Solving Large-Scale QAP Problems in Parallel with the Search Library ZRAM

Adrian Brüngger, Ambros Marzetta
Institute for Theoretical Computer Science
Swiss Federal Institute of Technology
CH-8092 Zürich, Switzerland

Jens Clausen
Dept. of Mathematical Modeling, Bldg. 321
Technical University of Denmark
DK 2800 Lyngby, Denmark

Michael Perregaard
DIKU, Dept. of Computer Science, University of Copenhagen
Universitetsparken 1
DK 2100 Copenhagen Ø,Denmark

Abstract

Program libraries are one tool to make the cooperation between specialists from various fields successful: the separation of application-specific knowledge from application-independent tasks ensures portability, maintenance, extensibility, and flexibility. The current paper demonstrates the success in combining problem-specific knowledge for the quadratic assignment problem (QAP) with the raw computing power offered by contemporary parallel hardware by using the library of parallel search algorithms ZRAM. Solutions of previously unsolved large standard test-instances of the QAP are presented.

1 Introduction

The goal of problem solving using systems with many processors working in parallel is to push the limits of the computable: solve problems faster than before, solve bigger problems, or ultimately solve previously unsolved problems. Obviously, the limits for a given problem type can be pushed the most if a combination of the fastest available hardware and the best solution methods is used. However, the problem specific expertise and the hardware and implementation expertise are seldomly found together. One way of relieving this problem is software libraries with clear interfaces to the user—this enables the user and the library builder to focus on their particular field of expertise. If in addition the library is designed for and used on a powerful platform such as a supercomputer, one would expect a substantial increase in solution capability for the problem in question.

The current paper address such a combination of solution methods, high performance hardware, and a software library. The problem addressed is the Quadratic Assignment Problem (QAP), one of the hardest among the $\mathcal{NP}$-hard combinatorial optimization problems. We have combined the specific QAP knowledge gained over the years represented in the form of the parallel code leading to the first solution of the classical Nugent 20 benchmark, and the raw computing power offered by state-of-the-art parallel hardware (the NEC Cenju-3 and the Intel Paragon) using ZRAM, a portable parallel program library for exhaustive search problems.

In the following, we briefly describe both the problem-specific methods used and the interface to the Branch-and-Bound engine of ZRAM. To estimate the resources in terms of time and search tree size necessary to solve a given instance, a tree-size estimator has been a useful tool–this is also described. The properties of the parallel algorithm are briefly mentioned. Although no fine tuning of the parallel code provided has been performed, good speedups as well as minimal parallel search overhead have been observed. Finally, the solutions of 10 previously unsolved QAP benchmark instances from QAPLIB are reported. Parallelism reduced the solution time for the largest of these from (estimated) 2 years to 12 days.

2 Branch and Bound as an interface between two worlds

A general algorithmic paradigm to find a minimum of the objective function for an $\mathcal{NP}$-hard combinatorial optimization problem defined on a space of exponential size is Branch and Bound. When Mitten [13] gave a formal description of Branch and Bound, he clearly identified the interface between the problem-specific parts and the problem-independent parts of the search algorithm. Branch and Bound can be viewed as the interface between the two worlds of application-specific knowledge and problem-independent tree search. This approach identifies four problem-dependent functions for Branch and Bound:

1. **The branching rule**
   The branching rule recursively divides the space of solutions into a number of disjoint subspaces thereby defining a search tree T, where each node represents a (sub-) problem instance to be solved.

2. **The bounding procedure**
   In principle, T could be explicitly computed. However, in most cases, T contains an exponential number of nodes making this impossible. By computing a lower bound on the objective function in a node of T, the expansion of the respective subtree can be omitted whenever the lower bound exceeds a currently known best solution, and a cut-off occurs.

3. **The solution test**
   The solution test simply determines whether an optimal solution has been found for a subproblem. Whenever this is the case, the recursive division of the state space stops.

4. **The upper bounding procedure (optional)**
   In order to get good initial solutions, thereby increasing the probability that cut-offs occur, an upper bounding procedure can be provided. It produces a feasible solution to the problem in a node of T. This solution is compared to the currently best known solution and could replace it.

