Noname manuscript No.
(will be inserted by the editor)

Machine Learning to Balance the Load in Parallel Branch-and-Bound

Alejandro Marcos Alvarez · Louis Wehenkel · Quentin Louveaux

Received: date / Accepted: date

Abstract We describe in this paper a new approach to parallelize branch-and-bound on a certain number of processors. We propose to split the optimization of the original problem into the optimization of several subproblems that can be optimized separately with the goal that the amount of work that each processor carries out is balanced between the processors, while achieving interesting speedups. The main innovation of our approach consists in the use of machine learning to create a function able to estimate the difficulty (number of nodes) of a subproblem of the original problem. We also present a set of features that we developed in order to characterize the encountered subproblems. These features are used as input of the function learned with machine learning in order to estimate the difficulty of a subproblem. The estimates of the numbers of nodes are then used to decide how to partition the original optimization tree into a given number of subproblems, and to decide how to distribute them among the available processors. The experiments that we carry out show that our approach succeeds in balancing the amount of work between the processors, and that interesting speedups can be achieved with little effort.

Keywords Parallel branch-and-bound · Hardness estimation · Machine learning · Unit commitment

Mathematics Subject Classification (2010) 90C57 · 90C27 · 68T20

A. Marcos Alvarez, L. Wehenkel, Q. Louveaux
Department of Electrical Engineering and Computer Science
Université de Liège
Liège, Belgium
E-mail: {amarcos,l.wehenkel,q.louveaux}@ulg.ac.be

1 Introduction

Branch-and-bound (B&B), and its variants, is probably the most popular algorithm used to solve mixed-integer programming (MIP) problems. Throughout the years, its internal mechanisms were improved and many additional features such as cutting planes, advanced branching strategies and presolve have been added to the core algorithm. These many improvements made it easy to solve problems of ever increasing size. However, since B&B is mainly devoted to solving NP-hard problems, some of them remain nowadays still too difficult to be solved by a single sequential B&B.

Parallelizing B&B on a large number of computers is a promising way to solve those problems that remain out of reach of traditional approaches. This rationale is strongly motivated by two arguments. First, B&B is a natural candidate for parallelization since it relies on the divide-and-conquer paradigm. Parallelizing B&B is indeed conceptually quite simple, and mainly consists in dividing the original optimization tree in several subtrees, or subproblems, and let each processor, or worker, work on its own part of the global tree. Two parallel implementations mainly differ in the way the original work is split among the available workers, and by the amount of communication involved in the optimization. The second argument in favor of parallel B&B is the explosion of parallel computing and affordable massively parallel computers that has been witnessed in the last two to three decades.

Based on these observations, many researchers started developing parallel B&B algorithms. One of the first reported attempts to parallelize B&B dates back to 1975 and is summarized in a 1988 paper by Pruul et al (1988). In that paper, Pruul et al describe a simple approach to parallelize B&B on a shared memory serial computer. They report a set of experimental results obtained on the travelling salesman problem and analyze the efficiency of their approach. One important finding is that the number of explored nodes might be less in the parallel case than in the serial case. As a consequence, the achieved speedup computed from the number of explored nodes might be higher than the number of processors on which the B&B has been parallelized. These findings further support the idea that parallel B&B is a front-running candidate to solve large MIP problems. It has to be noted however that there exist special cases where the parallel version of B&B performs worse than its serial counterpart (see, e.g., Lai and Sahni 1984).

Very early, balancing the load of each worker of a parallel B&B has become a major concern. Indeed, as for any parallel algorithm, load balance is a crucial aspect that must addressed in order to achieve interesting speedups. Throughout the years, several load balancing schemes have been proposed. For instance, El-Dessouki and Huen (1980) propose a mixed static and dynamic balancing scheme that gives each processor the responsibility to compute its own subtree, and allows them to help other processors when their own workload has been exhausted. Later, Karp and Zhang (1988) proposed a fully dynamic work distribution method that automatically balances the load of each worker by sending the newly created children to random processors.

Rao and Kumar (1987) also proposed several load balancing schemes tailored to different parallel architectures (see also Kumar and Rao 1987).

A common shortcoming of all the dynamic load balancing schemes is that they imply a large amount of communications. It became very early clear that the overhead cost induced by communication times was a major concern for all parallel B&B implementations. On the one hand, communication is desirable because it allows to better balance each processor's load by ensuring that no processor remains idle while others are working. Moreover, communication can also reduce the total amount of work to be done by all processors by sharing information about feasible solutions. But, despite its advantages, communication between processors remains very expensive and should be limited to its minimum. Laursen (1994) was one of the first to propose a method in which the processors do not communicate with each other and that allocates statically the workload to each worker. Of course, the key factor of success of this approach is to evaluate accurately enough the difficulty of the subtree given to each processor. If the workload is not well balanced between the workers, the utilization of the processors will not be optimal. One of the approaches proposed by Laursen consists in finding a function that predicts the number of nodes of a subtree, i.e., its difficulty, from a set of characteristics extracted from the subtree. Laursen carried out a series of experiments with several functions constructed from simple functional forms like the exponential or the logarithm. Unfortunately, each considered function was not able to consistently predict the difficulty of several classes of problems. Laursen concluded that it was not trivial to find such a function. Later, the idea proposed by Laursen (1994) has been further explored in a more principled approach by Otten and Dechter (2012) who used machine learning techniques to create a function that can be used to predict the difficulty of a subproblem based on easily computable features. They reported good results for a given class of MIP problems represented over graphical models solved by an AND/OR branch-and-bound.

In a slightly different fashion, Wah and Yu (1985) and Yang and Das (1994) have developed interesting approaches to evaluate the difficulty of a subproblem. Both approaches are based on probabilistic models that are used to predict the complexity of a subproblem. Despite the encouraging results that they report, the assumptions required by the probabilistic models seem very strong and unrealistic for a wide variety of problems.

The interest in parallel B&B is not limited to the field of optimization. Indeed, parallelizing B&B has also attracted some attention in the computer science community that developed several frameworks aimed at easing the implementation of personal specialized parallel B&B algorithms (see, e.g., Eckstein et al 2001; Dorta et al 2004). It must be noted that the pieces of work reported in this paper are by no means exhaustive, and we refer the reader to Gendron and Crainic (1994)'s paper for a wider, though older, survey of parallel B&B techniques.

Based on the previously made observations and on previous work, and further supported by the conclusions drawn by Linderoth (1998, p. 197), we

propose in this paper a new approach using machine learning to balance the load between several processors, and apply our approach to a set of unit commitment (UC) problems. The main contribution of this work is the development of an approach using machine learning to create and distribute several subproblems to a given number of processors such that the workloads of each processor are not too dissimilar. Moreover, we develop a set of new features that allow to represent a subproblem in order to predict its difficulty, in terms of the number of nodes. The experimental results show that the approach succeeds in efficiently balancing the load between several processors, and that it achieves interesting results with and without communication. It is to be noted that the developed features do not depend on the class of problems used to assess our method. They are virtually applicable to any type of MIP problem, although some adaptation might be necessary to improve the performance on a given class of problems. Moreover, we must emphasize that machine learning is mainly useful when the considered problems are related to each other, otherwise it is in general difficult for the algorithm to learn something from the available data. Because of these requirements, the proposed approach is primarily applicable to the situations where similar problems have to be repeatedly solved over time. Focusing on unit commitment (UC) problems is thus a straightforward choice since generation companies, or transmission system operators, have to repeatedly solve very similar UC problems again and again.

