ANGEWANDTE MATHEMATIK UND INFORMATIK
UNIVERSITÄT ZU KÖLN

Report No. 95.191

Solving large-scale traveling salesman problems with parallel Branch-and-Cut

by

Michael Jünger

Peter Störmer

1995

ZENTRUM FÜR PARALLELES RECHNEN (ZPR)
Institut für Informatik
Universität zu Köln
Pohligstraße 1
D-50969 Köln

Addresses of the authors:

Michael Jünger
Institut für Informatik
Universität zu Köln
Pohligstraße 1
D-50969 Köln
Germany
E-mail: mjuenger@informatik.uni-koeln.de
Telephone: +49/221/470-5313

Peter Störmer
Institut für Informatik
Universität zu Köln
Pohligstraße 1
D-50969 Köln
Germany
E-mail: stoermer@informatik.uni-koeln.de
Telephone: +49/221/470-5315

Abstract

We introduce the implementation of a parallel Branch-and-Cut algorithm to solve large-scale traveling salesman problems.

Rather than using the well-known models of homogeneous distribution and simple Master/Slave communication, we present a more sophisticated distribution that takes the advantage of several independent features of a Branch-and-Cut code.

Computational results are reported for several instances of the TSPLIB.

1 Introduction

In the last years the Branch-and-Cut method has shown to be the state-of-the-art for several hard combinatorial optimization problems, such as the traveling salesman problem (TSP), the linear ordering problem and the max-cut problem. Although being able to solve large problem instances, enormous amounts of time have to be invested. Speeding up the algorithms and thereby being able to solve larger problem instances is therefore most desirable.

On the side of the hardware, in the last decade a totally new architecture has been made available: parallel computers constructed with microprocessors. Today, the fastest computers in the world use high-level parallelism. They make possible the solution of problems a sequential computer is not able to cope with. Parallel computing is therefore receiving a rapidly increasing amout of attention.

In this paper, we describe the combination of these two powerful methods: a parallel Branch-and-Cut algorithm, solving large-scale traveling salesman problems. Since the Branch-and-Cut method is a general method to solve large combinatorial optimization problems, we plan to build a general parallel Branch-and-Cut framework in the future.

The original sequential Branch-and-Cut TSP code was written by Michael Jünger, Gerhard Reinelt and Stefan Thienel. A complete description as well as computational results can be found in [JRT94]. In this paper only those features will be explained that are necessary to understand the parallel algorithm.

The organization of the present paper is as follows.

Section 2 shows the history of prior practical work in the field of parallel Branch-and-Bound and parallel Branch-and-Cut.

In section 3 we give a general account of our parallel implementation. Section 4 presents the fully parallelized Branch-and-Cut algorithm and all its features. The final section 5 shows computational results for an implementation on Thinking Machines' CM-5.

2 Survey of prior work

The literature of parallel Branch-and-Bound algorithms gives particular attention to the following three topics:

- speedup anomalies,
- tree search strategies,
- decentralized versus centralized control.

The *speedup* of a parallel algorithm is defined as

$$speedup_p = \frac{T_s}{T_p}$$

where T_s is the worst-case running time of the fastest known sequential algorithm for the given problem, and T_p is the worst-case running time of the parallel algorithm on p processors ([Akl89]). Clearly, the larger the speedup, the better the parallel algorithm.

An *absolute* superlinear speedup, i. e., a speedup achieved by a parallel algorithm that is greater than the number of processors used, is theoretically not possible, since otherwise the sequential algorithm with running time T_s is not really the fastest possible. However, since the assumption that a single processor can always emulate multiple processors without a loss of efficiency is questionable for real-world algorithms, we will talk of superlinear speedup *relative* to the serial algorithm at hand (see also [Qui94, BT98] for this discussion).

Very early on, it became evident that parallel versions of standard Branch-and-Bound techniques can have *speedup anomalies*, such as (relative) superlinear speedups (also called *acceleration anomalies*) and so-called *deceleration anomalies*.

Acceleration anomalies occur in Branch-and-Bound because of the order in which the search tree is evaluated. This order affects the amount of work to be done later on. The size of the search tree can vary as the number of processors increases. In [Eck93] this is called *search anomaly*.

The case when a given Branch-and-Bound algorithm takes more time when instantiated on $p_2 > p_1$ processors than on p_1 is called deceleration anomaly.

Many researchers discussed conditions under which both types of anomalies can or cannot occur. Theoretical treatises on speedup and performance for parallel Branch-and-Bound algorithms may be found in [QD86, LSp85, IYF79, LW84, LW86a, LW86b, LW90]. [KL86] give a tutorial introduction to the literature on parallel computers and algorithms that is relevant for combinatorial optimization.

The effects of parallel Branch-and-Bound algorithms that expand several nodes simultaneously were shown in simulations by [IFY79, IYF79, LW84, LW86a, LW86b, LW90, LSa84, Moh82] as well as in experiments by [WM84]. [LSa84] and [LW84, LW86a, LW90] studied the likelihood of branch-and-bound algorithms exhibiting deceleration and acceleration anomalies.

In Branch-and-Cut the philosophy is to do a much bigger effort of finding lower bounds than in Branch-and-Bound. Therefore, the computation of a single Branch-and-Cut node takes much more time than in Branch-and-Bound. Moreover, the number of nodes in the enumeration tree is far bigger in Branch-and-Bound. The described effects are thus expected to be less frequent for our parallel Branch-and-Cut algorithm. This is also shown in [Can88, CH90]. In their parallel Branch-and-Cut algorithm for general zero-one integer programming problems they observed deceleration anomalies only for problems which could be solved very quickly.

Another important research topic is the implementation of tree search strategies. Much of the early research concentrated on depth-first and breadth-first search because of the memory limitations of available computers. [PNR88] claim that using depth-first search results in finding better solutions earlier and reducing the Branch-and-Bound tree significantly. [IFY79, IYF79, EH80, FM85] showed that with the depth-first approach and an appropriately chosen number of processors linear and superlinear speedup can be obtained. [WM84] made similar experiments with best-first approaches. [GK92] provide an overview of parallel algorithms with different search strategies for solving discrete optimization problems.

A third prominent topic in the literature is that of central versus decentralized control of the search process. One possibility is to keep the "pool" of active Branch-and-Bound nodes in shared memory, using locking mechanisms to prevent multiple processors from making inconsistent changes to the pool. Experiments and simulations considering this model may be found in [IFY79, PM92, PNR88, Rou87]. On systems with distributed memory an essentially equivalent scheme is to distinguish one particular processor as a "Master" responsible for assigning subproblems to "Slave" or "Worker" processors (see e. g. [dRT88]). [Eck93] and [KG92] say that this scheme suffers from a lack of theoretical "scalability". However, we use this Master-Slave model and show good results for it. This, again, has the reason in the difference between Branch-and-Bound and Branch-and-Cut: In Branch-and-Cut, introducing a Master does not lead to a bottleneck, since the computation time for one Branch-and-Cut node on the Slaves is far larger than in Branch-and-Bound.

The other possibility for algorithms on computer systems without shared memory is a "homogeneous" distribution of processors. This means that all processors do essentially the same, and the pool of active Branch-and-Bound nodes has to be divided and distributed over the processors. Proposals for such approaches date back to [EH80]. [KZ88] have proposed and analyzed a very simple load distribution scheme (see also [MS90] and [Ran90]), which is compared to a centralized scheme in [Eck93]: each newly generated subproblem is sent to the pool of a randomly chosen processor. On some multiprocessor systems this may be too costly in terms of communication. This is especially true of older systems. Thus, works such as [FM85, NK87] propose methods in which work is transferred between pairs of processors only when one processor completely runs out of work. Other authors have suggested schemes in which a work balancing is achieved by a periodic exchange of information about the work pools ([AM88, KRN88, LM89, LM92, LMRT92, NK87, Vor86, Vor87]).

[AM88, Eck93, MRRT91, RRM93] include comparisons of central and distributed control in various contexts.

3 Structure of the algorithm

Our parallel Branch-and-Cut implementation has the structure shown in figure 1.

Each of the small boxes in the figure can be viewed as a sequential program with the possibility of sending and receiving data to and from other programs. The lines leading from one box to another show that there may be communication between these programs.

The different programs have the following task: There is a Master having the overall view of the advance of the solution, several Node Solvers that are responsible for the computation of the nodes of the Branch-and-Cut tree, an Auxiliary Problem Solver (APS) that solves subproblems which are not really necessary for the correctness of the solution but speed up the algorithm, and a Pool that stores globally valid constraints.

