the bound *MV* is tight. For more than nine number of computers, *MV* is tight only when a few number of computers go down. Comparing with the results in [17], the performance difference expressed by the difference between *SV* and *MV* is larger than expected.

This paper provides an exhaustive solution of the pre-problem formulation of static fault tolerance. Optimal sequences are calculated, i.e. sequences that give optimal load distribution for any number of crashed computers. The problem formulation is very natural for the scenario of static recovery schemes.

In order to use these schemes one must have some mechanism for detecting when a computer fails. This can be done in different ways, e.g. using heartbeat or liveness chains [24] or by ordinary time-outs in the case of external system feeding data into a node in the cluster. Failure detection is often handled by the cluster software, and the optimal recovery schemes can in those cases be directly used in the nodelist in Sun Cluster [27], the priority list in MC/ServiceGuard (HP) [12], the placement policy in TruCluster (DEC) [11], cascading resource group in HACMP (IBM) [14], and the

node preference list in Windows Server 2003 clusters (Microsoft, earlier called MSCS) [22].

There is clearly a slowdown caused by the overhead associated with reloading a task (process) on a new node after a failure. The fact that we are using static recovery schemes (and not dynamic ones), can to some extent decrease this problem. When using static schemes we know on which computer the process should be restarted in recovery list by continuously propagating the current state of the process to the next node in the chain, and thus reducing the time for reloading the process in case of a failure. This is not possible when we use dynamic schemes since, in that case we do not know on which node the process should be restarted.

8 References

1. Bertsekas, D.P., Özveren, C., Stamoulis, G.D., Tsitsiklis, J.N., *Optimal Communication Algorithms for Hypercubes*, Journal of Parallel and Distributed Computing, 11, 1991, pp. 263-275
2. Bloom, G. S., Golomb, S. W., *Applications of Numbered, Undirected Graphs*, Proceedings of the IEEE, 65(4), 1977, pp. 562-571
3. Chabridon, S., Gelenbe, E.: *Failure Detection Algorithms for a Reliable Execution of Parallel Programs*, in Proceedings of 14th Symposium on Reliable Distributed Systems SRDS'14, Bad Neuenahr, Germany, September 1995
4. Chinchani, R., Upadhyaya, S., Kwiat, K., *A Tamper-Resistant Framework for Unambiguous Detection of Attacks in User Space Using Process Monitors*, in Proceedings of First IEEE International Workshop on Information Assurance IWIA'03, March 24-24, 2003, Darmstadt, Germany, pp. 25-36
5. Cristian, F., *Understanding Fault Tolerant Distributed System*s, Communication of the ACM, 34(2), 1991, pp. 56-78
6. Dimitromanolakis, A, *Analysis of the Golomb Ruler and the Sidon Set Problems, and Determination of large, near-optimal Golomb Rulers*, Dept. of Electronic and Computer Engineering Technical University of Crete, June, 2002
7. Gelenbe, E., *A Model for Roll-back Recovery With Multiple Checkpoints*, in Proceedings of 2nd International Conference on Software Engineering, San Francisco, California, US, October 1976, pp. 251-255
8. Gelenbe, E., Chabridon, S., *Dependable Execution of Distributed Programs*, Elsevier, Simulation Practice and Theory, 3(1), 1995, pp. 1-16
9. Gelenbe, E., Derochete, D., *Performance of Rollback Recovery Systems under Intermittent Failures*, Communication of the ACM, 21(6), 1978, pp. 493-499
10. Greenberg, D. S., Bhatt, S. N., *Routing multiple paths in hypercubes*, in Proceedings of Second Annual ACM Symposium on Parallel Algorithms and Architectures, Island of Crete, Greece, 1990, pp. 45 - 54
11. Hewlett-Packard Company, *TruCluster Server - Cluster Highly Available Applications*, Hewlett-Packard Company, September 2002
12. Hewlett-Packard, *Managing MC/ServiceGuard*, Hewlett-Packard, March 2002

