TECHNISCHE
UNIVERSITÄT
MÜNCHEN

INSTITUT FÜR INFORMATIK

Sonderforschungsbereich 342:
Methoden und Werkzeuge für die Nutzung
paralleler Rechnerarchitekturen

"Anwendungsbezogene Lastverteilung" ALV'98

25.-26. März 1998

München

Arndt Bode, Andreas Ganz, Claudia Gold, Stefan Petri
Niels Reimer, Björn Schiemann, Thomas Schnekenburger
(Herausgeber)

TUM-I9806
SFB-Bericht Nr. 342/01/98 A
Februar 98

TUM–INFO–02-I9806-100/1.–FI

Alle Rechte vorbehalten
Nachdruck auch auszugsweise verboten

©1998 SFB 342 Methoden und Werkzeuge für
die Nutzung paralleler Architekturen

Anforderungen an: Prof. Dr. A. Bode
Sprecher SFB 342
Institut für Informatik
Technische Universität München
D-80290 München, Germany

Druck: Fakultät für Informatik der
Technischen Universität München

Workshop
"Anwendungsbezogene Lastverteilung"
ALV'98

25.-26.März 1998
München

Veranstaltet vom Sonderforschungsbereich 342
"Werkzeuge und Methoden für die Nutzung paralleler Rechnerarchitekturen"
Querschnittsthema "Anwendungsbezogene Lastverteilung"
und vom Graduiertenkolleg
"Kooperation und Ressourcenmanagement in verteilten Systemen"

Herausgeber, Organisationskomitee:

Arndt Bode, TU München, Sprecher SFB 342
Andreas Ganz, TU München
Claudia Gold, TU München
Stefan Petri, MU Lübeck
Niels Reimer, TU München
Björn Schiemann, Siemens AG
Thomas Schnekenburger, Siemens AG

Vorwort

Die anwendungsbezogene Lastverteilung ist das zentrale Thema bei der Optimierung von Anwendungsprogrammen auf parallelen und verteilten Systemen. Das Thema hat daher auch eine besondere Rolle im Sonderforschungsbereich 342 „Werkzeuge und Methoden für die Nutzung paralleler Rechnerarchitekturen“ und im Graduiertenkolleg „Kooperation und Ressourcenmanagement in verteilten Systemen“ der Technischen Universität München. Als Querschnittsthema im Sonderforschungsbereich 342 wurde die anwendungsbezogene Lastverteilung zunächst bezogen auf Multiprozessorsysteme mit verteiltem Speicher untersucht. In den letzten Jahren hat sich der Schwerpunkt der Arbeiten dann auf Netze von (Arbeitsplatz-)Rechnern verschoben, die ebenfalls durch verteilte Speicher gekennzeichnet sind, intern jedoch auch das Modell des gemeinsamen Speichers benutzen können. Fragen des Lastausgleichs und der Ressourcenverwaltung sind dabei ebenso eng verknüpft wie die des anwendungsintegrierten und des systemintegrierten (anwendungsübergreifenden) Lastausgleichs. Je nach Zielsystem, Betriebssystem, Programmiermodell und Anwendung sind hier unterschiedlichste Verfahren vorgeschlagen worden. Es ist das Ziel des Workshops, Forscher und Anwender, die im Bereich der dynamischen anwendungsbezogenen Lastverteilung tätig sind, zusammenzubringen und dabei insbesondere die Arbeiten im Sonderforschungsbereich 342 und dem Graduiertenkolleg „Kooperation und Ressourcenmanagement in verteilten Systemen“ mit externen Kooperationspartnern eng zu verknüpfen.

München, den 28.1.1998

Arndt Bode
Sprecher SFB 342

Inhaltsverzeichnis

Load Balancing for Problems with Good Bisectors, and Applications in Finite Element Simulations
Stefan Bischof, Ralf Ebner, Thomas Erlebach (Technische Universität München) **1**

An Algorithm for Load Distribution in Interactive Theorem Proving
Thomas Honigmann (Humboldt-Universität Berlin) Andreas Wolf (Technische Universität München) **13**

Skew-tolerantes, dynamisches LPT-Scheduling zur Join-Verarbeitung in parallelen Shared-Disk-Datenbanksystemen
Holger Märtens (Universität Leipzig) **23**

A Decentral, Adaptive Load Sharing Approach for Workstation Clusters using Fuzzy Logic
Thomas Böhme (Technische Universität Ilmenau) Herwig Unger (Universität Rostock) **33**

Load Balancing for Parallel Adaptive FEM
Ralf Diekmann, Frank Schlimbach (Universität Paderborn) Chris Walshaw (University of Greenwich, London) **41**

Load Balancing in High-Performance Distributed DB/DC-Systems: Modeling, Analysis and Clustering of Workload for Data-Affinity Based Load Distribution
Reinhard Riedl (Universität Zürich) **53**

Towards a Scalable Parallel Algorithm for Polynomial Real Root Isolation
Thomas Decker, Werner Krandick (Universität Paderborn) **65**

Dynamic Load Distribution with the WINNER System
Olaf Arndt, Bernd Freisleben, Thilo Kielmann, Frank Thilo (Universität Siegen) **77**

The PLB-Library: Dynamic Load Balancing on NOWs
Frank Meisgen, Ewald Speckenmeyer (Universität Köln) **89**

Object Mobility in ARTS
Lars Büttner (GMD-FIRST) **101**

Lastverwaltung in Workstation-Clustern auf Basis Mobiler Agenten
Claus Grewe, Wolfgang Obelöer, Holger Pals (Medizinische Universität zu Lübeck) **111**

Tree Shaped Computations as a Model for Parallel Applications
Peter Sanders (Max-Planck-Institut Saarbrücken) **123**

NSR - A Tool for Load Measurement in Heterogeneous Environments
Christian Röder, Thomas Ludwig, Arndt Bode (Technische Universität München) **133**

Load Balancing for Problems with Good Bisectors, and Applications in Finite Element Simulations

Stefan Bischof Ralf Ebner
Thomas Erlebach

Institut für Informatik
Technische Universität München
D-80290 München, Germany

{bischof|ebner|erlebach}@informatik.tu-muenchen.de
http://www{mayr|zenger}.informatik.tu-muenchen.de/

Abstract

This paper studies load balancing issues for classes of problems with certain bisection properties. A class of problems has α-bisectors if every problem in the class can be subdivided into two subproblems whose weight is not smaller than an α-fraction of the original problem. It is shown that the maximum weight of a subproblem produced by Algorithm HF, which partitions a given problem into N subproblems by always subdividing the problem with maximum weight, is at most a factor of $\lfloor 1/\alpha \rfloor \cdot (1-\alpha)^{\lfloor 1/\alpha \rfloor - 2}$ greater than the theoretical optimum (uniform partition). This bound is proved to be tight. Two strategies to use Algorithm HF for load balancing distributed hierarchical finite element simulations are presented. For this purpose, a certain class of weighted binary trees representing the load of such applications is shown to have $1/4$-bisectors. This establishes a performance guarantee of $9/4$ for load balancing in this case.

1 Introduction

Load balancing is one of the major research issues in the context of parallel computing. Irregular problems are often difficult to tackle in parallel because they tend to overload some processors while leaving other processors nearly idle. For

these applications it is very important to find methods for obtaining a balanced distribution of load. Usually, the load is created by processes that are part of a (parallel) application program. During the run of the application, these processes perform certain calculations independently but have to communicate intermediate results or other data using message passing. One is usually interested in achieving a balanced load distribution in order to minimize the execution time of the application or to maximize system throughput.

Load balancing problems have been studied for a huge variety of models, and many different solutions regarding strategies and implementation mechanisms have been proposed. A good overview of recent work can be obtained from [SHK95], for example. [SS97] reviews ongoing research on dynamic load balancing, emphasizing the presentation of models and strategies within the framework of general classification schemes.

In this paper we study load balancing for a very general class of problems. The only assumption we make is that all problems in the class have a certain bisection property. Such classes of problems arise, for example, in the context of distributed hierarchical finite element simulations. We show how our general results can be applied to numerical applications in several ways. The remainder of the paper is structured as follows. In Section 2 we present and analyze a very general and simple algorithm that computes a good load distribution for classes of problems with α-bisectors. Section 3 shortly explains distributed finite element simulations with recursive substructuring. Two strategies for applying the algorithm from Section 2 to these applications are discussed. Section 4 shows that certain weighted trees, which model the load of applications in numerical simulations like the one discussed in Section 3, have $1/4$-bisectors. This implies a performance guarantee[1] of $9/4$ for load balancing these applications. Section 5 summarizes our results.

2 Using Bisectors for Load Balancing

In many applications a computational problem cannot be divided into many small problems as required for an efficient parallel solution directly. Instead, a strategy similar to *divide and conquer* is used repeatedly to divide problems into smaller subproblems. We refer to the division of a problem into two smaller subproblems as *bisection*. Assuming a weight function w that measures the resource demand, a problem p cannot always be bisected into two subproblems p_1 and p_2 of equal weight $\mathsf{w}(p)/2$. For many classes of problems, however, there is a bisection method that guarantees that the weights of the two obtained subproblems do not differ too much. The following definition captures this concept more precisely.

[1] An algorithm has performance guarantee ρ if the maximum load produced by the algorithm is at most a factor of ρ larger than the maximum load of an optimum solution.

```
Input: problem p, positive integer N
begin
    P ← {p};
    while |P| < N do
        begin
        q ← a problem in P with maximum weight;
        bisect q into q_1 and q_2;
        P ← (P ∪ {q_1, q_2}) \ {q};
        end;
    output P;
end.
```

Figure 1: Algorithm HF (Heaviest Problem First)

Definition 1 *Let* $0 < \alpha \leq \frac{1}{2}$. *A class* $\mathcal{P}$ *of problems with weight function* $\mathsf{w} : \mathcal{P} \to \mathbb{R}^+$ has α-bisectors *if every problem* $p \in \mathcal{P}$ *can be efficiently divided into two problems* $p_1 \in \mathcal{P}$ *and* $p_2 \in \mathcal{P}$ *with* $\mathsf{w}(p_1) + \mathsf{w}(p_2) = \mathsf{w}(p)$ *and* $\mathsf{w}(p_1), \mathsf{w}(p_2) \in [\alpha\mathsf{w}(p); (1-\alpha)\mathsf{w}(p)]$.

Note that this definition requires for the sake of simplicity that all problems in $\mathcal{P}$ can be bisected, whereas in practice this is not the case for problems whose weight is below a certain threshold. We assume, however, that the problem to be divided among the processors is big enough to allow further bisections until the number of subproblems is equal to the number of processors. This is a reasonable assumption for most relevant parallel applications.

Figure 1 shows Algorithm HF, which receives a problem p and a number N of processors as input and divides p into N subproblems by repeated application of α-bisectors to the heaviest remaining subproblem. A perfectly balanced load distribution on N processors would be achieved if a problem p of weight $\mathsf{w}(p)$ was divided into N subproblems of weight exactly $\mathsf{w}(p)/N$ each. The following theorem gives a worst-case bound on the ratio between the maximum weight among the N subproblems produced by Algorithm HF and this ideal weight $\mathsf{w}(p)/N$.

Theorem 2 *Let* $\mathcal{P}$ *be a class of problems with weight function* $\mathsf{w} : \mathcal{P} \to \mathbb{R}^+$ *that has* α*-bisectors. Given a problem* $p \in \mathcal{P}$ *and a positive integer* N*, Algorithm HF uses* $N - 1$ *bisections to partition* p *into* N *subproblems* $p_1, \ldots, p_N$ *such that*

$$\max_{1 \leq i \leq N} \mathsf{w}(p_i) \leq \frac{\mathsf{w}(p)}{N} \cdot \left\lfloor \frac{1}{\alpha} \right\rfloor \cdot (1-\alpha)^{\lfloor \frac{1}{\alpha} \rfloor - 2}.$$

Proof: It is obvious that Algorithm HF uses $N-1$ bisections to partition p into N subproblems. In the following we show that the stated inequality regarding the maximum weight among these subproblems holds.

We introduce the bisection tree T to represent the run of the algorithm on input p and N. The root of T is the problem p. If the algorithm bisects a problem q into q_1 and q_2, nodes q_1 and q_2 are added to T as children of node q. In the end, T has N leaves, which correspond to the N subproblems computed by the algorithm, and all problems that were bisected by the algorithm appear as internal nodes with exactly two children. Figure 2 gives an example of a bisection tree for a problem of weight 44 from a class of problems with $\frac{1}{7}$-bisectors. We follow the convention of drawing the node with greater weight among two children of the same parent as a left child of that parent.

The following properties hold for bisection trees arising from classes of problems with α-bisectors. Let the leaves of the tree be $p_1, \ldots, p_N$ and let $m := \max_{1 \le i \le N} \mathsf{w}(p_i)$.

(a) $\mathsf{w}(q) \ge m$ for all internal nodes q

(b) $\mathsf{w}(q) \ge \frac{1}{1-\alpha}\mathsf{w}(q')$ if q' is a child of q

(a) holds because the algorithm always bisects a subproblem of maximum weight; since one of the p_i has weight m, there must have been at least one subproblem of weight $\ge m$ during the whole run of the algorithm, and thus the algorithm never bisected a problem of weight $< m$. (b) follows directly from $\mathsf{w}(q') \le (1-\alpha)\mathsf{w}(q)$, which holds because the algorithm uses α-bisectors.

Now remove from the bisection tree all internal nodes which are not parent of a leaf. This partitions the bisection tree into a number of disjoint branches, whose shape can be that of a *leaf-branch* (one of the internal nodes of the branch has two leaf children) or that of an *internal branch* (all the internal nodes of the branch have exactly one leaf child). The branches obtained from the example tree of Figure 2 are shown in Figure 3. Our goal is to derive a lower bound for the average weight of the leaves in each branch.

Consider a leaf-branch with k internal nodes, $k \ge 1$. Denote its internal nodes by $v_1, v_2, \ldots, v_k$ such that v_{i+1} is the parent of v_i for $1 \le i \le k-1$. Furthermore, let c_i denote the leaf child of v_i for $2 \le i \le k$, and let c_0 and c_1 denote the leaf children of v_1. As (a) implies $\mathsf{w}(v_1) \ge m$, we have by (b) $\mathsf{w}(v_i) \ge \left(\frac{1}{1-\alpha}\right)^{i-1} m$ for $1 \le i \le k$ and $\mathsf{w}(c_i) \ge \alpha \left(\frac{1}{1-\alpha}\right)^{i-1} m$ for $1 \le i \le k$. The average weight of the leaves $c_0, \ldots, c_k$ can now be bounded from below as follows:

$$\frac{1}{k+1}\sum_{i=0}^{k} \mathsf{w}(c_i) \;\; \ge \;\; \frac{1}{k+1}\left(m + \sum_{i=2}^{k} \alpha \left(\frac{1}{1-\alpha}\right)^{i-1} m\right)$$

$$
\begin{aligned}
&= \frac{1}{k+1}\left(m(1-\alpha)+\alpha m\sum_{i=0}^{k-1}\frac{1}{(1-\alpha)^i}\right)\\
&= \frac{1}{k+1}\left(m(1-\alpha)+m(\alpha-1)\left(1-\frac{1}{(1-\alpha)^k}\right)\right)\\
&= \frac{m}{(k+1)(1-\alpha)^{k-1}}\,.
\end{aligned}
$$

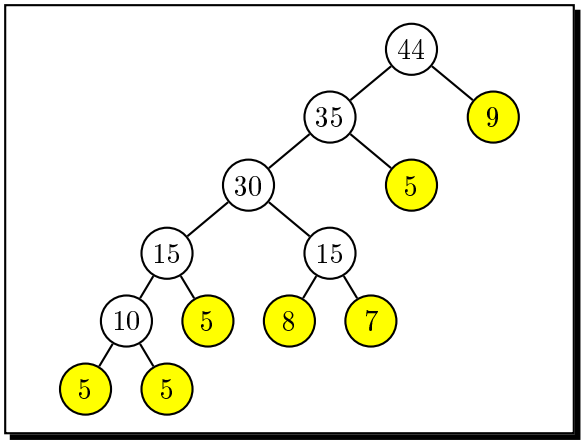


Figure 2: Example of a Bisection Tree

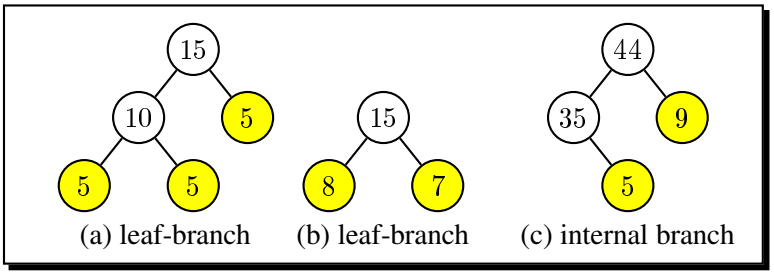


Figure 3: Branches Obtained from the Example Tree

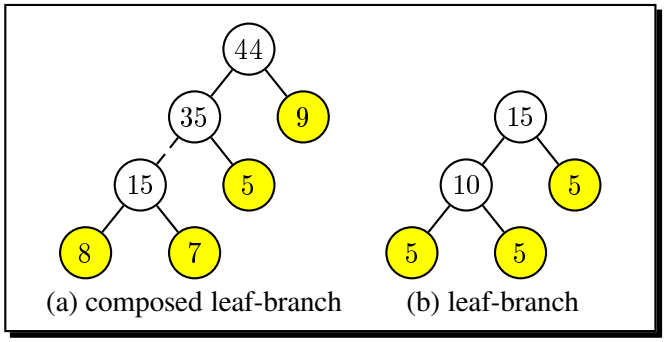


Figure 4: Composed Leaf-Branches Obtained from the Example Tree

If there are internal branches, we do not deal with them separately but instead attach them to leaf-branches. For example, one can consider the leaf-branches one by one and attach to each leaf-branch all internal branches that intersect the path from the leaf-branch to the root of the bisection tree and that have not been attached to a different leaf-branch beforehand. Here, attaching an internal branch to a leaf branch means making the root of the leaf-branch a child of the bottom-most internal node of the internal branch, resulting in a new leaf-branch. We call the leaf-branches obtained by attaching zero or more internal branches to an original leaf-branch *composed leaf-branches*. Observe that conditions (a) and (b) are satisfied for these composed leaf-branches as well. Hence, the lower bound above also pertains to the average weight of the leaves in such a composed leaf-branch. The bisection tree from Figure 2 contained two leaf-branches and one internal branch as illustrated in Figure 3. Attaching the internal branch to one of the leaf-branches gives the composed leaf-branches shown in Figure 4.

As every leaf of the bisection tree appears in exactly one composed leaf-branch, we conclude that $\min_{k\in\mathbb{N}} \frac{m}{(k+1)(1-\alpha)^{k-1}}$ is a lower bound on the average weight of all leaves in the bisection tree. Therefore, we obtain

$$\mathsf{w}(p) = \sum_{i=1}^{N} \mathsf{w}(p_i) \geq Nm \min_{k\in\mathbb{N}} \frac{1}{(k+1)(1-\alpha)^{k-1}} \,. \tag{1}$$

Besides, we observe that

$$\min_{k\in\mathbb{N}} \frac{1}{(k+1)(1-\alpha)^{k-1}} = \frac{1}{\max_{k\in\mathbb{N}} \left((k+1)(1-\alpha)^{k-1}\right)} \,, \tag{2}$$

and we claim that $(k+1)(1-\alpha)^{k-1}$ as a function of $k \in \mathbb{N}$ is maximized for $\hat{k} = \lfloor 1/\alpha \rfloor - 1$. To see this, let $f(k) = (k+1)(1-\alpha)^{k-1}$ and consider the ratio $f(k)/f(k-1) = (1-\alpha)(k+1)/k$. We obtain:

$$\frac{f(k)}{f(k-1)} = \begin{cases} > 1 & \text{for } k < \frac{1}{\alpha} - 1 \\ = 1 & \text{for } k = \frac{1}{\alpha} - 1 \\ < 1 & \text{for } k > \frac{1}{\alpha} - 1 \end{cases} .$$

For a fixed value of α, $f(k)$ is monotonely increasing from $k = 1$ to $k = \lfloor 1/\alpha \rfloor - 1$ and monotonely decreasing for larger values of k. If $1/\alpha$ is not an integer greater than 2, $f(k)$ is maximum only for $k = \lfloor 1/\alpha \rfloor - 1$. If $1/\alpha$ is an integer greater than 2, $f(k)$ is maximum for $k = \lfloor 1/\alpha \rfloor - 1$ and for $k = \lfloor 1/\alpha \rfloor - 2$. In any case we have $\max_{k\in\mathbb{N}} \left((k+1)(1-\alpha)^{k-1}\right) = f(\lfloor 1/\alpha \rfloor - 1) = \lfloor 1/\alpha \rfloor \cdot (1-\alpha)^{\lfloor 1/\alpha \rfloor - 2}$, and the theorem follows with (1) and (2). □

For some values of α, Table 1 gives worst-case bounds on the ratio between $\max_{1\leq i\leq N} \mathsf{w}(p_i)$ and $\frac{\mathsf{w}(p)}{N}$ as well as a value of k for which $(k+1)(1-\alpha)^{k-1}$ is

Table 1: Worst-Case Ratio of Algorithm HF for Different Values of α

α	k	ratio
0.02	49	18.96
0.04	24	9.78
0.06	15	6.73
0.08	11	5.21
0.10	8	4.30
0.12	7	3.72
0.14	6	3.29
0.16	5	2.99
0.18	4	2.76
0.20	3	2.56

α	k	ratio
0.21	3	2.50
0.22	3	2.43
0.23	3	2.37
0.24	3	2.31
0.25	2	2.25
0.26	2	2.22
0.27	2	2.19
0.28	2	2.16
0.29	2	2.13
0.30	2	2.10

α	k	ratio
0.31	2	2.07
0.32	2	2.04
0.325	2	2.025
0.33	2	2.01
0.331	2	2.007
0.332	2	2.004
0.333	2	2.001
0.334	1	2.00
0.40	1	2.00
0.50	1	2.00

maximized. These bounds show that the worst-case deviation from the ideal load distribution, in which $\mathsf{w}(p_i) = \frac{\mathsf{w}(p)}{N}$ for all $1 \leq i \leq N$, is bounded by a small constant for a wide range of α. Note that in many cases an ideal load distribution cannot be achieved by any algorithm.

Now we give a lower bound for the worst-case ratio between the maximum weight subproblem generated by Algorithm HF and the ideal value given by a uniform partition. This will show that the upper bound from Theorem 2 is tight. The proof is omitted due to space limitations.

Theorem 3 *For each $0 < \alpha \leq \frac{1}{2}$ there exists a class of problems $\mathcal{Q}^\alpha$ that has α-bisectors and contains a family of problems $(q^l)_{l\in\mathbb{N}}$ such that*

$$\lim_{l\to\infty} \frac{\max_{1\leq i\leq N_l} \mathsf{w}(q_i^l)}{\frac{\mathsf{w}(q^l)}{N_l}} = \left\lfloor \frac{1}{\alpha} \right\rfloor \cdot (1-\alpha)^{\left\lfloor \frac{1}{\alpha} \right\rfloor - 2},$$

where $N_l = \lfloor 1/\alpha \rfloor \cdot 2^l - 1$ and $q_1^l, q_2^l, \ldots, q_{N_l}^l$ are the subproblems generated by Algorithm HF on input q^l and N_l.

3 Application of Algorithm HF within Distributed Finite Element Simulations

In this section, we present the application of Algorithm HF for load balancing in the field of distributed numerical simulations with the finite element (FE) method [Bra97, Bur87]. The FE method is used in statics analysis, for example, to calculate the response of objects under external load. In this paper, we consider a

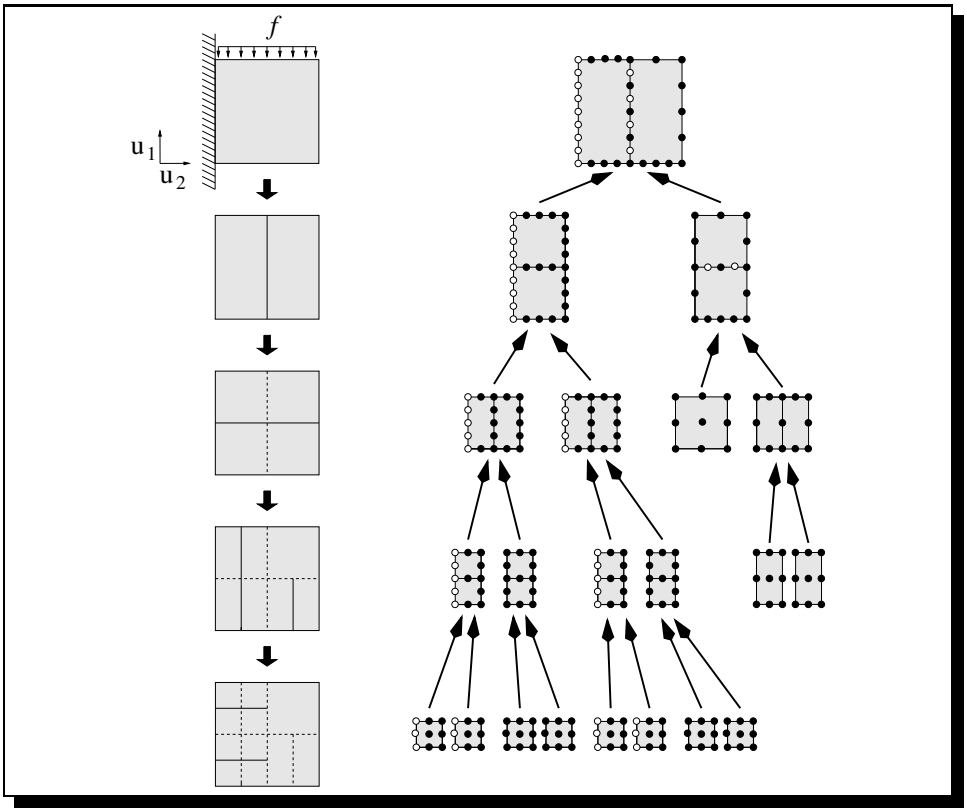


Figure 5: A short cantilever under uniform load: the recursive substructuring phase, resulting in a coarse discretizing mesh (left), and the bottom-up accumulation of equations in the resulting binary tree data structure (right)

short cantilever under plane stress conditions, a problem from plane elasticity. The quadratic cantilever is uniformly loaded on its upper side. Its left side is fixed as shown in Figure 5, top left. The definition of the object's shape, its structural properties, and the boundary conditions lead to a system of partial differential equations describing the relation between external loads, internal forces, and displacement.

We substructure the physical domain of the cantilever recursively. With the method of [Hüt96, HS94], a tree data structure is built reflecting the hierarchy of the substructured domain (see Figure 5). This data structure contains a linear equation system $Su = f$ for each tree node. Its *stiffness matrix* S determines the unknown displacement values in the vector u at the black points shown in Figure 5 (right), dependent on the external and reaction forces f. In the leaves, the equation systems are constructed by standard FE discretization and the so-called virtual displacement method. For an internal node, the linear equation system

is assembled out of the equation systems of its children. White points in the figure are ghost points with prescribed displacement (either interpolated or fixed by boundary conditions) and are not part of the linear equation system.

For the solution of all those equation systems, we use an iterative solver which traverses the tree several times, promoting displacements in top-down direction and reaction forces in bottom-up direction. Since the adaptive structure of the tree is not known *a priori*, a good load balancing strategy is essential before the parallel execution of the solving iterations.

We assign a load value $\ell(p) = C_b n_b(p) + C_s n_s(p)$ to each tree node p, with $n_b(p)$ points on the border without boundary conditions, and $n_s(p)$ points (not ghost points) belonging to the separator line of node p. The load value $\ell(p)$ models the computing time of node p, where the constants C_s and C_b are independent of node p, and $C_s \approx 6C_b$.

In order to distribute the work load among the processors, we interpret the FE tree as an approximate (potential) bisection tree by accumulating the load of all nodes in the subtrees rooted in node p to get the weight value $\mathsf{w}(p)$:

$$\mathsf{w}(p) = \begin{cases} \ell(p) & \text{if } p \text{ is leaf} \\ \ell(p) + \mathsf{w}(c_1) + \mathsf{w}(c_2) & \text{if } p \text{ is root or internal node} \end{cases}$$

where c_1 and c_2 are the children of p. The weight values are collected during the tree construction phase by simply counting the number of points of each tree node.

For a bisection step according to Algorithm HF, we remove the root node p from the tree of weight values. This yields two subtrees rooted at the children c_1 and c_2 of p, and the root node p is ignored for the remainder of the load balancing phase. Such bisection steps do not exactly match Definition 1, because they satisfy only $\mathsf{w}(c_1) + \mathsf{w}(c_2) = \mathsf{w}(p) - \ell(p)$. However, $\ell(p)$ is negligible compared to $\mathsf{w}(p)$ in our application. Hence, the results of Theorem 2 are well approximated.

Algorithm HF chops N subtrees off the FE tree, each of which can be traversed in parallel by the iterative solver. These N subtrees contain the main part of the solving work and may be distributed over the available N processors, whereas the upper $N - 1$ tree nodes cannot exploit the whole number of processors, anyway.

Unfortunately, we can not bound the bisection factor α in arbitrary adaptive FE simulations. Therefore, another strategy using Algorithm HF allows the removal of a single edge of a tree as a bisection step. This strategy partitions the entire given FE tree into N subtrees of approximately equal size. Section 4 shows that FE trees satisfying the conditions

$$\begin{aligned} \ell(p) &\leq \ell(c_1) + \ell(c_2) \\ \ell(p) &\geq \ell(c_i) \qquad\qquad (i = 1, 2) \end{aligned}$$

have good bisectors. This application of Algorithm HF also takes into account that the main memory resources of the processors become the limiting factor if very

high accuracy of the simulation is required. In this case, finding a partitioning of the entire tree (not only a set of equal-sized subtrees ignoring their ancestors in the tree) is necessary.

4 Weighted Trees with Good Bisectors

Let $\mathcal{T}$ be the set of all rooted binary trees with node weights $\ell(v)$ satisfying:

(1) $\ell(p) \leq \ell(c_1) + \ell(c_2)$ for nodes p with two children c_1 and c_2
(2) $\ell(p) \geq \ell(c)$ if c is a child of p

The weight of a tree $T = (V, E)$ in $\mathcal{T}$ is defined as $\mathsf{w}(T) = \sum_{v \in V} \ell(v)$.

This class $\mathcal{T}$ of binary trees models the load of applications in hierarchical finite element simulations, as discussed in Section 3. Recall that in these applications the domain of the computation is repeatedly subdivided into smaller subdomains. The structure of the domains and subdomains yields a binary tree in which every node has either two children or is a leaf. The resource demands (CPU and main memory) of the nodes in this FE tree are such that the resource demand at a node is at most as large as the sum of the resource demands of its two children. In order to parallelize the computation, it is necessary to distribute the FE tree among a number of processors in a balanced way.

Note that Conditions (1) and (2) ensure that the two subtrees obtained from a tree in $\mathcal{T}$ by removing a single edge are also members of $\mathcal{T}$.

The following theorem shows that trees from the class $\mathcal{T}$ can be $\frac{1}{4}$-bisected by removal of a single edge unless the weight of the tree is concentrated in the root.

Theorem 4 *Let $T = (V, E)$ be a tree in $\mathcal{T}$, and let r be its root. If $\ell(r) \leq \frac{3}{4}\mathsf{w}(T)$, then there is an edge $e \in E$ such that the removal of e partitions T into subtrees T_1 and T_2 with $\mathsf{w}(T_1), \mathsf{w}(T_2) \in [\frac{1}{4}\mathsf{w}(T); \frac{3}{4}\mathsf{w}(T)]$.*

Proof: We give a simple method to find the required edge. Pick an arbitrary node v of T as a start node. While $T \setminus \{v\}$ contains a subtree T' with $\mathsf{w}(T') > \frac{3}{4}\mathsf{w}(T)$, replace v by the node adjacent to v which is contained in T'. This process always terminates after less than $|V|$ iterations at a node v such that all subtrees T' in $T \setminus \{v\}$ satisfy $\mathsf{w}(T') \leq \frac{3}{4}\mathsf{w}(T)$. We claim that at least one of these subtrees also satisfies $\mathsf{w}(T') \geq \frac{1}{4}\mathsf{w}(T)$, and thus the edge connecting v and T' can be picked as the required separator edge. In order to prove the claim we distinguish several cases regarding the position of v in T. For every case the assumption that all subtrees of $T \setminus \{v\}$ have weight $< \frac{1}{4}\mathsf{w}(T)$ will lead to a contradiction.

Case 1: v has degree 3. Let u be the parent of v. Let T_1 be the subtree of $T \setminus \{v\}$ that contains u, and let T_2 and T_3 be the other two subtrees. Assume that all

three subtrees have weight $< \frac{1}{4}\mathsf{w}(T)$. Consequently, v must have weight $> \frac{1}{4}\mathsf{w}(T)$ because $\mathsf{w}(T) = \mathsf{w}(T_1) + \mathsf{w}(T_2) + \mathsf{w}(T_3) + \ell(v)$. But then u must also have weight $> \frac{1}{4}\mathsf{w}(T)$ because it is the parent of v, and this implies $\mathsf{w}(T_1) > \frac{1}{4}\mathsf{w}(T)$. A contradiction.

Case 2: v has degree 2 and is the root of T. Let T_1 and T_2 be the two subtrees of $T \setminus \{v\}$, and let u_1 and u_2 be the corresponding children of v. If both subtrees have weight $< \frac{1}{4}\mathsf{w}(T)$, it follows also that $\ell(u_1) < \frac{1}{4}\mathsf{w}(T)$ and $\ell(u_2) < \frac{1}{4}\mathsf{w}(T)$, which implies $\ell(v) \leq \ell(u_1) + \ell(u_2) < \frac{1}{2}\mathsf{w}(T)$. On the other hand, $\mathsf{w}(T) = \mathsf{w}(T_1) + \mathsf{w}(T_2) + \ell(v)$ implies $\ell(v) > \frac{1}{2}\mathsf{w}(T)$. A contradiction.

Case 3: v has degree 2 and is not the root of T. Let u be the parent of v, and let c be the child of v. Let T_1 be the subtree in $T \setminus \{v\}$ that contains u, and let T_2 be the other subtree. If both subtrees have weight $< \frac{1}{4}\mathsf{w}(T)$, $\mathsf{w}(T) = \mathsf{w}(T_1) + \mathsf{w}(T_2) + \ell(v)$ implies $\ell(v) > \frac{1}{2}\mathsf{w}(T)$. But then $\ell(u) \geq \ell(v) > \frac{1}{2}\mathsf{w}(T)$. A contradiction.

Case 4: v has degree 1 and is a leaf of T. Let T_1 be the tree $T \setminus \{v\}$, and let u be the parent of v. Assume that $\mathsf{w}(T_1) < \frac{1}{4}\mathsf{w}(T)$. Then $\mathsf{w}(T) = \mathsf{w}(T_1) + \ell(v)$ implies $\ell(v) > \frac{3}{4}\mathsf{w}(T)$. But then $\mathsf{w}(T_1) \geq \ell(u) \geq \ell(v) > \frac{3}{4}\mathsf{w}(T)$. A contradiction.

Case 5: v has degree 1 and is the root of T. Let T_1 be the tree $T \setminus \{v\}$. As $\ell(v) = \ell(r) \leq \frac{3}{4}\mathsf{w}(T)$, $\mathsf{w}(T) = \ell(v) + \mathsf{w}(T_1)$ implies $\mathsf{w}(T_1) \geq \frac{1}{4}\mathsf{w}(T)$. □

According to Theorem 2 a problem p from a class of problems that has $\frac{1}{4}$-bisectors can always be subdivided into N subproblems $p_1, \ldots, p_N$ such that $\max_{1 \leq i \leq N} \mathsf{w}(p_i) \leq \frac{\mathsf{w}(p)}{N} \cdot \frac{9}{4}$. The following corollary gives a condition on trees in $\mathcal{T}$ that ensures that they can be subdivided into N subproblems using $\frac{1}{4}$-bisectors.

Corollary 5 *Let $T = (V, E)$ be a tree in $\mathcal{T}$, and let r be its root. Let N be a positive integer. If $\mathsf{w}(T) \geq \frac{4}{3}N\ell(r)$, Algorithm HF partitions T into N subtrees by cutting exactly $N - 1$ edges such that the maximum weight of the resulting subtrees is at most $\frac{9}{4} \cdot \frac{\mathsf{w}(T)}{N}$.*

Proof: Omitted due to space limitations.

5 Conclusion

The existence of α-bisectors for a class of problems was shown to allow good load balancing for a surprisingly large range of values of α. The maximum load achieved by Algorithm HF is at most a factor of $\lfloor 1/\alpha \rfloor \cdot (1-\alpha)^{\lfloor 1/\alpha \rfloor - 2}$ larger than the theoretical optimum (uniform distribution). This bound was proved to be tight. It gives a performance guarantee of factor 2 for $\alpha \geq 1/3$ and factor 3 for $\alpha \geq 1 - 1/\sqrt[4]{2} \approx 0.159$. Load balancing for distributed hierarchical finite

element simulations was discussed, and two strategies for applying Algorithm HF were presented. The first strategy tries to make the best use of the available parallelism, but requires that the nodes of the FE tree representing the load of the computation have good separators. The second strategy tries to partition the entire FE tree into subtrees with approximately equal load. For this purpose, it was proved that a certain class of weighted trees, which include FE trees, has $1/4$-bisectors. Here, the trees are bisected by removing a single edge. Partitioning the trees by removing a minimum number of edges ensures that only a minimum number of communication channels of the application must be realized by network connections. Our results provide performance guarantees for balancing the load of applications with good bisectors in general and of distributed hierarchical finite element simulations in particular. For the latter application, we showed that the maximum resulting load is at most a factor of $9/4$ larger than in a perfectly uniform distribution.

At present we are implementing the load balancing methods proposed in this paper and integrating them into the existing finite element simulations software. We expect considerable improvements as compared to the static (compile-time) processor allocation currently in use.

References

[Bra97] Dietrich Braess. *Finite Elemente*. Springer, Berlin, 1997. 2. überarbeitete Auflage.

[Bur87] D.S. Burnett. *Finite Element Analysis*. Addison-Wesley Publishing Company, 1987.

[HS94] Reiner Hüttl and Michael Schneider. Parallel Adaptive Numerical Simulation. SFB-Bericht 342/01/94 A, Technische Universität München, 1994.

[Hüt96] Reiner Hüttl. *Ein iteratives Lösungsverfahren bei der Finite-Element-Methode unter Verwendung von rekursiver Substrukturierung und hierarchischen Basen*. Dissertation, Technische Universität München, 1996.

[SHK95] Behrooz A. Shirazi, Ali R. Hurson, and Krishna M. Kavi. *Scheduling and Load Balancing in Parallel and Distributed Systems*. IEEE Computer Society Press, Los Alamitos, CA, 1995.

[SS97] Thomas Schnekenburger and Georg Stellner, editors. *Dynamic Load Distribution for Parallel Applications*. TEUBNER-TEXTE zur Informatik. Teubner Verlag, Stuttgart, 1997.

An Algorithm for Load Distribution in Interactive Theorem Proving*

Thomas Honigmann
Institute of Mathematics
Humboldt University Berlin
Ziegelstr. 13a
D-10099 Berlin
Germany
honig@mathematik.hu-berlin.de

Andreas Wolf
Computer Science Department
Munich University of Technology
Arcisstr. 21
D-80290 München
Germany
wolfa@informatik.tu-muenchen.de

Abstract

In interactive theorem proving, the factor of utilized wall clock time to prove a certain proof task is very important. Interactive users want to see solutions within a few seconds. Interactive provers often make use of automated theorem provers as background processes. These background processes have to be scheduled within a network of workstations with respect to other running applications of other users such that a response can be expected as fast as possible. We describe this problem and a possible prototypical solution within the interactive proof system ILF.

1 Introduction

Even interactive theorem provers usually offer the user a certain amount of routines for automatic solutions of proof parts. These tools can be subroutines and may be called heuristics as in KIV [10], or they can be automated theorem provers (ATPs) as in ILF [5], where proof results of asynchronously running ATPs are integrated into the proof attempt treated interactively. To increase the performance of integrated provers like in ILF, often different provers or different variants of the same prover are started competitively. Furthermore, different processes representing proof tasks coming from sub–tasks, e.g. making a case distinction, have to be handled at the same time.

These processes cannot share a single processor, at least not in acceptable time. Networks of workstations are a possible solution. The distribution of processes to a set of processors usually increases the performance of the automatic components of interactive provers. This is due to the fact, that the (interactively working) user accepts only response times up to at most one or two minutes. This time does *not* depend on the

*This work is supported by the Deutsche Forschungsgemeinschaft within the Schwerpunkt Deduktion and the Sonderforschungsbereich 342 subproject A5 (PARIS).

hardware used. So the processes of the automatic sub–systems have to be scheduled such that a proof (if there is one) can be expected within this time. But we do not know in advance which prover will find a proof.

Usually, the processes of more than one user share a workstation cluster. That makes the problem of finding an appropriate schedule of the sub–prover processes even more complicated.

In this paper, we discuss a possible solution to the load distribution problem for interactive theorem provers with automatic sub–components by considering the ILF system. The discussion is followed by a short assessment and concluded with a discussion on the possibilities for future work.

2 The ILF System

The ILF system can be characterized as "a shell for the integration of logic functions of different kinds – ATPs as well as interactive theorem provers, proof tactics, model checkers and proof presentation tools" ([5]). Its goal is to allow the user to focus his attention "... to the contents of the proof attempt, and away from of the technical details of a certain ATP". To achieve this goal, so called "experts" are introduced.

Experts can be considered as an abstraction of ATPs, although an expert could very well employ more than one ATP and run them competitively or cooperatively. The user may hand a first–order formula to an expert. The expert will then make an attempt to prove that this formula follows from the current theory and will report the result (e.g. success, failure because the time limit is exceeded, or total failure) back to the user. It is characteristic for ILF that the experts work asynchronously, that is, the user can continue to work with the system after an expert has been started, in particular he may start other experts to solve the same or another problem. That means the user is allowed and even encouraged to run more than one ATP simultaneously.

ILF also allows it to build theorem provers which make use of experts. An example is the interactive theorem prover ProofPad which allows the user to enter a (presumably incomplete) proof in the block calculus [6] and then tries to "fill the gaps" by using suitable experts.

3 The Problem

To understand how ILF handles the load distribution we have to take a closer look at the inner works of the experts. Currently, ILF supports the automated provers DISCOUNT [1], SETHEO [7], SPASS [13], ProTeIn [2], and OTTER [8]. It is planned to integrate

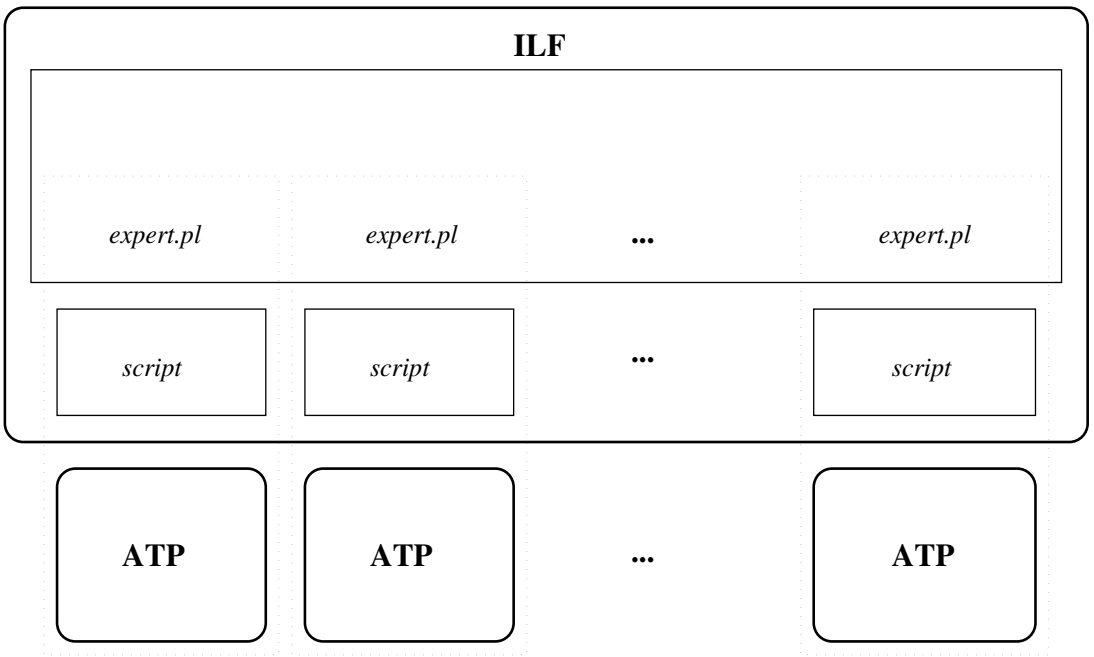


Figure 1: Architecture of ILF's experts

KoMeT [3]. Despite the differences between these provers, the corresponding experts basically have to do the same for each prover: They

1. generate the necessary input files,
2. start the prover,
3. terminate the prover if it has run unsuccessfully for a certain amount of time, and
4. analyze the prover's output.

It turned out to be convenient to split the implementation into two parts: the first part consisting of the tasks that can be done synchronously within ILF's core functionality and a second part which is executed asynchronously. Figure 1 tries to visualize the architecture of ILF's experts. The synchronous part (labeled *expert.pl* in figure 1) is implemented in *Prolog*, the implementation of the asynchronous part (*script*) was done using *perl* and shell tools. Task 1. is usually done by the synchronous part which then calls the shell script. The script starts the prover (task 2.), usually with a limit on the CPU time to account for task 3. It then waits for the prover to end, and (if necessary) transforms the prover's output into a *Prolog* readable form using *sed*, *awk*, and *perl*. Before terminating, the script notifies ILF's core, which reads the output files and informs the user of the result.

Especially when working in an environment like ProofPad the user may wish to use several experts at once. This can be done easily enough by the UNIX operation system using the provided multitasking. But, in our experience, even on a well–equipped (in terms of CPU power and main memory size) workstation, it is impractical to run more than three or four ATPs at the same time. Furthermore, some of the integrated ATPs are distributed systems itself (e.g. DISCOUNT [1]). Therefore it is clear, that ILF should be able to use other workstations in the local network. To make that possible, it is necessary to:

1. Pick the host(s) on which the ATP can run.
2. Transfer the input files to that host.
3. Run the prover.
4. Terminate the prover after a given *maximal run time*.
5. Transfer the output files back.

Most of these problems can be solved by using standard tools provided by the operating system. 2. and 5. are usually handled by the network file system (NFS) in a transparent way. The use of a remote shell (rsh) makes 3. and 4. on a remote machine as easy as on the local one. The typical maximal run time for a prover is less than five minutes. That is why migration of prover processes is not something that was dedicated much attention to. But unless one wants to use a simple *round robin* algorithm, 1. is a real challenge.

