Solving 0-1 Quadratic Problems with Two-Level Parallelization of the BiqCrunch Solver

Camille Coti *, Etienne Leclercq *, Frédéric Roupin *, Franck Butelle *
*LIPN, UMR 7030, Université Paris 13, Sorbonne Paris Cité
{coti,leclercq,roupin,butelle}@lipn.univ-paris13.fr

***Abstract*— In this paper we present MLTBiqCrunch, a hierarchically parallelized version of the open-source solver BiqCrunch [1]. More precisely, this version has two levels of parallelization: a coarse grain, assigning a thread to a node evaluation and a fine grain, parallelizing a node evaluation when some threads are not busy. We present experiments on some classical binary quadratic optimization problems with comparison of their scalability and raw performance. In particular, we obtain a superlinear speedup for some of the most difficult instances.**

I. Introduction

BiqCrunch [1] is a full open-source solver (publicly available online) for binary quadratic optimization problems. Such problems can be stated as 0-1 quadratic programs with m_I inequality constraints and m_E equality constraints:

$$\begin{cases} \max & z^T S_0 z + s_0^T z \\ \text{s.t.} & z^T S_i z + s_i^T z \leq a_i, \quad i \in \{1, \ldots, m_I\} \\ & z^T S_i z + s_i^T z = a_i, \quad i \in \{m_I + 1, \ldots, m_I + m_E\} \\ & z \in \{0,1\}^n \end{cases} \tag{1}$$

where the S_i's are real symmetric $n \times n$ matrices, the s_i's are vectors in $\mathbb{R}^n$, and the a_i's are real numbers. Note that if all $S_i = 0$ then one gets a 0–1 linear program. *BiqCrunch* requires the objective value of (1) to be integer for any feasible solution.

Many optimization problems can be stated as (1), for further details about applications and solvers the reader is referred to [2], [3]. A vast majority of solvers for continuous, mixed or integer problems, even to solve special cases (e.g. [4]) or relaxations of (1) (e.g. [5]) are multithreaded. Designing parallel versions is especially useful for Branch-and-Bound-like algorithms (e.g. [6]), and several authors investigated sophisticated approaches to take advantage of various architectures (e.g. [7], [8]). Other authors proposed approaches to provide a more general framework to design such parallel Branch-and-Bound algorithms (e.g. [9]). Some specific softwares are specialized to design this type of solvers, such as the COIN-OR High-Performance Parallel Search Framework [10] which provides a base layer of a hierarchy consisting of implementations of various tree search algorithms for specific problem types.

BiqCrunch uses sophisticated high-quality semidefinite bounds [11] and automatically sets the tightness of its bounding procedure node by node in the search tree. Moreover, triangle inequalities are dynamically added and removed from the underlying nonlinear relaxations in order to obtain stronger bounds. A complete description of the solver is given in [1] as well as its mathematical background. The *BiqCrunch* website is `http://lipn.univ-paris13.fr/BiqCrunch/`, where the source code, numerical results for several classical combinatorial problems and related papers can be downloaded. The distribution also includes converters and heuristics for some specific problems.

The evaluation of each node can be made independently from the other ones, making *BiqCrunch* a good candidate for parallel computing. However, the shape of the search tree developed by the branch-and-bound procedure does not immediately extract an optimal level of parallelism.

In this paper, we propose a two-level parallel execution, mixing parallel, low-level computation kernels and task-based, coarser-grained parallelism, to adapt the degree of parallelism at each level of granularity. After a quick review of the literature on related works, we describe *BiqCrunch* and how it can be parallelized in section II. We evaluate the performance exhibited by each level of parallelism, and its consequence on the overall performance (including the numerical effects of the reorganization of the computation) in section III. Moreover, we compare the new parallel version with the sequential version of the solver by solving three classical NP-hard combinatorial problems (Max-Cut, Max-Independent-Set, and Max-k-Cluster). Last, we discuss the results and open perspectives in section IV.

II. Multithreaded branch-and-bound

The choices we made for *MLTBiqCrunch* are inspired by previous works. For instance, a performance comparison is available in [12] between multi-core and many-core systems by solving big optimization problems with a Branch-and-Bound algorithm. Another branch-and-bound implementation is described in [8] using multi-GPU systems. While the previous papers are related to multi-CPU systems on one hand and to multi-GPU systems on another hand, [13] implements a Branch-and-Bound for heterogeneous architectures (both multi-CPU systems with GPU accelerators).

Nevertheless, the solver *BiqCrunch* has specific characteristics and features that should be taken into account. First, it was initially designed to be used on a standard personal computer, i.e. with a limited amount of memory and up to 8 cores. Second, the nonlinear relaxations used in *BiqCrunch* have a higher computational cost (from several seconds to several minutes) compared to other bounds used generally in Branch-and-Bound-like algorithms (such as linear programming for

instance). On the other hand, high-quality bounds are obtained here and therefore one can expect a small number of nodes to evaluate. Previous experiments with BiqCrunch2.0 show that actually, even for difficult combinatorial problems, this number is at most a few hundred. This means that the communication cost will be limited in a parallel version if the grain corresponds to one node evaluation.

However, the bounding procedure of *BiqCrunch* can be very fast since the quality of the relaxation is adjustable. Thus it may be hazardous to allocate many threads to evaluate a given node if other nodes are ready to be evaluated.

A. Single-threaded branch-and-bound

BiqCrunch is mainly written in C, and makes calls to Fortran libraries. The code actually makes heavy use of linear algebra functions (LAPACK [14] or the Intel Math Kernel Library (MKL)), it includes the nonlinear optimization routine L-BFGS-B [15], [16], and it is provided with an updated version of the branch-and-bound platform BOB [17]. Nevertheless, the current version of *BiqCrunch* uses only the serial features of the platform BOB (i.e. one core is used), although the latter is precisely designed to implement Branch-and-Bound-like algorithms that take advantage form the benefits of parallelism.

