Angewandte Mathematik und Informatik Universität zu Köln

Report No. 96.219

Precomputation based Load Balancing

by

Max Böhm, Ewald Speckenmeyer

1996

submitted for publication

Universität zu Köln
Institut für Informatik
Pohligstr. 1
D-50969 Köln

1991 Mathematics Subject Classification: 68Q22, 68Q25, 68T20
Keywords: load balancing, distributed computing, satisfiability problem

Precomputation based Load Balancing

Max Böhm, Ewald Speckenmeyer
Institut für Informatik
Universität zu Köln
Pohligstr. 1, D–50969 Köln
e–mail: boehm@informatik.uni–koeln.de, esp@informatik.uni–koeln.de

Abstract

An algorithm, called PLB is introduced, which redistributes workload in a processor network N in order to supply every processor of N with (about) the same amount of workload. PLB is defined in its basic form for trees, but can be extended to other topologies. The redistribution is done locally on the basis of information of over- or underload in subnetworks of N. We show, that PLB performs $O(\Delta)$ steps, only, where Δ denotes the diameter of N, and in the average case at most four times as many workload has to be migrated in complete binary trees compared to clique networks, the best possible networks. We describe an implementation of PLB and present experimental results when solving the Boolean satisfiability problem, demonstrating that PLB performs very well in practice.

1 Introduction

In this paper we introduce a simple load balancing algorithm, which is defined in its basic formulation for processor trees. It can easily be extended to processor networks built up as the cross product of trees, see [18, 400pp]. This class contains d-dimensional grids or hypercubes, e.g. Suppose $T = (V, E)$ to be a processor tree with n processors as nodes. The edges u — v of E represent the links of the network. When solving an instance I of some problem on the distributed network T, the load distribution among the processors of T at time t of the computation is expressed by a function $\lambda_t : V \rightarrow \mathbb{R}^{\geq 0}$, where $\lambda_t(v)$ denotes the amount of workload associated with processor v at time t. In order to hold up the processors busy all the time, it is convenient to redistribute the workload among the processors, such that $\lambda_t(u) = \lambda_t(v)$ holds, for all $u, v \in V$, which is called a *uniform load distribution*. Achieving a uniform load distribution may be NP-hard, if workload is not divisible. In case however, that workload can be divided arbitrarily, the problem of achieving a uniform load distribution always has a solution, which can be determined efficiently. For our analytical considerations throughout this paper we meet the assumption that workload is divisible arbitrarily. In real applications when solving hard combinatorial optimization problems on the basis of solution technics like Backtracking, Branch and Bound or others, e.g., this assumption is violated. But then we typically are able to divide efficiently pieces of load, i.e. single subproblems are split into several smaller

subproblems, such that we can "approximate sufficiently close" a uniform load distribution in practice, e.g. see [1]. Furthermore in case of NP-hard problems the major difficulty of determining $\lambda_t(v)$ precisely has to be dealt with, because determinig $\lambda_t(v)$ may be as hard as solving the original problem. Therefore we use heuristic estimations of $\lambda_t(v)$. For this reason it is necessary to rebalance the workload among the processors of T not only once at the beginning of the computation, but dynamically at run time when single processors have become idle or run the risk of becoming idle very soon.

In section 3 we introduce our load balancing algorithm, called *Precomputation based Load Balancing* (PLB). PLB redistributes the workload among the processors of the tree T in two phases. In the first phase each processor v determines the total amount of workload in the subtree T_v of T with root v and the total amount of load T_v has to send to or to receive from the remaining network in order to achieve a uniform load distribution for T. This information is associated as a real number $\delta(v, father(v))$ with the link between v and the father of v in the network. $\delta(v, father(v)) > 0$ denotes the amount of load sent from T_v to the remaining network in case that T_v contains more load than necessary under a uniform load distribution. $\delta(v, father(v)) < 0$ denotes the corresponding amount of work T_v has to receive from the remaining network in order to achieve a uniform load distribution. Phase 1 can be computed in $O(\Delta)$ units of time, where Δ denotes the diameter of T. Note that no load is moved in this phase. In the second phase load will be distributed locally on the basis of the information from the first phase. The phase is performed in several rounds. As long as a uniform load distribution has not yet appeared, each processor which has to send workload to neighboured processors sends as much of his load as possible via the connecting links, but not more than the current δ-values for these links indicate. Update the δ-values.

The number of rounds performed in the second phase varies between 0 and Δ in the worst case, see Theorem 1. In experiments with PLB we observed that this number of rounds is less than Δ most of the time. Now the question arises for the expected number of rounds performed in the second step. This number is important especially in applications where large packages of workload have to be migrated. For the special case of a linear processor array this number can be shown to grow like $\sqrt{\Delta}$ instead of Δ in the worst case, see [1]. The underlying model assigns wokload $\lambda_t(v)$ to processor v by a random variable X_v with expectation μ and standard deviation σ. The X_v are considered to be independent and identically distributed for every processor v.

