Solving the Traveling Salesman Problem with a Parallel Branch-and-Bound Algorithm on a 1024 Processor Network

S. Tschöke, M. Räcke, R. Lüling, B. Monien

Department of Mathematics and Computer Science
University of Paderborn
Germany
e-mail : sts@uni-paderborn.de

Abstract

This paper is the first to present a parallelization of a higly efficient best-first branch-and-bound algorithm to solve large symmetric traveling saleman problems on a massively parallel computer containing 1024 processors. The underlying sequential branch & bound algorithm is based on 1-tree relaxation introduced by Held and Karp (Lagrangean approach) and improved by Volgenant and Jonker.

The parallelization of the branch & bound algorithm is fully distributed. Every processor performs the same sequential algorithm but on a different part of the solution tree. To distribute subproblems among the processors we use a new direct-neighbor dynamic load-balancing strategy. The general principle can be applied to all other branch-and-bound algorithms leading to an "automatic" parallelization.

At present we can efficiently solve traveling salesman problems up to a size of 318 cities on networks of up to 1024 transputers. On hard problems we achieve an almost linear speedup.

Keywords: Combinatorial Optimization; Distributed Algorithms; Dynamic Load-Balancing; Parallel Branch-and-Bound; Symmetric Traveling Salesman Problem (STSP)

1 Introduction

The efficient solution of large combinatorial optimization problems is highly important for many applications in the field of science and engineering. One example for these problems is the placement and routing problem in the design of VLSI systems. Using todays technology and algorithmic methods, it is not possible to give optimal solutions for these problems. Therefore heuristic solution are computed using local search methods like simulated annealing [11] or genetic algorithms [21].

Many other problems from the areas of operations research and artificial intelligence can also be defined as combinatorial optimization problems. Among these is the traveling salesman problem. To solve such problems, an integer solution vector has to be found that respects some finite set of constraints and minimizes/maximizes a given function.

The symmetric traveling salesman problem has been intensively studied in the past. Optimal solutions are computed by methods like branch & bound and branch & cut. In general branch & cut [4] lead to more efficient algorithms than branch & bound but these methods are normaly tailored to a specific problem. Any efficient parallelization would also be specific to the traveling salesman problem, since the sequential cut algorithm based on polyhedral theory has to be parallelized to achieve high efficiency.

Our aim is to use efficient methods for the traveling salesman problem but to provide a parallelization that is also usable for other problems. Therefore we choose the traveling salesman problem as a benchmark.

The name branch & bound describes a large class of search techniques. Among these are depth-first branch & bound and best-first branch & bound. Depth-first branch & bound performs a depth first search through the solution space and cuts off a branch whenever its bound is worse than the best solution found up to that time. Best-first branch & bound always branches a subproblem with minimal bound. This technique leads to very good computational performance, but it has to pay for this with large storage requirements. We will therefore use a best-first algorithm for the solution of the traveling salesman problem.

We use the method of 1-tree relaxation introduced by Held and Karp [6, 7] which was refined by several authors. These refinements were compared by Balas and Toth in [14]. It was found that the improvements of Volgenant and Jonker [28, 29, 30] lead in general to the best results. Therefore we use ideas of their branch and bound algorithms.

This algorithm was shown to be very efficient. In fact, the parallilization of a trivial branch & bound procedure is very simple. Achieving saturation of larger networks becomes increasingly complex when highly efficient algorithms are used to perform the branch & bound operations. This is because the resulting search tree becomes smaller and it is therefore much harder to keep all processors busy during runtime.

The implementation of branch & bound on a shared memory system is straight forward, since each processor has direct access to the global heap. The simulation of global memory on a distributed memory system for the solution of branch & bound problems was described in [17]. This technique is only useful in special cases or for small numbers of processors and leads in general to communication bottlenecks. Only if the computation time for one branching step is much larger than the communication time to transfer one load unit, bottlenecks can be avoided.

The distributed implementation of branch & bound was studied by several authors. Some of this work can be found in [1, 8, 10, 12, 17, 18, 19, 24, 27, 31, 32]. Distributed implementations of branch & bound solving the traveling salesman problem can be found in [24, 5]. Most of these work is done on smaller networks using branch and bound algorithms which are not as efficient as the ones described in [14]. Therefore only relatively small instances of the traveling salesman problem could be solved using these algorithms .

Our parallelization of this algorithm is fully distributed. The communication between processors is done solely by message passing on a fixed interconnection network. There is no global or shared memory between processors. Every processor has its own local heap and performs a sequential branch & bound algorithms on this heap. If a processor computes a new solution of the problem, it is broadcasted to all processors in the network.

Since subproblems are generated and consumed dynamically, it is necessary to use a distributed dynamic load balancing algorithm to distribute the workload equally through the processor network. We use a technique that is based on a workload balancing between neighbored processors which leads to a global balancing in the whole network.

We achieved the following results:

- Using our load balancing algorithm we have shown that it is possible to saturate networks of up to 1024 processors if the workload is large enough.
- Symmetric traveling salesman problems of up to 318 cities can be solved using branch & bound approaches in resonable time
- The used principle is general applicable, which means that every sequential branch & bound algorithm can "automatically" be transformed into a parallel algorithm.

The paper is organized as follows: A description of our branch and bound procedures solving the traveling salesman problem is given in section 2. In section 3 we discuss some general questions concerning distributed load balancing and present our parallelization of branch & bound using distributed heap management and dynamic load balancing. Section 4 gives the experimental results on different transputer networks. Some conclusions final the paper.

2 The sequential Branch & Bound Algorithm

Given a complete weighted undirected graph $G = (\{1, \ldots, n\}, E)$ and a costmatrix C on G. A tour is a circle in G which visits each vertex exactly once. The symmetric traveling salesman problem (STSP) is the problem to find a tour of minimal length. In this section we shortly describe the best-first branch & bound procedures we used to solve the STSP.

The TSP is a well studied problem of combinatorial optimization, where many good lower and upper bounds are known. A good sequential best-first branch & bound algorithm consists of four different parts. A method to compute lower bound (relaxation), a branching strategy, good heuristic solutions for bounding and methods to reduce the solution space. So it has to spend much effort on the computation of a single subproblem in the branch-and-bound procedure, but therefore the search tree can be kept as small as possible. Trying to parallelize one might think that it may be better to use less efficient but faster algorithms for the single task (subproblem) and taking into the bargain a larger search tree. Because if you have more tasks is it easier to saturate and balance large networks of processors. But all our experiments show, that large TSP instances can only be solved with small search trees and good lower bounds. So less tasks means that a efficient load balancing is very important to guarantee the best-first approach.

2.1 Computation of Lower Bounds

As a strong lower bound we use the 1-tree relaxation. First let us define a 1-tree of the graph G as a spanning tree of the subgraph $G^{'} = (\{2, \ldots, n\}, E^{'})$ together with two edges incident to vertex one. Hence a 1-tree of minimal length is a minimal spanning tree of the subgraph $G^{'}$ together with the two shortest edges incident to vertex one. The observation that every 1-tree contains exactly one circle and that therefore a 1-tree whose verticies all have degree 2 is a tour, lead Held & Karp to the following formulation of the STSP as a 0-1 integer program:

$$P : min \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} c_{i,j}\, x_{i,j}$$

$$\begin{array}{lrcl} s.t & A_1\, x & = & 2 \\ & A_2\, x & \leq & b_2 \end{array}$$

$$\text{x integer}$$

where the (0,1) vector x indicates the selected edges. The restrictions $A_2\, x \leq b_2$ force x to be a 1-tree, and the restrictions $A_1\, x = 2$ force all verticies to have degree two. By examining this formulation we see that the degree constraints are those which make the linear program hard to solve. Therefore we relax this program by eliminating the constraints $A_1\, x = 2$ and punishing violations of these constraints by adding the term $\pi^T\, (A_1\, x - 2)$ to the cost function. The components of π are called the *Lagrangean multipliers*, the relaxed program P' is called the *Lagrangean Problem*:

$$\begin{array}{lll} & P'_0 : min & C^T x + \pi^T (A_1\, x - 2) \\ & \quad s.t. & A_2\, x \leq b_2; \quad x\ integer \\ \Leftrightarrow & P'_1 : min & \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (c_{i,j} + \pi_i + \pi_j)\, x_{i,j} \quad - 2 \sum_{i=1}^{n} \pi_i \\ & \quad s.t. & A_2\, x \leq b_2; \quad x\ integer \\ \Leftrightarrow & P'_2 : min & C^T x + \pi^T (d_x - 2) \\ & \quad s.t. & A_2\, x \leq b_2; \quad x\ integer \end{array}$$

where d_x is a vector containing the degree of each vertex.