When branching on variable z_i in problem (1), the BOB branch-and-bound platform [17] creates two new subproblems (nodes of the search tree), one where z_i is fixed to 0 and the other where z_i is fixed to 1. The subproblem that has the weakest bound (among all the nodes previously inserted into the global priority queue) is then selected to be the next subproblem to branch on. In the case of a tie, BOB selects the subproblem which is lower in the search tree (i.e., having the larger number of fixed variables).

At iteration k of the bounding procedure, the algorithm computes a bound F_k of all the feasible solutions of the subtree, and takes advantage of the fact that the optimal value of the combinatorial problem is an integer. Hence, if $F_k < \beta_k + 1$, then the node of the branch-and-bound tree is pruned, where β_k is the current best feasible solution (since all feasible solutions of the subproblem have an objective value no better than β_k). If this is not the case, then the branch-and-bound tree needs to be explored further.

The bounding procedure of *BiqCrunch* enjoys some nice features. It can actually be fast to run if the node is easy to prune, but is also able to provide tighter but more expensive bounds if necessary. Moreover, it stops when it is likely that a bound which is lower than $\beta_k + 1$ cannot be reached within a reasonable amount of time. The bounding procedure can be stopped anytime and will always return a valid upper-bound for the problem, thanks to duality properties (see [11]). Therefore, the computation times to evaluate the nodes are bounded, and this bound can be chosen. In addition, generic or specific heuristics take advantage of the fractional solution computed by the relaxation to build a feasible solution for the initial combinatorial problem (1), in order to try improving the current best feasible solution. This is done several times in the bounding procedure (for further details see Section 4.2. and Algorithm 3 in [1]).

The *BiqCrunch* solver stores the input problem matrices in a sparse format in memory to keep its memory requirements small. Moreover the memory usage of the nonlinear optimization routine L-BFGS-B is very low and optimized. Typically, a problem with 225 variables and 32206 constraints (which involves a 226×226 symmetric matrix, i.e. 25425 variables, to store the underlying relaxation variables) requires at most 32 MB to be solved. In order to design a parallel version of *BiqCrunch*, thanks to this very limited amount of memory, allocating a private working memory space for each thread is a simple and still low-cost solution, even on a standard personal computer.

B. Multithreaded computation kernels

BiqCrunch uses linear algebra kernels intensively: in particular, profiling data showed that it spends about 60% of execution time in `dsyevr`, which is itself spending about 20% of the total execution time in `dsytrd`. Therefore, the most basic step to take advantage of multicore architectures is to use multithreaded routines.

This is a fine-grain, low-level parallelism. This approach follows a *fork/join model*. Computation outside of the BLAS/LAPACK routines is sequential. Besides, each call to a routine has to pay the cost of spawning new threads and joining them at the end. Therefore, this parallelization model might not be sufficient.

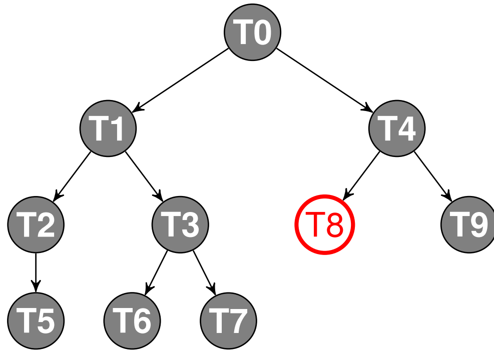
C. Task-parallelism

We have seen in section II-A that the branch-and-bound procedure creates a tree: the branch-and-bound search tree. Each node of this tree can create (or not) subproblems. Each of these subproblems forms a node, that can be computed independently from the other ones. Compared with the approach using multithreaded computation kernels, this is a coarser-grain parallelism.

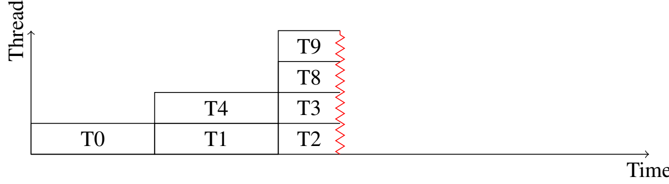
When generated, nodes of the search tree are put in a queue. When an idle thread is available, it pops a node from the queue and evaluates it. Therefore, this approach follows a *task-based parallelism model*. The priority system provided by BOB handles different priorities between the different nodes and, therefore, the different parallel tasks.

When the current best solution is updated (e.g. when an optimal solution is found), nodes with a evaluation which is not as good are removed from the queue by the BOB platform. Moreover, the other threads that are working may also stop their evaluation if their node can be pruned using this new bound (since the bounding procedure provides valid bounds during all the evaluation process : see remark section II-A).

At the beginning of the computation, only one node exists and therefore, only one thread is computing. As new nodes are generated, more threads can compute them in parallel. Therefore, the level of parallelism increases as nodes are generated. This approach is efficient when the problem generates



(a) Binary tree of the branch-and-bound.



(b) Parallel execution on 4 threads.

Fig. 1: Computation of a tree that generates 7 tasks, where the optimum is found on task T8.