On the basis of this model we study in section 4 the problem of determining the expected amount of workload sent over all links of the network summed up for all rounds of phase 2 of PLB. In case of a linear processor array the expected migrated load is of order $\Theta(n\sqrt{n}\,\sigma)$, see section 4.2, while in case of a complete binary tree the expected migrated load is of order $\Theta(n\,\sigma)$, only. This amount should be compared with $\Theta(n\,\sigma)$, the expected amount of load that has to be

migrated in a clique network, which yields a lower bound for every network, because no routing is necessary in clique networks. More precisely, the expected amount of workload that has to be migrated by PLB in a complete binary tree is at most 4 times as high as in every other network, if X_v is distributed normally with parameters μ and σ, or uniformly distributed in $[a, b]$, see section 4.3. In the worst case the amount of load that has to be migrated in the complete binary tree, amounts to $\Theta(n \log n \, \lambda_{mid})$, where λ_{mid} denotes the sum over the absolute values of the difference between $\lambda_t(v)$ and the workload of processor v after uniform load distribution has been achieved for T. It can be shown that the expectation of λ_{mid} under the above mentioned distributions is of order $\Theta(\sigma)$. I.e., the expected amount of workload that has to be migrated in a complete binary processor tree is by a factor of $\Theta(\frac{1}{\log n})$ less than in the worst case. On the other hand it is at most 4 times as high as in the clique network, which shows that complete binary trees are optimal networks up to a constant factor.

We want to mention that in case where the workload estimation $\lambda_t(v)$ and the encodings of the workload at v are linearly related, then the above considerations yield the total communication cost caused by PLB. Often however, this relationship does not hold.

In section 5 we describe our implementation of PLB and in section 6 we report experimental results with PLB when solving the boolean satisfiability problem on a distributed network. Our experiments demonstrate, that PLB performs the redistribution of workload very efficiently. Mean idle times of a few seconds per processor have been observed in experiments, only, even for networks of 1024 nodes.

2 Related Work

Dynamic load balancing has been considered in different contexts, e.g. scheduling and migration of tasks in operating systems, see [5, 19, 26, 31], distributed solving of combinatorial optimization problems, see [2, 3, 22, 23, 27, 28, 34], distributed solving of problems in the area of scientific computing, see [16, 25, 33, 35], and others. Several classification schemes were proposed, see [5, 24], e.g. local vs. global decision and local vs. global migration, centralized vs. distibuted algorithms. Theoretical properties of load balancing algorithms were studied, e.g. scalability and isoefficiency, see [10, 17]. Different diffusion and dimension exchange methods were analyzed, see [8, 12, 14, 15, 21, 36, 37].

3 The PLB Algorithm

In this paper we consider network connected *asynchronous message passing MIMD* systems. The network is represented by an undirected graph $G = (V, E)$. Each

node $v \in V$ is identified with a processor. Two processors u, v are connected by a bidirectional communication link if and only if $u - v \in E$.

When solving a problem in parallel, at time t every processor v holds a set of subproblems representing a certain amount of *workload* $\lambda_t(v) \geq 0$. $\lambda_t(v)$ denotes the running time required by processor v in order to complete the computation of the subproblems held by v. The function $\lambda_t : V \to \mathbb{R}^{\geq 0}$ is called a *load distribution*. $\overline{\lambda_t} := \frac{1}{|V|} \sum_{v \in V} \lambda_t(v)$ is called the *mean load*. If time t is fixed, we write $\lambda(v)$ and $\overline{\lambda}$ as a shorthand of $\lambda_t(v)$ and $\overline{\lambda}_t$ respectively. When solving NP-hard combinatorial optimization problems in parallel the load distribution often changes in an unpredictable way. The load has to be rebalanced dynamically to avoid idle times of processors (*load balancing*). We assume that each processor is able to partition its load into arbitrary small pieces. A piece can be sent over a link to a neighboured processor. Let $\vec{G} = (V, \vec{E})$ 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

$$\begin{aligned} & \text{a)} \quad \delta(u \to v) \;=\; -\delta(u \leftarrow v) \qquad \text{for all } u - v \in E \\ \text{and} \quad & \text{b)} \quad \lambda(v) + \sum_{u \to v \in \vec{E}} \delta(u \to v) \;=\; \overline{\lambda} \quad \text{for all } v \in V \end{aligned}$$

a *load balancing scheme*. Load balancing is performed by moving $\delta(u \to v)$ units of load from processor u to processor v for each $u - v \in E$ with $\delta(u \to v) > 0$.

The *Precomputation based Load Balancing* (PLB) algorithm, see [1, 30], performs load balancing in processor trees as follows. Let $T = (V, E)$ be a processor tree with n processors of height h and root $r \in V$. We assume that the degree of each node $v \in V$ is bounded by a constant. PLB works in two phases:

3.1 Precomputation Phase

In the *Precomputation Phase* the following steps are performed:

1. Each processor $v \in V$ calculates $subtreesum(v) := \sum_{u \in V(T_v)} \lambda(u)$, where T_v denotes the subtree of T rooted at v. This can be done in h parallel steps by using the recurrence

$$subtreesum(v) = \lambda(v) \; + \sum_{u \in sons(v)} subtreesum(u).$$

2. The root r of T calculates the *mean load* $\overline{\lambda} := \frac{subtreesum(r)}{n}$ which is broadcast to all processors. The broadcast operation can be executed in h parallel steps.