4 The Specification of the Solution

It was decided that our implementation to solve problem 1. should:

1. pick the host(s) with the lowest "workload" of all available hosts at a given time. The idea is that the ATP started on this host will probably find results fastest under the assumption that the only change in the workload of the hosts results from starting that ATP. Of course, determining a load index (to give "workload" an exact meaning) which is comparable between hosts is a problem in itself.
2. do not pick a host whose load index exceeds a certain threshold. Using a host with a high load index will slow down the proof attempt. Moreover, a high load will have an impact on other interactive users of the workstation cluster, too.
3. in the case where no host can be found (because of 2.) terminate all ATPs that have been working "long enough". To make this precise, the so called *flexy time* (in addition to the aforementioned maximal run time) was introduced: this is the (wall clock) time an ATP is guaranteed to work before it may be killed.

5 The Current Solution

The currently implemented solution consists of the program *iload* (written in C) and the *flexykill* shell script.

5.1 iload

iload uses the average number of jobs in the operating system's run queue over the last minute as reported by the *rstat()* function (part of Sun's RPC library) to describe the workload of a given host. We will refer to this value as $\mathit{avenrun}[0]$ since this is the name of the field in the structure filled in by *rstat()* which holds it (see the *rstat*(3) manual page for details). To be able to compare hosts with different hardware equipment this value is multiplied by a host specific factor. Currently these factors are chosen using a rule of thumb. We would like to find a method to determine such factors by a repeatable experiment or to actually calculate them somehow. $\mathit{avenrun}[0]$ reflects basically the load on the CPU. In our experience, in our context the memory seems to be a much less critical resource than the CPU load. Thus, we have not yet considered the current memory usage.

A recently started process has only a very small impact on $\mathit{avenrun}[0]$. Therefore, the same host would be picked several times when the requests are nearly simultaneous. This may happen if more than one user is working with ILF on the same network. Even for a single user this situation is very common when working with ProofPad — changing one line in the proof usually results in a couple of new proof tasks. To avoid this problem we use a "virtual load": the *iload server* keeps a list of all hosts and their virtual loads. Whenever *iload* picks a host, it notifies the server which increments the virtual load of this host by one. During the next 45 seconds the virtual load is decremented periodically, representing the load which is there but not yet visible in the actual load value.

In summary, *iload*'s load index can be defined as

$$\mathit{load_index}_{\text{host}} = \alpha_{\text{host}} \cdot (\mathit{avenrun}_{\text{host}}[0] + \mathit{virtual_load}_{\text{host}})$$

(α_{host} being the host specific factor mentioned above).

Another problem with nearly simultaneous *iload* requests is due to the fact, that each request is a separate (UNIX) process: In general, there would be no way to predict which request will be fulfilled first. This might be confusing for the user. Also, in a "*flexykill* situation" (see below) each request would start its own *flexykill*. This is useless in the best case and adds more load in the worst. Therefore a new *iload* process will check for other *iload* processes of the same user and (when necessary) wait until they are finished. The processes communicate with each other through signals and a

UNIX domain socket[1] with a fixed name. The process which acts as server at this socket will be called *user server* to distinguish it from the *iload server* which exists only once per network and will be used by any ILF user working there.

When asked to pick n hosts ($n \geq 1$) *iload* will do the following:

1. Check whether there is another *iload* process running by connecting as client to the socket.
   (a) If this succeeds, exchange process IDs with the *user server* and sleep until signal SIGUSR1 arrives. Then continue with 2.
   (b) In case of failure fork and start up an *user server* with a queue initialized with *iload*'s own process ID. The server will wait for connections at the socket and add the submitted process IDs to the queue. Upon receiving signal SIGUSR1 the server will remove one PID from the queue and forward the signal to the *iload* process which is now the queue's head. It will terminate itself when the queue is empty.
2. Determine the load of all hosts by calling the *rstat()* function. The necessary *rstatd* demons are available for most flavors of UNIX.
3. Determine the virtual load of all hosts from the *iload server*.
4. Add the actual and virtual load and multiply the sum by a host specific factor yielding a single workload value for each host. That factor was chosen to reflect the different performance potentials of different processor types within a real existing cluster of workstations. For example, we map a SPARC2 to 1000 and a SPARC10 to 3000. This selection is very arbitrary, but seems to work well.
5. Sort the hosts according to this value.
6. Compare the workload value of the n^{th} host in the list with the threshold value.
   (a) If it is lower: return the first n hosts from the list and notify the *iload server* that they are going to be used, or
   (b) If it is higher: call *flexykill* (see below), wait 10 seconds and continue with 2.
7. Send SIGUSR1 to the *user server*.

[1] *iload* requests of one user will all originate at the same host, so there is no need to use network sockets.

5.2 flexykill

flexykill is a very simple but efficient script for controlled run time limitation on demand. The limitation of the maximal run time can easily be done using the *limit* command of the C Shell. However, using only this feature, no adaption to the actual needs of the whole system is possible later on. This is because the limitations cannot be changed after starting a process. So this technique is not flexible at all. If there are no further proof tasks waiting to be scheduled (actually, no new sub–tasks have been created, e.g. if the user thinks about further steps), "more" time can be given to the running sub–provers. Otherwise, if there are many proof tasks waiting to be scheduled, the time given to the running processes should be "reduced" some way. So we decided to create a tool, which has to

1. determine the processes on a given list of hosts belonging to a certain prover expert,
2. note the process IDs of these processes,
3. terminate all noted processes which have already been running for more than a certain amount of time (the so called *flexy time*), if required.

The determination of processes belonging to an expert is done using *ps* and *grep*ping for a given pattern. Then, the determined process IDs will be integrated into short shell scripts for killing these processes on their hosts (using *rsh* and *kill*). These shell scripts get the canonical name `.flexykill.<JOBNR>` according to the job number `<JOBNR>` of the expert introducing *flexykill*. Then *flexykill* sleeps for a user definable number of seconds (*flexy time*). Afterwards the `.flexykill.<JOBNR>` file will be renamed to `flexykill.<JOBNR>`, making them accessible for execution, and *flexykill* terminates. If there is a need for termination of experts before their maximal run time, *flexykill* initiates the execution of as many `flexykill.<JOBNR>` files as necessary.

6 Assessment and Planned Improvements

In the literature, we find many systems organizing load distribution. Our solution may appear simple when compared to systems like *Dynasty* or *ALDY* as they are described in [9]. But these universal systems mostly seem too complex for our aims, cannot easily be tailored for our needs, or are available only as prototypes. To allow a comparison tables 1 to 3 classify our system according to the scheme given in [9].

Using the described algorithm for the load distribution problem has caused a significant improvement of the performance of the ILF system. The subjective impression

of persons working with ILF (and who did not know what if anything at all had been changed) was that the system responds much faster than before. Although the current implementation has worked well for some years, we are aware that it has a couple of shortcomings and we would like to improve it.

We conclude by listing some problems for future work. We believe that answers to theses questions will lead to such an improvement.

1. *avenrun*$[0]$ has been criticized (e. g. [12]) as being not appropriate to compare workload in a heterogeneous environment. We choose it because it can be obtained easily and without super user privileges.

2. A number of existing systems (e. g. *shortest expected delay* [11] or *CoCheck* [9]) uses a load index of $\alpha \cdot (1 + \textit{avenrun})$. This value seems to reflect better our intentions (find the host which will finish first) than the index $\alpha \cdot \textit{avenrun}$ we have been using.

3. It is possible to define a load index that takes into account other factors that might be important, i.e. the available memory, the used swap space and so on. Or, instead of combining the different factors into a single value, is it possible (and useful) to define a comparison algorithm for several of these factors?

4. Should interactive users be taken into account when determining the workload? One would think so, but how?

5. Are there adequate methods to measure the relative performance of hosts with different hardware other than running several provers and comparing the run times (which leads to a well–founded method of comparing the workload)?

6. Since we are going to replace rsh by PVM as the method of starting processes on remote hosts: We look for stable implementations of a PVM resource manager other than the default one (which uses round robin).

Objectives	
domain specific	yes
intent	adaptive
function	assignment
Integration Level	
monitoring	operating system
strategy	runtime
mechanisms	runtime
compiler support	none
Structure	
problem structure	independent
subproblems	processes
targets	processes

Table 1: General classification

Load Index Properties	
index dimension	single–dimensional
index type	service
aggregation time	comprehensive
aggregation space	system related
Load Value Composition	
index combination	none
Model Usage Policies	
index measurement	on demand
index propagation	after measurement
model adaptivity	fixed

Table 2: Load model classification

System Model	
Model Flavor	fairness
Target Topology	NOW
Entity Topology	non-interacting
Transfer Model	
Transfer Space	systemwide
Transfer Policy	non-preemptive
Information Exchange	
Information Space	systemwide
Information Scope	complete
Coordination	
Decision Structure	centralized
Decision Mode	cooperative
Participation	partial
Algorithm	
Decision process	dynamic
Initiation	central
Adaptivity	fixed
Cost Sensitivity	none
Stability control	partial

Table 3: Strategy classification

References

[1] J. Avenhaus, J. Denzinger, and M. Fuchs. DISCOUNT: A System for Distributed Equational Deduction. In *Proc. RTA-6*, pp. 397–402, Springer, 1995.

[2] P. Baumgartner and U. Furbach. PROTEIN: A Prover with a Theory Extension Interface. In *Proc. CADE-12*, pp. 769–773, Springer, 1994.

[3] W. Bibel, S. Brüning, U. Egly, and T. Rath. KoMeT, system description. In *Proc. CADE-12*, pp. 783–787, Springer, 1994.

[4] B. I. Dahn, J. Gehne, Th. Honigmann, L. Walther, and A. Wolf. Integrating Logical Functions with ILF. Preprint 94–10, Technical Report, Humboldt University Berlin, Department of Mathematics, 1994.

[5] B. I. Dahn, J. Gehne, Th. Honigmann, and A. Wolf. Integration of Automated and Interactive Theorem Proving in ILF. In *Proc. CADE-14*, pp. 57–61, Springer, 1997.

[6] B. I. Dahn and A. Wolf. A Calculus Supporting Structured Proofs. *Journal for Information Processing and Cybernetics (EIK)*, (5–6): pp. 261–276, 1994.

[7] C. Goller, R. Letz, K. Mayr, and J. Schumann. SETHEO V3.2: Recent Developments. In *Proc. CADE-12*, pp. 778–782, Springer, 1994.

[8] W. McCune. Otter 2.0. In *Proc. CADE-10*, pp. 663–664, Springer, 1990.

[9] T. Schnekenburger and G. Stellner (eds.). Dynamic Load Distribution for Parallel Applications. Teubner-Texte zur Informatik, vol. 24, B. G. Teubner, 1997.

[10] W. Reif. The KIV-System: Systematic Construction of Verified Software. In *Proc. CADE-11*, pp. 753–757, Springer, 1992.

[11] B. Schnor, S. Petri, R. Oleyniczak, and H. Langendörfer. Scheduling of Parallel Applications on Heterogeneous Workstation Clusters. In *Proc. PDCS-96*, vol. 1, pp. 330–337, ISCA, 1996.

[12] S. Stille. Lastbalancierung in verteilten Systemen. Masters thesis, Braunschweig University of Technology, IBR, 1993.

[13] C. Weidenbach, B. Gaede, and G. Rock. SPASS & FLOTTER, Version 0.42. In *Proc. CADE-13*, pp. 141–145, Springer, 1996.

A Decentral, Adaptive Load Sharing Approach for Workstation Clusters using Fuzzy Logic

Thomas Böhme

Department of Mathematics
Technical University Ilmenau
D-98684 Ilmenau, PF 10 0565, Germany
phone: +49 3677 69 3630, fax: +49 3677 69 3206
E-mail: tboehme@theoinf.tu-ilmenau.de

Herwig Unger

Department of Computer Science
University of Rostock, Rostock
D-18051 Rostock, Germany
phone: +49 381 498 3362, fax: +49 381 498 3366
E-mail: hunger@informatik.uni-rostock.de

Abstract

This paper presents a load sharing heuristic for distributed computing on workstation clusters. The approach is novel in that it combines the use of predicted resource requirements of processes (CPU-time, memory requirements, density of the I/O-stream) and a fuzzy logic controller which makes the placement decision. The heuristic is distributed, i.e. each node runs a copy of the prediction and load sharing code, and its implementation is based on PVM. Using a benchmark program (Choleski factorization) experiments were conducted to compare the proposed heuristic against standard PVM and an older version of the presented heuristic without the fuzzy logic controller.

1 Introduction

Message passing systems such as PVM, Express, Linda and P4 offer the opportunity to use workstation clusters as an inexpensive alternative for distributed computing. However, to enhance the performance of such systems "additional system management software" is needed [11]. An important step toward this goal is to design algorithms for the distribution of processes among the nodes of a cluster. During the past decade a lot of work has been done in this field. (For an overview see [1], [9] and [11].)

The present contribution deals with a load sharing heuristic. In general terms, a load sharing algorithm works as follows: When a process p arrives for execution, the algorithm assigns p to a node of the cluster in such a way that the performance is good in some sense.

In order to enable such an action the load sharing algorithm needs to "know" the actual load of each node of the cluster. One solution is to poll all nodes of the cluster periodically. Since the load changes rapidly this has to be done frequently what causes much additional load and consequently, reduces the system's performance. Furthermore, it is not hard to see, that even frequent polling cannot guarantee the accuracy of a placement decision made without any knowledge about the system's future behaviour ([5], [8], [14]). A better solution is, therefore, to use a prediction. The proposed load sharing heuristic uses a simpler prediction scheme than the one used in [5] which is outlined in section 2.1.

The load of a node depends on several load factors such as the length of the CPU ready queue, the CPU-times of the processes in the CPU queue, the amount of memory in use, and the density of the I/O-stream. Since there is no suitable mathematical model that relates these factors with the performance of the system, most of the load sharing algorithms in the literature use a single figure to describe the load and distribute the processes in such a way that about the same load is assigned to each node of the cluster. The proposed heuristic uses a different approach. Three load factors are combined by a fuzzy logic controller which picks the node a new process is assigned to. A different approach which also involves a fuzzy logic controller is described in [6].

A load sharing system, called ALICE [1], using the proposed load sharing heuristic was implemented based on PVM. Here only those processes are subject to the placement decision that are created by a call of the `pvm_spawn`-function. An implementation on a command shell level and for MPI is in preparation.

An experiment has been conducted to compare the proposed load sharing heuristic against standard PVM 3.3.11 and an older version of the presented heuristic without the fuzzy logic controller. The results of the experiment are described in section 3.

2 Heuristic

The load sharing system ALICE consists of two parts - the predictor and the fuzzy logic controller (see fig.1).

[1] **A**daptive **L**oad Balancing System for **I**nhomogeneous **C**omputing **E**nvironments

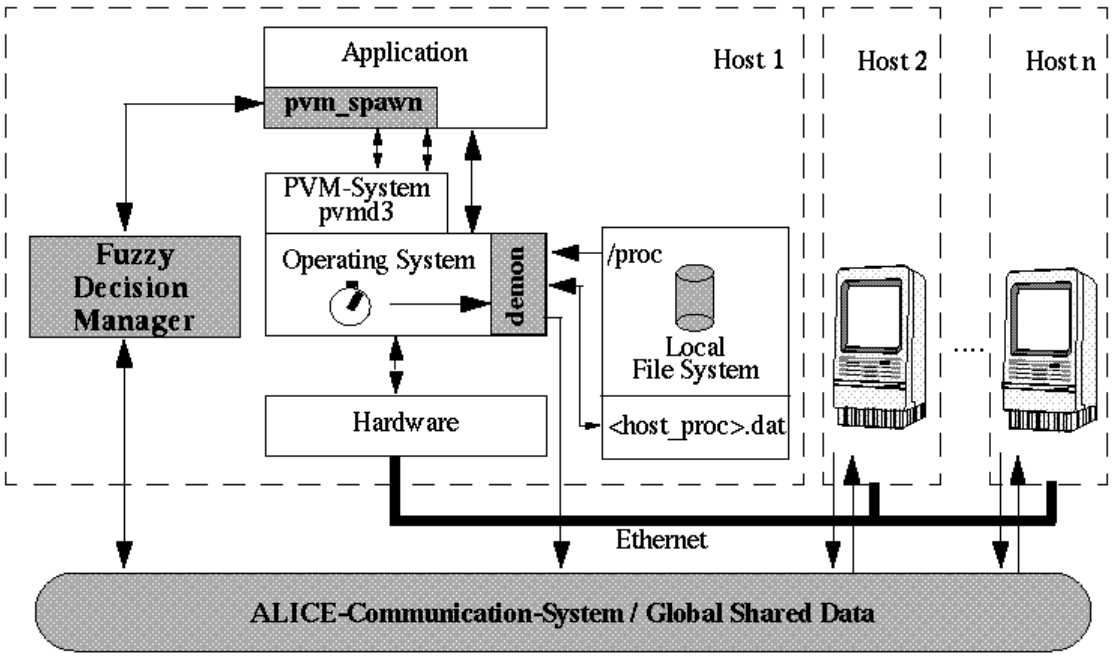


Figure 1: Structure of the Load Sharing System

The predictor is realized by a demon (called the **l**oad **p**rediction demon - llp-demon) which is incorporated in the operating system of each node. Since the decision manager must be invoked only if the `pvm_spawn`-function is called, the `pvm_spawn`-function has been substituted by a new function that includes the fuzzy logic controller.

In general the resource requirements of a process depend on the program the process executes as well as on the owner of the process. Furthermore, due to differences in the architectures of the processor nodes, one and the same process can have different resource requirements if it runs on different nodes. Therefore, a process p is identified by its program and its owner, and predictions of its resource requirements $\tau(p,i)$ (CPU-time), $\mu(p,i)$ (memory requirements) and $\omega(p,i)$ (density of the I/O-stream) are computed separately for each node i. Using these predictions of resource requirements of processes predictions of three load indexes of each node i are computed:

- $NIT(i)$ - a prediction of the time the processor is idle for the next time
- $MEM(i)$ - a prediction of the amount of memory in use
- $IO(i)$ - a prediction of the remainig I/O-capacity

The load sharing heuristics works as follow. When `pvm_spawn` is called on some processor node i, an identification of the program and the owner is sent to any other node. Every node j then returns its load indexes $NIT(j)$, $MEM(j)$ and $IO(j)$ as well as the predictions of the respective resources requirements $\tau(p,j)$, $\omega(p,j)$ and $\mu(p,j)$ of p.

These data are fed to the fuzzy decision manager which identifies the node p is executed on.

2.1 Prediction Scheme and llp-deamon

The llp-deamon consists of two parts (see fig.2.).

One part builds and maintains a list of the processes running on the respective node. In [9] it is shown that only processes that run longer than 1/3-1/2 sec generate a significant load and therefore, only processes running at least 1/2 sec are included in the list. The second part of the llp-deamon computes the predictions of the resource requirements of processes and the load indexes. It does not depend on the used platform what makes it easy to adapt the heuristic to other platforms.

The predictions of the resource requirements of a process p are weighted means of the actual resource requirements of the last N executions of this process on the respective node. In (1) the computation is shown for $\tau(p,i)$. $\omega(p,i)$ and $\mu(p,i)$ are computed in a similar way.

$$\tau_{new}(p,i) \quad = \quad e^{-\lambda}\tau_{old}(p,i) + \frac{t}{N} \tag{1}$$

The computation is done in a recursive way. $\tau_{new}(p,i)$ and $\tau_{old}(p,i)$ denote the new and the old value of $\tau(p,i)$, respectively, and t is the CPU-time of the last execution of p on the node i. The weight $e^{-\lambda}$ serves to limit the influence of the older values on the prediction, and is subject to an adaptation process.

The size of the memory in use, $MEM(i)$, of the processor node i is estimated by the sum of the memory requirements of those processes which are currently running on i. Using the predictions of the CPU-times of the processes running on the node i, $NIT(i)$ is computed subject to the following rules.

(i) At the start of the system $NIT(i)$ is zero.

(ii) When a new process p is started on i, $NIT(i)$ is updated by adding the prediction $\tau(p,i)$ of CPU-time of p.

(iii) $NIT(i)$ is periodically decreased by the difference of the allocated CPU-time of the processes since the last update.

(iv) If a process terminates earlier or uses the CPU longer than predicted, $NIT(i)$ is corrected.

The prediction of the remaining capacity of the I/O-channel is computed similarly.

2.2 Fuzzy Logic Controller

The basic idea of the fuzzy logic controller is the following. In order to determine where to run a new process p, for each node i a real number $l(i)$ is computed in such a way that

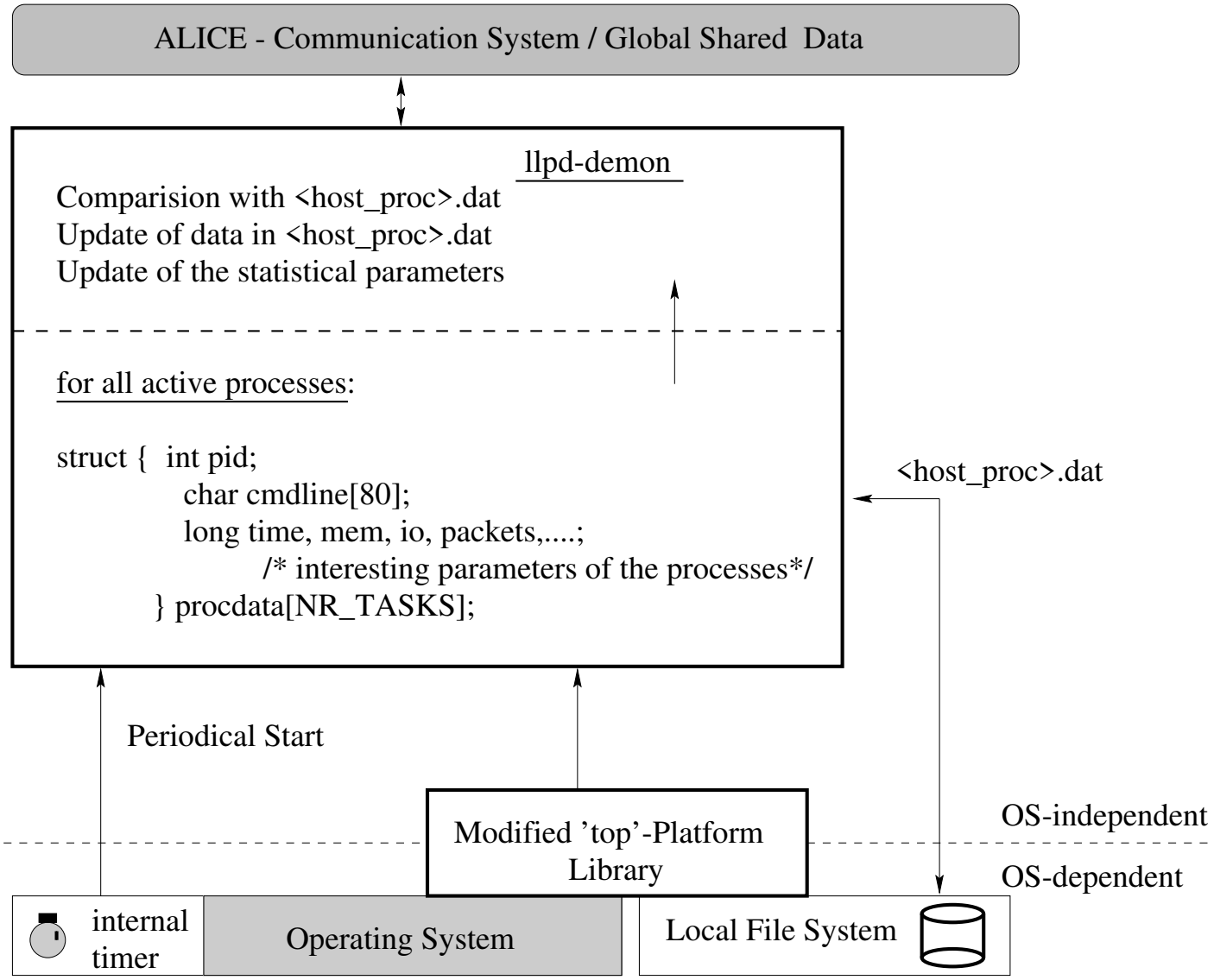


Figure 2: Structure of the llp-deamon

$l(i)$ is the greater the better the process p will fit on the node i. p is assigned to the node with the greatest $l(i)$. The main steps of the computation of $l(i)$ are outlined below.

1) Fuzzifications $NIT^f(i)$, $MEM^f(i)$, $IO^f(i)$ and $\tau^f(p,i)$, $\mu^f(p,i)$, $\omega^f(p,i)$ of the load indexes $NIT(i)$, $MEM(i)$, $IO(i)$ and the resource requirements $\tau(p,i)$, $\mu(p,i)$, $\omega(p,i)$, respectively, are computed.

2) Using a rule base $\mathcal{R}_1, ..., \mathcal{R}_s$, for each rule $\mathcal{R}_j$ a figure $l_j(i)$ is computed.

3) $l(i)$ is obtained from $l_1(i)$,...,$l_k(i)$ by a defuzzification algorithm.

It is well beyond the scope of this article to give an introduction to fuzzy logic (For more on fuzzy logic and fuzzy logic controllers the reader is referred to [16].) Nevertheless, some remarks should be made.

A fuzzy set in some set M is characterized by a function f from M to the closed interval $[0,1]$. f is called the membership function and, for an element x of M, $f(x)$ is considered to be the degree of membership of x in the fuzzy set. In order to define a fuzzification of a number y, finitely many fuzzy sets (called linguistic terms) are defined in the set y is taken from. If f_1,...,f_r are the membership functions of the linguistic terms, the fuzzification y^f of y (with respect to the linguistic terms f_1,...,f_r) is the vector $y^f = (f_1(y), ..., f_r(y))$. The linguistic terms are usually associated with linguistic labels such as *small*, *medium*, *large* etc.. Here five linguistic terms *very low*, *low*, *medium*, *high* and *very high* are defined for each input variable. The linguistic terms used for the output variable $l(i)$ are *very bad*, *bad*, *medium*, *good* and *excellent*.

The rule base consists of rules of the following type

$\mathcal{R}_k$: *If* $NIT(i)$ is very low *and* $MEM(i)$ is low *and* $IO(i)$ is medium *and* $\tau(p,i)$ is high *and* $\mu(p,i)$ is medium *and* $\omega(p,i)$ is low, *then* $l(i)$ is good

Given the fuzzifications of the input variables for each rule a figure is computed that describes the degree as to which the specific rule applies to the input. These figures define a fuzzy set which is then defuzzified using the center of gravity method. For details see [16].

The knowledge about the controlled system is encoded in the definition of the linguistic terms and the rules of the rule base. Thus the correctness of the decisions made by the fuzzy logic controller depends directly on the quality of the knowledge base formed by the linguistic trems and the rule base. In the present version of our system the knowledge base is based on experience.

3 Experiments

The above described load sharing heuristic was implemented on a cluster consisting of three PC's (i386 and i486) running under LINUX and connected via EtherNet. A distributed benchmark program performing a Cholesky factorization of a matrix taken from

[13] was run with standard PVM, an older version of the presented load sharing heuristic ([14]) and the proposed heuristic. The CPU-time and memory requirements of the processes created by the execution of this program are strongly influenced by the calling parameters.

system	factor [sec]	fsolve [sec]	bsolve [sec]	total [sec]
PVM 3.3.11	10.0	14.1	12.0	30.4
old version	10.0	9.1	9.2	24.2
new version	9.8	8.8	8.9	23.8

Table 1: Additional processes with medium memory use and high CPU-load

system	factor [sec]	fsolve [sec]	bsolve [sec]	total [sec]
PVM 3.3.11	9.1	7.0	7.1	21.2
old version	9.1	7.3	7.4	19.9
new version	7.0	7.2	5.0	17.1

Table 2: Additional processes with high memory use and low or medium CPU-load

The nodes also run additional processes of different characteristics. The obtained results are shown in table 1 and table 2. It turned out that, especially if the additional processes have high memory requirements and low or medium CPU-time, the new load sharing heuristic achieves significantly better results than the older version without fuzzy logic.

4 Concluding Remarks

Using ideas from [4], it is planned to utilize a genetic algorithm to adapt the knowledge base to the specific requirements of the cluster in a subsequent version of the presented heuristic. Presently, a more sophisticated simulation of our system is in preparation. Furthermore, it is planned to apply the ideas exhibited in this paper to very large distributed systems.

References

[1] M. A. Baker, G. C. Fox, H. W. Yau, "A Review of Commercial and Research Cluster Management Software", Technical Report, Syracuse University New York, 1996, http://www.cs.rice.edu/ lwlutz/NHSEmain0.html.

[2] W. Becker, "Dynamische, adaptive Lastbalancierung für große heterogen konkurrierende Anwendungen", PhD-Thesis, University of Stuttgart, 1995.

[3] K. P. Bubendorfer, "Resource Based Policies for Load Distribution", PhD-Thesis, University of Wellington, 1996.

[4] O. Cordón, F. Herrera, et al., "Genetic algorithms ans fuzzy logic in control processes", Technical Report #DECSAI-95109, ETS de Ingeniería Informática, Universidad de Granada, March, 1995

[5] M. Devarakonda, R. K. Iyer, "Predictability of process resource usage: A measurement-based study of UNIX", *IEEE Trans. Software Eng.*, vol.15, no.12, Dec. 1989.

[6] S. Dierkes, L. Frese, "Load balancing with a fuzzy decision algorithm: Description of the approach and first simulations", Technical Report Universitt Dortmund, 1995.

[7] D. H. J. Epema, et al., "A world-wide flock of Condors: load sharing among workstation clusters", TR-95-130, TU Delft, 1995.

[8] K. G. Goswami, M. Devarakonda, R.K. Iyer, "Prediction-Based Dynamic Load Sharing Heuristics". *IEEE-Trans. Parallel and Distributed Systems*, Vol. 4(6), pp. 638-648, 1993.

[9] M. Harchol-Balter, A. B. Downey, "Exploiting Process Lifetime Distributions for Dynamic Load Balancing", Technical Report TR-95-021, ICSI, Berkeley, 1995.

[10] G. Hipper, D. Tavangarian, "A new Architecture for Efficient Parallel Computing in Workstation Clusters", Int. Conference HPCN Challenges in Telecomp and Telecom: Parallel Simulation of Complex Systems and Large-Scale Applications, Delft, Netherlands, 1996.

[11] J. A. Kaplan, M. L. Nelson, "A Comparision of Queueing, Cluster and Distributed Computing Systems", NASA Technical Memorandum 109025, Langley Research Center, 1993.

[12] P. Mehra, "Automated Learning of Load Balancing Strategies for a Distributed Computer System", Technical Report, University of Illinois, 1993.

[13] B. K. Schmidt, V. S. Sunderam, "Empirical Analysis of Overheads in Cluster Environments", in: *Concurrency: Practice and Experience*, 1993.

[14] H. Unger, H. Unger, G. Hipper, "Stochastics and Learning Methods for Load Balancing in a Distributed Workstation Cluster System", in: Proc. of the Int. Conference of Applied Informatics, Innsbruck, 1997.

[15] G. Wilhelms, "Dynamische adaptive Lastverteilung für PVM mittels unscharfer Benutzerprofile", PhD-Thesis, Univ. Augsburg, 1994.

[16] H.-J. Zimmermann, *Fuzzy Set Theory and Its Application*, Kluwer Academic Publisher, Boston, Dordrecht, London, 1991.

Load Balancing for Parallel Adaptive FEM *

Ralf Diekmann,[1] Frank Schlimbach,[1] and Chris Walshaw[2]

[1] Department of Computer Science, University of Paderborn, Germany.

[2] School of Computing and Mathematical Sciences,

The University of Greenwich, London, UK.

Extended Abstract

Abstract

We present a dynamic distributed load balancing algorithm for parallel, adaptive finite element simulations using preconditioned conjugate gradient solvers based on domain-decomposition. The load balancer is designed to maintain good partition aspect ratios. It calculates a balancing flow using different versions of diffusion and a variant of breadth first search. Elements to be migrated are chosen according to a cost function aiming at the optimization of subdomain shapes. We show how to use information from the second step to guide the first. Experimental results using Bramble's preconditioner and comparisons to existing state-ot-the-art load balancers show the benefits of the construction.

1 Introduction

Finite Elements (or Finite Differences or Finite Volumes) can be used to numerically approximate the solutions of partial differential equations (PDEs) describing, for example, the flow of air around a wing or the distribution of temperature on a plate which is partially heated [2, 30]. The domain on which the PDE is to be solved is discretized into a mesh of *finite elements* (triangles or rectangles in 2D, tetrahedra or hexahedra in 3D) and the PDE is transformed into a set of linear equations defined on these elements [30]. The coupling between equations is given by the neighborhood in the mesh. Usually, iterative methods like Conjugate Gradient (CG) or Multigrid (MG) are used to solve the linear equational systems [2].

The quality of solutions obtained by such numerical approximation algorithms depends heavily on the discretization accuracy. In particular, in regions with steep solution gradients, the mesh has to be refined sufficiently, i.e. the elements should be small

*This work is partly supported by the DFG-Sonderforschungsbereich 376 "Massive Parallelität: Algorithmen, Entwurfsmethoden, Anwendungen".

in order to allow a good approximation. Unfortunately the regions with large gradients are usually not known in advance. Thus, meshes either have to be refined regularly resulting in a large number of small elements in regions where they are not necessary, or the refinement takes place during the calculation based on error estimates of the current solution [28]. Clearly, the second variant should be favored because it generates fine discretizations only in those regions where they are really needed. The resulting solutions have the same quality as if the mesh was refined regularly but the number of elements, and thus the time needed to determine the solution, is only a small fraction of the regular case.

The parallelization of numerical simulation algorithms usually follows the single-program multiple-data (SPMD) paradigm: The same code is executed on each processor but on different parts of the data. This means that the mesh is partitioned into P subdomains where P is the number of processors, and each subdomain is assigned to one processor [12]. Iterative solution algorithms mainly perform local operations, i.e. new values at certain parts of the mesh only depend on the values of nearby data (modulo some global dependencies in certain iteration scemes which we neglect here as they are independent of the way the mesh is partitioned). Thus, the parallelization requires communication mainly at the partition boundaries.

The efficiency of such a SPMD parallelization mainly depends on the equal distribution of data on the processors. As the individual iteration steps are synchronized by a data exchange at partition boundaries the time the processors need to perform one iteration step should be the same on each processor. This can be achieved by splitting the mesh into equal sized subdomains. If each processor contains the same number of elements, the time each needs for one iteration will be approximately the same [12].

In the case of adaptive refinement, the distribution of data on the processors becomes unbalanced if the number of newly generated elements is not the same on each processor (which usually is the case with solution adaptive refinement). Therefore, the partition has to be altered in order to re-establish a balanced distribution.

As already mentioned, iterative solvers are easy to parallelize and they mainly require communication of data at partition boundaries. Thus, most existing FEM partitioners and load balancing algorithms aim at minimizing the boundary length which is a valid measure for the time needed to complete a communication step.

A powerful method to speed up linear solvers is to improve the condition of the equational system [2, 5, 30]. *Domain Decomposition* methods (DD) belong to the most efficient preconditioners. Unfortunately, their speed of convergence depends on the way the partition is generated. The shape (the Aspect Ratio) of subdomains determines the condition of the preconditioned system and has to be considered if partitions are changed due to load balancing.

PadFEM (**P**arallel **ad**aptive **FEM**) is an object-oriented environment supporting parallel adaptive numerical simulations [3, 24]. *PadFEM* includes graphical user interfaces, mesh generators (2D and 3D, sequential and parallel) [22], mesh partitioning algorithms [12], triangular and tetrahedral element formulations for Poisson and Navier-Stokes problems [30], different solvers (especially DD-PCGs) [2, 5], error es-

timators [28], mesh refinement algorithms [20], and load balancers [12].

The load balancing module in *PadFEM* determines the amount of load to move between different subdomains in order to balance the distribution globally. The neighborhood between subdomains defines the *cluster-* or *quotient-graph* [12]. On this graph the algorithm determines a *balancing flow* which tells the partitions how many data items (i.e. elements) they have to move to each of their neighbors. This balancing flow calculation can optionally use different kinds of diffusive methods [4, 9], in particular first and second order diffusion iterations [13, 15], and a heuristic for convex non-linear min-cost flow [1].

In a second step, the elements to be moved are identified. When choosing these elements, the load balancer tries to optimize partition Aspect Ratios in addition to load balance. The elements at partition boundaries are weighted by a cost function consisting of several parts. The migration chooses elements according to their weight and moves them to neighbors.

The element weight functions are used to "guide" the balancing flow calculation. We consider the existing partition and define weights for the edges of the cluster graph expressing a kind of "cost" for moving elements over the corresponding borders. The aim is to avoid a large flow demand between partitions whose common border is very small.

The next section gives an introduction into the field of balancing adaptive meshes. Section 3 describes the element migration strategies used within *PadFEM*. This second step in load balancing is presented first, as parts of it are used in Section 4 where the balancing flow calculation is described. Section 5 finishes the paper with a number of results about Aspect Ratios and iteration numbers of the DD-PCG solver within *PadFEM*. Comparisons are made to *Jostle*, the Greenwich parallel partitioning tool [29].

2 The Problem

In most cases, the computational load of a finite element problem can be measured in terms of numbers of elements. Thus, a load balanced parallel execution requires a partition of the mesh into subdomains containing the same amount of elements each. Such an initial partition can be generated using any of the existing very efficient graph partitioning algorithms (see e.g. [12] for an overview) or by generating the mesh in parallel [7].

If the mesh is refined adaptively, the existing partition will probably become unbalanced. Scalable load balancing algorithms can take benefit from an existing partition by moving elements on the *quotient graph* defined by the adjacency between subdomains. The quotient graph contains one node for each partition. Edges denote common borders between the corresponding partitions. Figure 1 shows an unbalanced partition of a simple mesh (domain "Square") into four subdomains (left) and the resulting quotient graph (center). Any reasonable load balancing taking the existing distribution into account has to move data (elements) via edges of this graph. We add weights to nodes

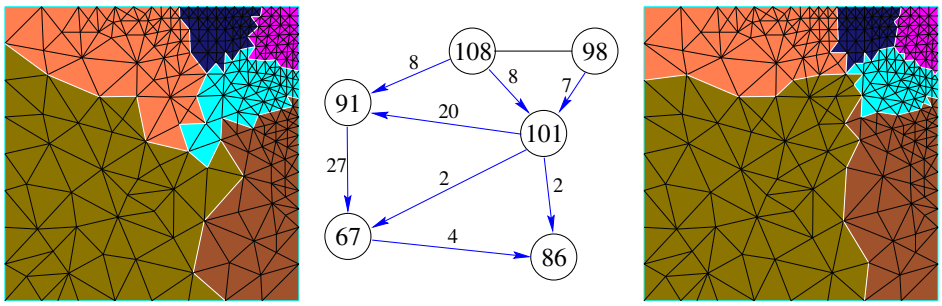


Figure 1: An unbalanced partition (left), the corresponding quotient graph with balancing flow (center), and the resulting balanced partition after element migration.

of the graph according to their load and determines a balancing flow on the edges. This flow has the properties that after shifting a corresponding amount of load (elements) between adjacent subdomains, the partition is globally balanced. The flow is given as edge labels in Figure 1. It can be determined by using network flow algorithms or by local iterative methods such as Diffusion or Dimension Exchange [4, 9].

In a second step - the element migration phase - the load is really moved. This phase is given the flow on the quotient graph and its task is to choose elements whose movements fulfill the flow demands. The choice of elements may consider additional cost criteria such as minimizing the boundary length or optimizing the shape of subdomains (which is actually done in *PadFEM*).

Most existing implementations of adaptive grids perform load balancing by repartitioning. This is sometimes the only possibility for parallelizing existing numerical codes whose data structures may support adaptivity but certainly not a movement of elements between subdomains of a distributed memory implementation. Attempts to minimize the amount of necessary data movement can be found in [23].

The tools *Metis* [21] and *Jostle* [29] are designed to support partitioning and load balancing of adaptive mesh calculations in parallel. Both use Hu&Blake's algorithm [18] for determining the balancing flow and both use a multi-level strategy for shifting elements, where the coarsening is done inside subdomains only. The tools differ in implementation details. *Jostle* uses the concept of *relative gain* optimization at partition boundaries. *Metis* uses so called *enforcement levels* and greedy refinement. Both tools optimize according to *Cut*, Aspect Ratio is not directly considered.

The *SUMAA3D Project* at Argonne is especially designed to support large-scale scientific computations in parallel. The mesh partitioning in *SUMAA3D* is based on *Unbalanced Recursive Bisection* (URB) which aims at generating better shaped partitions than standard recursive bisection [19]. By moving partition boundaries, the tool is also able to balance load in a parallel adaptive setting.

The group in Leeds have designed an environment supporting parallel adaptive mesh computations [25]. The load balancer uses a variant of multi-level Diffusion [17] which minimizes the amount of data movement.

Load balancers particularly designed to optimize the subdomain Aspect Ratio can be found in [11, 16, 27]. The tool *PAR*^{2} was originally constructed as parallel partitioner, but it can also be used within an adaptive environment [11]. The method described in [16] is an iterative partitioner trying to improve the subdomain Aspect Ratios in a number of steps. In this work, DD-PCG solvers are used, but not within an adaptive environment. Other attempts try to optimize subdomain shapes by using Meta Heuristics such as Simulated Annealing or Tabu Search [27].

3 Element Migration

The basic idea of the element migration phase is to rate elements at partition boundaries according to a cost function and to greedily choose elements to be moved to neighboring processors based on this rating, until the necessary load has been moved. The migration phase starts with a valid balancing flow determined by the flow calculation phase which will be described in Section 4. For now, let us assume that the flow is represented as values x_{ij} on edges (i, j) of the quotient graph (meaning that x_{ij} elements have to be moved from partition i to partition j). The aim of the migration phase is to balance the load in such a way that the Aspect Ratio of subdomains is maintained, improved, or, at least, worsend as less as possible.

For a subdomain P_i, let A_i be its area[1] and B_i be the length of its boundary. We define the Aspect Ratio AR_i of i by $AR_i = B_i^2/(16A_i)$. This definition of the Aspect Ratio is quite different from the usual one found in the literature [11] but in our experience we found that it expresses the desired tasks of shape optimization best [14].

The simplest way to rate elements at partition boundaries is to count the change in cut size (of the element graph) when moved to other partitions. For an element e, let $\mathit{ed}(e, i)$ be its number of edges adjacent to elements in P_i. Then, the change in cut when e is moved from P_i to P_j is given by $\Gamma_{cut}(e, i, j) = \mathit{ed}(e, i) - \mathit{ed}(e, j)$. For many FEM meshes constructed of triangles, the resulting element graph has a maximal degree of three and the change in Γ_{cut} is at most 1. It is known [6] that local graph partitioning algorithms perform badly on degree-3 graphs. So it is no surprise that this element rating function does not produce good results. We will see comparisons later on. Nevertheless, it is used in several applications, and if it is combined with other sophisticated methods like multilevel schemes, it can still work very well [29].

The most promising rating functions consider the distances of elements from the center of their subdomain. If all boundary-elements have the same distance to this center, then the domain is a circle and has a good Aspect Ratio. Elements lying far away from the center might be good candidates to move. Let $\mathit{dist}(e, i)$ be the distance of element e to the center of subdomain P_i. The function $\Gamma_{rd}(e, i, j) = \mathit{dist}(e, i) - \mathit{dist}(e, j)$ is called the *relative distance* of element e. The idea behind this function is to prefer elements which are far away from the center of their own subdomain and,

[1] Throughout this paper, we are dealing with 2D domains, only.

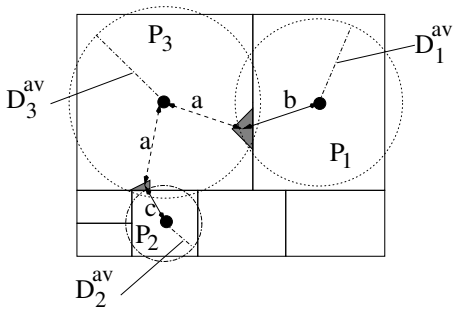


Figure 2: Scaled Distances.

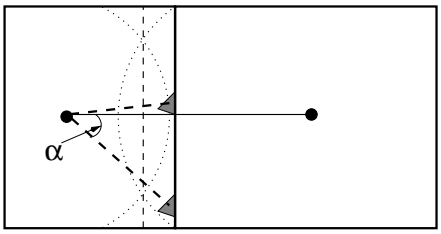


Figure 3: Angle.

additionally, near to the center of the target partition. Such element rating functions can be found in some implementations [8, 11]. The problem is that they do not account for the real physical size of a domain. A large partition might easily move elements to a small one, even if this does not improve the Aspect Ratio of any of them. We can define D_i as the average distance of the centers of all elements of subdomain P_i from its center. D_i can be used to normalize the distance function Γ_{rd} [11]. $\Gamma_{srd}(e,i,j) = \mathit{dist}(e,i)/D_i - \mathit{dist}(e,j)/D_j$ is called the *scaled relative distance function*. Figure 2 shows the advantage of Γ_{srd} in contrast to Γ_{rd}. Without scaling, P_3 would prefer to move elements to P_2, whereas a move towards P_1 would improve the Aspect Ratios of P_1 and P_3.

If we consider the shape of a common border between two subdomains, the best would be to have a straight line perpendicular to the interconnection of the centers of both. If the border is circular in any direction, then one of the domains becomes concave which is not desirable. If elements are moved between subdomains which are very different in size, then the influence of the smaller one on Γ_{srd} becomes larger with increasing distance from the line connecting both centers. Thus, elements far away from this line are more likely to be moved than those lying on or nearby this line. The effect can be observed in Figure 4 where the domain "Cooler" is refined in 10 steps from initially 749 elements to finally 7023. The refinement mainly takes place at the upper part of the domain and the subdomains lying in the corresponding region give up elements according to Γ_{srd}.

To avoid this effect, we superimpose Γ_{srd} by another function $\Gamma_\alpha(e,i,j) = \cos\alpha$,

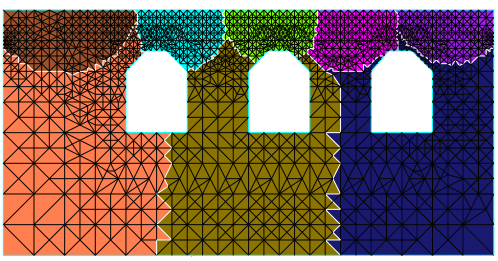


Figure 4: Migration according to Γ_{srd}.

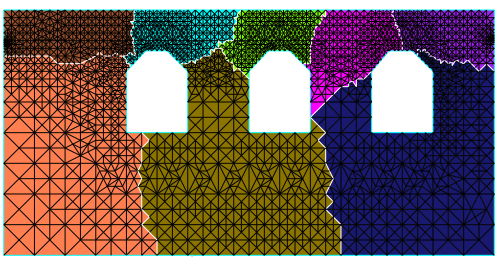


Figure 5: Migration according to $\Gamma_{srd} + \Gamma_\alpha$.

	AR		Cut	
	Max	Avrg	Max	Avrg
Γ_{cut}	2.03	1.66	190	128
Γ_{rd}	1.53	1.33	142	92
Γ_{srd}	1.47	1.29	144	90

Table 1: AR and cut obtained with different rating functions.

where α is the angle between the lines $C_i \leftrightarrow C_j$ and $C_i \leftrightarrow c_e$. Figure 3 shows the construction. The value of $\cos\alpha$ is large if the angle α is small, i.e. the superposition slightly increases the values of those elements which lie in the direction of the target partition. Figure 5 shows the effect on the same example of Fig 4. As desired, the borders between subdomains are much straighter and the target domains are less concave.