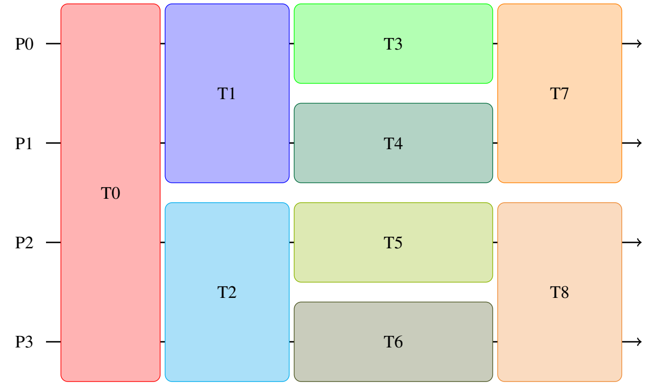


Fig. 2: Possible (perfect) thread occupation of 9 tasks on 4 threads with hierarchical parallelism.

a large number of nodes, in order to amortize the low level of parallelism of the initial phase.

A short example is given in Figure 1. The initial node T0 generates two nodes T1 and T4. The sequential version (represented by the tree in Figure 1a) computes the nodes in the numerical order indicated (from 1 to 9) if we assume that the value of their evaluation implies it : the branch-and-bound does a best-first search, so if T2 has a better evaluation than T4, T2 will be chosen first. The optimum is found on T8 : the sequential version has already evaluated T5, T6 and T7 whereas the parallel version (Figure 1b) has not, and potentially stops the execution of T2, T3 and T9 because the best solution has been updated.

A drawback of this approach is possible load unbalance. If a node takes significantly longer than the other ones to be computed, it can delay the whole computation while the other threads are waiting for it to complete. However, in practice, this case does not happen and several mechanisms guarantee bounded evaluation times and roughly equivalent computation time (see section II-A).

D. Two-level parallelization

In order to improve the exploitation of the multiple core platform when the branch-and-bound tree has not generated enough nodes to keep them all busy, both previous approaches can be combined together in a *hierarchical parallelization*. The core idea is to use multiple threads to evaluate a node when threads are idle, and one thread when there are enough nodes to assign one to each thread.

A possible schedule is given by Figure 2 (note that the tasks are not necessarily related to the ones on Figure 1). At the beginning of the computation, only one node exists in the branch-and-bound tree. Therefore, all the threads are used to evaluate it. It generates two nodes: each of them is evaluated on two threads. These nodes generate four nodes in total, which is equal to the number of threads: each node is evaluated on a single thread. At the end, the tree narrows and only two nodes are generated, evaluated on two nodes each.

Choosing the number of threads to evaluate a node is not trivial. If some threads are idle when a node evaluation begins, later during the evaluation of this node, other nodes might be generated and need these threads to compute them. In our system, coarse-grain parallelism has a higher priority than the fine-grain one on thread occupation. Therefore, idle threads are assigned to new node evaluation rather than on multithreaded node evaluation. Various heuristics can be defined to determine the number of threads to be used to compute a given task.

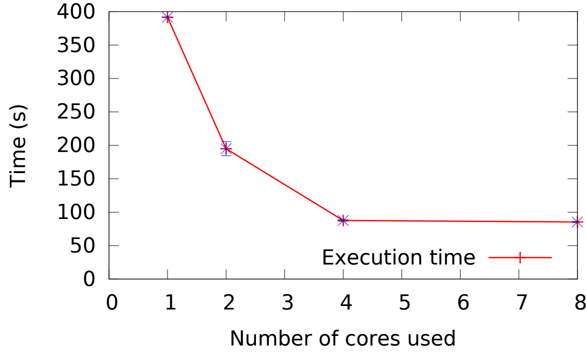
III. Performance evaluation

We evaluated and compared the performance of our implementation of the algorithms described in section II. In particular, we compared their scalability and raw performance. The problem instances are described thoroughly and the numerical results obtained with the current version of *BiqCrunch* are given on the *BiqCrunch* website.

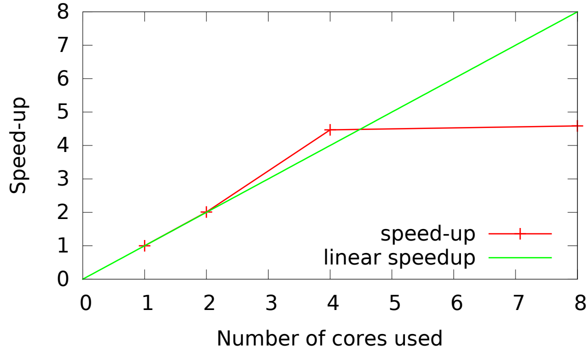
A. Scalability

We limited the number of cores used by the multithreaded *BiqCrunch* and multithreaded BLAS in order to avoid using too many cores. In particular, if our heuristic makes BiqCrunch choose to use a number of cores for the BLAS routines such that, later, new tasks are executed and the total number of threads used exceeds the number assigned to *BiqCrunch*, the system limits *BiqCrunch* in such a way that it does not use more cores than indicated.

We used a 32-core machine that features two Intel Xeon CPU E5-2630 v3 running at 2.4 GHz and 32 GB of RAM. The machine runs a Linux 3.16.0 kernel. All the code was compiled using the GNU gfortran and gcc 4.9.2 compilers with -O3 optimization flag. We compiled the code against OpenBLAS 0.2.12 and LAPACK 3.5.0. *BiqCrunch* provides L-BFGS-B version 3.0, that calls LINPACK and BLAS routines provided



(a) Task scalability.



(b) Speedup.

Fig. 3: Scalability with the brock-200-4 problem.

with the source code. We modified it in order to call routines from the BLAS and LAPACK libraries installed on our system (with a wrapper to call equivalent LAPACK routines instead of the LINPACK ones).

a) Task parallelism: The performance of BiqCrunch increases when threads are added to the computation (see section III-C). We evaluated the scalability of the multi-threaded computation (one thread per node of the search tree) on various problems. For instance, Figure 3 presents the scalability (Figure 3a) and the speedup (Figure 3b) obtained by the computation of `brock-200-4`, a Max-Independent Set problem with $n = 200$ issued from the DIMACS challenge that maximizes the total weight of the vertices in the independent set.