3. Each processor $v \in V \setminus \{r\}$ calculates

$$\begin{aligned} \delta(v \to father(v)) &:= subtreesum(v) \;-\; |V(T_v)| \cdot \overline{\lambda} \quad \text{and} \\ \delta(v \leftarrow father(v)) &:= -\delta(v \to father(v)). \end{aligned}$$

Note that in the Precomputation Phase no load migration takes place. Only two messages are sent along each link. Since T is a tree the total work of the Precomputation Phase is $work(n) = O(n)$ and the parallel time is $time(n) = O(\Delta)$, where Δ denotes the diameter of T.

3.2 Balancing Phase

In the *Balancing Phase* the actual load migration according to δ is performed in several *rounds*. In each round each processor u sends as much load as possible to each neighbour v with $\delta(u \to v) > 0$ such that (1) at most $\delta(u \to v)$ units of load are sent from u to v (accumulated) and (2) the total amount of load sent by u does not increase the load which u held at the beginning of the round. Note that this strategy is nondeterministic. However, the number of rounds is limited by Δ.

Theorem 1: *The number of rounds required by PLB to achieve a uniform load distribution among the processors of a processor tree is bounded by its diameter* Δ.

Proof: $v_1 - v_2 - \cdots - v_k$ with $\delta(v_i \to v_{i+1}) > 0$ is called a *chain*. (1) The length of any chain is bounded by the diameter of the network. (2) Since v_1 of a maximal chain does not receive any load it is able to send the total required amount of load to all of its neighbours in this round. Therefore the length of a maximal chain is reduced by at least one in each round. □

The total work of the Balancing Phase is $work(n) = O(\Delta\, n)$ and the parallel time is $time(n) = O(\Delta)$.

PLB is defined for processor trees. It can be extended to d-dimensional topologies using a modified *dimension exchange* method, see [1, 8]. The strategie is described in section 5 for the 2-dimensional mesh.

4 Average Case Analysis of PLB

In this section we analyse the expected total amount of load sent via all links of the processor tree during the active phases by PLB until a uniform load distribution has been achieved.

Let $V = \{v_1, \ldots, v_n\}$ be a set of n processors and $X_1, \ldots, X_n$ be *independent and identically distributed* (i.i.d.) random variables each having mean μ and standard deviation σ. We consider a random load distribution defined by $\lambda(v_i) = X_i$, $1 \le i \le n$.

Let $T = (V, E)$ be a tree. The amount of load $\delta(u \to v)$ which has to be moved from u to v depends on the nodes of the two connected components T_u and T_v of the graph $T \setminus u - v$ only, but not on the topology of T_u and T_v. Without

loss of generality we therefore can assume a linear array topology, see figure 1. We define random variables

$$D_{n,i} \;:=\; \delta(v_i \to v_{i+1}) \;=\; \sum_{j=1}^{i} X_j \;-\; \frac{i}{n}\sum_{j=1}^{n} X_j. \qquad (1)$$

The expectation $E[|D_{n,i}|]$ is the expected amount of load migration over an edge $u \text{ — } v \in E$ with $|V(T_u)| = i$ and $|V(T_v)| = n - i$ for any tree $T = (V, E)$.

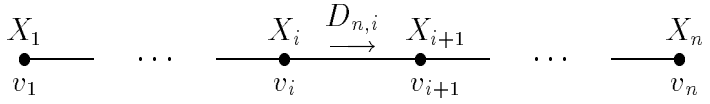

Figure 1: Random variables $X_1, \ldots, X_n$ and $D_{n,i}$ in a linear array.

4.1 Expected Load Migration for an Edge

We determine the asymptotic distribution of $D_{n,i}$ for every fixed $0 < \alpha < 1$ and $i = \alpha n$ for $n \to \infty$. The random variable

$$D_{n,i} \;=\; D_{n,\alpha n} \;\stackrel{(1)}{=}\; (1-\alpha)\sum_{j=1}^{i} X_j \;-\; \alpha \sum_{j=i+1}^{n} X_j \;=:\; \sum_{j=1}^{n} Y_j$$

is a linear combination of $X_1, \ldots, X_n$. The *Central Limit Theorem*, tells us that a sum of independent random variables $Y_1, Y_2, \ldots$ is asymtotically *normal* distributed, if *Lindeberg's* equation holds, see [11, 20]. Then

$$P\left\{ \frac{(\sum_{i=1}^{n} Y_i) - E[\sum_{i=1}^{n} Y_i]}{\sqrt{Var(\sum_{i=1}^{n} Y_i)}} \le a \right\} \;\to\; \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{a} e^{-x^2/2}\, dx \;=:\; \Phi(a)$$

for $n \to \infty$. In our case Lindeberg's equation holds, since $X_1, X_2, \ldots$ are identically distributed. Easy calculation leads to

$$E[D_{n,\alpha n}] \;=\; 0 \qquad \text{and} \qquad Var(D_{n,\alpha n}) \;=\; (1-\alpha)\alpha n \sigma^2 \;=:\; \tilde{\sigma}^2_{n,\alpha}.$$