Both the APS and the Pool consist of a Master and a quantity of Slaves. From "outside", only the Master can be accessed. It receives messages and thereupon decides which Slave has to get which job.

The parallel TSP code was designed following four major goals:

(1) As many independent parts of the original sequential algorithm should be executed independently, as possible.

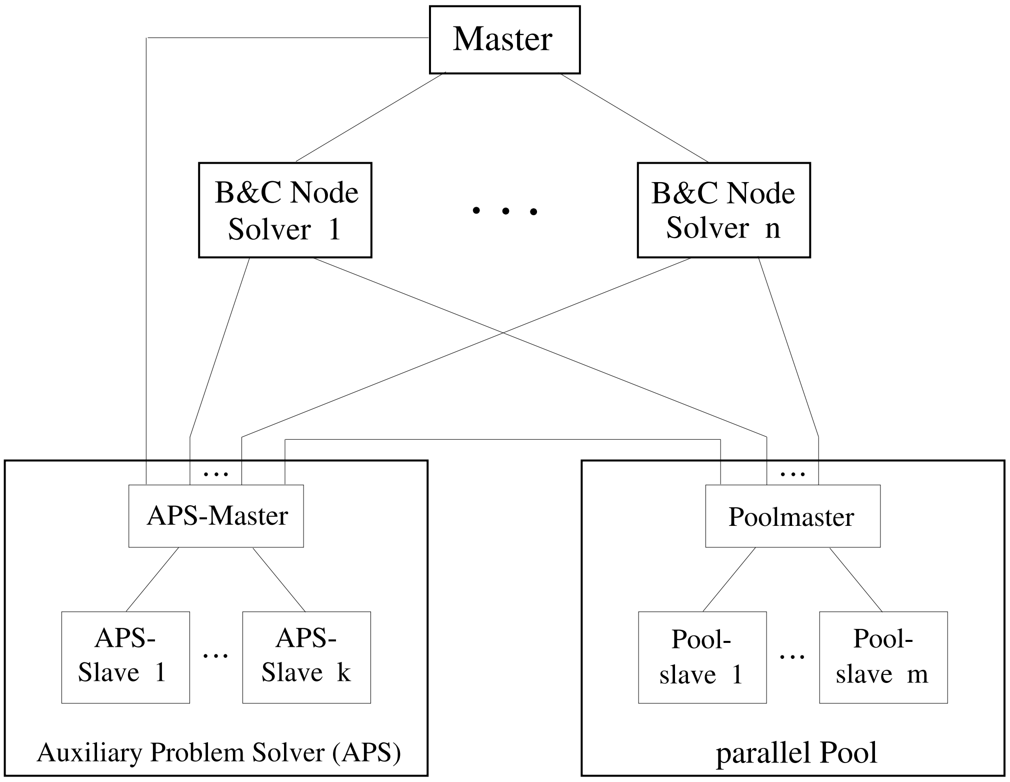

Figure 1: Structure of the parallel Branch-and-Cut code

(2) The program should be flexible with respect to reusability for a general Branch-and-Cut code in the future as well as the problems we wanted to solve and the number of processors available.

(3) Algorithms finding upper bounds (i. e. tours) and new violated inequalities, which could not be employed in the sequential code because of their long running time, should run in parallel to the rest of the program.

(4) The communication between processors should be minimized.

To achieve a reusability of the code ((2) of the above goals), we use the programming language C++. This allows us to create classes which are valid also for a more general Branch-and-Cut code than the specific TSP solver. This includes classes for the message passing, the Pool and many other parts of the program. Classes for message passing are useful when the program shall be ported to a different parallel computing environment, e. g., a workstation cluster or a different parallel computer. Then mainly the message passing class has to be reimplemented, while the other parts of the program can be preserved.

In a general Branch-and-Cut code the Pool has to store several different types of constraints. Therefore it is necessary to have a general pool class from which specific pools can be derived. The functionality of the pool will always be the same (functions to put constraints into and get constraints out of the Pool, pool separation etc.), only the way the stored constraints look like is different.

To be able to specify a desired configuration of the available processors we use a *Config file*. In this file all information about the processors, their tasks and the according parameters can be defined. This allows us to arrange the processors in a way suitable for the given problem. If e. g. the pool separation is needed very rarely, the number of Poolslaves can be set rather small. Then perhaps more processors are needed as Branch-and-Cut Node Solvers. It is also possible to start two processors with the same task but different parameters.

Goal (3) shows a very powerful feature of parallel programming: We are in the position to speed up a sequential code by using a set of processors not only for the original task, but also for new tasks, whose solution was impossible before.

The minimization of communication is certainly another important aim, although it lies in the nature of our algorithm to have a lot of communication e. g. to update the bounds, the Pool and the new solutions. However, especially for the parallel Pool we found some possibilities to reduce the communication in order not to get a bottleneck when many Node Solvers are working.

All parts of the model will be explained in detail in the following chapter.

4 The algorithm in detail

4.1 Node Solvers

The Branch-and-Cut Node Solvers are the processors responsible for the evaluation of the Branch-and-Cut nodes. They receive the data for a new node from the Master and start the evaluation. The main routine consists of a separation phase where new constraints are separated and added to the LP, an LP solving phase where the augmented LP is solved and a communication phase where new updating information is received from the Master. These phases are explained in the following subsections. There are several possibilities when the main routine is left. Then the Node Solver sends an according message to the Master as well as necessary information and waits for information of the next Branch-and-Cut node to compute. Table 1 summarizes the possible cases for the main routine to be left.

Table 1: Events that lead to an end of a Branch-and-Cut node computation

event	**message**	**corresp. information**
the guarantee is reached	`GUAR_REACHED`	—
no more constraints found	`BRANCH`	- LP value - statuses of the constraints - constraint ID of each constraint - statuses of the variables - values of the variables - sparse graph - set variables
a contradiction occurred	`CONTRADICTION`	- LP value - statuses of the variables - set variables
`llb` $\geq$ `gub`	`FATHOM`	- LP value - statuses of the variables - set variables
LP is infeasible		
frac. solution is feasible		

We will now describe the different phases of the Node Solver program.

4.1.1 Starting a new Branch-and-Cut node

Before starting the main loop, information about the current Branch-and-Cut node has to be received. This information contains the following:

- the current global upper bound,
- data about the parent of the Branch-and-Cut node, i. e., the number of equalities, the constraint IDs of all active constraints in the last LP solved (constraint IDs are unique numbers corresponding to slots in the parallel Pool, s. section 4.3), the number of variables and the statuses of the constraints and of the variables,
- data about the current LP (statuses of the variables and the constraints)
- the sparse graph,
- other information, e. g., the ID of the Branch-and-Cut node, the level in the Branch-and-Cut tree and a flag whether the node is the root of the remaining tree.

Having received this information, the Node Solver is able to regenerate a first LP by receiving the constraints that were active in the last LP of the parent from the parallel Pool. Then the main loop is entered.

There are some functions which need the resources of the parallel Pool very intensively. These are mainly the regeneration of the LP at the beginning of a Branch-and-Cut node computation or when new variables have been added, and the pricing. In these cases, those constraints in the Pool that are active in the current LP are used to restore the needed information. In a straightforward implementation, all these constraints can be received from the parallel Pool again and again as soon as they are needed. But this would encumber the Pool with too many requests. Therefore, in order to avoid bottlenecks at the Poolmaster, the active constraints are stored not only in the parallel Pool, but also in a local pool of a Node Solver. This pool is used as follows. At the beginning of a Branch-and-Cut node computation the Node Solver loads all constraints that were active on the parent node into the local pool. It may even be the case, that in this local pool there exist already some of these constraints because they were used by the Node Solver in the computation of other Branch-and-Cut nodes as well. Then loading this constraint can be omitted. Having finished this initial loading, the first LP can be regenerated without a further access to the parallel Pool. In the separation phase, all separated constraints are not only sent to the parallel Pool, but also stored in the local pool. Also constraints that come from the Pool during a pool separation are stored here. Thereby, the pricing and the regeneration just act on locally held constraints. That leads to less load on the parallel Pool and a smaller overall running time.

4.1.2 Main loop of a Node Solver

As LP solver we use CPLEX by R. E. Bixby.

Solving the LP yields a fractional solution that can be exploited in several ways. First, it is used for the separation of constraints. Second, it can help find better feasible solutions, since it contains a number of variables equal to 1 and a certain number of variables whose values are close to 1. The exploitation of fractional solutions in this way is done by the APS. Therefore, every fractional solution that does not contain subtours is sent to the APS-Master as soon as it is known from the LP solving phase.

It may happen that the fractional solution is also feasible. In this case the feasible solution is sent also the APS, because the APS administers all feasible solutions.