Let π be constant, than formulation P'_1 provides an easy way to compute a solution of P'. We only have to compute a minimal 1-tree according to a changed cost matrix.

Let x be a feasible solution for P, then x is also feasible for P' and furthermore for every vector π the inequality $C'(x) \leq C(x)$ holds. Hence every solution for P' gives a lower bound for the length of the optimal tour. The quality of this lower bound depends highly on the choice of the Lagrangean multipliers π. Let us write this as $L(\pi)$. To compute a high lower bound we employ the subgradient method to find a vector π maximizing $L(\pi)$.

As described in [15], a subgradient at point π_0 is defined to be a vector t such that for all $\pi \in R$: $L(\pi) \leq L(\pi_0) + (\pi - \pi_0)\, t$. The significance of a subgradient t is that the half-space $\{\pi | (\pi - \pi_0)^T\, t \geq 0\}$ contains all solutions whose cost values are larger than the cost at π_0. Thus, any subgradient points to a direction of ascent of $L(\pi)$ at π_0. The idea of the subgradient method is to choose a subgradient at the current solution π_0, and to walk along it a sufficiently small step such that $L(\pi)$ still increases. Let π be constant and x be a vector that achieves the minimum in P', then formulation P_2' provides the easily computable subgradient $(d_x - 2)$, i.e. walking into a direction where vertices whith one edge or with more than 2 edges are punished. The size δ of the step taken in the direction of this subgradient is choosen heuristically, and is not constant during the computation of a lower bound.

So the computation of a lower bound means computing a series of minimal spanning trees with Prime's and Kruskal's algorithm (which of them performs better and is choosen depends on the number of availiable edges). The quality of the lower bound LB of a subproblem SP is determined by two heuristically choosen parameters (subgradient step size $\delta(i)$ and the maximal number of 1-tree computed during the subgradient loop $max.tree$). We have seen that we have to make longer subgradient iterations in the beginning of the branch & bound especially while computing the initial lower bound. We choose $max.tree$ in $O(n)$ (5n) to get a good initial lower bound and in general $(n/5)$. In the parallel case we have to pay for a good initial lower bound with idle times in the begining. But in our experiments we see that this strategy was always superior to a worse initial lower bound and less idle time.

```
SP = node of the search tree (subproblem)
initialize $\pi_0$
$i$       := 0
$w_{max}$ := 0
repeat
    $T$ := minimum 1-tree $T$ of $SP$ according to the costs $(c_{i,j} + \pi_i + \pi_j)$
    $w_\pi$ := cost of $T$
    if ( $w_\pi > w_{max}$)
        $w_{max} := w_\pi$
        $\pi_{max} := \pi$
        $T_{max} := T$
    $\pi_{i+1} := \pi_i + \delta(i)\,(d_x - 2)$
    $i := i + 1$
until ($i > max.tree$)
$LB(SP) := w_{max}$     ;lower bound of the subproblem
```

2.2 Branching Strategy

Typically, a branching-strategy successively partitions the current set of feasible solutions into subsets (in our case into disjoint subsets) and calculates a lower bound on the cost of solutions in each subset. The branching hopefully improves the lower bounds for each of the newly created subproblems, but an increase cannot be guaranteed.

In case of the traveling salesman problem partioning is usually accomplished by requiring and forbidding certain edges, so that each subset is characterized by a set R of required edges and a set F of forbidden edges. A derived subproblem $SP_{R,F}$ represents the set of all 1-trees $T_{R,F}$ which includes the edges of R and excludes the edges of F. Then the weight $w_{T_{R,F}}(\pi)$ of the minimal 1-tree of $T_{R,F}$ is a lower bound $LB(SP_{R,F})$on the cost of any tour for the derived subproblem $SP_{R,F}$. The lower bound function as described before is executed on every node of the search tree.

In case of the symmetric TSP two simple rules are used to extend these sets of required and forbidden edges.

(a) if two edges incident to a node are required, all other edges incident to this node can be forbidden.

(b) if there are no more than two not forbidden edges incident to a node both edges are required.

In their branching scheme, Volgenant & Jonker [28] select a node n of the current best 1-tree with degree greater than two. Because of rule (a) there are always at least two non-required edges in this 1-tree incident to node n, say $e1$ and $e2$. Now branching partitions a subproblem $SP_{R,F}$ into disjoint subproblems SP_1, SP_2 and SP_3 with:

$$\begin{aligned} SP_1 &= SP_1(R \cup \{e1, e2\}, F) \\ SP_2 &= SP_2(R \cup \{e1\}, F \cup \{e2\}) \\ SP_3 &= SP_3(R, F \cup \{e1\}) \end{aligned}$$

If node n is already incident to a required edge we know subset SP_1 to be infeasible. This leads to a branching into two subsets. Hence, the degree of the searchtree varies between degree three and two. The closer the current lower bound is to the optimal solution, the more edges are required and the more nodes of the search tree have degree two. To select the node n and the edges $e1$ and $e2$ we use a heuristic based on ideas of Volgenant & Jonkers [28].

1. Choose node n to be on the subtour of the current best 1-tree (a 1-tree always consists of a cycle and some branches).
2. If possible take n to be incident to a required edge in order to prefer branching with degree two.
3. If there is still a choice of n take the node with the smallest number of feasible edges incident to it.

The idea of the heuristic is to break up the 1-tree circle and to keep the search tree small. We tried several other strategies with higher braching degree. But they were all not very efficient.

2.3 Computation of Upper Bounds

Good upper bounds are necessary to reduce memory consumption, especially when using a best-first branch & bound strategy on a network of distributed processors. In our case a subproblem, that is sent through a network to a certain processor, keeps the whole information (the sets R and F and the Langragian multipliers π) to determine a lower bound on the subproblem and to branch. So we do not have to route through the network to collect data.

We use a simple but effective well-known heuristic to compute upper bounds suggested by Volgenant & Jonker in their algorithm. It is based on a heuristic combining a minimum 1-tree and perfect matching developed by Christofides [2]. It can be used on every 1-tree availiable during the solution procedure and needs less than 1% of the computional time. A feasible tour is directly generated out of the current best 1-tree.

We confine its use to 1-trees consisting of a cycle with branches that have only nodes with degree not greater than two. In this Cristofides' perfect matching can be heuristically solved very simple:

For each branch — with one node with degree 1, say $n1$, and one node on the cycle, say $n3$ — there are two nodes, say $n2$ and $n4$, on the cycle connected to $n3$. Now we choose the cheapest way out of two to construct a part of a tour, namely the sequence
$\cdots - n2 - n1 - \cdots - n3 - n4 - \cdots$ or $\cdots - n2 - n3 - \cdots - n1 - n4 - \cdots$.
This means for the current length l of the heuristic solution while contructing a tour:
$l := l + min\{c_{n1,n2} - c_{n2,n3}, c_{n1,n4} - c_{n4,n3}\}$. On this contructive solution we run the 2-opt improvement heurstic. These are not the best known heuristics, but for instances up to 300 cities we got sufficient upper bounds.

2.4 Reduction of the Solution Space

The traveling salesman problem is a large 0-1 integer program. It is a goal of every algorithm solving 0-1 integer programs to reduce the solution space if possible, i.e. to eliminate variables that can be shown to be zero or one in every optimal solution.