We use the central limit theorem as described above to derive the asymptotic distribution of $D_{n,\alpha n}$ for $0 < \alpha < 1$. We have

$$P\left\{ \frac{D_{n,\alpha n} - E[D_{n,\alpha n}]}{\sqrt{Var(D_{n,\alpha n})}} \le a \right\} \;=\; P\left\{ \frac{D_{n,\alpha n}}{\tilde{\sigma}_{n,\alpha}} \le a \right\} \;\to\; \Phi(a)$$

$$\iff \qquad F_{D_{n,\alpha n}}(\tilde{\sigma}_{n,\alpha} a) \;\to\; \Phi(a) \qquad \text{for } n \to \infty. \qquad (2)$$

With the notation

$$f(n) \sim g(n) \quad \Longleftrightarrow \quad \lim_{n\to\infty} \frac{f(n)}{g(n)} = 1,$$

see [13], the asymptotic expected load migration over an edge v_i — v_{i+1} with $i = \alpha n$ and $0 < \alpha < 1$ for $n \to \infty$ is

$$\begin{aligned}
E[|D_{n,\alpha n}|] &= \int_{-\infty}^{\infty} |x| \frac{d}{dx} F_{D_{n,\alpha n}}(x)\, dx \\
&\overset{(2)}{\sim} \int_{-\infty}^{\infty} |x| \frac{d}{dx} \Phi\left(\frac{x}{\tilde{\sigma}_{n,\alpha}}\right) dx \\
&= \frac{2}{\sqrt{2\pi}\tilde{\sigma}_{n,\alpha}} \underbrace{\int_{0}^{\infty} x e^{-x^2/2\tilde{\sigma}_{n,\alpha}^2}\, dx}_{=\tilde{\sigma}_{n,\alpha}^2} \\
&= \sqrt{\frac{2}{\pi}}\tilde{\sigma}_{n,\alpha} = \sqrt{\frac{2(1-\alpha)\alpha n}{\pi}}\,\sigma = \Theta(\sqrt{n}\,\sigma). \qquad (3)
\end{aligned}$$

The expected load to be migrated over such an "inner" edge $v_i - v_{i+1}$ is $\Theta(\sqrt{n}\,\sigma)$. Note that this result is independent of the distribution of the random variables X_i (it depends only on their standard deviation σ).

4.2 Expected Load Migration for the Linear Array

In the *linear array* LA_n with n processors the expected sum of migrated loads is

$$\begin{aligned}
E[M(LA_n, \lambda)] &= \sum_{i=1}^{n-1} E[|D_{n,i}|] \\
&\sim n \int_0^1 E[|D_{n,\alpha n}|]\, d\alpha \qquad (\alpha = \frac{i}{n}) \\
&\overset{(3)}{\sim} n \int_0^1 \sqrt{\frac{2(1-\alpha)\alpha n}{\pi}}\,\sigma\, d\alpha \qquad \left(\int_0^1 \sqrt{\alpha - \alpha^2}\, d\alpha = \tfrac{\pi}{8}\right) \\
&= \frac{\sqrt{2\pi n}}{8}\, n\sigma = \Theta(n\sqrt{n}\,\sigma). \qquad (4)
\end{aligned}$$

This means that the expected sum of migrated loads in a linear array with n processors is $\Theta(n\sqrt{n}\,\sigma)$.

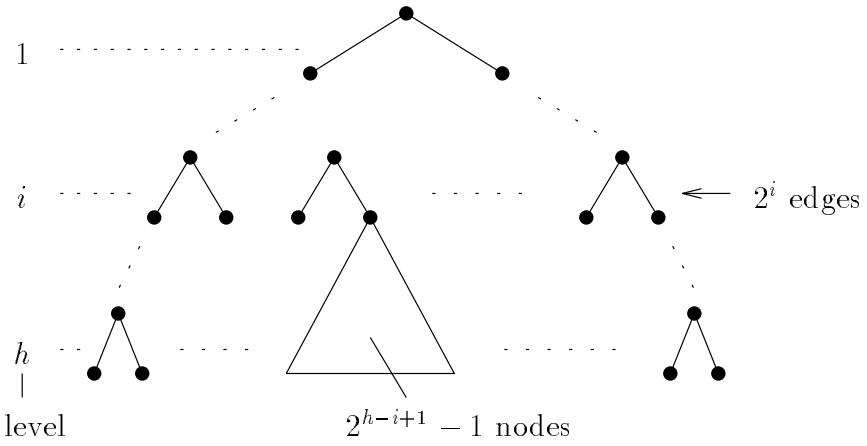

Figure 2: Expected load migration for an edge at level i is $E[|D_{n,2^{h-i+1}-1}|]$.