Then the communication phase starts. Here, the Node Solver sends a request message to the Master, and the Master replies by sending the following information:

- The lower bound of the root of the remaining Branch-and-Cut tree.
- The current global upper bound. This is very important, since new upper bounds may show that this Branch-and-Cut node can be fathomed.
- A list of new edges. If the APS found a new tour, the edges of this tour are added to the sparse graph on the Master. The Master then sends these new edges also to the Node Solvers in the communication phase in order to update their sparse graphs. By this means we augment the set of active variables without pricing.
- A flag whether the Branch-and-Cut node n that the Node Solver is currently working on has become the root of the remaining Branch-and-Cut tree. This may happen if other Node Solvers fathomed all other children of the parent node of n.

In the separation phase new constraints are generated that are violated in the current fractional solution. We use a hierarchical separation in three stages. The second and the third stage are only executed if the respective previous stages did not generate new inequalities. In the first stage we call the separation routines for subtour elimination and simple combs. In the second stage we try to identify two-matching constraints. Each constraint found is sent to the parallel Pool. The Pool sends back a unique number (the so-called *constraint ID*) that can be used for further access to the constraint in the Pool. After the second stage it has to be decided which constraints shall be added to the current constraint matrix of the LP. Since every constraint has a constraint ID and the Pool stores a constraint only once (s. section 4.3), the constraint ID can be used to determine whether a new constraint does already exist in the current LP. Of course, only those constraints are added to the LP that do not exist yet. The third separation stage is the pool separation. To start its execution a small `POOLSEP` message is sent to the Poolmaster together with the current fractional solution. The Poolmaster then sends back all separated constraints. The pool separation checks for violated comb or clique tree inequalities that are not already in the LP. We could do pool separation also with subtour elimination constraints and two-matching constraints before calling the respective separation routines, but usually direct separation is more efficient for these types of inequalities.

In the current implementation we use an exact subtour elimination constraint separator ([CP80]), a heuristic for the separation of two-matching constraints of [PRi90], which is based on the exact separation algorithm of [PRa82] and heuristics for separating comb constraints and clique tree constraints (based on ideas of [GH91]). The latter are separated by the APS-Slaves (s. section 4.4).

There is an important difference with respect to the process of finding violated inequalities between a program with a single Node Solver as in the sequential TSP code and a program with a set of Node Solvers as described here. Certainly, the computation of the Branch-and-Cut tree is reasonably faster. But this is not the only innovation. More important is, that the constraints which are stored in the parallel Pool by one Node Solver may be violated in the solution of another Node Solver. Therefore, the computation of the Branch-and-Cut nodes in parallel brings up a new quality of problem solving: The Node Solvers “help” each other find new violated inequalities, rather than one Node Solver profiting of its former computations, as in the sequential case. [Can88] and [CH90] specify this as the main reason for super-linear speedup.

4.2 Master

The Master is the processor having a "global" view of the solution progress. The Node Solvers get their jobs from the Master and send back the results to it. The Master is responsible for the distribution of the Branch-and-Cut nodes to the Node Solvers. Moreover, it stops the whole Branch-and-Cut algorithm as soon as either the required guarantee is reached, or no more Branch-and-Cut nodes have to be processed.
In detail, the Master has to store the following information:

- The global upper and the global lower bound.

  The upper bounds are obtained from the Auxiliary Problem Solver.

  The local lower bounds are received continously from the Node Solvers. This may lead to an increase of the global lower bound and therefore stop the whole algorithm, if the guarantee requirement is reached.

- A list of all Branch-and-Cut nodes. This is the Branch-and-Cut tree known so far. It includes the set variables and their related statuses (`settolowerbound`, `settoupperbound`) for each node, the local lower bound of the corresponding subproblem, the constraint IDs of all constraints active in the last LP solved, and the basis of the last processed LP (i. e., the statuses of the variables and the constraints) in order to avoid phase 1 of the simplex algorithm.

- Information about each Node Solver. This means the state in which the Node Solver is (busy, idle) and a pointer to the current Branch-and-Cut node it is working on.

- The "global" sparse graph and reserve graph. These graphs are needed to start a new Branch-and-Cut node. They are initialized at the beginning of the algorithm. When a Node Solver stops the computation of a Branch-and-Cut node (since one of the events shown in table 1 occurred), it sends back it's current sparse graph. The Master's sparse graph is then augmented by all new edges.

A sketch of the algorithm running on the Master is shown on page 13. The main routine waits for incoming messages and reacts according to them. The message type coming most frequently is the so-called *update request* of the Node Solvers. Here, the Node Solver wants to know whether there is some important information available. The Master then sends back the data described in section 4.1.2 (the communication phase). To this end an edge list for each Node Solver is maintained. This list stores all edges that may be new to the corresponding Node Solver. It is initially empty and augmented by new edges of the feasible solutions coming from the APS and by new edges in the sparse graphs coming from the Node Solvers when branching is necessary. As soon as a Node Solver is requesting for update, its edge list is sent to it and emptied afterwards. This way, the "global" sparse graph is available for every Node Solver. This is done, because variables which became active in a Branch-and-Cut node have a good probability to be in the optimum tour and should be in the LP as soon as possible.

Other functions the Master has to execute are:

- `initialize_algorithm`
  This includes: reading the problem data, initializing global variables, constructing a first tour, constructing the sparse and the reserve graph, sending initializing messages to the Node Solvers, the APS and the parallel Pool.

- `start_node_solver`
  When a Node Solver is started to process a Branch-and-Cut node, the necessary data has to be sent (explained in section 4.1.1).

- `determine_global_lower_bound`
  Whenever a new local lower bound was received from a Node Solver, or a Branch-and-Cut node could be marked inactive (because of fathoming, branching or contradictions), the new global lower bound is determined, which is the minimum of the lower bounds of all active Branch-and-Cut nodes. If the global lower bound has changed, the guarantee requirement might be satisfied, and the algorithm can stop.

- `select_new_node`
  A Branch-and-Cut node is selected from the set of active Branch-and-Cut nodes. If the list of nodes is empty and no Node Solver is busy, the optimality of the best known tour can be concluded. Otherwise the selected node is sent to the next idle Node Solver. After a successful selection, variable settings have to be adjusted according to the information stored in the Branch-and-Cut tree. If it turns out that some variable must be set to 0 or 1, yet has been fixed to the opposite value in the meantime, we have a contradiction. In this case the node is fathomed. If the local lower bound `llb` of the selected node is greater than or equal to the global upper bound `gub`, we fathom the

Algorithm for the Master

```
initialize_algorithm;
start_node_solver with the root problem;
do
  { wait for a message from the APS or a Node Solver;
    switch (received_message)
      { case new_gub:                          // from the APS
           store new gub;
           augment_sparse_graph;
           store_edge_list for each busy Node Solver;
           if (guarantee_reached)
             global_stop = true;
           break;

        case new_llb:                          // from a Node Solver
           determine_global_lower_bound;
           if (guarantee_reached)
             global_stop = true;
           break;

        case update_request:                   // from a Node Solver
           send_back_update_information; break;

        case satisfied:                        // from a Node Solver
           global_stop = true; break;

        case branch, fathom, contradiction:    // from a Node Solver
           receive information shown in table 1;
           fathom or branch the node (received_message);
           determine_global_lower_bound;
           while (there is an idle Node Solver and
                    the Branch-and-Cut tree is not empty)
               { select_new_node;
                 if (no more node)
                   { if (all Node Solvers idle) global_stop = true; }
                 else
                   start_node_solver with the selected node;
               }
           break;
      }
  }
  while (global stop == false);
  tell_all_processors_to_stop;
```

node immediately without sending it to any Node Solver and continue with the selection process. A Branch-and-Cut node has pointers to its parent and its two children. So it is sufficient to store a set variable only once in any path from the root to a leaf in the Branch-and-Cut tree. If a new problem is selected, only the highest ancestor of the old node and the new leaf has to be determined. The set variables on the path from the old node to the common are then reset, while the set variables on the path from the common ancestor to the new leaf are set.

- `tell_all_processors_to_stop`
  At the end, the Master sends a `STOP` message to all other processors. They print some statistics and stop immediately whatever they were working on.

4.3 Parallel Pool

During the whole computation, we keep a pool of active and nonactive facet defining inequalities of the traveling salesman polytope. The *active* inequalities are the ones that are in the LP of a Node Solver and both stored in the pool and in the constraint matrix of the corresponding Node Solver, whereas the inactive constraints are only present in the pool. An inequality becomes *inactive*, if it is not binding in any of the LP solutions of the Node Solvers. When required, it is easily regenerated from the pool and made active later in the computation.