A first comparison of the different rating functions can be found in Table 1. It shows the maximum and average Aspect Ratios as well as the Cuts if the domain "Square" is refined in 10 steps from initially 115 elements to finally 10410. The values for Aspect Ratios given here and in the remaining of the paper are scaled to 1.0 being an "optimal" shape. Average values of ≤ 1.4 can be considered as acceptable, values of > 1.6 are unacceptable. It can be observed that Γ_{cut} is not only bad in optimizing AR but also in finding partitions with good cut. The explanation has already been given: As the graph has a maximal degree of 3, local methods fail. Γ_{rd} and Γ_{srd} are quite similar to each other because the difference in size of the subdomains is not very large in this example. The effects of Γ_α are not directly visible in terms of maximum or average AR. This kind of "cosmetic" operation is more visible in Figures 4 and 5.

4 Balancing Flow Calculation

4.1 Possible Solutions

Let $G = (V, E, w)$ be the quotient graph of a load balancing problem (cg. Fig. 1). G has $n = |V|$ nodes, $m = |E|$ and there are load values w_i attached to each node $i \in V$. The task of the balancing problem is to find a flow x_{ij} of load on edges $(i, j) \in E$ of the graph such that after moving the desired numbers of elements, the load is globally balanced, i.e. $w_i = \bar{w}\ \forall i \in V$. As the flow is directed, we need to define (implicit) directions for edges. We may assume that edges are directed from nodes with smaller numbers to nodes with larger numbers. A flow of $x_{ij} < 0$ on edge $(i, j) \in E$ then means that $-x_{ij}$ load has to be moved from node j to i.

Let $A \in \{-1, 0, 1\}^{n\times m}$ be the *node-edge incidence matrix* of G. The edge directions are expressed by the signs (from $+1$ to -1). The matrix $L = AA^T$ is called the *Laplacian matrix* of G. If we define $x \in \mathbb{R}^m$ to be the vector of flow values, and if we would like to minimize the total amount of flow, then we are searching for an x

minimizing $\sum |x_{ij}|$ such that $Ax = w - \bar{w}$. Such a problem is known as the *min-cost flow problem* [1].

The question of what are good cost criteria for such a flow is not easy to answer. It can be observed easily that any solution to to the min-cost flow problem shifts the flow only via shortest paths in the network. Thus, it often uses only a small fraction of the available edges which might not be desirable, especially when afterwards the shape of subdomains has to be considered.

A compromise between the conflicting goals of shifting not too many load but using all edges more or less equally might be the Euclidean norm of the flow defined as $\|x\|_2 = \sqrt{\sum x_{ij}^2}$. This measure turns the min-cost flow into a non-linear, but convex optimization problem which is still solvable in polynomial time. For this special type of problem, solutions can be found easily. Hu and Blake, for example, propose to use Lagrange multipliers for a solution of $Ax = w - \bar{w}$ [18]. The methods requires to solve $L\lambda = w - \bar{w}$ and the flow is afterwards given by $x_{ij} = \lambda_i - \lambda_j$. $L\lambda = w - \bar{w}$ is easy to solve in parallel, because L directly corresponds to the quotient graph G (in fact, L is topologically equivalent to G).
Other local iterative methods determine x directly by performing $x^t = \alpha A^T w^t$ and $w^{t+1} = w^t - Ax^t$. This iteration is similar to $w^{t+1} = Mw^t$ with iteration matrix $M = (I - \alpha L)$ and parameter $\alpha \leq$ *max_deg*(G). It is known as the *Diffusion Method* [4, 9]. Some schemes try to speed up the convergence of this local iterative method by using non-homogeneous iteration matrices M, optimal values of α [13], and over-relaxations like $w^{t+1} = \beta M w^t + (1 - \beta) w^{t-1}$ [15]. It can be shown that all these local iterative methods determine flows x with minimal $\|x\|_2$ [10].

4.2 Average Weighted Flow

The *Average Weighted Flow* algorithm (*AWF*) proceeds in two steps. It first determines (source/ sink/amount) triples giving pairs of processors and the amount of load to move between them. Afterwards, these flows are routed via paths in the graphs.

The first step starts from each source node searching for sinks nearby which are reachable via "good" paths in the graphs. For a path $p = v_1, \ldots, v_k$, we define the *path quality* $q(p) = \frac{1}{|p|} \sum_{i=1}^{k-1} \omega_{v_i v_{i+1}}$ as the average quality of edges on the path. As we are searching for paths with high quality, the task is to maximize $q(p)$. Unfortunately, this makes the problem intractable. If $q(p)$ were just the sum of the ω's, then we would search for longest paths without loops, an NP complete problem. Additionally, the averaging causes the loss of transitivity. A "best" path from i to j via k must not longer include the best paths from i to k and from k to j.

We use a variance of breadth first search (BFS) as a heuristic to find approximately "best" paths. The algorithm starts from a source and searches - in normal BFS style - for paths to sinks. A sink node has a "distance" from the source according to the quality of the best path via which it has been reached during the BFS. If a node is found again (due to circle closing edges), its "distance" is updated. The algorithm

	Square		Cooler		Smiley		Tower	
	Diff	*AWF*	Diff	*AWF*	Diff	*AWF*	Diff	*AWF*
Γ_u	1.351	1.340	**1.289**	**1.289**	1.387	1.411	1.341	1.320
Γ_{srd}	1.364	**1.320**	1.302	1.289	1.393	**1.340**	1.344	1.311
Γ_α	1.353	1.344	1.294	1.296	1.387	1.368	1.340	1.336
$\Gamma_{rd} + \Gamma_\alpha$	1.361	1.328	1.296	**1.289**	1.390	1.374	1.765	**1.300**

Table 2: AWF in comparison to Diffusion.

visits each edge at most once and, thus, has a running time of $O(m)$. For each sink node, all the paths from the source found during the search are stored together with their quality. For building the (source/sink/amount) triples, the load $w_i - \bar{w}$ of a source i is distributed (logically) to its "best" sinks, each of them filling up to $\bar{w}$. The time for this phase is dominated by the search. As there are at most n sources, the total running time is $O(n \cdot m)$.

The second phase routes the flow demands between source/sink pairs via paths in the network. The task is here to distribute the flow evenly over (a subset of) the possible paths from the source to the sink. The first phase already calculated possible paths. If a flow of x_{st} has to be shifted from a source s to a sink t and if there are two possible paths p_1 and p_2 between them, then the fraction of $\frac{q(p_1)}{q(p_1)+q_(p_2)}$ is shifted via p_1, the rest via p_2.

Table 2 shows a comparison of *AWF* and Diffusion. Again, several different element rating functions are chosen to guide the migration. The values in Table 2 are mean values of all results for a given problem and a given flow algorithm and edge weighing function. It can be observed that an improvement in the achievable AR of around 5% is possible by weighing the edges properly (Γ_u is again the non-weighted case). *AWF* behaves generally better than Diffusion, sometime the improvements are large ($\approx 25\%$), in most cases they are around 5%. Γ_{srd} turns out to be a good edge weighing function, and also the use of Γ_α helps sometimes.

5 Results

Figure 6 shows the development of AR, Cut and number of iterations of the DD-PCG solver over a number of refinement levels of the Domain "Square" (shown in Fig. 1) refined in a number of steps from 736 to 15421 elements. Experiments using regular meshes and regular domains show an increase in the number of iteration by one when the number of elements are doubled. For irregular meshes such as those used here, one can not hope for such a good convergence behavior. The left picture shows the behavior if Γ_{cut} is used as element rating function, the right if $\Gamma_{srd} + \Gamma_\alpha$ is used. It can be observed that the number of iterations is very unstable and generally larger if Cut is the optimization goal. If shape is optimized, the solver behaves much better.

Generally, AR expresses the development of the number of iterations much better

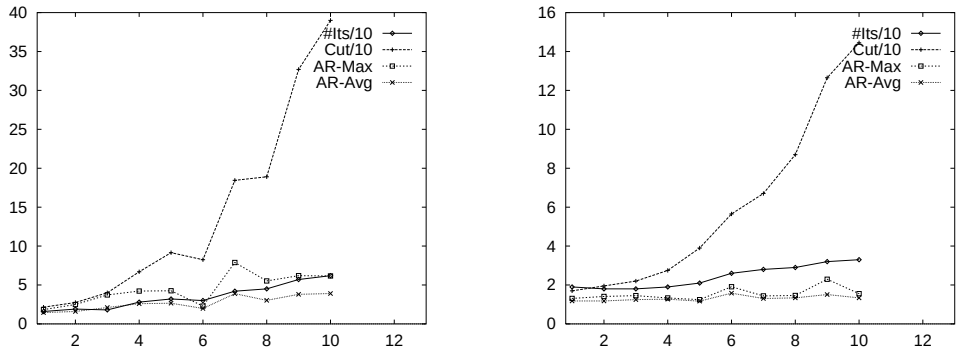


Figure 6: Development of AR, Cut and #Iterations for domain "Square".

than Cut. We conclude from the experiments that an optimization according to AR is reasonable.

We also compare the results of the AR-optimization in *PadFEM* to those achievable with a state-of-the-art load balancer for adaptive meshes. *Jostle* has been developed at the University of Greenwich and is a universal partitioning and load balancing tool for unstructured mesh computations. *Jostle* includes multi-level techniques for sequential partitioning as well as for parallel load balancing. The element migration decision uses Γ_{cut} as rating function. Together with a special border optimization based on KL and with sophisticated tie-breaking methods, *Jostle* is able to generate and maintain well shaped partitions, even if the Aspect Ratio is not directly considered.

Figure 7 shows the development of AR over typical runs of (solve/refine/balance...) for our four test examples. The left part shows the results if *Jostle* (without multi-level coarsening – the JOSTLE-D configuration described in [29]) is used as load balancer, the right gives the results if the AR-optimization of *PadFEM* is used (with best possible parameter combinations). It can be observed that the AR is decreasing with increasing numbers of elements if the *PadFEM*-balancer is used. The opposite is true if *Jostle* determines the elements to be moved. This shows that our strategy is able to operate as high-quality load balancer in adaptive environments maintaining well shaped subdo-

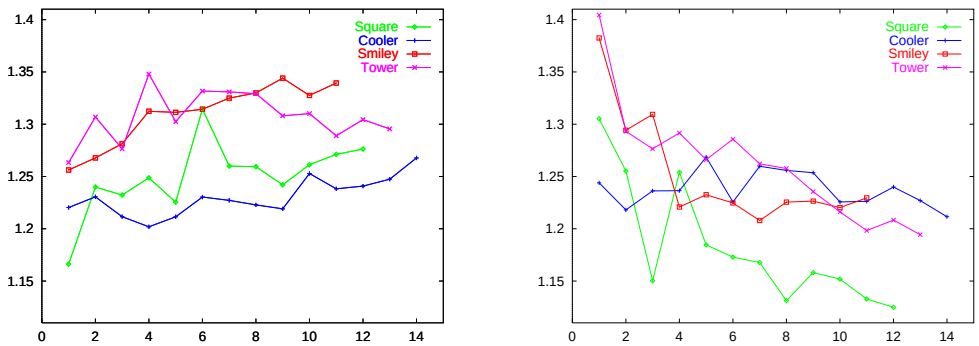


Figure 7: The development of the AR for increasing levels of refinement.

mains over a long period of time. Standard load balancers like *Jostle* (but with *Metis*, it would be just the same) are not directly able to be used over a longer period without complete repartitionings from time to time - at least if the partition Aspect Ratio is a measure of importance. Numerical results show that *PadFEM* is able to improve the Aspect Ratio of mesh partitions by around 10% over a state-of-the-art general purpose load balancing tool.

6 Conclusions

We have presented a two-step load balancing algorithm for adaptive mesh calculations on distributed memory machines. The balancing algorithm is distributed and takes existing partitions into account. The algorithm first determines a balancing flow and moves elements afterwards. The element migration is designed in particular to optimize subdomain aspect ratios.

Experiments using a preconditioned Conjugate Gradient solver based on Domain Decomposition and comparisons to existing balancing tools show the benefit of our approach.

References

[1] R.K. Ahuja, T.L. Magnanti, J.B. Orlin: *Network Flows - Theory, Algorithms, and Applications*. Prentice Hall, 1993.

[2] S. Blazy, W. Borchers, U. Dralle: *Parallelization methods for a characteristic's pressure correction scheme.* In: E.H. Hirschel (ed), *Flow Simulation with High-Performance Computers II*, Notes on Numerical Fluid Mechanics, 1995.

[3] S. Blazy, R. Diekmann, U. Dralle, F. Neugebauer, R. Preis, T. Römke: *Parallel Adaptive PCG.* In: B.H.V. Topping (ed): "Advances in Computational Mechanics with Parallel and Distributed Processing", Civil-Comp Press, Edinburgh, 1998 (to appear).

[4] J.E. Boillat: *Load Balancing and Poisson Equation in a Graph.* Concurrency - Practice and Experience 2(4), 289-313, 1990.

[5] J.H. Bramble, J.E. Pasciac, A.H. Schatz: *The construction of preconditioners for elliptic problems by substructuring I.+II.*, Math. Comp. 47+49, 1986+87.

[6] T.N. Bui, S. Chaudhuri, F.T. Leighton, M. Sisper, *Graph Bisection Algorithms with Good Average Case Behaviour.* Combinatorica 7 (2), 171-191, 1987.

[7] M. Burghardt, L. Laemmer, U. Meissner: *Parallel adaptive Mesh Generation.* In: B.H.V. Topping (Ed.), Proc. EuroConference on Parallel Computing in Computational Mechanics, Civil-Comp Press, Edinburgh, 45-52, 1997.

[8] N. Chrisochoides, C.E. Houstis, E.N. Houstis, S.K. Kortesis, J.R. Rice: *Automatic Load Balanced Partitioning Strategies for PDE Computations.* Proc. ACM Int. Conf. on Supercomputing, 99-107, 1989.

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

[10] R. Diekmann, A. Frommer, B. Monien: *Load Balancing on Graphs.* In preparation.

[11] R. Diekmann, D. Meyer, B. Monien: *Parallel Decomposition of Unstructured FEM-Meshes.* Concurrency - Practice and Experience, to appear.

[12] R. Diekmann, B. Monien, R. Preis: *Load Balancing Strategies for Distributed Memory Machines.* Techn. Rep. tr-rsfb-96-050, CS-Dept., Univ. of Paderborn, 1997.

[13] R. Diekmann, S. Muthukrishnan, M.V. Nayakkankuppam: *Engineering Diffusive Load Balancing Algorithms....* IRREGULAR '97, Springer LNCS 1253, 111-122, 1997.

[14] R. Diekmann, R. Preis, F. Schlimbach, C. Walshaw: *Aspect Ratio for Mesh Partitioning.* Report, University of Paderborn, to appear.

[15] B. Ghosh, S. Muthukrishnan, M.H. Schultz: *First and Second Order Diffusive Methods for Rapid, Coarse, Distributed Load Balancing.* Proc. 8th ACM SPAA, 72-81, 1996.

[16] C. Farhat, N. Maman, and G. Brown: *Mesh Partitioning for Implicit Computations via Iterative Domain Decomposition:....* Int. J. Numer. Methods Engrg., 38:989-1000, 1995.

[17] G. Horton: *A multi-level diffusion method for dynamic load balancing.* Parallel Computing 19 (1993), pp. 209-218

[18] Y.F. Hu, R.J. Blake: *An optimal dynamic load balancing algorithm.* Techn. Rep. DL-P-95-011, Daresbury Lab., UK, 1995. (to appear in: Concurrency - Practice and Experience)

[19] M.T. Jones, P.E. Plassmann: *Parallel Algorithms for the Adaptive Refinement and Partitioning of Unstructured Meshes.* Proc. IEEE HPCC, 478-485, 1994.

[20] M.T. Jones, P.E. Plassmann: *Parallel Algorithms for Adaptive Mesh Refinement.* SIAM J. on Scientific Computing, 18, 686-708, 1997.

[21] G. Karypis, V. Kumar: *A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs.* SIAM J. Scientific Computing, to appear.

[22] F. Neugebauer, R. Diekmann: *Improved Mesh Generation: Not Simple but Good.* Proc. 5th Int. Meshing Roundtable, Pittsburgh, PA, Oct. 1996.

[23] L. Oliker, R. Biswas: *Efficient Load Balancing and Data Remapping for Adaptive Grid Calculations.* Proc. 9th ACM SPAA, 33-42, 1997.

[24] *PadFEM Documentation.* `http://www.uni-paderborn.de/cs/PadFEM/`

[25] N. Touheed, P.K. Jimack: *Parallel Dynamic Load-Balancing for Adaptive Distributed Memory PDE Solvers.* 8th SIAM Conf. Par. Processing for Scientific Computing, 1997.

[26] D. Vanderstraeten, R. Keunings, C. Farhat: *Beyond Conventional Mesh Partitioning Algorithms and the Minimum Edge Cut Criterion: Impact on Realistic Applications.* 6th SIAM Conf. on Parallel Processing for Scientific Computing, 611-614, 1995.

[27] D. Vanderstraeten, C. Farhat, P.S. Chen, R. Keunings, O. Zone: *A Retrofit Based Methodology for the Fast Generation and Optimization of Large-Scale Mesh Partitions:....* Comput. Methods Appl. Mech. Engrg. 133, 25-45, 1996.

[28] R. Verfürth: *A Review of a posteriori Error Estimation and Adaptive Mesh-Refinement Techniques.* John Wiley & Sons, Chichester, 1996.

[29] C. Walshaw, M. Cross, M. Everett: *Dynamic load-balancing for parallel adaptive unstructured meshes.* 8th SIAM Conf. Par. Processing for Scientific Computing, 1997.

[30] O.C. Zienkiewicz: *The finite element method.* McGraw-Hill, 1989.

Load Balancing in High-Performance Distributed DB/DC-Systems: Modeling, Analysis and Clustering of Workload for Data-Affinity Based Load Distribution

Reinhard Riedl, University of Zurich

Abstract

To exploit data-affinity, or not to exploit data-affinity, that is the question. Whether it is preferable to deal with the complexity of data reference behaviour of real-world workload, or to ignore the lots of delays in response time, and by restricting oneself to simple solutions, delete them from appearances.

The above question indicates the answer, though. Despite of the complexity of real-life high-performance transaction processing in distributed DB/DC-systems, data-affinity may be exploited to reduce response times for some workloads. Starting from the question 'How does the problem look like?' we present a scientific approach to the subject, the main part of which is concerned with a technique to break up the problem of data-affinity based load distribution into a static and a dynamic problem. For that purpose we introduce representation schemes for workload and for distribution strategies, respectively, which yield transposition results for sender-initiated and receiver initiated algorithms, and for load clustering and data clustering, respectively. At the end, we present results obtained from the analysis of real database traces which empiricaly support the above assertion.

0 Introduction

By folklore, distributed systems refer to the distribution of services, workload, knowledge (or intelligence), and of control, which primarily results in an increase of the complexity and a grace-less degradation of the quality of service. Still, the idea in behind the use of distributed systems is the hope that a distribution of computing power not only increase reliability and availability, but also reduces response times. In this paper we shall primarily address the question, whether data-affinity based workload distribution makes sense in high-performance computing: Is it possible to reduce the data access times in such a way that overall response times are effectively

decreased and the users indeed observe better performance of the system? Our discussion will focus on distributed high-performance transaction processing DB/DB (database / data communication) systems with a front-end consisting of terminals and a back-end consisting of nodes for execution.

The response-time throughput graph is one of the major yardsticks for the quality of service of transaction processing systems. How can a distributed system be managed that a minimal response time is achieved for any given throughput (where throughput is equal to the workload feed)? The trivial answer to this question reads "Distribute the workload properly!", that is "Perform load balancing!" However, what does load balancing mean? And, first of all, who is to be put in control of that procedure? And where is the knowledge maintained which is needed for that procedure?

Obviously, there are no easy answers to these questions. We have tried to achieve a better understanding of these questions by adopting local and even only partial local views. This paper reports on the results of our research.

The generic local view consists of a central control, distributed system states and distributed system knowledge. We do not assume a priori knowledge, but foster some believe in the observability of semi-static workload structures. The resulting problem of minimizing response times for a given workload feed still remains a most complex one. We shall demonstrate that one may capture some of its features by filtering out partial views on that problem, as there is some evidence that a view at the real problem may be composed by superpositions of partial views. The latter assertion which is supported by simulation results, as we shall indicate at the end.

The local views correspond to the major contributions to the response time for a transaction, each separated from the others. Of course, one is primarily interested in those contributions to response time, which may indeed be influenced by load balancing activities. Including those components which may be quantitatively changed by way of side-effects, this set of critical views comprises waiting times (for CPU processing), data access times, context transfer times, CPU processing times, waiting times due to lock contention, overhead due to information collection, overhead due to decision making, and tailback effects. The definition of an additive cost function of the enumerated cost factors is the most simple way of superposing the corresponding partial views, and we shall therefore use it (minus some terms) as a generic cost model.

The introduction of a cost function poses the questions for what kind of distribution activities costs ought to be defined and at which occasions they ought to be evaluated and the distribution ought to be performed, respectively. Evaluation makes sense any time the available information about the system state and the workload has changed, and an evaluation activity will usually go immediately ahead of a distribution activity. This trivial observation still leaves open the question whether to employ time-triggered or event-triggered activities. For the sake of clarity we shall limit our discussion to the problem of where to place a load item in case a distribution activity is desirable. Event-triggered systems are a natural choice for the study

of this decision problem, where the triggering events are arrivals of so far unknown tasks. We shall thus discuss adaptive sender-initiated load distribution, which takes place immediately upon arrival. This means little restriction for two reasons. On the one hand, there is a strong duality between decisions for sender-initiated load distribution and decisions for receiver-initiated load distribution (based on affinity matrices). On the other hand, the decision problem is not altered by the triggering event. In the general case there is only the additional problem of choosing a task for migration from the chosen waiting queue. (That may be done for a sender-initiated distribution by comparing evaluations of the cost functions affiliated with different tasks.)

Our simulations have shown that one of the main problems in the design of a load distribution algorithm is the choice of the appropriate granularity of workload to be distributed. Various trade-offs have to be considered for that purpose. Load distribution may be performed on the level of atomic transactions or even on the level of dialogue steps, but the general observation that any migration is expensive suggests the choice of a coarser granularity for distribution activities such as conversations consisting of possibly various transactions related by a common context, namely the user, or even the choice of terminal sessions, consisting of possibly various conversations. In any case, the problem to be solved is the choice of a node for execution of the load granule to be distributed. This task may be performed based upon information about the current load state of the system, records of past workload feed, past distribution decisions and commits at the various nodes, and possibly some trace records.

An analysis of real-life scenarios suggests a two level model with the distribution of terminal sessions and the distribution of transactions. In this case the corresponding allocation decisions are the variables of the generic cost function above, which are mapped on resulting response times. For the sake of simplicity we shall restrict our considerations in this paper to a one level model with the distribution of transactions only. Since the distribution on both levels of granularity has to be performed separately, the adaption of the obtained results to the two level model is straightforward.

One can identify an algorithm for each summand, which heuristically minimizes it. There is no evidence, though, that some combination - e.g. a superposition based on affinity matrices to be defined below - will yield an algorithm minimizing overall response time. Some of the identified algorithms might - and indeed do - produce side effects, which increase other summands. However, a switching strategy, which tries to identify the dominating factor and then applies the affiliated distribution algorithm will yield satisfactory performance for a wide area of load scenarios. In [5] more sophisticated switching strategies inspired by fuzzyfied strategies from classical control theory have been discussed. The results obtained in our simulations (performed in the LYDIA project) using non-fuzzy combinations of single theme algorithms in the spirit of [3] equally back the assumption that 'good' heuristic load distribution algorithms may be composed starting from optimal single theme

algorithms for each of the cost factors.

We shall proceed now as follows. First, we shall sketch an architecture model and we shall give figures of a real-life scenario. Second, we shall describe a set of models describing the problems we intended to solve. Third, we shall define the generic cost function in a semi-formal way. Fourth, we shall identify heuristically optimal single theme algorithms each minimizing one of the cost factors. Fifth, we shall introduce affinity matrices and we shall indicate how they can be used to represent load distribution algorithms. Sixth, we shall discuss the requirements and constraints for algorithms, which try to minimize data access times and we shall indicate how the corresponding decision problem may be split into a static and a dynamic allocation problem. Seventh, we shall elaborate on the static problem, namely the clustering problem for database traces. Eighth, we shall present empirical results for the static allocation problem, which we have obtained from the analysis of real database traces. And ninth, we shall present some results obtained in simulations.

This paper presents the author's view at the research performed in the Esprit project LYDIA. The cited empirical results as well as part of conceptual techniques for analysis have been obtained in collaboration with Eike Born (Siemens Nixdorf Munich), Thomas Delica (Siemens Nixdorf Munich), and Lutz Richter (University of Zurich). The author would like thank Erol Gelenbe (Paris V) for some intriguing questions he has posed during the project (one of which will be answered below), Christos Nikolaou, who has managed the project, and all the other partners.

As for related research, you may find some of it in the publications of the LYDIA project [7], where the kind reader will also find lots of references to related research projects. We also refer to [6] for related approaches and to [4] for classification of adaptive load balancing algorithms.

1 The case scenario

In this section we shall describe our case scenario. We consider a data processing system which consists of a considerably large number- between, say, 4 and 64 - of processing units addressed as nodes. Each node is equipped with its own main memory, (possibly shared) hard disk environment, address space, and operating system. The nodes communicate with each other via a high speed network, supplying with node-to-node links. Terminals are connected via a frontend systems to all nodes of the distributed system. And a data communication system supports with basic communication functions to remote processing nodes, in particular terminal sessions as well defined connections of a terminal to an application at a node.

Users invoke operations on a database, which is physically stored on several disks, and which is accessed via a database handler offering read and write operations. The database address space is partitioned into database pages, which are identified by page numbers. To reduce access times, database pages are cached in node-specific main memory page buffers after having been retrieved from the disk. In general,

replication of database pages is allowed.

Users interact with the database system by initiating database transactions, which fulfill the so-called ACID requirements of atomicity (with respect to crashes), consistency (with respect to transactions), isolation (i.e serializabilty), and durability (of successfully completed transactions). The system component that provides the user with basic functionality to implement transactions is the transaction-monitor, which performs concurrency control and recovery tasks. We assume that the first is achieved by a pessimistic two-phase locking protocol: In an acquisition phase, a lock is acquired for each data item to be accessed and these locks are released only in a release phase which must not be followed by another acquisition phase.

The workload the system has to deal with consists of a transaction feed, initiated by users, who are identified by their user identity (UID). Each transaction may consists of one or more dialogue steps, where a dialogue step is the processing between one terminal entry and the next terminal output. Individual transactions are identified by a transaction number.

A conversation - identified by conversation identity (CID) - is a consecutive execution of several transaction programs with additional support and data representation from the transaction-monitor. Data corresponding to a conversation are forwarded from transaction to transaction via a communication buffer and a conversation specific memory. Various consecutive conversations constitute a terminal session, which is started and finished on request of a user and which is identified by its terminal identity (TID). In an application different transaction programs - identified by a transaction program code (TAC) - are bound together. During a terminal session each user is connected to a certain transaction monitor application session identified by a application identity (AID).

CID, TAC, UID, TID and AID are known before a transactions starts execution, while data accesses and CPU consumption are unknown at that time. Typically, 10^5 or even more transactions will be executed in one hour, performing some 10^7 data accesses (to some $10^{5.5}$ database pages).

In the following discussion we shall somewhat modify this above scenario for reasons indicated in the introduction. We shall consider only a part of the above system with all terminals connected to one node which distributes load granules upon their arrival to set of n fixed execution nodes. No redistribution is admitted and transfer costs for load units essentially depend on the context data, which are necessary for processing the load unit. The smaller the granule is, the larger is the amount of context data to be transferred. In case of transactions these are the conversation specific long time memory, the conversation specific communication pool, the terminal specific long time memory, the communication management data and name plus address of the transaction monitor application. Therefore, load granules considered for distribution will be transactions. Choosing a finer granulation would increase costs of implementation dramatically. Choosing a coarser granularity would still demand the support for a partial rollback of these units by, say, single transactions.

2 The load model, the action model, and the solution model

We model the load as a sequence of transactions (considering only committing ones). Each transaction is represented by the time it enters the system, the time it leaves the system, and a set of genes. Genes are partitioned into two subsets, those which are observable a priori, that is before execution starts, and those which are observable only a posteriori, that is during or after execution.

The a priori observables are (by abuse of terminology) the CID, TAC, UID, TID and the AID. The a posteriori observables are the accesses to database pages. As repeated data accesses do not essentially increase the accumulated access time for a particular page, we ignore information about the number of accesses performed by one single transaction to a particular database page. Thus load can be formally represented by a primary workload matrix, whose lines correspond to transactions, whose first five columns correspond to the a priori observables, and each of whose further columns corresponds to one particular database page. In case of evaluation of database traces the enumeration of the latter will usually contain little if any useful semantic information. In case that database tracing is performed past performance may be represented by the type of matrices defined in the above paragraph.

As distribution algorithms cannot distinguish between transactions with identical a priori observables we partition the workload into classes in such a way that the members of a class have got identical a priori observables, and no two members of different classes have got identical a priori observables. This guarantees that any incoming new transactions may be immediately assigned to a class (of the cross product of ... a priori observables). Based on this 'finest possible' partition we define a reference matrix $A(t) = (a_{i,j})$, whose rows correspond to the classes and whose columns correspond to the database pages: An entry $a_{i,j}(t)$ counts the number of transactions of class i which have referenced page j, where only those transactions are considered which have committed before time t. This matrix represents the statistic knowledge of data access patterns which can be exploited by the load distribution algorithm for the reduction of data access times. (Moreover, it might also be used to some extent for the application of shortest job first algorithms, but we shall not discuss related issues in this paper.)

The only actions we consider for load balancing are the migrations of transactions from the node where they enter the system to the node where they are executed. Thus the problem which has to be solved then by the load balancing algorithm is the choice of the optimal node for the execution of a transaction belonging to some transaction class. All tasks are considered equal and tailback effects as well as overhead are neglected. Moreover, costs due to lock contention are neglected as well, because the control of conflict ratio is a single node problem, which has been extensively studied in literature.

3 The generic cost function

For a transaction belonging to some class i the cost function R_i affiliated with the above solution model reads as follows:

$$R_i(.) = P(.) + A_i(., R(t), D(t)) + W(., t) + C_i(., D(t)),$$

where $R(t)$ denotes the reference matrix and $D(t)$ denotes the record of all distribution activities up to time t. Hereby, we assume that the processing time solely depends on the processing capacity at a node. The enumeration of the variables $R(t)$ and $D(t)$ for data access times indicates that some sensible guess may be made on the data access times based on the knowledge of data access patterns and past distribution decisions. The enumeration of the variable t for the waiting times indicates that waiting times will depend on the number of transactions currently residing at a node. And the enumeration of the variable $D(t)$ for the costs of context transfer indicates that these costs depend on the past routing decisions taken for transactions belonging to the same conversation.

We might reformulate this cost function by considering transaction classes as variables and by introducing system parameters, namely the reference matrix, the distribution record, the contents of the waiting queues, and possibly the recently observed CPU processing capacities. In that case the cost function relates (decision) variables - namely distribution activities and transaction classes - system parameters, and response times. In practice its calculation will be constrained by the restricted availability of system parameters.

4 Single theme algorithms

Each of the four summands may be heuristically minimized by one of four algorithm, three of which are more or less easy to specify. Processing times are minimized by choosing the fastest node. Waiting times are minimized by choosing the shortest waiting queues. And costs of context transfer are minimized by executing all transactions belonging to the same conversation at the same node. Finally, data access times are 'somewhat' minimized by choosing that node, where most data needed may be accessed locally. (Here we do not differentiate between read and write accesses for reasons of heuristic simplicity. In particular, we ignore the fact that write accesses are most expensive in case that data reside in remote caches).

There remain one or two unknowns, though. Which database pages will be accessed by a given transaction? And where do these pages reside currently? The latter is known in case of function shipping, when accesses to remote pages lead to the invocation a subtransaction. But in case of data shipping, when pages to be accessed are fetched from remote nodes, knowledge about the place where each page currently resides cannot be used for load distribution (due to the large amount of database pages).

5 Affinity matrices

Affinty matrices describe the affinty between a task class and a node for execution. The rows of an affinity matrix correspond to (nonempty) task classes and the columns correspond to nodes. The higher an entry in a column is the more preferable is it to execute a task from the task class there. One may define an affinity matrix for each partial view, that is for each cost factor. Then the entries in the matrix relate to amount contributed by the factor to the overall execution time. For example, the entry may be chosen to be the inverse of this amount. In that particular case there is a bijective mapping between the evaluations of the cost factors and the entries in the affinity matrix and the decision problem thus can be rewritten in terms of affinity matrices.

There is a duality between sender-initiated and receiver-initiated distribution problems: Both may be described in terms of affinity matrices and a transposition of the matrix switches from sender-initiation to receiver-initiation and backwards. Moreover, single theme algorithms may be combined based on simple algebraic operations on affinity matrices.

6 Data affinity

The main constraint on the exploitation of data affinity is the enormous complexity of the problem. Fortunately, one does not have to keep track of hundreds of thousands of database pages or tens of millions of data accesses and one does not have to know where the pages reside, or which page will exactly be accessed. One only ought to be able to guess the likelihood that remote data will have to be accessed.

A natural choice for a distribution decision which intends to minimize remote data accesses is that node, where the last transaction with identical a priori observables has been routed to. However, only very few transactions will have identical a priori observables. Therefore, little statistical meaning can be attributed to the presentation of workload given in the reference matrix. Moreover, the influence on the data distribution of the precedessors of a transaction as members of its workload class will be rather insignificant due to the smallness of the classes. Nevertheless, it is only the a priori observables which may be used for an a priori guess on the data accesses.

There is one way out of that dilemma, namely the definition of some distance function on the classes of the finest partition, which attributes small values to pairs of classes which are likely to exhibit similar data accesses and large values to those which are likely to exhibit a rather disjoint access behaviour. Based on such a distance function the affinity between a transaction class in the 'finest workload partition' and a node may be calculated as a properly weighted average of the inverse distance values between the class under consideration and the classes of those transactions which have been executed at the node in the past. As on-line tracing and on-the-fly evaluation of traces creates too much overhead this distance function

ought to be based on off-line database traces. In case of workload which changes its data access patterns in a highly dynamic manner this makes little sense and consequently, there is little chance to exploit data affinity then. Another constraint on the use of such distance functions is the necessity to obtain differing affinities. This implies in particular that for most classes one must be able to separate two large sets of classes from the finest partition by their distances.

In any case the use of distance functions breaks up the problem of data-affinity based load distribution into a static and a dynamic part. The static part consists of the definition of the proper distances and the dynamic part consists of the weighting and averaging procedures indicated above.

The most simple distance functions for our purposes are those which take only the values 0 and 1. They correspond to a crisp workload partitioning problem or rather to a clustering problem. Distance functions taking continuous values correspond to the equivalent fuzzy partitioning problem, or clustering problem, respectively.

Let us restate the (crisp) clustering problem: Find a partition of all transactions into classes based on UID, TID, TAC, CID, and AID, such that accesses of transactions belonging to disjoint classes access nearly disjoint pages, while transactions belonging to the same class access nearly the same pages. Workload partitioned that way may again be represented by a reference matrix which now depends on the partitioning and whose rows correspond to the classes of the partitioning.

In order that this clustering may be exploited for data-affinity based load distribution, the set of classes ought to be small. Empirical analysis has shown, however, that for efficient use the requirements on the clustering have to be relaxed. It is likely that in real-life workload one will encounter a few quite homogeneous classes and a sea of sporadic small island classes. Those may be put in a separate non-affinity class. Members of this class should then not be considered for data-affinity reasoning, and their distribution ought to rely on the other cost factors only.

Note that workload partitioning does not necessarily correspond to 0, 1-distances. Based on a given partitioning we may define a distance function, which attributes the value 0 to classes (according to the finest partitioning) of the same class (according to the partitioning obtained by clustering), and we may attributed constant distance values to pairs made up by transaction classes (according to the finest partitioning) from the same two classes (according to the partitioning obtained by clustering).

7 The clustering problem

The clustering problem identified above is a twin of the race problem: do there exist human races and in the case they do, can they be identified by a particular gene? Well, this is of course a modern scientific formulation which neglects the aspect of descent.

About one decade ago, a scientific answer has been given to the question for races identified by colour. They do not exist. Variance in genes of people with the

same colour are larger than distances in genes between people of different colours. A similar answer to our clustering problem would render data-affinity based load distribution an impossible task.

In the case that clustering makes some sense there still remains the question of how to judge on the quality of a clustering. The test case is the static workload distribution based on an optimal fixed allocation of classes to nodes. Heuristic estimates for this test case may be obtained from the evaluation reference matrices (with respect to the given clustering). Each of these estimates also defines an optimization problem, whose solution, difficult to achieve, though, would be a heuristically optimal clustering. One such heuristic is the distance of the reference matrix to the block diagonal matrices (modulo reordering of rows and columns) measured in terms of off-block-diagonal entries. Another one is the evaluation of the independent reference model (based on the independence of class-to-page reference events) corresponding to the case of function shipping with one copy of each page only. Empirical evaluations have shown the second to be a more controversial evaluation measure than the first.

Let us note that the representation technique of using reference matrices allows to transfer data clustering techniques into transaction clusterings techniques and vice versa by simply transposing the problems represented by the reference matrix similarly as in the case of sender-initiated and receiver-initiated algorithms.

8 Empirical workload analysis

In the LYDIA project we have analyzed two real workload traces supplied by Siemens Nixdorf. One consists of about 10000 transactions with about 1000000 page accesses to about 40000 pages, and the other consisting of about 1200 transactions with about 800000 page accesses to 70000 pages. The analysis was based on primary workload matrices.

At first sight, the results of our analysis seem to be controversial: On the one hand 45%, respectively 70% of all pages were only accessed by one transaction, while only 20%, respectively 5% were accessed by more than 5 different transactions. On the other hand, neglecting multiple accesses by one transaction, only 9%, respectively 32% of all page accesses concerned those pages, that were accessed by only one transaction, while 72%, respectively 37%, of all accesses concerned those pages which were accessed by at least half a dozen different transactions.

We concluded that even though many pages were 'private' pages there was a considerable potential for affinity-based scheduling. The latter conclusion was backed by the observation that for the larger trace the size of the footprint, i.e. the number of accessed pages, grew sublinearly as a function of the number of committed transactions. The curve was well approximated by $c \times t^{2/3}$, where c was nearly equal to the average number of accessed pages per transaction. (Quite honestly, the smaller trace did not exhibit such a behaviour, though.)

Erol Gelenbe has asked the author whether some form of Zipf's law could be observed. Surprisingly enough, for the two analyzed traces, the answer is indeed yes, although the exponent is different than the exponent observed for natural languages. (We only considered the rarely used words = the rarely accessed pages.) The number of pages accessed by i different transactions times $i^{2.5}$ appears to be nearly constant for small i. Oh fortuna, velut luna, statu variabilis ... . We are not impertinent enough, though, to derive conclusions on the entropy of the traces as compared withe entropy of natural language, which offer themselves.

A further analysis revealed that partitioning based on conversation identities is indeed a good heuristic candidate to start the clustering procedure: About half of the potential which may be expected to be achieved by optimal clustering is already due to such a preclustering. (This does not imply that such an 'optimal' clustering will respect class partitioning imposed by conversation identities.) In [1] it has been demonstrated that the application of standard hierarchical clustering procedures - such as the Ward algorithm - may further improve the quality of clusterings slightly.

One may conclude that a look at the growth rate of footprints and at the distribution of accesses to .-times accessed pages is indeed a first good test, whether there is some potential for clustering, or equivalently for data-affinity based load distribution, respectively. Moreover, standard partitionings according to one of the a priori observables should be analyzed first, before more sophisticated clustering algorithms are applied.

The main difficulty in clustering database traces stems from the high dimension of the problem space. In the two analyzed traces, the dimensions were 100000, and 40000 respectively. So far, we could not solve the clustering problem for such high-dimensional spaces in a satisfactory way (as nobody else seems to have done).

We would like to note that the observed footprint growth rates render any simulation with a fixed number of database pages quite a deceitful experiment. However, a discussion of the affiliated problem of synthetic traces generated from real traces is beyond the scope of this paper and it will be published elsewhere.

More detailed empirical results may be found in [2] .

9 Simulation results

We have carried out simulations with database traces, which have confirmed the results of the empirical workload analysis. Data-affinity based routing relying on standard partitions reduces remote data accesses by a factor 2-3, which results in a significant decrease of response time. Moreover, if the distribution is based on the partitioning according to conversation identities, response times are further reduced due to reduced context transfer times.

While JSQ (=join the shortest waiting queue) outperforms static affinity-based routing, the combination of JSQ and data-affinity based strategies seem to be the best solution for the load balancing problem, if conversations are not split.

10 Concluding remark

We have presented a scientific approach to the load balancing problem, which focuses on data-affinity. The empirical analysis of real traces has demonstrated that this approach is indeed practically applicable to real-world scenarios. The answer to the question whether or not data-affinity based load balancing will pay off, depends on both the workload and the actual system architecture, but we have gained evidence that there do exist real-world scenarios for which data-affinity based routing would make sense. Due to their generic nature, some results and techniques for analysis also apply to more general distributed systems. Still, a lot of research needs to be done. Don't believe those who try to convince you that the load balancing problem is solved! Nix ois Schwindler! Ikonoklasten! Theaterkritiker!!

References

[1] Lastclusterung für verteilte Hochleistungsdatenbanksysteme, N. Auerbach, Diplomarbeit, University of Zurich, 1997;

[2] Characterization of Workloads for Distributed DB/DC-Processing, E. Born, T. Delica, L. Richter, and R. Riedl, Deliv. 2.1 Esprit III P-8144, 1996;

[3] An affinity-based dynamic load balancing protocol for distributed transaction processing systems, S. Haldar and D. Subrammanian, Performance Evaluation 17, 1993;

[4] Classification of Load Distribution Algorithms, R. Riedl and L. Richter, Proc. 4th Euromicro Workshop on Parallel and Distributed Processing, 1996;

[5] Fuzzy Control of Load-Balancing for Distributed DB/DC Transaction Processing, R. Riedl and R. Salomon, Proc. 2nd ICSC Symp. Fuzzy Logic and Applications, 1997;

[6] Performance Analysis of Affinity Clustering on Transaction Processing Coupling Architectures, Yu P.S., Cornell C., Dias M., Balakrishna I., IEEE Trans. Knowledge and Data Engineering, Vol 6, No 5, 1994;

[7] http://www.ics.forth.gr/pleiades/projects/LYDIA/

Towards a Scalable Parallel Algorithm for Polynomial Real Root Isolation *

Extended Abstract

Thomas Decker and Werner Krandick
Department of Mathematics and Computer Science
University of Paderborn, Germany
e-mail: {decker, krandick}@uni-paderborn.de
URL: http://www.uni-paderborn.de/cs/decker.html
URL: http://www-math.uni-paderborn.de/~krandick

1 Introduction

Many problems in mathematics, science, and engineering require the computation of polynomial roots. If the degree of the polynomials is 2, 3, or 4, one can use formulae for the roots; if the degree is higher, search methods must be used. Numerical search methods are fast, but, in many cases, they return wrong results. Validated numerical methods will never return a wrong result, but they might give up on certain inputs. As opposed to this, computer algebra methods return correct results in every case; however, they require more time and memory than numerical methods. What makes them interesting for parallel computation is their two-step strategy. First, they compute *isolating intervals* for the roots. These are disjoint intervals that each contain exactly one root of the polynomial and together contain all the roots. Once the roots have been isolated, the isolating intervals can be refined independently of each other. We investigate how the first phase, root isolation, can be parallelized. We restrict ourselves to the case of polynomial real roots—noting that real root isolation can be used as a subalgorithm in the isolation of non-real complex roots.

Our implementation uses the *Daisy*-library (Data Management Library for Parallel Systems) [6] which consists of two cooperating parts, the load balancer *VDS* (Virtual Data Space) [5], and the memory manager *Diva* (Distributed Variables) for the provision of global variables. In contrast to load-balancing systems based on process-migration, *VDS* describes tasks by data-items

*This work was supported by the "DFG Sonderforschungsbereich 376 : Massive Parallelität - Algorithmen, Entwurfsmethoden, Anwendungen", by the EU ESPRIT Long Term Research Project 20244 (ALCOM-IT), and by grant ERB-FMGE-CT95-0051 of the TMR Programme of the European Union.

that specify sub-problems which can be processed by any of the processors in the system. For this reason we refer to *VDS*-items as *orders*. So, both parts of *Daisy* deal with data items. *VDS* distributes its (non-addressable) items so as to achieve a balanced distribution whereas *Diva* assigns and duplicates its (addressable) items in order to exploit the locality of the application [10]. We found that combining global variables and load balancing considerably facilitates programming a parallel system.

Parallel real root isolation was attempted before by Collins, Johnson, and Küchlin [4]. These authors encountered a scaling problem that we solve with new algorithmic ideas. For efficiency, we use validated computations with software-supplied multiprecision floating point numbers. Our prototypical implementation already shows some of the benefits that we expect from a fully developed method.

Section 2 explains some aspects of the Descartes method for isolating polynomial real roots. Section 3 describes the *Daisy*-system and the parallel program. Section 4, finally, summarizes results obtained thus far.

2 Isolating polynomial real roots

Real root isolation is a binary tree search. At the root node there is an interval I that is known *a priori* to contain all the roots of the input polynomial A. Isolating intervals are found by iterated bisection of I. For each subinterval $J \subset I$ that is obtained the algorithm asks whether it contains no root or exactly one root of A. If it learns that J does not contain any root of A, it discards J; if it learns that J contains exactly one root of A, it outputs J as an isolating interval. In all other cases the algorithm continues and bisects J.

A parallelization was first attempted by Collins, Johnson, and Küchlin [4]. For every node in the search tree the computations of its left and right subtrees were executed in parallel. However, it was found that for random input polynomials the approach does not lead to an effective speed-up, not even by a factor of 2—no matter how many processors were used. This observation was explained by showing that the search trees tend to be narrow. In fact, the average width of search trees for random input polynomials of degree 100 turned out to be just 1.54.

Since not much parallelism is available at the tree level, we pursue a more fine-grained approach to parallelization. To explain our ideas we need to describe the algorithm in some more detail.

Various criteria can be used to test whether an interval contains no root or exactly one root of A. We chose a root isolation method that uses Descartes' rule of signs. A detailed description of the method was given by Krandick [9]. Descartes' rule links the number of *sign variations* in the coefficient sequence of a polynomial P, to the number of positive real roots of P. Sign variations are sign changes from plus to minus and from minus to plus. The polynomial $Q = x^5 - 12x^3 - 3x^2 + 1$ has $\mathrm{var}(Q) = 2$ sign variations. Our method applies the theorem to a transformation P^* of P to get a statement about the number of roots of P in J. In particular,

- if $\mathrm{var}(P^*) = 0$, then P has *no* root in J,
- if $\mathrm{var}(P^*) = 1$, then P has *exactly one* root in J;
- if $\mathrm{var}(P^*) > 1$, then P might have *any number* of roots in J.