4.3 Expected Load Migration for the Complete Binary Tree

Let $T_{h,2}$ be the *complete binary tree* of height h with $n = 2^{h+1} - 1$ processors. We calculate the expected sum of migrated loads $E[M(T_{h,2}, \sigma)]$. For reasons of symmetrie each of the 2^i edges at level i, $1 \leq i \leq h$, have identical expected load migration. Each of these edges connects a subtree of height $h-i$ with $2^{h-i+1}-1$ processors with the rest of the graph, see figure 2. The expected sum of migrated loads is

$$
\begin{aligned}
E[M(T_{h,2}, \lambda)] &= \sum_{i=1}^{h} 2^i E[|D_{n,2^{h-i+1}-1}|] \\
&\sim \int_0^h 2^x E[|D_{n,2^{h-x+1}-1}|]\, dx \\
&\overset{(3)}{\sim} \int_0^h 2^x \sqrt{\frac{2\,(1-2^{-x})\,2^{h-x+1}}{\pi}}\,\sigma\, dx \\
&\sim \frac{2\,\sigma}{\sqrt{\pi}} \int_0^h \sqrt{2^{h+x}}\, dx \\
&= \frac{4\left(2^h - \sqrt{2^h}\right)}{\sqrt{\pi}\,\ln 2}\,\sigma \\
&\sim \frac{2\,n}{\sqrt{\pi}\,\ln 2}\,\sigma \approx 1.63\,n\,\sigma = \Theta(n\,\sigma). \qquad (5)
\end{aligned}
$$

The expected sum of migrated loads in the complete binary tree is less than that in the linear array. Now lets consider the complete graph K_n with n processors. Load balancing can be performed by moving $\delta(u \to v)$ units of load directly from

u to v for each edge u — v with $\delta(u \to v) > 0$. Therefore we have

$$E[M(K_n, \lambda)] \;=\; \frac{n}{2}\, E[|X - E[X]|] \;=\; c\, n\, \sigma. \qquad (c \in \mathbb{R}) \tag{6}$$

This is a lower bound of the expected sum of migrated loads for any network topology. The constant c depends on the distribution of the random variables, e.g. if the X_i are uniformly distributed in $[a, b]$ we have $E[|X - E[X]|] = \frac{b-a}{4}$ and $\sigma = \frac{b-a}{\sqrt{12}}$. Then it is

$$E[M(K_n, \lambda)] \;=\; \frac{\sqrt{3}}{4}\, n\, \sigma \quad \Rightarrow \quad \frac{E[M(T_{h,2}, \lambda)]}{E[M(K_n, \lambda)]} \;\stackrel{n\to\infty}{\longrightarrow}\; \frac{8}{\sqrt{3\pi}\ln 2} \;\approx\; 3.76.$$

If the random variables are normal disributed with parameters μ and σ, we have

$$E[M(K_n, \lambda)] \;\sim\; \frac{1}{\sqrt{2\pi}}\, n\, \sigma \quad \Rightarrow \quad \frac{E[M(T_{h,2}, \lambda)]}{E[M(K_n, \lambda)]} \;\stackrel{n\to\infty}{\longrightarrow}\; \frac{2\sqrt{2}}{\ln 2} \;\approx\; 4.08.$$

Note that the expected sum of migrated loads in a complete binary tree is asymptotically within a constant factor of that in the complete graph.

4.4 Upper Bounds of Load Migration

Let $\lambda : V \to \mathbb{R}^{\geq 0}$ be a *load distribution*. As a measure of load imbalance we define

$$\lambda_{mid} \;:=\; \frac{1}{n} \sum_{v \in V} |\lambda(v) - \overline{\lambda}|.$$

Upper bounds of the sum of migrated loads can be expressed in terms of n and λ_{mid}. The sum of migrated loads in the complete graph is

$$M(K_n, \lambda) \;=\; \frac{1}{2} \sum_{v \in V} |\lambda(v) - \overline{\lambda}| \;=\; \frac{n}{2}\lambda_{mid} \;=\; \Theta(n\, \lambda_{mid}). \tag{7}$$

Let $T = (V, E)$ be a tree with $|V| = n$ processors and diameter Δ. As can easily be verified, the sum of migrated loads in T is bounded by

$$M(T, \lambda) \;\leq\; \Delta\, M(K_n, \lambda) \;=\; \Delta\, \frac{n}{2}\lambda_{mid}. \tag{8}$$

$$\Rightarrow \qquad M(LA_n, \lambda) \;=\; O(n^2\, \lambda_{mid}), \tag{9}$$

$$M(T_{h,2}, \lambda) \;=\; O(n\, \log n\, \lambda_{mid}) \qquad (n = 2^{h+1} - 1) \tag{10}$$

In table 1 we summarize the average case results of the previous sections and give corresponding sharp upper bounds. Note that $E[\lambda_{mid}] = \Theta(\sigma)$ for every fixed class of random distributions (e.g. for the uniform distribution we have $E[\lambda_{mid}] = \frac{\sqrt{3}}{2}\sigma$). Therefore the bounds are comparable. The bounds for the hypercube and mesh topologies are related to the d-dimensional PLB algorithm, briefly introduced in section 5. For a derivation of these bounds see [1].

topology G	processors	$E[M(G,\lambda)]$	$M(G,\lambda)$
complete graph K_n	n	$\Theta(n\,\sigma)$ (6)	$\Theta(n\,\lambda_{mid})$ (7)
linear array LA_n	n	$\Theta(n\sqrt{n}\,\sigma)$ (4)	$O(n^2\,\lambda_{mid})$ (9)
complete binary tree $T_{h,2}$	$n = 2^{h+1}-1$	$\Theta(n\,\sigma)$ (5)	$O(n \log n\,\lambda_{mid})$ (10)
tree T with diameter Δ	$n = \|V\|$	$O(n\sqrt{n}\,\sigma)$ (4)	$O(n\,\Delta\,\lambda_{mid})$ (8)
hypercube HC_d	$n = 2^d$	$\Theta(n\,\sigma)$	$O(n\ \log n\,\lambda_{mid})$
d-dimensional mesh	$n = m^d$	$\Theta(n\ \sqrt[2d]{n}\,\sigma)$	$O(d\,n\ \sqrt[d]{n}\,\lambda_{mid})$

Table 1: Average and worst case bounds of load migration.