The pool can only provide its full power if all constraints are available to all Node Solvers. Therefore, a straightforward implementation could make one processor maintain the pool and answer the requests (also called *jobs*) coming from the Node Solvers. However, the main feature of the pool is the pool separation, i. e. finding new violated inequalities without being forced to call a separator. This pool separation has to be fast if it shall be worthwhile. Furthermore, many Node Solvers are requesting for the pool separation, which could lead to a bottleneck. Therefore, in order to make the pool separation as efficient as possible, we "cut" the Pool into several pieces and put each piece onto a different processor. This leads to the structure of the parallel Pool shown in figure 1.

The Poolmaster is the interface between the Poolslaves and the other processors (APS, Node Solvers and Master). It holds the information on which Poolslave which inequality can be found and knows, what the Poolslaves are doing at every moment (this is done by a class `poolslave_manager` that stores for each Poolslave what job it is working on, or if it is idle). Furthermore, it holds a queue of the incoming requests and decides, which job is the next to be executed. This is done as follows. Each type of request has a priority. As soon as a request is coming, it is stored in a priority queue at a position according to its priority. Whenever it may be possible that a request is executable (this is the case when either the request came into the queue or a Poolslave stopped its current job), the queue is scanned for executable jobs, starting at the node carrying the job with the highest priority and stopping if an executable job was found or all jobs were scanned. Some jobs need all Poolslaves for execution, some just occupy one Poolslave. If two jobs do not need the same Poolslave (e. g. because the requested constraints lie on different slaves), they can be executed in parallel.

The algorithm running on the Poolmaster is shown on page 16.

The following functions are supported by the parallel Pool. They are listed in decreasing order of priority:

- Initialization
  For the initialization of the parallel Pool the Poolmaster receives the maximum number of constraints (from the Master). This number is sent to each Poolslave, who then calculates its own local number of constraints out of the global maximum and allocates space according to this local number. Initialization has the highest priority and is executed by all Poolslaves.

- Insertion of an inequality
  The Poolmaster receives the inequality to be stored. Then it immediately checks whether the inequality has to be stored at all, because it may be the case that it is already in the pool. To this end, a hashing number is computed for the inequality. This number is looked up in a hash table that contains the numbers of all constraints in the pool together with their constraint IDs. If the constraint does already exist, its constraint ID is sent back immediately, and the request is not stored in the queue at all. Otherwise, the request is stored in the queue. It is executable as soon as one Poolslave is idle. When an insertion request is executed, the sender of the inequality gets back a new constraint ID. Whenever the sender needs access to this constraint in the future, it has to specify this constraint ID to identify the constraint. Then the inequality is sent to the Poolslave that has the smallest load and is idle. In our implementation the Poolslave with the smallest load is the one that holds the smallest number of constraints. Another possibility is to measure the load as the sum of the sizes of all stored constraints.

Algorithm for the Poolmaster

```
initialize_internal_data_structures;
do
  { wait for a message from any processor;

    switch (type of sender)
      { case Node Solver, APS, Master:
          if (job is executable immediately)
            execute_job;
           else
            store_job_in_prio_queue;
          break;

        case Poolslave:
          look up, which job the Poolslave has been executing;
          receive result according to the job and forward it if necessary;
          if (Poolslave is last one working on this job)
            remove_job;
          break;
      }

    do
      { scan the priority queue for an executable job with highest priority;
        if (found)
          send job and necessary data to all participating Poolslaves;
      }
    while (an executable job could be found);

  }
while (parallel pool is not stopped by Master);

print_statistics;
```

The insertion has a high priority, because it is important to have constraints in the pool as soon as possible. A pool separation already stored in the request queue might find them violated for the given fractional solution.

- Extraction of a constraint
  The Poolmaster receives the constraint ID of the demanded constraint, forwards the request to the respective Poolslave and sends the constraint back to the requester on receipt from the Poolslave.

- Deletion of an inequality
  Having received the constraint ID of the inequality to delete, the Poolmaster can execute this request when the Poolslave that carries this constraint gets idle.

- getting and setting constraint elements
  A constraint in the pool has the following entries:

  - `constr_len`
  - entry-array
  - `constr_ID`
  - `rhs`
  - `n_active`
  - `lp_index`

  `constr_len` just stores the length of the entry-array. For a more detailed description of the entry-array see [JRT94].

  `constr_ID` stores the constraint ID of the constraint, `rhs` is the right hand side. The element `n_active` holds the number of active Branch-and-Cut nodes where this constraint is active. The `lp_index` is an array that stores for each Node Solver which row of the constraint matrix of the LP the constraint lies in, or 0, if it is not in the LP of the corresponding Node Solver.

  All these elements can be demanded. Therefore, the constraint ID and the desired entry type have to be sent. The Poolmaster looks up in its constraint table where to find the constraint and forwards the request to the respective Poolslave. This Slave sends back the entry to the Poolmaster who forwards the result back to the requester.

The elements `n_active`, `rhs` and the Node Solver's own `lp_index` can be set by a similar setting request. Here also the new value has to be sent.

- Cleanup
  If the pool is getting too full, cleanup requests can tell the Pool to delete unnecessary constraints. In a *soft cleanup* the Poolmaster tells the Poolslaves to throw away all those constraints whose `n_active` entry and all `lp_index` elements are equal to 0, i. e., all inactive constraints. In a *hard cleanup* (used if the soft cleanup did not delete enough constraints) also those constraints are deleted that have their `n_active` entry greater than 0 while all the `lp_index` elements are equal to 0. These are those constraints that are currently not active in any LP, but that were active in the last LP of the parents of active Branch-and-Cut nodes.

  In our implementation the Master checks the load of the pool every 20th update request and calls a soft cleanup if more than 85% of all constraint slots are occupied. A hard cleanup is called if the soft cleanup did not delete a single constraint at all.

- FORALL requests
  A special issue can be used to speed up the communication in cases where elements of all constraints in the Pool are needed. A Node Solver can send a `FORALL` flag followed by the request for an element of the Pool (e. g. the size of a constraint). The Poolmaster then sends back the element-constraint ID pair of all constraints in the Pool. With this method of getting information out of the parallel Pool a lot of overhead can be avoided. Without the `FORALL` flag, the requester has to send a request for each constraint in the Pool. Using the flag, only two messages (the flag and the request for an element) are necessary to get the same information.

- Pool separation
  For the pool separation a fractional solution is needed. This is sent to all Poolslaves participating in the pool separation. All separated constraints are sent back to the Poolmaster and collected here. As soon as the last Poolslave has sent its constraints, the whole constraint list is sent back to the requester (one of the Node Solvers).

  A pool separation is not really necessary for the correctness of the Branch-and-Cut algorithm. It helps the Node Solvers raise their local lower bounds.

Therefore, the pool separation need not be executed by all Poolslaves. Hence it is considered executable as soon as 75% of the Poolslaves are idle. By this technique large waiting times for the Node Solvers are avoided.

Some small requests can be answered by the Poolmaster immediately. These are questions about the maximum number of constraints storable in the parallel Pool, about the number of constraints stored currently and whether a constraint with a given constraint ID exists in the pool.

The Auxiliary Problem Solver has a link to the Poolmaster to send newly separated constraints to it. These are stored on one of the Poolslaves just like every other constraint.

One problem arises when several processors have access to the parallel Pool. When a Node Solver inserts a new constraint, this is marked inactive initially (by setting `n_active` and all `lp_index`es to 0). It becomes active when the corresponding Node Solver sets its `lp_index`. This is normally happening directly after the insertion. However, it may be that between these two requests (insertion and setting the `lp_index`) the Master calls a soft cleanup. Then the new constraint will be deleted although it is needed directly afterwards. Therefore, all constraints coming from the Node Solvers get a `protection` flag that protects them from being deleted. As soon as the `lp_index` element gets set, this protection is cancelled. By this means an accidential deletion of constraints is avoided. Constraints coming from the APS do not get this flag, because they will not be needed in an LP immediately.

4.4 Auxiliary Problem Solver (APS)

Generally, the Auxiliary Problem Solver is supposed to serve as a "device" that provides global bounds (upper bounds for a minimization problem and lower bounds in a maximization problem) and runs other algorithms that help shorten the overall running time and that can and should be executed independently from the rest of the Branch-and-Cut algorithm. It is written with regard to a general use for problems that can be solved with the Branch-and-Cut method.

In our case, the APS has the following two tasks:

(1) finding traveling salesman tours and thereby global upper bounds,

(2) separating new inequalities.