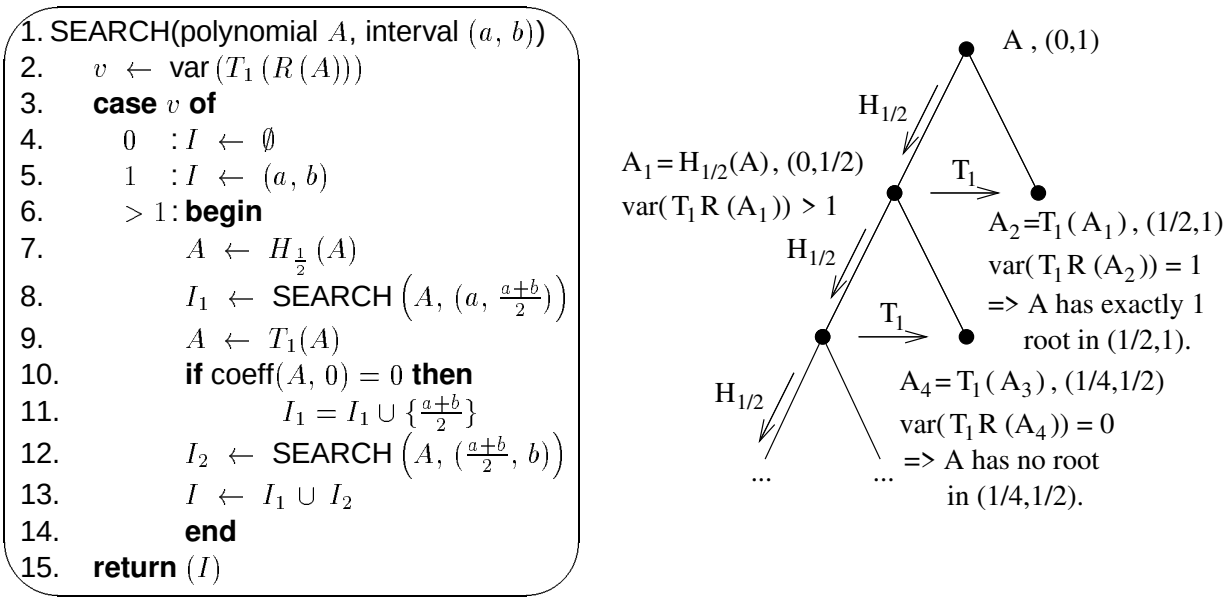


Figure 1: Root isolation is a binary search.

So, our method mainly computes transformations of polynomials.

Figure 1 shows that three transformations are used. The most expensive of these is the transformation T_1 that computes the coefficients of the polynomial $B(x) = A(x+1)$ from the coefficients of $A(x)$. The cost for this *translation by* 1 dominates the cost of the other operations and already for small degrees of A takes up more than 90 per cent of the total computing time.

Indeed, if $A(x)$ is a polynomial of degree n with integer coefficients of maximal absolute value $|A|_\infty = d$ then the time to compute $T_1(A)$ is dominated by $n^3 + n^2 L(d)$ while the cost for the other transformations is dominated by $n\,L(d)$.

So, we parallelize root isolation by parallelizing translation by 1. Since this operation needs to be performed for each node of the tree we expect that our algorithm will scale well even if the tree is narrow.

For sequential computation of T_1 we use a method similar to Horner's scheme. A computation of all the coefficients by Horner's scheme requires a computing time that is dominated by $n^3 + n^2 L$, where n denotes the degree of A, and L denotes the length of the longest coefficient of A.

For parallel computation, we observe that translation by 1 can be described as multiplication

by Pascal's triangle,

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & \dots \\ & 1 & 2 & 3 & 4 & \dots \\ & & 1 & 3 & 6 & \dots \\ & & & 1 & 4 & \dots \\ & & & & 1 & \dots \\ & & & & & \vdots \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ \vdots \end{pmatrix} = \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ \vdots \end{pmatrix},$$

where $a_0, \dots, a_n$ are the coefficients of $A(x) = \sum_{k=0}^{n} a_k x^k$, and $b_0, \dots, b_n$ are the coefficients of $B(x) = A(x+1) = \sum_{k=0}^{n} b_k x^k$. Clearly, the coefficients b_k can be computed independently of each other. The time required for each such computation is dominated by $n^2 L$.

So, computing scalar products independently on n processors will lead to a speed-up that is proportional to the degree of the polynomial. However, it is more realistic to assume that the number p of processors is fixed, and that—at best—we can distribute the total amount of work evenly. Indeed, computing n scalar products with p processors requires time $n^3 L/p$ which, for $p \ll n$, does not compare well with the sequential use of Horner's scheme.

The factor L is a serious obstacle to performance since it grows as the method descends in the tree. Indeed, one can show that from one level of the tree to the next the coefficient length increases by up to $n+1$ bits.

To prevent the coefficient size from growing we use floating point arithmetic. We control round-off error by using interval arithmetic [1], an idea that has already been successfully applied to a sequential version of the Descartes' method [8]. What is new here is the use of software-supplied multiprecision floating point arithmetic. By experiment, we investigate what precision should be used for a given input.

Since the coefficients cannot grow beyond a fixed floating point precision l, the time required for translation by 1 with Horner's scheme is now dominated by $n^2 l$ since we have to perform $n(n+1)/2$ additions of numbers of length l. Computing a scalar product in floating point requires up to n products of numbers of length l, and hence the time to compute n scalar products is dominated by $n^2 l^2$.

One can show that the width of the tree is at most n. So, if h is the height of the tree, the total number of translations by 1 is dominated by hn. Therefore, the total computing time of a sequential root isolation that uses Horner's scheme and floating point arithmetic is dominated by $n^3 hl$ while the total computing time of parallel root isolation that uses scalar products and floating point arithmetic is dominated by $n^3 hl^2/p$. Since the required precision l is usually quite low, this approach might indeed lead to a good parallel method.

First experiments seem to confirm this impression. There are cases where our parallel method beats sequential execution, and we are confident that we can further improve the implementation.

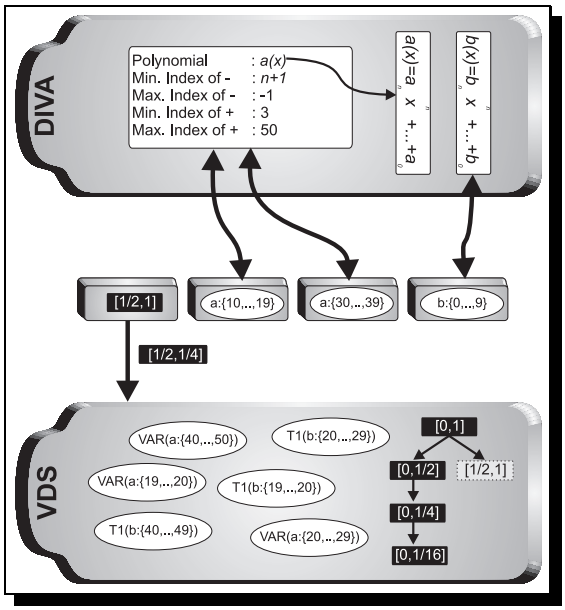


Figure 2: The two *DAISY* tools: Distributed shared memory (*Diva*) and the load-balancing layer *VDS*.

3 Parallel root isolation

3.1 The *Daisy*-library

The data management library *Daisy* (Fig. 2) provides tools for the simulation of shared memory (*Diva*) and for load balancing (*VDS*) in a single, comprehensive library. Currently, *Daisy* is available for Parsytec's PARIX and PowerMPI platforms as well as for PVM.

Load Balancing with *VDS*. The Virtual Data Space tool *VDS* simulates a global data space for data-items stored in distributed heaps, stacks, and other abstract data types. The work packets are spread over the distributed processors as evenly as possible with respect to the incurred balancing overhead. Although the items are not addressable, it is often necessary to distinguish between items that are processed in a completely different manner. Therefore, each item belongs to exactly one *class* defined by the application. For each class, the application specifies the *class-type* that influences the way, *VDS* treats the corresponding objects. In the current version *VDS* offers the following class-types.

- *Normal* items are weighted by a load measure reflecting the processing cost of the item. Consequently, the load of a processor with respect to a specific class is given as the sum of the load-weights of all the placed items of this class. *VDS* attempts to distribute the items in such a way that all the processors have about the same load (quantitative load balancing). All items which are located on the same processor are processed in a FIFO-order.

- For example, in best-first branch&bound it is necessary to influence the order in which the items are processed by mean of priorities. For this purpose *VDS* provides *weighted*

items, which are balanced with respect of both, their processing cost, and their priority (quantitative and qualitative load balancing).

- The third kind of item, *thread items*, provides an easy and intuitive way to model multi-threaded computations as it is done in Cilk [2]. In this computational model, threads may send messages (results) to their parents. The standard distribution method for classes of this type is workstealing. The order in which the items are processed is basically in LIFO-manner.

For the distribution of the items, *VDS* offers a variety of load balancing algorithms like work-stealing, diffusion, random placement, and round-robin placement, which means that the items are distributed according to a fixed order of the processors (spreading). Moreover, *VDS* provides methods for termination detection, and performance evaluation.

Shared Memory Simulation with *Diva*. The distributed variables library *Diva* provides functions for simulating shared memory on distributed systems. The code idea is to provide an access mechanism to distributed variables rather than to memory pages or single memory cells. The variables can be created and released at runtime. Once a global variable is created, each processor in the system has access to it.

For latency hiding, reads and writes can be performed in two separate function calls. The first call initiates the variable access, and the second one actually reads its value as soon as it is locally available. The time between initiation and completion of a variable access can be hidden by other local instructions or variable accesses. In order to ensure a quick data access, *Diva* incorporates a tree-based caching-mechanism which has been proven to be 3-competitive with respect to optimal strategies [10].

Furthermore, *Diva* provides useful functions for synchronization, locking of global variables, group management, global operations, and garbage collection.

***VDS* and *Diva*.** The two parts complement each other as can be seen from the following two aspects of our application.

For example, one of the orders may be to compute a number of coefficients of a transformed polynomial. A possible approach would be to store the polynomials inside these orders. Unfortunately, the coefficients of the polynomials and hence the polynomials themselves tend to become quite large, up to about 30 KByte. If the polynomials were stored inside the orders it would be expensive to migrate orders, and consequently the load balancer would perform badly. Instead, we store the polynomials as global *Diva*-variables; inside the orders we just store pointers to these objects.

In the following aspect of our application *VDS* and *Diva* form even more vital complements. Not all transformations T_1R need to be executed completely. Some transformations can be aborted as soon as it becomes clear that $T_1R(P)$ will have at least two coefficient sign variations. To make this abortion possible the processors engaged in the computation of the various coefficients of $T_1R(P)$ from time to time exchange information about the coefficient signs they found so far. This exchange cannot be done by message–passing since the processors are not

Class	Balancing	Usage
T_1_order	Spreading	requests the computation of a part of $T_1(A)$.
T_1_result_msg	no LB	contains the computed coefficients of $T_1(A)$.
VAR_order	Spreading	requests the computation of a part of var($T_1(R((A)))$).
VAR_result_msg	no LB	contains the computed sign variation.
SEARCH_order	Workstealing	invokes the SEARCH-algorithm (Fig. 1).

Table 1: The different classes used for the parallelization of the SEARCH-algorithm.

aware of each other. Using *Diva* we were able to solve the cooperation problem by using a single global variable that controls the exchange.

3.2 The parallel algorithm

The basic algorithm for the isolation of positive real roots is shown in Fig.1. In addition to executing each translation by 1 in parallel we also execute the SEARCH-calls in lines 8 and 10 in parallel.

The basic parallelization technique is the same in all three operations (SEARCH, T_1, VAR). *VDS*-applications are programmed in terms of items (orders) specifying *what* to do, and of *handlers* which are interpreting and executing them. For our problem, this means that a processor initiating one of the operations stores the data that controls the state of the computation in a local control block and generates *orders* that specify the operation that has to be performed, the input (a pointer to a polynomial stored in a global variable), and the address of the control block.

Table 1 presents the various class-types we used. The operations T_1 and VAR are parallelized by generating a set of orders, each specifying a set of coefficients to be computed. Since these orders can be computed independently of each other we use standard *VDS* items for their distribution. Since only few operations are active at any one time, we obtained the best results by using the spreading strategy for the distribution of these items. The results of the operations are collected explicitly by *VDS*-messages. In order to model the divide-and-conquer structure of the search algorithm we use the thread concept of *VDS*.

The T_1-operation. The T_1-operation is called by the SEARCH_order-handler which executes the algorithm shown in Figure 1. For the computation it generates for each processor one T_1_order-item storing a pointer to a control-block, a pointer to the global variable containing the polynomial, and a set of indices of coefficients to be computed. Then *VDS* distributes the orders among the processors. The handlers of these orders compute the coefficients according to the specified index-set and store them together with the address of the control-block in T_1_result_msg-items that are then sent back to the processor that initiated the transformation. The result-handler collects the coefficients and, on his part, calls the handler of the corresponding SEARCH_order-item as soon as all coefficients have been computed. When doing this it refers

to the state-information in the control-block (address of the parent-thread, list of coefficients collected so far) determined by the pointer in the result-message.

The VAR-operation. This operation is very similar to T_1. However, the main difference is that the computation of the coefficients should be stopped as soon as two sign variations have been found. This has two consequences: The first regards the way the set of all indices $\{0, \ldots, n\}$ is partitioned among the orders. We want to detect sign-variations quickly—even if there are only few. So we let each subset of indices sample the whole range of indices. A second consequence is that we use a global state-variable containing information about the smallest and largest index of negative and positive coefficients found so far. As soon as these two intervals overlap, we know that there are at least two sign-variations. As long as they do not overlap, we have to continue to compute coefficients. If, eventually, the intervals do not overlap, we know whether all signs are negative, positive, or whether there is exactly one sign-variation from plus to minus or from minus to plus. Obviously, updating this state-variable after every coefficient computation would lead to the earliest possible termination of the process. On the other hand, it would sequentialize the computation and in fact delete any parallelism. It is therefore necessary to strike a compromise between avoiding superfluous work and enabling parallel computation. The optimal number of indices which are computed between two updates of the state-variable depends on the computation time for each coefficient and on the probability of an early termination of the process, something that depends on the input-polynomial. In the next section we will present experimental results to illustrate the situation for various classes of input polynomials.

Similar to the T_1-operation, a handler detecting the termination of the process after an update of the state-variable sends a result-message (`VAR_result_msg`) which contains the number of sign-variations found (either 0, 1, or 2) to the initiating processor. The corresponding result-handler immediately calls the `SEARCH_order`-handler which will continue its work by checking the number of sign-variations found.

The SEARCH-operation. A listing of a simplified `SEARCH_order`-handler only counting the roots is shown in Fig.3. All necessary parameters and local variables of the SEARCH-algorithm are stored inside the *VDS*-item `current` and are accessible via the type-cast in line 3.

First of all, the handler activates a VAR-operation working on the global variable `thread→A`. In the case of an early termination of the VAR-operation where more than one sign-variation has been found, it may happen that another processor just wants to start working on the polynomial when the `SEARCH_order`-handler continues working. In this situation, freeing the global variable would immediately lead to a runtime error. Problems of this kind often arise when several processes are accessing the same global data. For resolving access-conflicts, every Diva-variable contains a handle-counter representing the number of parallel processes having access to it. If this counter is incremented by one (line 6), the variable actually has to be freed twice, which solves the conflict.

The counter `thread→status` ensures that after the termination of the VAR-operation, the execution of the handler is continued in line 9. If the sign-variation (`thread→v`) is bigger than one, the polynomial is read into the local memory and stretched. In order to make the

```
 1. void SEARCH_order_Handler (VDS_Object *current) {
 2.
 3.   SEARCH_Thread *thread = (SEARCH_Thread *)VDS_Object_data (current);
 4.
 5.   switch (++thread->status) {
 6.     case 1 : { diva_increase_handle_counter (thread->A, 1);
 7.                VAR_spawn (current, thread->A, &(thread->v));
 8.                break; }
 9.     case 2 : { if (thread->v <= 1) {
10.                  generate_result (current, thread->v);
11.                  diva_free (thread->A);
12.                  VDS_Delete_object (current); }
13.                else {
14.                  diva_read (thread->A, poly);
15.                  diva_free (thread->A);
16.                  thread->A = store_poly (stretch (poly, -1));
17.                  diva_increase_handle_counter (thread->A, 1);
18.                  VDS_Spawn (current, VDS_New_thread (SEARCH_order,
19.                             new_thread (thread->A)),
20.                             1, VDS_Res_list (1, &(thread->left)));
21.                  T1_spawn (current, thread->A, &(thread->B)); }
22.                break;}
23.    case 3 : { if (is_zero (coeff (thread->B, 0))) thread->roots++;
24.               VDS_Spawn (current, VDS_New_thread (SEARCH_order,
25.                          new_thread (store_poly (thread->B)),
26.                          1, VDS_Res_list (1, &(thread->right)));
27.               VDS_Consume_results (current, 2,
28.                                    VDS_Res_list (2, &(thread->left),
29.                                                     &(thread->right)));
30.               break; }
31.    case 4 : { thread->roots +=
32.                 *((int *)VDS_Result_data(current, &(thread->left))) +
33.                 *((int *)VDS_Result_data(current, &(thread->right)));
34.               generate_result (current, thread->roots);
35.               VDS_Delete_object (current);
36.               break; } } }
```

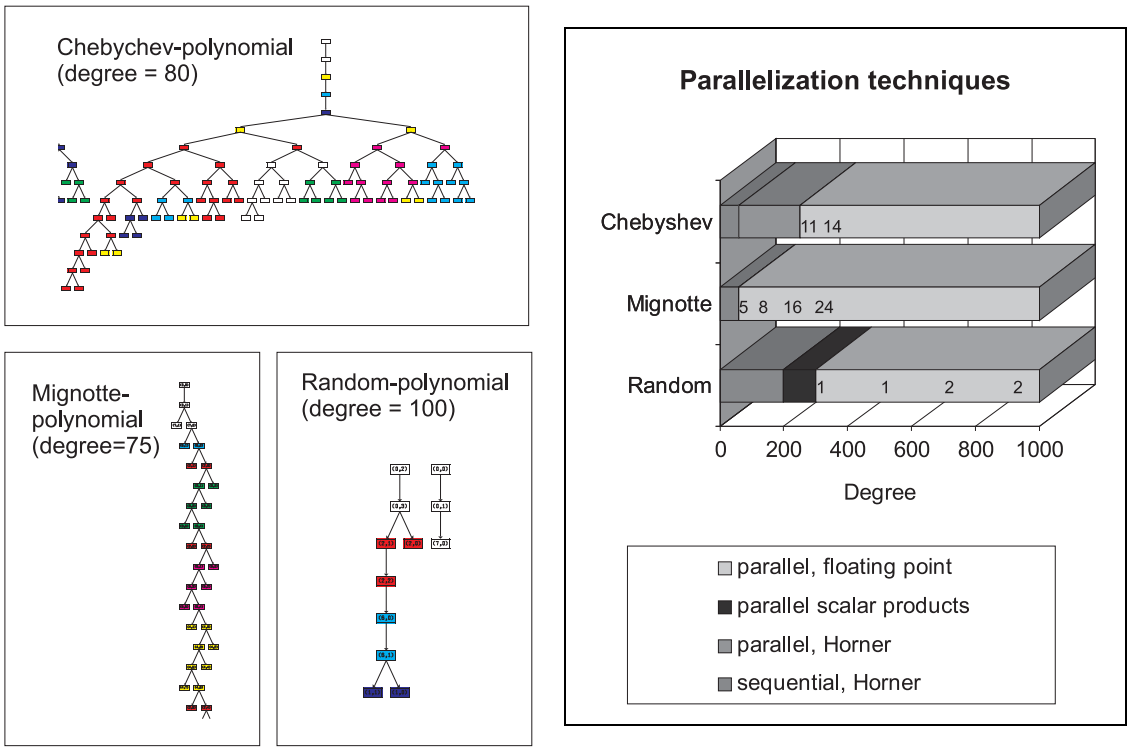
Figure 3: The `SEARCH_order`-handler.

resulting polynomial accessible for all processors, we store it in a new global variable. Again we have to increase the handle-counter of this variable because two independent operations (T_1 and SEARCH) start working on this polynomial afterwards.

After the T_1-transformation is completed, the execution of the handler continues in line 23 first checking whether the the boundary between the left and the right half is a root and then spawning the SEARCH-thread for the right half of the interval. The call of `VDS_Consume_results` tells the *VDS*-system that we are now waiting for the results which will be produced by the generated child-threads. As soon as the requested results are generated, *VDS* puts the corresponding item back into the pool of available items. Afterwards, one of the processors continues working on this item by executing the last step of the `SEARCH_order`-handler which generates the result of the SEARCH-algorithm (in this case the number of roots in the specified interval).

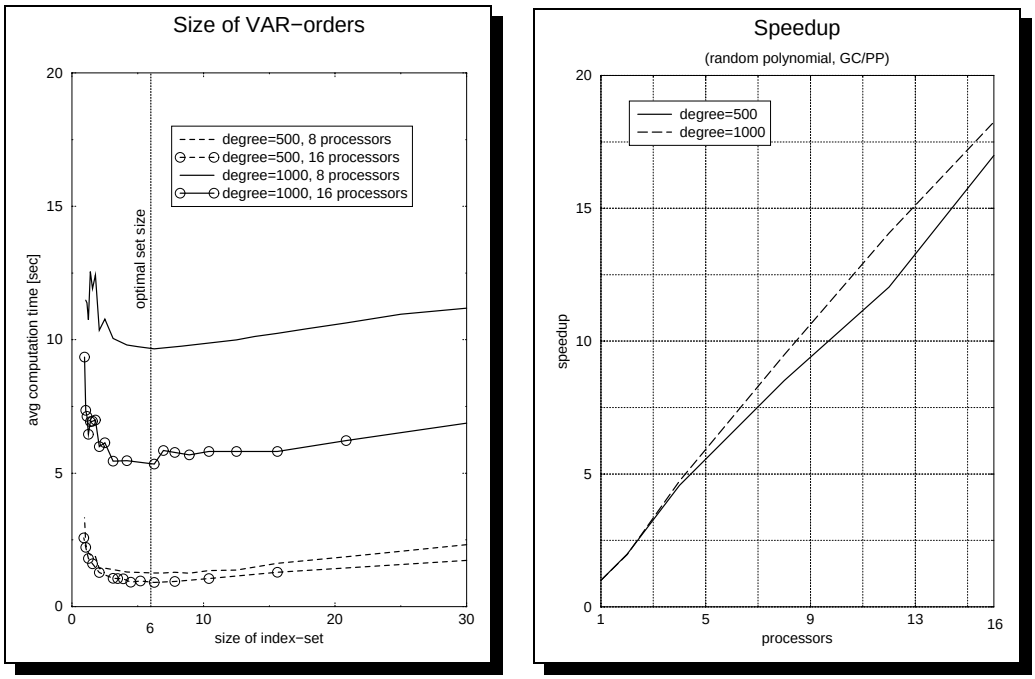
4 Experimental results

We perform all experiments on the GigaCluster/PowerPlus, a distributed memory computer that uses a 4×8 fat mesh interconnection network and 80MHz/64MB Motorola Power PC 601 processors. Our programs, the *Daisy* routines, and *SACLIB* [3] routines that we use are written in C and executed under PARIX 1.3.1. As base of the number system we use $\beta = 2^{29}$.



(a) (b)

Figure 4: Search trees can be very varied (a). Accordingly, there is not one "best method"(b).



(a) (b)

Figure 5: Diagram (a) shows the influence of the size of the index-sets in the `VAR_order`-items on the performance of the parallel VAR-operation. With the optimal set-size nearly linear speedups could be achieved with 16 processors.

We generate various input polynomials, and we measure the time that is needed for isolating their real roots. The polynomials belong to three different classes.

Random polynomials are polynomials whose coefficients are randomly generated integers that are uniformly distributed in the interval $(-2^{10}, 2^{10})$. Polynomials of degree n whose coefficients are independent random variables, uniformly distributed in the interval $(0, 1)$, are known [7] to have an average number of real roots that asymptotically approaches the function $2 \ln n/\pi$. Search trees for random polynomials tend to be small. Such polynomials arise in many applications.

Chebyshev polynomials have only real roots. Chebyshev polynomials (of the first kind) are recursively defined as follows. $T_0(x) = 1$, $T_1(x) = x$, $T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x)$. The roots of $T_n(x)$ are $\cos((2k+1)\pi/(2n))$ where $k = 0, \ldots, n-1$. Search trees for Chebyshev polynomials are wide and deep.

Mignotte polynomials are of the form $A_n(x) = x^n - 2(5x-1)^2$. Such polynomials have a real root to the left of $1/5$, another one to the right of $1/5$, and the distance of these roots is at most $2/5^{n/2+1} < 1/2^{n+1}$, see, for example, [9]. Search trees for Mignotte polynomials are narrow and extremely deep. These polynomials were constructed [11] to give root isolators a hard time.

Figure 4(a) shows examples of search trees that arise in the computation of the real roots of polynomials from the three classes. Figure 5(a) shows how the total computing time depends on the size of the index sets described in Section 3.2. The data was obtained for random polyno-

mials. There seems to be an optimal size, 6, of the index set that is independent of the degree of the polynomial and the number of processors used. Figure 4(b) shows, for each class of inputs and various degrees of the polynomial, which root isolation method performed best in each case. In the case of random polynomials we made experiments up to degree 1000, in the other cases only up to degree 300. For the parallel experiments we used 16 processors. There are four methods. The method labeled "sequential, Horner" is a highly optimized sequential method from *SACLIB* that computes the coefficients of the translation by 1 with the same asymptotic complexity as Horner's scheme. The method labeled "parallel, Horner" is our adaptation of the method by Collins, Johnson, and Küchlin [4]; only left and right subtrees are processed in parallel, the translation by 1 is executed sequentially using the algorithm from *SACLIB*. The method labeled "parallel scalar products" computes the translation by 1 in a distributed way using scalar products and exact arithmetic. Moreover, left and right subtrees are searched in parallel. The method labeled "parallel, floating-point" works in the same way but uses software-supplied multiprecision floating point interval arithmetic. Figure 5(b) shows that the parallel floating point method scales well in the number of processors.

Clearly, for small degrees the sequential method is best. Then there seems to be a degree range where exact methods are best; this may be due to the overhead of interval arithmetic—at least in our current implementation. For higher degrees, the parallel floating point method is best. The bar chart also indicates the precision (number of words) that was needed to perform the floating point calculations in each case.

References

[1] Alefeld, G., Herzberger, J. *Introduction to Interval Computations*. Academic Press, New York, 1983.

[2] R D. Blumhofe, C F Joerg, B. C Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proc. of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, (PPOPP '95)*, pages 207–216, 1995.

[3] George E. Collins et al. `SACLIB` user's guide. Technical Report 93-19, Research Institute for Symbolic Computation, RISC-Linz, Johannes Kepler University, A-4040 Linz, Austria, 1993.

[4] G. E. Collins, J. R. Johnson, and W. Küchlin. Parallel real root isolation using the coefficient sign variation method. In R. E. Zippel, editor, *Computer Algebra and Parallelism. Second International Workshop. Ithaca, USA, May 1990*, number 584 in LNCS, pages 71–87. Springer-Verlag, 1992.

[5] T. Decker. Virtual Data Space - A universal load balancing scheme. In *Proc. of the 4th International Symp. on Parallel Algorithms for Irregularly Structured Problems*, pages 159–166, 1997.

[6] T. Decker and B. Vöcking. The *Daisy* library. Users manual. Available via URL `http://www.uni-paderborn.de/SFB376/projects/a2/daisy.html`

[7] Alan Edelman and Eric Kostlan. How many zeros of a random polynomial are real? *Bulletin (New Series) of the American Mathematical Society*, 32(1):1–37, January 1995.

[8] Johnson, J. R., and Krandick, W. Polynomial real root isolation using approximate arithmetic. In *Int. Symp. on Symbolic and Algebraic Computation* (1997), W. Küchlin, Ed., ACM, pp. 225–232.

[9] Werner Krandick. Isolierung reeller Nullstellen von Polynomen. In J. Herzberger, editor, *Wissenschaftliches Rechnen*, pages 105–154. Akademie Verlag, Berlin, 1995.

[10] B. Maggs, F. Meyer auf der Heide, B. Vöcking, and M. Westermann. Exploiting locality for data management in systems of limited bandwidth. In *Proc. of the 38 IEEE-FOCS*, 1997.

[11] M. Mignotte. Some useful bounds. In B. Buchberger, G. E. Collins, and R. Loos, editors, *Computer Algebra: Symbolic and Algebraic Computation*, 259–263. Springer, New York, 2nd ed., 1982.

Dynamic Load Distribution with the WINNER System

Olaf Arndt, Bernd Freisleben, Thilo Kielmann, Frank Thilo
University of Siegen, Dept. of Electrical Engineering and Computer Science
Hölderlinstr. 3, D–57068 Siegen, Germany
{arndt|freisleb|kielmann|thilo}@informatik.uni-siegen.de

Abstract

Networks of workstations offer large amounts of idle computing capacities. Exploiting these resources imposes the problem of efficiently distributing a given workload without disturbing the workstations' primary (console) users. Recently, specific resource management systems have been developed for this purpose [1, 4, 6, 8, 9, 11, 14].

This paper presents WINNER, a resource management system which has been designed in order to utilize the maximal amount of computing power, imperceptible even by active console users. WINNER automatically assigns tasks to suitable machines, supporting sequential as well as parallel applications, the latter based on PVM [3]. WINNER's modular structure as well as its mechanisms for monitoring and distributing workload will be presented. Performance measurements of a sample PVM application indicate the suitability of our approach.

1 Introduction

Workstation clusters are economically attractive system architectures for parallel processing applications. Because workstations are typically idle for significant fractions of the time they are running, their idle times can be exploited for parallel evaluation of computation–intensive tasks without costly investments in dedicated parallel computing hardware. Success and widespread use of systems like PVM [3] and MPI [5] indicate the feasibility of the workstation–cluster approach to parallel computing.

Workstation clusters are typically evolving systems. This is due to the fact that, for financial and technological reasons, it is not feasible to completely replace existing clusters whenever new components become available. Instead, cluster environments evolve and hence form constantly changing collections of heterogeneous components with different capabilities and requirements. Furthermore, the fraction of workstations in an existing cluster that is available for parallel computations is changing frequently. This is due to (1) individual machines being switched on or off by their primary users, (2) failures of single machines, and most importantly (3) by the workload of their primary users who should not be disturbed by parallel computations additionally operating in the cluster.

These properties of workstation clusters convert task assignment and load distribution into non–trivial tasks. Traditional strategies known from dedicated parallel machines can no longer be applied because they rely on exclusively accessible and furthermore identical processing elements.

For solving these problems, so–called resource management systems have recently been introduced [1, 4, 6, 8, 9, 11, 14]. Such systems aim at exploiting idle computing resources of workstation clusters without disturbing the workstations' primary users.

In this paper, we present WINNER, a resource management system designed for exploiting the maximally available computing power of workstation clusters. It does so by carefully co–scheduling computing tasks with the processes of the workstations' console users while minimizing interferences between both task groups. In the following, WINNER's modular structure as well as its mechanisms for monitoring and distributing workload will be presented. Performance measurements of a sample PVM application indicate the suitability of our approach.

2 WINNER System Design

WINNER has been designed for typical Unix workstation cluster environments, consisting of a central server and several workstations. For a WINNER cluster, the server is required to provide shared file systems and user accounts for all connected workstations. The various tasks of the WINNER system are distributed over three kinds of so–called *manager* processes. Similar to the structure of the Prospero Resource Manager [9], WINNER consists of system managers, node managers, and job managers. The distribution of these manager processes across the workstations is shown in Figure 1. Additionally, there exist several user–interface tools e.g. for status reports and for influencing the usage of a user's workstation. Manager processes and user–interface tools communicate with each other by message exchange over UDP sockets. They hence directly rely on the operating system without the need for additional software products.

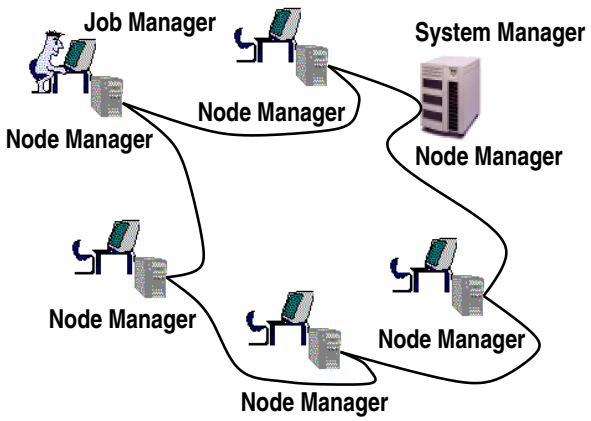


Figure 1: Manager processes in a WINNER cluster.

The system manager is the central server process of a WINNER cluster and as such a single point of failure. It is intended to be located on a cluster's server machine. In the case of a server failure, all machines of a cluster are typically not operative (e.g. due to missing file service). Therefore, the simultaneous failure of the WINNER system can be neglected. The system manager's tasks include (a) collecting the load information of all respective workstations, (b) managing the currently active jobs, and (c) deciding which hosts are assigned to a particular job request. While each WINNER

cluster has exactly one system manager, in the case of multiple, cooperating clusters several system managers interact in order to share workload.

On every host participating in a WINNER cluster, a node manager performs the tasks related to the machine it runs on. First of all, it periodically measures the host's utilization and reports these data to the system manager. Furthermore, node managers are responsible for starting and controlling WINNER processes on their node. Processes of WINNER jobs run at two different priority levels. While a console user is actively working, WINNER processes run at a low priority level in order to avoid disturbing primary users. Without an active console user, WINNER processes run at a slightly higher priority level compared to normally started user processes. This feature has been introduced in order to motivate people to start their jobs via the WINNER system instead of using the traditional Unix tools like rsh. Therefore, the node manager is responsible for switching the process priorities of the WINNER processes on its machine accordingly.

System and node managers run as daemon processes. In contrast, job managers are invoked by a WINNER user in order to execute a sequential or parallel job. Thus, job managers are part of WINNER's user interface. Their duties are acquiring resources from the system manager, starting processes on the acquired nodes via the respective node managers, and controlling and possibly redirecting input and output of the started processes. WINNER's simplest job manager is called wrun. It allows the execution of a sequential (possibly interactive) job. It works almost like the standard UNIX command rsh, except that the job is started on the currently best suited workstation in a cluster.

Because one of its basic design goals is to avoid disturbing the primary (console) users, WINNER provides a user–interface tool by which a primary user can determine how WINNER makes use of his or her workstation: By default, a workstation is only used if the primary user has reached a particular idle time (currently 15 minutes). But using this tool, the user is able to release the local workstation indicating that WINNER can start tasks even if the user is not idle. Alternatively, the primary user can also completely block his or her workstation, such that WINNER will not start any task while the user is logged on until the setting will be changed again.

Additionally, the primary user can request to reduce load caused by WINNER tasks. This is done by WINNER by either migrating processes to different machines, or by interrupting or even aborting WINNER processes in case other methods fail.

3 Load Measurement and Distribution

In order to perform suitable task placement decisions, the system manager must have an accurate global view of the current utilization of the workstations. To achieve this, each node manager provides it with the information related to its own workstation. We will now discuss the duties of both kinds of manager processes.

3.1 Node Manager

On startup, each node manager performs a simple benchmark loop, evaluating the machine's speed in integer operations, floating point calculations, and memory access. This benchmark's result is a single number proportional to the host's overall relative performance. It is reported to the system

manager along with the node name, IP address, and other static data like the total amount of main memory and the number of CPUs on multiprocessor machines.

Afterwards, the node manager regularly queries several load characteristics of its local host and reports them to the system manager if necessary. The set of dynamically changing information consists of the following data:

Load average: The most prominent value is the classical Unix load average. It is calculated by the operating–system kernel by averaging the number of processes in the run queue within a certain time interval.

Run queue length: Unfortunately, most Unix systems only provide very inert load average values, the fastest of which is averaged over the last 60 seconds. As this index changes too slow to be of good use for load distribution decisions, WINNER transforms these values into an index more sensitive to load changes. To understand how this is done, one has to know the way UNIX kernels calculate their load average every i seconds:

$$a_{n+i} = \beta\, a_n + (1 - \beta)\; a$$

where a_n is the last load average i seconds ago, a is the current length of the CPU run queue, a_{n+i} is the current load average to be calculated and β is the smoothing factor determining by which the old average a_n contributes to a_{n+i}. Because this smoothing is performed exponentially, a value of $\beta = e^{-i/60}$ is used for the 60 second average.

Neglecting variations of the run queue length a and assuming an interval of 10 seconds between two consecutive load measurements performed by WINNER's node manager, we get an approximation for the average run queue length a out of two consecutive load values:

$$\begin{aligned} a_{n+10} &= \beta^{10/i}\, a_n + (1 - \beta^{10/i})\, a \\ a_{n+10} &= e^{-10/60}\, a_n + (1 - e^{-10/60})\, a \\ a &= \frac{a_{n+10} - e^{-10/60}\, a_n}{1 - e^{-10/60}} \end{aligned}$$

Using the above equation, WINNER calculates the average run queue length a during the last period. This value reacts much faster to load changes than a_n itself as it is reported by the operating system. This behaviour is shown in Figure 2 in which a (hypothetical) machine with load 0 starts a compute–intensive task at time $t = 0$, yielding a load value of 1 until this task terminates at $t = 60$, which hence resets the load value to 0. The load average as it is computed by the UNIX kernel slowly increases up to a peak value of 0.7 after 60 seconds and then decreases back to 0.21 after 120 seconds. In contrast, WINNER's load average immediately changes its value to 1 and to 0, as soon as it is recomputed after the real load has changed.

CPU states: The relative amount of time the processor(s) spent in one of the different CPU states during the last interval is tightly related to the load average. The states are differentiated into idle time, user time, system time, and nice time. Idle time reflects the amount of time the processor was idle, i.e. executing the system idle thread, whereas user, system, and nice time reflect the time the processor was executing user code, kernel code, and user code with reduced priority, respectively.

This data is often more accurate in low–load situations than the load average. However, once the CPU idle time is close to zero, one cannot distinguish between situations in which there is either just one or whether there are more processes in the CPU run queue.

User idle time: The presence of a primary (console) user is detected. If such a user is present, the time of his or her last input is queried in order to calculate how long the user has been idle.

Free memory size: The amount of free virtual memory is queried. This information can be used to exclude workstations which are low on virtual memory from the list of available hosts for avoiding situations where a process cannot execute successfully due to memory shortage.

Free disk space: The node manager measures the amount of free disk space for temporary files. This is used as a simple threshold by the system manager for deciding whether or not a host is suitable for a given job.

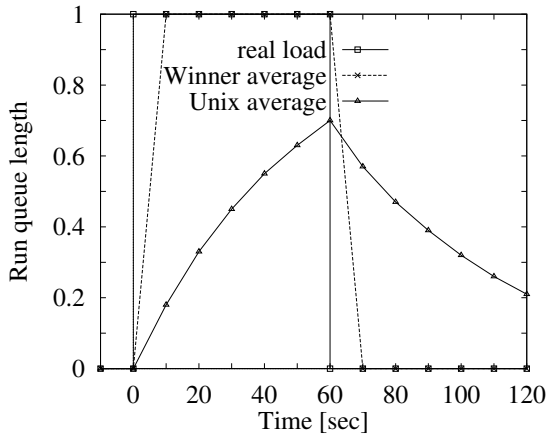


Figure 2: Load averages computed by WINNER and the UNIX kernel.

Finally, the newly collected load information is compared to the latest data set that was sent to the system manager. If these do not differ by more than a certain threshold, the new data is not sent. This policy is used in order to reduce network traffic. However, if no load information has been sent for some time, the data is reported nevertheless for informing the system manager that the node manager is still alive.

3.2 System Manager

The system manager stores the information about its nodes in a table consisting of one entry per host. Each entry contains both the static and dynamic information about this host. Upon reception of new load information from a node manager, the system manager updates the corresponding table entry. Thus the system manager always has a global and current view of all the workstations' utilization.

When a job manager requests a new node for its job, the system manager tries to choose the most appropriate machine. It does so by taking three different aspects into consideration:

First, it checks whether a node is at all available for task execution. This depends on the console user's setting and his or her idle time. E.g. if the console user is active and has not changed the default setting, his/her workstation will not be used.

Second, the node's current load information is checked against the resource requirements possibly specified with the job manager's request. Currently, such requirements can be specified by the user via resource description files or on the job manager's command line. If a node does not fulfill all requirements (like e.g. size of main memory or temporary disk space) it will not be used.

If the node passes these two tests successfully, the processing performance it could provide for the new job is estimated using the following heuristic:

Based on the workstation's base speed S_b (which was measured by the node manager), its current speed S_c is calculated as $S_c = S_b/\lambda$, where λ denotes the fraction of available processing power in the presence of the current workload. Assuming a constant load, λ could be calculated as $\lambda = l_a + 1$ (where l_a is the load average as measured by the node manager). This reflects the fact that after starting a new process, there are λ active processes sharing the CPU.

Because the assumption of constant load is unrealistic, using the load average as it is reported by the kernel yields inaccurate results. E.g. this may be the case when many short running processes cause misleading (high) values for the run queue length. Hence, for achieving more accurate values, WINNER's system manager instead calculates λ by using t_i (the percentage of time the processor was idle): $\lambda = 2 - t_i$, yielding $\lambda \approx 1$ for high percentages of idle time and $\lambda \approx 2$ for higher CPU usage.

In the case of small values of t_i, it can be assumed that the actual workload consists of more than one process. In this case, the load average value should be used for calculating λ in order to reflect the higher load. A threshold of $t_i = 0.15$ is used for choosing between the two cases. This threshold value has been determined empirically as a feasible compromise between possible inaccuracies of both possibilities for computing λ. Its value is hence computed as follows:

$$\lambda = \begin{cases} l_a + 1, & t_i < 0.15 \\ 2 - t_i, & t_i \geq 0.15 \end{cases}$$

Another problem arises when the system manager receives several requests for available hosts within a short time interval. Without further measures, the system manager would choose the same node for all these requests resulting in the fastest host being swamped with new processes. This is due to the fact that it takes some seconds for the selected workstation's load to change and some additional time for this information to be reported back to the system manager. In order to avoid this problem, a bias procedure was introduced to the system manager's calculations which assigns a temporary penalty to recently allocated nodes. This penalty is withdrawn when the system manager receives new load information from the respective node.

Figure 3 shows the output of WINNER's status tool which illustrates a cluster's status based on the system manager's information. On the left side of the status window, the names of the machines are listed. Available machines are printed in black whereas unavailable workstations (with active console users) are printed in green (light grey).

On the left side of Figure 3, the relative speed of the given machines is indicated by the size of the corresponding bars. The total size of a bar corresponds to the statically measured machine speed. The black fraction of the bar indicates the currently available fraction of the processor speed ($1/\lambda$).

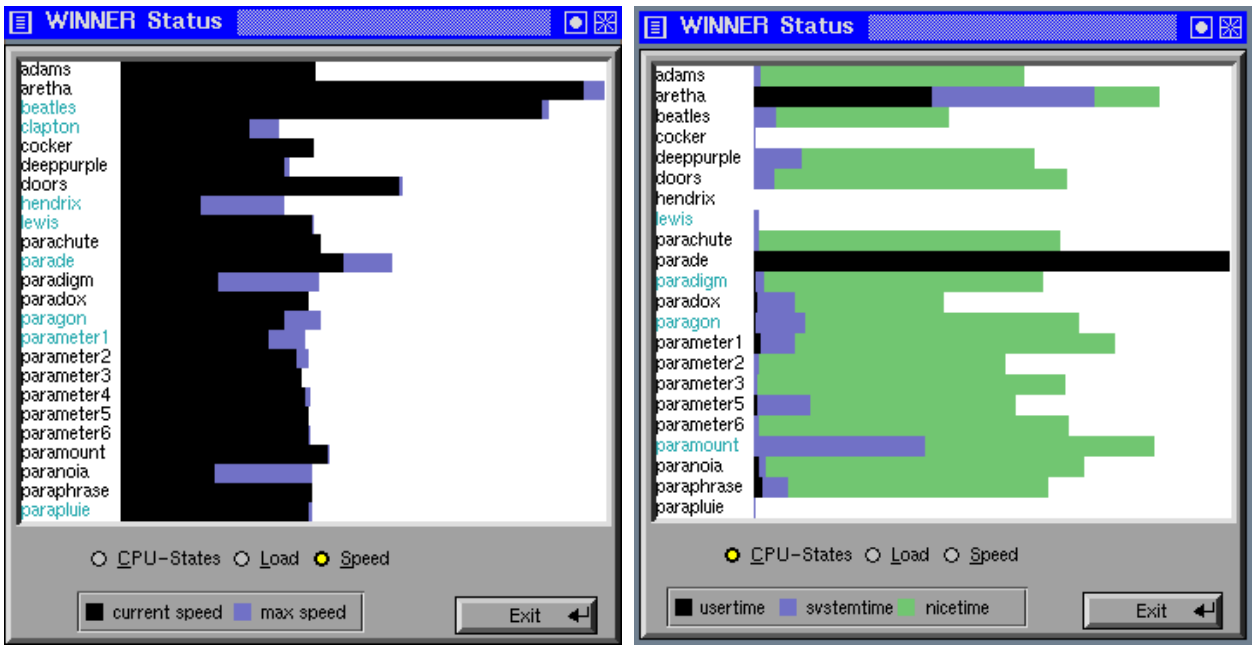


Figure 3: The WINNER status tool, displaying relative speed and processor states.

The bar's blue fraction (printed in grey) is currently in use by other processes. In this snapshot, almost all available CPU power is usable for WINNER jobs.

On the right side of Figure 3, the CPU states (of an unrelated snapshot) are shown. For each workstation, the user time fraction is shown in black. System time is shown in blue (dark grey) while nice time (allocated by WINNER jobs) is shown in green (light grey). Finally, white fractions indicate idle CPU time. In this snapshot, almost all machines run WINNER jobs.

4 Automatic Load Distribution for Sequential Tasks

Once a user decides to start a job via WINNER in order to automatically utilize a cluster's spare computing capacity, he or she does so by invoking the proper job manager. The job manager, in turn, queries the local node manager (by contacting it via its well–known port) for the address of the system manager. After being contacted, the system manager determines a suitable target machine (as described in Section 3.2) and authorizes the job manager at the target machine's node manager. The job manager finally requests program execution at the target host.

During program execution, signals received by the job manager are forwarded to the remote process by sending a message to the controlling node manager. Reversely, the node manager informs the job manager about important events regarding the process, like most importantly, its termination. For interactive jobs, standard input and output are redirected to the job manager's terminal. Similarly, the job manager's display is addressed for X–window applications.

In order to reduce the impact on the target machine's console user, the node manager influences the WINNER processes running on its node. Besides the reduction of process priority and temporarily suspension, processes may also be migrated to different hosts. WINNER can do so with (sequential) processes that are linked with the libckpt checkpointing library [10]. In this case, the

node manager forces the process to write a checkpoint file which is later used for process restart after the system manager has selected a new target host.