In the remainder of the paper, we start first, in Section 2, by giving some preliminary information about the considered problem and by introducing important concepts. We then describe the method that we propose in Section 3. A short theoretical analysis is next carried out in Section 4. Section 5 then describes a series of experiments that we perform in order to validate our approach. Finally, Section 6 concludes this paper and draws some lines of future work.

2 Preliminaries

We first describe here in a more detailed way the problem addressed in this paper, and give a brief introduction to machine learning for the beginner.

2.1 Problem statement

In this paper, we want to develop an efficient parallel version of a branch-and-bound algorithm that minimizes the amount of communication and that achieves high speedups. To do so, we will split the original optimization tree into several subtrees that cover together the initial tree. Each subtree is then given to a processor (a processor can be responsible for several subtrees) that is asked to solve the subproblem defined by each subtree. Communication between processors is ideally forbidden, but a small amount can still be allowed in

order to benefit from the solutions found by other processors. Because communication should be maintained at its minimum, the workload of each processor, i.e., the difficulty of the given set of subtrees, should ideally be balanced between all processors so that high speedups can be achieved. Balancing the workload is the problem tackled in this paper.

In the context of optimization, the workload is basically the time a solver needs to find the optimal solution, but other difficulty measures are also commonly used. For instance, in the case of B&B, it is acknowledged that the number of nodes explored by the algorithm before optimality is proved favorably estimates the difficulty of a problem. In this paper, we focus on the latter difficulty measure, i.e., the number of nodes, as it is more robust to perturbations during the experiments, and roughly linearly dependent (up to a time factor) on the optimization time. Consequently, the focus of this paper is to develop a method that balances the numbers of nodes of the subtrees that each processor is responsible for.

In this work, we address binary Mixed-Integer Linear Programming (MILP) problems of the form

$$\begin{aligned} \min \quad & \boldsymbol{c}^\top \boldsymbol{x} \\ \text{s.t.} \quad & \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b} \\ & x_j \in \{0,1\} \quad \forall j \in I \\ & x_j \in \mathbb{R}_+ \quad \forall j \in C, \end{aligned} \tag{1}$$

where $\boldsymbol{c} \in \mathbb{R}^n$, $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ and $\boldsymbol{b} \in \mathbb{R}^m$ respectively denote the cost coefficients, the coefficient matrix and the right-hand side. I and C are two sets containing the indices of the integer and continuous variables respectively. We denote the solution at a given node of the B&B by $\boldsymbol{x}^*$ and we will call, with a little abuse, the variable x_i, with $i \in I$, a fractional variable if it has a fractional value in the current solution $\boldsymbol{x}^*$.

2.2 Introduction to machine learning

Machine learning (ML) is the field of artificial intelligence that is concerned by the automatic construction, or learning, of functions from data. Let us assume we are interested in some task $\mathcal{T}$ that maps states s belonging to $\mathcal{S}$, which is the set of possible input states of the task, to an output space $\mathcal{Y}$. Basically, ML focuses on the construction of functions f imitating the behavior of $\mathcal{T}$. More specifically, the functions f map inputs from a space Φ to the output space $\mathcal{Y}$, i.e., $f(\cdot) \in \mathcal{F} : \Phi \mapsto \mathcal{Y}$, where $\mathcal{F}$ is the set of possible mappings from Φ to $\mathcal{Y}$. Formally, a machine learning algorithm $\mathcal{A}$ is a procedure of the form

$$\mathcal{A} : (\Phi \times \mathcal{Y})^N \mapsto \mathcal{F}$$

that takes as input a dataset $D = ((\boldsymbol{\phi}_i, y_i))_{i=1}^N \in (\Phi \times \mathcal{Y})^N$, and that outputs a function $f \in \mathcal{F}$ that minimizes some loss function $\mathcal{L}$ on the dataset D.

Stated in mathematical terms, the function f^* resulting from the application of algorithm $\mathcal{A}$ to dataset D is given by

$$f^* = \mathcal{A}(D) = \underset{f \in \mathcal{F}}{\operatorname{argmin}} \sum_{i=1}^{N} \mathcal{L}(y_i, f(\boldsymbol{\phi}_i)).$$

Note that the set $\mathcal{F}$ of possible output functions depends on the particular class of machine learning algorithm that is used.

Ideally, the input of function f should be $s \in \mathcal{S}$, the input state of $\mathcal{T}$ itself. However, representing the complete state is often a difficult problem, e.g., because its dimensionality is too big, or because it contains a lot of irrelevant information. For this reason, in the machine learning community, the inputs $\boldsymbol{\phi}$ of the functions f are usually 'features' representing a simplified version of the input state. Formally, features are (vectors of) characteristics extracted heuristically from the input state $s \in \mathcal{S}$, i.e.,

$$\mathcal{C}(s) = \boldsymbol{\phi} \in \Phi \subset \mathbb{R}^d,$$

such that those features represent part of the current state, ideally the part that most influences the output. The features often critically condition the efficiency of learning methods. As they represent only part of the current state of the task, it is important that the parts described by the features are indeed correlated to the desired output. For this reason, the features need to be carefully designed and tailored to the problem of interest.

The strength of machine learning relies on its ability to generalize behaviors observed on data with very few assumptions needed. This makes it a powerful tool when one wants to imitate unknown functions for which no, or very little, information is available. The main requirement is that the machine learning procedure needs a dataset containing pairs of inputs and outputs $(\boldsymbol{\phi}_i, y_i)$ obtained from the task $\mathcal{T}$ that the ML algorithm is trying to imitate. Those input-output pairs can be obtained by simulation or through a black-box function, there is no need to actually know the functional representation of the real underlying function to be learned.

3 Description of the method

In this section, we describe the method that we devised in order to balance the load of each individual processor of a parallel branch-and-bound. We first describe how we generate a set of subproblems that span the entire optimization tree. We next describe how the subproblems can be allocated to the different processors.

3.1 Generating a partition of the original optimization tree

The approach that we propose to generate a partition of the optimization tree is very much alike a traditional branch-and-bound. It is represented in Algorithm 1.