Job (2) is mainly done by the Node Solvers in their separation phase. However, our separation algorithm for comb constraints and clique tree constraints takes too much running time to be called in every separation phase. Therefore, in order to get these constraints despite this fact, we call the separator on the APS-Slaves.

The structure of the Auxiliary Problem Solver is the same as that of the parallel Pool: a Master (here called *APS-Master*) receives messages from "the outer world" and holds them in a queue. The *APS-Slaves* do jobs ordered by the APS-Master.

4.4.1 APS-Master

The "messages" are the fractional solutions received from the Node Solvers when their LP solving phase is over. These fractional solutions are stored in a queue and get a priority corresponding to each job type (tour heuristics or separation).

As soon as an APS-Slave gets idle, the following steps are executed by the APS-Master:

- determine which type of job shall be executed according to some "outer" strategy described below,
- find the fractional solution in the queue that has the highest priority for the chosen job type,
- send the chosen fractional solution to the idle APS-Slave together with information about job type and "inner" strategy.

When a fractional solution has been used once for all job types, it is deleted from the queue.

The priorities for the two types of jobs are calculated as follows. For tour heuristics, the priority is the number of ones in the fractional solution. We believe that the more ones a solution has, the higher is the probability that this solution leads to a good tour. Thus, the solution with the most ones is the candidate for the next tour improvement. Since no such priority can be given with respect to the separation, we use the same here as for the tour heuristics. This is done in order to be able to delete a fractional solution from the queue as soon as possible: a fractional solution that is chosen for tour heuristics will also be chosen for separation and can be removed afterwards.

The tasks to be distributed over the APS-Slaves have to be chosen in a resonable way. It may be desirable to modify the amount of calls for the given job types while the overall solution advances. Therefore, we implemented a strategy class that watches the process of finding bounds and changes the current strategy accordingly. This is done in the following way: The whole strategy for the APS consists of several so-called *outer strategies*. An outer strategy carries the information how many APS-Slaves (in percent) shall work on each of the job types. This can be for example 20% for the first, 30% for the second and 50% for the third job type. For each job type in each outer strategy there may be defined a list of several *inner strategies*. Inner strategies are strategies that concern the execution of the specific types of jobs. If, e. g., job type two in the example above has three different strategies, the inner strategies for this job type can be defined as 10% with strategy 1, 5% with strategy 2 and 15% with strategy 3. Thus, not only do the outer strategies define which percentage is used for each job type, but also how many APS-Slaves inside the job type shall work on which inner strategy.

Each outer strategy is coupled with a guarantee. This guarantee has to be fulfilled for the current best feasible solution in order to employ the corresponding strategy.

All strategy information can be defined by the user in a *strategy file*. This file is read by the APS-Master at the beginning of the algorithm.

In the implementation of our parallel Branch-and-Cut code for the TSP, we chose a strategy that we consider reasonable due to our experiences with the sequential TSP code. The need for the results from the two available job types shifts from finding global upper bounds to the separation of constraints while proceeding in the overall solution. At the beginning it is important to find good tours. Therefore short-time improvement heuristics are used to get first results. Often, a tour close to the optimal tour can be found rather early. Then the hard work is to find an optimal tour and to prove it by raising the lower bound. Therefore the tour improvement should be switched to heuristics consuming more time but being able to find tours of a better quality, and the separation of constraints should be reinforced as the algorithm advances.

A strategy file that worked well for our purposes is shown in table 2: the tour improvement starts with fast heuristics that find first good tours. The heuristics increase in complexity, taking more time but finding better upper bounds. Less processors work on the tour improvement as the program advances. The constraint separation is executed by a growing number of processors, generating increasingly more constraints.

We have five different strategies for the tour heuristics. The higher the number of the inner strategy the more complicated gets the heuristic and the longer is the corresponding running time. The separation of constraints has always the same strategy. Furthermore, if a Node Solver sends a feasible solution to the APS-Master

Table 2: TSP strategy file for the APS-Master

stra-tegy	**guar-antee**	**tour heuristics**				**comb and clique tree separation**
1	∞	20% 0	30% 1	30% 2	10% 4	10% 0
2	1.0%	30% 1	30% 2	20% 3	10% 4	10% 0
3	0.2%	10% 1	10% 2	50% 3	20% 4	10% 0
4	0.1%	10% 2	30% 3	20% 4		40% 0
5	0.05%	10% 2	10% 3	10% 4		70% 0

because a fractional solution was feasible, the current outer strategy is set to the last possible strategy. This is done, because feasible solutions from the Node Solvers are supposed to yield very good upper bounds, and therefore a strategy should be chosen that tries to prove the current best upper bound.

In figure 1 a connection between the APS-Master and the Master can be seen. This connection is used whenever an APS-Slave has found a feasible solution that is better (in terms of the TSP: a tour that is shorter) than the current best one. Then this solution is sent to the Master who will use it for further processing. In our TSP code the Master looks up its sparse graph and adds those edges of the tour that are currently not in the graph.

APS-Slaves that work on a separation job do not send their separated constraints to the APS-Master (who could forward them to the parallel Pool), but directly to the Pool. This connection is left out for clarity in figure 1. It speeds up the insertion of constraints into the Pool. The APS-Master only gets a message from a separating APS-Slave when the separation is over. Then this APS-Slave is known to be free for the next job, and the APS-Master will supply it with the task the current strategy orders.

The algorithm running on the APS-Master is shown on page 23.

4.4.2 APS-Slaves

The APS-Slaves wait for messages from the APS-Master. Each message consists of a flag that specifies the job that has to be done and corresponding information for that job. In our TSP code there are two possibilities:

- separation of comb and clique tree constraints: fractional solution with highest separation priority (see above)

Algorithm for the APS-Master

```
initialize_internal_data_structures;
do
  { wait for a message from any processor;

    switch (type of sender)
      { case Master:        // this message is received only once at the beginning
          receive initialization;
          initialize_global_data;
          break;

        case Node Solver:
          if (message is fractional solution)
            { store solution in the queue;
              while (an idle APS-Slave exists and a job is in the queue)
                send job to idle APS-Slave according to the current strategy;
            }
           else          // message is feasible solution
            if (solution is better than current best known)
              { store solution as best known;
                set outer strategy to last possible strategy;
              }
          break;

        case APS-Slave:
          if (APS-Slave finished)
            { look up, which job the APS-Slave has been executing;
              receive result according to the job;
              update strategy if necessary;
              if (a job is in the queue)
                send job to idle APS-Slave according to the current strategy;
            }
           else          // APS-Slave requests for best upper bound
            send back current global upper bound;
          break;
      }
  }
while (APS is not stopped by Master);

print_statistics;
```

- tour heuristics: fractional solution with highest tour priority, best known upper bound

The APS-Slaves starts the computations as soon as all necessary data is transmitted. Having finished the job, an APS-Slave gets back into the idle state and waits for the next message.

For the traveling salesman problem a host of heuristics is available. Usually, they are employed independent of lower bound computations. They are used to give a good feasible solution before a Branch-and-Bound algorithm is started. Then it is left to the Branch-and-Bound algorithm to find further tours and the optimum. In our opinion the fractional LP solutions occuring in the lower bound computations on the Node Solvers give hints on the structure of optimum or near optimum tours. Therefore, rather than running heuristics in an isolated way we constructed an algorithmic framework that integrates upper and lower bound computations: On the one hand fractional solutions are used to find upper bounds, on the other hand, feasible solutions are used to augment the sparse graph and the set of active variables of the Node Solvers.

A fractional LP solution contains a number of variables equal to 1, and some variables have values close to 1. We exploit this to form a starting tour for subsequent improvement heuristics in the following way. The edges are sorted in descending order according to their values in the current LP solution. The resulting list is scanned, and edges become part of the tour if they do not produce a subtour with the edges selected so far. This gives a system of paths which now have to be connected. To this end, a savings heuristic ([CW64]), originally developed for vehicle routing problems, is used.

Having computed a first tour we try to improve it by local modifications. Here we use variants of the three-opt heuristic and of the Lin-Kernighan heuristic ([LK73]). The basic principle is to build complicated tour modifications that are composed of simple moves where not all of these moves necessarily have to decrease the tour length. To obtain reasonable running times the effort to find the parts of the composed move has to be limited. This limitation depends on the current inner strategy and the progress of the improvement heuristics. If the length of the current tour is close to the best known solution, more CPU time is spent, and the modification possibilities of the Lin-Kernighan heuristic and the three-opt exchange are extended. Before employing time-consuming heuristics the APS-Master is queried about the current best upper bound. It may be the case that other APS-Slaves have found better tours in the meantime and a costly improvement heuristic has become worthless because it is anticipated that our limited heuristics will not yield a better solution within moderate time limits. Also, trying to improve the same tour several times is avoided by using a hashing scheme for the detection of identical tours.