We can see that it scales well, up to a certain number of threads. Unlike small problems, this problem is not limited by the number of nodes in the search tree: it evaluates 185 nodes. Therefore we believe that the scalability is limited by thread management and synchronization costs.

However, this approach faces a strong limitation: in practice, some problems generate only a few nodes, or even only one. If the optimal solution is found on the first node, evaluating nodes in parallel is completely useless, because the one and only node is evaluated by one thread.

b) Multithreaded computation kernels: In order to take advantage of the multiple cores available even when the structure of the search tree does not allow enough parallel tasks (as described in section II-C), we called the BLAS routines using multiple threads. However, on small instances, experimentally, the performance is roughly the same using 1 to 10 threads.

c) Hybrid parallelism: We evaluated the performance of the hybrid approach in two contexts: with a number of tasks (used to evaluate the nodes) equal to the number of cores used (a configuration similar to the one presented by Figure 2), and with a number of nodes smaller than the number of cores and several threads per node. The latter configuration tries to scale beyond the scalability limits of the node parallelism by assigning several threads to evaluate one node: if solving the problem scales up to 16 node evaluations in parallel, we assigned two threads per node in order to use 32 cores in total: it is a nested parallelism approach. The former uses several threads per node when some threads are idle because the search tree has not generated enough nodes to keep them busy: it is close to a greedy approach.

Figure 4 presents the scalability of solving the `bqp-250-6` problem (a pure binary quadratic problem with $n = 250$, available in the OR-library and BiqMac libraries, and used in [18], [19]) using half of the idle threads per node evaluation. We can see that it scales poorly. We have limited to 8 threads, since the nodes' queue list is never longer. We analyzed the execution of *BiqCrunch* and we noticed that, because of the asynchronous nature of the scheduling of the threads that evaluate the nodes, *BiqCrunch* tends to use more threads than the number of cores assigned to the computation (recall that we limited the number of cores available for each run, for fairness purpose).

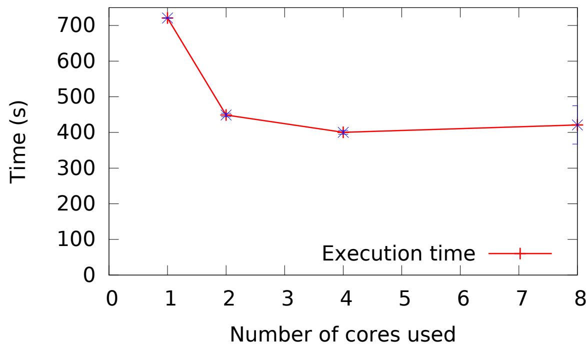
In Figure 6, we are presenting the performance obtained by the `brock-200-4` problem with 2 threads per node evaluation.

We can see that it "extends" the scalability of the parallel implementation, but the overall performance is only a few percent better than with one thread per node evaluation (Figure 3a). It can possibly be explained by the relatively small speedup obtained by using multithreaded node evaluation in general.

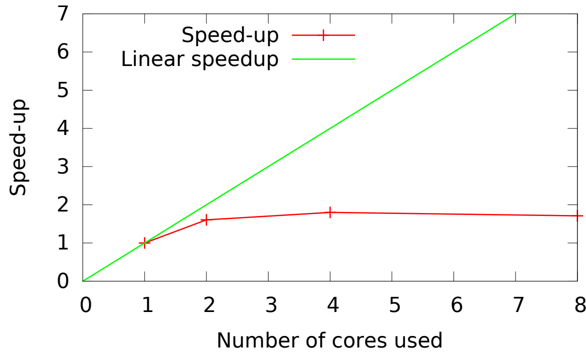
In order to set the balance between the two levels of parallelism, we used performance profiles [20]. Figure 5 gives the performance profiles obtained for a set of 45 Max-k-Cluster problems with $n = 100$ used in several papers (e.g. [21]) and publicly available on the *BiqCrunch* website. The number of threads assigned to BLAS during the node evaluation ranges from 1 (sequential BLAS) up to 8 (in this case all the nodes are evaluated sequentially and BLAS uses all the cores). If one considers a set $\mathcal{S}$ of problems used to benchmark the solvers, then for each problem $p \in \mathcal{S}$, we define $t_p^{\min}$ as the minimum time required to solve p over all the solvers. Then, for each solver, we consider the performance profile function θ, which is defined as

$$\theta(\tau) = \frac{1}{|\mathcal{S}|} \left| \left\{ p \in \mathcal{S} : t_p \leq \tau t_p^{\min} \right\} \right|, \quad \text{for } \tau \geq 1, \qquad (2)$$

where t_p is the time required for the solver to solve problem p.

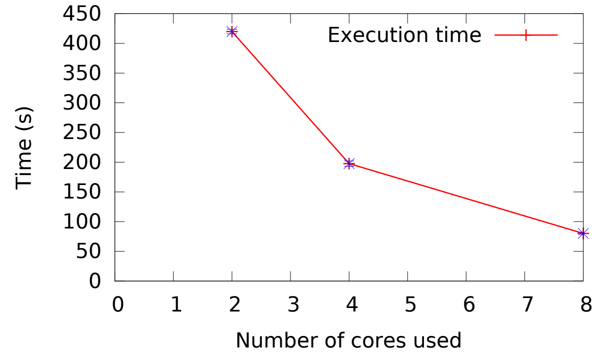


(a) Task scalability.