In this algorithm, we generate a partition of the original optimization tree containing at most k elements. From now on, the notation p represents a subproblem of the original problem, i.e., a problem for which a certain number of binary variables are fixed either to 0 or 1. In particular, p_0 designates the root node, i.e., a version of the original problem in which no binary variable is fixed. The algorithm first starts with p_0 that is added to a queue. Then, the procedure is as follows. The algorithm retrieves and removes a subproblem p from the queue and iteratively creates a certain number of children of p by setting each unfixed binary variable in p to 0 and then to 1. Thus, for each unfixed binary variable, we create two children by fixing that variable either to 0 or to 1. A set of features describing each child created in that way is then computed with a given function $\mathcal{C}$. The computed features are subsequently used as input of a learned complexity function f_{nodes} that returns an estimate of the number of nodes required to solve the child subproblem represented by the features. The predictions of the numbers of nodes of each child are next used to compute a score according to which the unfixed binary variables in the subproblem p are ranked. Once every unfixed binary variable has been scored, the two child subproblems corresponding to the variable that has the lowest score are added to the queue. The presented procedure is then repeated by removing from the queue the subproblem whose predicted number of nodes is the greatest, until the queue fulfills a given stopping criterion or until a maximum queue size has been reached.

In the end of the procedure, each element of the queue represents the root node of a subtree forming the sought partition of the original optimization tree.

The behavior of the proposed partitioning algorithm depends on three main factors: the function f_{nodes} predicting the number of nodes of a subproblem; the way the features are computed, i.e., the implementation of function $\mathcal{C}(\cdot)$; and the implementation of the function $score(\cdot)$. The rest of this section details how these functions were implemented in this work.

3.1.1 Assessing the difficulty of a subproblem

In order to create a partition of the original optimization tree that balances well the workload of each processor, our procedure requires that a function able to predict the difficulty of a subproblem is available. As previous research indicates (Wah and Yu 1985; Yang and Das 1994; Laursen 1994), it is not trivial to find a simple mathematical formulation for such a function. We thus decided to resort to machine learning in order to create that function.

There exist several machine learning frameworks that could be used to construct a function. In this work, we apply the supervised learning framework. In this approach, a dataset containing input-output pairs is needed by the machine learning algorithm to construct the desired function. The input-output pairs should be observations of the system that the function is supposed to imitate. In this work, the inputs consist in feature vectors, i.e., vectors of scalars, that represent some characteristics of the subproblem. The output of

Algorithm 1 Optimization tree partitioning algorithm

$q = p_0$ ▷ p_0 is the root node
while true **do**
  if $|q| \geq k$ **then** ▷ k is the maximum number of elements in the partition
    break
  else if $|q| \geq 3N$ **then** ▷ N is the number of available processors
    if $\max_{p \in q} f_{\text{nodes}}(\mathcal{C}(p)) \leq \frac{3}{4N} \sum_{p \in q} f_{\text{nodes}}(\mathcal{C}(p))$ **then**
      break
    end if
  end if
  $p = \text{argmax}_{p' \in q} f_{\text{nodes}}(\mathcal{C}(p'))$
  $s^* = +\infty$
  for $i \in U_p$ **do** ▷ U_p is the set of indices of the unfixed binary variables in p
    $p_{\text{left}} = p$ with x_i set to 0
    $\hat{n}_{\text{left}} = f_{\text{nodes}}(\mathcal{C}(p_{\text{left}}))$ ▷ $\mathcal{C}$ is the function that generates the features
    $p_{\text{right}} = p$ with x_i set to 1
    $\hat{n}_{\text{right}} = f_{\text{nodes}}\left(\mathcal{C}\left(p_{\text{right}}\right)\right)$ ▷ $\hat{n}$ is an estimate of the number of nodes of p
    $s = score(\hat{n}_{\text{left}}, \hat{n}_{\text{right}})$
    if $s < s^*$ **then**
      $s^* = s$
      $p^*_{\text{left}} = p_{\text{left}}$
      $p^*_{\text{right}} = p_{\text{right}}$
    end if
  end for
  $q = q \setminus p \cup \left\{p^*_{\text{left}}, p^*_{\text{right}}\right\}$
end while
return q

the function is the number of nodes the subproblem requires in order to be solved. The output thus gives an idea of how difficult a subproblem is.

More specifically, the function that we learn is

$$f_{\text{nodes}} : \Phi \subset \mathbb{R}^d \mapsto \mathbb{R},$$

where Φ is the set of possible feature vectors that is included in $\mathbb{R}^d$. In principle, the output should be an integer, but this is not guaranteed by the machine learning algorithm. We thus allow the estimated number of nodes to be a general scalar instead of an integer.

The supervised learning framework requires that a dataset of input-output pairs observed from the system we are trying to imitate is available for learning. In our case, such a dataset is not available and we must thus create one so that our method can be applied. In order to do that, we first select a set of problems. We next randomly generate, for each problem in this set, a certain number of subproblems by randomly fixing each variable in a subset of the binary variables to 0 or 1. For instance, if the problem contains 20 binary variables, we will select a random subset of them, and randomly fix each variable in this subset to 0 or 1. The created subproblem is then optimized until optimality is reached. This gives the number of nodes, and, hence, the difficulty of the subproblem that will be the output part of a pair of the dataset. The input part is given by the feature vector that will be computed for the randomly

generated subproblem. This procedure is repeated until enough data has been generated.

Once the dataset is created, we can apply a supervised machine learning algorithm in order to learn the function f_{nodes} from the observed data. In this work, we used the random forests algorithm (Breiman 2001).

3.1.2 Computing the features of a subproblem

As mentioned in the previous section, the input of the function f_{nodes} should be a vector of scalars. This section describes how these features are computed for a given subproblem.

A subproblem p is created by fixing a certain amount of binary variables either to 0 or to 1. We denote by F_p the set of the indices of the fixed binary variables, and the set of unfixed binary variables by U_p. The indices of the variables fixed to 0 and 1 are respectively contained in the sets F_{p0} and F_{p1}. Note that we assume that all problems are in the form (1) and that we will use the same notations.

We denote the LP solution of the root node of the original problem by $\boldsymbol{x}_0^*$, and the solution at the root of the subproblem p by $\boldsymbol{x}_p^*$. Similarly, the value of the objective function obtained with the solutions $\boldsymbol{x}_0^*$ and $\boldsymbol{x}_p^*$ are denoted o_0^* and o_p^*, respectively. We moreover assume that a heuristic solution is available from the beginning. The heuristic function $h(\cdot)$ applied to solution $\boldsymbol{x}_0^*$ gives the solution $\boldsymbol{x}_0^h = h(\boldsymbol{x}_0^*)$, and the value of the objective function for this solution is o_0^h. This heuristic solution allows us to compute the initial gap g_i at the root node of subproblem p, that is,

$$g_i = \frac{o_0^h - o_p^*}{o_0^h}.$$

Note that this gap can be negative since the LP objective of the subproblem can be greater than the objective of the heuristic solution computed at the root node of the original problem.

Additionally, we compute, for each subproblem p, a new right hand side $\overline{b}_i$, for each constraint i. Indeed, since some variables are fixed in p, the values of their coefficients in the constraint matrix $\boldsymbol{A}$, multiplied by their value, can be subtracted from the initial $\boldsymbol{b}$. We thus define the new right hand side as

$$\overline{b}_i = b_i - \sum_{j \in F_{p1}} A_{ij},$$

since it is not necessary to subtract the coefficients of the variables fixed to 0.