For the TSP this means finding superfluous edges that cannot be part of an optimal tour and finding indispensable edges that must be part of every optimal tour. Obviously indispensable edges may be put into the set R of required edges and the superfluous edges into the set F of forbidden edges. We implemented two methods to increase the number of forbidden and required edges suggested by Volgenant and Jonker [29, 30]:

The first method is the edge exchange in 1-trees [29]. The idea is that if the weight of the minimal 1-tree, that results from forcing a certain edge into the 1-tree, exceeds the current upper bound, this edges will be superfluous and if the weight of the minimal 1-tree, that results from excluding a certain edge, exceeds the current upper bound, this edges is indispensible. Hence we need good upper bounds, because the better the availiable upper bound is the more edges will be identified to be required or to be forbidden. The other method is the identification of nonoptimal edges [30]. The fact that every optimal tour has to be 2-optimal is exploited to determine edges that cannot be contained in any optimal tour.

Reduction of the solution space is very important. Even with large processor networks we were not able to solve instances over 150 cities without reduction methods.

3 Parallel Algorithm

In this section we give a description of our load balancing strategy. On distributed systems a general and problem-independent method to parallelize best-first branch-and-bound methods is search tree decomposition. The heap with the subproblems (nodes of the search tree) has to be distributed. A load balancing algorithm is performed on each processor in parallel to the sequential branch-and-bound algorithm. The load-balancer knows the local load situation and needs information from other processors in order to decide where to send the tasks.

3.1 Objectives of dynamic load balancing

In general we can distinguish static and dynamic load balancing. In the case of static load balancing, which is more often referred as the mapping problem, the task is to map a static process graph to a fixed interconnection topology to minimize dilation, processor load differences and edge congestion. For an overview see [20]. If the load situation of a distributed system changes dynamically by generation and consumption of load units in an unpredicatable way, it is necessary to use a dynamic load balancing strategy which reacts on these changes of the load. The decision of load remapping can either be centralized or distributed. Since centralized strategies are only useful for small distributed systems we suggest a new distributed dynamic load balancing strategy.

A good load balancing algorithm has to achieve the following objectives:

- *minimize search overhead.* A sequential best-first algorithm explores in each step the subproblem with the minimal lower bound (first heap element), i.e. in a distributed system with n processors a load balancing method would be optimal if at every time every processors has one of the n subproblems with the best lower bound. Therefore distributed minimum computation or a centralized heap would be necessary, which is in general too inefficient. So a distributed algorithm will probably produce search overhead, i. e. the solution tree which is computed by the parallel algorithm is larger than that of the sequential algorithm, using the same branch-and-bound algorithm.
- *minimize idle times*, i.e. no processor should run out out work.
- *minimize communication.* Every load balancing strategy has to spend time on distributing workload and information through the network, which is not needed in the sequential case.

These goals are partly contrary. Reducing search overhead implies increase of communication or you will get idle times. Reducing communication implies a more unbalanced network with search overhead and idle times. A distributed dynamic load-balancing algorithm has to adopt a middle course to get the best speed-up.

In a parallel branch-and-bound algorithm *load* of a processor can be defined as the subproblems generated during the branching and then put on the heap. To evaluate the load in a load balancing method we have to define a weight function w on the local heap elements of a processor. Let $p_i \in \{p_1, \ldots, p_n\}$ a processor, UB the bound of the current best solution (upper bound), $LB(sp)$ a lower bound on a subproblem sp and $H_{p_i} = \{sp_1, \ldots, sp_k\}$ the heap elements (subproblems) of processor p_i, with $\forall\ sp \in H_{p_i} : LB(sp) < UB$). For the TSP we define the following weight functions $w : \{p_1, \ldots, p_n\} \longrightarrow N$:

$$w_{LB}(p_i) = min\,\{LB(sp)\,|\,sp \in H_{p_i}\} \qquad (1)$$

$$w_{\#}(p_i) = |H_{p_i}| \qquad (2)$$

Our aim is to optimize the load balancing according to both objectives (quality w_{LB} and quantity $w_{\#}$ of the load). We choose w_{LB} (first heap element) as a weight function for the quality of load. This is because in an efficient TSP algorithm where the breadth of search tree is relatively small, you have to keep all minimal lower bounds of the local heaps on an equal level to provide best-first branch-and-bound and to avoid as much search overhead as possible.

When trying to keep the lower bound on an equal level, the number of load units (heap elements) on each processor is probably not very well balanced. This implies on the one hand idle times during the computation when new solutions (upper bounds) are found and large parts of the heaps are bounded. On the other hand the memory consumption of the processor is not equal. So some processors can run out of memory while others have enough free memory. Therefore we need as a second weight function the number of heap elements $w_{\#}$. It would be also possible to define one weight function for both objectives, see [18] for other weight functions, but we suggest two heap-weight function because we perform a different kind of balancing on each of the weight functions.

Now our load balancing strategy manages to globally keep all processors on a nearly equal level according to the defined heap weights, although only the heapweights of neighbored processors are kept equal. For this purpose, each processor knows the heapweights w_{LB} and $w_{\#}$ of his neighbors in the network. In case of our transputer network every processor has four neighbors. A processor sends some subproblems to a neighbor if the quality or the quantity of its heapelements is a certain amount higher than on its neighbors. If the local heap situation changes the neighbors will be informed. With this strategy we reach maximal processor saturation and minimal search overhead, while communication overhead and idles times are very low.

3.2 Load Balancing Algorithm

Dynamic distributed load balancing algorithms can be characterized by the knowledge used to decide when to distribute load units (decision base) and by the space in which a processor migrates such load (migration space). We can distinguish between the global and local decision base. In the first case a processor needs knowledge about the global system or nearly every processor in the network to decide about a load migration. If a local decision base is used, a migration decision is purely based on the local situation and that of the neighboring processors in the network. In the same way we distinguish between a global and local migration space [19].

Our algorithm uses a local decision and migration space. Local migration space seems us to be the most natural way of load balancing in large networks and dynamic load situations. This is based on our results which we gained by a comparison of different known and some new load balancing strategies, described in [19, 18]. From these experiments it seems to be better to decide only about the direction a packet of workload should take in the network, instead of the exact destination processor if the local load situation is likely to change very fast.

The basic principle of this algorithm is to balance the workload of a processor in a way that the heap weights w_{LB} and $w_{\#}$ of a processor p_i and its neighbors $\{p_1, \ldots, p_k\}$ are on a nearly equal level. Since the load situation varies dynamically the algorithm tries to achieve a situation in which the maximum weight difference $\max_{j \in \{1,\ldots,k\}} \mid w(p_i) - w(p_j) \mid$ is less than a fixed Δ. Since we balance according to the quality (minimal lower bound) and the quantity of load on each processor we try to reach a situation where all processors work on subproblems with the same lower bound and all processors have an equal memory consumption and an equal number of heap elements. A processor tries to achieve a balance with its neighbors which are four in our case. Because each processor belongs to overlapping balanced islands, this strategy leads to a balance throughout the whole network.

The following parameters determine our load balancing process:

- Δ_{LB} : a process p_i initiates a load balancing activity or participates on a load balancing activity of its neighbor p_j, only if their weights w_{LB} (minimal lower bound on heap elememts) differ by more than Δ_{LB} cost units.

- $\Delta_{\#}$: a process p_i initiates a load balancing activity or participates on a load balancing activity of its neighbor p_j, only if their weights $w_{\#}$ (number of the heap elememts) differ by more than $\Delta_{\#}$ elements.

- $info.delay$: a neighbor is informed about the new local weight situation ($w_{LB}(p_i)$, $w_{\#}(p_i)$), only if the last information was done more than $info.delay$ time units ago.

- $work_{LB}.delay$, $work_{\#}.delay$: a workload unit (subproblem) is send to a neighbor, only if the last work was send to this neighbor more than $work.delay$ time units ago.

- $send_{LB}.rate$, $send_{\#}.rate$: the number of subproblems send to a neighbor is $send.rate$ percentage of the load difference, i.e. 50% means that the load between the two interacting processor is expected to be equal after sending.