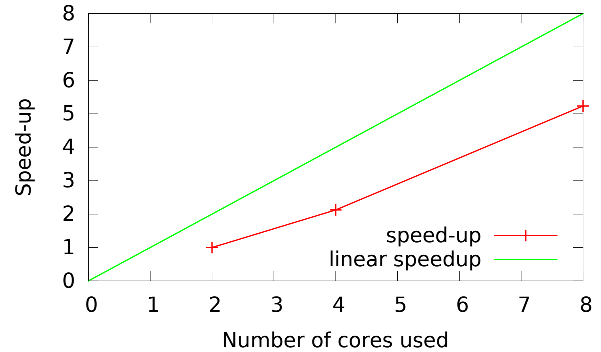


(b) Speedup.

Fig. 4: Scalability with the bqp-250-6 use-case with half of the idle threads per node evaluation.



(a) Task scalability.



(b) Speedup.

Fig. 6: Scalability with the brock-200-4 problem with 2 threads per node evaluation.

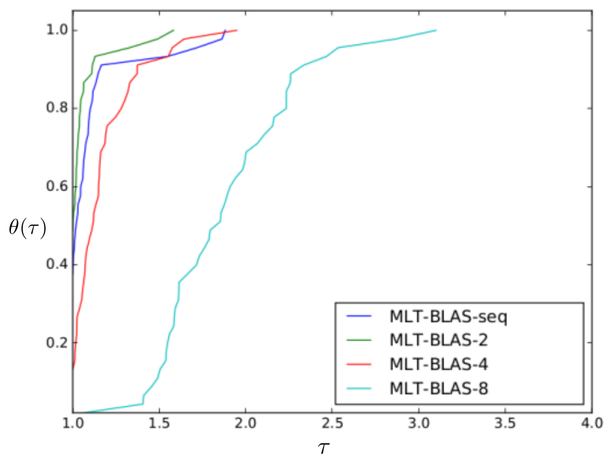


Fig. 5: Performance profiles using different balancings of the hybrid approach. Each curve $\theta(\tau)$ corresponds to a given setting (from 1 up to 8 threads assigned to BLAS).

The function θ is therefore a cumulative distribution function, and $\theta(\tau)$ represents the probability of the solver to solve a problem from $\mathcal{S}$ within a multiple τ of the minimum time required by all solvers considered. These results confirm the one obtained in Figure 6: the best choice is to run the BLAS routines using at most two threads.

B. Numerical issues

For now, the two-level parallelism is still not fully satisfactory. We have noticed that using the current parameters (e.g. tolerance) of the linear algebra functions, the multithreaded version of BLAS tends to be numerically unstable when the underlying nonlinear relaxation is very tight (see [11] for further details about how adjusting the tightness of the relaxation). We have improved this stability by setting new values, but a lot of factors come into play here.

First, there is a "giving up" function in the bounding procedure that stops the evaluation of a node when the progress of L-BFGS is too small compared to the value of the best current feasible solution. Consequently, this can occur at a different moment of the computation if a different number of threads are allocated to the BLAS functions.

Second, the branching procedure actually depends on the fractional solution to select the variable to branch on, and these values can be slightly different when using the multithreaded version of BLAS. Nevertheless, for most problems, it must be pointed out that this second parallelization level does not improve a lot the solver performance. Indeed, the proportion of computation time during which the number of nodes in the queue is smaller than the number of threads is often negligible (except for "easy" problems). Consequently, for

difficult problems (i.e. that require a large number of node evaluations), each thread will be kept busy most of the time. Hence, it is possible to avoid these issues by using only a one-level parallelization (i.e. one thread corresponds to one node evaluation). But of course, we must investigate in depth the reasons behind this numerical instability to address the problem: this is an ongoing work.

C. Performance comparison and computational results

In this section, we present computational results obtained for three classical NP-hard combinatorial problems that can be stated as 0-1 quadratic programs. All the tests are run using the same computer: a DELL T-1600 equipped with an Intel Xeon E3-1270 CPU running at 3.40GHz with 8 cores. The same parameters (see the *BiqCrunch* documentation) are set for both solvers except for the number of cores: BiqCrunch2.0 uses a single core and *MLTBiqCrunch* uses four cores (except in Figure 7 where the number of cores ranges from two to eight).

We chose instances that are not solved at the root of the search tree by *BiqCrunch*, and thus are relevant in our context. All the problems are publicly available and have been used by several authors [18], [21] (see the *BiqCrunch* website `http://lipn.univ-paris13.fr/BiqCrunch/` for further details and references).

In the Max-Independent-Set (MIS) problem (see Table I), we are given a graph $G = (V, E)$ with vertex weights w_i, and the objective is to maximize the total weight of the vertices in an independent set (a set S of vertices having no two vertices joined by an edge in E):

$$(\text{MIS}) \quad \begin{array}{ll} \text{maximize} & \sum_i w_i z_i \\ \text{subject to} & z_i z_j = 0, \qquad \forall (i,j) \in E \\ & z \in \{0,1\}^n. \end{array} \tag{3}$$

In the Max-k-Cluster problem (see Tables II, III), we are given an edge-weighted graph with n vertices and a natural number k, and the objective is to find a subgraph of k nodes having maximum total edge weight:

$$(\text{Max-}k\text{-Cluster}) \quad \begin{array}{ll} \text{maximize} & \frac{1}{2}\sum_{ij} w_{ij} z_i z_j \\ \text{subject to} & \sum_{i=1}^n z_i = k \\ & z \in \{0,1\}^n. \end{array} \tag{4}$$

In the Max-Cut problem (see Tables IV, IV, VI, VII,VIII, IX), we are given an edge-weighted graph with n vertices, and the objective is to maximize the total weight of the edges between a subset of vertices and its complement:

$$(\text{Max-Cut}) \quad \begin{array}{ll} \text{maximize} & \sum_{ij} w_{ij} z_i (1 - z_j) \\ \text{subject to} & z \in \{0,1\}^n. \end{array} \tag{5}$$

MLTBiqCrunch is always faster and in some cases it generates fewer nodes. In some other cases (for example `brock200_1`) the optimal solution is found late in the traversal of the search tree; that explains the much larger number of nodes for *MLTBiqCrunch*. Let us to point out that solving this problem requires only 47 MB with *MLTBiqCrunch*. It involves 200 binary variables (20 100 for the underlying relaxations) and 5267 equality constraints.