In order to compute additional features, we also optimize, for a very short period of time, the subproblem p with a traditional B&B. More specifically, we allow 5,000 nodes to be explored. Note that the algorithm uses as primal bound the value of the objective function found by the heuristic at the root node, i.e., o_0^h. When this budget is exhausted, we extract a certain number of characteristics from the optimization. This phase is called probing. At the end

of the optimization budget, we retrieve the dual bound $o_{dual}^{probing}$ and the new primal bound $o_{primal}^{probing}$. With these values, we can compute the final gap g_f at the end of the probing with

$$g_f = \frac{o_{primal}^{probing} - o_{dual}^{probing}}{o_{primal}^{probing}}.$$

Besides the previous values that must be recomputed for each new subproblem p, we carry out some preliminary calculations whose results will be subsequently used to extract characteristics of any subproblem p. More specifically, we compute the relative objective increase observed between the root node of the original problem and the subproblem created by fixing a specific binary variable x_j, with $j \in I$, to 0 or 1. We thus obtain two vectors $\boldsymbol{oi}_0$ and $\boldsymbol{oi}_1$, such that

$$oi_0(j) = \frac{o^*_{p_{x_j=0}} - o^*_0}{|o^*_0|} \quad \text{and} \quad oi_1(j) = \frac{o^*_{p_{x_j=1}} - o^*_0}{|o^*_0|},$$

where $p_{x_j=0}$ (respectively $p_{x_j=1}$) is the subproblem created by fixing variable x_j to 0 (respectively 1), and leaving all other variables unfixed. These vectors are computed once and for all in the beginning, and will be used to compute some of our features.

The above description merely introduces some notations and some values used to compute our features describing a subproblem. The complete list that we use in this work is given in Table 1. In this table, the features are separated into five categories, each one of which is meant to represent different dynamics of the problem. The first category of features captures basic characteristics of the subproblem, as well as some differences between the subproblem and the original root problem, like the increase of the LP objective between the roots of both problems. The second category aims at representing the different interactions that exist between the fixed binary variables of the subproblem and the other binary variables in the cost function and in the constraints. Then, the features in the third category model the sparsity of the subproblem with different measures computed from the subproblem and the original problem. The fourth category is similar to the second one except that its goal is to evaluate the connections between all variables (fixed, unfixed binary and continuous variables) in the objective function as well as in the constraints. Finally, the fifth category contains the features that are computed after the probing phase. These features give a small glimpse of the optimization of the subproblem.

3.1.3 Scoring a variable

In the algorithm that we propose, a score is used to determine which variable it is better to branch on in order to expand the current tree by two newly created nodes. The way the score is computed influences the behavior of the

Table 1 Features used to describe a subproblem

Feat. #	Description
1	$\left\|o_0^* - o_p^*\right\| / \left\|o_0^*\right\|$
2-3	$\|F_{p0}\|/\|I\|$ and $\|F_{p1}\|/\|I\|$
4	$\|U_p\|/\|I\|$
5	$\left(\sum_{j\in F_{p0}} \|0 - x_0^*(j)\| + \sum_{j\in F_{p1}} \|1 - x_0^*(j)\|\right) / \|F_p\|$
6-9	$\min_{j\in F_{p0}} oi_0(j)$, plus the max, mean and std of those values
10-13	$\min_{j\in F_{p1}} oi_1(j)$, plus the max, mean and std of those values
14-17	$\min_{j\in U_p} 0.5 * oi_0(j) + 0.5 * oi_1(j)$, plus the max, mean and std of those values
18	$\left(o_0^h - o_p^*\right) / \left\|o_0^h\right\|$
19	$\sum_{j\in F_p} c_j / \sum_{j\in I} c_j$
20	$\sum_{j\in U_p} c_j / \sum_{j\in I} c_j$
21-22	$\min_{i=1\ldots m} \left(b_i - \sum_{j\in F_{p1}} A_{ij}\right) / b_i$, plus the max of those values
23-24	$\min_{i=1\ldots m} \sum_{j\in U_p} A_{ij} / b_i$, plus the max of those values
25-26	$\min_{i=1\ldots m} \left(\sum_{j\in U_p} A_{ij} - \overline{b}_i\right) / \overline{b}_i$, plus the max of those values
27-28	$\min_{i=1\ldots m} \sum_{j\in F_p} A_{ij} / \sum_{j\in I} A_{ij}$, plus the max of those values
29-30	$\min_{i=1\ldots m} \sum_{j\in U_p} A_{ij} / \sum_{j\in I} A_{ij}$, plus the max of those values
31-34	$\text{mean}_{j\in F_p} \left\|A_{:j}\right\|_0 / m$, plus the min, max, and std of those values
35-38	$\text{mean}_{j\in U_p} \left\|A_{:j}\right\|_0 / m$, plus the min, max, and std of those values
39-42	$\text{mean}_{j\in C} \left\|A_{:j}\right\|_0 / m$, plus the min, max, and std of those values
43	$\sum_{j\in F_p} c_j / \sum_{j=1}^n c_j$
44	$\sum_{j\in U_p} c_j / \sum_{j=1}^n c_j$
45	$\sum_{j\in C} c_j / \sum_{j=1}^n c_j$
46-47	$\min_{i=1\ldots m} \sum_{j\in F_p} A_{ij} / \sum_{j=1}^n A_{ij}$, plus the max of those values
48-49	$\min_{i=1\ldots m} \sum_{j\in U_p} A_{ij} / \sum_{j=1}^n A_{ij}$, plus the max of those values
50-51	$\min_{i=1\ldots m} \sum_{j\in C} A_{ij} / \sum_{j=1}^n A_{ij}$, plus the max of those values
52-53	$\min_{i:\overline{b}_i\geq 0} \left(\sum_{j\in U_p:A_{ij}\geq 0} A_{ij} + \sum_{j\in C:A_{ij}\geq 0} A_{ij}\right) / \overline{b}_i$, plus the max of those values
54-55	$\min_{i:\overline{b}_i\geq 0} \left(\sum_{j\in U_p:A_{ij}< 0} A_{ij} + \sum_{j\in C:A_{ij}< 0} A_{ij}\right) / \overline{b}_i$, plus the max of those values
56-57	$\min_{i:\overline{b}_i< 0} \left(\sum_{j\in U_p:A_{ij}\geq 0} A_{ij} + \sum_{j\in C:A_{ij}\geq 0} A_{ij}\right) / \overline{b}_i$, plus the max of those values
58-59	$\min_{i:\overline{b}_i< 0} \left(\sum_{j\in U_p:A_{ij}< 0} A_{ij} + \sum_{j\in C:A_{ij}< 0} A_{ij}\right) / \overline{b}_i$, plus the max of those values
60	$\left(o_{dual}^{probing} - o_p^*\right) / o_{dual}^{probing}$
61	$\left(o_0^h - o_{primal}^{probing}\right) / o_0^h$
62	ratio between the number of open nodes left after the probing budget is exhausted and the number of explored nodes
63	maximum depth of the probing tree
64	depth of the last full level (i.e., the level l such that the numbers of nodes in the levels $l' = 1 \ldots l$ are $2^{l'}$) in the probing tree
65	waist of the probing tree (i.e., level l with the largest number of nodes)
66	$100 * (g_i - g_f)/\|g_i\|$

tree partitioning algorithm. The first score that we propose aims at balancing the difficulty, i.e., the number of nodes, of each newly created children. The proposed score is as follows:

$$score(\hat{n}_{\text{left}}, \hat{n}_{\text{right}}) = \frac{\hat{n}_{\text{left}} + \hat{n}_{\text{right}}}{2} + \left| \frac{\hat{n}_{\text{right}} - \hat{n}_{\text{left}}}{2} \right| + \left| \frac{\hat{n}_{\text{left}} - \hat{n}_{\text{right}}}{2} \right|,$$

where $\hat{n}_{\text{left}}$ and $\hat{n}_{\text{right}}$ respectively indicate the predicted size of the left and right subproblems.