5 Implementation

We have implemented the PLB algorithm in C. For the $m \times m$ mesh with $n = m^2$ processors we use a modified algorithm which first balances all linear arrays $v_{i,1} - v_{i,2} - \cdots - v_{i,m}$ in parallel and then all linear arrays $v_{1,j} - v_{2,j} - \cdots - v_{m,j}$ in parallel, $1 \le i, j \le m$, see figure 3. For flexibility and portability reasons we

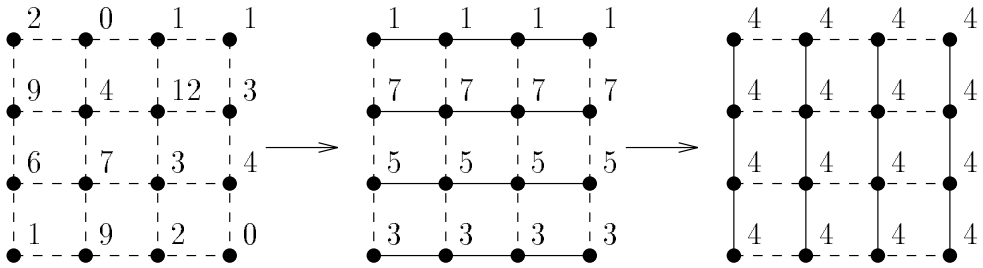

Figure 3: Load balancing in the 2-dimensional mesh.

have defined an abstract interface between the load balancing module and the problem solving module. Each processor v manages a list L_v of subproblems. The load $\lambda(v)$ of processor v is the sum of loads $\lambda(p)$ of the subproblems p in L_v, i.e. $\lambda(v) = \sum_{p \in L_v} \lambda(p)$. A *worker thread* and a *balancer thread* are operating on the list. The worker thread performs the following steps:

while not terminated **do**
  get a subproblem p of L_v
  if $\lambda(p) < \beta\,\overline{\lambda}$ **then** solve p
    else split p into $p_1, \ldots, p_k$
      insert $p_1, \ldots, p_k$ in L_v
  endif
end

For efficiency reasons a subproblem is solved sequentially if its estimated load is less than $\beta\,\overline{\lambda}$. We have chosen $\beta = 0.05$, which turned out to be a good choice, in our experiments.

The balancer thread periodically performs the PLB algorithm. In the precomputation phase the load balancing scheme is calculated and termination detection is done. The balancing phase is actually performed if a processor runs out of work, only. When a balancing phase is activated a processor v, which has to send the load l to a neighboured processor w, executes the following greedy strategy:

while $l > \gamma\,\overline{\lambda}$ **do**
    get a subproblem p of L_v
    send p to processor w
    $l := l - \lambda(p)$
end

Processor v sends problems to w until at least $l - \gamma\,\overline{\lambda}$ units of load have been migrated. We have chosen $\gamma = 0.5$. Since the loads of subproblems are estimated only, an exact load balancing is not neccessary. Organizing load balancing according to this implementation will not lead to a uniform load distribution, in general. Experiments however have demonstrated that it results in a load distribution preventing processors from getting idle besides two or three seconds during the start and the end-phase of the run time.

6 Experimental Results

We have chosen the *Satisfiability Problem* (SAT) as demonstration problem. Input of the SAT problem is a Boolean formula in *conjunctive normal form* (CNF). Output is *true*, if a truth assignment of the variables exists that satisfies the formula, and *false* otherwise. The problem was shown to be NP-complete, see [6]. It can be solved by a backtracking search. We have implemented a version of the *Davis-Putnam* algorithm improved by a special branching heuristic for variable selection with efficiently implemented data structures, see [3, 4]. A node in the search tree is representing a subproblem p. p is encoded by the partial truth assignment of variables. In the branching step the, Boolean formula F_p which corresponds to p is simplified as much as possible by using the *unit clause rule*. Then an unassigned boolean variable is chosen, for which the number of occurences in the shortest clauses of the simplified formula is maximal. If there are several candidates, the number of occurences in clauses of second smallest clause length are used to determine the varibale and so on. We called this heuristic *lexicographic heuristic*. By assigning the chosen variable *true* and *false* we get the two subproblems p_1 resp. p_2. The lexicographic heuristic turned out to be very effective when solving the SAT problem for random formulas of the fixed clause length model, see [4]. We used the following two load estimation functions:

(a) Let k be the number of all assigned variables of subproblem p (this includes variables assigned by the unit clause rule). The load of p is approximated by $\lambda(p) = \alpha^{-k}$ $(\alpha > 1)$.