There are several possibilities for traversing the search tree: depth-first (*DFS*), breadth-first (*BFS*) or best-first (*BeFS*, where the nodes are expanded according to lower bounds; the one with the smallest lower bound is expanded first). *DFS* has a linear space-complexity in the depth of the search tree since it stores the subproblems on a stack. *BFS* must store a FIFO queue and *BeFS* must store a priority queue of active nodes, both having exponential worst-case size in the size of the input. The drawback of the preferable space requirement of *DFS* is that *DFS* potentially expands too large a number of nodes. It can easily be seen that *BeFS* expands a minimum number of nodes of T until it finds the optimum node (assuming that the lower bounding function increases along each path from the root to a leaf in the search tree). If we, however, know the optimal solution value in advance, *DFS* also expands only the minimum number of nodes of T. So if we have a high-quality heuristic producing near-optimal or even optimal solutions, the application of *DFS* is strongly suggested. Recent research [9] indicates that *DFS* may be competitive also when no near-optimal solution is known, the reason being that *BeFS* is inefficient when it comes to fast identification of good feasible solutions.

3 The Quadratic Assignment Problem

The Quadratic Assignment Problem is often formulated as the problem of assigning n given facilities to n given locations such that the total cost of the assignment is minimized. The total cost has a a) fixed contribution from the assignments of the facilities (e.g. the cost of establishing each facility in the chosen location) and b) a variable contribution from the interaction between each pair of facilities depending on the location of the involved facilities. Hence in its most general form, QAP is given by a two-dimensional matrix C with c_{ik} equal to the fixed cost of assigning facility i to location k, and a four-dimensional matrix G with $g_{ikjl} \geq 0$, where g_{ikjl} corresponds to the variable cost of assigning facility i to location k and facility j to location l:

$$
\begin{aligned}
QAP(G,C) := \quad & \min \quad \sum_{i=1}^{n}\sum_{k=1}^{n}\sum_{j=1}^{n}\sum_{l=1}^{n} g_{ikjl}x_{ik}x_{jl} + \sum_{i=1}^{n}\sum_{k=1}^{n} c_{ik}x_{ik} \\
& \text{s.t.} \quad \textstyle\sum_{i=1}^{n} x_{ik} = 1, \qquad k \in \{1,...,n\} \\
& \qquad\ \ \textstyle\sum_{k=1}^{n} x_{ik} = 1, \qquad i \in \{1,...,n\} \\
& \qquad\ \ x_{ik} \in \ \{0,1\}, \qquad i,k \in \{1,...,n\}.
\end{aligned}
$$

Most research on QAP has focused on the Koopmans–Beckman version of the problem, in which the variable costs can be broken into a flow component and a distance component. For each pair of facilities (i, j) a flow of communication $f(i, j)$ is known, and for each pair of locations (k, l) the corresponding distance $d(k, l)$ is known. The transportation cost between facilities i and j, given that i is assigned to location k and j is assigned to location l, is then $f(i, j) \cdot d(k, l)$.

Each feasible solution corresponds to a permutation of the facilities, and letting S denote the group of permutations of n elements, the problem can hence in this case be stated as

$$\min_{\pi \in S} \sum_{i=1}^{n} \sum_{j=1}^{n} f_{i,j} \cdot d_{\pi(i),\pi(j)}$$

Initially no facilities have been placed on a location, and subproblems of the original problem arise when some but not all facilities have been assigned to locations. The number of feasible solutions grows exponentially: For a problem with n facilities to be located, the number of feasible solutions is $n!$, which for $n = 20$ is appr. 2.43×10^{18}.

Next we briefly review each of the four components of a Branch and Bound algorithm mentioned in the preceding section for the QAP application. Our description follows [7].

3.1 The Gilmore–Lawler Bound

The Gilmore–Lawler Bound GLB is based on the idea of joining exact cost information and simple bound information for the cost of assigning facility i on location k into one matrix, L, which is then used as cost matrix in the solution of a linear assignment problem (LAP). The exact cost information consists of the costs which will be incurred independently of succeeding assignments, whereas the bound information relates to the costs which vary depending on the succeeding assignments.

Regarding the variable costs, these relate to the cost of communication between facility i to be located next and all other unassigned facilities. Sorting the flow coefficients from i to unassigned facilities ascendingly and the distance coefficients from location k to free locations descendingly, the scalar product of these two vectors is a lower bound on the cost incurred from the flow between i and the unassigned facilities. More formally, defining f'_i and d'_k to be the i-th row of F respectively k-th row of D with the diagonal elements f_{ii} respectively d_{kk} left out, the matrix $L = (l_{ik})$ is defined by

$$l_{ik} := \langle f'_i, d'_k \rangle_- + f_{ii} d_{kk} + c_{ik}. \tag{1}$$

GLB is then obtained by solving an LAP with cost matrix L as formalized in the following proposition.