The other proposed scoring criterion is designed such that the total amount of work is minimized. It is given by

$$score(\hat{n}_{\text{left}}, \hat{n}_{\text{right}}) = \max\left(\hat{n}_{\text{left}}, \hat{n}_{\text{right}}\right).$$

This score does not take into account the difficulty equilibrium between the two created nodes. It is assumed that the balance can be achieved later when the generated subproblems are distributed to the workers.

3.2 Distributing nodes to processors

In the case where the number k of generated subproblems is equal to the number of processors, the distribution of the work is trivial. If the number of nodes in the partition is greater than the number of processors, one must find a way to distribute the work evenly between the processors such that the workload is well balanced between each worker.

There exist several ways this distribution could be done. In this work, we applied a simple greedy method although other, more formal, approaches are applicable. The greedy routine that we use is detailed in Algorithm 2. The output of this algorithm is an array, one element per processor, of queues specifying which subproblems have to be solved by a given processor.

Algorithm 2 Greedy subproblem allocation

let N be the number of processors
let h be an array of N scalars
let q be an array of N queues
let l be a list containing the subproblems of the tree partition
$h(i) = 0$, $\forall i$
$q(i) = \emptyset$, $\forall i$
while $l \neq \emptyset$ **do**
    $p = \text{argmax}_{p' \in l} f_{\text{nodes}}(p')$ ▷ select the most difficult remaining subproblem
    $k = \text{argmin}_{i=1\ldots N} h(i)$ ▷ identify the queue whose expected workload is the least
    $h(k) = h(k) + f_{\text{nodes}}(p)$
    $q(k) = q(k) \cup p$ ▷ add the current subproblem to the chosen queue
    $l = l \setminus p$
end while
return q

A more formal approach to allocate the subproblems to the workers is to solve the following problem:

$$\begin{aligned}
\min \quad & \sum_{i=1}^{N} z_i && (2)\\
\text{s.t.} \quad & \sum_{j=1}^{k} a_{ij}\hat{n}_j = w_i \quad \forall i = 1 \dots N \\
& \sum_{i=1}^{N} a_{ij} = 1 \quad \forall j = 1 \dots k \\
& m - w_i \leq z_i,\ m - w_i \geq -z_i \quad \forall i = 1 \dots N \\
& a_{ij} \in \{0, 1\},\ w_i, z_i \in \mathbb{R}^+,
\end{aligned}$$

where k and N respectively correspond to the number of subproblems to be allocated and to the number of processors, and $\hat{n}_j$ and m respectively represent the predicted number of nodes of a subproblem and the ideal average load of each processor, i.e., $\frac{\sum_j \hat{n}_j}{N}$.

4 Theoretical analysis

In this section, we present a short theoretical analysis that can be derived from our method. Indeed, one of the side advantages of using machine learning is that it is possible to get in advance, i.e., without performing the optimization, an estimate of the number of nodes that a subproblem needs in order to be solved to optimality. Moreover, the error of this estimate can be estimated as well. In the following, we show how these estimates can be used to get an approximation of the speedup of the method before the optimization is carried out.

In general, it is easier to reach an equilibrium between the workloads of each processor when the chunks that have to be distributed are smaller. For this reason, it might be better to generate a number of subproblems that is greater than the number of processors. Each worker j thus has a queue q_j containing the subproblems for which it is responsible. Thanks to machine learning, the number of nodes n_i that a subproblem p_i requires can be estimated with the learned function, i.e., $\hat{n}_i = f_{\text{nodes}}(p_i)$. In practice however, the prediction is not perfect, and we can assume that the real number of nodes n_i required to solve the subproblem to optimality is a random variable that is distributed around the predicted value $\hat{n}_i$.

Assuming that the mean of n_i is $\hat{n}_i$ and that its standard deviation is $\hat{\sigma}_i$, the following theorems characterize a given subproblem allocation. We propose two theorems that characterize, respectively, the speedup obtained with a parallel work distribution, and its absolute duration, i.e., the maximum number of nodes over all processors. Both theorems provide information that can be

used in different situations. Indeed, the speedup is useful when one wants to characterize whether the processors are efficiently used, while the absolute duration is of interest when one wants to know how quickly an optimization job will terminate. Note that, in this context, the speedup has to be understood as the ratio between the total amount of work carried out by all processors (i.e., the sum of the numbers of nodes of all processors), and the largest amount of work (i.e., the largest number of nodes over all processors).

Theorem 1 (Speedup approximation) *Let k be a number of subproblems that have been generated in such a way that their union covers the entire original optimization tree, and let each subproblem p_i be allocated to one queue q_j of one of the N available workers w_j, the speedup SU obtained by this work distribution is bounded below by l_{SU} and above by u_{SU} with probability ε, i.e.,*

$$l_{SU} \leq SU \leq u_{SU},$$

where $l_{SU} = 1 + \frac{l(N-1)}{u}$ and $u_{SU} = N$, with probability at least

$$\varepsilon = \prod_{j=1}^{N} \rho\left(l, u, \mu_{w_j}, \sigma_{w_j}\right),$$

with $\mu_{w_j} = \sum_{t:p_t \in q_j} \hat{n}_t$, $\sigma^2_{w_j} = \sum_{t:p_t \in q_j} \hat{\sigma}^2_t$, and

$$\rho\left(l, u, \mu_{w_j}, \sigma_{w_j}\right) = \int_l^u \frac{1}{\sigma_{w_j}\sqrt{2\pi}} \exp - \frac{\left(x - \mu_{w_j}\right)^2}{2\sigma^2_{w_j}} \mathrm{d}x.$$

Proof The main mechanisms of the proof of this theorem are based on the probability theory. Since we assume that $k > N$, each worker is responsible for the optimization of a certain number of subproblems. The total number of nodes required in order to finish a 'job' is thus the sum, over all subproblems p_i optimized by a processor w_j, of the number of nodes n_i that these subproblems require in order to be fully optimized. We define a new random variable G_{w_j} for the worker w_j such that

$$G_{w_j} = \sum_{i:p_i \in q_j} n_i.$$

Assuming that the central limit theorem applies in this situation, and that the variables n_i are independent of each other, the random variable G_{w_j}, which represents the total amount of work the processor carries out, is distributed according to a normal distribution with parameters

$$\mu_{w_j} = \sum_{t:p_t \in q_j} \hat{n}_t \quad \text{and} \quad \sigma^2_{w_j} = \sum_{t:p_t \in q_j} \hat{\sigma}^2_t.$$

