Mitigating Anomalies in Parallel Branch-and-Bound Based Algorithms for Mixed-Integer Nonlinear Optimization

Prashant Palkar[1(✉)] and Ashutosh Mahajan[2]

[1] Institute of Mathematics, University of Augsburg, 86159 Augsburg, Germany
prashant.palkar@math.uni-augsburg.de
[2] Industrial Engineering and Operations Research, IIT Bombay, Powai, Mumbai 400076, India
amahajan@iitb.ac.in
http://www.ieor.iitb.ac.in/ppalkar

Abstract. We address detrimental anomalies in parallel versions of two state-of-the-art algorithms for convex mixed-integer nonlinear programs (MINLPs): nonlinear branch-and-bound (NLP-BB) and the LP/NLP based branch-and-bound (QG). A detrimental anomaly is when a parallel algorithm performs worse than its sequential counterpart. Unambiguous node selection functions have been developed in the past to avoid these anomalies. We extend this notion of unambiguity to subroutines for branching and generating linearization cuts. We implement the proposed unambiguous branching and cut generation strategies alongside unambiguous node selection in NLP-BB and QG in the open-source MINLP solver Minotaur. Our computational experiments on convex instances from the MINLPLib library show that detrimental anomalies can be reduced to a great extent in practical algorithms. We also compare these algorithms with opportunistic parallel versions. Our results highlight that opportunistic versions perform better in terms of wall clock times, while the deterministic versions avoid detrimental anomalies with theoretically established guarantees and also provide reproducible results, a feature that is desirable while developing parallel algorithms. We recommend settings in Minotaur that yield opportunistic or deterministic runs for the parallel NLP-BB and QG algorithms.

Keywords: Mixed-integer nonlinear programming · Parallel branch-and-bound · Anomalies · LP/NLP based branch-and-bound

1 Introduction

Mixed-Integer Nonlinear Programs (MINLPs) are discrete optimization problems that involve integer-constrained decision variables and nonlinear functions. An MINLP can be mathematically expressed as

$$\begin{aligned} &\underset{x}{\text{minimize}}\, c^T x \\ &\text{subject to}\;\; g(x) \leq\; b, \\ &\qquad\qquad x \in \mathcal{X}, x_j \in \mathbb{Z} \quad \forall j \in \mathcal{I}, \end{aligned} \tag{P}$$

I. Ljubić et al. (Eds.): ISCO 2022, LNCS 13526, pp. 143–156, 2022.
https://doi.org/10.1007/978-3-031-18530-4_11

where the set $\mathcal{I}$ contains the indices of integer-constrained variables, the constraint functions $g : \mathbb{R}^n \to \mathbb{R}^m$ are assumed to be nonlinear and twice continuously differentiable, and the set $\mathcal{X}$ is non-empty and compact. MINLPs arise in many important real-life applications [2,4,16], however, they are difficult to solve as their special cases such as MILPs belong to the class of NP-hard problems [8].

State-of-the-art algorithms for MINLPs are based on branch-and-bound (BB) framework. It partitions the search space recursively into smaller and usually disjoint regions until a solution is found, or no further partitioning is possible. BB starts by solving a relaxation that is easier to solve and whose solution provides a valid lower bound on the optimal value (say z^*) of (P). Then the search-space is branched to create smaller subproblems. If a solution to any of the subproblems is feasible for (P), its objective value provides an upper bound on z^*. This partitioning continues until the lower bound and the upper bound on z^* coincide. This setup can be analyzed as a tree-search where the tree-nodes denote the subproblems and the edges denote the branches that divide a subproblem.

The BB framework is naturally suitable for parallel processing due to the presence of independent subproblems. The performance of a parallel BB algorithm depends on the way it has been implemented on a software platform. Different MILP and MINLP solvers use distinctive data structures, classes, subsolvers, etc., which make each implementation/solver unique in its own way. Several studies have addressed the practical aspects of parallel BB algorithms [3,6,7,14,18]. Theoretical analysis of parallel BB algorithms has also been performed earlier [9–11,13]. Some of these studies focus on the unpredictable performance of parallel branch-and-bound algorithms with respect to the number of processors used. These phenomena are referred to as "anomalies". We focus on reducing "detrimental anomalies" that may arise due to two subroutines in BB for convex MINLPs: selecting a branching candidate, and adding linearization inequalities at a node. While the issue of node-selection has been studied in the past [10,11], these two aspects have not been addressed to the best of our knowledge. We concentrate on two well-known BB based algorithms for convex MINLPs: NLP-BB and LP/NLP based BB, also called QG (an acronym for Quesada and Grossmann [15] who proposed this algorithm).