5 Automatic Load Distribution for Parallel PVM Tasks

For parallel applications, WINNER provides a job manager for use with the popular PVM system. This job manager automatically selects a number of appropriate hosts for the PVM virtual machine (usually a tedious task when performed manually), starts the user's parallel application, and improves PVM's scheduling by utilizing WINNER's load distribution mechanism.

In order to avoid changes to the PVM system and to application programs, WINNER follows the approach introduced with the CARMI system [11] for integrating a resource management system by implementing the following specific PVM tasks:

PVM resource manager: One task registers as the resource manager for the virtual machine. It hence intercepts certain PVM calls from other tasks in order to influence the scheduling policy. The resource manager receives requests from other tasks out of which the ones regarding the spawning of new PVM processes are of main interest, because they allow the resource manager to establish its own scheduling policy using WINNER's workload distribution capabilities. This way, PVM's default round–robin scheme can be made obsolete.

PVM hoster: This is a single process (per virtual machine) responsible for starting slave PVM daemons on remote hosts (normally done by PVM itself via rsh). This task is necessary when creating or expanding the virtual machine. WINNER registers a hoster process in order to be able to start the PVM daemon processes via the WINNER infrastructure. Hence, the PVM daemons become part of the current WINNER job and can be controlled by the respective node managers.

PVM tasker: Without a dedicated tasker process, new PVM tasks are started directly by the PVM daemon which is running on the respective host. WINNER's PVM job manager registers one tasker on every host that is part of the virtual machine for the current job. This enables the job manager to start newly spawned PVM processes via the WINNER interfaces and hence enables to control them via the node managers.

A user invokes the job manager (wpvm) specifying the desired number of hosts for the virtual machine, as well as the name of the user application's master program and its command line parameters. Just like a job manager for sequential jobs, wpvm contacts the system manager in order to get assigned as many hosts as the user specified. If possible, the system manager will select the most suitable (e.g. the fastest available) workstations. If not enough machines are available, as many hosts as possible are acquired.

The fastest of these hosts is selected to act as the master host for the virtual machine. wpvm spawns the resource manager task on the master host. This task initializes the PVM virtual machine by starting the master PVM daemon and registering itself as the resource manager. In turn, it reports back to the job manager, signaling that the PVM master daemon is up and running.

Next, the hoster task is started on the master host, the set of workstations out of which the VM should be composed of is transmitted to the resource manager, and the virtual machine is expanded to include all these hosts. Finally, one tasker process is started on each workstation.

After all taskers have successfully registered themselves to the VM, wpvm spawns the application's master task on the master host. The master task then will presumably spawn some PVM child processes (via pvm_spawn()). The task placement decisions for these processes are then automatically redirected to the resource manager which uses WINNER for choosing a suitable host for each task.

When all user tasks have terminated, the parallel job has obviously finished. wpvm will now shut down the PVM virtual machine, wait for all auxiliary tasks (resource manager, hoster and taskers) to finish, and finally terminate itself.

6 Example: Parallel Knapsack Packing

For illustrating WINNER's effect on application runtimes, a parallel algorithm for solving the *knapsack problem* has been implemented using the PVM system. The knapsack problem is defined as the problem of finding a set of items each with a weight w and a value v in order to maximize the total value while not exceeding a fixed weight limit. In the implemented *divide–and–conquer* algorithm (taken from [2]), the problem for n items is recursively divided into two subproblems for $n-1$ items, one with the missing item put into the knapsack and one without it.

Whereas the first subproblem is handed over to another processor, the second one is recursively computed within the same node, yielding a dynamically shaped task tree. The assignment of tasks to processors and the related workload distribution is the primary problem of such tree computations. In order to avoid prohibitive runtime overheads, this task tree has to be pruned efficiently. For this purpose, we rely on the heuristic scheme proposed in [2].

Besides efficient pruning, the dynamically created tasks have to be assigned to the available processors. This can directly be done by the PVM system. But because PVM itself has no information about the speed of the individual processors (and the dynamically occupied fractions thereof), PVM typically yields only sub–optimal tasks assignments. Opposed to this, a resource management system like WINNER is able to provide the necessary information and thus contributes to much better task assignments.

Figure 4 shows runtime results of the parallel knapsack program for two different problem sizes, comparing task assignment performed by PVM with the assignment performed by WINNER. The measurements have been performed on a cluster of Digital AlphaStations, running Digital UNIX 4.0 and PVM 3.3.11. In order to provide a fair comparison, all machines have not been used by other users during the runtime measurements. Hence, the given machines have been fully available to the parallel application and WINNER only had to take speed differences into account. Furthermore, the PVM tests used the same machines that had been (automatically) chosen by WINNER before. As can be seen from the figure, the task assignments performed by WINNER lead to faster program execution with both tested problem sizes and with all three numbers of workstations used for our experimental setup. With additional workload caused by interactive users, even better results for WINNER can be expected. The verification of this hypothesis by comprehensive runtime measurements is subject to ongoing work.

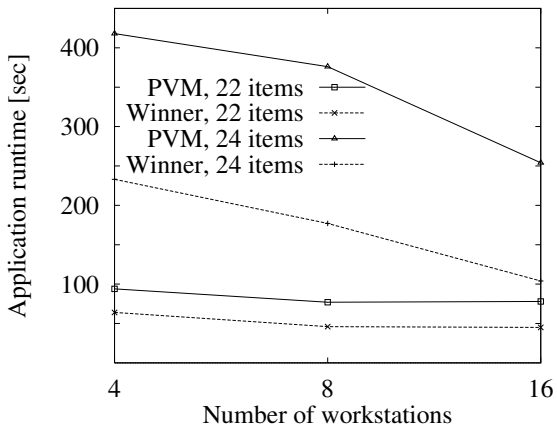


Figure 4: Comparison of task assignment between PVM and WINNER.

7 Related Work

As described above, WINNER's most prominent features include an elaborate and adaptive gathering and evaluation of load characteristics suitable for clusters of workstations with heterogeneous computing speeds. Presence or absence of primary (console) users influence the host selection strategy as well as the execution priorities of active jobs in order to minimize the impact on the console user's work. WINNER's modular structure yields a system which is easily extensible and adaptable to further workstation platforms or kinds of jobs. In the following, we will briefly compare WINNER's features to some related systems.

Piranha [1] provides very good adaptive load distribution for parallel applications. Unfortunately, it only supports a very narrow range of applications, implemented with the Linda programming paradigm. Therefore, it is not suitable for managing general workload.

YALB [14] lacks a proper consideration of console users. Its distributed architecture directly relies on the ability to send low–bandwidth broadcast messages between the participating hosts. Moreover, it does not directly support parallel applications at all.

The Condor system [8] uses a very simple load evaluation strategy for job placement decisions. In fact, it just considers the current load average and the number of active Condor jobs as a threshold. Furthermore it has a very stringent way of reacting to console users (stopping and migrating active jobs). CARMI [11] operates on top of Condor, providing a resource management API to PVM applications. CARMI allows such applications to exploit dynamically changing sets of hosts. However, existing PVM applications have to be rewritten which can be rather tedious as programmers have to explicitly handle the addition and removal of hosts. This problem is somewhat alleviated by the accompanying WoDi library, which provides a comfortable interface for implementing manager/worker applications on top of CARMI.

The Prospero Resource Manager (PRM) [9] lends its modular structure to WINNER. However, PRM's evaluation of utilization information is rather simplistic in that it is only used to decide whether or not a workstation is available for job assignment. Moreover, while a host is assigned to a job, this machine is not available for further jobs. PRM supports PVM applications by providing its own version of PVM, yielding compatibility problems in the advent of future PVM releases.

ALDY [4] is a library of functions providing basic mechanisms and strategies for load distri-

bution of parallel programs. ALDY is not restricted to a single communication platform. Instead it can be adapted by implementing several platform–dependent callback functions, which are automatically invoked by the ALDY scheduler. Application programmers have to define virtual objects along with a mapping to application objects, namely processes. Load balancing is performed my migrating a special kind of active, virtual objects called agents. ALDY determines when an agent should be migrated by considering the number of active agents within a process as its load index. External load information (i.e. memory constraints or active console users) are not taken into account. However, ALDY is useful for balancing the workload of a single parallel application.

Finally, the load balancing extension to PVM introduced in [6] collects real load information which it presents to PVM applications by means of a load vector as well as a dedicated, load balancing spawn function. Its implementation consists of several load daemons, the structure of which resemble WINNER's design. Unlike WINNER, this system is restricted to PVM and therefore only performs load balancing for single PVM applications. In contrast, WINNER works on the middleware level and is hence capable of simultaneously scheduling several (sequential as well as parallel) applications without undesirable interferences between their load distribution decisions.

8 Conclusions and Future Work

The WINNER system constitutes a modularly constructed system for dynamic load distribution in workstation cluster environments. It elaborately gathers and evaluates load characteristics and employs this information for adaptive assignments of workload to available machines. Presence or absence of primary (console) users influence the host selection strategy as well as the execution priorities of active jobs in order to minimize the impact on the console user's work.

WINNER's modular structure yields a system which is easily extensible and adaptable to further workstation platforms or kinds of jobs. Currently, WINNER runs on top of Digital UNIX, Linux, and Solaris. Job managers are available for sequential tasks; one for batch execution and another one for interactive programs (including X–Window applications). Furthermore, a parallel version of GNU–make has been developed acting as a job manager for distributing its tasks via WINNER. Finally, a job manager for parallel PVM programs has been presented.

Work on the WINNER project is still in progress. Currently, we investigate the integration of virtual memory utilization into the scheduling algorithm of our node managers in order to further reduce impacts on console users. Other future developments include the introduction of multiple clusters that exchange workload via a queueing system as well as support for shared–memory multiprocessor machines. Additionally, we are working on improvements of our PVM support by providing load information on the application level as introduced by [6], by implementing task assignments based on the ideas of SED–scheduling [13], and we also plan to integrate checkpointing and migration facilities like to ones implemented in the CoCheck system [12]. Finally, we plan to support MPI [5] and Objective Linda [7] as additional parallel programming platforms.

References

[1] N. Carriero, D. Gelernter, D. Kaminsky, and J. Westbrook. Adaptive Parallelism with Piranha. Technical Report 954, Yale University, Dept. of Computer Science, 1993.

[2] B. Freisleben and T. Kielmann. Automated Transformation of Sequential Divide–and–Conquer Algorithms into Parallel Programs. *Computers and Artificial Intelligence*, 14(6):579–596, 1995.

[3] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine – A Users Guide and Tutorial for Network Parallel Computing*. MIT Press, 1994.

[4] C. Gold and T. Schnekenburger. Using the ALDY Load Distribution System for PVM Applications. In A. Bode, J. Dongarra, T. Ludwig, and V. Sunderam, editors, *Proc. EuroPVM'96*, number 1156 in Lecture Notes in Computer Science, pages 278–287, Munich, Germany, 1996. Springer.

[5] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message–Passing Interface*. MIT Press, 1994.

[6] D. J. Jackson and C. W. Humphres. A simple yet effective load balancing extenison to the PVM software system. *Parallel Computing*, 22:1647–1660, 1997.

[7] T. Kielmann. Programming Heterogeneous Workstation Clusters based on Coordination. In *Proc. ICCI'96, 8th International Conference of Computing and Information*, Waterloo, Ontario, Canada, June 1996. Published as special issue of the CD-ROM Journal of Computing and Information (JCI).

[8] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor – A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference on Distributed Computer Systems*, pages 104–111. IEEE, 1988.

[9] B. C. Neuman and S. Rao. The Prospero Resource Manager: A scalable framework for processor allocation in distributed systems. *Concurrency: Practice and Experience*, 6(4):339–355, 1994.

[10] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent Checkpointing under Unix. In *Proc. Usenix Winter 1995 Technical Conference*, pages 213–223, New Orleans, LA, 1995.

[11] J. Pruyne and M. Livny. Parallel Processing on Dynamic Resources with CARMI. In D. G. Feitelson and L. Rudolph, editors, *Proc. IPPS'95 Workshop*, number 949 in Lecture Notes in Computer Science, pages 259–278, Santa Barbara, CA, USA, 1995. Springer.

[12] J. Pruyne and M. Livny. Managing Checkpoints for Parallel Programs. In D. G. Feitelson and L. Rudolph, editors, *Proc. IPPS'96 Workshop*, number 1162 in Lecture Notes in Computer Science, pages 140–154, Honolulu, Hawai'i, 1996. Springer.

[13] B. Schnor and M. Gehrke. Dynamic–SED for Load Balancing of Parallel Applications in Heterogeneous Systems. In *Proc. Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'97)*, Las Vegas, Nevada, USA, 1997.

[14] S. Stille. Lastbalancierung in verteilten Systemen. Diploma Thesis, Technische Universität Braunschweig, Germany, 1993. (in German).

The PLB-Library: Dynamic Load Balancing on NOWs

F. Meisgen, E. Speckenmeyer

Department of Computer Science
University of Cologne
E-mail: (meisgen,esp)@informatik.uni-koeln.de

Abstract

We present a parallel library for dynamic load balancing on networks of workstations (NOWs), using an efficient, distributed load balancing algorithm, called PLB. Its performance is empirically demonstrated solving the satisfiability problem on an heterogeneous network of 28 workstations and its efficiency is compared with that of a master-slave algorithm.

Keywords: dynamic load balancing; workstation clusters; heterogeneous systems; parallel search algorithm; satisfiability problem;

1 Introduction

Clusters of powerful workstations with high-speed interconnection networks are working at many organisations. Their main task is to supply users with short reply times for interactive work, so typical systems have a low average CPU utilization and high idle times [CS93]. Thus the computing resources of workstation networks may catch up with the power of mainframes or even with special-purpose supercomputers [RNS95]. To make these hidden, economical CPU cycles usable to parallel programs, dynamic load balancing is needed essentially for most applications. This can be done on the operating system level by scheduling tools which support parallel programming, e.g. Stardust [CP97] or on the application level by the usage of parallel compilers or load balancing libraries like Dome [ÁBL+95] and BOB [CR95]. The first library is designed for data parallel programs and performs an automatic load balancing, while the second one can be used for the distributed solution of combinatorial optimization problems. Most libraries hide the complex load balancing process, but force the programmer to use special data structures like vectors or container elements. One of the most flexible approaches based on a process-worker-task programming model is realized by ALDY [GS96].

Our research efforts concentrate on a C++ library called PLB-Lib (Precomputation-based Load Balancing), that reduces the complexity of the load balancing task by using the flexibility of classes. The library is implemented on top of the MPI (Message Passing Interface) [For95], so it needs – in contrast to other libraries – no threads. The PLB-Lib can handle different load balancing algorithms and has already been used for the distributed solution of a data-parallel simulation program [Mei97], a typical problem arising in the area of scientific computing. In this paper we focus on parallel backtracking [BS97], a common tree search technique for the solution of hard combinatorial optimization problems, which

yields strong load imbalances due to the irregular generation of subproblems at runtime. Thus dynamic load balancing has to be performed to avoid idle times and search overhead.

In section 3 we compare two dynamic load balancing algorithms, a master-slave approach and the precomputation algorithm, which has originally been developed for dedicated parallel systems. It is a distributed algorithm, which can achieve a perfect load balance in one sweep over the network. This has been proven for different topologies, like hyper-trees [BS94], and as a special case, for k-ary n-cubes [WS97]. Its quality and scalability has been shown empirically for different applications on parallel architectures of up to 1024 processors [BS97]. Now we extend the precomputation algorithm to heterogeneous workstation clusters, where the sources of load imbalances are enlarged by different power of computers, different speed of network connections and time dependent CPU utilizations caused by the multi-user environment.

2 The PLB-Library

2.1 The Load Balancing Problem in Heterogeneous Networks

The processor network is represented by a vertex-weighted connected graph $G(V, E, w)$ with $w : V \to \mathbb{R}^+$. Each node $v \in V$ is identified by a processor, which has a *power weight* $w(v)$, defined as follows. A sequential process, which executes T time units on processor v will need $w(v) \cdot T$ time units on the benchmark processor v_1. If we choose v_1 to be the fastest machine in the network, this corresponds to the weight definition of Zhang and Yan [ZY95]. Hence, $w(v_1) = 1.0$ and $w(v) < 1.0$ if v is slower than v_1. The power weight of a subset $V' \subseteq V$ of the network is defined as $w(V') := \sum_{v \in V'} w(v)$. The edges $e \in E$ should model the characteristics of the physical network.

Denote by $\lambda_t(v)$ the amount of workload of processor v at time t. The load $\lambda_t(v)$ of processor v is assumed to be proportional to the running time for doing all the work located at v, if v were as fast as the benchmark processor v_1. $\overline{\lambda_t} := \frac{1}{w(V)} \sum_{v \in V} \lambda_t(v)$ is called the *mean weighted load*. The load of a processor v is called *optimal*, if $\lambda(v) = w(v) \cdot \overline{\lambda} =: \lambda_{opt}(v)$.

Let $\vec{G} = (V, \vec{E}, w)$ with $\vec{E} := \{u \to v, u \leftarrow v \mid u - v \in E\}$ be the *total orientation* of G. We call a function $\delta : \vec{E} \to \mathbb{R}$ with $\delta(v \to u) = -\delta(v \leftarrow u)$ for all $v - u \in E$ a *load transfer scheme*. It is called *load balancing scheme*, if after moving $\delta(v \to u)$ units of load from processor v to processor u for every $v - u \in E$ with $\delta(v \to u) > 0$ all processors have optimal load $\lambda_{opt}(v)$.

Usually global load balancing algorithms calculate a load balancing scheme, while local ones try to approximate a balanced load situation. On networks with cycles there exists different load balancing schemes, and when defining a cost function (e.g. minimize the load transfer) this optimization problem can be solved by network flow techniques.

2.2 Components of the PLB-Lib

The PLB-Library covers the following topics:

- Handling of communication topologies, which can be read from files or built automatically by evaluating the speed of network connections.

- Online measurement of power weights.
- Choosing different load balancing algorithms.
- Utilities for parallel parameters and statistics.

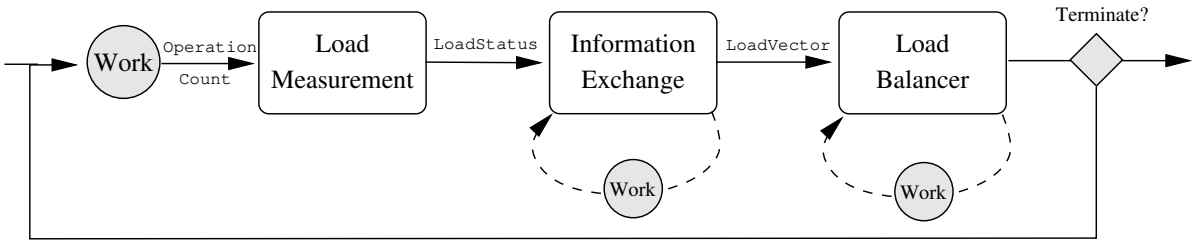


Figure 1: Control flow when using the PLB-Lib

We describe the usage of the PLB-Lib in a parallel program with no global synchronization point as shown in Fig. 1. At the beginning of the main loop all processors try to work a fixed amount of time t. If the application program can calculate an estimation of the work done (e.g. number of array elements processed, number of search tree nodes visited), this information is used to calculate the operations per second, which determines the power weight of all processors. If such information cannot be delivered by the application, the PLB-Lib executes a small benchmark program, e.g. some Dhrystone loops [Wei88]. Furthermore this information can be used to estimate the load of a machine. These values are stored together with other items describing the load distribution in a *LoadStatus* object, serving as input for the information exchange algorithm. The information exchange has to be implemented with nonblocking MPI send and receive functions, such that the application program can be executed while waiting for communication. The algorithm used computes a *LoadVector* object for every processor, such that the combination of all *LoadVectors* describes a load transfer scheme. The balancer exchanges application data according to the values of the *LoadVector*, it receives load via every edge with $\delta(v \to u) < 0$ and sends data for all edges with $\delta(v \to u) > 0$. Furthermore it offers different strategies prefering either a simple passing of incoming packages to target processors or instead sending its own packages.

Special Issues

- Types of exchange algorithms already implemented are master-slave, local-diffusion and precomputation.
- The quality of the load balancing routine strongly depends on the quality of load estimations.
- To avoid communication overhead, the balancer process should be executed only if necessary, i.e. when the length of a problem queue on one processor decreases below a particular limit.

- If the load situation is very imbalanced — a typical situation occurring at the beginning and at the end of computations — it is convenient to execute the load balancing algorithm at short intervals. In the middle of the computation, if all processors have sufficient work, the information exchange algorithm should be started less often. This is done by varying the working time t_{work} within a fixed interval $t_{min} \leq t_{work} \leq t_{max}$.

2.3 The Precomputation Algorithm

The precomputation algorithm calculates a perfect load balancing scheme for a processor tree in the following way: Let $T = (V, E, w)$ be a node-weighted processor tree of height h and with root r. T_v denotes the subtree of T rooted at v.

1. Every processor $v \in V$ determines the load $\lambda(T_v)$ and weight $w(T_v)$ of its subtree by first receiving those information from its sons and afterwards sending to its father:

$$\begin{aligned} \lambda(T_v) &= \lambda(v) + \sum_{u \in sons(v)} \lambda(T_u) \\ w(T_v) &= w(v) + \sum_{u \in sons(v)} w(T_u) \end{aligned}$$

2. The root r of T broadcasts the *mean load* $\overline{\lambda} := \frac{\lambda(T_r)}{w(T_r)}$ to all processors.

3. Every processor $v \in V$ calculates

$$\begin{aligned} \delta(v \to father(v)) &:= \lambda(T_v) - w(T_v) \cdot \overline{\lambda} \quad \text{and} \\ \delta(v \to u) &:= w(T_u) \cdot \overline{\lambda} - \lambda(T_u) \quad \forall u \in sons(v). \end{aligned}$$

This strategy is illustrated by the example in Fig. 2. Each node v contains four entries, the first two entries describe the current loading of processor v given by load $\lambda(v)$ and weight $w(v)$. The third and fourth entry contain the values calculated by the precomputation algorithm: the load $\lambda(T_v)$ and weight $w(T_v)$ of the subtree T_v. The amount $\delta(v \to father(v))$ of load that v must send to its father is represented by the weight of the respective dotted arrow. The direction of an arrow switches if $\delta(v \to father(v)) < 0$, i.e. v receives load from its father.

During the precomputation algorithm only two messages are sent along every edge. Since T is a tree the total work is $O(|V|)$ and the parallel time is $O(h)$.

Note: the parallel time of the load balancing phase is limited by the height times the maximal degree of a node, when called with the scheme generated by the precomputation algorithm. It has been shown for unweighted networks that this time is significantly shorter in the average [BS97], for weighted networks this is still an open question.

2.4 The Master-Slave Algorithm

We compare the precomputation algorithm with a master-slave algorithm, for a comparison to neighborhood-exchange algorithms, see [Böh96]. The main difference between the

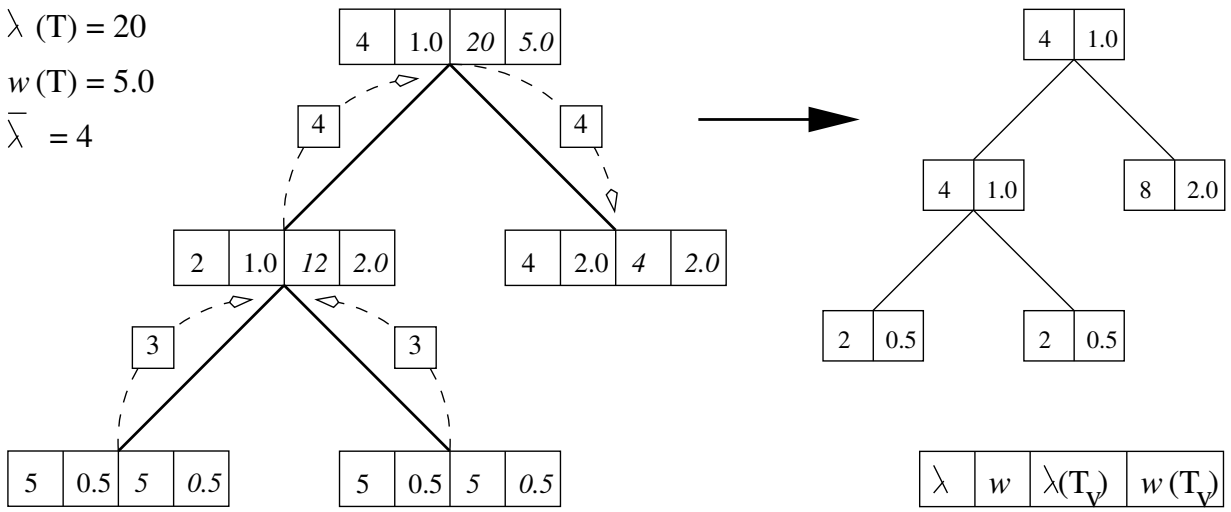


Figure 2: The load balancing process in a weighted processor tree. On the left hand side the processor tree after execution of the precomputation algorithm is shown and on the right hand the balanced tree, all nodes having optimal load.

various load balancing strategies is based on the computation of a load balancing scheme. In the master-slave procedure the master receives a *LoadStatus* object from all slaves, calculates a load balancing scheme according to an optimization criterion (e.g. minimizing the sum of transfered load, minimizing the number of communications) and finally sends a *LoadVector* to each slave. When using the PLB-Lib different objective functions and optimization algorithms can be used without recompilation. In our experiments we have used a mixed strategy, which minimizes the sum of migrated load restricted to transfer schemes with maximum node degree 6. $O(|V|)$ is a lower bound of the parallel execution time for each master-slave algorithm, when ignoring the time needed to calculate a balancing scheme, which is done by the master. When dealing with larger networks and with optimization algorithms based on network flow techniques, this time can be the dominating factor of the parallel runtime. In contrast to the master-slave algorithm the precomputation algorithm needs for communication parallel time $O(log(|V|))$ when using a suitable tree topology and parallel time $O(1)$ to calculate a load balancing scheme.

3 Experimental Results

3.1 Test Problem

As application problem we have selected the Satisfiability problem (SAT), which is well-known to be NP-complete, see [Coo71]. We will only give a rough description of the SAT problem and the parallel solver, for more details see [BS97]. The input of the SAT problem is a Boolean formula in conjunctive normal form (CNF). The output is *true*, iff a truth assignment of the variables exists which satisfies the formula, and *false* otherwise. We use an optimized version of a fast sequential SAT solver [BS97, BKB93], which performs a Davis-Putman backtracking search.

A node in the search tree represents a subproblem p, encoded by a partial truth assignment to the variables. In the branching step the Boolean formula F_p, which corresponds to p, is simplified as much as possible and then an unassigned Boolean variable is selected using a lexicographic heuristic [BKB93]. By assigning *true* and *false* to the chosen variable, we get the two subproblems p_1 and p_2, respectively. One of both is stored in a queue, so the open subproblems are available for load balancing.

In order to illustrate the usage of the PLB-Lib a sketch of the parallel SAT program code is shown in Fig. 3. The constructor of the PlbManager object (line 1) initializes

```
  main(int argc, char** argv) {
1.      PlbManager plb(&argc, &argv); // Initialize MPI and PLB
2.      sat_init(argc, argv); // Initialize user application
        // Usage of parallel parameter files and protocol functions
3.      int algo=plb.prot->Set("loadscheme","algo",PLB_LBS_MS);
4.      PlbBaseLoadScheme* loadscheme=0;
5.      if (algo == PLB_LBS_MS) // Choose load transfer sceme
6.        loadscheme=new PlbLBSMasterSlave(&load_status);
7.      else loadscheme=new PlbLBSPrecomp(&load_status);
        // Work until solution found or all subproblems processed
        while (1) {
8.        int operation_cnt=work(worktimer, ...); // work a fixed time
9.        plb_power_weight.Update(operation_cnt, worktimer);
          // Calculate a load transfer scheme
10.       load_status.Init(my_load, plb_power_weight.Get(), ...);
11.       loadscheme->Init(&load_status);
12.       while (loadscheme->Step()) { work(...); }
          // If necessary, execute load balancer
13.       satbalancer.Init(loadscheme->LoadVector(), ...);
14.       while (satbalancer.Step()) { work(...); }
        }
15.     return 0; // The destructor of PlbManager calls MPI_Finalize()
  }
```

Figure 3: Sketch of the program code for the SAT problem

MPI, processes argument options and creates required objects which handle parameters, statistics, topology and other basic functionalities. The variable `algo` is assigned the value of the corresponding entry in the section `loadscheme` of the parameter file, the default value is `PLB_LBS_MS` (line 3). Depending on this value, the object for calculating a load transfer scheme is created (line 5-7). This constructor needs a pointer to the used LoadStatus object, which has to be derived from the PlbBaseLoadStatus class. In the main loop the user-defined function `work` is called (line 8), which tries to work a fixed time. The time length is varied by PLB depending on the load situation of the last iteration. In the SAT work routine a problem is taken from the queue and either split in two subproblems or

solved directly by the sequential algorithm, a heuristic decision causing variations of the working time. The `load_status` object is initialized with the latest load and weight of the processor (line 10) and some other values, e.g. if a solution has been found. Let k be the number of all assigned variables of subproblem p, then the load of p is estimated by $\lambda(p) = \alpha^{-k}$ ($\alpha > 1$). Experiments have shown that $\alpha = 1.04$ is a good choice for load estimation in case of 3-SAT, i.e. all clauses have length 3, see [Böh96]. During the information exchange phase (line 12) and the load balancing phase (line 14) only short work phases are performed to achieve a fast parallel execution time. The balancer can either be derived from one of the LoadBalancer base classes of the PLB-Lib or constructed from the scratch using the information delivered by the LoadVector only.

We have generated four samples of 3-SAT formulas with 350, 400, 450, and 500 variables and a ratio of the 'number of clauses' and the 'number of variables' of 5.0, typically causing the unsatisfiability of all instances, so the search-tree sizes of the parallel and the sequential algorithm are the same. Instances with this ratio turn out to be hard for Davis-Putnam based SAT-solvers from an experimental point of view, so currently only a few fast implementations of SAT-solvers are able to recognize unsatisfiability of instances of such formulas in reasonable time, see [BKB93, CA93]. For each sample 20 instances were generated and solved three times with the sequential algorithm, thus eliminating external influences. The best of these runs was compared with the best parallel run to determine speedup and efficiency.

3.2 Test Configuration

All experiments were performed on the heterogeneous, geographically distributed workstation cluster shown in Fig. 4. It consists of 28 SUN workstations with a total power weight of approximately 22, compared to a SUN Ultra 1/140. We have made experiments with 7, 14, 21 and 28 processors, where the group from 1-14 are SUN Ultra's (140MHz) with low user interactions, connected by a 10MBit Ethernet. Processors 15-21 are different small (SPARCstation-5) and medium (SPARCstation-20) workstations, often used by interactive users. Finally, processors 23-28 are located at a different site, consisting of heavily loaded SUN Ultra's with 167/200MHz processors. The network connection to this side has an average bandwidth of 300KByte/sec and an average latency of about 5ms, compared to 1ms in the LAN, measured by our MPI implementation of the PingPong benchmark [Hoc96, section 3.5].

These experiments were made while the workstations were performing their usual tasks, thus we were working in a nondedicated environment. Furthermore all tests were run with low priority due to strong requirements of system administration. The power weights of the processors were determined during runtime and scaled to the benchmark processor 1.

The program was compiled with g++ 2.7.2, with maximum optimization, it uses the mpich 1.1 implementation of the MPI 1.1 standard [For95].

3.3 Performance Metrics

In contrast to homogeneous computing the definitions for performance metrics like speedup, efficiency and scalability are differing in the field of heterogeneous computing [DBP94,

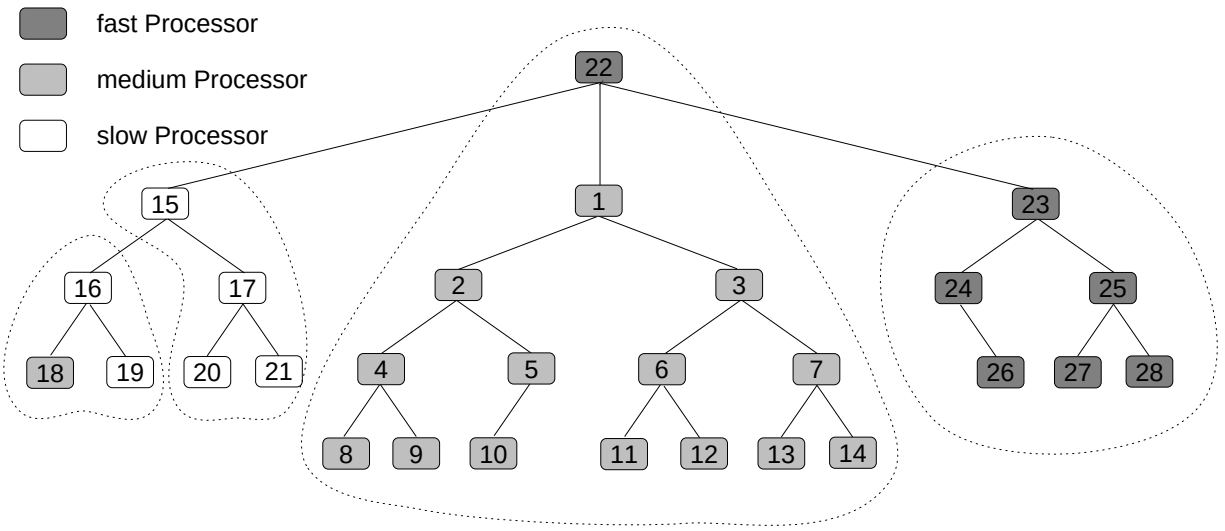


Figure 4: Logical tree structure of the physical network, local Ethernets are surrounded by dashed lines.

ZY95]. The *speedup* S_p for a heterogeneous network $G(V, E, w)$ on problem p and base machine v_1 is calculated as follows: For a subset $U \subseteq V$ define $t(U, p)$ to be the time required to solve problem p on processor set U, then $S_p := t(v_1, p)/t(V, p)$. Then the efficiency E_p is defined as $E_p := S_p/w(V)$. While the speedup depends on the choice of the base processor v_1, the efficiency is independent. Let v_1, v_2 be different base processors, w_1 and w_2 the accompanying weight functions and $S_p^1, S_p^2, E_p^1, E_p^2$ the corresponding speedup and efficiency functions.

$$S_p^2 = \frac{t(v_2, p)}{t(V, p)} = \frac{w_1(v_2) \cdot t(v_1, p)}{t(V, p)} = w_1(v_2) \cdot S_p^1$$

$$E_p^2 = \frac{S_p^2}{w_2(V)} = \frac{w_1(v_2) \cdot S_p^1}{w_1(v_2) \cdot w_1(V)} = E_p^1$$

Thus for examining performance results for heterogeneous networks, one should prefer efficiency to speedup. In this case any processor can be chosen for benchmarking.

3.4 Results

In table. 1 the master-slave algorithm is compared to the precomputation algorithm, which uses the communication topology shown in Fig. 4, on the sample with 450 variables. It turns out that for small numbers of processors (up to 21) both strategies perform comparably. Although the precomputation algorithm transfers significantly more problems than the master-slave algorithm (column 4 and 8, average number of problems per processor) resulting in a higher time needed for the balancing phase (column 5 and 9, average time per processor), this is compensated by lower idle times. However, for the set of 28 processors including the machines at the geographically separated site, the precomputation algorithm is superior compared to master-slave. The idle times of about 20 seconds of the

Table 1: Precomputation vs. master-slave, results for 3-SAT with 450 variables and 2250 clauses

net-work size	Precomputation efficiency	idle [sec]	problems migrated	baltime [sec]	Master-Slave efficiency	idle [sec]	problems migrated	baltime [sec]
7	0.98	1.23	319.11	0.37	0.99	1.18	119.95	0.23
14	0.98	1.06	479.66	0.60	0.96	1.48	163.04	0.34
21	0.93	1.53	565.43	1.26	0.92	1.64	158.23	0.96
28	0.92	2.92	436.06	1.97	0.78	20.13	72.54	5.54

master-slave algorithm are caused by the slowdown of the information exchange phase due to the higher latency of the network connection.

Fig. 5 shows test results with the precomputation algorithm for problem instances of different sizes with 350 variables up to 500 variables. The average parallel runtime on 28 processors varies between 15 seconds and 1148 seconds. As can be seen from the data, parallel runtimes below one minute are not reasonable (350 variables $\rightarrow$ 15 seconds, 400 variables $\rightarrow$ 50 seconds). An efficiency of about 90% is achieved for samples with more than 450 variables, and an average parallel runtime of approximately 4 minutes. Idle times are small, and are increasing slowly with network and problem size.

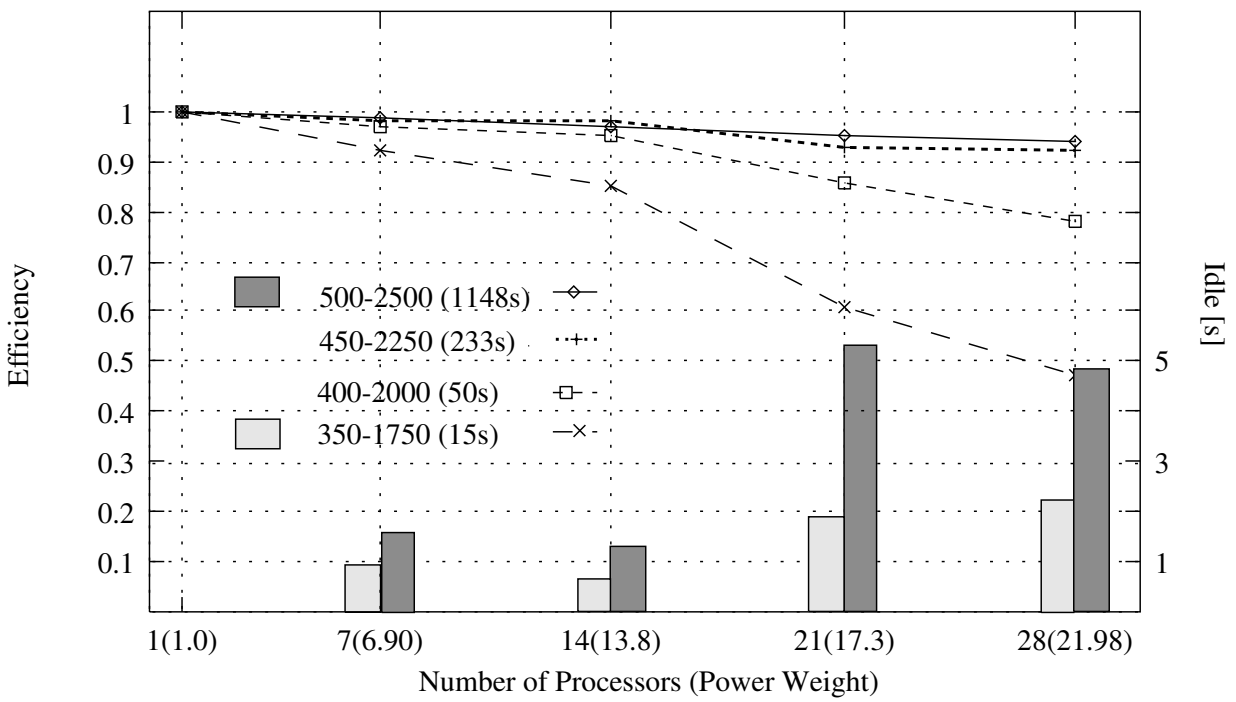


Figure 5: Scalability of the precomputation algorithm

4 Conclusions and Future Work

We have presented a rough description of the PLB-Library and the precomputation algorithm for load balancing on heterogeneous networks. The efficiency of the PLB-Lib is demonstrated in experiments with a parallel SAT-Solver and the precomputation algorithm is compared to a master-slave algorithm.

In our current research we try to reduce the load migration of the precomputation algorithm and to allow combinations of different load transfer scheme generators on separated parts of the network. Another interesting question focuses the choice of an optimal virtual topology. Furthermore we are looking for a tool to visualize the load and network characteristics provided by the PLB-Lib.

References

[ÁBL+95] J.N.C. Árabe, A. Beguelin, B. Lowekamp, E. Seligman, and M. Starkey. Dome: Parallel programming in a heterogeneous multi-user enviroment. Technical Report CMU-CS-95-137, Carnegie Mellon University, April 1995.

[BKB93] M. Buro and H. Kleine Büning. Report on a SAT competition. In *EATCS Bulletin*, volume 49, pages 143–151, February 1993.

[Böh96] M. Böhm. *Verteilte Lösung harter Probleme: Schneller Lastausgleich.* PhD thesis, Universität zu Köln, January 1996. Shaker Verlag, Aachen.

[BS94] M. Böhm and E. Speckenmeyer. Effiziente Lastausgleichsalgorithmen. In R. Grebe and R. Flieger, editors, *Parallele Datenverarbeitung aktuell: TAT'94*, pages 53–60. IOS Press, 1994.

[BS97] M. Böhm and E. Speckenmeyer. Precomputation-based load balancing. In F. Hoßfeld, E. Maehle, and E. W. Mayer, editors, *Proceedings of the 4th PASA Workshop,1996*, pages 177–194. World Scientific, 1997.

[CA93] J. M. Crawford and L. D. Auton. Experimental results on the crossover point in satisfiability problems. Technical report, DIMACS, Rutgers University, New Brunswick, NJ 08903, 1993.

[Coo71] S. A. Cook. The complexity of theorem-proving procedures. In *Proc. of the 3rd Annual ACM Symposium on Theory of Computing*, pages 151–158, 1971.

[CP97] Gilbert Cabillic and Isabelle Puaut. Stardust: An environment for parallel programming on networks of heterogenous workstations. *Journal of Parallel and Distributed Computing*, 40(1):65–80, January 1997.

[CR95] B. Le Cun and C. Roucatrol. BOB: a unified platform for implementing branch-and-bound like algorithms. Technical Report N* 95/16, Université de Versailles, May 1995.

[CS93] Clemens H. Cap and Volker Strumpen. Efficient parallel computing in distributed workstation environments. *Parallel Computing*, 19(11):1221–1234, 1993.

[DBP94] V. Donaldson, F. Berman, and R. Paturi. Program speedup in a heterogenous computing network. *Journal of Parallel and Distributed Computing*, 21(1):316–322, 1994.

[For95] The Message Passing Interface Forum. The Message Passing Interface Standard. http://www.mcs.anl.gov/mpi/, June 1995.

[GS96] C. Gold and Th. Schnekenburger. Using the ALDY load distribution system for PVM applications. In *Parallel Virtual Machine - EuroPVM'96*, LNCS 1156, pages 278–287. Springer, 1996.

[Hoc96] R. W. Hockney. *The Science of Computer Benchmarking.* SIAM, 1996.

[Mei97] F. Meisgen. Dynamic Load Balancing for Simulations of Biological Aging. *International Journal of Modern Physics C*, 8(3):575–582, June 1997.

[RNS95] V. J. Rego, H. Nakanishi, and V. Sunderam. On the effectiveness of superconcurrent computations on heterogeneous networks. *Journal of Parallel and Distributed Computing*, 24(2):177–190, February 1995.

[Wei88] R. P. Weicker. Dhrystone benchmark: Rationale for version 2 and measurement rules. *SIGPLAN Notices*, 23(8), 1988.

[WS97] Min-You Wu and Wei Shu. DDE: A modified dimension exchange method for load balancing in k-ary n-cubes. *Journal of Parallel and Distributed Computing*, 44(1):88–96, July 1997.

[ZY95] X. Zhang and Y. Yan. Modeling and characterizing parallel computing performance on heterogenous networks of workstations. In *Proceedings of the 7th IEEE Symposium on Parallel and Distributed Processing*, pages 25–34. IEEE Computer Society Press, Oct 1995.

Object Mobility in ARTS

Lars Büttner
GMD–FIRST
Rudower Chaussee 5, Bldg. 13.10, 12489 Berlin
lb@first.gmd.de

Abstract

ARTS is an open object oriented framework that provides customizable runtime services for parallel processing within a global distributed object space. This paper focuses on migration mechanisms which enable dynamic load balancing in this environment. They are designed to efficiently support different types of application objects with respect to their data structure and granularity. Performance numbers are given for several migration mechanisms. These mechanisms have been integrated in several applications for computational fluid dynamics.

1 Introduction

Originally ARTS was a remote object invocation (ROI) [1] system designed as an extension of the parallel PEACE operating system family [2]. In PEACE system services as well as C++ applications are described by means of annotated C++-classes. These classes are pre-processed by a tool called DOG [3] which produces stub routines for remote object invocation and uses any C++ compiler as back-end. Efficiency and minimal abstractions were major design goals such that even frequently used low-level system services could apply high-level cooperation paradigms instead of crude message passing.

Today ARTS is an open object oriented framework which provides the basic services for distributed and parallel processing within a global object space. The global space is populated with both *active* and *passive* objects that reside within possibly distributed *domains*. Active objects are statically associated with an individual flow of control (a thread) whereas passive objects can only be manipulated by active objects. Domains are virtual processors (servers) providing logically encapsulated object spaces with controlled access to all objects inside. Dependent on a server's configuration, many domains may share an address space or may have separate address spaces either on the same processing node or somewhere in a network.

This paper describes the basic migration mechanisms that enable dynamic load balancing in global object spaces. An ensemble of mechanisms for specific needs is presented in section 2. They have already been integrated and evaluated in several computational fluid

dynamics (CFD) applications. Section 3 explains the integration principle. Performance figures for these migration mechanisms are given in section 2.3. Other approaches to object mobility are shortly described in section 4.

2 Object Mobility

Object mobility is a key concept for dynamic load balancing [4, 5, 6] and reconfiguration issues. ARTS therefore provides migration mechanisms to move objects between domains during application runtime. The basic model of a fixed global object space is extended to a dynamically reconfigurable space. A base abstraction is used to introduce mobility – class `MobileObject`.

2.1 Migration of Passive Objects

Instances of classes derived from `MobileObject` are free to move between domains. Class `MobileObject` provides the appropriate migration interface. Migration is initiated through a `migrate()` operation on the object. That is, the migration itself is executed by the control flow handling all other operations on this object. Therefore the migration is not preemptive. Non-preemptive object migration has the major advantage that synchronization of object migrations can use the basic synchronization of method invocations and thus costs no extra overhead.

Upon migration a remote copy of the object is created by means of a virtual `clone()` method that needs to be defined by the application class or can be automatically generated. Figure 1 shows all migration phases in detail. If the mobile object has no dynamic data phases 2, 4 and 5 are omitted (depicted with dashed lines). Migration phases are numbered to indicate their execution order.

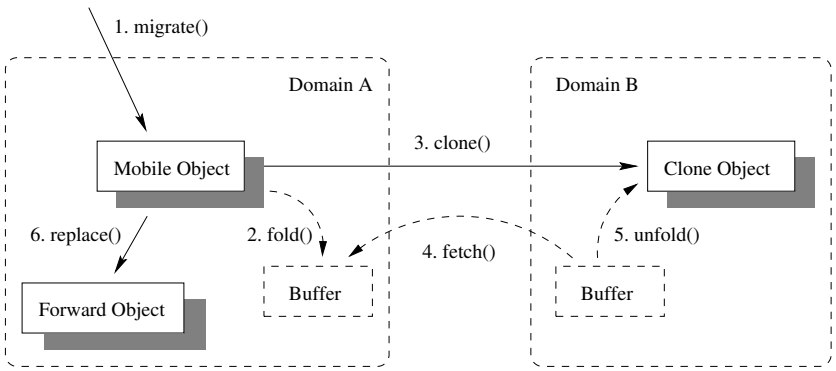


Figure 1: Migration mechanism

The copy of the object is created by means of a remote copy constructor. This new clone is initialized with the data of the original object. That data may of course contain references to other objects in the global space. Thus all remote references are retained.