Then, arbitrarily choosing two values l and u, we can compute the probability that the total number of nodes explored by one worker is comprised between l and u:

$$\rho\left(l, u, \mu_{w_j}, \sigma_{w_j}\right) = \int_l^u \frac{1}{\sigma_{w_j}\sqrt{2\pi}} \exp - \frac{\left(x - \mu_{w_j}\right)^2}{2\sigma_{w_j}^2} \mathrm{d}x.$$

Given that the variables n_i are independent of each other, so are the variables G_{w_j}. Thus, the probability ε that the total amount of work carried out by each worker is comprised between l and u is given by

$$\varepsilon = \prod_{j=1}^{N} \rho\left(l, u, \mu_{w_j}, \sigma_{w_j}\right).$$

Finally, the lower and upper bounds l_{SU} and u_{SU} on the speedup can be computed from the bounds l and u on the number of nodes of each worker by

$$l_{\mathrm{SU}} = 1 + \frac{l\,(N-1)}{u} \quad \text{and} \quad u_{\mathrm{SU}} = N,$$

where l_{SU} and u_{SU} respectively represent the worst and the best case. The best bound on the speedup u_{SU} corresponds to the case where all workers carry out the same amount of work. On the other hand, the worst case l_{SU} corresponds to the case where $N-1$ workers carry out an amount of work equal to l, while the remaining worker is responsible for an amount of work equal to u. □

Note that the probability ε given by Theorem 1 is actually a lower bound on the probability that the speedup falls in the range $[l_{\mathrm{SU}}, u_{\mathrm{SU}}]$. Indeed, there are situations where the amounts of work of the processors are outside the range $[l, u]$, but still yield a speedup comprised between l_{SU} and u_{SU}.

This theorem can be used to determine beforehand whether the chosen work distribution would lead to interesting speedups or not. It is to be noted that the previous analysis is valid when communication between processors is forbidden. If the workers are given the possibility to communicate, for example a primal bound, the expected speedup would be greater than the one estimated by Theorem 1. Moreover, we must emphasize the fact that this speedup computation assumes that the serial amount of work, i.e., when a single subproblem (the root) is optimized by a single processor, is equal to the sum of the individual amounts of work of each subproblem. This is not entirely true, but, for the sake of simplicity, this approximation is used in order to compute in advance the speedup for a given work distribution. Thus, rather than giving a real speedup, the previous theorem might be more useful to estimate the actual utilization of the processors.

In addition to the speedup, the mechanisms of the previous theorem can be used to determine the probability that a worker w_j explores more than a given number of nodes.

Theorem 2 *Based on the same assumptions as Theorem 1, the probability ε that each worker explores no more than a given number t of nodes is given by*

$$\varepsilon = \prod_{j=1}^{N} \varphi\left(t, \mu_{w_j}, \sigma_{w_j}\right),$$

with

$$\varphi\left(t, \mu_{w_j}, \sigma_{w_j}\right) = \int_{-\infty}^{t} \frac{1}{\sigma_{w_j}\sqrt{2\pi}} \exp - \frac{\left(x - \mu_{w_j}\right)^2}{2\sigma_{w_j}^2} \mathrm{d}x.$$

Proof The proof of this theorem follows immediately from the fact that the random variables G_{w_j} are normally distributed. The probability that one of these variables is less than a given value t is directly computable, and the probability that all variables are less than t is obtained by computing the product of each individual probability since the G_{w_j} are assumed to be independent of each other. □

Given the presented theorems, and provided that the considered learning algorithm is able to characterize the variance of a prediction, one can easily evaluate the performance of a given partitioning of the original problem and its distribution among several processors. In a similar way, the proposed theorems could be used, with some adaptations, to find the optimal work distribution, instead of evaluating a given subproblem allocation.

5 Experiments

In this section, we first detail our experimental procedure, and then present some of the experiments that we carried out to assess our method, together with their analysis.

5.1 Experimental setting

We describe here the problems that we used to evaluate our approach and the general experimental procedure that lead to the presented results.

5.1.1 Problem sets

We evaluate our approach on a set of unit commitment (UC) problems. The problems that we consider are a minimalist version of UC problems. Their

mathematical form is given by

$$
\begin{aligned}
\min \quad & \sum_{j=1}^{n_{NS}} c_j^{NS} \sum_{i=1}^{T} x_{ij}^{NS} + \sum_{j=1}^{n_S} c_j^S \sum_{i=1}^{T} x_{ij}^S \qquad (3)\\
& + \sum_{j=1}^{n_{NS}} f_j^{NS} \sum_{i=1}^{T} y_{ij}^{NS} + \sum_{j=1}^{n_S} f_j^S \sum_{i=1}^{T} y_{ij}^S + \sum_{j=1}^{n_S} u_j^S \sum_{i=2}^{T} z_{ij}\\
& \forall i = 1 \dots T : \sum_{j=1}^{n_{NS}} x_{ij}^{NS} + \sum_{j=1}^{n_S} x_{ij}^S \geq d_i\\
& \forall i = 1 \dots T,\ \forall j = 1 \dots n_{NS} : x_{ij}^{NS} \leq M_j^{NS} y_{ij}^{NS}\\
& \forall i = 1 \dots T,\ \forall j = 1 \dots n_S : x_{ij}^S \leq M_j^S y_{ij}^S\\
& \forall i = 2 \dots T,\ \forall j = 1 \dots n_S : z_{ij} - y_{ij}^S + y_{i-1j}^S \geq 0\\
& x_{ij}^{NS}, x_{ij}^S \in \mathbb{R}^+;\ y_{ij}^{NS}, y_{ij}^S, z_{ij} \in \{0, 1\}\,.
\end{aligned}
$$

In this formulation, T, n_{NS}, and n_S respectively represent the number of time periods, the number of power plants without startup costs, and the number of plants with startup costs. The other parameters c_j, f_j, and u_j denote the variable, fixed and startup costs of each power plant. Finally, the M_j and d_i respectively denote the nominal (maximum) power of each plant and the demands that have to be satisfied at each time period.

In this work, we consider that the number of periods is 12, and that the number of power plants with and without startup costs is both 5. Moreover, all the UC problems that we consider differ only by the demand of each period, i.e., all the parameters are identical except for the demands that are different for each problem. In order to create our problems, we generate randomly a first set of parameters including the demands, which will constitute the basis of our UC problems. Then, the demands for each problem are randomly updated by adding to the initial vector of demands $\left[\overline{d}_1, \dots, \overline{d}_T\right]$ a unique randomly drawn term d_m, and a random term for each time period d'_i. The final demand vectors are thus of the form $\left[d_m + \overline{d}_1 + d'_1, \dots, d_m + \overline{d}_T + d'_T\right]$, where the d_m changes from problem to problem, and the d'_i from problem to problem and from period to period. We generate one set of 300 problems that constitute a learning set, and a set of 20 problems to evaluate our approach. Those problem sets will be made available online and can be sent upon request.