Proposition 1 *Let F and D be symmetric $n \times n$ matrices and let $C \in \Re^{n \times n}$. Let L be the matrix with entries as defined in (1). Then*

$$QAP(F, D, C) \geq GLB(F, D, C) := LAP(L). \tag{2}$$

For non-root nodes of the search tree we proceed as follows. When no assignments have been made, the cost incurred by locating facility i on location k is $f_{ii} d_{kk} + c_{ik}$. Suppose now that facility j has already been assigned to location l. The contribution $f_{ij}d_{kl}$ must then be added to the exact cost information. This has to be done for all assigned facilities. The process can be seen as a modification of the initial linear costs c_{ik} leading to a QAP of reduced dimension, but with the same properties as the full problem. Note that the contribution originating in the flow between fixed facilities is 0 initially, and increases gradually with an increasing number of assigned facilities.

The complexity of calculating GLB is dominated by the solution of the linear assignment problem and is hence $O(n^3)$. However, note that if a QAP in the so called general form is considered in which the cost of communication between two facilities i and j assigned to two locations k and l cannot be broken down into factors f_{ij} and d_{kl}, then the bound calculation has complexity $O(n^5)$. The reason is that for each combination of unassigned facility and free location, an LAP has to be solved in order to find the corresponding coefficient of L. For problems in the Koopmans–Beckmann form, these LAPs have coefficients corresponding to products of flow coefficients and distances and the optimal solution can hence be found by computing minimal scalar products as described above.

3.2 The branching rule of Mautor and Roucairol

The processing of each node consists essentially of bounding and branching. Branching is performed as described in [12] supplemented with the forcing of assignments as described in [8]. Branching is performed on facilities and is based on the reduced cost information generated when solving the LAP with cost matrix L in the GLB calculation. If the sum of the GLB computed and the reduced cost of entry (i, k) is greater than or equal to the value of the current best solution, the GLB for the subspace generated by assigning facility i to location k will exceed the current best solution value thus implying that the optimal solution will not belong to the subspace. Hence a number of assignments can be ruled out. If a facility (or location) exists for which only one assignment permitting a solution better than the current best is left, a forced assignment takes place. After a forced assignment, a new GLB calculation is performed and the branching process is repeated. The process stops when each facility and each location has at least two possible assignments. The location i with fewest remaining possible assignments is then chosen for branching, and a new subproblem is created for each k among these by assigning i to k.

In addition the symmetry testing of [2, 12] is implemented. Note that even if branching on locations exactly as described above for facilities is possible, the symmetry testing requires that branching is performed on facilities.

3.3 Finding a first feasible solution

The first feasible solution to be used as upper bound can for QAP be found using any one out of a number of very efficient heuristics as e.g. simulated annealing and tabu search. These in general produce the optimal solution for the problem instances solvable to optimality. Thus, the solution by Branch and Bound can be seen as an optimality check. However, for other combinatorial optimization problems, such as the Job Shop Scheduling Problem, the situation is different, cf. [14].

For QAP we have used a variant of simulated annealing, which enabled us to find the optimal solution of all problems solvable to optimality by Branch and Bound within 5 seconds of parallel computing time. Details can be found in [8, 4].

4 The library ZRAM of parallel search algorithms

ZRAM [5] is a library of parallel search algorithms. Its layered architecture (Figure 1) makes it easily portable and extensible. It has four layers separated by clearly defined interfaces.

- The *hardware* layer hides all machine dependencies. ZRAM requires the underlying system to be capable of passing messages. A shared memory is not necessary, nor does ZRAM use any information about the topology of the underlying network. The MPI version of the hardware layer is portable to a broad range of machines including the NEC Cenju-3 and workstation networks.
- The *virtual machine* abstracts from the hardware and contains higher-level functionality convenient for programming parallel search algorithms. Among its features are dynamic load balancing, distributed termination detection and checkpointing. Computations can be interrupted and continued later with a different number of processors, which makes ZRAM suitable for long-term computations.
- The *layer of search algorithms and data structures* contains parallel search engines for branch and bound, reverse search [1] and backtrack as well as a general implementation of Knuth's tree size estimation [11].
- In the top layer, a variety of applications demonstrate the usability of the ZRAM interfaces. The quadratic assignment problem, the traveling salesman, the 15-puzzle and a vertex cover algorithm all use the Branch and Bound engine. Vertices of polyhedra, connected induced subgraphs, polyominoes and Euclidean spanning trees are enumerated by the reverse search engine. The *application layer* generally contains no explicit parallelism.