The outline of the paper is as follows. Section 3 presents the opportunistic parallel extensions of NLP-BB and QG in Minotaur[1] [12]. In Sect. 4, we define unambiguous branching functions and show how a parallel NLP-BB algorithm without detrimental anomalies can be implemented using unambiguous algorithmic components (referred to as "nondetrimental" NLP-BB). Section 5 presents unambiguous functions for generating linearization cuts and a nondetrimental QG algorithm. Section 6 shows the computational performance of the opportunistic and the nondetrimental algorithms and Sect. 7 presents the conclusions.

[1] Available at http://github.com/minotaur-solver/minotaur.

2 Anomalies in Parallel Algorithms

Existence of anomalies in parallel tree-search based algorithms is shown in [9] and sufficient conditions to avoid detrimental anomalies caused by ambiguous node selection are proposed in [10]. These results are based on the concept of an "iteration" in a parallel BB algorithm, which we now define.

Definition 1 (Iteration). *An iteration is one cycle of all operations such as node-presolving, node-processing, branching, adding cuts, checking stopping conditions, inserting new nodes in the memory, etc., that a set of processors perform simultaneously.*

Instead of the wall clock time, the number of iterations taken by a parallel algorithm is suited for our analysis because it does not vary based on the hardware, software implementation and factors like the computational load on a system at a point in time. Let k denote the number of processors, and $T(k, 0)$ be the number of iterations taken, where 0 indicates that we seek an exact optimal solution (see [10] for an analysis when solutions with a predefined tolerance are acceptable). We make the following assumptions.

Assumption 1. *The processors operate "synchronously" i.e., at most k open nodes are selected and solved simultaneously in an iteration.*

Assumption 1 restricts a processor from starting a new cycle of operations until all the other processors have also finished their part in the iteration. Due to this synchronization, the processors that finish earlier incur a waiting time. This waiting for synchronization makes the procedure reproducible, but possibly at a cost of longer running time.

Assumption 2. *All the subsolvers used within the algorithm are deterministic.*

The term "deterministic" in Assumption 2 means that the same solution would be obtained using the subsolvers if the same initial conditions are provided to them. Typically, an LP or an NLP subsolver is used in most MINLP algorithms. Assumption 2 holds for certain LP and NLP solvers subject to the use of sufficiently small tolerance values.

Definition 2 [10]**.** *A behavior exhibited by a parallel tree-search algorithm using k processors is a detrimental anomaly if $T(k, 0) > T(1, 0)$.*

A depiction of a detrimental anomaly is shown in Figs. 1–2 where a parallel algorithm ($k = 2$) performs worse than the sequential algorithm (depicted as $k = 1$). Table 1 lists the nodes that are processed in each iteration by each processor for both the cases. The number of iterations required when two processors are used (5 iterations) is more than that for the sequential algorithm (3 iterations). As mentioned in [10], one of the main reasons for detrimental anomalies is the ambiguous selection of nodes in the sequential and the parallel versions. One can observe in Table 1 that while the sequential algorithm processes node labeled 4

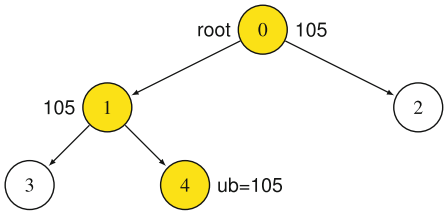

Fig. 1. A sequential branch-and-bound tree ($k = 1$). The algorithm processes nodes $0, 1, 4$ respectively and then terminates.

Table 1. Nodes solved in different iterations of the sequential ($k = 1$) and the parallel algorithm ($k = 2$).

	Node processed		
	$k = 1$	$k = 2$	
Iter	`thread0`	`thread0`	`thread1`
1	0	0	–
2	1	1	2
3	4	9	10
4	–	11	12
5	–	13	14

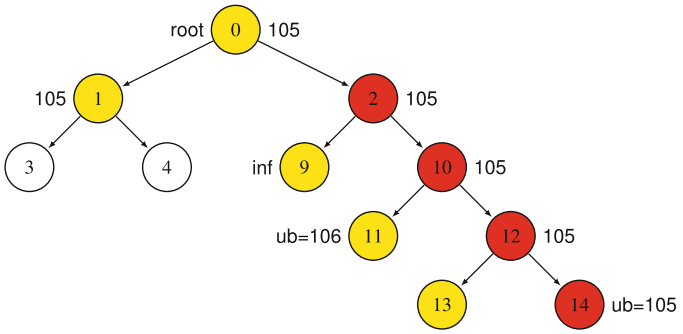

Fig. 2. A BB tree explored using two threads ($k = 2$). Two nodes are solved in parallel in each iteration except in the first iteration. `thread0` solves the yellow colored nodes and `thread1` solves the red ones. The algorithm terminates after node 14 is processed.

in Iteration 3, the parallel algorithm selects nodes 9 and 10 (and not 4) and ends up processing six other nodes (in three more iterations).