As soon as a heuristic yielded a tour that is shorter than the best known one, this is sent to the APS-Master. Thereby, other processors get the new information very quickly and can take advantage of it.

For a full description of the exploitation of LPs see [JRT94]. Details about TSP heuristics and their computational performance can be found in [JRR95] and [Rei92].

5 Implementation

Regarding the structure shown in figure 1 we wanted to implement our parallel Branch-and-Cut algorithm on a massively parallel computer, whose diameter, i. e., the maximal distance between two processors, is as small as possible. A grid architecture for example would be unsuitable. Some processors communicate with nearly all others (e. g. the Master, the Poolmaster and the Node Solvers), and in this type of topology too much time would be spent by sending via other processors. A grid is much more useful for applications where the underlying problem can be mapped well into it and where the communication is restricted to the next neighbors.

Furthermore, a massively parallel computer with a lot of processors and a small memory on each processor is no adequate environment for our needs. A computer with a few hundred processors each having a bigger memory is much more useful.

With respect to these considerations, we implemented the parallel Branch-and-Cut code on Thinking Machines' CM-5 ([Thi92]).

5.1 Thinking Machines' CM-5

The CM-5 we used for our computations is a MIMD multiprocessor ([Fly72]) consisting of 64 SPARC-2 microprocessors with 32 MByte RAM on each processor. The topology of the so-called *data network* (the network used for the communication between two processors) is a *fat tree* (4-ary hypertree, s. figure 2 and [Qui94]).

The processing nodes are the leaves of the tree, whereas all other processors in the tree manage the communication. A 4-ary hypertree with depth d has 4^d leaves and $2^d(2^{d+1}-1)$ nodes in all. The diameter of this network is $2d = 6$ in our specific

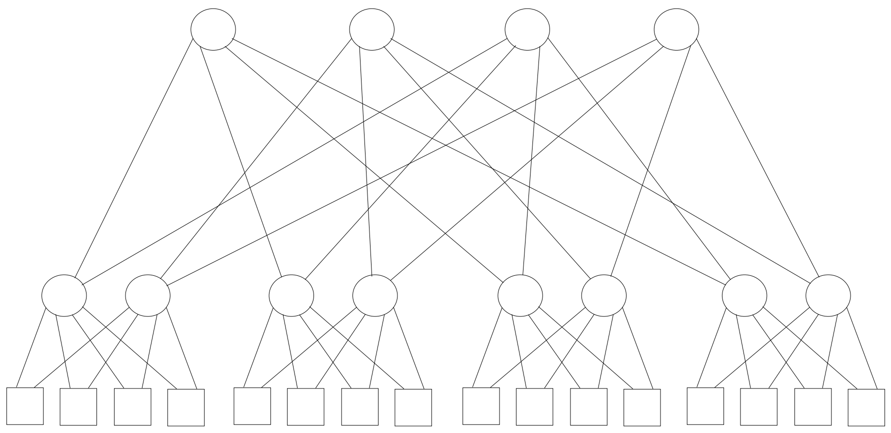

Figure 2: A fat tree (4-ary hypertree) with 16 leaves

case of 64 processing nodes. The 64 processors can be separated into partitions of sizes 16, 32 or 64.

The Branch-and-Cut algorithm was implemented in the C++ programming language ([Str93]), using the CMMD 3.2 message passing environment available for the CM-5 ([Thi94]), under the CMOST 7.3 operating system.

The program was compiled with the GNU C++ compiler with optimization option O3.

5.2 Computational results

We tested the parallel algorithm for several instances of the TSPLIB ([Rei91a, Rei91b]) on 16, 32 and 64 processors. In order to get a reference for speedup considerations, we ran our sequential Branch-and-Cut code (written in C) on one processor.

All computational results are shown in tables 4 and 5. The larger problems were only tested on 32 and 64 processors. Very small problems were run on 16 processors. The strategy of the APS was shown on page 22. The distribution of Slaves and Node Solvers can be seen in table 3.

The instances `rat575`, `d657`, and `u724` could not be solved because of a lack of memory on the processors. As one can see also in the figures 3 through 5 we gain a good speedup especially for the larger problems. Small instances yield only a small speedup, because the possibility to solve several Branch-and-Cut nodes in parallel is very small, and the number of pool separations is low.

The speedup for 32 and 64 processors is better than the one for 16 processors. However, the difference between running times on 32 processors and on 64 processors is not very large. Often 32 processors took less time than 64 processors. This results from the fact that the number of Branch-and-Cut nodes active at one time (i. e., nodes that can be computed parallely) is often not larger than 15. Then the remaining Node Solvers in the partition of 64 processors stay idle all the time. This means that the 32 processor partition and the 64 processor partition actually used the same amount of Node Solvers. The differences in the running times then result just in small differences that come from constraints that were added by the APS-Slaves sooner or later.

The problem instance `p654` is a bit pathological. Here, the global lower bound is equal to the optimum value very soon in the root node, and then the algorithm takes its time in searching for the corresponding tour. It is just a matter of luck, when this tour is found by the APS-Slaves, and in our case the 16 processor partition was more lucky than the 64 processors, and 32 processors gave the best result.

Unfortunately, no larger problem instances could be solved because of the memory limitations of the CM-5 we used. The results show that the larger the instances get, the more reasonable is the use of many Node Solvers.

Table 3: Distribution of Slaves and Node Solvers on different partitions

16 processors:	32 processors:	64 processors:
5 Node Solvers	13 Node Solvers	30 Node Solvers
5 APS-Slaves	8 APS-Slaves	16 APS-Slaves
3 Poolslaves	8 Poolslaves	15 Poolslaves

Table 4: Computational results comparing the sequential B&C and the parallel B&C algorithm

Probl.	**seq. B&C**		**parallel B&C**			$\frac{par\ time}{seq\ time} \cdot 100$ [%]
	time	#nodes	#procs	time	#nodes	
pr76	15:48	263	16	5:28	309	34.6
			32	3:41	251	23.3
			64	4:54	245	31.0
gr120	0:28	3	16	0:14	3	50.0
bier127	0:16	1	16	0:08	1	50.0
pr152	3:27	13	16	2:12	39	63.8
			32	1:23	19	40.1
			64	2:46	19	80.2
rat195	7:50	39	16	2:16	17	28.9
			32	1:47	23	22.8
			64	1:52	23	23.8
d198	6:04	27	16	1:44	47	28.3
			32	1:40	65	27.5
			64	2:43	75	44.8
gr229	29:49	53	16	5:42	53	19.1
			32	2:11	23	7.3
			64	3:25	33	11.5
gil262	5:24	9	16	3:08	31	58.0
			32	3:01	17	55.9
			64	3:44	21	69.1
pr264	0:46	3	16	0:30	3	65.2
mat280	0:28	3	16	0:22	3	78.6
pr299	84:02	109	16	11:00	85	13.1
			32	7:17	87	8.7
			64	10:01	47	11.9
lin318	10:50	19	16	2:23	11	22.0
			32	3:20	19	30.8
			64	3:25	15	31.5
pr439	297:51	223	32	26:37	173	8.9
			64	25:25	131	8.5
pcb442	106:25	435	32	8:16	265	7.8
			64	7:52	173	7.4
att532	216:19	133	32	26:01	213	12.0
			64	25:09	199	11.6

Table 5: Computational results comparing the sequential B&C and the parallel B&C algorithm – continued

Probl.	**seq. B&C**		**parallel B&C**			$\frac{par\ time}{seq\ time} \cdot 100$ [%]
	time	#nodes	#procs	time	#nodes	
ali535	47:13	13	16	7:33	15	16.0
			32	6:58	17	14.8
			64	7:21	21	15.6
u574	65:15	7	32	14:38	81	22.4
			64	12:12	49	18.7
rat575	1158:05	829	—			
p654	15:09	55	16	4:59	51	32.9
			32	3:14	35	21.3
			64	5:34	85	36.7
d657	1718:57	713	—			
gr666	165:17	51	16	14:59	53	9.1
			32	24:45	213	15.0
			64	14:53	73	9.0
u724	3368:19	1189	—			
rat783	52:50	13	32	8:55	53	16.9
			64	6:46	9	12.8