In case of the TSP we choose $\Delta_{LB} = 1$ to avoid as much search overhead as possible and to keep the network very well balanced over the whole time of computation. We choose $\Delta_{LB} = 3$ (= max. degree of the search tree generated by the branching strategy) to achieve a very well balanced memory comsumption in the whole network. This gives us the possibility to solve larger instances. to keep the network very well balanced over the whole time of computation. The *delay* times of roughly the average computation time of a single subproblem (computing lower bounds) perform best. They are very importaant to avoid trashing effects of sending load back and forth all the time. A $send_{\#}.rate$ of 65% of subproblems according to quantity of heap does best, i.e. sending more than half of the difference lead to a globally better balanced network. The $send_{LB}.rate$ was kept constant between 1 and 3, i.e. at least 1 and at most 3 subproblems were distributed in one balancing operation.

On a single processor the load-balancing process is running in parallel to the branch-and-bound process. To describe the algorithm in detail we have to define four message items. If a local branch-and-bound process is out of work it sends an (IDLE)-message to the local balancer. The message (SOLUTION) broadcasts upper bounds. (INFO) contains the heapweights w_{LB} and $w_{\#}$ and (WORK) a subproblem. Initially the branch-and-bound process of one processor p_0 computes the initial subproblem and sends it to the local balance process. Then the branch-and-bound processes of all processors send an (IDLE) message to their local balance process. The load-balancers are *always* waiting for messages from their branch-and-bound process or their neighboring processors. As in the sequential case the branch-and-bound process is performing the branch-and-bound loop, i.e. if not terminated it waits for a subproblem, does the braching step, computes lower bounds and sends derived subproblems which are not bounded back to the load balancer which does the heap management. After that it sends again an (IDLE)-message. Additionally to the load-balancing a distributed termination detection is done.

```
PROC Load_balancing_process ()

  initialization()
  while not terminated do
    on receipt of a message from

      1: on receipt of (IDLE) from b&b-process
           b&b.process.idle := TRUE
           if not empty_heap(H) then
             send (WORK, get_first_subproblem(H)) to b&b-process
             b&b.process.idle := FALSE
             inform_neighbors()

      2: on receipt of (SOLUTION, S) from b&b-process
           if Cost(S) < UB then
             UB := Cost(S)
             send (SOLUTION, S) to all neighbors

      3: on receipt of (SOLUTION, S) from neighbor $p_j$
           if Cost(S) < UB then
             UB := Cost(S)
             send (SOLUTION, S) to all neighbors except $p_j$

      4: on receipt of (WORK, subproblem) from neighbor $p_j$ or b&b-process
           if b&b.process.idle then
             send (WORK, subproblem) to b&b-process
             b&b.process.idle := FALSE
           else
             insert_into_local_heap (H, subproblem)
             $\pi$ := new random permutation
             foreach neighbor $j$ of $p_i$ do
                 $\text{balance}_{LB}(\pi(j))$
             $\pi$ := new random permutation
             foreach neighbor $j$ of $p_i$ do
                 $\text{balance}_{\#}(\pi(j))$
             inform_neighbors()

      5: on receipt of (INFO, $w_{LB}$, $w_{\#}$) from neighbor $p_j$
           $weight_{LB}[p_j] := w_{LB}$
           $weight_{\#}[p_j] := w_{\#}$
           $\text{balance}_{LB}(p_j)$
           $\text{balance}_{\#}(p_j)$
           inform_neighbors()

      6: termination detection
END
```

The load-balance process has to handle five different message events. (1) If the branch-and-bound process sends an (IDLE)-message, it tries to reply the subproblem with the locally minimal lower bound (first element), if availiable. If the local heap changes, the neighbors are informed. Rules (2) and (3) provide a easy way of broadcasting upper bounds.

On receipt of a subproblem (4) the load-balancer replies the subproblem to the branch-and-bound process, if this is still idle. If not then the subproblem is inserted to the local heap and balancing with the neighbors is started according to the defined parameters in a random order. First balancing according to the lower bound is done. If the delay time is over and a processor has a better minimal lower bound than one of its neighbor, it sends the second heap element (second lowest lower bound on the processor) to its neighbor. We found out that is is very important to keep the best subproblem for the own branch-and-bound process. This avoids the case that a branch-and-bound process gets idle and wants a new subproblem but the best one is just on the way between the processors in some buffers. This strategy reduces the search overhead.

After balancing according to the lower bound, the number of heap elements is probably unbalanced. To balance the heap size without disturbing the just balanced lower bound, now the element are taken from the end of the heap array. These subproblems have worse lower bounds, i.e. not only the number of elements are balanced but also the subproblems of worse quality which leads to relatively equal heap after a new solution is broadcasted and lot of worse subproblem are deleted. After the load balancing operation all neighbors are informed of changes in the heap weights.

On receipt of heapweights w_{LB} and $w_{\#}$ (5) from a neighbor the local weight information of this neighbor is updated. Than the balancing procedures are performed with this neighbor. At the end all neighbors are informed of changes in the heap weights. For details of the functions *balance*, $inform\_neighbors$ and *initialization* see the appendix.

This strategy is completely problem independent and can applied to other branch-and-bound algorithms as well.

4 Results

There are two goals we want to reach with parallelization. We want to gain a good *speedup* that means to be nearly n times faster with n processors than with one processor, and we want to solve larger problem instances.

For speedup measurements it is very important how much time the algorithm spends on the computation of each node in the solution tree. The algorithm starts with one initial subproblem (the root of the solution tree) and hence in the first phase of the program run there are less subproblems than processors. Let us call the time from programstart to the moment when all processors are working for the first time the start-up time. The effect which the start-up time has on the speedup, depends on the relation between the total runtime and the start-up time. For this reason only a modest speedup can be gained for a problem instance which needs only a small number of subproblems to be computed.

There are possibilities to avoid the idle times during the start-up phase, or to use them differently. One way to use this idle times is to compute upper bounds. This is significant, because a good upper bound is not only important for bounding, if the upper bound is used during the computation of lower bounds (as it is the case for the algorithm implemented by us) it has a strong positive influence on the computation of the lower bounds and hence on the number of evaluated subproblems. This way of using the idle times can be used in general. One way to avoid this idle times is to cluster the processors and parallelize the computation of a single subproblem. A disadvantage of this method is that it cannot be applied to all branch & bound algorithms.

Problem	(4)	(8)	(16)	(32)	(64)	(128)	(256)	(512)	(1024)
pr76		8,00	16,17	32,72	64,08	126,20	234,35	428,76	773,16
gr120	3,32	7,03	11,40	18,03	19,61	21,50	19,61		
pr124	3,95	7,67	15,43	22,96	27,99	28,81	24,67		
bier127	3,83	7,01	9,81	11,00	12,11	11,16	11,61		
pr136				32,00	56,37	103,97	169,93		
u159	4,32	3,69	11,17	11,78	12,96	13,31	13,22		
rat195		8,00	15,49	28,47	60,53	89,37	138,27		
lin318		8,00	15,34	28,54	29,59	29,52	29,54		

Table 1: Speedup of TSPLIb instances on De Bruijn Networks

Problem	(4)	(8)	(16)	(32)	(64)	(128)	(256)	(512)	(1024)
avg70	3,81	7,49	13,41	20,55	22,63	23,14	22,98		
best70	3,89	7,81	14,90	26,81	42,77	56,83	57,35		
avg100	3,77	7,59	14,84	26,37	33,53	36,95	38,66		
best100	3,94	8,02	16,78	32,34	60,38	117,18	203,20		
avg120	3,92	7,90	15,67	31,09	58,63	101,38	154,35		
best120	3,96	7,94	15,80	32,15	63,37	116,79	207,29		
avg150	3,84	7,73	15,82	30,97	58,27	106,48	170,91	290,92	484,75
best150	4,12	7,88	15,91	31,98	64,45	131,61	237,46	422,61	760,12
avg250			16,00	28,11	53,55	102,15	176,83	306,17	521,54
best250			16,00	30,74	61,03	119,38	219,65	398,34	651,62

Table 2: Speedup of random-euclidean TSPs on De Bruijn Networks

Let us now watch our computational results. A library of symmetric traveling salesman problem instances known from the literature was provided to us by G. Reinelt [25]. In addition to these instances we generated a set of random euclidean problems. We solved random euclidean problems up to a size of 250 cities. The problem *lin318* was the largest problem from the library, which was solved. Table 1 and 2 show the speedup results on De Bruijn (diameter log n) networks. One must take into account that traveling salesman problem instances although they have the same number of cities can differ very much in the effort which has to be spend on solving them. Instances with less then 4000 subproblems cannot saturate a network of 1024 processors. All these instances can only gain good speed-ups up to 64 or 128 processors. The computational times of hard problem as the pr86 or random 250 cities problems on the largest network are between 200 and 800 seconds with at most 26000 subproblems.