Sufficient conditions on node selection functions to avoid detrimental anomalies are as follows.

Definition 3 [10]. *Given a set of open nodes $\mathcal{P}$, a heuristic node selection function $h(\cdot)$ is referred to as unambiguous if it satisfies the following two properties.*

1. $h(P_i) \neq h(P_j)$, *for any* $P_i, P_j \in \mathcal{P}$, $P_i \neq P_j$
2. $h(P_i) \leq h(P_j)$, *if* P_j *is a descendant of* P_i.

All nodes encountered while moving down the tree starting from a node are referred to as the "descendants" of that node. Similarly, the nodes encountered while moving up the tree starting from a node are called its "ancestor" nodes. The function h maps nodes in $\mathcal{P}$ to real values. A node with a lower heuristic function value has a higher priority for getting selected.

Theorem 1 [10]. *If h is unambiguous, then $T(k,0) \leq T(1,0)$.*

Although Figs. 1–2 demonstrate how ambiguity in node selection cause detrimental anomalies, ambiguity in other components of BB can also give rise to detrimental anomalies. In fact, the tree might evolve differently in presence of an ambiguous algorithmic component such as branching, cut generation or primal heuristics when using different number of processors. State-of-the-art branching rules decide upon a branching disjunction using the scores calculated based on the information obtained from the nodes already processed in the BB tree. Considering the BB trees shown in Figs. 1–2, Table 2 demonstrates how an ambiguous branching decision occurs if all available information is used for branching at the node labeled 4. The column "br. info." mentions the set of nodes processed. It can be seen in Iteration 3 that the sequential ($k = 1$) and parallel ($k = 2$) algorithms have access to different information sets and are likely to select a different branching disjunction at the node labeled 4, which will result in different child nodes, hence different BB trees.

Table 2. Ambiguous branching resulting in generation of dissimilar nodes

	$k = 1$		$k = 2$			
	`thread0`		`thread0`		`thread1`	
Iteration	node	br. info.	node	br. info.	node	br. info.
1	0	–	0	–	–	–
2	1	{0}	1	{0}	2	{0}
3	4	{0,1}	4	{0,1,2}	3	{0,1,2}

In [13, Assump. (A1)], the authors assume that the branching scheme at a node P_i depends only on the information obtained along the path from P_i to the root node. However, their analysis does not include unambiguous branching functions. In this work, we explicitly define functions for unambiguous branching and unambiguous cut generation. We compare the performance of parallel nondetrimental algorithms obtained using these functions with parallel "opportunistic" algorithms in Minotaur.

3 Opportunistic Parallel Branch-and-Bound in Minotaur

The parallel implementation of NLP-BB in Minotaur uses classes available in its MINLP framework. A single pool of open nodes ($\mathcal{P}$) is maintained by the class `TreeManager` to be processed simultaneously by k threads (one corresponding to each of the k processors) until $\mathcal{P}$ is empty. In the beginning, the first thread solves the root relaxation and creates two child subproblems (if not pruned). An idle thread then requests a node (if any) from `TreeManager` for processing. Each thread sends the child nodes generated after branching to the `TreeManager` that are added to $\mathcal{P}$. Node-level parallel extensions of NLP-BB and QG have been

implemented in Minotaur [17, Sec. 4.1 and Sec. 4.3] that use OpenMP `for` loops that induce synchronization at the end of the loop. This section mentions new node-level parallel extensions of the NLP-BB and the QG algorithms that process multiple tree-nodes simultaneously in a more "opportunistic" way compared to the shared-memory parallel algorithms presented in [17, Sec. 4.1 and Sec. 4.3]. Here, we use the term opportunistic in two respects. First, the threads attempt to get a new open node from $\mathcal{P}$ as soon as they finish a cycle without waiting for the other threads. Second, this algorithm is "not deterministic" in terms of reproducibility of results.

Parallel NLP-BB. We refer to the opportunistic parallel extension of NLP-BB in Minotaur as *mcbnbOpp* (here, the prefix *mc* indicates multi-core). This algorithm completely avoids the synchronization of threads after each iteration unlike the algorithm in [17, Sec. 4.1] that has an implicit synchronization at the end of OpenMP `for` loop. This means that if a thread finishes processing a node earlier than the other threads, it will not wait.