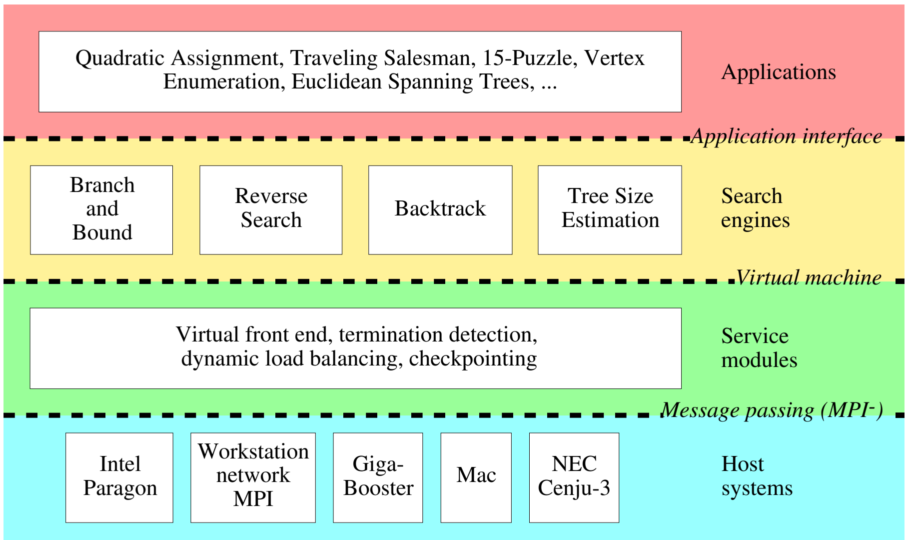
ZRAM is written in ANSI C, as this is the only language available on all the computers we use. ZRAM differs from other available search libraries (such as BOB, [10]) in the generality of the functions offered, in its layered structure and in the convenient application interface.

4.1 Dynamic load balancing

In practice, the size and shape of a search tree is unknown at the beginning of a computation, and hence it cannot at that time be partitioned into subtrees of equal size. Therefore, dynamic load balancing is necessary, i.e. redistribution of the work among processors during computation. Implementations of dynamic load balancing face a trade-off between processor utilization and communication overhead

A dynamic load balancing mechanism is provided by the virtual machine of ZRAM. It is used for depth-first Branch and Bound, reverse search and for the tree size estimator. In an abstract global view, the virtual machine manages just one distributed data structure: a

Figure 1: ZRAM architecture



global set of work units. For Branch and Bound, a work unit is a subtree represented by its root node. Every processor repeatedly removes a work unit from the global set, does whatever work it has to do, and inserts zero or more new smaller work units into the set. When the virtual machine detects that the set is empty, the algorithm terminates.

Viewed locally, every processor manages its own local list of work units. It can remove items from the list and insert others. When the list becomes empty, it sends an “I need work” message to some other randomly selected processor. If this second processor's list contains at least two elements, it sends some of these back to the requesting processor. Otherwise the second processor forwards the “I need work” message to a third processor. To keep the algorithm simple, the third and all succeeding processors are selected in a round-robin fashion rather than randomly. The statistics gathered during some QAP runs of several hours show that “I need work” messages are almost never forwarded except in the last few minutes before the program terminates. A simple and fast load balancing algorithm as described hence seems sufficient for coarse-grain parallelism.

4.2 Checkpointing and dynamic modification of the number of processors

The solution of a QAP instance can take several days on a powerful parallel computer, and hence the number of available processors may change, and system chrashes may appear during solution. Parallel combinatorial optimization software has to cope with such issues. It must be possible to interrupt a computation without losing a lot of work and to restart it later with a different number of processors from the same point it left off.

The virtual machine of ZRAM transfers data between processors in order to balance the load. It can as well transparently save the same data to disk. The global data, which are never modified, are saved once at the beginning of the computation. At regular intervals (every hour) during the tree search the computation is interrupted, the dynamic load balancing algorithm is brought into a known state, and the global set of work units is saved to disk. The virtual machine then calls a function in the search engine which saves the current upper bound. On machines supporting a signal facility, checkpointing can also be triggered by sending a signal to the process group before killing the job.

To restart the computation, the virtual machine first reads the global data and broadcasts it to all processors. It then reads the set of work units and redistributes it onto the new set of processors. Finally it calls a s earch engine function to read the current upper bound.

4.3 The estimator

Since we cannot predict the running times of a Branch and Bound algorithm by traditional complexity analysis, another approach is needed for estimating the resources such as running time involved in the actual solution of an instance. The "difficulty" of a given instance has to be estimated in order to decide whether it is solvable by the available algorithm in reasonable time or not.