All our experiments are thus performed on our randomly generated sets. There are two reasons why we decided to use such problems for our experiments. First, machine learning requires that the problems that we use for learning and for testing are similar enough. If the problems in the learning set are too dissimilar from those in the test set, nothing useful for the test can be learned from the provided data. In this first study, we thus decided to focus only on a single class of problems, with similar characteristics. In principle, the approach can be extended to take into account different classes of problems and more dissimilar problems, but this demands more data to be generated

(and considerably more time). Moreover, the features would probably need to be adapted to capture a most likely larger set of problems dynamics. Second, the choice of a set of UC problems with a same cost structure and varying demands is also motivated by its similarity to practical situations. Indeed, in the real world, the generation companies, or the transmission system operators, have to repeatedly solve similar problems with a similar cost structure (the plants do not change very often), but with a varying demand. Our problem setting is thus strongly motivated by an obvious similarity with practical applications.

5.1.2 Experimental procedure

Once the problem sets are at our disposal, our experimental procedure can be applied. It is composed of three steps: (1) we generate a dataset D_{np} of pairs composed of features of subproblems and the corresponding numbers of nodes, (2) we learn from D_{np} a function able to predict the size of a subproblem, and (3) we apply our partitioning algorithm in order to generate several subproblems and analyze the results.

Note that, in all experiments, including steps 1 and 3 of our experimental procedure, we give to B&B an upper (primal) bound on the problem. This primal bound is very loose, and is computed with a simple heuristic that merely consists in rounding up each fractional variable in the LP solution of the root node of the original problem.

Step 1: dataset generation In order to create a dataset of pairs $(\boldsymbol{\phi}_l, n_l)$, we first generate, for each problem in our learning problem set, a certain number (250) of random subproblems. Each subproblem is created by randomly choosing a subset of the binary variables and by randomly fixing each variable in this subset to either 0 or 1. Then, for each subproblem, we compute the features $\boldsymbol{\phi}_l$ corresponding to this subproblem, and we optimize the subproblem until optimality. The number of nodes n_l required to fully optimize the subproblem is added, together with the features vector $\boldsymbol{\phi}_l$, to the dataset D_{np} as a pair $(\boldsymbol{\phi}_l, n_l)$. The dataset D_{np} contains around $75,000$ learning examples and is used as input of the learning algorithm to create the function $f_{\text{nodes}}(\cdot)$.

Step 2: learning a function predicting the number of nodes We now apply a supervised machine learning algorithm to the dataset D_{np} to learn a function predicting the number of nodes required to solve a subproblem until optimality. In this work, we use the *random forests* algorithm (Breiman 2001). Our choice is motivated by the computational efficiency and the simple mechanisms of the random forests. Another advantage is that the performances of the random forests are very robust against the choice of their parameters. The random forests actually have two main parameters: N, which is the number of trees in the ensemble method, and $n_{\min}$, which is the number of training samples contained in a node below which that node becomes a leaf. The number of trees is set to the value of $N = 50$ in our experiments. The parameter $n_{\min}$

controls the complexity of the trees and is set to a value of $n_{\min} = 10$. Because the experiments show that the parameters values have little impact on the performances of the method, the values that we give to those parameters have been chosen based on our experience. The exact understanding of these parameters is beyond the scope of this paper, and we refer the reader to Breiman (2001) for a deeper explanation.

Step 3: comparing several partitioning schemes After having generated the dataset D_{np} and applied the learning algorithm, we can compare our learned partitioning scheme to other schemes. Besides the one proposed in this paper and due to the lack of clear competitors, we have imagined two extremely simple approaches that we use to compare our method with. The first one, that we call 'random', consists in generating a certain number of subproblems partitioning the original optimization tree completely randomly. The procedure is as follows. Imagine that there is a list that stores, such as in B&B, all open nodes. The list is first initialized with the original root node. While the number of nodes in the list is less than the desired number of elements in the partition, the procedure takes one node randomly from the list. That node is examined and the unfixed binary variables are identified. Then, one unfixed binary variable is chosen randomly, and two child nodes are created by fixing the randomly chosen variable to 0 and 1, respectively. This procedure yields a totally random partitioning of the original tree. The second approach that we propose is similar to the previous one, except that the next node to split into two children is not chosen randomly. Indeed, we rather open the nodes in a breadth-first manner such that the tree resulting from the random partitioning is balanced. We naturally name this approach 'balanced'.

When the number of nodes is equal to the number of processors, distributing the work among the different workers is easy. When the number of elements in the partition is greater than the number of processors, we must find a way to distribute the work between them. When the partition is generated with our learned method, we use Algorithm 2 to distribute the work between all workers. When the random or balanced schemes generate the partition, we randomly distribute the subproblems to each worker while balancing the number of subproblems that each processor is responsible for, i.e., we attribute to each processor a number k/N of subproblems.

Note that the subproblems generated by our approach depend on the scoring function that is used. We proposed two different scoring functions in Section 3.1.3. However, our experiments show that the second one, i.e., the max, is more efficient than the first one. Thus, for the sake of conciseness, we focus, from now on, on the scoring function $score(\hat{n}_{\text{left}}, \hat{n}_{\text{right}}) = \max(\hat{n}_{\text{left}}, \hat{n}_{\text{right}})$.

In order to evaluate the proposed approach, we generate, for each problem in our test set, several partitions of increasing size with the three proposed partitioning schemes. We then gather the results and analyze them. Furthermore, we consider a setting without communications and a setting where the best primal bound is known by each processor (updates of this primal bound are performed on a regular basis).

We evaluate our approach on several problems contained in our test set (20 UC problems). CPLEX 12.2 is used as the main B&B solver. Note that presolve is applied to each problem at the root node and then is disabled for the subproblems. Moreover, in order to assess only the performances of the partitioning strategies, we disable heuristics and cuts in CPLEX.

5.2 Experiments and results

We now give some experimental results obtained through our experiments, together with their analysis.

5.2.1 Learning to predict the number of nodes

In this section, we report some results regarding the accuracy of the learned complexity function, i.e., $f_{\text{nodes}}(\cdot)$. In order to quantify the precision of the function, we split the dataset D_{np} into two sets: one set is used for learning the complexity function, and the other set to test the learned function. After the function is learned, we predict, for each feature vector in the second set, a number of nodes, which can then be compared to the real value stored in the test set. Assuming that there are t elements in the test set, the real values (numbers of nodes) contained in the set are denoted by n_i with $i = 1 \dots t$, and the corresponding predictions obtained with the learned function are denoted by $\hat{n}_i$.