Parallel QG. The parallelization scheme of opportunistic parallel extension of QG (*mcqgOpp*) is similar to that of *mcbnbOpp*. The two major dissimilarities from *mcbnbOpp* are that LPs (instead of NLPs) are solved at nodes, and linearization cuts are generated at nodes that yield integer solutions by solving an NLP in which integer variables are fixed.

4 Reducing Detrimental Anomalies in Parallel NLP-BB

If none of the algorithmic components in NLP-BB induce ambiguity, the overall algorithm would avoid detrimental anomalies. First, we focus on branching and define unambiguous branching functions.

4.1 Unambiguous Branching Functions

Typically, a node is constructed by adding a set of branching constraints to its parent. We define unambiguity (as in Definition 3) for branching functions for creating simple variable disjunctions. This definition can be easily extended to other branching functions. Variable disjunctions usually select a branching variable from a set $\mathcal{I}_C := \{j \in \mathcal{I} : x^*_j \notin \mathbb{Z}\}$ of candidates at a node with an optimal solution x^*.

Definition 4. *Consider a node $P_i \in \mathcal{P}$ that has been processed and that is not pruned. Let x^* be the optimal solution of P_i and $\mathcal{I}_C := \{j \in \mathcal{I} : x^*_j \notin \mathbb{Z}\}$. A branching variable selection function $\nu(\cdot)$ over $\mathcal{I}_C$ at P_i is referred to as unambiguous if*

- *$\nu(\cdot)$ uses information obtained only from P_i and its ancestors,*
- *$\nu(j) \neq \nu(k)$ for $j, k \in \mathcal{I}_C, j \neq k$.*

Without loss of generality, the candidate with the highest function value can be used for branching. We refer to a branching scheme as unambiguous if it uses an unambiguous branching function.

The assumption that branching functions must be unambiguous is implicit in the examples and results in [9,10]. However, for sophisticated branching schemes used in state-of-the-art solvers, unambiguity cannot be assumed. Hence, we formally prove that if an unambiguous branching function is used to branch at a node, then "equivalent" child nodes would be created in both the sequential and the parallel BB tree. We say that two nodes are equivalent if they represent the same subproblem. We state this result for simple variable branching. Let ϕ_1 and ϕ_k denote the BB tree explored by the sequential algorithm and the parallel algorithm (using k processors), respectively. Consider a node P_s where $s \in \mathbb{N}$ represents a label (a unique identifier of a node in a BB tree). For the two child nodes of P_s, we assign a label $2s$ to the left child node (generated using the $\leq$ disjunction) and $2s+1$ to the other child node. We refer to this labeling mechanism as the $2s\_2s+1$ rule. The root node is assigned a label 1.

Proposition 1. *Let the nodes in ϕ_1 and ϕ_k be generated using an unambiguous branching function and labeled using the $2s\_2s+1$ rule. Let $P_s^1 \in \phi_1$ and $P_s^k \in \phi_k$ be two nodes with the same label s. Then P_s^1 and P_s^k are equivalent.*

Proof. The equivalence of P_s^1 and P_s^k can be shown using equivalence of their corresponding ancestor nodes upto the root node. The $2s\_2s+1$ rule implies that the labels of the ancestor nodes on the unique paths from the root node (P_1) to P_s^1 and P_s^k, respectively, are equal. Since, P_1 is common to both ϕ_1 and ϕ_k, the branching disjunctions at the root node are equivalent by Assumption 2 and Definition 4. Hence, the ancestor nodes P_2^1 and P_2^k or P_3^1 and P_3^k are equivalent. By Definition 4, $\nu(\cdot)$ generates the branching disjunctions at these nodes using scores obtained from only the current node and its ancestors, which are equivalent. Similarly, the equivalence of P_s^1 and P_s^k is shown. □

Since k is arbitrary in Proposition 1, nodes with the same label are equivalent irrespective of the number of threads used in the algorithm when an unambiguous branching rule is used. It can be easily verified that some well-known branching strategies are naturally based on unambiguous branching functions. For example, the lexicographic branching scheme (*lex*) that selects the variable with the smallest subscript in $\mathcal{I}_C$ satisfies Definition 4. Also, strong branching (*str*), which involves partly or fully solving an LP/NLP subproblem satisfies Definition 4 if a deterministic LP/NLP subsolver is used and if the objective values obtained are all distinct. However, while *lex* is known to result in large BB trees, *str* is considered expensive due to the requirement of solving many LPs/NLPs. Thus, we present an unambiguous variant of a practically effective branching scheme called the reliability branching [1].