Only if the original object holds machine level pointers to objects in the original domain serialization of these dynamically allocated data has to be performed (`fold()`) before the remote copy constructor is executed. This constructor then needs to fetch the serialized data by means of a remote `fetch()` operation and reconstruct (`unfold()`) the serialized data.

Buffer management for serialized data is very simple in the presented approach. The migration destination domain (domain B in Figure 1) is also informed about the buffer size during invocation of `clone()`. Thus the buffer can be allocated and afterwards the data are read into this buffer. The arrow directed from domain B to domain A indicates that this read operation is initiated by the migration destination domain.

After this cloning procedure a `Forward` instance with the same identifier like the cloned object is created to replace the original object (`replace`). The original object is deleted. This forward object replies to all messages with a message that causes the client to send the message to the new destination (Fig. 2).

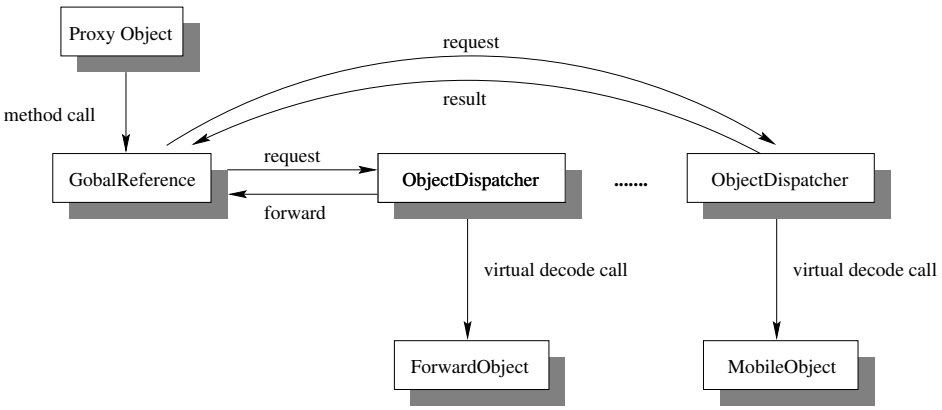


Figure 2: Message Forwarding

If an object moves multiple times the forwarding chain will naturally grow. Therefore we apply dynamic chain shortening. Every `ForwardObject` instance caches the reference of the domain which finally replies to the message. Thus the direct connection is automatically reestablished after the first call through the forwarding chain. The forward objects in the chain are collected when the migrated object is destroyed. The collection of forward instances is initiated by sending a destruction request to the head of the chain to recursively remove the chain.

A complexity analysis of object finding in distributed systems is given in [7]. We have chosen a chain model similar to [8] because a home based model requires extra communication during the migration phase. Our application analysis shows that chains are usually short (almost always one hop only). This stems from the fact that parallel applications typically use all of their remote references at least once during a computation phase. If object redistribution is applied between computing phases at most one hop is inserted for all moved objects. This hop is removed after the first use of the respective reference.

Therefore we wanted to minimize the migration overhead itself to make load balancing feasible in highly dynamic systems.

The presented migration mechanism could lead to deadlocks if circular migrations are executed. Most of the time the load balancing algorithm itself does not issue those migrations which lead to circular dependencies. However if distributed load balancing algorithms are used deadlocks can occur in boundaries between load balancing domains. Thus a modified version of the basic migration mechanism exists which prevents deadlocks.

The simplest solution is to asynchronously execute phase 3 (Fig. 1), that is, the migration source domain is not blocked until the migration is finished and is able to process incoming `clone()` operations even if they are issued from the previous migration destination domain. Unfortunately this is only efficient if there is no buffer of dynamic data to transfer, otherwise there had to be a costly resource management operation to protect the buffer from deletion before the migration is finished.

The deadlock-free migration mechanism therefore addresses during phase 3 a special control flow in each migration destination domain that solely handles incoming `clone()` operations and afterwards hands over the newly created clone to the proper control flow.

At present there are three migration mechanisms the application/system programmer can choose from:

- deadlock-free migration of objects solely containing static data
- migration of arbitrary objects with possible deadlocks (load balancing algorithm has to take care of)
- deadlock-free migration of arbitrary objects

Section 2.3 shows the performance of all three mechanisms with the obvious conclusion – the more complex the mechanism the more costly the execution.

2.2 Migration of Active Objects

Active objects can either be *autonomous* or *recurrent* (Fig.3). Autonomous objects redefine the default `action()` method and can explicitly accept new messages during method execution using an `accept()` method. Thus an autonomous object always has both a dynamic stack context and a (passive) object context. In contrast recurrent objects do not redefine the default `action()` method of `ActiveObject` and never accept new messages before the execution of a method is completed. Therefore the stack context of a recurrent object is (logically) undefined between two method calls and a recurrent object can be restarted from scratch when the object is migrated (section 2).

We can also migrate recurring active objects and replace them with active forward objects with a minimal stack. Migration of recurrent active objects is essentially the same as passive object migration because recurring active objects do not have a stack context at migration time. At the destination site the thread of the recurrent object is therefore simply restarted.

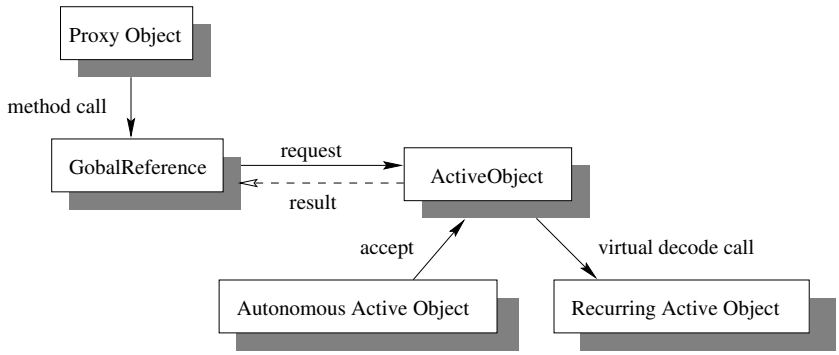


Figure 3: Invocation of Active Objects

On top of this approach additional mechanisms have to provide the handling of runtime contexts. Autonomous active objects can only be moved if either a compiler is able to generate code to handle the stack layout at the new destination or the autonomous object can recover its runtime context from the passive part of the object's state.

2.3 Performance Measurements

All measurements were done with our system running with PEACE on MANNA . MANNA is a massively parallel computer with two 50 MHz i860 processors and a 50 MBytes/sec crossbar connection. Table 1 lists the execution times of PEACE and the total time of language level calls using ARTS . All operations are executed between separated address spaces on different MANNA nodes. PEACE provides an inter-process communication (IPC) that realizes a rendezvous protocol consuming 165 μsec exchanging a fixed size packet of 64 bytes. Bulk data transfer is possible using the fetch operation. The caller reads a consecutive memory area from the callee's address space. The minimum time for this operation is 126 μsec. Depending on the data size additional transfer time is needed.

Operation	Time (μsec)
Inter-Process Communication (IPC)	165
bulk data transfer (fetch)	126
ROI with argument size not exceeding an IPC packet	175
ROI with implicit bulk data transfer	318
remote object creation	266

Table 1: Times of Basic Operations (PEACE and Applications using ARTS)

The last three lines of table 1 list the total times of application level operations on top of ARTS , that is they include compiler generated stub code, ARTS operations and PEACE calls. A remote object invocation with argument size not exceeding an IPC packet needs

175 μsec. We approximated the basic overhead of our platform to be 10 μsec by subtracting communication times from an empty ROI. If caller and callee reside in the same address space (local object invocation) and the same PEACE communication primitives are applied, the overhead drops to 5 μsec. This is caused by caching effects. Thus the measured 10 μsec overhead describes almost the worst case. A remote object invocation including an implicit bulk data transfer is mapped to one IPC to synchronize caller and callee and a following fetch operation. Thus the overhead of this ROI call is 27 μsec (318 - 165 - 126).

Our migration mechanism has some operations that can be executed in parallel. Optimization merely exploits this parallelism by overlapping some migration phases. Table 2 lists three possible variants together with the sequential version as comparison.

Migration Mechanism	Time (μs)
sequential migration	**526**
partial parallel migration (overlapping migrate() and migration)	**406**
parallel migration (overlapping migrate(), clone() and replace(); no fold/fetch/unfold)	**178**
deadlock-free migration (overlapping migrate() and migration)	**602**

Table 2: Performance of Migration Mechanisms

One obvious improvement can be done in every case. Migration activation should be non-blocking. This drastically reduces the migration time. The caller (for instance the load balancing algorithm) is only blocked to send the migration request to the mobile object and could for instance issue several more migration requests during migration execution. Another optimization are class specific new/delete operators. They are somewhat useful in case of multiple migrations from or to one specific domain.

If the object has no dynamic data the migration reaches its maximum speed. Referring to Figure 1 all migration phases shown with dashed lines can be omitted. Since the migration destination domain has not to be synchronized with the source domain (`fetch()` is not executed) the clone object creation is non-blocking. This case could be used for migration of small objects.

Deadlock-free migration of arbitrary objects is surprisingly more expensive than the sequential version. Although migration request and migration are overlapped a lot of time is consumed at migration destination site. A second control flow (co-routine) has to be scheduled. Several synchronized hash table accesses and function calls during clone installation on the actual control flow are executed.

Our experiences with numerical applications show that either the object is small and solely contains static data or it consists of static and dynamic data and is at least of medium size [9].

3 Mobile Application Objects

Application level objects are mobile if their class declaration inherits from class `MobileObject` which represents the described migration mechanisms.

```
class Block : public MobileObject { ... };
```

ARTS applications are equipped with an underlying dynamic, adaptive load balancing system that in turn relies on mobile objects. The coupling of load balancing and application is accomplished through a class `BalancedObject`. Using inheritance this class implements the connection between load balancing (`BalancedObject`), monitoring (`LoadSensor`) and object mobility (`MobileObject`).

```
class BalancedObject : public MobileObject, public LoadSensor  {
/*!shared!*/ private:
    long obj_load;
    ...
public:
    virtual void balance()
    {
        obj_load = loadIndex();
        ...
        migrate( /* destination */ );
        ...
    }
    ...
};
```

In this example load balancing is initiated by invocation of `balance()`, that is, every instance of `BalancedObject` decides the necessary steps to evenly distribute the work load. Thus a fully distributed load balancing algorithm is realized. Method `balance()` first measures the local system load and migrates its own object to some selected destination.

Mechanisms and strategies of our load balancing systems can be adapted to application specific needs by method redefinition and configuration during runtime.

A code fragment of a real CFD application (NSS2000/NSS3000, developed by GENIAS Software GmbH, Neutraubling) highlights the applicability of our approach to object mobility. NSS2000 (two-dimensional) and NSS3000 (three-dimensional) are tools which solve Euler- or Navier-Stokes-Equations in flux formulation on non-equidistant coordinates. Simulations of compressible atmospheric flows (stationary or instationary) for Mach 0.3 to Mach 25 can be computed.

This example shows a method redefinition (method `loadIndex()`) to measure the load index. Application NSS3000 uses the number of iterations a `Block`–instance needs to reach local convergence as a prediction of the future object load. The default processor utilization is inapplicable because neither it could be decided which `Block`–instance to migrate nor is

it a reliable load prediction. In this case the redefinition adapts the monitoring system to the application and leaves everything else unchanged.

```
class Block : public BalancedObject {
/*!shared!*/ private:
    long last_NumOfIterations;
    ...
public:
    long loadIndex() {
        return last_NumOfIterations;
    }
    ...
};
```

4 Related Work

Nowadays a large number of systems provide object mobility. They can be categorized into three approaches:

1. based on a programming language
2. based on a library as extension to a widely used programming language
3. based on interface definition languages using stub generators

An example for the first approach is Emerald [10]. It is a single language system providing an object-oriented programming language, a sophisticated compiler and an appropriate runtime system. Although the Emerald concept is sound and most advanced it lacks the applicability in common numerical programming because of the special programming language which is not used in industry.

Dome [11] is used to develop Grand Challenge applications, such as molecular dynamics, nuclear physics and distributed simulations on heterogeneous networks of workstations. Object mobility is provided through libraries of distributed objects. Dome is implemented as templated library in C++, thus it belongs to the second approach. However the flexibility of Dome is restricted to distributed data structures given by the library such as scalars or vectors.

COOL2 [12] explores the model of a centralized computing system on a distributed memory environment. Object mobility depends on virtual memory on top of a shared memory system which is provided by the CHORUS micro-kernel. Thus this approach is to heavyweighted to be used in massively parallel system.

Compared to these approaches ARTS has the strong advantage to be efficient and flexible at the same time as well as applicable in everyday application programming due to the use of C++ with some minimal extensions (special comments in class declarations evaluated by a stub generator). In this connection object mobility and thus dynamic load balancing are for free.

5 Conclusion

Mobility of application level objects is the premise for dynamic load balancing. In general application behavior is not exactly predictable (if at all) [4, 5] which determine migration mechanisms as time critical. Migration overhead has to be neglectable in comparison to the achieved speedup. Thus efficient migration depends on following parameters:

- frequency of application load changes
- ratio of communication and computation
- granularity of application objects

The most important aspect is the dynamic behavior of the work load. This behavior together with object granularity specify the constraints for effective dynamic load balancing.

The ARTS approach to object mobility is suitable and efficient under those constraints as shown with several CFD applications. The initial overhead of all our migration mechanisms can be neglected compared to the computation time of one local iteration in CFD applications although the final transfer time for all object data consume several milliseconds (due to large buffers of serialized data). Furthermore our experiences using the presented migration mechanisms (as well as the ARTS platform itself) have shown that they can be integrated with minimal effort into existing parallel object-oriented applications.

References

[1] H. M. Levy, E. D. Tempero. "Modules, Objects, and Distributed Programming: Issues in RPC and Remote Object Invocation", *Software—Practice and Experience*, vol 21(1), January 1991.

[2] W. Schröder-Preikschat. "The Logical Design of Parallel Operating Systems", *Prentice Hall International*, 1994.

[3] J. Nolte, W. Schröder-Preikschat. "An Object-Oriented Computing Surface for Distributed Memory Architectures", *Proceedings of the Twenty-Sixth Annual Hawaii International Conference on System Sciences*, HICSS-26, January, 1993.

[4] D. L. Eager, E. D. Lazowska, J. Zahorjan. "Adaptive Load Sharing in Homogeneous Distributed Systems", *IEEE Transactions on Software Engineering* vol 12(5), May 1986.

[5] J. Kramer, O. Kremien. "Methodical Analysis of Adaptive Load Sharing Algorithms", *IEEE Transactions on Parallel and Distributed Systems*, vol 3(6), November 1992.

[6] M. Nuttall. "Survey of Systems Providing Process or Object Migration", *Research Report DoC 94/10*, Imperial College London, Department of Computing, May 1994.

[7] R. J. Fowler. "The Complexity of Using Forwarding Addresses for Decentralized Object Finding", *Proceedings of 5th ACM Symposium on Principles of Distributed Computation*, August 1986.

[8] S. Bijnens, W. Joosen, P. Verbaeten. "Object Parallelism in XENOOPS", *Proceedings of the Conference on Object-Oriented Numerics*, April 1993.

[9] T. Stickdorn, U. Haack, L. Büttner, W. Schröder-Preikschat. "Dual, Mobile and Reactive Objects - Operating System Support for Load Balancing Parallel VR Applications", *Proceedings of 13th Annual Hawaii International Conference on System Sciences*, January 1997.

[10] E. Jul. "The Emerald Model of Dynamic Object Placement and Mobility", *Proceedings of ECOOP'92 Workshop on Dynamic Object Placement and Load Balancing in Parallel and Distributed Systems.*

[11] J. Arabe, A. Beguelin, B. Lowekamp, E. Seligman, M. Starkey, P. Stephan. "Dome: Parallel programming in a heterogeneous multi-user environment", *Proceedings Supercomputing '95.*

[12] P. Amaral, C. Jaquemot, P. Jensen, R. Lea, A. Mirowski. "Transparent object migration in COOL2", *Proceedings of ECOOP'94 Workshop on Dynamic Object Placement and Load Balancing in Parallel and Distributed Systems.*

Lastverwaltung in Workstation-Clustern auf Basis Mobiler Agenten

Claus Grewe, Wolfgang Obelöer und Holger Pals
Medizinische Universität zu Lübeck, Institut für Technische Informatik
Ratzeburger Allee 160, D-23538 Lübeck
Tel.: 0451/500 3690, Fax: 0451/500 3687
Email: {grewe, obeloeer, pals}@iti.mu-luebeck.de

Schlüsselwörter: Workstation-Cluster, Lastverwaltung, Mobile Agenten, Java

Kurzfassung: Eine Grundvoraussetzung für eine gute Leistungsausbeute in Workstation-Clustern ist die effiziente Verteilung von Ressourcenanforderungen. Ein geeignetes Mittel zur Auflösung dynamischer Lastungleichgewichte stellt die Verschiebung (Migration) von Überlasten dar. Allerdings besitzen Clustersysteme oft eine heterogene Struktur, d.h. beispielsweise unterschiedliche Prozessorarchitekturen und/oder Betriebssysteme. Diese Eigenschaft erschwert die Migration ganz erheblich. Hier ist die Betrachtung von Mobilen Agenten sehr interessant, da diese auch in heterogenen Systemen eine Migration unterstützen. In diesem Beitrag werden neben der allgemeinen Betrachtung von Mobilen Agentensystemen insbesondere deren Einsatz für lastbalancierte parallele Berechnungen betrachtet. Als Grundlage für die Untersuchungen dient das FLASH-System (**Fl**exible **A**gent **S**ystem for **H**eterogeneous Cluster), welches ein existierendes Agentensystem mit einer flexiblen Lastverwaltung kombiniert.

1 Einleitung

Workstation-Cluster gewinnen durch ihr gutes Preis-/Leistungsverhältnis gegenüber dedizierten Parallelrechnern zunehmend an Bedeutung. Als Rechenknoten werden in diesen Clustern Standard-PCs oder -Workstations verwendet. Die Kommunikation erfolgt über ein Verbindungsnetzwerk, dessen Geschwindigkeit entscheidenden Einfluß auf die Leistungsfähigkeit des Gesamtsystems hat. Die Programmierung solcher Systeme findet bevorzugt unter Einsatz spezieller Programmpakete wie z.B. PVM [Gei93] oder MPI [MPI95] statt, in denen Prozesse oder Threads miteinander mittels Nachrichtenaustausch (Message-Passing) kommunizieren. In einem solchen Umfeld erweist sich eine gute Leistungsausbeute als eine schwierige Aufgabe, denn für eine effiziente Systemauslastung, welche z.B. auf die Minimierung der Applikationsantwortzeit abzielt, ist häufig noch der Programmierer verantwortlich. Neben der Verteilung der Ressourcenanforderungen der eigenen Software sind weitere Effekte wie beispielsweise datenabhängige Laufzeiten oder variable Hintergrundlasten zu berücksichtigen. Einen Ausweg bietet der Einsatz automatisierter Lastverwaltungen (kurz LV). Diese können sowohl als Basisdienst in oder oberhalb des Betriebssystems als auch innerhalb der Anwendung integriert sein. Sie befreien den Programmierer von der Aufgabe des Lastausgleichs und nutzen z.B. das Mittel der Verschiebung (Migration), um entweder Prozesse oder Datenbereiche von über- auf unterlastete Knoten zu verschieben. Allerdings bestehen Clustersysteme im Gegensatz zu dedizierten Parallelrechnern häufig aus heterogenen Komponenten, d.h. die einzelnen Knoten weisen i.allg. unterschiedliche Prozessorarchitekturen und/oder Betriebssysteme auf. Diese Eigenschaft erschwert die Migration von Prozessen ganz erheblich, so daß bisher nur wenige Ansätze zur heterogenen Prozeßverschiebung bekannt sind [MDP96]. Sie basieren z.B. auf einer Neukompilation des Sourcecodes während der Migration oder speziellen Übersetzungstechniken [Ple96].

Vor diesem Hintergrund erweist sich die Betrachtung sogenannter Mobiler Agenten als sehr interessant, denn diese besitzen per Definition die Fähigkeit, sich autonom von einem Ort zu einem anderen bewegen zu können. Dabei führen sie ihre Berechnungsmethoden und Daten mit sich und können auf den besuchten Rechnerknoten die unterschiedlichsten Arbeiten ausführen. Aus diesem

Grund eignen sie sich auch für die Realisierung verteilter Anwendungen. Für den Einsatz Mobiler Agenten sind allerdings spezielle Umgebungen (Agentensysteme) notwendig, welche den Agenten die notwendige Infrastruktur zur Interaktion (Kommunikation) und Mobilität (Migration) bieten. Im Laufe der letzten Jahre sind einige recht unterschiedliche Ansätze solcher Agentensysteme konzipiert und realisiert worden.

In diesem Bericht wird die Umgebung **FLASH** (**Fl**exible **A**gent **S**ystem for **H**eterogeneous Cluster) vorgestellt, die sich auf ein vorhandenes Agentensystem (Voyager in Verbindung mit Java) stützt und dieses um einen Systemdienst zur Lastverwaltung erweitert. Die entstandene Umgebung eignet sich besonders für Untersuchungen im Umfeld agentenbasierter, verteilter Anwendungen, da sie es ermöglicht, Aspekte der LV mit in die Migrationsentscheidungen der Agenten einfließen zu lassen. Der Bericht nennt weiterhin erste Erfahrungen, die mit dem System gewonnen werden konnten.

In Kapitel 2 wird zunächst ganz allgemein der Agentenbegriff eingeführt und anschließend die Subklasse der Mobilen Agenten abgeleitet sowie einige Realisierungen von Agentensystemen vorgestellt. Kapitel 3 beschreibt die Struktur und Realisierung der FLASH-Umgebung und die eingebundene flexible LV-Umgebung. Mit der FLASH-Umgebung wurden erste Leistungsuntersuchungen ausgeführt, worauf das Kapitel 4 detailliert eingeht. Der Bericht endet mit einer Zusammenfassung und einem Ausblick auf zukünftige Tätigkeiten.

2 Agenten

Agenten, deren Konzepte ursprünglich aus dem Bereich der Künstlichen Intelligenz (KI) stammen, finden aktuell ein großes Interesse bei einem breiteren Publikum. Dabei kann festgestellt werden, daß es unterschiedliche Definitionen des Agentenbegriffes gibt. Aus diesem Grund wird nachfolgend zunächst eine Begriffsdefinition vorgestellt, welche als Basis für diesen Bericht dient. Danach folgt eine Betrachtung der interessanten Subklasse der Mobilen Agenten sowie ein Überblick typischer Aufgabenfelder für ihren Einsatz. Zusätzlich werden die Vorteile Mobiler Agenten und ihre Probleme diskutiert sowie einige bestehende Systeme beschrieben.

2.1 Was ist ein Agent?

Infolge der unterschiedlichen Ziele bestehender Agentenansätze hat sich bisher keine allgemein anerkannte Definition des Agentenbegriffes durchgesetzt. Woolridge und Jennings [WoJ95] schlagen deshalb eine gestaffelte Definition vor, wobei die erste allgemeingültiger und weicher (weak) ist, die zweite hingegen strenger (strong) und eher in Richtung KI weist. Die weichere Definition beschreibt Agenten als eine Hardware oder (noch häufiger) ein software-basierendes Computersystem, welches die nachfolgenden Eigenschaften besitzt:

- **Autonomie:** Agenten arbeiten ohne direkte Intervention von Menschen oder anderen Agenten und besitzen eigenständige Kontrolle über ihre Aktionen sowie ihren internen Zustand.
- **Soziale Fähigkeiten:** Agenten können untereinander (und evtl. mit dem Benutzer) mittels einer speziellen Agentenkommunikationssprache (z.B. KQML [FFK94]) interagieren.
- **Reaktivität:** Agenten nehmen ihre Umgebung wahr (z. B. die physikalische Welt oder andere Agenten in ihrer Umgebung) und reagieren rechtzeitig auf Änderungen, die in ihr auftreten.
- **Proaktivität:** Agenten reagieren nicht nur auf ihre Umgebung, sondern zeigen ein zielgerichtetes Verhalten durch Aufnahme von Initiative.

Die strengeren Begriffsdefinitionen, deren Anforderungen über die obigen hinausgehen, finden sich hauptsächlich im Bereich der KI. Hier werden z.B. mentale Fähigkeiten wie Wissen (knowledge), Überzeugungen (belief) und Intentionen (kurz BDI) oder emotionale Fähigkeiten gefordert bzw. berücksichtigt. Da die in diesem Bericht nachfolgend vorgestellten Agenten diese Fähigkeiten nicht besitzen, werden sie nicht weiter betrachtet.

Weitere Eigenschaften, die im Zusammenhang mit Agenten genannt werden, sind z.B. Mobilität, Rationalität oder Wahrhaftigkeit, d.h. ein Agent gibt wissentlich keine falschen Informationen weiter.

Von besonderer Bedeutung erweist sich dabei die Fähigkeit der **Mobilität**, welche großes Interesse in zahlreichen Bereichen der Computerwissenschaften hervorgerufen und Agenten, die über diese Fähigkeit verfügen, einen eigenen Namen gegeben hat - **Mobile Agenten**.

2.2 Mobile Agenten

Mobile Agenten genügen i.allg. der weicheren Begriffsdefinition von Woolridge und Jennings und besitzen darüber hinaus die Fähigkeit, sich autonom von einem Ort zu einem anderen zu bewegen und bei ihrer Migration durch das Netzwerk die Berechnungsdaten, den Programmcode und/oder den Ausführungszustand mit sich zu führen (vgl. Objektmigration). Ihre Proaktivität unterscheidet sie dabei wesentlich von anderen „mobilen" Objekten wie z.B. Java-Applets, die von einem Benutzer erst per Browser geladen werden müssen (Code on Demand). Das Einsatzgebiet für Mobile Agenten ist sehr groß, umfaßt jedoch typischerweise die Verarbeitung verteilter Daten sowie die Inanspruchnahme oder das Annoncieren verteilter Dienste, wie z.B.

- die Suche, Filterung und Verdichtung verteilter Informationen,
- die asynchrone Überwachung (Monitoring) bestimmter Ereignisse,
- die Bereitstellung sogenannter persönlicher Assistenten**,**
- den elektronischen Kommerz sowie
- die verteilte Anwendungsbearbeitung (Parallelverarbeitung).

Der große Vorteil Mobiler Agenten besteht darin, daß sie die unterschiedlichsten Netzwerkanwendungen gleichmäßig gut unterstützen. Da sie ihre eigenen Berechnungsmethoden mit sich führen, erweitern sie die Fähigkeiten der teilnehmenden Rechnerknoten enorm [BDF98]. Im Falle einer Alternativimplementierung z.B. mittels Remote Procedure Calls (RPC) besitzen die Rechenknoten nur einen eingeschränkten, vordefinierten Satz an Funktionen. Diese müssen zudem im voraus auf allen teilnehmenden Knoten installiert werden. Ähnliches gilt für die Kommunikationsprotokolle, die ebenfalls im voraus festgelegt werden müssen. Mobile Agenten hingegen können zur Laufzeit untereinander ihre Kommunikationsprotokolle dynamisch anpassen bzw. ändern. Ebenso können Mobile Agenten sich an eine dynamische Umgebung adaptieren, da sie ihre Umgebung wahrnehmen und auf Veränderungen autonom reagieren können.

Zu den weiteren möglichen Vorteilen der Mobilen Agenten zählen laut [HCK95] deren Robustheit gegenüber Netzausfällen, die gute Skalierbarkeit verteilter Anwendungen, der geringere Aufwand für sichere Transaktionen sowie die Unterstützung abgekoppelter Operationen. Letzteres ist insbesondere für den Bereich des sogenannten Mobile Computing von Belang. Portable Geräte wie Notebooks oder Laptops sind nicht ständig am Netz angeschlossen (z.B. infolge gestörter Übertragungsmedien oder Energiesparaspekten) und verfügen i.allg. über leistungsschwächere Prozessoren, geringere Bandbreiten zum Netzwerk sowie wenig Speicherkapazitäten [BSH97]. Der Einsatz Mobiler Agenten ermöglicht es hier, komplexe Aufgaben auszulagern und Ergebnisse asynchron entgegenzunehmen.

Weiterhin gestatten Mobile Agenten wesentlich anschaulichere Modellierungen realer, aktiver Objekte. Beispielsweise kann das Verhalten eines Fischschwarms, welches zur Klasse der individuenbasierenden Simulationen zählt [LRS94], durch Mobile Agenten simuliert werden, indem jeder einzelne Agent genau einen Fisch des Schwarms repräsentiert. Bei einer konventionellen Implementierung z.B. mittels Message-Passing werden die Fische als ein Datum im Speicher abgelegt und von einer übergeordneten Komponente verwaltet, die auch das Verhalten der Fische simuliert. Ein letzter und bedeutender Punkt, der den Einsatz Mobiler Agenten motiviert, ist die u.U. massive Reduktion von Kommunikationskosten [Whi96]. Da Mobile Agenten sich auf Datenbestände zubewegen können, wird vermieden, daß z.B. für eine Suche erst alle Daten über ein Netzwerk zur Suchroutine transportiert werden müssen. Mobile Agenten sind in den letzten Jahren Gegenstand breiterer Forschungsanstrengungen geworden, weshalb noch eine Reihe nicht vollständig gelöster Fragestellungen existieren. Im folgenden Abschnitt werden einige der Probleme betrachtet.

2.3 Probleme Mobiler Agenten

Bevor ein Agent überhaupt durch ein Netzwerk bestehend aus heterogenen Rechenknoten (PCs bis Supercomputer) und u.U. verschieden physikalischen Netzwerken migrieren kann, sind zunächst eine Vielzahl technischer Probleme zu lösen. In Kapitel 1 wurde bereits auf die Schwierigkeiten hingewiesen, die bei der Migration von Prozessen in einem heterogenen Umfeld entstehen. Aus diesem Grund werden sogenannte **Agentensysteme** (Mobile Agent System) eingesetzt, welche die Basisinfrastruktur für Mobile Agenten bereitstellen (siehe nächsten Abschnitt) und eine Abstraktion der zugrunde liegenden Netzwerke und Betriebssysteme gestatten. Ihr Konzept weist i.allg. ein Kommunikationsmodul, eine Ausführungseinheit sowie einen gemeinsamen Datenbereich auf, über den die Agenten lokal miteinander interagieren [BDF98]. Auf der untersten Ebene wird von den Agentenumgebungen eine Abbildung eines logischen auf das physikalische (u.U. heterogene) Netzwerk vorgenommen.

Mobile Agenten werden oft in einer Programmiersprache entwickelt. Zur Aufhebung der Heterogenitätsproblematik werden sie bevorzugt entweder in maschinenunabhängige Zwischenformate transformiert oder direkt interpretiert. Ein Beispiel für ein Zwischenformat ist der Java Bytecode [GoM96]. Java ist eine objekt-orientierte Programmiersprache, die im Bereich der Agentenumgebungen inzwischen die de-facto-Führung [MCR96] besitzt, und sich im besonderen Maße für WWW-Anwendungen eignet. Des weiteren ist sie architekturneutral und mit kleinen Einschränkungen (an denen gearbeitet wird) sicher. Außerdem wird Java durch zahlreiche Firmen unterstützt.

Trotz der Vorherrschaft von Java existieren weitere Sprachen, welche von Agentenumgebungen genutzt werden. Hierzu zählen z.B. Telescript, Tcl/Tk oder Python. Alle drei genannten Beispiele werden interpretiert. Während Telescript von vornherein zur Realisierung Mobiler Agenten entwickelt wurde [Gen95], sind die meisten heutzutage eingesetzten Sprachen Weiterentwicklungen klassischer Ansätze. Aus diesem Grund stellen sie keine fertige Lösung, sondern eine Basis zur Realisierung Mobiler Agenten dar. Die Problemstellungen, die dabei innerhalb von Agentenumgebungen gelöst werden müssen, betreffen folgende Punkte [MGW97]:

Namensvergabe: Jeder Agent braucht einen Namen zur Unterscheidung. Dabei tritt die Schwierigkeit auf, daß ein Agent mehrere Namen besitzen kann oder daß ein Namen zu mehreren Agenten gehört.

Adressierung und Lokalisation: Die Adresse spezifiziert, wo sich im System ein Mobiler Agent aktuell befindet. Damit ein Agent einen anderen ansprechen kann, muß er ihn zunächst im System lokalisieren.

Kontrolle: Einige Sprachen weisen keine Mittel zur Terminierung, Suspendierung oder Fortsetzung von Agenten auf. Diese Möglichkeiten sind allerdings notwendig, um einen Agenten auf eine Migration vorzubereiten bzw. ihn nach der Migration zu reaktivieren.

Export des Ausführungszustandes: Infolge der Komplexität bieten zahlreiche Sprachen keine Möglichkeit, den Ausführungsstatus zu sichern. Somit muß zusätzlich ein Endlicher Automat programmiert werden, welcher den aktuellen Ausführungszustand verwaltet.

Sicherheit: Mobile Agenten unterliegen strengsten Sicherheitskriterien, da sowohl sie als auch die aufgesuchten Rechenknoten sicherheitskritische Daten beinhalten können. Aus diesem Grund müssen sowohl die Agenten als auch die Knoten vor- sowie untereinander geschützt werden.

Alle hier genannten Probleme sind Gegenstand der Forschung, wobei insbesondere die Sicherheit der Agenten von besonderen Interesse ist [BHR97]. Von ihr hängt maßgeblich die Akzeptanz Mobiler Agenten z.B. im Bereich des elektronischen Kommerzes ab. Für den Einsatz Mobiler Agenten zur Parallelverarbeitung fallen die Sicherheitsanforderungen häufig wesentlich geringer aus. Im Rahmen der erwähnten Forschungsaktivitäten sind bereits einige Agentenumgebungen entwickelt worden. Im folgenden Abschnitt werden zwei von ihnen exemplarisch vorgestellt.

2.4 Agentenumgebungen

Derzeitig stehen eine Reihe von Agentensysteme frei zur Verfügung. Hierzu zählen z.B. die Aglets Workbench [LaO97] (IBM Research Center Japan), Agent TCL [Kot97] (Dartmouth College), Mole [BHR97] (Universität Stuttgart) oder TACOMA [JRS95] (Universitäten Tromsø und Cornell). In diesem Abschnitt werden stellvertretend die beiden Agentenumgebungen **Mole** und **TACOMA** näher beschrieben. Sie wurden deshalb gewählt, da sie inzwischen einen ausgereiften Stand erreicht haben und sich vom Grundkonzept her unterscheiden und somit einen Einblick in unterschiedliche Ansätze erlauben. Ein sehr mächtiges Werkzeug stellt darüber hinaus die Voyager-Umgebung [Obj97] von ObjectSpace, Inc. dar. Voyager dient u.a. als Basis für das entwickelte FLASH-System, auf das im nachfolgenden Kapitel noch detailliert eingegangen wird.

Die in Java realisierte Agentenumgebung **Mole** ist ein laufendes Forschungsprojekt an der Universität Stuttgart [BHR97]. Die Umgebung dient der Grundlagenforschung in den Bereichen Sicherheits-, Kommunikations- und Migrationstechniken. Mole nutzt (wie viele weitere Umgebungen) einen sog. „sanften Migrationsansatz“ auf der Basis des Remote Method Invocation Package (RMI) [Sun97]. Während der Migration werden nur der Code und die Daten eines Agenten, nicht aber sein Ausführungszustand in eine systemunabhängige Darstellung transformiert (Serialisierung), zum Zielrechner transportiert und dort innerhalb eines neuen Threads wieder hergestellt. Den Ausführungszustand muß der Programmierer innerhalb des Datenbereiches selbst pflegen. Anhand des kodierten Ausführungszustandes entscheidet anschließend eine Startmethode, an welcher Stelle der Agent seine Ausführung fortsetzt.

Mole kennt zwei Typen von Agenten, die Anwendungs- und die Serviceagenten. Letztere sind immobil und müssen vom Administrator gestartet werden, da sie als einzige die Möglichkeit besitzen, direkt auf Systemfunktionen und -ressourcen zuzugreifen. Zur Wahrung der Sicherheit setzt Mole hier auf das sog. Sandkastenprinzip (Sandbox Security Model), in dem alle potentiell gefährlichen Prozeduraufrufe zunächst von speziellen Sicherheitskomponenten untersucht und freigegeben werden müssen. Damit Agenten miteinander kommunizieren können, bedarf es einer eindeutigen Namensvergabe. Mole adressiert Agenten global eindeutig durch ihren Namen und einen DNS-Namen des Knotens, auf dem sie residieren. Die zur Kommunikation angewandte Methodik ist abhängig von den jeweiligen Kommunikationspartnern. Mobile Agenten kommunizieren mit Serviceagenten mittels Java-RPC, Agenten untereinander per Message-Passing und innerhalb von Agentengruppen wird eine anonyme Kommunikation unterstützt.

Im Gegensatz zur Mole-Umgebung, welche sich auf die Java-Laufzeitumgebung stützt, verfolgt die Agentenumgebung **TACOMA** (**T**romsø **A**nd **CO**rnell **M**oving **A**gents) einen systemnäheren Ansatz. Das TACOMA-Projekt beschäftigt sich mit Sicherungsmaßnahmen für den elektronischen Kommerz, Scheduling- und Fehlertoleranztechniken sowie der Fragestellung, inwieweit Mobile Agenten und Betriebssysteme voneinander profitieren können [JRS95]. Die Abstraktionen und Mechanismen der Mobilen Agenten lehnen sich deshalb stark an klassische Verwaltungsstrukturen an. Jeder Agent besitzt zur Aufbewahrung von Daten eine Mappe (briefcase) mit mehreren Ordnern (folders), welche jeweils eine Liste von uninterpretierten Bitsequenzen enthalten. Zusätzlich existieren pro Knoten statische Mappen (file cabinets), in denen die Mobilen Agenten z.B. Nachrichten asynchron hinterlegen können. Interaktionen mit anderen Benutzer- und Serviceagenten üben die Mobilen Agenten mittels der meet-Operation aus. Hierbei geben sie an, mit wem sie kommunizieren möchten. Zusätzlich übergeben sie ihre eigene Mappe, in der die eigentliche Dienstbeschreibung bzw. Nachricht abgelegt ist. Beispielsweise migriert ein Benutzeragent, indem er auf einen speziellen Systemagenten (rexec agent) wartet und in seiner Mappe zwei Ordner aufweist, die besagen, wohin er migrieren möchte (HOST folder) und wer auf dem Zielknoten sein Ansprechpartner ist (CONTACT folder). Der Ansprechpartner kann z.B. ein Agent, eine Shell oder ein Compiler sein. Ein Compiler erwartet in der Mappe zusätzlich einen Ordner, der den Sourcecode des Agenten

enthält (CODE folder). Dieser Ansatz ermöglicht es, daß TACOMA zur Zeit die fünf Programmier- bzw. Skriptsprachen Tcl/TK, C, Perl, Python und Scheme unterstützt.

Die hier vorgestellten Agentensysteme eignen sich in unterschiedlichem Maße für die Realisierung einer Umgebung für parallele Anwendungen auf der Basis Mobiler Agenten. Für die Realisierung des FLASH-Systems fiel die Wahl auf das Voyager-System, worauf das folgende Kapitel eingeht.

3 Mobiles Agentensystem FLASH

Es existieren - wie beschrieben - eine Reihe unterschiedlicher Konzepte und Realisierungen im Bereich der Agentensysteme. Diese wurden bisher zumeist genutzt, um z.B. Aufgaben aus den Bereichen der Informationssuche zu lösen oder die Schnittstelle zwischen dem Benutzer und der Maschine zu verbessern. Im **FLASH**-Projekt (**Fl**exible **A**gent **S**ystem for **H**eterogeneous Cluster) wird angestrebt, das Konzept der Mobilen Agenten im besonderen zur Realisierung verteilter Berechnungsaufgaben (z.B. komplexe Simulationen, mathematische Lösungsverfahren, etc.) anzuwenden bzw. zu untersuchen. Bei den anvisierten Zielsystemen handelt es sich um heterogene Cluster, deren Knoten sich in Leistung, Speicher und Netzwerkanbindung unterscheiden. In diesem Umfeld spielt u.a. die LV eine wichtige Rolle. Die Vorteile Mobiler Agenten kommen dann besonders zur Geltung, wenn sie effiziente Migrationsentscheidungen treffen, indem sie die eigenen Ressourcenanforderungen sowie die aktuelle und - wenn möglich - zukünftige Systembelastung mit einbeziehen. In diesem Zusammenhang sind u.a. eine Reihe von Fragen bzgl. der eingesetzten LV-Strategien zu klären. Die Untersuchungen hierzu (z.B. zur Verteilungsstrategie von Lastinformationen) finden in einer flexiblen, konfigurierbaren und leicht zu handhabenden Umgebung (FLASH) statt. Die Umgebung besteht aus zwei wesentlichen Teilkomponenten. Zum einen baut sie auf ein vorhandenes Agentensystem (Voyager, siehe Abschnitt 3.2) auf und erweitert zum anderen dieses um verschiedene Systemdienste. Nachfolgend wird die Struktur sowie die erste Realisierung der FLASH-Umgebung beschrieben.

3.1 Struktur

Das FLASH-System besitzt in der ersten Umsetzung vier grundlegende Komponenten. Drei sind die innerhalb des jeweilig zugrunde gelegten Agentensystems realisierten System-, Knoten- und Benutzeragenten, die hierarchisch angeordnet sind und miteinander interagieren. Die vierte Grundkomponente sind die Schnittstellen für den Zugriff auf verschiedene Systemdienste. Jede der vier Grundkomponenten erfüllt ihre eigene, abgegrenzte Aufgabenstellung:

- Die Aufgabe der **Benutzeragenten (BA)** besteht in der Ausführung der eigentlichen Anwendung. Hierzu migrieren sie durch das System, z.B. auf der Suche nach freien Ressourcen. Die von ihnen getroffenen Migrationsentscheidungen sind u.a. abhängig von inneren Zuständen, externen Ereignissen (Interaktion) oder vom lokalen Knotenagenten. Die Benutzeragenten sind die einzigen Komponenten im System, deren Funktionalität vom Anwender explizit programmiert werden muß. Die weiteren Komponenten besitzen eine i.allg. von der Anwendung unabhängige Aufgabenstellung.
- Die **Knotenagenten (KA)** sind auf jedem teilnehmenden Knoten angesiedelt und bieten den Benutzeragenten die Anbindung an die Systemdienstschnittstellen sowie das Wissen darüber, welche weiteren Benutzeragenten sich gerade auf dem lokalen Knoten befinden. Außerdem können die Knotenagenten je nach gewählter LV-Strategie bei Migrationsentscheidungen behilflich sein. Knotenagenten sind nicht mobil, da sie den Benutzeragenten auf jedem besuchten Knoten als fester Ansprechpartner dienen. Allerdings können Knotenagenten Mobile Agenten (z.B. Kuriere) zur Erfüllung bestimmter Aufgaben kreieren.
- An oberster Stelle befinden sich die stationären **Systemagenten (SA)**, die z.B. das Wissen darüber beinhalten, welche Knoten am Agentensystem teilnehmen. Sie kommunizieren mit den Knotenagenten, falls z.B. Veränderungen am Agentensystem vorliegen. Die Aufgabe der

Systemverwaltung kann sowohl verteilt als auch zentral von einem einzelnen Systemagenten übernommen werden (siehe Abbildung 1).

- Die Schnittstellen zu den verschiedenen **Systemdiensten** gestatten es, eine Reihe von Beschränkungen vorhandener Agentensysteme zu überwinden. So verbieten i.allg. die für Agentensysteme geltenden scharfen Sicherheitskriterien, daß Mobile Agenten Zugriffe auf systemrelevante Daten und/oder Funktionalitäten tätigen. Hierdurch wird aber auch verhindert, daß die Agenten beispielsweise aktuelle Lastzustände beim System erfragen oder alternative Kommunikationsmedien (z.B. schnelle Kommunikation der BA) nutzen können. In der aktuellen Realisierung von FLASH ist deshalb eine Schnittstelle zu einer verteilten LV-Umgebung vorhanden, die an sich ein eigenständiges Werkzeug darstellt und maschinenspezifisch implementiert ist. Über die Systemdienstschnittstelle werden u.a. systemnah ermittelte Lastindizes ausgetauscht, welche die Agenten für Migrationsentscheidungen nutzen können. Die LV-Umgebung kann flexibel für unterschiedliche Anforderungen konfiguriert werden (siehe Abschnitt 3.3).

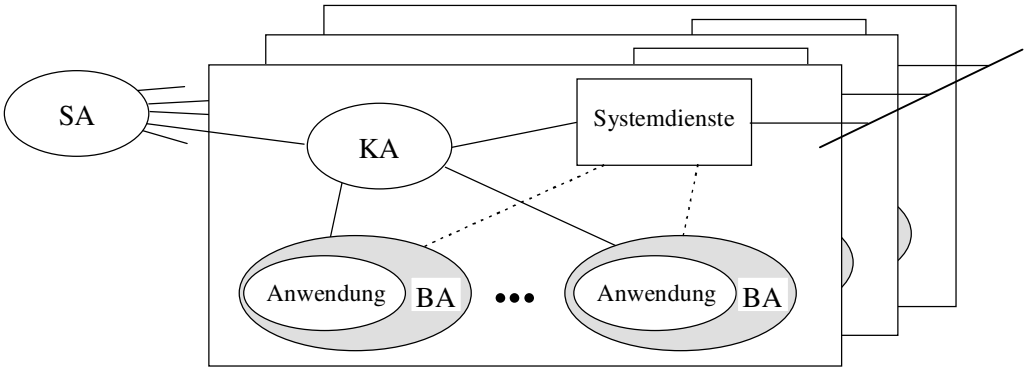


Abbildung 1: FLASH-Struktur (Flexible Agent System for Heterogeneous Cluster)

Die vorgestellte Struktur läßt sich mit verschiedenen Agentensystemen realisieren. Aktuell wird das Agentensystem Voyager zur Realisierung genutzt, welches Thema des folgenden Abschnittes ist.

3.2 Umsetzung der Agenten

Die Realisierung der drei Agententypen fand unter Verwendung von Java und dem Programmpaket Voyager [Obj97] von ObjectSpace, Inc. statt. Voyager ist frei verfügbar, vollständig in Java realisiert und weist eine Reihe von außergewöhnlichen und innovativen Eigenschaften auf [KiZ97]. Voyager wurde verwendet, da es zum Zeitpunkt der Realisierung des FLASH-Systems ausgereift erschien und eine gute Dokumentation aufwies. Das wesentliche Schlüsselelement Voyagers sind seine mobilen Objekte. Diese können frei migrieren, die Methoden anderer, auch entfernter Objekte aufrufen, einen eigenständigen Lebenszyklus besitzen sowie untereinander synchron oder asynchron kommunizieren. Zugriffe auf entfernte Methoden finden in der gleichen Art und Weise wie lokale Zugriffe statt. Ein aus der Basisklasse der mobilen Objekte abgeleitetes Objekt ist der Mobile Agent, der zusätzlich über die Fähigkeit der Autonomie verfügt. Alle notwendigen Referenzen und Abhängigkeiten werden von Voyager gepflegt, so daß der Programmierer sich nicht darum kümmern muß, wo sich ein Agent aktuell im System befindet (Ortstransparenz). Voyager bietet die gesamte Infrastruktur zur Verwaltung und Kommunikation der mobilen Objekte über sog. virtuelle Referenzen, die durch eine Neukompilation mittels des Virtual Code Compilers („vcc“) erzeugt werden.