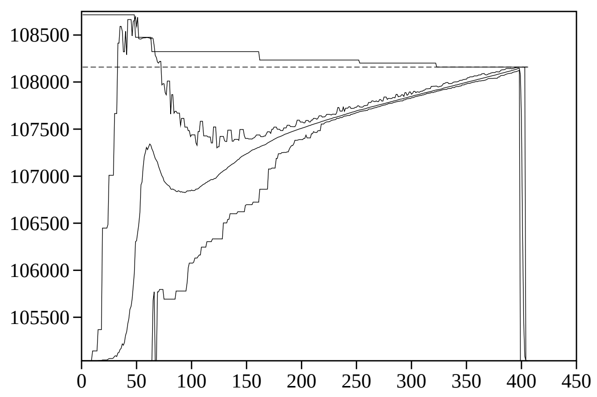


Fig. 1a

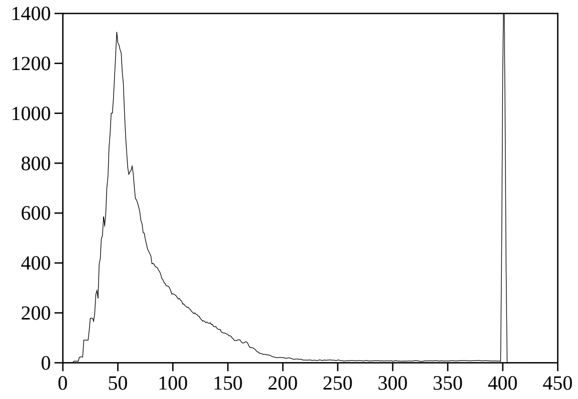


Fig. 1b

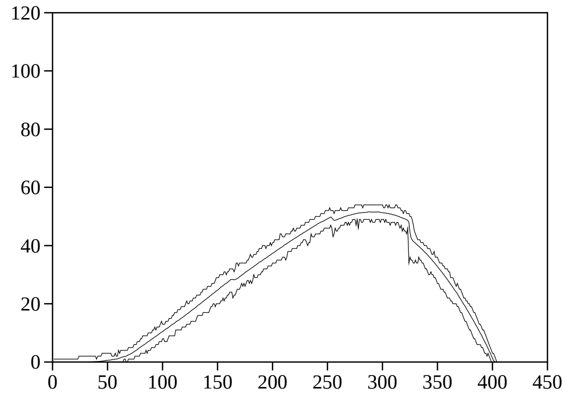


Fig. 1c

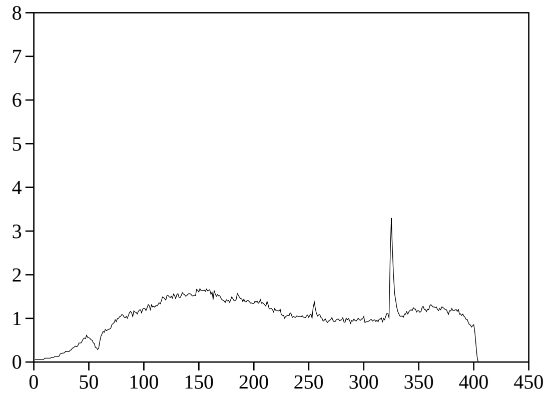


Fig. 1d

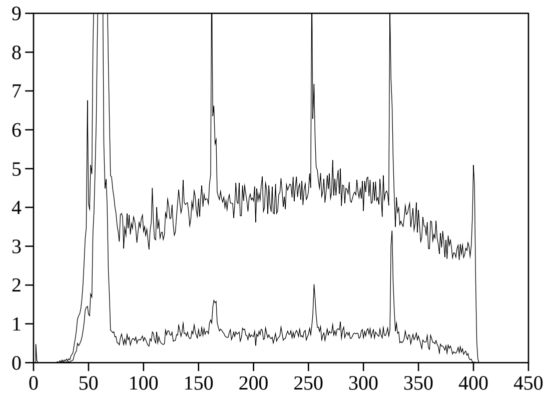


Fig. 1e

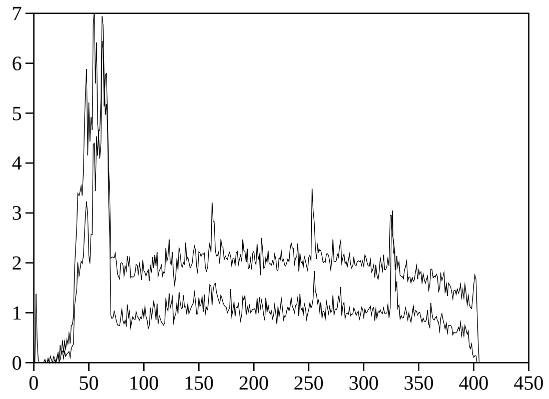


Fig. 1f

The following figures 1 - 3 show the solving of the pr76 instance of the TSP-Library on 256 processors
(a) shows the lower bound on the processors (minimum, average and maximum) additionaly the upper bound and the optimal solution 108159 (dashed).

(b) standard deviation of the lower bound of (a)

(c) shows the number of subproblems during computation (minimum, average and maximum).

(d) standard deviation of (c).

(e) number of send messages (INFO) above (WORK)

(f) standard deviation of (e)