4.2 Unambiguous Reliability Branching Scheme

Reliability branching (*rel*) is a hybrid scheme that attempts to combine the benefits of *str* and the pseudocost branching scheme; see [1] for a detailed description

of these schemes. *rel* updates the pseudocost scores of variables when they are used for branching as the BB tree evolves. However, as shown in Table 2, using information from all the processed nodes could result in ambiguous branching. Hence, we present a scheme that uses limited information available in the BB tree in a way that avoids ambiguities.

***ancestRel* Branching**

An unambiguous version of *rel* can be obtained by using pseudocosts only from a node P_i and its ancestors.

Since *ancestRel* uses some node-processing information generated in the tree, it possibly provides better branching decisions compared to simple unambiguous branching schemes like *lex*.

A variable x_j with the maximum branching score is chosen for branching. Any ties between candidates with the same score are broken lexicographically.

Corollary 1. *The ancestRel branching strategy with the lexicographic tie-breaking rule is unambiguous.*

Proof. Since, the pseudocosts from only the ancestor nodes are used to calculate scores, the selected branching candidate is independent of the number of processors used to generate this tree by Proposition 1. Also, the set of branching candidates, $\mathcal{I}_C$, at a node P_i is not ambiguous because the NLP subsolver used is deterministic, and the lexicographic tie-breaking rule ensures that a unique branching candidate is chosen. Hence, the conditions of Definition 4 are satisfied by *ancestRel*. □

Compared to *sharedRel* branching rule in [17] that is ambiguous, the implementation of *ancestRel* incurs a storage overhead because at each node, it stores a list of variables used for branching at its ancestors, their pseudocosts, the number of times they are branched, and the iteration when they are last strong-branched. Parameters like the number of variables on which strong branching must be applied and the limit on the number of NLP/LP iterations for strong branching are as per default settings of Minotaur.

4.3 A Hybrid Unambiguous Node Selection Strategy

Node selection strategies like depth-first, width-first or best-first along with tie-breaking rules have been shown to be unambiguous [10,13]. State-of-the-art solvers generally use hybrid node selection mechanisms to mitigate the drawbacks and combine the advantages of the above mentioned search strategies. We show that one such hybrid strategy called the best-then-dive strategy coupled with a tie-breaking rule is unambiguous. This strategy first selects a node with the best lower bound, and then keeps diving (processing one of the two immediate child nodes) until a node is pruned.

Let P_i be a node that has been processed and branched, and has an optimal value $\hat{z}$. Assume that P_j is the child node preferred for diving, we assign the following heuristic function values to the child nodes of P_i:

$$h(P_j) = \begin{cases} -\infty, & \text{if } P_j \text{ is the preferred child of } P_i, \\ \hat{z}, & \text{otherwise.} \end{cases}$$

In case a node gets pruned, an open node with the lowest lower bound value is selected. Very often, multiple such nodes exist, for which an unambiguous tie-breaking mechanism is required.

Proposition 2. *The best-then-dive node selection strategy with the* $2s\_2s + 1$ *tie-breaking rule is unambiguous.*

We omit the proof as it is easy to show that both the conditions of Definition 3 are met by this strategy.

4.4 Nondetrimental NLP-BB

We denote by *mcbnbDeter*, the parallel extension of NLP-BB in Minotaur that uses the following unambiguous components.

- best-then-dive node selection strategy with the $2s\_2s + 1$ tie-breaking rule
- *ancestRel* branching strategy with the lexicographic tie-breaking rule
- a deterministic NLP solver.

In addition to the above, we require synchronization of threads at few stages in the algorithm, for example, to avoid passing ambiguous initial conditions to the subsolvers, as well as unambiguity of other algorithmic components. We also disabled "guided diving" in Minotaur, which sometimes causes ambiguity depending upon when the best solution is obtained during an iteration.

Theorem 2. *The algorithm mcbnbDeter satisfies* $T(k, 0) \leq T(1, 0)$ *for* $k > 1$.

Proof. The result follows from [10, Theorem 1] and the unambiguity of the node selection function (Proposition 2) and the branching function (Corollary 1), and due to the deterministic NLP solver (Assumption 2). □

5 Reducing Detrimental Anomalies in Parallel QG

In this section, we formally define the notion of unambiguity for cutting planes, an integral component of branch-and-cut based algorithms.