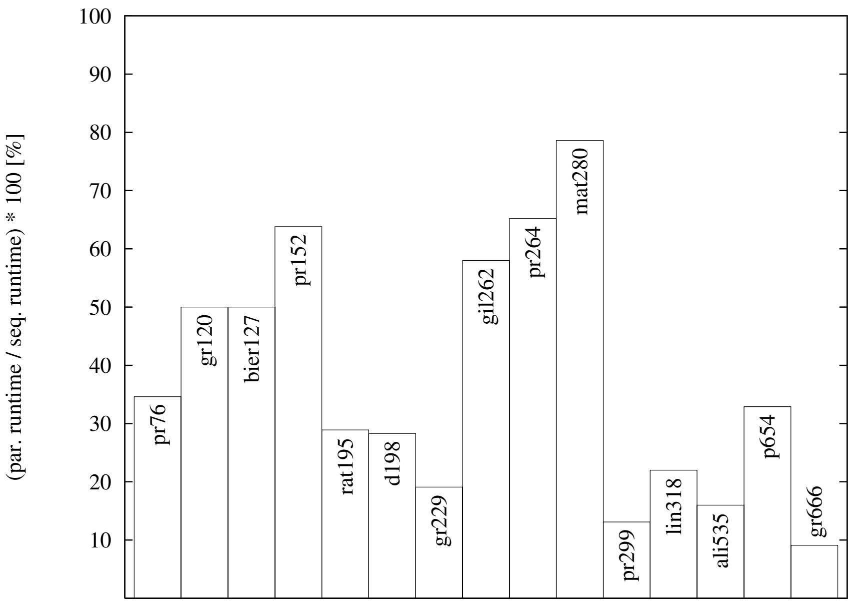

Figure 3: Computational results on 16 processors

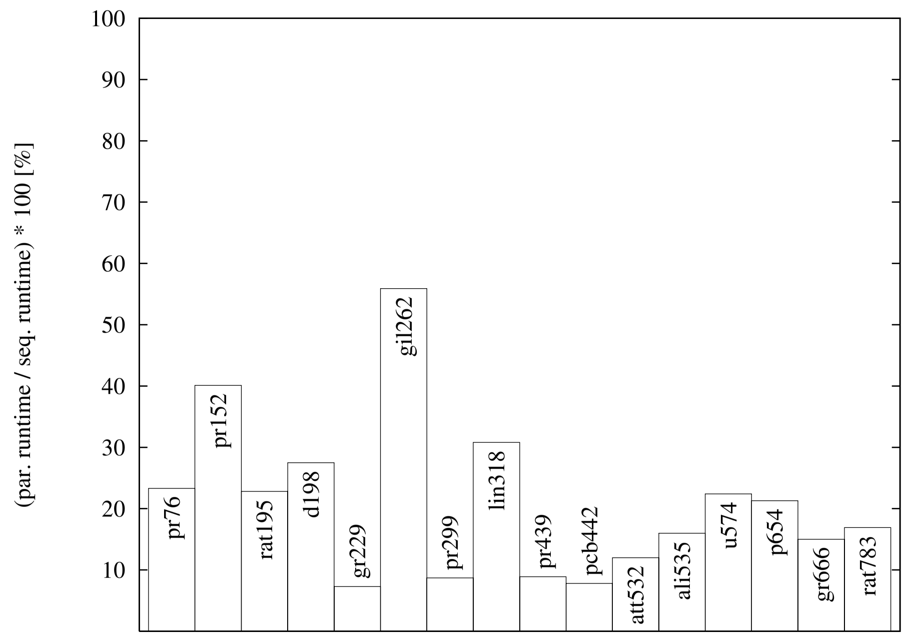

Figure 4: Computational results on 32 processors

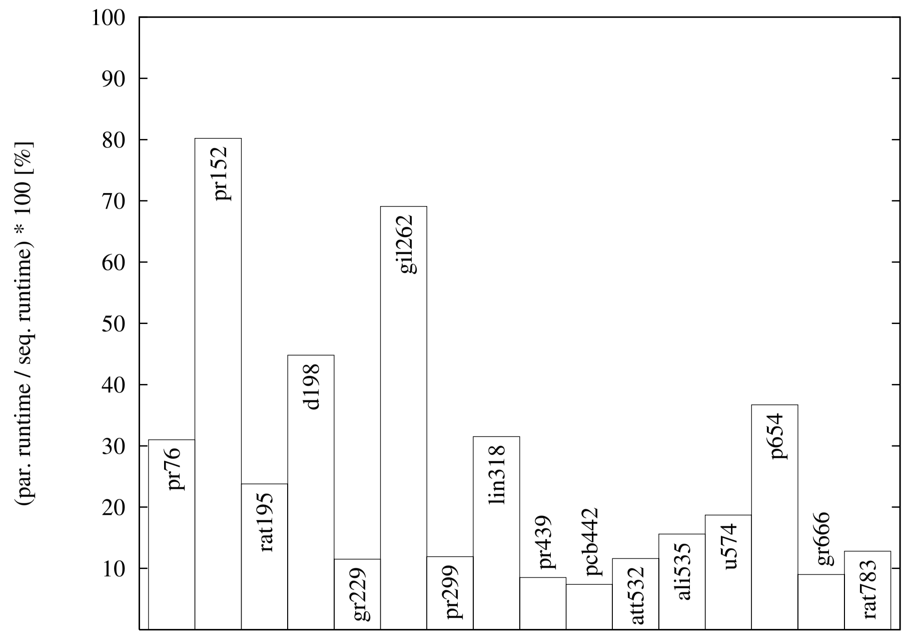

Figure 5: Computational results on 64 processors

6 Conclusion and future work

We introduced the implementation of a parallel Branch-and-Cut algorithm that distributes the overall work neither homogeneously nor in a simple Master/Slave correlation but in a more sophisticated manner that takes advantage of the several independent features of a Branch-and-Cut code. The computational results show that this implementation makes sense especially for large instances.

However, the current static work distribution, i. e., each processor is given its task (APS-Slave, Node Solver, etc.) in advance and works with it throughout the solution, is rather restrictive. It showed that a great amount of Node Solvers on the partition of 64 processors leads to more speedup with respect to 32 processors only for very large instances, since most of the Node Solvers are idle most of the time. Therefore, we are currently working on a dynamic work distribution. This will work as follows: Starting the algorithm one Node Solver will compute the root node, while the other processors are employed as usual. The amount of Node Solvers will then grow and shrink according to the actually necessary number due to branching and fathoming of Branch-and-Cut nodes. Node Solvers that run out of work can become either APS-Slaves or Poolslaves and vice versa. By this means we expect to gain an even larger speedup, since more processors are at hand for the computation of upper bounds and constraints or for a faster pool separation.

References

[AM88] Abdel-Rahman T. S. and T. N. Mudge: Parallel branch and bound algorithms on hypercube multiprocessors, in: *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, Pasadena, CA (1988).

[Akl89] Akl S. G.: The Design and Analysis of Parallel Algorithms, *Prentice-Hall* (1989)

[Bay72] Bayer R.: Symmetric binary b-trees: Data structure and maintenance algorithms, *Acta Informatica* 1 (1972) 290–306.

[BT98] Bertsekas D. P. and J. N. Tsitsiklis: Parallel and distributed computation: numerical methods, *Prentice-Hall* (1989).

[Can88] Cannon T. L.: Large-scale zero-one linear programming on distributed workstations, *Ph.D. dissertation, Department of Operations Research and Applied Statistics, George Mason University, Fairfax, VA* (1988).

[CH90] Cannon T. L. and K. L. Hoffman: Large-scale 0-1 linear programming on distributed workstations, *Annals of Operations Research* 22 (1990) 181–217.

[CW64] Clarke G. and J. W. Wright: Scheduling of vehicles from a central depot to a number of delivery points, *Operations Research* 12 (1964) 568–581.

[CLR90] Cormen T. H., C. E. Leiserson, and R. L. Rivest: Introduction to algorithms, *MIT Press* (1990) Cambridge.

[CP80] Crowder H. and M. W. Padberg: Solving large-scale symmetric traveling salesman problems to optimality, *Management Science* 26 (1980) 495–509.

[dRT88] de Bruin A., A. H. G. Rinnooy Kan, and H. W. J. M. Trienekens: A simulation tool for the performance evaluation of parallel branch and bound algorithms, *Mathematical Programming* 42 (1988) 245–271.

[Eck93] Eckstein J.: Parallel Branch-and-Bound Algorithms for General Mixed Integer Programming on the CM-5, Thinking Machines Corporation, Technical Report TMC-257 (1993).

[EH80] El-Dessouki O. I. and W. H. Huen: Distributed Enumeration on Between Computers, *IEEE Transactions on Computers C-29* 9 (1980) 818–825.

[FM85] Finkel R. A. and U. Manber: DIB – A distributed implementation of backtracking, *ACM Transactions on Programming Languages and Systems* 9 (1987) 235–256.