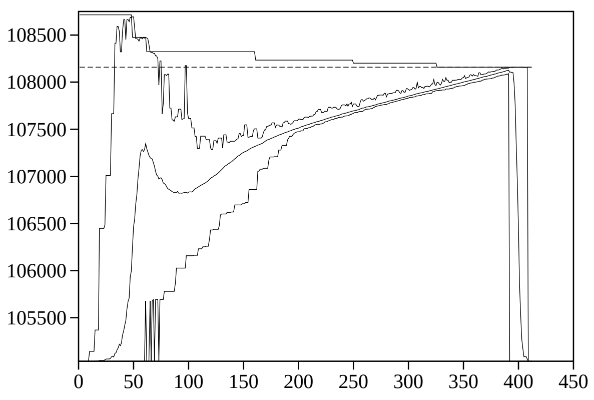


Fig. 2a

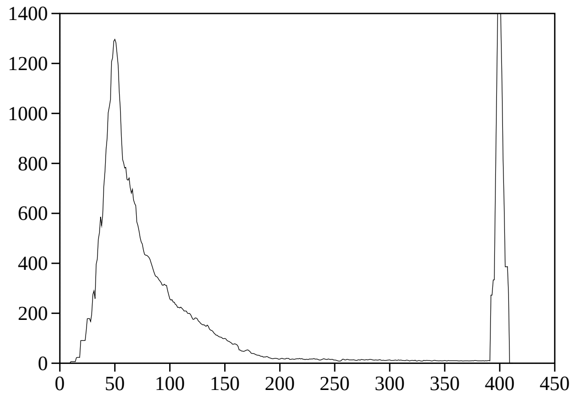


Fig. 2b

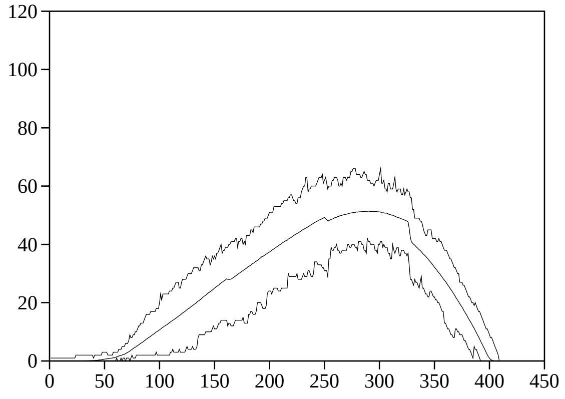


Fig. 2c

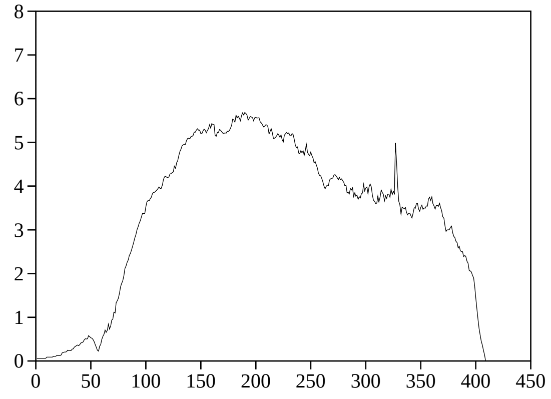


Fig. 2d

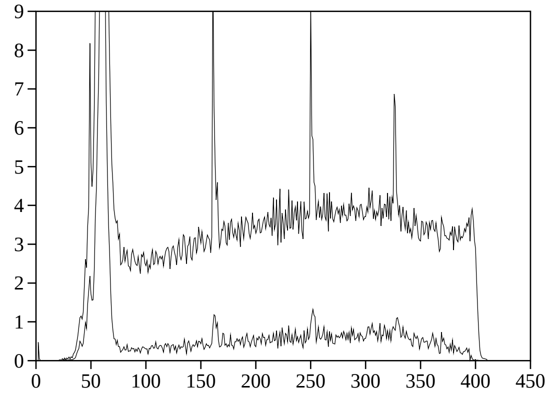


Fig. 2e

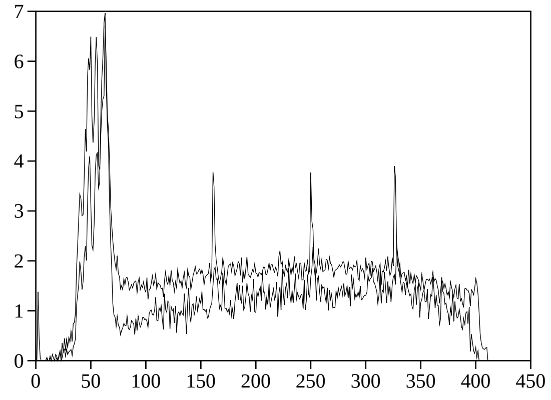


Fig. 2f

Figure 1 shows our good load-balancing strategy with short idle times and little search overhead. Figure 2 shows more idle times. Here the second weight function $w_{\#}$ is turned off. If we turn off weight function w_{LB} instead of $w_{\#}$ the computation is completely inefficient, because now the load-balancer does not guaratee a best-first branch-and-bound strategy anymore. Appendix B shows a 3-dimensional view of the performance of load-balancing alogorithm.

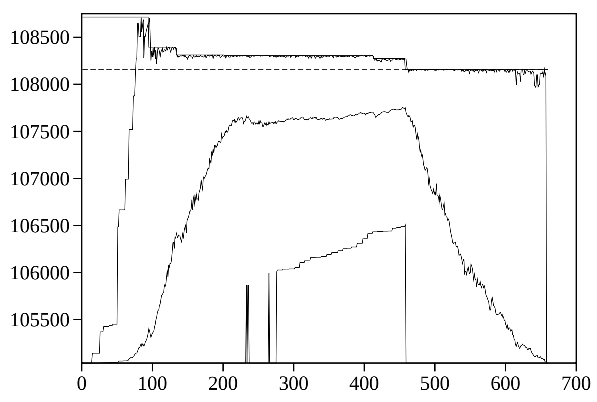


Fig. 3a

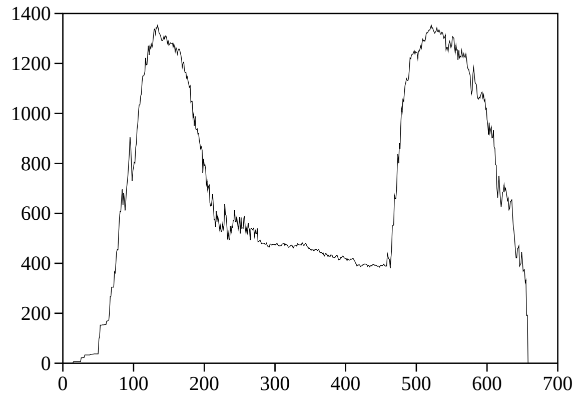


Fig. 3b

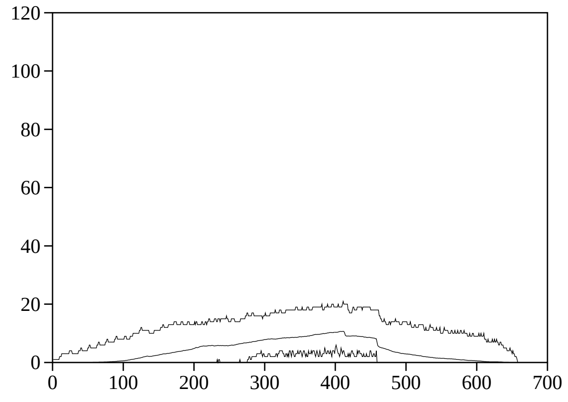


Fig. 3c

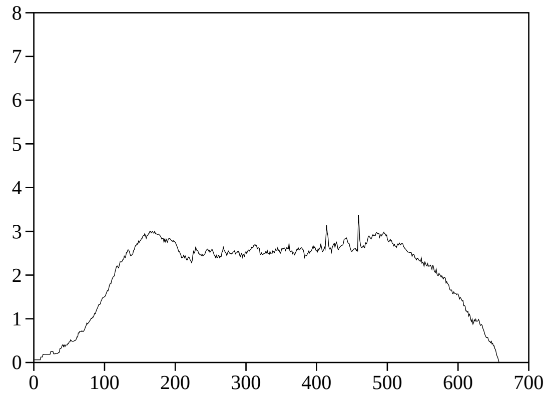


Fig. 3d

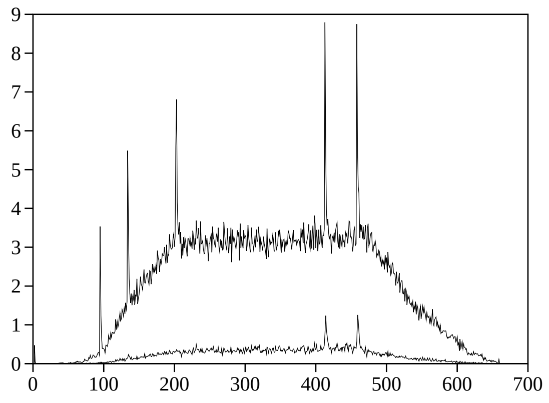


Fig. 3e

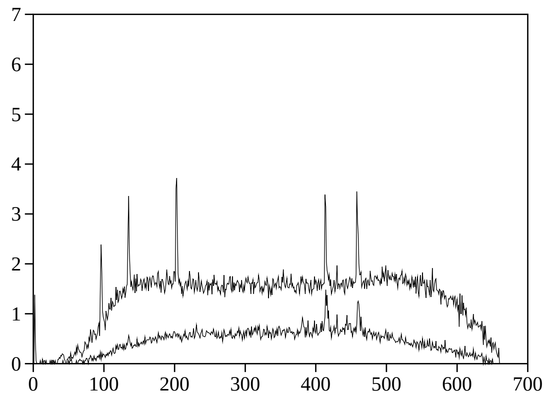


Fig. 3f

5 Conclusions

In this paper we presented a highly efficient distributed algorithm, solving the symmetric traveling salesman problem. We were able to solve instances of up to 318 cities in reasonable time. In general we could achieve a nearly linear speedup for networks of up to 1024 processors, if the computation load was high enough. If the workload is too small to saturate the whole network, which is often the case at the beginning of the computation we propose to compute upper bounds by those processors who are idle in these phases. These upper bounds lead to a significant reduction of the search space for our branch and bound procedures, since they enlarge the sets of forbidden and required edges. This technique can also be applied to other problems.