Definition 5. *A vector valued function* $\pi(\cdot)$ *for generating linearization cuts at a node* P_i *is referred to as unambiguous if* $\pi(\cdot)$ *depends only on the information obtained from* P_i *and its ancestors.*

We consider the basic linearization cuts in QG, which are obtained using an integer optimal solution x^* obtained at a node P_i. An NLP is solved in which the integer variables are fixed to their values in x^*. Using a point $\hat{x}$ returned from the NLP solver, a linearization cut is obtained. The point $\hat{x}$ is either an integer feasible solution to (P) or a point from a feasibility problem [2, Sec. 3.2.1] that minimizes some measure of constraint violation at this node. In traditional QG, these linearizations are applied to all open nodes in $\mathcal{P}$. However, we store and apply these cuts only to the descendants of P_i to avoid generation of ambiguous relaxations or nodes when using different number of threads. We refer to the strategy that uses this cut generating function as *cutGenQG*.

Proposition 3. *The cut generation strategy cutGenQG is unambiguous.*

Proof. Since the integer feasible LP solution x^* at a node is obtained from a deterministic LP solver, the resulting fixed NLP will not be ambiguous. This ensures that the point $\hat{x}$ returned from a deterministic NLP solver, and hence, the cuts generated at P_i are not ambiguous. All descendants of P_i exhibit similar behavior, hence, *cutGenQG* satisfies the conditions of Definition 5. □

Next, we denote by *mcqgDeter*, the parallel extension of QG in Minotaur that uses the following unambiguous components.

- best-then-dive node selection strategy with the $2s\_2s + 1$ tie-breaking rule
- *ancestRel* branching strategy with the lexicographic tie-breaking rule
- *cutGenQG* cutting plane strategy
- a deterministic NLP solver and a deterministic LP solver

Theorem 3. *The algorithm mcqgDeter satisfies* $T(k, 0) \leq T(1, 0)$ *for* $k > 1$.

Other linearization inequalities proposed in [17] for QG can also be included in an unambiguous way in *mcqgDeter*. However, by restricting the application of these cuts only to descendant nodes, the relaxations at other nodes would be weaker than those in the traditional QG and might result in a larger BB tree.

Reproducibility of Results The use of unambiguous components in mcbnbDeter and mcqgDeter results in a deterministic behavior of these algorithms. Reproducibility in parallel algorithms is desired for performance analysis, debugging during code development, etc. In Minotaur, we provide appropriate options that synchronize various subroutines, and ensures unambiguity.

6 Computational Results

We have carried out our computational experiments on a server with two 64-bit Intel(R) Xeon(R) E5-2670 v2, 2.50 GHz CPUs with 10 cores each and sharing 128 GB RAM. Our schemes are implemented in Minotaur[2]. The algorithms are written in C++ and complied with GCC-4.9.2. We use OpenMP-4.0 provided by

[2] Available at http://github.com/minotaur-solver/minotaur.

GCC for our parallel constructs. IPOPT-3.12 with MA97 linear-systems solver is used for solving NLPs and CPLEX-12.8 for solving LPs. We have disabled hyperthreading to highlight the effect of explicit parallelism. We have used 374 convex instances from MINLPLib [5] (we refer to them as testset TS). A limit of one hour on the wall clock time has been used for all our experiments.

First, we briefly mention the performance of opportunistic algorithms. *mcbnbOppor16* could solve 29 additional instances compared to *mcbnbOppor1* and reduced the time taken by more than 60%. This performance is better compared to the opportunistic schemes reported earlier in [17].

The "scalability graphs" [17] for *mcbnbOppor* are shown in Fig. 3. The plot for *mcbnbOppor1* is a base step function for which the peak value (about 0.71 here) indicates the fraction of instances solved using *mcbnbOppor1*. The ordinate corresponding to a value at, say 2^{-1}, indicates the fraction of instances that are solved by a multithreaded variant by a factor of two or more as compared to *mcbnbOppor1*. For example, *mcbnbOppor16* solves 40% of the instances at least twice as fast as *mcbnbOppor1*. The rightmost values on the plots show the fraction of instances that could be solved within the time limit.

mcbnbDeter can be run in Minotaur using the following options: *–brancher ancestRel –tb_rule* $2s\_2s+1$ *–mcbnb_deter_mode 1*. The scalability graphs in terms of wall clock time are shown in Fig. 4. Overall, the performance of *mcbnbOppor* is better than *mcbnbDeter* in terms of wall clock time because the former exploits parallelism in an opportunistic way. However, *mcbnbDeter* variants can provide a guarantee to not be worse than *mcbnbDeter1* and also be reproducible.

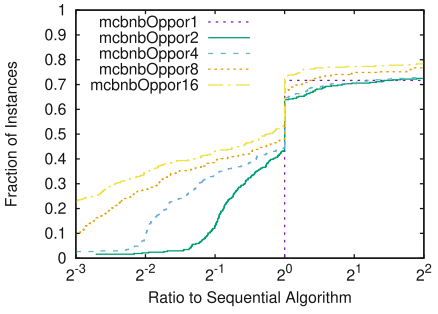

Fig. 3. Scalability graphs of wall clock times taken by *mcbnbOppor* on TS.