Aufbauend auf diese Eigenschaften wurden für FLASH verschiedene Agentenobjekte realisiert. Eine hierarchische Infrastruktur aus System- und Knotenagenten unterstützt den Auf- und Abbau von Agenten und die Kommunikation im System bzw. mit der Außenwelt. Die Benutzeragenten bieten u.a. die Möglichkeit, Kommunikation sowohl über blockierende und nichtblockierende Remote Procedure Calls (RPC) als auch über PVM abzuwickeln, LV-Algorithmen für die Migrations-

entscheidung auszuführen und über eine festgelegte Schnittstelle eine graphische Kontrolloberfläche (siehe Abbildung 4) anzusprechen. Insbesondere ermöglichen sie durch Vererbung die leichte Einbringung von benutzerspezifischem Verhalten, d.h. sie stellen alle Grundfunktionalitäten eines Mobilen Agenten zur Verfügung.

Wie bereits beschrieben, erlauben Agentensysteme i.allg. aus Sicherheitsgründen keine Zugriffe auf systemspezifische Funktionalitäten. Im Falle von Java und Voyager bedeutet dieses, daß ausschließlich Lastindizes zur Verfügung stehen, die innerhalb der virtuellen Maschine erhoben werden (z.B. Anzahl der Threads und freier Speicher in der virtuellen Maschine). Aus diesem Grund besitzt das FLASH-System eine Schnittstelle an eine bestehende Lastverwaltung, welche als Systemdienst bereitsteht und Thema des nachfolgenden Abschnittes ist.

3.3 Systemdienst zur Lastverwaltung

Der Systemdienst zur Lastverwaltung erweitert den Wahrnehmungsbereich der Mobilen Agenten um den Aspekt der aktuellen und zukünftigen Systembelastung. Hierdurch wird es den Agenten ermöglicht, ihre Migrationsentscheidungen bzgl. vorhandener Ressourcen abzustimmen. Die Realisierung dieses Dienstes baut auf einer eigenständigen **Lastverwaltung** auf, die ursprünglich als flexible Werkzeugumgebung zur Unterstützung der Tätigkeiten Bewertung, Entwurf und Optimierung von LV-Algorithmen konzipiert und realisiert wurde [GrO97a, GrO97b].

Das Konzept der flexiblen LV basiert auf einer Aufteilung der Gesamtfunktionalität in unabhängige Teilkomponenten. Die vier wesentlichen Komponenten (in Abbildung 2 hinterlegt) sind: Sensor (Datenerfassung), Bewertung und Entscheidung (beinhalten den LV-Algorithmus) sowie Aktor (Verschiebekomponente). Hinzu kommen zwei Komponenten zur Steuerung des Systems und zur Datenverwaltung. Die genannte Aufteilung des Gesamtsystems in unabhängige Komponenten ermöglicht eine freie Konfiguration der Umgebung, so daß sie sich sowohl verteilten, gruppierten als auch zentralen Strukturen anpassen kann. Weiterhin müssen in einem System nicht auf jedem Knoten alle Komponenten vorhanden sein.

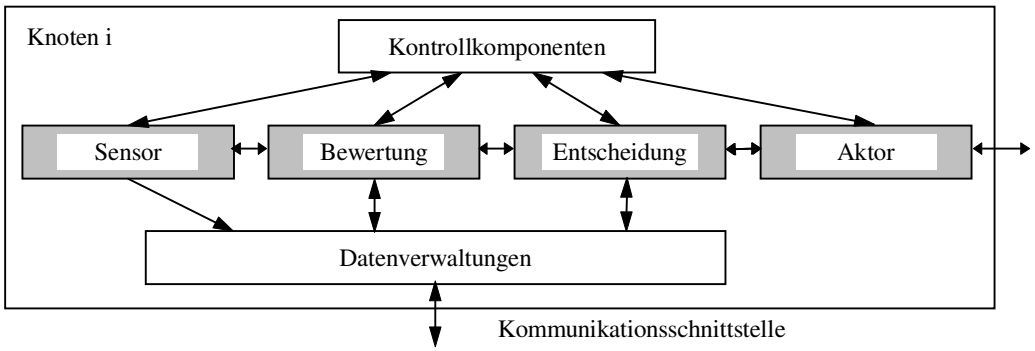


Abbildung 2: Lastverwaltungskomponenten auf einem Knoten

Das Werkzeug kann sowohl in dedizierten Parallelrechnern als auch in Workstation-Clustern eingesetzt werden. Die aktuelle Realisierung der flexiblen LV nutzt zur Kommunikation die Message-Passing Bibliothek PVM und wurde bisher für die Plattformen Sun-Workstations (Solaris), Intel-PCs (Linux) und SGI-Workstations (IRIX) portiert.

Insbesondere bietet das Werkzeug auf den verschiedenen Systemen ein breites Angebot verschiedener Lastindizes (darunter auch prognostizierende Parameter [GrO97b]) und eine Schnittstelle zur Anbindung an andere Systeme, wie z.B. Meß- und Auswertungswerkzeuge.

Die Konfigurierbarkeit des LV-Werkzeugs sowie die letzten beiden Punkte erleichterten eine Integration in das Agentensystem. Die LV-Umgebung wurde derart konfiguriert, daß sie den Bedürfnissen des Agentensystems genügt. Da Mobile Agenten die Fähigkeit der Migration besitzen,

kann auf den Aktor der Umgebung verzichtet werden. Die verbleibende LV-Umgebung ist in der Lage, Migrationsvorschläge zu erarbeiten und diese den Agenten mitzuteilen. Für den Fall, daß die Mobilen Agenten die Entscheidungen wann und wohin sie migrieren autonom treffen und ausführen, kann weiterhin auf die Bewertungs- und Entscheidungskomponenten verzichtet werden. Ein wichtiger Aspekt ist die Verbreitung erhobener Lastindizes. Diese Aufgabe wird aktuell durch die Datenverwaltungen innerhalb der LV-Umgebung übernommen, wobei hierfür in Zukunft der Einsatz Mobiler Agenten geplant ist.

Die Anbindung der LV an das Agentensystem erfolgt über die Kommunikationsschnittstelle (siehe Abbildung 2). Hierbei ist zu beachten, daß die beiden Umgebungen unterschiedlich implementiert wurden. Als Kommunikationsschnittstelle zwischen den in Java unter Nutzung von Voyager realisierten Agenten und der in C und PVM realisierten LV wurden mehrere Wege untersucht. Eine einfache Möglichkeit bietet die Einrichtung einer Socket-Verbindung. Wesentlich eleganter geschieht die Anbindung allerdings unter Verwendung des Programmpaketes JavaPVM [Thu97], welches die in Java integrierte C-Schnittstelle nutzt, um auf die normale PVM-Bibliothek zuzugreifen. Somit ließ sich der Aufwand für die Anbindung stark reduzieren, da die LV PVM ohnehin als Kommunikationsbibliothek einsetzt.

Mit dem vorgestellten FLASH-System wurden erste Messungen durchgeführt und Erfahrungen gesammelt, die im nächsten Kapitel präsentiert werden.

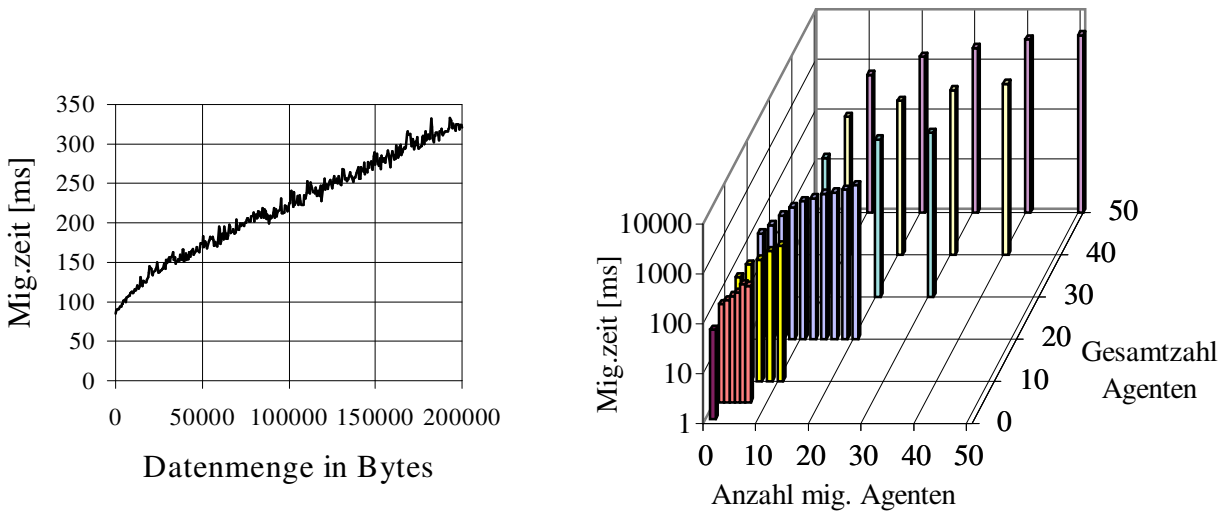
4 Messungen und Erfahrungen

Bei den ersten Messungen standen vor allem grundlegende Leistungsaspekte der realisierten FLASH-Umgebung im Vordergrund. Untersucht wurde u.a., welche Migrationszeiten die Mobilen Agenten in Abhängigkeit von ihrer Größe und der Anzahl gleichzeitig migrierender Agenten aufweisen. Weitere Messungen betrachteten die Migrationsentscheidungen bei einer realen Anwendung. Ausgangspunkt für die Messungen ist ein heterogener Cluster bestehend aus verschiedenen Sun-Workstations (SPARC 5 bis UltraSPARC 1, 75-200 MHz), einer SGI O2 Workstation (MIPS R5000, 180 MHz) sowie diversen PCs (Pentium, 90-133 MHz). Die Maschinen sind alle über Standard-Ethernet miteinander verbunden und besitzen unterschiedlich große Speicher- und Festplattenkapazitäten.

Zur Bestimmung der Migrationszeiten in Abhängigkeit der Datengröße wurde ein einfacher Agent realisiert, der zwischen zwei Knoten migriert und dabei ein Datenfeld unterschiedlicher Größe beinhaltet. Abbildung 3 a zeigt den Verlauf der gemittelten Migrationszeit bei ansonsten unbelastetem Netzwerk in Millisekunden. Deutlich zu sehen ist der fast lineare Anstieg der Migrationszeit. Die bestimmenden Faktoren sind hier vor allem die Zeit für die Serialisierung des Objektes sowie den Transport der Daten.

Die zweite Untersuchung beschäftigte sich mit der mittleren Migrationsdauer von Agenten (0 Byte zus. Datenmenge) bei mehreren gleichzeitig stattfindenden Migrationen. Dazu wurde für verschiedene, konstant gehaltene Agentenzahlen jeweils die Anzahl der migrierenden Agenten zwischen zwei Knoten sukzessive erhöht. Abbildung 3 b stellt die mittlere Migrationsdauer im logarithmischen Maßstab dar. Es zeigte sich, daß die Migrationsdauer nur im geringen Maße von der Anzahl der vorhandenen Agenten abhängt, dagegen sehr stark von der Anzahl der gleichzeitig ausgeführten Migrationen. So dauert im Mittel bei gleichzeitiger Migration von 10 Agenten eine Migration 543 ms bei 10 vorhandenen Agenten und 583 ms bei 50 vorhandenen. Für die Migrationen von 20, 30 und 50 gleichzeitig migrierenden Agenten wurden im Mittel bei 50 vorhandenen Agenten 1361, 2025 und 3673 ms pro Agent gemessen. Diese Zeiten liegen sowohl an der sequentiellen Bearbeitung der Agenten auf den beiden beteiligten Knoten als auch an der hohen Netzwerklast. Dabei dürfte die Situation, daß alle Agenten gleichzeitig von einem Knoten migrieren wohl eher zum Ausnahmefall gehören.

Ein Vergleich mit klassischen Migrationsansätzen auf Prozeßbasis zeigt, daß die gemessenen Werte nur von modernen und stark optimierten Ansätzen eingestellt bzw. unterboten werden. Beispielsweise benötigt das Betriebssystem Sprite 330 ms zur Migration eines 100 KByte großen Prozesses, Choices sogar nur 14 ms [Rou95]. Allerdings gilt es zu bedenken, daß die Migrationsmechanismen innerhalb der genannten Betriebssysteme angesiedelt sind und Heterogenität nicht unterstützen.



a) variable Datenfeldgröße **b)** variable Anzahl migrierender Agenten

Abbildung 3: Migrationszeiten

Die weiteren Untersuchungen beziehen sich auf die LV von verteilten Anwendungen auf der Basis Mobiler Agenten. Hierzu wurde als Beispielanwendung ein paralleler Raytracer[1] (siehe Abbildung 4) eingesetzt, der als Verbund unabhängiger Mobiler Agenten realisiert ist. Diese Anwendung wurde gewählt, da sich zeigte, daß das eingesetzte Ethernet ohnehin bereits einen Engpaß darstellt. Zusätzliche Kommunikationen aufgrund von Interaktionen durch die Benutzeragenten hätte die Messung stark verfälscht. Für die Zukunft sind der Einsatz eines schnelleren Netzwerkes und Untersuchungen mit abhängigen Agenten geplant. Die Untersuchungen gehen von folgenden Voraussetzungen aus: Auf einem heterogenen Cluster werden eine Anzahl von Mobilen Agenten gestartet, wobei jeder Agent ein gleich großes Teilstück einer komplexen Berechnungszene bearbeitet.

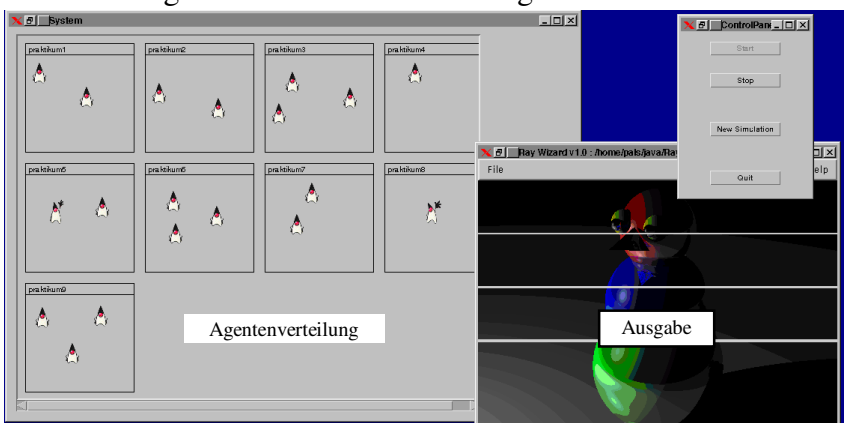


Abbildung 4: Oberfläche von FLASH mit Kontrollfenster (rechts) und Ausgabe der Anwendung (rechts)

Da es bei dieser ersten Untersuchung um das prinzipielle Verhalten des Systems ging, wurde zur LV ein einfacher Algorithmus eingesetzt. Er geht davon aus, daß alle Agenten eine gleichmäßige CPU-Belastung hervorrufen. Die Heterogenität des Clusters wird dadurch berücksichtigt, daß für jeden Knoten ein Leistungsparameter bestimmt wurde. Dieser Parameter spiegelt das Leistungsverhältnis jedes einzelnen Knotens im Vergleich zur schnellsten Maschine (eine Sun-UltraSparc-Workstation mit 200

[1] Die Implementierung des Raytracers in C ist im Vergleich zu Java rund 6 mal schneller.

MHz und 380 MB Speicher) wider. Der Wert wird dynamisch durch den Systemdienst LV angepaßt, sobald sich auf einer Maschine die Hintergrundlast verändert. Die auf einem Knoten vorhandene Last wird normiert (Division der Anzahl der lokalen Agenten durch den Leistungsparameter) und erlaubt so, eine Lastbewertung in heterogenen Umgebungen.

Zur Laufzeit beobachtet der Systemagent die Verteilung der Agenten im System und bestimmt, ob ein Ungleichgewicht vorliegt und welcher Knoten aktuell am stärksten belastet ist. Um für diese Messung das Verhalten in Grenzen steuern zu können, werden nur dem zugehörigen Knotenagenten Migrationsrechte eingeräumt. Der Knotenagent bestimmt geeignete Ziele für migrierwillige Benutzeragenten. Die Benutzeragenten wiederum bestimmen für sich autonom, ob sie migrieren möchten (die notwendigen Informationen, wie z.B. Anzahl Agenten und Lastdifferenz zu anderen Knoten, werden vom Knotenagenten zur Verfügung gestellt). Abbildung 5 zeigt die zeitliche und örtliche Verteilung der normierten Last, die durch 49 Agenten auf 20 Rechnern hervorgerufen wird. Dargestellt sind die ersten 250 Sekunden einer Messung. Alle Agenten wurden zum Startzeitpunkt auf der schnellsten Maschine plaziert. Wie deutlich zu sehen ist, migrieren nach und nach die Benutzeragenten zu anderen Rechnern im Cluster, so daß nach rund 70 Sekunden eine Gleichverteilung vorliegt. Die normierten Lastwerte müssen in diesem Zustand nicht übereinstimmen, da stets nur ganze Agenten migrieren können. Im dargestellten Zeitraum kommt es z.B. durch Terminierung von Agenten zu neuen Ungleichgewichten. Aus diesem Grund kann beispielsweise beobachtet werden, daß der Startrechner (R1) zwischenzeitlich eine sehr geringe Last aufweist, die aber zum Zeitpunkt 210 Sekunden wieder ausgeglichen ist. Trotz Verwendung des sehr einfachen LV-Algorithmus konnte ein Lastgleichgewicht entsprechend der Leistungen der einzelnen Maschinen hergestellt und erhalten werden.

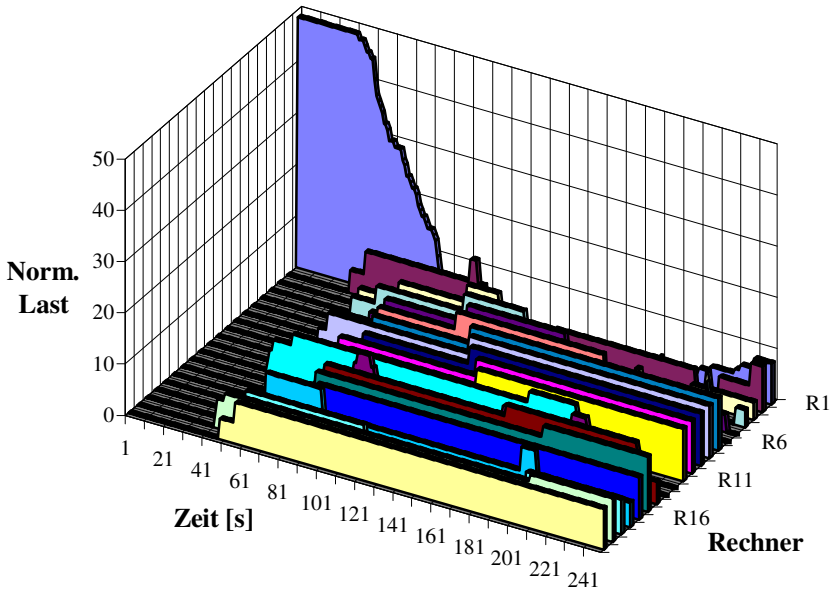


Abbildung 5: Verteilung der normierten Last

5 Zusammenfassung und Ausblick

Das vorgestellte FLASH-System bietet eine Grundlage zur Untersuchung von parallelen Berechnungen auf Basis Mobiler Agenten. Durch den verwendeten objekt-orientierten Ansatz läßt sich anwendungsspezifisches Verhalten trotz der recht komplexen Umgebung (bestehend aus Java, Voyager, C, PVM, JavaPVM) leicht einbringen. Dieses liegt daran, daß die Agenten bereits mit einer umfangreichen Basisfunktionalität ausgestattet sind. Prinzipiell zeigte sich, daß aufgrund der schnellen und auch leicht zu handhabenden Migration durch das Voyager-System ein hohes Maß an Dynamik und Flexibilität in Bezug auf mögliche Lastverwaltungsmaßnahmen zu beobachten war. Schon der Einsatz einer sehr einfachen Strategie zur Migrationsentscheidung in den Benutzeragenten zeigte balanciertes Verhalten. Allerdings führte die Verwendung von Java zu Leistungseinbußen, die jedoch durch eine Erweiterung des Systems reduzierbar sind. So können die Benutzeragenten z.B. auf jedem Knoten mittels vorhandener Systemdienste überprüfen, ob eine systemspezifische Berechnungsroutine im jeweiligen Maschinencode vorliegt und wahlweise die Aufgabe selbst lösen oder von der Routine berechnen lassen.

Zukünftige Arbeiten werden sich mit an die Dynamik angepaßten Lastverwaltungsalgorithmen beschäftigen und in diesem Zusammenhang die Nutzung Mobiler Agenten zur Verteilung von Lastinformationen untersuchen. Weitere wichtige Aspekte sind die Bereitstellung eines direkten Zugriffs auf Hochgeschwindigkeitsnetzwerke und die Einbeziehung von Fehlertoleranztechniken.

Literatur

[BDF98] L.F. Bic, M.B. Dillencourt, M. Fukuda: Mobile Network Agent. In: Encyclopedia of Electrical and Electronics Engeneering, John Willy & Sons, Inc., 1998

[BHR97] J. Baumann, F. Hohl, K. Rothermel, M. Straßer: Mole - Concepts of a Mobile Agent System. Technical Report TR-1997-15, IPVR, Fakultät Informatik, Universität Stuttgart, 1997

[BSR97] W. Böhmak, A. Schill, A. Held: Realisierung von Mobile Computing mittels Java RMI. In. Proc. 4. GI-ITG Fachtagung Arbeitsplatz-Rechensysteme (APS'97), 46-54, Koblenz 1997

[FFK94] T. Finin, R. Fritzson, D. McKay, R. McEntire: KQML As An Agent Communication Language. In Proc. 3th Intl. Conf. on Information and Knowledge Management (CIKM'94), 456-463, ACM Press, Gaithersburg 1994

[Gei93] A. Geist et.al.: PVM 3 User's Guide and Reference Manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, 1993

[Gen95] General Magic: Magic Cap Concepts. General Magic, Inc., Sunnyvale, California, 1995

[GoM96] J. Gosling, H. McGilton: The Java Language Environment. White Paper, Sun Microsystems, Inc., Palo Alto 1996

[GrO97a] C. Grewe, W. Obelöer: Flexible Lastverwaltung für Parallelrechner und Workstation-Cluster. In Proc. ARCS'97, Vorträge der Workshops im Rahmen der 14. ITG/GI Fachtagung, 255 - 265, Rostock 1997

[GrO97b] C. Grewe, W. Obelöer: Lastprognose für eine flexible Lastverwaltung in Workstation-Clustern. In Proc. 4. GI-ITG Fachtagung Arbeitsplatz-Rechensysteme (APS'97), 67-77, Koblenz 1997

[HCK95] C.G. Harrison, D.M. Chess, A. Kershenbaum: Mobile Agents: Are they a Good Idea? Research Report, IBM Research Division, T.J. Watson Center, 1995

[JRS95] D. Johansen, R. van Renesse, F.B. Schneider: Operation System Support for Mobile Agents. In. Proc. 5th Workshop on Hot Topics in Operation Systems (HOTOS-V). 42-45, IEEE Press, 1995

[KiZ97] J. Kiniry, D. Zimmerman: Special: A Hands-On Look at Java Mobile Agents. IEEE Internet Computing, Vol. 1, No. 4, 21-30, 1997

[Kot97] D. Kotz et al.: Agent TCL: Targeting the Needs of Mobile Computers. IEEE Internet Computing, Vol. 1, No. 4, 58-67, 1997

[LaO97] D.B. Lange, M. Oshima: Programming Mobile Agents in Java™ - With the Java Aglet API. IBM Research Center Japan, Tokio1997

[LRS94] H. Lorek, K. Rössig, M. Sonnenschein: Verteilte Simulation individuenbasierter Modelle. Proc. Umwelt Informatik, Hamburg 1994

[MCR96] D.S. Milojicic, M. Condict, F. Reynolds, D. Bolinger, P. Dale: Mobile Objects and Agents. In „Distributed Object Computing on the Internet" Advanced Topics Workshop, at the 2nd USENIX Conf. On Object Oriented Technologies ans systems (COOTS), Toronto 1996

[MDP96] D.S. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler, S. Zhou: Process Migration. The Open Group Research Institute Technical Report, 1996

[MGW97] D.S. Milojicic, S. Guday, R. Wheeler: Old Wine in New Bottles - Applying OS Process Migration Technology to Mobile Agents. The Open Group Research Institute, Technical Report, 1997

[MPI95] MPI Forum: MPI: A Message-Passing Interface Standard, Version 1.1, University of Tennessee, Knoxville, Tennessee 1995

[Obj97] ObjectSpace: ObjectSpace Voyager™ Core Package Version 1.0. Technical Overview, ObjectSpace, Inc., 1997

[Ple96] C. Pleier: Prozeßverlagerung in heterogenen Rechnernetzen basierend auf einer speziellen Übersetzungstechnik. Herbert Utz Verlag Wissenschaft, München 1996

[Rou95] E.T. Roush: The Freeze Free Algorithm for Process Migration. PhD Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1995

[Sun97] Sun Microsystems: Java Remote Method Invocation - Distributed Computing for Java. Java White Paper, Sun Microsystems, Inc., Palo Alto 1997

[Thu97] D.A. Thurman: JavaPVM. `http://www.isye.gatech.edu/chmsr/JavaPVM/`, 1997

[Whi96] J. White: Mobile Agents White Paper. Technical Report, General Magic, Inc., Sunnyvale 1996

[WoJ95] M.J. Woolridge, N.R. Jennings: Agent Theories, Architectures, and Languages: A Survey. In Proc. ECAI-94 Workshop on Agent Theories, Architectures, and Languages, 8.-9. August 1994, Amsterdam, LNAI 890, 1-39, Springer Verlag, Berlin 1995

Tree Shaped Computations as a Model for Parallel Applications

Peter Sanders
Max-Planck-Institute for Computer Science
66123 Saarbrücken, Germany
E-mail: sanders@mpi-sb.mpg.de

Abstract

It is shown how a large class of applications can be parallelized by modeling them as *tree shaped computations*. In particular this class contains many highly irregular and completely unpredictable computations as they occur in heuristic search. We explain why the model even remains useful in the presence of some frequently observed subproblem dependencies.

1 Introduction

Many algorithms in operations research and artificial intelligence are based on the backtracking or depth first principle for traversing large implicitly defined trees (e.g. [ABF93, FMM94, FM87, KC94, KGGK94, Kor85, MT90]). In addition, some adaptive numerical algorithms for numerical integration by [PTVF92], for finding eigenvalues of tridiagonal matrices [CRY94] or for nonlinear optimization [NM96] have a similar structure. We will see, that even modeling the seemingly unrelated problem of loop scheduling in this way can be advantageous.

We introduce a model for the parallelization and load balancing aspects of these applications named *tree shaped computations* which makes the common properties visible while hiding unnecessary detail. Essentially, the model is applicable whenever we have applications which can be subdivided into independent subtasks. The model makes almost no assumptions about shape or predictability of the computations and can therefore handle highly irregular problems. Nevertheless, using *random polling*, a well known and simple receiver initiated load balancing algorithm, all tree shaped computations can be parallelized almost optimally.

Tree shaped computations, random polling and some other algorithms and a spectrum of applications have been investigated in the PhD thesis [San97]. The present paper can be considered a summary of the modeling aspects of this work. There is a large body of related work. We can only give a cross section and refer to the references in [San97] for a more detailed discussion. Early work on random polling and an application independent library is described in [FM87]. Random polling and other receiver initiated load balancing methods are also of central importance for parallel functional and logical programming languages (e.g., [ABF93, KC94]). Tree shaped computations can be considered a generalization of the α-splitting model used in [KGGK94]. A related model based on a subclass

of multithreaded computations is used in the Cilk project [BL94, BFJ+96]. The ZRAM library [BMFN98] is another recent implementation effort.

We neither want to focus on the theoretical model and its analysis alone nor on a particular application. Rather, we try to help bridging the gap between theory and application going from the abstract to the concrete. In Section 2 we therefore first introduce the abstract model and complement this by the practically most interesting algorithmic results in Section 3. Then we explain how the the simple model can be adapted to situations like depth first tree traversal, branch-and-bound or loop scheduling. Only then, in Section 5 we give some more detailed examples.

2 The Abstract Model

All the work to be done by a tree shaped computation is initially subsumed in a single *root problem* I_{root} located on a processing element (PE) numbered 0. All other PEs start idle, i.e., they only have an *empty problem* $I_\emptyset$.

What makes parallelization attractive, is the property that problem instances can be subdivided into *subproblems* which can be solved independently by different PEs. We model this property by a *splitting operation* $\text{split}(I)$ which splits a given (sub)problem into two new subproblems subsuming the parent problem. Let T_{split} denote a bound on the time required for the split operation.

The operation $\text{work}(I,t)$ transforms a given subproblem I by performing sequential work on it for t time units. The operation also returns when the subproblem is exhausted.

What makes parallelization difficult, is that the *size*, i.e., the execution time $T(I) := \min\{t \mid \text{work}(I,t) = I_\emptyset\}$, of a subproblem cannot be predicted. In addition, the splitting operation will rarely produce subproblems of equal size. For the analysis we assume however that subproblems are independent in the sense that

$$\forall I : \text{split}(I) = (I_1, I_2) \implies T(P) = T(I_1) + T(I_2) \tag{1}$$

regardless when and where I_1 and I_2 are worked on. In Section 4.5 we discuss what happens if this assumption is violated.

Next we quantify some guaranteed "progress" made by splitting subproblems. Every subproblem I belongs to a *generation* $\text{gen}(I)$ recursively defined by $\text{gen}(I_{\text{root}}) := 0$ and $\text{split}(I) = (I_1, I_2) \implies \text{gen}(I_1) = \text{gen}(I_2) = \text{gen}(I) + 1$. For many applications, it is easy to give a bound on a *maximum splitting depth* h which guarantees that the size of subproblems with $\text{gen}(I) \geq h$ cannot exceed some *atomic grain size* T_{atomic}. Since h is the only factor which constrains the shape of the emerging "subproblem splitting tree", it can be viewed as a measure for the irregularity of a problem instance.[1]

Finally, subproblems can be moved to other PEs by sending a message.

3 Load Balancing Algorithms

In [San97] a number of load balancing algorithms are investigated using a detailed model for message passing parallel computers with P PEs coupled via various interconnection

[1] Obviously, very regular instances with large h are possible. But in applications where this is frequently the case, one would look for a splitting function exploiting these regularities to decrease h.

networks. Here, we contend ourselves with an outline of the practically most important algorithm using a simplified version of the the LogP model [CKP+93] as the machine model. The communication costs are expressed in terms of $T_{\text{rout}} := L + o + g$, i.e., the sum of communication latency, sending overhead and gap between messages. We assume that the characteristic message length is defined in such a way that a subproblem can be specified using a single message.

The *random polling* algorithm is very simple: Each PE handles exactly one (possibly empty) subproblem at any point in time. If a PE runs out of work it sends requests to randomly chosen PEs until a busy one is found which splits its piece of work and transmits it to the requester. This algorithm was discovered independently multiple times. Refer to [FM87] for an early reference. Despite of its simplicity and the unpredictablility of tree shaped computations, random polling is very efficient:

Theorem 1. *The expected parallel execution time for solving a tree shaped computation using random polling is*

$$\mathbf{E}T_{\text{par}} \leq (1+\varepsilon)\frac{T_{\text{seq}}}{P} + \mathrm{O}\left(T_{\text{atomic}} + h\left(\frac{1}{\varepsilon} + T_{\text{rout}} + T_{\text{split}}\right)\right) \textit{ for any } \varepsilon > 0 \ .$$

In particular, $\mathrm{O}(h)$ consecutive splitting and routing operations and an overall message traffic in $\mathrm{O}(Ph)$ are sufficient. This bound is optimal in the sense that that there are tree shaped computations which require at least as many splits.

Even in its simplest form, this algorithm also performs very well in practice. For small instances a few fine points can have a noticable influence however.

- In the basic algorithm we have a flurry of acitvity before all PEs get something to do. Although it can be shown that this period is very short, we can save some time by broadcasting the root problem initially and splitting it into P disjoint parts. This can be done without communication in time $\lceil \log P \rceil \, T_{\text{split}}$.
- We should use a termination detection protocol which works in $\mathrm{O}(T_{\text{rout}} \log P)$ time. In [San97] it is explained how this can be done using an asynchronous reduce-add operation.
- For some network interfaces it makes sense to try overlapping communication and computation. For this purpose we propose to maintain two subproblems on each PE. When one becomes empty start a request and switch to the other subproblem while waiting for a reply. More precisely, each PE should work on the local subproblem with the largest generation and split the subproblem with the smallest generation when answering a request.

The algorithm also works well if the speed of the PEs in a network of workstation varies dynamically due to external load since the additional irregularity introduced by this is comparably small. We can even tolerate a complete deactivation of a worker process as long as it still answers load requests. Such a mode is desirable for "guest" jobs on interactively used workstations. Even the time for splitting subproblems can be saved if we introduce the additional rule that a deactivated worker process sends its entire subproblem when it gets a request.

4 Making the Model Concrete

In Section 4.1 we start by explaining how the main application area of tree shaped computations – parallel depth first heuristic search in implicitly defined trees – can be modeled by tree shaped computations. Then Section 4.2 complements this with other seemingly unrelated applications where tree shaped computations can nevertheless be useful. Section 4.3 then gives a number of ways how an application can be interfaced with a load balancer based on tree shaped computations. Some simple aspects of many applications which are *not* modeled by tree shaped computations turn out to be implementation details in Section 4.4. If the independence assumption (1) is violated, we have a more serious problem which is discussed in detail in Section 4.5.

4.1 Depth First Traversal of Trees

A sequential algorithm for depth first traversal represents its state using a stack of nodes. (This may or may not be the recursion stack.) It pops nodes from the stack, evaluates them and pushes successors building a new level of the stack for each new level of the tree. Such a stack can also be used to represent a subproblem for the corresponding tree shaped computation and the operation "work" is the same as the sequential traversal algorithm. T_{atomic} is a bound on the time needed to evaluate a single node. In order to split a subproblems we produce two stacks which represent disjoint collections of subtrees. In the simplest case, we split off of a single subtree, preferably as deep as possible in the stack. If the the maximum branching factor of the tree is large, it is better to consider all the nodes in the deepest level of the stack together and to divide them between the two child problems in an alternating fashion. By this simple measure we can guarantee that $h \leq d \lceil \log B \rceil$ if d is the depth of the tree and B its maximum branching factor. Sometimes we can do even better by dividing nodes on all levels of the stack together. These different approaches to tree splitting have been described in [RK87].

One of our motivations for introducing tree shaped computations was to abstract from these implementation details without making the model less accurate.

4.2 Scheduling Intervals

Parallel programming languages and even preprocessor directives for Fortran of C/C++ often allow the automatic parallelization of for-loops whose individual iterations represent independent computations [Ope97]. If the individual iterations have strong variations in their execution time, the common practice is to use a centralized scheduler giving out chunks of iteration indices to the worker PEs. Even if heuristics are used which start with large chunks and later polish load imbalance by fading the chunk size, for large P this can become a delicate tradeoff between insufficient load balancing and communication overload of the master. A more robust and scalable solution is to use tree shaped computations by representing a subproblem as an interval of iteration indices. Splitting simply halves the interval of indices not yet iterated.

Very irregular instances of the loop scheduling problem are for example observed if the individual iterations are in fact tree search problem. For example, in airline crew pairing generation [GHL97] a heuristic enumeration of round trips for anonymous crew members is done. Each loop iteration is a backtrack search starting with a particular connection. In [AKRS90] each loop iterations represents a possible fault in a VLSI circuit

and a backtracking search is used to find a test pattern covering this fault. This application is so irregular that random polling is used to parallelize both the outer loop and the backtrack search.

Many numerical and geometrical divide-and-conquer applications can be modeled by computations working on a multidimensional interval which is adaptively subdivided (e.g., [PTVF92]). Again, tree shaped computations are a good model for this if the subproblems can be treated largely independently and if the amount of work is irregularly distributed over the subintervals. Note that there can be splitting functions which use application specific knowledge to select the dimension in which to cut or to cut not in equal halves in order to achieve a smaller h. A hybrid between interval based and tree based schemes are applications working with quad-trees or oct-trees.

4.3 Implementation Interface

Tree shaped computations can directly be used as an implementation interface. In the PIGSeL library [San96], the slimmest interface requires the application to supply functions for initializing, splitting, packing and unpacking subproblems. More abstract interfaces can automate some of this. In the PIGSeL library there is an interface based on specifying node expansions. We have already seen that a compiler can parallelize loops and in parallel functional or logical programming languages the run time system can manipulate the stack to split recursive computations (e.g., [ABF93, KC94]).

4.4 Initialization, Completion and Subproblem Encoding

Tree shaped computations omit some details from the model which do not influence the load balancing problem much but are nevertheless of some practical importance. First, on distributed memory machines most applications require some replicated information to be broadcast to all PEs. In particular, this often encompasses the entire description of the problem instance. This is the reason why we can later assume that a subproblem representation is so compact that it fits into a single message. For example, the state of a Prolog computation can be compactly represented as a bit string encoding the decisions made by the program [KC94].

Tree shaped computations should not be mixed up with divide-and-conquer applications like quicksort however, where we start with very large subproblem descriptions which become shorter quickly. On the one hand, these problems are often more predictable than difficult tree shaped computations because there is a strong correlation between the size of a subproblem representation and the work left to be done. On the other hand, we dearly need this information in order to minimize communication. Considering representation lengths to be short and uniform is too crude an approximation here. (At least if we cannot assume a shared memory.)

Tree shaped computations are only concerned with how the work is distributed and not how the results are collected. This is usually much easier than the distribution however, because many subproblems will turn out not to contribute to the result. At worst, we can always retrace the distribution to combine subproblem solutions to an overall solution. This is particularly simple if the function combining the results is associative and commutative because we can then simply use a global reduction operation (e.g., counting solutions, finding one best solution).

4.5 Speculativity

Perhaps the most severe limitation of tree shaped computation is that the independence assumption (1) is not always true. We now explain why the model can nevertheless be useful even in the presence of subproblem dependencies.

Finding the first solution: Often tree search applications stop when they find the first node which represents a solution. Translated into our language this means that all subproblems suddenly become empty. This leads to the well known phenomenon of *speedup anomalies*, i.e., speedups $S \ll P$ or $S \gg P$ because the parallel algorithm happens to find a solution very late or very early. But often these algorithms are used for verifying that no solution exist, e.g., in order to prove the unsatisfiaility of a logical formula [BS96]. In this case no anomalies occur. As long as there are no heuristics ordering the successors of a node by their likelihood to lead to a solution, we can reasonably expect that negative anomalies do not outweigh positive anomalies and the only measure we have to take is to stop all PEs quickly when a solution is found (e.g., [SMV87, RK93]). Even with node ordering heuristics, many practical applications work surprisingly well. We can add splitting heuristics which try to produce subproblems which have an about equal chance of leading to a solution. Furthermore, parallel search can even achieve superlinear speedup on the average relative to sequential depth first search since it is less likely run into dead ends. We give an example in Section 5.2. More extreme cases of superlinar speedup are analyzed in [Ert92].

Depth-first branch-and-bound behaves similar to applications where we are looking for the first solution. Here, whenever an *improved* solution is found, all other subproblems should learn about the new quality bounds and are thereby reduced in size. We should not simply broadcast new bounds since this can lead to severe contention if many new bound are found concurrently. Rather the bounds should first be send along a reduction tree to PE 0. Then suboptimal bounds can be thrown away early and we still need only $O(T_{\text{rout}} \log P)$ time.

Other pruning heuristics: Note that pruning heuristics in backtrack search only invalidate the independence assumption if they depend on the evaluation of subtrees whereas many simple heuristics are only a function of the path leading from the root to the present node.

A path in a backtrack tree usually represents a sequence of decisions leading to a solution or a dead end. A heuristics that is sometimes useful tries to prove that backtracking a particular decision cannot lead out of the dead end. In this case all alternatives of this decision can be pruned and backtracking proceeds further up the tree. Although this heuristics *can* generate dependencies it is usually most effective on small subtrees whereas the load balancing mostly involves large subtrees which are usually independent.

In contrast, game-tree-search with $\alpha\beta$-pruning produces dependencies throughout the tree and consequently parallelization is rather difficult. Nevertheless, random polling turned out to be a good load balancing algorithm for game-tree-search [Fel93, FMM94]. In this case tree shaped computations can still be considered a good model for the load balancing aspect of the application although additional more application-specific models for the "speculativity" aspect are needed.

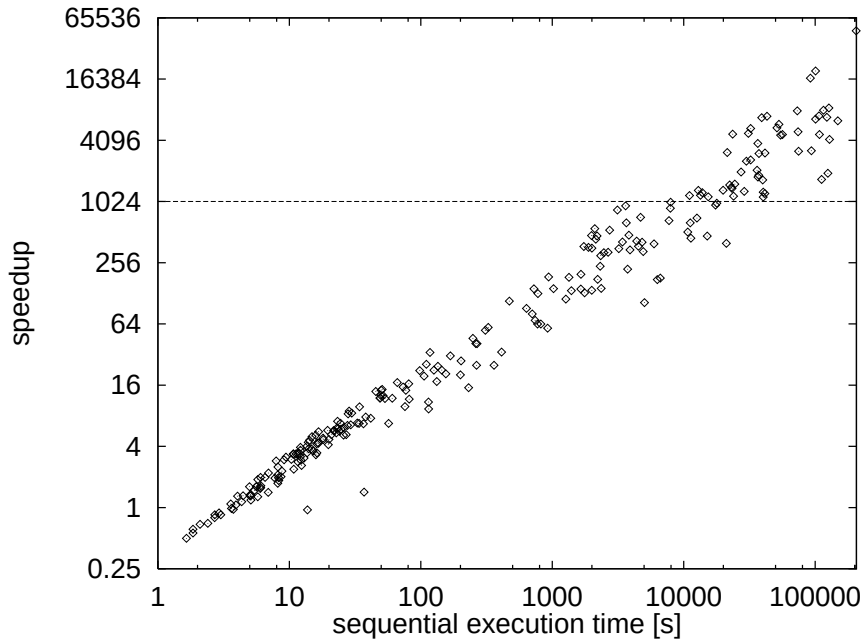


Figure 1: Speedup for 256 instances of the knapsack problem on 1024 PEs.

5 Implementation experiences

A number of example applications have been implemented most of them using the library PIGSeL [San96]. In the following we do not reiterate that whenever tree shaped computations are a good approximation to the real situation we get good parallel efficiency whenever the parallel execution time exceeds a few seconds even on many PEs. Rather, we want to focus on the experiences regarding how well the model fits.

5.1 Golomb rulers

stem from discrete mathematics [BG77] but have applications in radio astronomy and coding theory. A ruler of length m with k marks is defined by integer positions $0 = m_1$, $m_2, \ldots, m_k = m$ with the property $|\{m_j - m_i : 1 \leq i < j \leq k\}| = k(k-1)/2$, i.e., the ruler can be used to measure a maximum number of distances. For a given k we want to find a ruler with minimal m. Most systematic search algorithms for this problem can be viewed as a variant of depth first branch-and-bound. There are many additional heuristics but they do not introduce additional dependencies. Speedup anomalies are quite small because finding the optimal solution usually takes less time than verifying that this is the optimal solution. Also, good heuristics are available which sometimes yield the optimal solution immediately. In this case, a pure verification search is done which is perfectly modeled by tree shaped computations. This is not unusual for branch-and-bound applications.

5.2 The 0-1 knapsack problem

is one of the most intensively studied problems in combinatorial optimization [MT90]. An instance is defined by m items with weight w_i and profit p_i and a knapsack of capacity M. We are looking for $x_i \in \{0,1\}$ such that $\sum p_i x_i$ is maximized subject to the constraint $\sum w_i x_i \leq M$, i.e., we want to achieve a maximal profit from items in the knapsack without exceeding its capacity. For large m and arbitrary w_i, the best known algorithms are based

on a very fine-grained depth first branch-and-bound search [MT90]. An experiment was done with random instances generated using a statistics which yields large, difficult to solve yet tractable instances with very deep irregular search trees. The double-logarithmic plot in Figure 1 shows the relation between speedup and sequential execution time for 256 random instances with $m = 2000$, random $w_i \in [0.01, 1.01]$, random $p_i \in [w_i + 0.1, w_i + 0.125]$ and $M = \sum w_i/2$ on 1024 PEs of a Parsytec GCel.[2]

From a load balancing point of view it is interesting that the strategy to split on all levels of the tree at once is quite successful here. But more astonishing is that the average speedup over the 256 instances generated was 1410 on 1024 processors. This indicates that simple sequential depth first search is not robust enough for difficult instances because it sometimes bogs down in huge subtrees which do not contain the optimum.

5.3 The 15-puzzle

[Kor85] is a well known toy widely used as a benchmark in AI. You have to shift 15 scrambled squares in a 4×4 frame into the right order. In [Kor85] it is solved using iterative deepening search, i.e., a sequence of depth first searchs where the number of moves from the starting position is limited. All but the last iterations are perfectly modeled by tree shaped computations. The last iteration stops when the first solution is found. Speedup anomalies are present but not overwhelming. The extremly fine-grained search used in [Kor85] is an interesting challenge for the implementation. Using tree shaped computations as an implementation interface directly it was possible to use a single search algorithm for all levels of the tree and to get responsiveness to splitting requests with little overhead although less than 100 machine instructions are needed per node expansion.

5.4 The Firing Squad Synchronization Problem

is a classical problem in cellular automata theory asking for (time-optimal) algorithms for synchronizing a one-dimensional cellular automaton (using a minimal number of states). Using a heuristic backtrack search in the space of transition tables it was possible to show that there is no time-optimal solution with four states [San94]. Even a massively parallel implementation of this algorithm on 16384 PEs searches only a few percent more nodes although the dead-end heuristics mentioned above is crucial for its sequential performance.

6 Conclusions

Tree shaped computations are a good model for a wide range of applications. With random polling we have an algorithm which parallelizes them very efficiently although very irregular and completely unpredictable computations are allowed. At the same time the model is the basis for an efficient and very slim interface between the load balancer and a reusable and portable load balancing library. Even if dependencies between subproblems are present, the predictions made by the simple model are often correct and the load balancer works well. Nevertheless, more accurate models are an important area for future

[2]We thank the Paderborn Center for Parallel Computing (PC^2) for making this machine available.

work. The subroutine call semantics modeled by the *fully strict multithreaded computations* used in the Cilk system [BL94] are one step into this direction. However, they do not model the unpredictable dependencies observed in the examples considered here.