When using *MLTBiqCrunch*, we have observed a super-linear speedup for several problems, especially for the most difficult instances (see Table III). Actually, as pointed out in Section II-C, the current best feasible solution can be updated earlier (maybe several times) and therefore, fewer nodes are generated in the search tree. Moreover, since the bounding procedure can be interrupted at any time, a superlinear speedup can even occur with the same number of nodes in the search tree when several bounding procedures are stopped earlier at the same time.

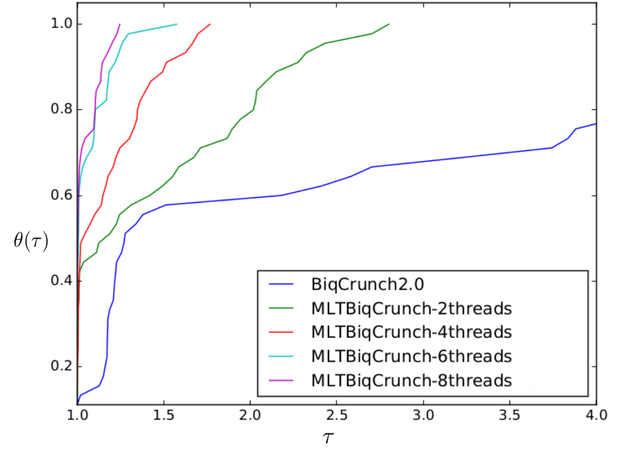


Fig. 7: Performance profiles of *BiqCrunch*2.0 and *MLTBiqCrunch* (each curve $\theta(\tau)$ corresponds to a given number of threads).

In Figure 7, we illustrate the expected performance from a standard user point of view when using *MLTBiqCrunch* instead of *BiqCrunch* (the current version is BiqCrunch2.0). This figure gives the performance profiles [20] obtained for the set of problems used in Figure 5. Obviously, increasing the number of threads improves the performance profiles of the solver. Recall that now BLAS uses at most two threads, and thus all the additional free cores are assigned to evaluate the available nodes in the queue.

TABLE I: CPU times and number of nodes in the search tree to solve Max-Independent-Set problems (DIMACS library)

			BiqCrunch 2.0		MLTBiqCrunch	
	n	m	nodes	time (s)	nodes	time (s)
MANN_a9	45	72	5	3.90	3	0.64
keller4	171	5100	155	155.24	113	88.55
brock200_1	200	5066	1393	1822.81	2861	747.38
brock200_2	200	10024	53	87.01	79	73.78
brock200_3	200	7852	107	157.45	321	113.04
brock200_4	200	6811	185	263.32	185	77.51

IV. Conclusion

In this paper, we have analyzed and compared the performance gain of two parallelization strategies for the *BiqCrunch*

TABLE II: CPU times and number of nodes (in the search tree) averaged over five instances for each triple (n,k,d) (d is the graph density) required to solve medium-sized Max-k-Cluster problems

n	k	$d(\%)$	BiqCrunch 2.0 nodes	BiqCrunch 2.0 time (s)	MLTBiqCrunch nodes	MLTBiqCrunch time (s)
120	30	25	64.6	133.04	54.6	29.56
		50	110.6	177.20	109.0	43.73
		75	236.6	297.84	222.4	70.05
	60	25	28.6	59.09	27.4	21.36
		50	43.8	90.66	43.0	28.41
		75	19.0	38.95	19.0	16.76
	90	25	1.0	3.53	1.0	3.54
		50	6.2	28.79	7.4	20.05
		75	1.0	2.38	1.0	2.24

TABLE III: CPU times and number of nodes (in the search tree) averaged over five instances for each triple (n,k,d) (d is the graph density) required to solve to solve large Max-k-Cluster problems

n	k	$d(\%)$	BiqCrunch 2.0 nodes	BiqCrunch 2.0 time (s)	MLTBiqCrunch nodes	MLTBiqCrunch time (s)
160	40	25	501.0	1927.53	535.4	397.59
		50	6061.6	**15411.70**	6430.6	**3731.51**
		75	4427.8	10798.50	5103.8	2624.43
	80	25	207.4	854.28	195.4	177.25
		50	505.8	1791.06	536.6	471.94
		75	2017.4	7242.98	2101.4	1786.57
	120	25	10.2	74.53	10.6	38.75
		50	7.0	63.67	6.2	30.95
		75	3.8	30.64	5.0	28.39

TABLE IV: CPU times and number of nodes in the search tree to solve the w100.d050 max-cut problems.

problem	BiqCrunch 2.0 nodes	BiqCrunch 2.0 time (s)	MLTBiqCrunch nodes	MLTBiqCrunch time (s)
0	307	434.02	345	121.88
1	111	188.84	109	53.47
2	57	93.21	59	32.07
3	297	401.06	319	115.11
4	471	646.58	451	168.42
5	349	529.17	363	147.94
6	99	135.20	99	49.42
7	33	62.39	31	21.14
8	403	557.02	401	149.92
9	33	66.20	31	25.49

branch-and-bound solver. We have seen that a coarse-grain, task-based approach gives a satisfying speed-up, but is limited by the start-up phase of the computation, when the search tree is not wide enough to take advantage of all the available cores. On the other hand, we have seen that a fine-grain, kernel-level parallelization is too fine-grained to give a good speed-up, even in these phases.