Knuth [11] has introduced a tree-size estimator that allows a classification in this sense: by evaluating a relatively small number of paths of the search tree and computing the degree of every node on these paths, the size of the full tree can be estimated. The ZRAM implementation of this estimator has been (trivially) parallelized: every processor independently follows some number of paths in the search tree and collects the data needed. At the end of the computation, one processor gathers the results and computes the mean and standard deviation.

5 Computational results

5.1 Hardware: NEC Cenju-3 and Intel Paragon

In our computation we used two massively parallel computers: the NEC Cenju-3 and the Intel Paragon XP/S22. Both systems share the following basic properties:

- distributed memory
- SPMD (single program, multiple data)
- message passing

The Intel Paragon XP/S22 system consists of 148 compute nodes, each containing 3 Intel i860XP processors with a peak performance of 75 MFLOPS each (usually one is used for communication and two are real application processors). All nodes appear to be connected to all other nodes, and communication performance is uniform (the bisection bandwidth for the system used is 5.6 GByte/s). Physically, the nodes are arranged in a two-dimensional mesh. A fixed-function component at each compute node performs the actual routing of the messages.

The NEC Cenju-3 is a research prototype machine that contains 128 compute nodes (MIPS VR4400SC) with 64 MByte of local memory each. The peak performance reaches 50 MFLOPS per compute node. The compute nodes are connected via a multistage network based on 4x4 switches, achieving about 40 MByte/s communication bandwidth between two compute nodes.

5.2 Standard test instances from the QAPLIB

The algorithms were tested on a set of symmetric standard test instances retrieved from the QAPLIB [6]. We further constructed two new nugent-type instances nug21 and nug22 from the larger standard instance nug30. Both are now included in the QAPLIB.

5.3 Quality of the ZRAM estimator for the QAP

We have estimated the sizes of the search-trees for several non-trivial instances – all the instances we could solve that led to trees with more than 10 million nodes – and compared the estimates with the actual number of nodes generated when solving the instances. The sample size was 10 000 paths in the search trees. The number of nodes evaluated in the estimate was therefore less than 1 percent of the number of nodes evaluated in the solution. Figure 2 shows the accuracy of the estimates. The top curve shows the relative deviation of the estimated upper bound on the number of nodes (horizontal line, normalized to 1). The bottom curve shows the relative deviation of the estimated lower bound of nodes in the respective search trees. Note that the scale for the deviation is logarithmic. For most instances (nug21, rou20, nug17, tai17a, had18, tai20a, had16 and nug18) the estimates are very accurate; the actual number of nodes differ less than ten percent from the estimates. For some examples (nug22, esc16c and esc16d) the estimated upper bound is pessimistic, but the estimated lower bound is still very close to the actual number of nodes in the search tree. Only for three instances (had20, nug20 and esc16b) do the estimated number of nodes differ significantly from the actual number of nodes; the estimates are too conservative.

5.4 Load balancing

Load balancing prevents processors from starvation by feeding them with useful work while keeping the communication overhead as small as possible. The first characteristic variable to measure the parallelism overhead is the work done. When exploring a search tree in parallel, the evaluation order for the nodes in the tree differs inherently from the one occurring in a sequential implementation. New upper bounds are detected at different times. As a consequence, a parallel Branch and Bound algorithm usually expands a different number of nodes in a search tree than the corresponding sequential implementation. This is not the case for the QAP. Since the initial heuristic for problems solvable to optimality generates an optimal solution, no superfluous work (expansion of nodes with a lower bound that exceeds the solution cost) is done, except for the nodes in the search tree that have a lower bound equal to the optimal solution. However, to prevent heavy communication in the starting phase of the branch and bound search, ZRAM simultaneously expands the tree on all processors until the tree reaches a size where every processor can start working on its own part of the search tree. Therefore, the total number of expanded nodes increases with an increasing number of processors, but the increase in negligible, cf. Figure 3.