13. Huang, C., McKinley, P. K., *Communication Issues in Parallel Computing across ATM Networks*, IEEE Parallel and Distributed Technology: Systems and Applications, 2(4), 1994, pp. 73-86
14. IBM. HACMP, *Concepts and Facilities Guide*, IBM, July 2002
15. Kameda, H., Fathy, E.-Z.S., Ryu, I., Li, J., *A performance Comparison of Dynamic vs. Static Load Balancing Policies in a Mainframe - Personal Computer Network Model*, Information: an International Journal, 5(4), 2002, pp. 431-446
16. Klonowska, K., Lundberg, L., Lennerstad, H.: *Using Golomb Rulers for Optimal Recovery Schemes in Fault Tolerant Distributed Computing*, in Proceedings of 17th International Parallel & Distributed Processing Symposium IPDPS 2003, Nice, France, April 2003, pp. 213, CD-ROM
17. Klonowska, K., Lundberg, L., Lennerstad, H., Svahnberg, C.: *Using Modulo Rulers for Optimal Recovery Schemes in Distributed Computing*, in Proceedings of 10th International Symposium PRDC 2004, Papeete, Tahiti, French Polynesia, March 2004, pp. 133-142
18. Krishna, C.M., Shin, K.G., *Real-Time Systems*, McGraw-Hill International Editions, Computer Science Series, 1997, ISBN 0-07-114243-6
19. Lundberg, L., Häggander, D., Klonowska, K., Svahnberg, C., *Recovery Schemes for High Availability and High Performance Distributed Real-Time Computing*, in Proceedings of 17th International Parallel & Distributed Processing Symposium IPDPS 2003, Nice, France, April 2003, pp. 122, CD-ROM
20. Lundberg, L., Svahnberg, C., *Optimal Recovery Schemes for High-Availability Cluster and Distributed Computing*, Journal of Parallel and Distributed Computing, 61(11), 2001, pp. 1680-1691
21. Mahmood, A., McCluskey, E.J., *Concurrent Error Detection Using Watchdog Processors - A Survey*, IEEE Transactions on Computers, 37(2), 1988, pp. 160-174
22. Microsoft Corporation, *Server Clusters: Architecture Overview for Windows Server 2003*, Microsoft Corporation, March 2003
23. Pande, S.S., Agrawal, D.P., Mauney, J., *A threshold scheduling strategy for Sisal on distributed memory machines*, Journal on Parallel and Distributed Computing, 21(2), 1994, pp. 223-236
24. Pfister, G.F., *In Search of Clusters*, Prentice-Hall, 1998
25. Reinhardt, S.K., Mukherjee, S.S., *Transient Fault Detection via Simultaneous Multithreading*, in Proceedings of 27th Annual International Symposium on Computer Architecture (ISCA), Vancouver, British Columbia, Canada, June, 2000
26. Stalling, W., *Computer Organization & Architecture,* Designing for Performance, Sixth Edition, Prentice Hall, 2003, ISBN 0-13-049307-4
27. Sun Microsystems. *Sun Cluster 3.0 Data Services Installation and Configuration Guide*, Sun Microsystems, 2000
28. TruCluster, *Systems Administration Guide*, Digital Equipment Corporation, http://www.unix.digital.com/faqs/publications/cluster_doc
29. Vaidya, N.H., *Another Two-Level Failure Recovery Scheme: Performance Impact of Checkpoint Placement and Checkpoint Latency*, Technical Report 94-068, Department of Computer Science, Texas A&M University, December 1994

30. Willebeek-LeMair, M., Reeves, A.P., *Strategies for Dynamic Load Balancing on Highly Parallel Computers*, IEEE Transactions on Parallel and Distributed Systems, 9(4), 1993, pp. 979-993
31. Young, M., Taylor, R.N., *Rethinking the Taxonomy of Fault Detection Techniques*, in Proceedings of International Conference Software Enginering (ICSE), ACM, May, 1989, pp. 53-62
32. http://www.distributed.net/ogr/index.html
33. http://www.research.ibm.com/people/s/shearer/grtab.html

Appendix A: Non-Atomic Loads

Hitherto, we have considered the case when all work performed by a computer must be moved as one atomic unit. Obviously, there could be more than one process on each computer and these processes may in some cases be redistributed independently of each other, and in that case there is one recovery list for each process. In this appendix we consider the case where there are p processes on each computer. We assume that the workload is evenly split between the processes.

Previous studies show that, if there is a large number of processes on each computer, the problem of finding optimal recovery schemes becomes less important [19]. The reason for this is that the intuitive solutions become better. Consequently, the importance of obtaining optimal recovery schemes is largest when p is small compared to n. In the main part of this paper, we have obtained results for $p = 1$. The techniques used for obtaining those results are also useful when we consider $p > 1$.

We now define bound vectors of type p. The bound vectors discussed in Section 4.3 were of type one, i.e. $p = 1$. A bound vector of type p is obtained in the following way (the first entry in the bound vector is entry 1):

1. $t = p$; $i = 1$; $j = 1$; $r = 1$; $v = 1$
2. **If** $r = 1$ **then** $t = t+1$; $r = j$; $v = v+1$ else $r = r$-1
3. Let entry i in the Bound vector of type p have the value t/p
4. $i = i+1$
5. **If** $v = p$ **and** $r = 1$ **then** $j = j+1$; $v = 0$
6. **Go to** 2 until the bound vector has the desired length.