Although the evaluation of each node is hardly data-parallel, parallelizing the evaluation of each node is an interesting approach that deserves some consideration. The bigger granularity of this approach might give better results that the one

TABLE V: CPU times and number of nodes in the search tree to solve the w100.d090 max-cut problems.

problem	BiqCrunch 2.0 nodes	BiqCrunch 2.0 time (s)	MLTBiqCrunch nodes	MLTBiqCrunch time (s)
0	229	360.93	213	97.04
1	1555	2288.50	1559	646.63
2	551	809.24	529	215.36
3	779	1080.00	879	312.68
4	321	491.79	297	136.62
5	7	16.96	7	14.33
6	55	118.44	63	43.40
7	185	283.12	171	77.38
8	93	192.86	99	57.93
9	259	368.84	297	111.79

TABLE VI: CPU times and number of nodes in the search tree to solve the pw100.d050 max-cut problems.

problem	BiqCrunch 2.0 nodes	BiqCrunch 2.0 time (s)	MLTBiqCrunch nodes	MLTBiqCrunch time (s)
0	945	1099.56	1121	415.40
1	317	386.65	293	112.24
2	365	452.35	399	148.70
3	91	116.66	93	43.31
4	467	631.61	373	162.82
5	123	172.50	115	52.99
6	745	1054.98	663	283.41
7	149	227.40	139	71.93
8	43	86.13	43	31.13
9	203	278.38	241	100.75

TABLE VII: CPU times and number of nodes in the search tree to solve the pw100.d090 max-cut problems.

problem	BiqCrunch 2.0 nodes	BiqCrunch 2.0 time (s)	MLTBiqCrunch nodes	MLTBiqCrunch time (s)
0	291	407.79	303	128.56
1	523	674.41	479	178.22
2	135	197.70	153	62.19
3	111	158.40	119	52.87
4	235	316.92	227	93.91
5	307	502.81	319	144.38
6	221	264.78	245	84.40
7	503	687.72	529	199.04
8	181	316.72	175	88.35
9	137	227.82	141	65.92

TABLE VIII: CPU times and number of nodes in the search tree to solve the pm1d100.d090 max-cut problems.

problem	BiqCrunch 2.0 nodes	BiqCrunch 2.0 time (s)	MLTBiqCrunch nodes	MLTBiqCrunch time (s)
0	635	796.95	739	235.88
1	1187	1464.82	1159	372.35
2	885	1070.42	823	262.49
3	189	266.69	249	100.45
4	567	720.50	573	194.33
5	155	209.20	159	61.25
6	139	203.60	127	51.13
7	57	104.56	57	35.06
8	47	64.67	37	20.05
9	243	309.30	239	87.01

TABLE IX: CPU times and number of nodes in the search tree to solve the g05.n100 max-cut problems.

	BiqCrunch 2.0		MLTBiqCrunch	
problem	nodes	time (s)	nodes	time (s)
0	379	417.26	387	129.32
1	1683	1886.18	1889	648.07
2	103	138.52	91	38.30
3	589	554.84	705	172.63
4	33	44.26	35	16.71
5	107	167.96	105	45.98
6	107	151.56	109	43.59
7	255	331.53	257	92.36
8	163	198.11	175	60.99
9	219	222.28	219	61.44

based on multithreaded computation routines, would the data dependencies allow it.

Overall, the coarse-grain, node-level parallelization presents good results, with a satisfying speed-up on large problems that generate a non-trivial number of nodes. Large instances can be solved in less than an hour, which is very positive: these instances can be solved in reasonable time on a desktop workstation. Smaller instances can already be solved in reasonable time, so they are not the core target of *MLTBiqCrunch*, which aims at making it possible to solve 0-1 quadratic problems on mainstream desktop computers. In that sense, the multithreaded version we are presenting here fulfills this goal.

Quite surprisingly, we have noticed that the small loss of precision suffered by parallel computation routines, due to the reorganization of the computation in the kernels, can affect the branch-and-bound computation dramatically, causing a slower convergence or, more annoyingly, creating extra nodes in the search tree. The numerical stability and accuracy of the parallel computation routines is therefore of major importance. Another perspective for future works consists in exploring the gain provided by extended-precision or arbitrary-precision routines, such as MPACK [22] or xBLAS [23].

References

[1] N. Krislock, J. Malick, and F. Roupin, "Biqcrunch: A semidefinite branch-and-bound method for solving binary quadratic problems," *ACM Trans. Math. Softw.*, vol. 43, no. 4, pp. 32:1–32:23, Jan. 2017. doi: 10.1145/3005345. [Online]. Available: http://doi.acm.org/10.1145/3005345

[2] S. Burer and A. Letchford, "Non-convex mixed-integer nonlinear programming: A survey," *Surveys in Operations Research and Management Science*, vol. 17, no. 2, pp. 97 – 106, 2012. doi: 10.1016/j.sorms.2012.08.001. [Online]. Available: https://doi.org/10.1016/j.sorms.2012.08.001

[3] M. Bussieck, S. Vigerske, J. Cochran, L. Cox, P. Keskinocak, J. Kharoufeh, and J. Smith, *MINLP Solver Software*. John Wiley, Inc., 2010, updated Feb 21, 2012. [Online]. Available: https://doi.org/10.1002/9780470400531.eorms0527

[4] CPLEX, *IBM ILOG CPLEX V12.1 User's Manual for CPLEX*, IBM Corporation, 2009.