[Fly72] Flynn M. J.: Some Computer Organizations and their Effectiveness, *IEEE Transactions on Computers C-21* 9 (1972) 948–960.

[GK92] Grama A. Y. and V. Kumar: Parallel Processing of Discrete Optimization Problems: A Survey, University of Minnesota, Minneapolis, MN 55455 (1992)

[GH91] Grötschel M. and O. Holland: Solution of large-scale symmetric traveling salesman problems, *Mathematical Programming* 51 (1991) 141–202.

[GS78] Guibas L. J. and R. Sedgewick: A diochromatic framework for balanced trees, in: *Proceedings of the 19th annual symposium on foundations of computer science*, IEEE Computer Society (1978) 8–21.

[IFY79] Imai M., T. Fukumura, and Y. Yoshida: A parallelized branch-and-bound algorithm: implementation and efficiency, *Systems, Computers, Controls* 10, 3 (1979) 62–70.

[IYF79] Imai M., Y. Yoshida, and T. Fukumura: A parallel searching scheme for multiprocessor systems and its application to combinatorial problems, in: *Proceedings of the 6th International Joint Conference on Artificial Intelligence* (1979) 416–418.

[JRR95] Jünger M., G. Reinelt, and G. Rinaldi: The Traveling Salesman Problem, in: *M. Ball, T. Magnanti, C. L. Monma, and G. Nemhauser (eds.): Handbooks in Operations Research and Management Science, Vol. 7*, North Holland (1995).

[JRT94] Jünger M., G. Reinelt, and S. Thienel: Provably Good Solutions for the Traveling Salesman Problem, *Zeitschrift für Operations Research* 40 (1994) 183–217.

[KZ88] Karp R. M. and Y. Zhang: A randomized parallel branch-and-bound procedure, in: *Proceedings of the ACM Annual Symposium on Theory of Computing* 20 (1988) 290–300.

[KL86] Kindervater G. A. P. and J. K. Lenstra: An introduction to parallelism in combinatorial optimization, *Discrete Applied Mathematics* 14 (1986) 135–156.

[KG92] Kumar V. and A. Gupta: Analyzing Scalability of Parallel Algorithms and Architectures, revised version of November 1992, Technical Report TR 92R-003, Department of Computer Science, University of Minnesota, Minneapolis, MN (1991).

[KN87] Kumar V. and V. Nageshwara Rao: Parallel depth-first search. Part II. Analysis, *International Journal of Parallel Programming* 16 (1987) 501–519.

[KRN88] Kumar V., K. Ramesh, and V. Nageshwara Rao: Parallel best-first search of state-space graphs: a summary of results, in: *Proceedings of the AAAI-88 Seventh National Conference on Artificial Intelligence*, St. Paul, MN (1988).

[LSa84] Lai T.-H. and S. Sahni: Anomalies in parallel branch and bound algorithms, *Communications of the ACM* 27, 6 (1984) 594–602.

[LSp85] Lai T.-H. and A. Sprague: Performance of parallel branch-and-bound algorithms, *IEEE Transactions on Computers, C-34* 10 (1985) 962–964.

[LW84] Li G.-J. and B. W. Wah: Computational efficiency of parallel approximate branch-and-bound algorithms, in: *Proceedings 1984 International Conference on Parallel Processing* (1984) 473–480.

[LW86a] Li G.-J. and B. W. Wah: Coping with anomalies in parallel branch-and-bound algorithms, *IEEE Transactions on Computers, C-35* 6 (1986) 568–573.

[LW86b] Li G.-J. and B. W. Wah: How good are parallel and ordered depth-first searches?, in: *Proceedings 1986 International Conference on Parallel Processing* (1986) 992–999.

[LW90] Li G.-J. and B. W. Wah: Computational efficiency of parallel combinatorial OR-tree searches, *IEEE Transactions on Software Engineering 16* 1 (1990) 13–31.

[LK73] Lin S. and B. W. Kernighan: An effective heuristic algorithm for the traveling salesman problem, *Operations Research* 21 (1973) 498–516.

[LM89] Lüling R. and B. Monien: Two strategies for solving the vertex cover problem on a transputer network, in: *Proceedings of the Third International Workshop on Distributed Algorithms*, Nice (1989).

[LM92] Lüling R. and B. Monien: Load balancing for distributed branch and bound algorithms, in: *Proceedings of the International Parallel Processing Symposium*, Beverly Hills, CA (1992)

[LMRT92] Lüling R., B. Monien, M. Räcke, and S. Tschöke: Efficient Parallelization of a Branch & Bound Algorithm for the Symmetric Traveling Salesman Problem, Department of Mathematics and Computer Science, University of Paderborn (1992).

[MS90] Manzini G. and A. M. Somalvico: Probabilistic performance analysis of heuristic search using parallel hash tables, in: *Proceedings of the International Symposium on Artificial Intelligence and Mathematics*, Ft. Lauderdale, FL (1990).

[MRRT91] McKeown G. P., V. J. Rayward-Smith, S. A. Rush, and H. J. Turpin: Using a transputer network to solve branch and bound problems, in: *P. Welch, D. Stiles, T. L. Kunii, A. Bakkers (eds.): Transputing '91, IOS Press* Amsterdam (1991).

[Moh82] Mohan J.: A study in parallel computation - the traveling salesman problem, Technical Report CMU-CS-82-136, Computer Science Department, Carnegie-Mellon University (1982).

[NK87] Nageshwara Rao V. and V. Kumar: Parallel depth-first search. Part I. Implementation, *International Journal of Parallel Programming* 16 (1987) 479–499.

[PRa82] Padberg M. W. and M. R. Rao: Odd minimum cut sets and b-matchings, *Mathematics of Operations Research* 7 (1982) 67–80.

[PRi90] Padberg M. W. and G. Rinaldi: Facet identification for the symmetric traveling salesman polytope, *Mathematical Programming* 47 (1990) 219–257.

[PM92] Pekny J. F. and D. L. Miller: A parallel branch and bound algorithm for solving large asymmetric traveling salesman problems, *Mathematical Programming* 55 (1992) 17–33.

[PNR88] Pruul E. A., G. L. Nemhauser, and R. A. Rushmeier: Branch-and-bound and parallel computation: A historical note, *Operations Research Letters* 7, 2 (1988) 65–69.

[Qui94] Quinn M. J.: Parallel Computing: Theory and Practice, *McGraw-Hill* (1994).

[QD86] Quinn M. J. and N. Deo: An upper bound for the speedup of parallel best-bound branch-and-bound algorithms, *BIT* 26, 1 (1986) 35–43.

[Ran90] Ranade A.: A Simpler Analysis of the Kharp-Zhang Parallel Branch-and-Bound Method, Report UCB/CSD90/586, Computer Science Division, University of California, Berkeley, CA (1990).

[RRM93] Rayward-Smith V. J., S. A. Rush, and G. P. McKeown: Efficiency considerations in the implementation of parallel branch-and-bound, *Annals of the Operations Research* (1993)

[Rei91a] Reinelt G.: TSPLIB - A traveling salesman probllem library, *ORSA Journal on Computing* 3 (1991) 376–384.

[Rei91b] Reinelt G.: TSPLIB - Version 1.2, Report No. 330, Schwerpunktprogramm der Deutschen Forschungsgemeinschaft, Universität Augsburg (1991).

[Rei92] Reinelt G.: Fast heuristics for large geometric traveling salesman problems, *ORSA Journal on Computing* 4 (1992) 206–217.

[Rou87] Roucairol C.: A parallel branch and bound algorithm for the quadratic assignment problem, *Discrete Applied Mathematics* 18 (1987) 211–225.

[Str93] Stroustrup B.: The C++ programming language, second edition, *AT&T Bell Laboratories*, Murray Hill, New Jersey (1993)

[Thi92] Thinking Machines Corporation: The Connection Machine CM-5, Technical Summary, Cambridge, MA (1992)

[Thi94] Thinking Machines Corporation: CMMD Reference Manual, version 3.2, Cambridge, MA (1994)

[Vor86] Vornberger O.: Implementing branch-and-bound in a ring of processors, in: *Proceedings of the CONPAR 86: Conference on Algorithms and Hardware for Parallel Processing, Lecture Notes of Computer Science* 237, Springer Verlag (1986) 157–164.

[Vor87] Vornberger O.: Load Balancing on a Network of Transputers, *Distributed Algorithms 1987, Lecture Notes of Computer Science* 312, Springer Verlag (1987) 116–126.

[WM84] Wah B. W. and Y. W. E. Ma: MANIP – a multicomputer architecture for solving combinatorial extremum-search problems, *IEEE Transactions on Computers C-33* 5 (1984) 377–390.