A bound vector of type two looks like this (note that the load on each computer is normalized to one when all computers are up and running): {3/2, 4/2, 4/2, 5/2, 5/2, 6/2, 6/2, 6/2, 7/2, 7/2, 7/2, 8/2, 8/2, 8/2, 8/2, 9/2, 9/2, 9/2, 9/2, ...}

A bound vector of type three looks like this: {4/3, 5/3, 6/3, 6/3, 7/3, 7/3, 8/3, 8/3, 9/3, 9/3, 9/3, 10/3, 10/3, 10/3, …}

A bound vector of type four looks like this: {5/4, 6/4, 7/4, 8/4, 8/4, 9/4, 9/4, 10/4, 10/4, 11/4, 11/4, 12/4, 12/4, …}

Based on p and n we now define BV in the following way: $BV(i)$ = max (entry i in the bound vector of type p, $\lceil pn/n - i \rceil / p$).

When $p = 2$ and $n = 12$ we get: BV = {3/2, 4/2, 4/2, 5/2, 5/2, 6/2, 6/2, 6/2, 7/2, 12/2, 24/2}.

For $p \leq n$, we have previously defined a recovery scheme - the p-process recovery scheme - that is optimal as long as at most $\lfloor \log_2 \lfloor n/p \rfloor \rfloor$ computers break down [20]. We call it p-process because we consider the case where there are p processes on each computer when all computers are up and running.

We now define *the modulo p-process recovery scheme*. The scheme is defined when $p \leq n$ (remember that optimal recovery schemes are most important when p is small compared to n). We will show that this recovery scheme is optimal also when significantly more than $\lfloor \log_2 \lfloor n/p \rfloor \rfloor$ computers break down.

Let R_0 be the oprimal recovery scheme for some n and $R_0(k)$ be k entry in R_0. Let then $R_{i,j}$ denotes the optimal recovery list for process j ($0 \le j < p$) on computer i ($0 \le i < n$). We define $R_{i,j}$ based on R_0, i.e. the recovery scheme for process zero in the single-process recovery scheme defined in Section 6 (Table 2).

If $R_0(k) + j\lfloor n/p \rfloor < n$, then $R_{i,j}(k) = (R_0(k) + i + j\lfloor n/p \rfloor) \bmod n$, else $R_{i,j}(k) = (R_0(k) + i + j\lfloor n/p \rfloor + n) \bmod n$, where $R_{i,j}(k)$ denotes integer number k in the lists for process j on computer i.

The table below shows the recovery lists when $n = 11$ and $p = 2$.

$R_{0,0}$	1,6,3,10,2,4,5,7,8,9
$R_{0,1}$	6,0,8,4,7,9,10,1,2,3
$R_{1,0}$	2,7,4,0,3,5,6,8,9,10
$R_{1,1}$	7,1,9,5,8,10,0,2,3,4
$R_{2,0}$	3,8,5,1,4,6,7,9,10,0
$R_{2,1}$	8,2,10,6,9,0,1,3,4,5
$R_{3,0}$	4,9,6,2,5,7,8,10,0,1
$R_{3,1}$	9,3,0,7,10,1,2,4,5,6
$R_{4,0}$	5,10,7,3,6,8,9,0,1,2
$R_{4,1}$	10,4,1,8,0,2,3,5,6,7

ABSTRACT

This thesis consists of two parts: performance bounds for scheduling algorithms for parallel programs in multiprocessor systems, and recovery schemes for fault tolerant distributed systems when one or more computers go down.

In the first part we deliver tight bounds on the ratio for the minimal completion time of a parallel program executed in a parallel system in two scenarios. Scenario one, the ratio for minimal completion time when processes can be reallocated compared to when they cannot be reallocated to other processors during their execution time. Scenario two, when a schedule is preemptive, the ratio for the minimal completion time when we use two different numbers of preemptions.

The second part discusses the problem of redistribution of the load among running computers in a parallel system. The goal is to find a redistribution scheme that maintains high performance even when one or more computers go down. Here we deliver four different redistribution algorithms.

In both parts we use theoretical techniques that lead to explicit worst-case programs and scenarios. The correctness is based on mathematical proofs.

ISSN 1653-2090

2007:18

ISBN 978-91-7295-126-6