Figure 2: Quality of the ZRAM estimator for some hard QAP instances

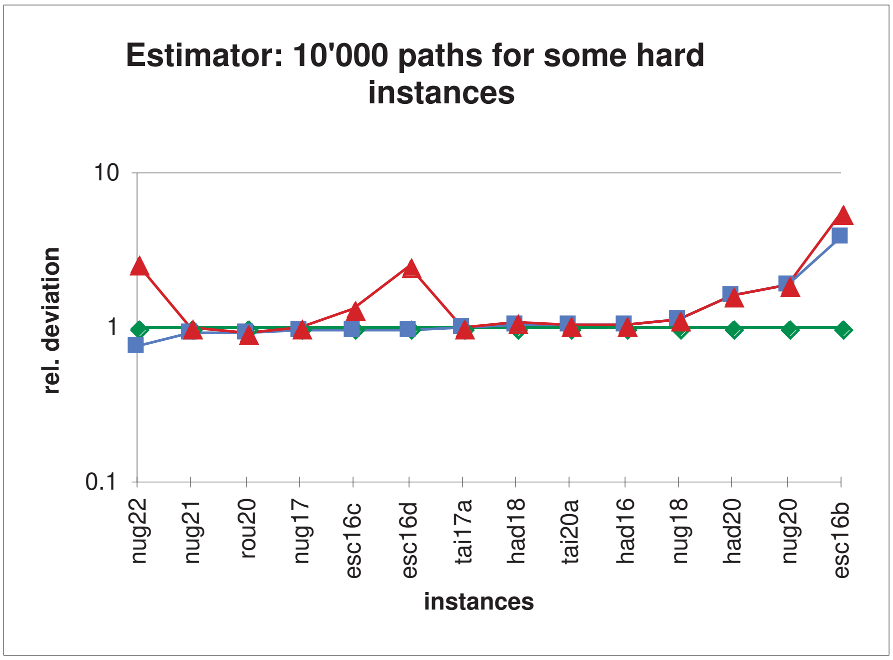


Figure 3: The parallel Branch and Bound evaluates an almost constant number of nodes when varying the number of processors.

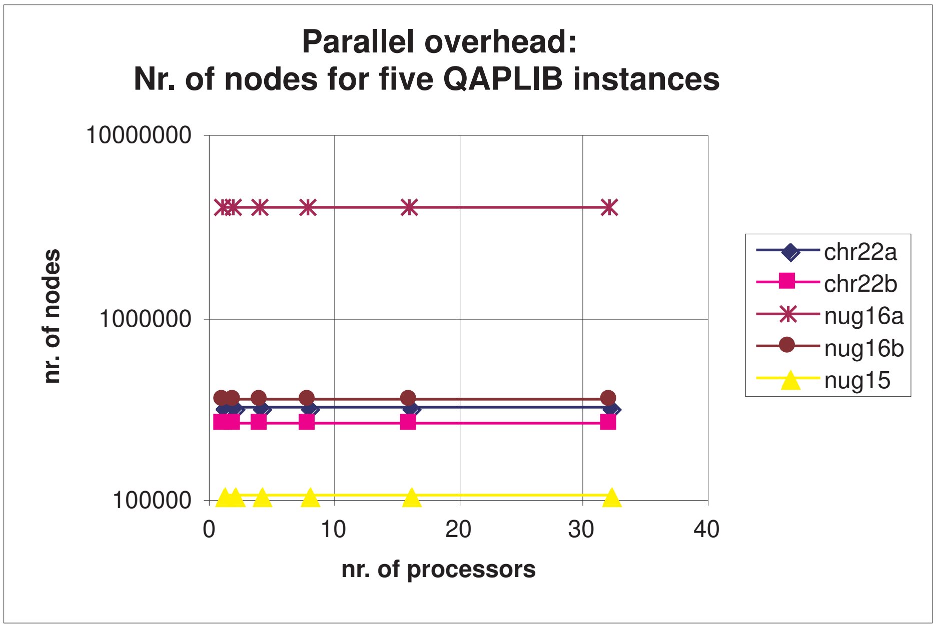
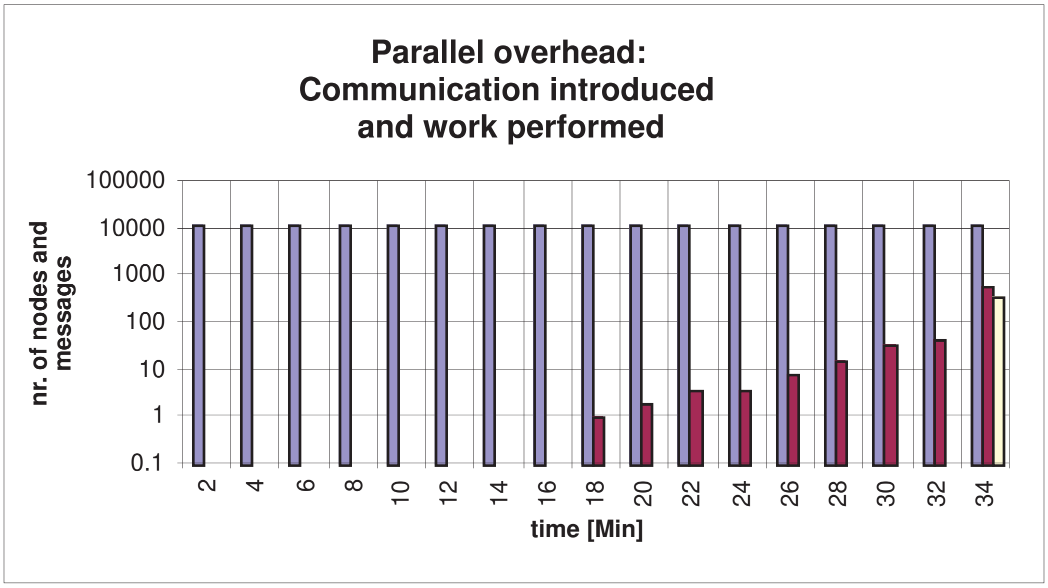