The load balancing strategy was found to be very efficient for the parallelization of best-first branch & bound algorithms, especially when using networks with short diameter as it is the case for the De Bruijn and Torus network. The used weight function of our load balancing strategy lead to equal workload distribution and minimal search overhead.

Since the computation time for one branching step will increase using more sophisticated branch and bound algorithms, the speedup will also increase if the number of computed subproblems is constant. This implies that larger problems can be solved with increased speedup.

Therefore we can conclude that the load balancing question for this problem has been solved although the networks were very large.

Appendix A: Procedures

```
PROC initialization()
    initialize_local_heap($H_{p_i}$)
    $terminated$          :=  $FALSE$
    $b\&b.process.idle$   :=  $FALSE$
    $UB$                  :=  Cost(heuristic solution)
    $weight_{LB}.old$     :=  0
    foreach neighbor $j$ of $p_i$ do
        $weight_{LB}[j]$   :=  $weight_{\#}[j] := 0$
        $last.info[j]$     :=  $last.work_{LB}[j] := last.work_{\#}[j] := 0$

    if $processor.ID = 0$ then
        wait for (WORK, $initial.subproblem$)
        insert_into_local_heap($H$, $initial.subproblem$)
        $weight_{LB}.old$  :=  $w_{LB}(p_i)$   ;(= $LB(initial.subproblem)$)
END

PROC balance$_{LB}(j)$
  if ($weight_{LB}[j] - w_{LB}(p_i) > \Delta_{LB}$)  then
      if ($timer - last.work_{LB}[j] > work_{LB}.delay$)  then
          $n := f(weight_{LB}, send_{LB}.rate)$
          repeat $n$-times
              send (WORK, get_second_subproblem ($H$))  to  neighbor $p_j$
          $last.work_{LB}[j] := timer$
END

PROC balance$_{\#}(j)$
  if ($w_{\#}(p_i) - weight_{\#}[j] > \Delta_{\#}$)  then
      if ($timer - last.work_{\#}[j] > work_{\#}.delay$)  then
          $n := (w_{\#}(p_i) - weight_{\#}[j]) * send_{\#}.rate$
          repeat $n$-times
              send (WORK, get_last_subproblem ($H$))  to  neighbor $p_j$
          $last.work_{\#}[j] := timer$
END
```

```
PROC inform_neighbors()
    foreach  neighbor j of $p_i$  do
        if  $(timer - last.info[j] > info.delay)$  or  $(w_{\#}(p_i) = 0)$  then
            send (INFO, $w_{LB}(p_i)$, $w_{\#}(p_i)$) to neighbor $p_j$
            $last.info[j] := timer$
    $weight_{LB}.old := w_{LB}(p_i)$
END
```

References

[1] E. Altmann, T. A. Marsland, T. Breitkreutz, *Accounting for Parallel Tree Search Overheads*, Proceedings of the International Conference on Parallel Processing 1988, pp. 198-201

[2] N. Christofides, *Worst-case Analysis of a new heuristic for the for the traveling salesman problem*, Carnigie-Mellon University, Pittburgh (1976)

[3] R. Feldmann, B. Monien, P. Mysliwietz, O. Vornberger, *Distributed Game Tree Search*, in: Kanal, Gopalakrishnan, Kumar, Parallel Algorithms for Machine Intelligence and Pattern Recognition, North Holland/ Elsevier Publ. Co.

[4] M. Grötschel, M. W. Padberg, *Polyhedral theory and Polyhedral computations*, in : E. L. Lawler et. al. : The Traveling Salesman Problem, John Wiley & Sons, 1985, pp. 251-360

[5] B. Gendron, T. G. Crainic, *Parallel Branch and Bound on Quadputers*, Proc. of 11th European Congress on Operational Research, Aachen 1991, pp. 51-52

[6] M. Held and R.M. Karp, *The traveling salesman problem and minimum spanning trees*, Operations Research 18 (1970) 1138-1162

[7] M. Held and R.M. Karp, *The traveling salesman problem and minimum spanning trees: part II*, Mathematical Programming 1 (1971) 6-25

[8] V. K. Janakiram, D. P. Agrawal, R. Mehrotra, *A randomized Parallel Branch and Bound Algorithm*, Proceedings of the International Conference on Parallel Processing 1988, 69-75

[9] R. M. Karp, Y. Zhang, *A randomized Parallel Branch and Bound Procedure*, Proceedings of the ACM Symposium on Theory of Computing 1988, pp. 290 - 300

[10] G. A. P. Kindervater, J. K. Lenstra, *Parallel Computing in Combinatorial Optimization*, Annals of Operations Research, 14 1988, pp. 245-289

[11] S. Kirkpatrick, C. D. Gelat, M. P. Vechi, *Optimization by simultaed annealing*, Science, vol. 220, 1983, pp. 671-680

[12] B. Kröger, O. Vornberger, *A Parallel Branch and Bound approach for solving a two dimensional cutting stock problem*, Technical Report No. 25/1990, University of Osnabrück, Department of Mathematics and Computer Science

[13] E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, D.B. Shmoys, *The Traveling Salesman Problem, A Guided Tour of Combinatorial Optimization*, John Wiley & Sons 1985 361-402

[14] E. L. Lawler, D. E. Wood, *Branch and Bound Methods: A survey*, Operations Research 14, 1966, pp. 699-719

[15] Th. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout*, B.G.Teubner; John Wiley & Son, 1990, 182-191

[16] F. C. H. Lin, R. M. Keller, *The Gradient Model Load Balancing Method*, IEEE Transactions on Software Engineering, Vol. 13, No. 1 January 1987

[17] R. Lüling, B. Monien, *Two Strategies for Solving the Vertex Cover Problem on a Transputer Network*, 3 rd Int. Workshop on Distributed Algorithms, 1989 Lecture Notes in Computer Science 392, pp. 160-171

[18] R. Lüling, B. Monien, *Load Balancing for distributed Branch and Bound Algorithms* Proc. of 6th Int. Parallel Processing Symposium 1992, pp. 543-549

[19] R. Lüling, B. Monien, F. Ramme, *Load Balancing in Large Networks : A Comparative Study* to appear in: Proc. of 3rd IEEE Symposium on Parallel and Distributed Processing, Dallas, 1991

[20] B. Monien, H. Sudborough, *Embedding one Interconnection Network in Another*, Computing Suppl. 7 (1990), pp. 257-282

[21] H. Mühlenbein, M. Gorges-Schleuter, O. Krämer, *Evolution algorithms in combinatorial optimization*, Parallel Computing, vol 7, 1988, pp. 65-85

[22] L. M. Ni, C. W. Xu, T. B. Gendreau, *Drafting Algorithm - A Dynamic Process Migration Protocoll for Distributed Systems*, 5 th Int. Conf. on Distributed Computing Systems 1985, pp 539-546

[23] M. Padberg, G. Rinaldi, *A Branch and Cut Algorithm for the Resolution of Large Scale Symmetric Traveling salesman problems*,

[24] M. J. Quinn, *Analysis and Implementation of Branch and Bound Algorithms on a Hypercube Multicomputer*, IEEE Transactions on Computers, Vol. 39, No. 3, 1990

[25] G. Reinelt, *TSPLIB – A Traveling Salesman Problem Library*, ORSA Journal on Computing 3 (1991), pp. 376-384