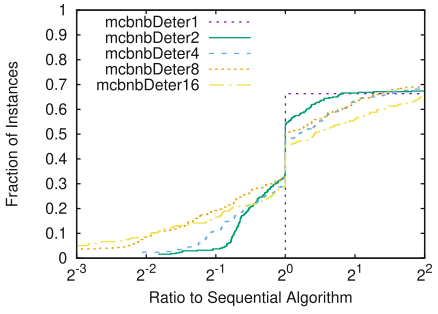

Fig. 4. Scalability graphs of wall clock times for *mcbnbDeter* without guided diving.

We are able to avoid detrimental anomalies in *mcbnbDeter* using the above mentioned options in Minotaur in terms of the number of iterations except for seven instances. Disabling guiding diving (using `--guided_dive 0`) eliminates detrimental anomalies in three of these instances. The remaining instances exhibit anomalies due to ambiguity induced by floating point precision in Minotaur and the subsolvers. Figure 4 and 7 demonstrate good scalability of *mcbnbDeter* both in terms of wall clock times and the number of iterations, respectively.

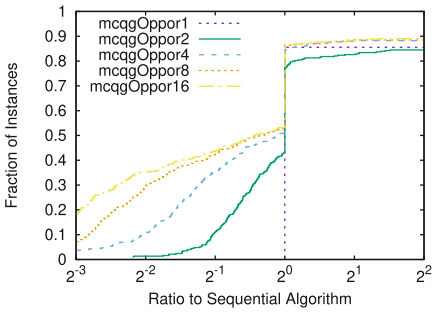

Fig. 5. Scalability graphs of wall clock times taken by *mcqgOppor* on TS.

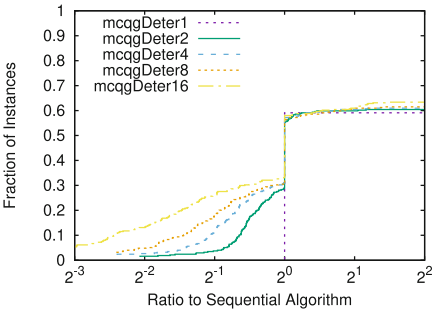

Fig. 6. Scalability graphs of wall clock times taken by *mcqgDeter* on TS.

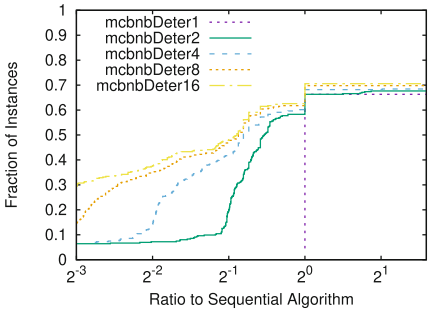

Fig. 7. Scalability graphs of no. of iterations for *mcbnbDeter* without guided diving.

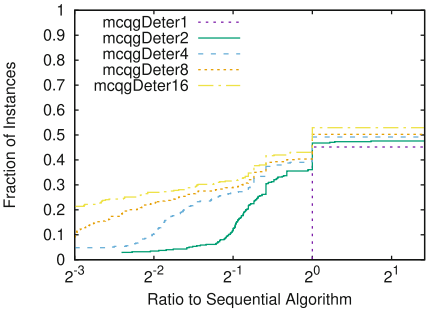

Fig. 8. Scalability graphs of number of iterations taken by *mcqgDeter* on TS.

Figure 5 shows the scalability graphs for the opportunistic variants of *mcqg*. *mcqgOppor16* could solve about 40% of the instances in half the time compared to *mcqgOppor1*. For difficult instances, improvement up to 88% is obtained, and overall, 16 additional instances could be solved. *mcqgDeter* also scales well with the number of threads in both wall clock times and the number of iterations as shown in Fig. 6 and 8, respectively. For 41 instances, *mcqgOppor* exhibits anomalous behavior in terms of wall clock time. We note that this list is longer compared to that corresponding to *mcbnbOppor*. On the other hand, five instances solved by multithreaded *mcqgDeter* took more iterations than that by *mcqgDeter1*. Again, this is because of the ambiguities induced by floating point precision in Minotaur.