(b) The load of each subproblem p is set $\lambda(p) = 1$, i.e. the load of a processor is the length of its list.

The PLB algorithm was tested on a message passing MIMD system built of 32×32 T805/30Mhz transputers (GCel1024). We generated two samples of random 3-SAT formulas each of 50 instances named 400-2000-3 (400 variables 2000 clauses) and 350-1505-3 (350 variables, 1505 clauses). All 50 instances of the first sample are unsatisfiable. 31 (19) instances are unsatisfiable (satisfiable) of the second sample. Instances of these classes turned out to be hard to solve for Davis-Putnam based SAT-solvers from an experimental point of view. Currently only some fast implementations of SAT-solvers can recognize unsatisfiablility of instances of these classes in reasonable time, see [3, 4, 7, 9]. We experimentally determined $\alpha = 1.04$ for the load estimation (a).

6.1 Scalability of PLB

The results for solving the unsatisfiable instances of the sample 350-1505-3 on a mesh of n processors are given in table 2. We measured the parallel running time (*time*) and standard deviation (std), the mean idle time per processor (*idle*), the number of balancing phases (*bal*), the mean number of sent subproblems per processor (*send*) and the efficiency. The results are averaged on the unsatisfiable instances of the samples. Satisfiable instances are not taken into account, since the sizes of the search trees of the parallel and the sequential algorithm differs. Superlinear speedups are possible, see [29].

n	*time* (std)	*idle*	*bal*	*send*	*efficiency*
$4 \times 8 = 32$	3217s (1361s)	4.8s	252	99	99.3%
$8 \times 8 = 64$	1614s (681s)	4.0s	325	135	99.0%
$8 \times 16 = 128$	810s (341s)	3.2s	336	203	98.6%
$16 \times 16 = 256$	408s (171s)	2.7s	312	184	97.8%
$16 \times 32 = 512$	207s (86s)	3.2s	210	218	96.4%
$32 \times 32 = 1024$	108s (43s)	4.4s	110	244	92.8%

Table 2: Results for 31 unsatisfiable instances of 350-1505-3.

6.2 Comparison of PLB and Local Averaging Strategies

To demonstrate the efficiency of PLB we solved the sample 400-2000-3 on different network topologies, the 16×16 mesh and the linear array of 256 processors. We give results for both load estimation functions (a) and (b). The results are summarized in table 3. PLB performs load balancing very fast, even in networks with large diameter. The mean number of migrated problems per processor is extremely low for the mesh topology. It increases in case of the linear array.

topologie	load est.	*time*	*idle*	*bal*	*send*	*efficiency*
mesh	(a)	121.2s	2.4s	277	206	95.0%
	(b)	128.5s	4.3s	426	194	89.7%
linear array	(a)	121.2s	1.7s	118	1134	95.0%
	(b)	130.3s	3.5s	210	2294	88.4%

Table 3: Results for the PLB algorithm (sample 400-2000-3, $n = 256$)

For comparison with *Local Averaging* load balancing strategies, we have parallelized our SAT-solver using an alternative load balancing tool, the PPBB-lib, see [32]. This library is developed at the University of Paderborn. It offers a number of local decision/local migration load balancing strategies. The stategie we have chosen (LADE), works as follows: Each processor stores information on the number of subproblems of its neighbours. A processor informs its neighbours if its number of subproblems changes by more than 10%. A processor periodically balances the number of subproblems with its least loaded neighbour. PPBB-lib running on PARIX supports virtual topologies used by the load balancing algorithm which are mapped on the physical 2-dimensional mesh of processors. The results are given in table 4. The mean number of sent load informations per processor (*info*) and the mean number of sent subproblems per processor (*send*) are reported. The efficiency is acceptable as long as the diameter of the network is low.

virt. topologie	*time*	*idle*	*info*	*send*	*efficiency*
mesh	141.7s	3.7s	3025	2446	81.3%
DeBruijn	142.4s	2.4s	3216	2557	80.9%
hypercube	168.1s	2.8s	9263	6147	68.5%
ring	340.9s	210.2s	1511	1362	33.8%
torus	141.3s	2.5s	3252	2583	81.5%

Table 4: *Local Averaging* algorithm of PPBB-lib (sample 400-2000-3, $n = 256$).

References

[1] M. Böhm. *Verteilte Lösung harter Probleme: Schneller Lastausgleich.* PhD thesis, Universität zu Köln, 1996.

[2] M. Böhm and E. Speckenmeyer. A dynamic processor tree for solving game trees in parallel. In *Methods of Operations Research*, volume 63, pages 479–489, 1989.

[3] M. Böhm and E. Speckenmeyer. A fast parallel SAT–solver — efficient workload balancing. Technical Report 94-159, Angewandte Mathematik und Informatik, Universität zu Köln, 1994. accepted for publication in: Annals of Mathematics and Artificial Intelligence.

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

[5] T. L. Casavant and J. G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engineering*, SE-14:141–154, 1988.

[6] 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.

[7] 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.

[8] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *Journal of Parallel and Distributed Computing*, 7(2):279–301, October 1989.

[9] O. Dubois, P. Andre, Y. Boufkhad, and J. Carlier. Can a very simple algorithm be efficient for solving the sat problem? Technical report, DIMACS, Rutgers University, New Brunswick, NJ 08903, 1993.