Figure 4: Work ratio and number of messages sent during the solution of nug17 on 16 processors.



The second characteristic overhead introduced with parallel Branch and Bound is the trade-off between communication and load balance. At the cost of communication, the load can be balanced between the processors. We measured the communication overhead by counting the number of messages that were sent during a given time period. Figure 4 shows the typical behavior of the ZRAM load balancer for a QAP instance (nug17 solved with 16 processors on the Intel Paragon). The x-axis indicates the time periods: the rightmost group of bars shows what happened in the last two minutes the algorithm was running. The leftmost bar in each group of bars shows the number of evaluated nodes per second in that time period. Note that the y-axis has a logarithmic scale. The next bar in the group gives the number of work request messages that were sent during the time period. The third bar gives the number of forwarded request messages. There is no communication at all at the beginning of the algorithm since all processors are busy, so no work requests are sent. After half of the total time has elapsed, some processors become idle and their work requests are immediately answered by other processors. In the very last phase of the tree search, most of the work requests have to be forwarded to a third processor since the processor which is asked for work is idle itself. The communication overhead does not slow down the progress of the algorithm at all: the rate at which nodes are evaluated remains almost constant during the whole running time of the algorithm.

5.5 Speedup

Although there are theoretical limitations on speedup [3], an almost linear speedup should be achievable for adequate problem size of well suited problems. QAP problems are ideal from this point of view. For the Intel Paragon with 32 processors, we achieved a speedup between 25 and 28.5 for five non-trivial instances. The actual running times for the instances varied from 413 seconds (nug16b with 362 768 nodes in the search tree) up to 4436 seconds (nug16a with 2 920 487 nodes in the search tree) on one processor. For further details see [4].

5.6 Solution of 10 previously unsolved QAP instances

The ZRAM estimator allows the classification of the difficulty of a collection of QAP instances accurately. Amongst the QAPLIB instances, 10 problems were identified to be solvable in reasonable time. Table I and Table II show the results. The largest instance solved is nug22 with about $5 \cdot 10^{10}$ nodes in the search tree. The solution took about 12 days with a varying number of processors involved. On a state-of-the-art single processor workstation, the computation would have taken about two years.

Table I: 10 previously unsolved instances from the QAPLIB

name	cost	hardware	procs	nodes	time [Min]
had16	3 720	NEC Cenju-3	32	18 770 885	4
had18	5 358	NEC Cenju-3	16	761 452 218	442
had20	6 922	Paragon	96	7 616 968 110	2875
tai17a	491 812	NEC Cenju-3	32	20 863 039	6
tai20a	703 482	Paragon	96	2 215 221 637	684
rou20	725 522	NEC Cenju-3	32	2 161 665 137	961
nug21	2 438	NEC Cenju-3	16	3 631 929 368	3213
nug22	3 596	NEC Cenju-3	48...96	48 538 844 413	12780
esc32e	2	NEC Cenju-3	32	12 515 753	10
esc32f	2	NEC Cenju-3	32	12 321 016	10

6 Conclusion

Cooperation between specialists in various fields of computer science - combinatorial optimization and development of parallel libraries for search problems - has led to the solution of previously unsolved QAP problems. The key issue to a successful combination of problem-specific knowledge and problem-independent tree search algorithms is the design of simple but well-defined interfaces. The usefulness of parallel search libraries has been demonstrated, and the clear layer structure of ZRAM has made the implementation of an application such as the QAP easy. Although ZRAM was not tuned for the special structure of the quadratic assignment problem, the introduced search overhead was minimal. Good speedups have been achieved while running times on one processor remained competitive. Since ZRAM is implemented on a variety of parallel machines including state-of-the-art hardware such as the NEC Cenju-3 and the Intel Paragon, the raw computing power offered by these machines was immediately available to solve several previously unsolved standard benchmark instances optimally.