[5] B. Borchers and J. G. Young, "implementation of a primal–dual method for sdp on a shared memory parallel architecture," *Computational Optimization and Applications*, vol. 3, no. 37, pp. 355–369, 2007. doi: 10.1007/s10589-007-9030-3. [Online]. Available: https://doi.org/10.1007/s10589-007-9030-3

[6] L. Barreto and M. Bauer, "Parallel branch and bound algorithm - a comparison between serial, openmp and mpi implementations," *Journal of Physics: Conference Series*, vol. 256, no. 1, p. 012018, 2010. [Online]. Available: http://stacks.iop.org/1742-6596/256/i=1/a=012018

[7] A. Bendjoudi, N. Melab, and E.-G. Talbi, "Hierarchical branch and bound algorithm for computational grids," *Future Generation Computer Systems*, vol. 28, no. 8, pp. 1168–1176, 2012. doi: 10.1016/j.future.2012.03.001. [Online]. Available: https://doi.org/10.1016/j.future.2012.03.001

[8] J. Gmys, M. Mezmaz, N. Melab, and D. Tuyttens, "Ivm-based parallel branch-and-bound using hierarchical work stealing on multi-gpu systems," *Concurrency and Computation: Practice and Experience*, 2016. doi: 10.1002/cpe.4019. [Online]. Available: https://doi.org/10.1002/cpe.4019

[9] D. A. Bader, W. E. Hart, and C. A. Phillips, *Tutorials on Emerging Methodologies and Applications in Operations Research. Chapter 5 : Parallel Algorithm Design for Branch and Bound*, h.j. greenberg, editor ed. Philadelphia, PA: Society for Industrial and Applied Mathematics, Kluwer Academic Press, 2004.

[10] Y. Xu, T. Ralphs, L. Ladanyi, and M. Saltzman, "Coin-or high-performance parallel search framework," projects.coin-or.org/CHiPPS.

[11] J. Malick and F. Roupin, "On the bridge between combinatorial optimization and nonlinear optimization: a family of semidefinite bounds for 0–1 quadratic problems leading to quasi-newton methods," *Mathematical Programming*, vol. 140, no. 1, pp. 99–124, 2013. doi: 10.1007/s10107-012-0628-6. [Online]. Available: http://dx.doi.org/10.1007/s10107-012-0628-6

[12] N. Melab, J. Gmys, M. Mezmaz, and D. Tuyttens, "Multi-core versus many-core computing for many-task branch-and-bound applied to big optimization problems," *Future Generation Computer Systems*, 2017. doi: 10.1016/j.future.2016.12.039. [Online]. Available: https://doi.org/10.1016/j.future.2016.12.039

[13] I. Chakroun and N. Melab, "Towards a heterogeneous and adaptive parallel branch-and-bound algorithm," *Journal of Computer and System Sciences*, vol. 81, no. 1, pp. 72–84, 2015. doi: 10.1016/j.jcss.2014.06.012. [Online]. Available: https://doi.org/10.1016/j.jcss.2014.06.012

[14] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, 3rd ed. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1999. [Online]. Available: https://doi.org/10.1137/1.9780898719604.pt2

[15] C. Zhu, R. Byrd, P. Lu, and J. Nocedal, "Algorithm 778: L-bfgs-b: Fortran subroutines for large-scale bound-constrained optimization," *ACM Trans. Math. Softw.*, vol. 23, no. 4, pp. 550–560, Dec. 1997. [Online]. Available: https://doi.org/10.1137/0916069

[16] J. Morales and J. Nocedal, "Remark on "Algorithm 778: L-BFGS-B: Fortran subroutines for large-scale bound constrained optimization"," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1–4, 2011. [Online]. Available: https://doi.org/10.1145/2049662.2049669

[17] B. Le Cun, C. Roucairol, and T. P. Team, "Bob: a unified platform for implementing branch-and-bound like algorithms," Laboratoire Prism, Tech. Rep., 1995.

[18] F. Rendl, G. Rinaldi, and A. Wiegele, *A Branch and Bound Algorithm for Max-Cut Based on Combining Semidefinite and Polyhedral Relaxations*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 295–309. ISBN 978-3-540-72792-7. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-72792-7_23

[19] N. Krislock, J. Malick, and F. Roupin, "Improved semidefinite bounding procedure for solving max-cut problems to optimality," *Mathematical Programming*, vol. 143, no. 1, pp. 61–86, 2014. doi: 10.1007/s10107-012-0594-z. [Online]. Available: https://doi.org/10.1007/s10107-012-0594-z

[20] E. D. Dolan and J. J. Moré, "Benchmarking optimization software with performance profiles," *Mathematical Programming*, vol. 91, pp. 201–213, 2002. [Online]. Available: https://doi.org/10.1007/s101070100263

[21] N. Krislock, J. Malick, and F. Roupin, "Computational results of a semidefinite branch-and-bound algorithm for k-cluster," *Computers and Operations Research*, vol. 66, pp. 153–159, 2016.

[22] M. Nakata, "The MPACK (MBLAS/MLAPACK); a multiple precision arithmetic version of blas and lapack," mplapack.sourceforge.net/.

[23] X. Li, J. Demmel, D. Bailey, Y. Hida, J. Iskandar, A. Kapur, M. Martin, B. Thompson, T. Tung, and D. Yoo, "XBLAS–extra precise basic linear algebra subroutines," www.netlib.org/xblas.