[26] J. A. Stankovic, I. S. Sidhu, *An Adaptive Bidding Algorithm for Processes, Clusters and Distributed Groups*, 4 th Int. Conf. on Distributed Computing Systems 1984, pp 49-59

[27] J. M. Troya, M. Ortega, *A study of parallel branch and bound algorithms with best bound first search*, Parallel Computing 11 1989, pp. 121-126

[28] T. Volgenant, R. Jonker, *A branch and bound algorithm for the symmetric traveling salesman problem based on the 1-tree relaxation*, European J. Operational Res. 9 (1982) 83-89

[29] T. Volgenant, R. Jonker, *The symmetric traveling salesman problem and edge exchange in minimal 1-trees*, European J. Operational Res. 12 (1983) 394-403

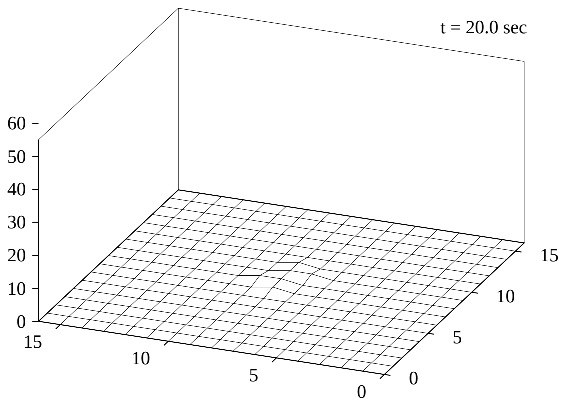
[30] T. Volgenant, R. Jonker, *Nonoptimal Edges for the Symmetric Traveling Salesman Problem*, Operations Research Vol. 32 No. 4 (1984) 65-74

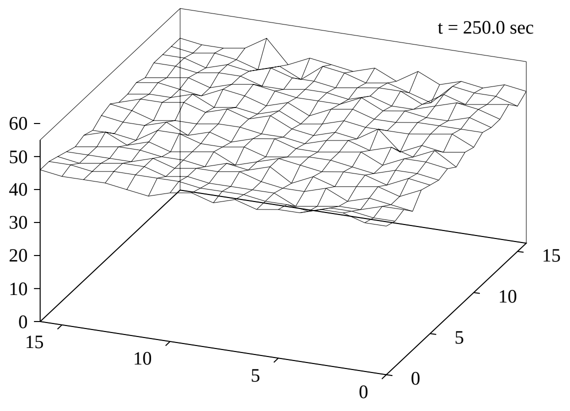
[31] O. Vornberger, *Implementing branch and bound in a ring of processors*, Proceedings of CONPAR 86, Lecture Notes of Computer Science 237, Springer Verlag, pp. 157-164, 1986

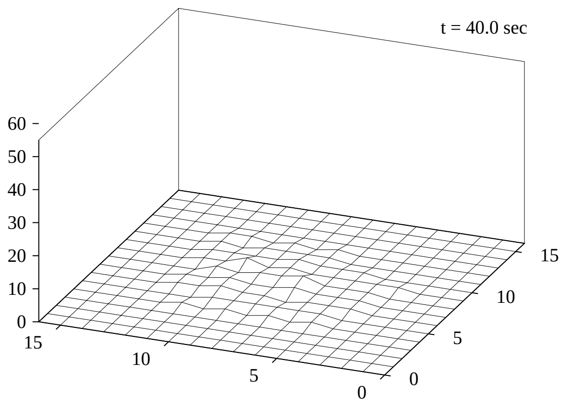
[32] O. Vornberger, *Load Balancing in a Network of Transputers*, Distributed Algorithms 1987, Lecture Notes of Computer Science 312, Springer Verlag, pp. 116 - 126

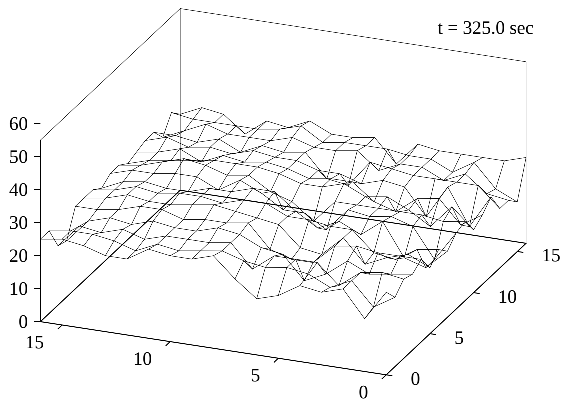
Appendix B:

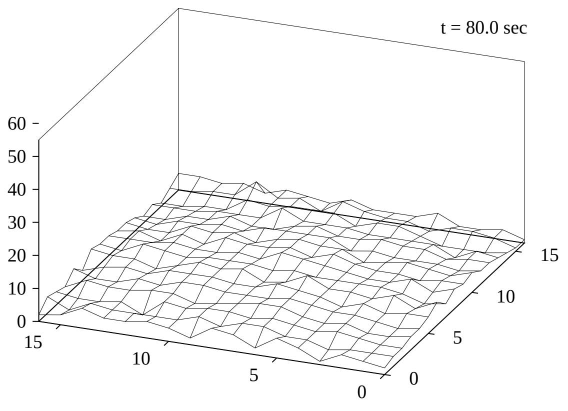
Loadbalancing of the number of subproblems during computation of the pr76 of 17x17 torus

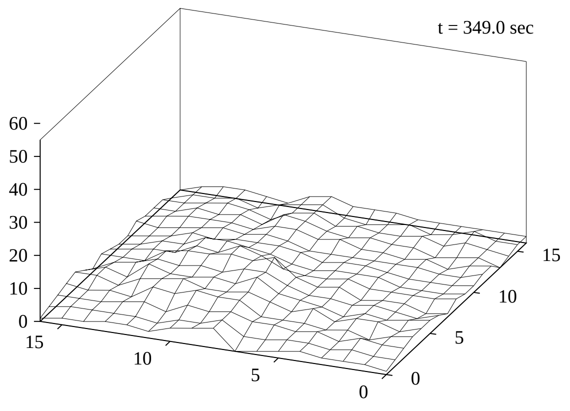

t = 20.0 sec

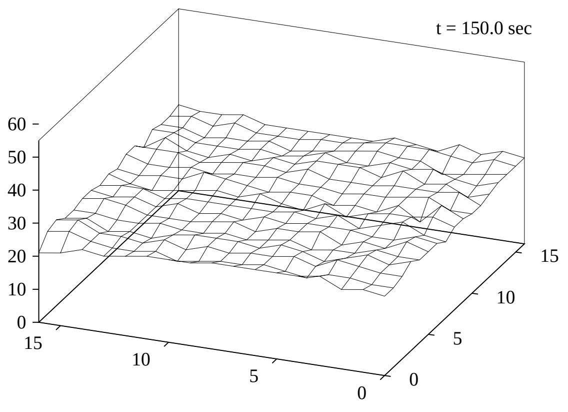

t = 250.0 sec

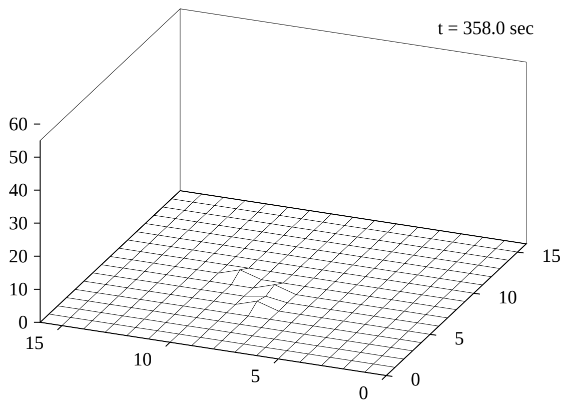

t = 40.0 sec


t = 325.0 sec


t = 80.0 sec


t = 349.0 sec


t = 150.0 sec


t = 358.0 sec