Table II: 10 previously unsolved instances from the QAPLIB: Solution assignments

name	assignment
had16	8 3 15 0 6 7 5 10 14 13 11 9 4 2 1 12
had18	7 14 15 13 6 17 5 10 0 9 11 4 12 2 1 16 8 3
had20	7 14 0 13 18 5 6 16 15 11 9 10 4 19 1 2 3 8 17 12
tai17a	11 1 5 6 3 7 13 4 10 2 15 12 16 8 0 9 14
tai20a	9 8 11 19 18 2 13 5 16 10 4 6 14 15 17 1 3 7 12 0
rou20	0 18 1 13 9 15 10 19 8 4 6 3 7 17 14 2 11 16 12 5
nug21	3 20 2 8 12 1 4 13 17 10 15 9 5 14 19 18 7 6 0 11 16
nug22	1 20 8 9 6 2 0 18 7 19 16 4 12 5 11 15 10 21 17 3 13 14
esc32e	0 1 4 5 7 15 12 18 8 31 6 21 23 19 3 11
	2 16 28 20 10 24 26 17 29 30 22 27 13 14 25 9
esc32f	0 1 4 5 7 15 9 6 8 27 29 3 31 30 21 11
	2 16 25 17 12 24 28 20 22 23 18 19 13 14 26 10

Acknowledgments. This work has been supported by the SNSF (Swiss National Science Foundation) and ETH Zürich. We would like to thank NEC Corporation and CSCS/SCSC (Swiss Center for Scientific Computing) for letting us use their NEC Cenju-3.

References

[1] D. Avis and K. Fukuda, Reverse search for enumeration, *Discrete Applied Mathematics*, **65**, (1996) 21–46.

[2] M. S. Bazaraa and O. Kirca, A Branch-and-Bound-Based Heuristic for Solving the Quadratic Assignment Problem, *Naval Research Logistics Quarterly*, **30** (1983), 287–304.

[3] A. Brüngger, A parallel best-first Branch and Bound algorithm for the traveling salesperson problem, in *Proceedings of the 9th International Parallel Processing Symposium, Workshop on Solving Irregular Problems on Distributed Memory Machines*, S. Ranka ed., 1995, 98–106.

[4] A. Brüngger, A. Marzetta, J. Clausen, and M. Perregaard, Joining Forces in Solving Large-Scale QAP in Parallel, *Proceedings of IPPS '97, 11. IEEE International Parallel Processing Symposium*, 1997, 418–427.

[5] A. Brüngger, A. Marzetta, K. Fukuda, J. Nievergelt, The Parallel Exhaustive Search Workbench ZRAM and its Applications, to appear in *Annals of Operations Research.*

[6] R. E. Burkard, S. Karisch and F. Rendl, QAPLIB—A Quadratic Assignment Problem Library, *Journal of Global Optimization* **10** (1997), 391–403. Also available via WWW from `http://www.diku.dk/~karisch/qaplib`

[7] J. Clausen, S. Karisch, M. Perregaard and F. Rendl, On the Applicability of Lower Bounds for Solving Rectilinear Quadratic Assignment Problems in Parallel, DIKU Report 96/24, to appear in *Computational Optimization and Applications.*

[8] J. Clausen and M. Perregaard, Solving Large Quadratic Assignment Problems in Parallel, *Computational Optimization and Applications* **8** (1997), 111–128.

[9] J. Clausen and Michael Perregaard, On the Best Search Strategy in Parallel Branch-and-Bound - Best-First-Search vs. Lazy Depth-First-Search, DIKU report 96/14, to appear in *Annals of OR.*

[10] B. Le Cun and C. Roucairol, BOB: a Unified Platform for Implementing Branch-and-Bound like Algorithms, Tech. Rep., 95/16, Laboratoire PRiSM, Universite de Versailles - Saint Quentin en Yvelines, 78035 Versailles Cedex, France, 1995. Also available via WWW from `http://www.masi.uvsq.fr/english/parallel/cr/bob_us.html`

[11] D. E. Knuth, Estimating the Efficiency of Backtrack Programs, *Math. Comp.*, **29** (1975), 121–136.

[12] T. Mautor, C. Roucairol, A new exact algorithm for the solution of quadratic assignment problems, *Discrete Applied Mathematics* **55** (1994), 281–293.

[13] L. G. Mitten, Branch-And-Bound Methods: General Formulation And Properties, *Operations Research*, **18** (1970), 24–34.

[14] M. Perregaard and J. Clausen, Solving Large Job Shop Scheduling Problems in Parallel, to appear in *Annals of OR.*