References

[ABF93] G. Aharoni, Amnon Barak, and Yaron Farber. An adaptive granularity control algorithm for the parallel execution of functional programs. *Future Generation Computing Systems*, 9:163–174, 1993.

[AKRS90] S. Arvindam, V. Kumar, V. N. Rao, and V. Singh. Automatic test pattern generator on parallel processors. Technical Report TR 90-20, University of Minnesota, 1990.

[BFJ+96] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall. An analysis of dag-consistent distributed shared-memory algorithms. In *8th ACM Symposium on Parallel Algorithms and Architectures*, pages 297–308, 1996.

[BG77] G. S. Bloom and S. W. Golomb. Applications of numbered undirected graphs. *Proceedings of the IEEE*, 65(4):562–570, April 1977.

[BL94] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *Foundations of Computer Science*, pages 356–368, Santa Fe, 1994.

[BMFN98] A. Brüngger, A. Marzetta, K. Fukuda, and J. Nievergelt. The parallel search bench zram and its applications. *Annals of Operations Research*, 1998. to appear.

[BS96] M. B"ohm and E. Speckenmeyer. A fast parallel SAT-solver - efficient workload balancing. *Annals of Mathematics and Artificial Intelligence*, 17:381–400, 1996.

[CKP+93] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subromonian, and T. v. Eicken. LogP: Towards a realistic model of parallel computation. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, San Diego, 1993.

[CRY94] S. Chakrabarti, A. Ranade, and K. Yelick. Randomized load balancing for tree-structured computation. In *Scalable High Performance Computing Conference*, pages 666–673, Knoxville, 1994.

[Ert92] W. Ertel. *Parallele Suche mit randomisiertem Wettbewerb in Inferenzsystemen*. Dissertation, TU München, 1992.

[Fel93] R. Feldmann. *Game Tree Search on Massively Parallel Systems*. Dissertation, Universität Paderborn, August 1993.

[FM87] R. Finkel and U. Manber. DIB – A distributed implementation of backtracking. *ACM Trans. Prog. Lang. and Syst.*, 9(2):235–256, April 1987.

[FMM94] R. Feldmann, P. Mysliwietz, and B. Monien. Studying overheads in massively parallel min/max-tree evaluation. In *ACM Symposium on Parallel Architectures and Algorithms*, pages 94–103, 1994.

[GHL97] C. Goumopoulos, E. Housos, and O. Liljenzin. Parallel crew scheduling on workstation networks using PVM. In *EuroPVM-MPI*, number 1332 in LNCS, Cracow, Poland, 1997.

[KC94] J. C. Kergommeaux and P. Codognet. Parallel logic programming systems. *ACM Computing Surveys*, 26(3):295–336, 1994.

[KGGK94] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing. Design and Analysis of Algorithms*. Benjamin/Cummings, 1994.

[Kor85] R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.

[MT90] S. Martello and P. Toth. *Knapsack Problems – Algorithms and Computer Implementations*. Wiley, 1990.

[NM96] A. Nonnenmacher and D. A. Mlynski. Liquid crystal simulation using automatic differentiation and interval arithmetic. In G. Alefeld and A. Frommer, editors, *Scientific Computing and Validated Numerics*. Akademie Verlag, 1996.

[Ope97] OpenMP. *OpenMP Fortran Application Interface*, 1.0 edition, 1997.

[PTVF92] W. H. Press, S.A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C*. Cambridge University Press, 2. edition, 1992.

[RK87] V. N. Rao and V. Kumar. Parallel depth first search. Part I. *International Journal of Parallel Programming*, 16(6):470–499, 1987.

[RK93] V. N. Rao and V. Kumar. On the efficiency of parallel backtracking. *IEEE Transactions on Parallel and Distributed Systems*, 4(4):427–437, April 1993.

[San94] P. Sanders. Massively parallel search for transition-tables of polyautomata. In *Parcella 94, VI. International Workshop on Parallel Proccessing by Cellular Automata and Arrays*, pages 99–108, Potsdam, 1994.

[San96] P. Sanders. A scalable parallel tree search library. In S. Ranka, editor, *2nd Workshop on Solving Irregular Problems on Distributed Memory Machines*, Honolulu, Hawaii, 1996.

[San97] P. Sanders. *Lastverteilungsalgorithmen für parallele Tiefensuche*. Number 463 in Fortschrittsberichte, Reihe 10. VDI Verlag, 1997.

[SMV87] E. Speckenmeyer, B. Monien, and O. Vornberger. Superlinear speedup for parallel backtracking. In C. D. Houstis, E. N.; Papatheodorou, T. S.; Polychronopoulos, editor, *Proceedings of the 1st International Conference on Supercomputing*, volume 297 of *LNCS*, pages 985–993, Athens, Greece, June 1987. Springer.

NSR – A Tool for Load Measurement in Heterogeneous Environments

Christian Röder, Thomas Ludwig, Arndt Bode

LRR-TUM
Lehrstuhl für Rechnertechnik und Rechnerorganisation
Technische Universität München, Institut für Informatik, D–80290 München
e–mail: {roeder|ludwig|bode}@informatik.tu-muenchen.de

Abstract

This paper presents the design and implementation of NSR– the Node Status Reporter. The NSR provides a standard mechanism for measurement and access to status information in clusters of heterogeneous workstations. It can be used by any application that relies on static and dynamic information about this execution environment. A key feature of NSR is its flexibility with respect to the measurement requirements of various applications. Configurability aims at reducing the measurement overhead and the influence on the observed cluster.

1 Motivation

During the last years, clusters of time-shared heterogeneous workstations became a popular execution platform for parallel applications. Parallel programming environments like PVM[1] [5] and MPI[2] [13] have been developed to implement message passing parallel applications. Late development stages and efficient production runs of parallel applications typically require sophisticated on-line tools, e.g. performance analysis tools, program visualization tools, or load management systems. A major drawback of existing tools like e.g. [1, 7, 9, 12] is that all of them implement their own monitoring technique to observe and/or manipulate the execution of a parallel application. Proprietary monitoring solutions have to be redesigned and newly implemented when shifting to a new hardware platform and/or a new programming environment.

In 1995, researchers at LRR-TUM and Emory University started the OMIS[3] project. It aims at defining a standard interface between tools and monitoring systems for parallel and distributed systems. Using an OMIS compliant monitoring system, tools can efficiently observe and manipulate the execution of a message passing based parallel application. Meanwhile, OMIS has been redesigned and is available in version 2 [8]. At the same time, the LoMan project [11] has been started at LRR-TUM. It aims at developing a cost-sensitive system-integrated load management mechanism for parallel applications that execute on clusters of

[1] PVM: Parallel Virtual Machine.
[2] MPI: Message Passing Interface.
[3] OMIS: On-line Monitoring Interface Specification.

heterogeneous workstations. Combining the goals of these two projects, we designed the NSR [6] that is a separate module to observe the status of this execution environment.

The goal of NSR is to provide a uniform interface for *data measurement* and *data access* in clusters of heterogeneous workstations. Heterogeneity covers two aspects: different performance indices of the execution nodes and different data representations due to different architectures. Although primarily influenced by the measurement requirements of on-line tools, the NSR serves as a powerful and flexible measurement component for any type of application that requires static and/or dynamic information about workstations in a cluster. However, the NSR does not post-process and interpret status information because the semantics of measured values is known by measurement applications only. In the following, we will use *measurement application* to denote any application that requires status information.

The document is organized as follows. The next section presents an overview of related work. Public domain tools for status measurement and standardization approaches are discussed. In § 3 we propose the reference model of NSR and discuss the steps towards its design. Selected implementation aspects are presented in § 4. The concept and implementation of NSR is evaluated in § 5. Finally, we summarize in § 6 and discuss future development steps.

2 Related Work

Available results taken into consideration when designing and implementing the NSR can be grouped into three different categories:

- Standard commands and public domain tools (e.g. `top`[4] and `sysinfo`[5]) to retrieve status information in UNIX environments.
- Load modeling and load measurement by load management systems for parallel applications that execute in clusters of heterogeneous workstations (e.g. load monitoring processes [4, 7]; classification of load models [10]).
- The definition of standard interfaces to separately implement software components of tools for parallel systems (e.g. OMIS [8], UMA[6] [14], SNMP[7] [2], and PSCHED [3]).

Typically, developers implement measurement routines for their own purposes on dedicated architectures or filter status information from standard commands or public domain tools. An overview on commands and existing tools is presented in [6]. The implementation of public domain tools is typically divided into three parts. A textual display presents the front-end of the tools. A controlling component connects the front-end with measurement routines that are connected by a tool-defined interface. Hence, porting a tool to a new architecture is limited to implement a new measurement routine. However, most tools only provide status information from the local workstation. Routines for transferring measured data have to be implemented if information is also needed from remote workstations.

The second category taken into considerations covers load modeling requirements of load management systems. A load management system can be modeled as a control loop that tries to manage the load distribution in a parallel system. Developers of load management techniques often adopt measurement routines to retrieve load values from the above mentioned public domain tools. However, it is still not yet clear which status information is best suited for load

[4] `top`: available at ftp://ftp.groupsys.com/pub/top.

[5] `sysinfo`: available at ftp://ftp.usc.edu/pub/sysinfo.

[6] UMA: Universal Measurement Architecture.

[7] SNMP: Simple Network Management Protocol.

modeling in heterogeneous environments. A flexible load measurement component can serve as a basis to implement an adaptive load model [10]. Furthermore, different combinations of status information could be tested to find the values that can adequately represent the load of heterogeneous workstations.

Standard interfaces are defined to allow separate development of tool components. Tool components can subsequently cooperate on the basis of well-defined communication protocols. The exchange of one component by another component (even from different developers) is simplified if both are designed taking into account the definitions of such interfaces. We already mentioned the OMIS project. In § 4.2 we will see that OMIS serves as the basis to define the status information provided by NSR.

The UMA project addresses data collection and data representation issues for monitoring in heterogeneous environments. Status information is collected independently of vendor defined operating system implementations. The design of UMA includes data collection from various sources like e.g. operating systems, user processes, or even databases with historical profiling data. Although most measurement problems are addressed, the resulting complexity of UMA might currently hamper the development and widespread usage of UMA compliant measurement systems. Furthermore, the concept does not strictly distinguish between data collection and subsequent data processing.

The PSCHED project aims at defining standard interfaces between various components of a resource management system. The major goal is to separate resource management functionality, message passing functionality of parallel programming environments, and scheduling techniques. Monitoring techniques are not considered by PSCHED at all. The PSCHED API is subdivided into two parts: 1) a task management interface defines a standard mechanism for any message passing library to communicate with any resource management system; 2) a parallel scheduler interface defines a standard mechanism for a scheduler to manage execution nodes of a computing system regardless of the resource manager actually used. By using a configurable measurement tool, the implementation of a general-purpose scheduler can be simplified.

3 The Design of NSR

The design of a flexible measurement component has to consider the measurement requirements of various applications. A measurement component serves as an *auxiliary module* in the context of an application's global task. Hence, the NSR is designed as a *reactive system*, i.e., it performs activities on behalf of the supported applications. According to the main goal of NSR to provide a uniform interface for *data measurement* and *data access*, we consider three questions:

1. *Which data is needed?* Different applications need different types of status information. Status information can be classified into *static* and *dynamic* depending on the time-dependent variability of its values. Static and dynamic information is provided for all major resource classes, i.e., operating system, CPU, main memory, secondary storage, and communication interfaces. Interactive usage of workstations in a time-shared environment gains increasing importance when parallel applications should be executed. Hence, the number of users and their workstation utilization are measured as well.
2. *When is the data needed?* On-line tools like e.g. load management systems, often have a cyclic execution behavior and measurements are *periodically* initiated. In this case, the period time between two successive measurements depends on the intent of measurement applications. However, measurement applications can have other execution behaviors as

well. Now, status information can be measured *immediately and exactly once* as soon as it is needed.

3. *From which set of nodes is the data needed?* On-line tools can be classified regarding their implementation as distributed or centralized. In case of a central tool component (e.g. the graphical user interface of a program visualization tool), status information of multiple workstations is typically needed at once. Status information of any workstation is accessible on any other workstation.

Different applications might need the same type of status information, although their global tasks are different. For example, a load management system can consider I/O statistics of parallel applications and maps application processes to workstations with high I/O bandwidth. The same status information can be used by a visualization tool to observe the I/O usage statistics in a workstation cluster. The following section presents the reference model of NSR before we list the design goals in detail.

3.1 The Reference Model

The reference model of NSR is illustrated in Fig. 1. The NSR builds a layer between measurement applications and the execution environment consisting of a cluster of heterogeneous workstations.

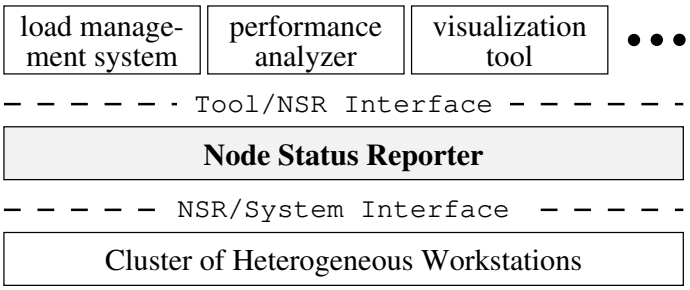


Figure 1: The Reference Model of NSR.

The reference model is summarized as follows. The NSR supports on-line tools for parallel applications like e.g. load management systems, performance analysis tools, or program flow visualization tools. However, any type of measurement application can utilize the NSR. Applications communicate with the NSR via the `Tool/NSR Interface`. Using this interface, applications can manage and control the execution behavior of NSR. Recall that the NSR is designed as a reactive system on behalf of the measurement applications. The applications are responsible to configure the NSR with respect to their measurement requirements, i.e., the type of status information and the time at which measurements have to be initiated. The interface provides uniform access to status information of any workstation that is observed by NSR. If necessary, the NSR distributes measured values between the workstations with respect to the applications' location.

The `NSR/System Interface` connects the NSR to a cluster of heterogeneous workstations and hides the details of workstation heterogeneity. The main purpose of this interface is to connect the NSR and measurement routines for various architectures. Furthermore, the interface provides a uniform data representation with unity scale. Developers are free to implement their own data measurement and data collection routines.

3.2 Design Goals

The following major objectives have to be pursued during the design of NSR with respect to the above listed measurement requirements:

Portability: The target architecture of NSR is a cluster of heterogeneous workstations. The same set of status information has to be accessible in the same way on every architecture for which an implementation exists. New architectures should be easily integrated with existing implementations.

Uniformity: For each target architecture, the measured values have to be unified with respect to their units of measure (i.e., bytes, bytes/second, etc.). Hence, status information becomes comparable also for heterogeneous workstations.

Scalability: Tools for parallel system typically need status information from a set of workstations. An increasing number of observed workstations must not slow down these applications.

Flexibility: The NSR provides a wide variety of status information. Measurement applications can determine which subset of information has to be measured on which workstations. Additionally, the frequency of measurements has to be freely configurable.

Efficiency: The NSR can simultaneously serve as a measurement component for several applications. If possible, concurrent measurements of the same data should be reduced to a single measurement.

3.3 Design Aspects

According to the design goals and the application requirements we consider the following design aspects:

Data measurement: The measurement of a workstation's status information issues system calls to the operating system. It is one of the main tasks of the operating systems to update these values and to keep them in internal management tables. System calls are time consuming and slow down the calling application. Additionally, different architectures often require different measurement routines to access the same values. For portability reasons, data measurement is decoupled from an application's additional work and performed asynchronously to it. The set of data to be measured is freely configurable by a measurement application.

Synchronization: It is likely to happen in time-shared systems that several measurement applications execute concurrently, e.g. several on-line tools for several parallel applications execute on an overlapping set of workstations in the cluster. A severe measurement overhead on multiple observed workstations can result from an increasing frequency of periodically initiated measurements. Multiple measurements of the same status information are converted to a single measurement, thereby reducing the measurement overhead. However, synchronized data measurement demands for a mechanism to control the synchronization. We use interrupts to synchronize data measurements. In case of periodic measurements, the period of time between any two measurements is at least as long as the period of time between two interrupts. However, this limits the frequency of measurement initiations and can delay the measurement application. We implemented two concepts to reduce the delays. Firstly, if a measurement application immediately requires status information then measurements can be performed independently from the interrupt period. Secondly, we use asynchronous operations for data measurement and data access.

Asynchronous Operations: Asynchronous operations are used to hide latency and overlap

data measurement and computation. An identifier is immediately returned for these operations. A measurement application can wait for the completion of an operation using this identifier. Measurements are carried out by NSR, while applications can continue their execution.

Data Distribution: Apart from data measurement, the status information is transparently transferred to the location of a measurement application. A data transfer mechanism is implemented within the NSR. Again, data transfer is executed asynchronously while applications can continue their execution.

4 Implementation Aspects

4.1 Client-Server Architecture

The architecture of NSR follows the client-server paradigm. The client functionality is implemented by the *NSR library*. A process that has linked the NSR library and invokes library functions is called *NSR client* (or simply *client*). An NSR client can request service from the *NSR server* (*server*) by using these library functions. The NSR server can be seen as a global administration instance that performs data measurement and data distribution on behalf of the clients. It is implemented by *a set of replicated NSR server processes* that cooperate to serve the client requests. The server processes are intended to execute in the background as other UNIX processes. To clarify our notation we call them *NSR daemon processes* (or simply *daemons*). A single NSR daemon process resides on each workstation in the cluster for which an implementation exists.

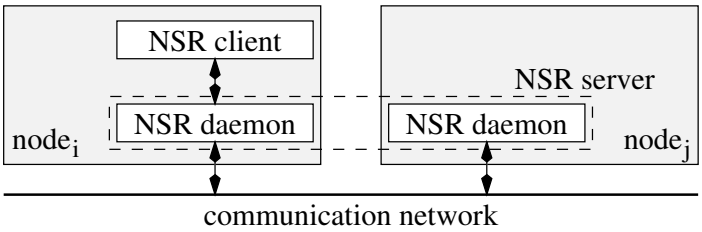


Figure 2: The Client-Server Architecture of NSR.

An exemplary scenario of this architecture is illustrated in Fig. 2. One NSR daemon process resides on workstation node$_i$, while a second daemon resides on workstation node$_j$. The client on node$_i$ can only communicate with the local NSR daemon process. Applications can not utilize the NSR server if a local NSR daemon process is not available. The `Tool/NSR Interface` is implemented on top of this interconnection.

The local NSR daemon process controls the execution of a client request. From the viewpoint of a client, its local NSR daemon process is called *NSR master daemon*. If necessary, the NSR master daemon forwards a request to all daemons that participate in serving a client's request. The latter daemons are also called *NSR slave daemon processes*. A daemon is identified by the name of the workstation on which it resides. Note, the attributes *master* and *slave* are used to distinguish the roles each daemon plays to serve a client request. However, the attributes are not reflected in the implementation. An NSR daemon process can be both master and slave because several clients can request service from the NSR server concurrently. The main difference between master and slave daemons is that an NSR master daemon keeps additional information about a requesting client. Slave daemons do not know anything about the

client that performed the request. The NSR/System Interface connects the workstation specific measurement routines with the NSR daemon processes. Daemons exchange messages via the communication network.

All information needed by the NSR server to serve the client requests are stored by the NSR daemon processes. Contrarily, a client is stateless with respect to data measurement. The result of a previously initiated library call does not influence the execution of future library calls. Hence, no information has to be stored between any two calls.

4.2 Status Information

The data structure nsr_info is the container for the status information of workstations. nsr_info is communicated between the NSR components.

```
struct nsr_info {
  long              mtag;
  long              tick;
  static_hostinfo   stthi;
  dynamic_hostinfo  dynhi;
};
```

The first entry mtag is a sequence number for the status information. Applications can identify old and new values by comparing their mtag values. The second entry tick is a timestamp for the status information. It indicates the interrupt number at which the measurement was carried out. In case of periodic measurements, a master daemon uses tick to control data assignment if several clients request status information with different periods of time. Information about resources and interactive usage are split into two data structures. Static information is stored in static_hostinfo and describes a workstations e.g. in terms of type of operating system, number of CPUs, size of main memory, etc. Furthermore, benchmark routines are used to determine performance indices of workstation resources. Dynamic information is stored in dynamic_hostinfo and represents the utilization values of workstation resources. For detailed information, we refer to the information service **node_get_info** in the OMIS document[8] [8, pp. 69 ff.]. The definition of this service matches the definition of these two data structures.

4.3 NSR Library Functions

The NSR library functions are split into two major groups. The first group covers *administrative functions* to manage the execution behavior of the NSR server. The second group includes *data access routines* to retrieve status information that has been measured by the daemons. Additional functions are provided that primarily ease the usage of the NSR.

Administrative functions are needed to configure the NSR server. They are available as *blocking* or *non-blocking* calls. The following example illustrates the difference between these two types. Initially, a client attaches itself to a set of NSR daemon processes with nsr_iattach(), that is a non-blocking call. The local daemon analyzes the parameters of the request, performs authentication checks for the client, and prepares one entry of its internal client management table. From the client request, the names of workstations are extracted from which status information should be retrieved. Subsequently, the master daemon forwards the request to all requested daemons which themselves store the location of the master daemon. As a result, the master daemon returns the identifier waitID for the request. The client

[8] http://wwwbode.informatik.tu-muenchen.de/~omis

can continue with its work. Later on, the client invokes `nsr_wait()` with `waitID` to finish the non-blocking call. This call checks whether all requested daemons participate to serve the client's request. The return value to the client is the identifier `nsrID`, that has to be passed with every subsequent library call. The master daemon uses `nsrID` to identify the requesting client. In any case, a non-blocking call has to be finished by `nsr_wait()` before a new call can be invoked by a client. A blocking library call is implemented by its respective non-blocking counterpart immediately followed by `nsr_wait()`.

Three additional administrative functions are provided by the NSR library. Firstly, a client can program each daemon by sending a measurement configuration message. A measurement flag determines the set of status information that should be measured by each daemon. In case of periodic measurements, a time value determines the period of time between any two measurements. A new configuration message reconfigures the execution behavior of all attached daemons. Secondly, the set of observed workstations can be dynamically changed during the client's execution time. Finally, a client detaches itself from the NSR server as soon as it does not need service any more.

The second group of NSR library functions provides access to the measured data. A request for immediate measurement blocks the caller until the status information of the requested workstation is available. In case of periodic measurements, the local daemon stores the status information of every node into a local memory segment. A library call reads the memory location to access the information. A client can read the status information of all observed workstations at once. Additionally, it can request the status information of one specific workstation. The NSR master daemon stores status information independently from the execution behavior of requesting clients. Again, it is the client's responsibility to periodically read the data.

4.4 NSR Daemon Functionality

At startup time, an NSR daemon process measures a default set of status data and multicasts the values to all daemons in the cluster. During this phase, it initializes an internal timer that periodically interrupts the execution of the NSR daemon process at fixed time intervals. The period of time between two successive interrupts is called `NsrTick`. Any time interval used by the NSR implementation is a multiplier of this basic `NsrTick`. As shown in Fig. 3, three main tasks can be executed by a single daemon: *client request handling*, *data measurement*, and *data management*.

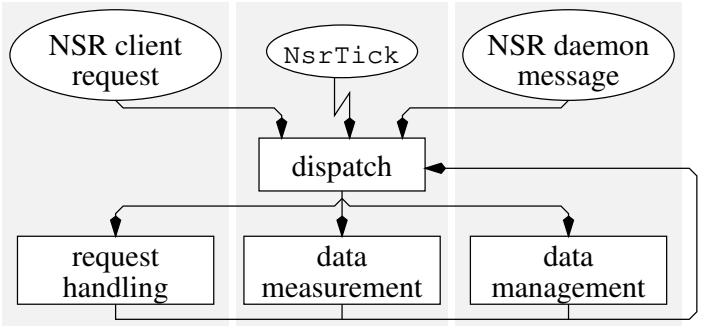


Figure 3: Functionality of an NSR Daemon

A central dispatching routine determines the control flow of the NSR daemon processes. It invokes one of the tasks depending on one of the following asynchronous events:

1. the expiration of `NsrTick`,
2. the arrival of an NSR client request, or
3. the arrival of another daemon message.

In the first case, a default set of status information is updated every $n \times$ `NsrTick` with a predefined number n as long as no client request has to be handled. This concept of *alive-messages* ensures that all daemons know about all other daemons at any time. Note, a daemon tries to propagate a *breakdown-message* in case of a failure. If client requests indicate periodic measurements then `NsrTick` is used to check for the expiration of a requested period of time. If necessary, the daemon measures exactly that status information requested at this point of time. The set of data to be measured is calculated as a superset of the status information currently requested by all clients.

The arrival of an NSR client request causes the dispatcher to invoke the request handling functionality of the daemon. Depending on the request type, the daemon updates an internal management table that keeps information about its local clients. Request handling also includes the forwarding of a client request to all daemons that participate to serve the client request. A result is sent to the client in response to each request.

A daemon can receive different types of messages from other daemons, i.e., alive-messages, breakdown-messages, reply-messages and update-messages. Reply-messages are sent by an NSR slave daemon in response to a forwarded client request. The result of a reply is stored in the client management table. Later on, the results of all slave daemons are checked if the daemon handles the previously mentioned `nsr_wait()` call. Finally, the update-messages indicate that a daemon sends updated status information in case of periodic measurements. New status information of a workstation invalidates its old values. Therefore, the daemon overwrites old workstation entries in the memory segment.

Each daemon in the cluster can perform the same functionality. Typically, several daemons cooperate to serve a client's request. Without going into further details, we state that the NSR server can handle requests from multiple clients. On the one side, measurement applications can be implemented as a parallel program and each process can be a client that observes a subset of workstations. On the other side, several measurement applications from different users can utilize the NSR server concurrently. Additionally, limited error recovery is provided. If necessary, a daemon that recovers from a previous failure is automatically configured to participate in serving the client requests again.

4.5 Basic Communication Mechanisms

Two types of interaction are distinguished to implement the NSR concept: the *client-server interaction* and the *server integrated interaction*, i.e., the interaction between the NSR daemons. Their implementations are based on different communication protocols.

Client-Daemon Communication:: The communication between a client and its local NSR daemon process is performed *directly* or *indirectly*. Direct communication is used for the administrative functions and involves both the NSR client and the NSR master daemon simultaneously. A TCP socket connection is established over which a client sends its requests and receives the server replies. The asynchronous communication uses a two phase protocol based on direct communication (see description above).

Indirect interaction between a client and the local daemon is used to implement data exchange in case of periodic measurements. In analogy to a mail-box data exchange scheme, the

daemon writes status information to a memory mapped file independently of the clients execution state. Similarly, any client located on the workstation can read this data independently of the daemon's status.

Daemon-Daemon Communication: The NSR server implementation is based on the IP multicast[9] concept. The main advantage of IP multicast over a UDP broadcast mechanism is that the dimension of a cluster is not limited to a LAN environment. A multicast environment consists of senders and receivers. Receiving processes are part of a multicast group. In our context, the NSR server is a multicast group consisting of its daemon processes. Communicated data are converted into XDR[10] to cover the heterogeneity of different architectures. In response to a forwarded client request, an NSR slave daemon sends its reply to the master daemon via an UDP socket connection.

5 Evaluation

We will now evaluate the concept and implementation of NSR. As a point of reference, we compare the public domain software product `top` with the execution of `nsrd`, i.e., the NSR daemon process. Two experiments have been performed on a SPARC Station 10 running the SUN Solaris 2.5.1 operating system. Firstly, we compared the measurement overhead of `top` and `nsrd`. Secondly, we compared the average CPU utilization of both tools to determine their impact on the observed workstation.

A comparison between the measurement overhead of `top` and `nsrd` is illustrated in Fig. 4. Additionally, it shows the cyclic execution behavior of `nsrd`. Both tools have been instrumented with timing routines to measure the duration of a single measurement.

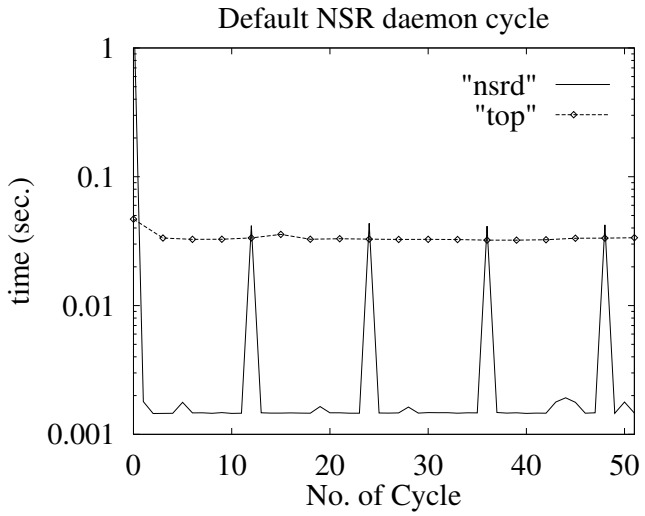


Figure 4: Measurement duration of `top` and `nsrd`.

The NSR daemon process measures its default set of values every $n \times$`NsrTick`, with `NsrTick` = 5 (seconds) and $n = 12$, while `top` updates its values every $m \times$ `NsrTick`, with $m = 3$. The set of measured information covers CPU and memory usage statistics. The set of values is approximately the same for both tools. For $i \bmod 12 \neq 0$, the `NsrTick` just

[9] IP multicast: information available at http://www.mbone.de.

[10] XDR: eXternal Data Representation.

interrupts the execution of `nsrd` that subsequently performs some internal management operations. The time values of Fig. 4 indicate that the measurement routines of `top` and `nsrd` are similar with respect to their measurement overhead. The reason for this similarity is that the measurement technique is almost identically implemented in both tools.

However, the `nsrd` outperforms `top` in terms of CPU percentage that is a measure for the CPU utilization of a process. The CPU percentage is defined as the ratio of CPU time used recently to CPU time available in the same period. It can be used to determine the influence on the observed workstation. The following lines show a shortened textual output of `top` when both tools execute concurrently.

```
  PID  USERNAME  ···   SIZE  ···    CPU  COMMAND
 3317  roeder    ···  1908K  ···  1.10%  top
 3302  root      ···  2008K  ···  0.59%  nsrd
```

The above snapshot was taken when both tools are executing on their regular settings. It combines both the time each tool needs to measure its dynamic values and the time to further process these data. The `top` utility prints a list of processes that is ordered by decreasing CPU usage of the processes. The `nsrd` writes the data to the memory mapped file, converts it into the XDR format, and multicasts it to all other daemons. From this global point of view, `nsrd` outperforms `top` by a factor of 2. The overhead of `top` mainly results from its output functionality. In summary, the NSR introduces less load to the observed system than the `top` program.

6 Summary and Future Work

We presented the design and implementation of NSR that is a general-purpose measurement component to observe the status of a cluster of heterogeneous workstations. The NSR provides transparent access to static and dynamic information for any requesting measurement application. A key feature of NSR is its flexibility with respect to measurement requirements of various applications. If necessary, applications can control and observe different subsets of workstations independently. From the viewpoint of NSR, several applications can be served concurrently.

Currently, we implemented a prototype of NSR for SUN Solaris 2.5 and Hewlett Packards HP-UX 10.20 operating systems. In the future we will enhance the implementation of NSR. By that, measurement routines for various architectures like IRIX 6.2, AIX 3.x, and LINUX 2.0.x will be implemented. Additionally, we will optimize the performance of NSR to reduce the measurement overhead. Regarding performance evaluation, we will evaluate the NSR library calls and the data distribution inside the NSR server. In spring 1998, we will release the NSR software package under the GNU general public license. On some architectures, measurements are performed by reading kernel data structures. In this case, read permission to access the kernel data has to be provided for an NSR daemon process. Security aspects are not violated because the NSR does not manipulate any data. Measurement applications can be implemented without special permission flags. In production mode, a NSR daemon process can automatically start up during the boot sequence of a workstation's operating system.

Finally, the NSR already serves as the load measurement component within the LoMan project [11]. In the future, the NSR will be integrated into an OMIS compliant monitoring system to implement the information service for the execution nodes of a parallel application.

Acknowledgment

We want to express our gratitude to our students Markus Lindermeier and Ingo Rohloff who performed much of the implementation work. We also thank all colleagues who gave useful comments during the design of the NSR.

References

[1] A. Beguelin. Xab: A Tool for Monitoring PVM Programs. In *Workshop on Heterogeneous Processing*, pp. 92–97, Los Alamitos, CA, Apr. 1993.

[2] J.D. Case, M. Fedor, M.L. Schoffstall, and C. Davin. Simple Network Management Protocol (SNMP). Internet Activities Board – RFC 1157, May 1990.

[3] D.G. Feitelson, L. Rudolph, U. Schweigelshohn, K.C. Sevcik, and P. Wong. Theory and Practice in Parallel Job Scheduling. In *IPPS'97 Workshop on Job Scheduling for Parallel Processing*, pp. 1–25, Geneva, Switzerland, Apr. 1997. University of Geneva.

[4] K. Geihs and C. Gebauer. Load Monitor $\mathcal{LM}$ – Ein CORBA-basiertes Werkzeug zur Lastbestimmung in heterogenen verteilten Systemen. In *9. ITG/GI Fachtagung MMB'97*, pp. 1–17. VDE-Verlag, Sep. 1997.

[5] G.A. Geist, J.A. Kohl, R.J. Manchek, and P.M. Papadopoulos. New Features of PVM 3.4 and Beyond. In *Parallel Virtual Machine — EuroPVM'95: Second European PVM User's Group Meeting, Lyon, France*, pp. 1–10, Paris, Sep. 1995. Hermes.

[6] M. Hilbig. Design and Implementation of a Node Status Reporter in Networks of Heterogeneous Workstations. Fortgeschrittenenpraktikum at LRR-TUM, Munich, Dec. 1996.

[7] D.J. Jackson and C.W. Humphres. A simple yet effective load balancing extension to the PVM software system. *Parallel Computing*, 22:1647–1660, Jun. 1997.

[8] T. Ludwig, R. Wismüller, V. Sunderam, and A. Bode. OMIS — On-line Monitoring Interface Specification. Tech. Rep. TUM-I9733, SFB-Bericht Nr. 342/22/97 A, Technische Universität München, Munich, Germany, Jul. 1997.

[9] B.P. Miller, J.K. Hollingsworth, and M.D. Callaghan. The Paradyn Parallel Performance Tools and PVM. Tech. rep., Department of Computer Sciences, University of Wisconsin, 1994.

[10] C. Röder. Classification of Load Models. In T. Schnekenburger and G. Stellner (eds.), *Dynamic Load Distribution for Parallel Applications*, pp. 34–47. Teubner-Texte, Leipzig, Germany, 1997.

[11] C. Röder and G. Stellner. Design of Load Management for Parallel Applications in Networks of Heterogeneous Workstations. Tech. Rep. TUM-I9727, SFB 342/18/97 A, SFB 0342, Technische Universität München, 80290 München, Germany, May 1997.

[12] C. Scheidler and L. Schäfers. TRAPPER: A Graphical Programming Environment for Industrial High-Performance Applications. In *Proc. PARLE'93, Parallel Architectures and Languages Europe*, vol. 694 of *Lecture Notes on Computer Science*, pp. 403–413, Berlin, Jun. 1993. Springer-Verlag.

[13] D.W. Walker. The Design of a Standard Message Passing Interface for Distributed Memory Concurrent Computers. *Parallel Computing*, 20(4):657–673, Apr. 1994.

[14] X/Open Group. *Systems Management: Universal Measurement Architecture Guide*, vol. G414. X/Open Company Limited, Reading, UK, Mar. 1995.

SFB 342: Methoden und Werkzeuge für die Nutzung paralleler Rechnerarchitekturen

bisher erschienen :

Reihe A

Liste aller erschienenen Berichte von 1990-1994
auf besondere Anforderung

342/01/95 A Hans-Joachim Bungartz: Higher Order Finite Elements on Sparse Grids

342/02/95 A Tao Zhang, Seonglim Kang, Lester R. Lipsky: The Performance of Parallel Computers: Order Statistics and Amdahl's Law

342/03/95 A Lester R. Lipsky, Appie van de Liefvoort: Transformation of the Kronecker Product of Identical Servers to a Reduced Product Space

342/04/95 A Pierre Fiorini, Lester R. Lipsky, Wen-Jung Hsin, Appie van de Liefvoort: Auto-Correlation of Lag-k For Customers Departing From Semi-Markov Processes

342/05/95 A Sascha Hilgenfeldt, Robert Balder, Christoph Zenger: Sparse Grids: Applications to Multi-dimensional Schrödinger Problems

342/06/95 A Maximilian Fuchs: Formal Design of a Model-N Counter

342/07/95 A Hans-Joachim Bungartz, Stefan Schulte: Coupled Problems in Microsystem Technology

342/08/95 A Alexander Pfaffinger: Parallel Communication on Workstation Networks with Complex Topologies

342/09/95 A Ketil Stølen: Assumption/Commitment Rules for Data-flow Networks - with an Emphasis on Completeness

342/10/95 A Ketil Stølen, Max Fuchs: A Formal Method for Hardware/Software Co-Design

342/11/95 A Thomas Schnekenburger: The ALDY Load Distribution System

342/12/95 A Javier Esparza, Stefan Römer, Walter Vogler: An Improvement of McMillan's Unfolding Algorithm

342/13/95 A Stephan Melzer, Javier Esparza: Checking System Properties via Integer Programming

342/14/95 A Radu Grosu, Ketil Stølen: A Denotational Model for Mobile Point-to-Point Dataflow Networks

342/15/95 A Andrei Kovalyov, Javier Esparza: A Polynomial Algorithm to Compute the Concurrency Relation of Free-Choice Signal Transition Graphs

342/16/95 A Bernhard Schätz, Katharina Spies: Formale Syntax zur logischen Kernsprache der Focus-Entwicklungsmethodik

Reihe A

342/17/95 A Georg Stellner: Using CoCheck on a Network of Workstations

342/18/95 A Arndt Bode, Thomas Ludwig, Vaidy Sunderam, Roland Wismüller: Workshop on PVM, MPI, Tools and Applications

342/19/95 A Thomas Schnekenburger: Integration of Load Distribution into ParMod-C

342/20/95 A Ketil Stølen: Refinement Principles Supporting the Transition from Asynchronous to Synchronous Communication

342/21/95 A Andreas Listl, Giannis Bozas: Performance Gains Using Subpages for Cache Coherency Control

342/22/95 A Volker Heun, Ernst W. Mayr: Embedding Graphs with Bounded Treewidth into Optimal Hypercubes

342/23/95 A Petr Jančar, Javier Esparza: Deciding Finiteness of Petri Nets up to Bisimulation

342/24/95 A M. Jung, U. Rüde: Implicit Extrapolation Methods for Variable Coefficient Problems

342/01/96 A Michael Griebel, Tilman Neunhoeffer, Hans Regler: Algebraic Multigrid Methods for the Solution of the Navier-Stokes Equations in Complicated Geometries

342/02/96 A Thomas Grauschopf, Michael Griebel, Hans Regler: Additive Multilevel-Preconditioners based on Bilinear Interpolation, Matrix Dependent Geometric Coarsening and Algebraic-Multigrid Coarsening for Second Order Elliptic PDEs

342/03/96 A Volker Heun, Ernst W. Mayr: Optimal Dynamic Edge-Disjoint Embeddings of Complete Binary Trees into Hypercubes

342/04/96 A Thomas Huckle: Efficient Computation of Sparse Approximate Inverses

342/05/96 A Thomas Ludwig, Roland Wismüller, Vaidy Sunderam, Arndt Bode: OMIS — On-line Monitoring Interface Specification

342/06/96 A Ekkart Kindler: A Compositional Partial Order Semantics for Petri Net Components

342/07/96 A Richard Mayr: Some Results on Basic Parallel Processes

342/08/96 A Ralph Radermacher, Frank Weimer: INSEL Syntax-Bericht

342/09/96 A P.P. Spies, C. Eckert, M. Lange, D. Marek, R. Radermacher, F. Weimer, H.-M. Windisch: Sprachkonzepte zur Konstruktion verteilter Systeme

342/10/96 A Stefan Lamberts, Thomas Ludwig, Christian Röder, Arndt Bode: PFSLib – A File System for Parallel Programming Environments

342/11/96 A Manfred Broy, Gheorghe Ştefănescu: The Algebra of Stream Processing Functions

342/12/96 A Javier Esparza: Reachability in Live and Safe Free-Choice Petri Nets is NP-complete

Reihe A

342/13/96 A Radu Grosu, Ketil Stølen: A Denotational Model for Mobile Many-to-Many Data-flow Networks

342/14/96 A Giannis Bozas, Michael Jaedicke, Andreas Listl, Bernhard Mitschang, Angelika Reiser, Stephan Zimmermann: On Transforming a Sequential SQL-DBMS into a Parallel One: First Results and Experiences of the MIDAS Project

342/15/96 A Richard Mayr: A Tableau System for Model Checking Petri Nets with a Fragment of the Linear Time μ -Calculus

342/16/96 A Ursula Hinkel, Katharina Spies: Anleitung zur Spezifikation von mobilen, dynamischen Focus-Netzen

342/17/96 A Richard Mayr: Model Checking PA-Processes

342/18/96 A Michaela Huhn, Peter Niebert, Frank Wallner: Put your Model Checker on Diet: Verification on Local States

342/01/97 A Tobias Müller, Stefan Lamberts, Ursula Maier, Georg Stellner: Evaluierung der Leistungsfähigkeit eines ATM-Netzes mit parallelen Programmierbibliotheken

342/02/97 A Hans-Joachim Bungartz and Thomas Dornseifer: Sparse Grids: Recent Developments for Elliptic Partial Differential Equations

342/03/97 A Bernhard Mitschang: Technologie für Parallele Datenbanken - Bericht zum Workshop

342/04/97 A nicht erschienen

342/05/97 A Hans-Joachim Bungartz, Ralf Ebner, Stefan Schulte: Hierarchische Basen zur effizienten Kopplung substrukturierter Probleme der Strukturmechanik

342/06/97 A Hans-Joachim Bungartz, Anton Frank, Florian Meier, Tilman Neunhoeffer, Stefan Schulte: Fluid Structure Interaction: 3D Numerical Simulation and Visualization of a Micropump

342/07/97 A Javier Esparza, Stephan Melzer: Model Checking LTL using Constraint Programming

342/08/97 A Niels Reimer: Untersuchung von Strategien für verteiltes Last- und Ressourcenmanagement

342/09/97 A Markus Pizka: Design and Implementation of the GNU INSEL-Compiler gic

342/10/97 A Manfred Broy, Franz Regensburger, Bernhard Schätz, Katharina Spies: The Steamboiler Specification - A Case Study in Focus

342/11/97 A Christine Röckl: How to Make Substitution Preserve Strong Bisimilarity

342/12/97 A Christian B. Czech: Architektur und Konzept des Dycos-Kerns

342/13/97 A Jan Philipps, Alexander Schmidt: Traffic Flow by Data Flow

342/14/97 A Norbert Fröhlich, Rolf Schlagenhaft, Josef Fleischmann: Partitioning VLSI-Circuits for Parallel Simulation on Transistor Level

Reihe A

342/15/97 A Frank Weimer: DaViT: Ein System zur interaktiven Ausführung und zur Visualisierung von INSEL-Programmen

342/16/97 A Niels Reimer, Jürgen Rudolph, Katharina Spies: Von FOCUS nach INSEL - Eine Aufzugssteuerung

342/17/97 A Radu Grosu, Ketil Stølen, Manfred Broy: A Denotational Model for Mobile Point-to-Point Data-flow Networks with Channel Sharing

342/18/97 A Christian Röder, Georg Stellner: Design of Load Management for Parallel Applications in Networks of Heterogenous Workstations

342/19/97 A Frank Wallner: Model Checking LTL Using Net Unfoldings

342/20/97 A Andreas Wolf, Andreas Kmoch: Einsatz eines automatischen Theorembeweisers in einer taktikgesteuerten Beweisumgebung zur Lösung eines Beispiels aus der Hardware-Verifikation – Fallstudie –

342/21/97 A Andreas Wolf, Marc Fuchs: Cooperative Parallel Automated Theorem Proving

342/22/97 A T. Ludwig, R. Wismüller, V. Sunderam, A. Bode: OMIS - On-line Monitoring Interface Specification (Version 2.0)

342/23/97 A Stephan Merkel: Verification of Fault Tolerant Algorithms Using PEP

342/24/97 A Manfred Broy, Max Breitling, Bernhard Schätz, Katharina Spies: Summary of Case Studies in Focus - Part II

342/25/97 A Michael Jaedicke, Bernhard Mitschang: A Framework for Parallel Processing of Aggregat and Scalar Functions in Object-Relational DBMS

342/26/97 A Marc Fuchs: Similarity-Based Lemma Generation with Lemma-Delaying Tableau Enumeration

342/27/97 A Max Breitling: Formalizing and Verifying TimeWarp with FOCUS

342/28/97 A Peter Jakobi, Andreas Wolf: DBFW: A Simple DataBase Frame-Work for the Evaluation and Maintenance of Automated Theorem Prover Data (incl. Documentation)

342/29/97 A Radu Grosu, Ketil Stølen: Compositional Specification of Mobile Systems

342/01/98 A A. Bode, A. Ganz, C. Gold, S. Petri, N. Reimer, B. Schiemann, T. Schnekenburger (Herausgeber): „Anwendungsbezogene Lastverteilung“, ALV'98

SFB 342 : Methoden und Werkzeuge für die Nutzung paralleler Rechnerarchitekturen

Reihe B

342/1/90 B Wolfgang Reisig: Petri Nets and Algebraic Specifications

342/2/90 B Jörg Desel: On Abstraction of Nets

342/3/90 B Jörg Desel: Reduction and Design of Well-behaved Free-choice Systems

342/4/90 B Franz Abstreiter, Michael Friedrich, Hans-Jürgen Plewan: Das Werkzeug runtime zur Beobachtung verteilter und paralleler Programme

342/1/91 B Barbara Paech1: Concurrency as a Modality

342/2/91 B Birgit Kandler, Markus Pawlowski: SAM: Eine Sortier- Toolbox -Anwenderbeschreibung

342/3/91 B Erwin Loibl, Hans Obermaier, Markus Pawlowski: 2. Workshop über Parallelisierung von Datenbanksystemen

342/4/91 B Werner Pohlmann: A Limitation of Distributed Simulation Methods

342/5/91 B Dominik Gomm, Ekkart Kindler: A Weakly Coherent Virtually Shared Memory Scheme: Formal Specification and Analysis

342/6/91 B Dominik Gomm, Ekkart Kindler: Causality Based Specification and Correctness Proof of a Virtually Shared Memory Scheme

342/7/91 B W. Reisig: Concurrent Temporal Logic

342/1/92 B Malte Grosse, Christian B. Suttner: A Parallel Algorithm for Set-of-Support
Christian B. Suttner: Parallel Computation of Multiple Sets-of-Support

342/2/92 B Arndt Bode, Hartmut Wedekind: Parallelrechner: Theorie, Hardware, Software, Anwendungen

342/1/93 B Max Fuchs: Funktionale Spezifikation einer Geschwindigkeitsregelung

342/2/93 B Ekkart Kindler: Sicherheits- und Lebendigkeitseigenschaften: Ein Literaturüberblick

342/1/94 B Andreas Listl; Thomas Schnekenburger; Michael Friedrich: Zum Entwurf eines Prototypen für MIDAS