7 Conclusions and Future Directions

It is important to study anomalies in parallel branch-and-bound algorithms to design better strategies in practice that can enhance scalability of parallel algorithms. We addressed detrimental anomalies in two convex MINLP algorithms,

NLP-BB and QG by extending the notion of unambiguity to functions for variable branching and generating cuts. We also showed that these theoretical ideas can translate to practically effective algorithmic components such as hybrid node selection strategies like best-then-dive (instead of pure strategies like depth-first, best-first, etc.), branching rules like *ancestRel* (instead of lexicographic branching rule), etc. Our computational experiments show that detrimental anomalies can be eliminated to a great extent practically. Opportunistic versions perform better in terms of wall clock times than the deterministic versions on average, because deterministic versions tend to synchronize more and incur some extra intervals of waiting time. However, deterministic versions can avoid detrimental anomalies with guarantees and also provide reproducible results.

The analysis presented so far depends on the number of iterations. It remains to be explored how unambiguity and speed can be achieved simultaneously. Also, unambiguity can be extended to other MINLP algorithms like the MILP based outer approximation. Another immediate extension of our work is to avoid general k_1-k_2 anomalies in MINLP algorithms where $k_2 > k_1 > 1$ is the number of processors used.

References

1. Achterberg, T., Koch, T., Martin, A.: Branching rules revisited. Oper. Res. Lett. **33**(1), 42–54 (2005)
2. Belotti, P., Kirches, C., Leyffer, S., Linderoth, J., Luedtke, J., Mahajan, A.: Mixed-integer nonlinear optimization. Acta Numer **22**, 1–131 (2013)
3. Berthold, T., Farmer, J., Heinz, S., Perregaard, M.: Parallelization of the FICO Xpress-Optimizer. Optim. Methods Softw. **33**(3), 518–529 (2018)
4. Boukouvala, F., Misener, R., Floudas, C.A.: Global optimization advances in mixed-integer nonlinear programming, MINLP, and constrained derivative-free optimization. CDFO. Eur. J. Oper. Res. **252**(3), 701–727 (2016)
5. Bussieck, M.R., Drud, A.S., Meeraus, A.: MINLPLib - a collection of test models for mixed-integer nonlinear programming. INFORMS J. Comput. **15**(1), 114–119 (2003)
6. Crainic, T.G., Le Cun, B., Roucairol, C.: Parallel branch-and-bound algorithms. Parallel Comb. Optim. **1**, 1–28 (2006)
7. Hart, W.E., Phillips, C.A., Eckstein, J.: PEBBL: An object-oriented framework for scalable parallel branch and bound. Technical report, Sandia National Laboratories (SNLNM), Albuquerque, NM (United States) (2013)
8. Kannan, R., Monma, C.L.: On the computational complexity of integer programming problems. In: Henn, R., Korte, B., Oettli, W. (eds.) Optimization and Operations Research. LNEMS, vol. 157, pp. 161–172. Springer, Berlin, Heidelberg (1978). https://doi.org/10.1007/978-3-642-95322-4_17
9. Lai, T.H., Sahni, S.: Anomalies in parallel branch-and-bound algorithms. Commun. ACM **27**(6), 594–602 (1984)
10. Li, G.J., Wah, B.W.: Coping with anomalies in parallel branch-and-bound algorithms. IEEE Trans. Comput. **100**(6), 568–573 (1986)
11. Li, G.J., Wah, B.W.: Computational efficiency of parallel combinatorial OR-tree searches. IEEE Trans. Softw. Eng. **16**(1), 13–31 (1990)

12. Mahajan, A., Leyffer, S., Linderoth, J., Luedtke, J., Munson, T.: Minotaur: a mixed-integer nonlinear optimization toolkit. Math. Program. Comput. **13**, 1–38 (2020)
13. Mans, B., Roucairol, C.: Performances of parallel branch and bound algorithms with best-first search. Discret. Appl. Math. **66**(1), 57–74 (1996)
14. Menouer, T.: Solving combinatorial problems using a parallel framework. J. Parallel Distrib. Comput. **112**, 140–153 (2018)
15. Quesada, I., Grossmann, I.E.: An LP/NLP based branch and bound algorithm for convex MINLP optimization problems. Comput. Chem. Eng. **16**(10–11), 937–947 (1992)
16. Sahinidis, N.V.: Mixed-integer nonlinear programming 2018. Optim. Eng. **20**(2), 301–306 (2019). https://doi.org/10.1007/s11081-019-09438-1
17. Sharma, M., Palkar, P., Mahajan, A.: Linearization and parallelization schemes for convex mixed-integer nonlinear optimization. Comput. Optim. Appl. **81**, 1–56 (2022)
18. Shinano, Y., Heinz, S., Vigerske, S., Winkler, M.: FiberSCIP - a shared memory parallelization of SCIP. INFORMS J. Comput. **30**(1), 11–30 (2017)