[10] S. Dutt and N. R. Mahapatra. Scalable load balancing strategies for parallel A* algorithms. *Journal of Parallel and Distributed Computing*, 22:488–505, 1994.

[11] W. Feller. *An Introduction to Probability Theory and Its Applications*, volume I. John Wiley, New York, 1957.

[12] B. Ghosh and S. Muthukrishnan. Dynamic load balancing in parallel and distributed networks by random matchings (extended abstract). In L. Snyder and C. E. Leiserson, editors, *Proc. of the 6th annual ACM Symposium on Parallel algorithms and architectures*, pages 27–29, New York, NY, 1994. ACM Press.

[13] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics*. Addison-Wesley, 2nd edition, 1993.

[14] G. Horton. A multi-level diffusion method for dynamic load balancing. *Parallel Computing*, 19(2):209–218, 1993.

[15] S. H. Hosseini, B. Litow, M. Malkawi, J. McPherson, and K. Vairavan. Analysis of a graph coloring based distributed load balancing algorithm. *Journal of Parallel and Distributed Computing*, 10(2):160–166, Oct 1990.

[16] R. Jeurissen and W. Layton. Load balancing by graph coloring, an algorithm. *Comput. Math. Appl.*, 27(3):27–32, 1994.

[17] V. Kumar, A. Y. Grama, and N. R. Vempaty. Scalable load balancing techniques for parallel computers. *Journal of Parallel and Distributed Computing*, 22(1):60–79, July 1994.

[18] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays–Trees–Hypercubes*. Morgan Kaufman Publishers, San Mateo, CA, 1992.

[19] F. C. H. Lin and R. M. Keller. The gradient model load balancing method. *IEEE Transanctions on Software Engineering*, 13(1):32–38, January 1987.

[20] J. W. Lindeberg. Eine neue Herleitung des Exponentialgesetzes in der Wahrscheinlichkeitsrechnung. *Mathematische Zeitschrift*, 15:211–225, 1922.

[21] B. Litow, S. H. Hosseini, K. Vairavan, and G. S. Wolffe. Performance characteristics of a load balancing algorithm. *Journal of Parallel and Distributed Computing*, 31:159–165, 1995.

[22] R. Lüling and B. Monien. Load balancing for distributed branch and bound algorithms. In *Proc. of the 6th. International Parallel Processing Symposium (IPPS '92)*, pages 543–549, 1992.

[23] R. Lüling and B. Monien. A dynamic distributed load balancing algorithm with provable good performance. In *Proc. of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 164–173, 1993.

[24] R. Lüling, B. Monien, and F. Ramme. A study of dynamic load balancing algorithms. In *Proceedings of the 3rd IEEE SPDP*, pages 686–689, 1991.

[25] J. G. Malone. Automated mesh decomposition and concurrent finite element analysis for hypercube multiprocessor computers. *Computer Methods in Applied Mechanics and Engineering*, 70(1):27–58, September 1988.

[26] L. M. Ni, C.-W. Xu, and T. B. Gendreau. A distributed drafting algorithm for load balancing. *IEEE Transactions on Software Engineering*, SE-11(10):1153–1161, October 1985.

[27] V. N. Rao and V. Kumar. Parallel depth first search. I: Implementation. *International Journal of Parallel Programming*, 16(6):479–499, 1987.

[28] V. N. Rao and V. Kumar. Parallel depth first search. II: Analysis. *International Journal of Parallel Programming*, 16(6):501–519, 1987.

[29] E. Speckenmeyer. Is average superlinear speedup possible? In *Proc. Computer Science Logic 1988 (CSL '88)*, pages 301–312. Springer Verlag (LNCS 385), 1989.

[30] E. Speckenmeyer, M. Böhm, and T. Seifert. Workload balancing on trees, grids, hypercubes and cliques. Technical report, Heinrich-Heine-Universität Düsseldorf, 1994.

[31] J. A. Stankovic and I. S. Sidhu. An adaptive bidding algorithm for processors, clusters and distributed groups. In *Proc. of the 4th International Conference on Distributed Computing Systems*, pages 49–59, 1984.

[32] S. Tschöke. A portable parallel branch-and-bound library (PPBB-lib). Technical Report 9, PC^2, Universität-GH Paderborn, June 1994.

[33] R. v. Hanxleden and R. L. Scott. Load balancing on message passing architectures. *Journal of Parallel and Distributed Computing*, 13(3):312–324, November 1991.

[34] O. Vornberger. Load balancing in a network of transputers. *Distributed Algorithms*, pages 116–126, 1987.

[35] R. D. Williams. Performance of dynamic load balancing algorithms for unstructured mesh calculations. *Concurrency*, 3(5):457–481, October 1991.

[36] C. Z. Xu and F. C. M. Lau. Analysis of the generalized dimension exchange method for dynamic load balancing. *Journal of Parallel and Distributed Computing*, 16(4):385–393, 1992.

[37] C.-Z. Xu and F. C. M. Lau. Optimal parameters for load balancing using the diffusion method in k-ary n-cube network. *Information Processing Letters*, 47(4):181–187, 1993.