In Table 2, we assess the performances of our learned function by computing the correlation and the mean relative error (MRE) between the predicted and real values, where the MRE is computed as follows:

$$\frac{1}{t}\sum_{i=1}^{t}\frac{|n_i - \hat{n}_i|}{\max(1, n_i)}.$$

In addition to computing the correlation and the MRE, we examine several cases. Indeed, the distribution of the errors throughout the dataset is not uniform: the distribution is much denser for smaller errors. For that reason, we try to evaluate the learned function differently depending on the magnitude of the errors. In order to do so, we successively consider a larger number of predictions. More specifically, we set a threshold on the relative error of each predicted value. Then, the extreme (highest) relative errors that are greater than the threshold are discarded from the set, and the performance measures (correlation and MRE) are computed from the remaining values. For instance, the first line of Table 2 represents the correlation and the MRE computed from all predictions that differ by no more than 50% from the real value. The last column of the table indicates the proportion of values that are not discarded by the threshold, and, hence, the proportion of available values that are taken into account for computing the performance measures.

The table shows that the learned function is able to catch the most important dynamics of the subproblems. Indeed, the MRE is maintained at a

Table 2 Performance measures of the learned complexity function. Each line represents a case where extreme (high) errors are removed from the dataset to compute the correlation, the mean relative error (MRE), and the proportion of considered elements (elements whose relative errors are below the given threshold).

Max. relative error (%)	Correlation	MRE (%)	Prop. elem. (%)
50	0.95	11.58	59.83
75	0.86	17.76	68.27
100	0.72	21.77	72.53
125	0.71	25.18	75.37
150	0.68	28.60	77.74
$+\infty$	0.47	188.12	100.00

very acceptable level when the largest errors are removed. For instance, the table indicates that the absolute relative error is less than 50% for around 60% of the predictions, and that, for those predictions, the mean relative error is around 12%. Both the correlation and the MRE get worse when the threshold on the error increases. Note that, even when we allow errors of up to 150%, the MRE remains quite low at around 28% for 77% of the predictions. The last line of the table illustrates the effect of the large errors. The (very) large errors pollute a lot the analysis of the results. Indeed, despite the fact that there are only a small amount of them, the large errors have a huge impact on the MRE and artificially worsen the results.

These results show that there exist some problems for which the predictions are not accurate enough. This was expected since the dataset that we use is obviously too small and does not satisfactorily cover the space of considered problems (especially the largest problems). Consequently, the prediction is less accurate for those problems that are not well represented in the training dataset. However, despite the performance decrease, the computed correlation shows that the predictions remain a good way to assess the difficulty of a subproblem. Furthermore, it is important to mention that we are not interested only in quantitative aspects of the predictions, but also in qualitative aspects. Indeed, being able to tell whether a subproblem will take much longer to solve than another one is a valuable piece of information that can be as useful as the accurate prediction of the number of nodes. In order to improve the prediction quality, more examples coming from other (larger) problems could be added to the set used to learn the prediction function. This would increase the density of examples in those regions of the problem space where the accuracy of the predictions is not high enough.

5.2.2 Important features

Besides the raw prediction accuracy, we have also analyzed the importance of each feature on the ability of the learned function to predict a correct output. There are several ways to assess whether a feature matters or not. In this work, we use two techniques. First, we examine the so-called 'features importances', which are values computed by the random forests at learning time and that

sum to 1 over all features. The greater the feature importance, the more relevant the feature is. Similarly, we use the cost of omission (COO) (Otten and Dechter 2012) to estimate the impact of a feature on the prediction ability. The cost of omission consists in omitting a given feature during the learning and the testing phases. We can then compute the estimated MRE obtained without the chosen feature. The difference between the MRE obtained without the feature and the MRE obtained with all features is an image of how important the feature is. If the value of the COO is positive, this means that the feature is important for the prediction. On the other hand, when the COO is negative, it implies that the feature has a negative impact on the prediction accuracy. Small values of the COO (either positive or negative) indicate that the feature is not very important and could just be a source of noise in the prediction. Note that we use in this work the normalized COO which consists in attributing to the highest COO a value of 100, all other COOs being scaled accordingly. Given that there exist some large errors than have a huge impact on the MRE, and, thus, on the COO, we compute the COOs for different thresholds of the relative errors, just as in the previous section. The results of the features importances are summarized in Table 3 for the 10 most relevant features.

The table indicates that feature #66, i.e., the gap decrease at the end of the probing phase, is very important. Also, and very interestingly, the impact of each feature seems to depend on the amplitude of the prediction errors. For instance, feature #18, i.e., the difference between the heuristic objective value and the LP objective value at the root of the subproblem, seems important for those values that are predicted quite accurately, but it becomes less relevant when larger errors are included in the computation of the COO. The same analysis can be carried out for all 10 features indicating that, although features #63, i.e., the maximum depth of the probing tree, and #66 are clear winners when all predictions are considered, the other features are important too in different situations. However, it would be very interesting to further study the impact of some features, like features #14 (the minimum of the average objective increase for the unfixed variables), #18, and #62 (the ratio between the number of unexplored and explored nodes after probing), on the performances of the parallel B&B, since it seems that removing them has a positive impact on the prediction accuracy when the entire dataset is considered.

5.2.3 Parallel optimization

We now compare the approach that we propose to the trivial ones in a real parallel optimization setting. We apply the three proposed partitioning schemes to our set of test problems and create increasingly larger partitions of their original optimization trees. Note that, in this case, the number of elements in the partition is equal to the number of processors. We perform two types of experiments: with and without communications. In the case where communications are allowed, their sole purpose is to render the best primal bound available

Table 3 Features importances as computed by the random forests algorithm and normalized costs of omission for the 10 most important features.

Feat. #	Feat. imp.	COO with threshold on max. allowed error (%)					
		50	75	100	125	150	$+\infty$
63	0.6193	31.2	25.0	41.8	42.9	35.7	55.4
66	0.3529	-7.2	2.8	100.0	100.0	100.0	100.0
14	0.0052	9.7	-3.3	-2.3	-4.5	-10.9	-5.6
18	0.0043	100.0	100.0	-24.2	-30.1	-32.6	1.3
1	0.0041	4.6	-6.7	4.4	2.3	-0.7	-0.8
59	0.0031	3.5	-3.5	1.6	-2.6	-6.8	2.8
40	0.0017	4.8	-6.9	1.1	-0.9	-1.8	4.4
62	0.0014	20.1	8.1	12.5	9.2	5.8	-1.8
15	0.0011	5.1	-5.2	-0.7	-6.7	-6.5	1.1
61	0.0011	4.2	-8.9	5.5	9.0	10.9	1.6

to all processes. The communication is thus maintained at its minimum but remains yet very useful to achieve good performances. The communication works as follows. There is, in shared memory, a single scalar that stores the objective value of the best known integral solution. Every 10,000 nodes, each processor reads the shared primal bound and updates its local primal bound accordingly. This allows all processors to be aware of the best available solution in order to early prune unpromising branches of the tree. Moreover, each time a new integral solution is found, the processor responsible for that discovery updates, if necessary, the shared primal bound. This mechanism is very useful in reducing the total amount of work carried out by all processors, while being very light in terms of communications.

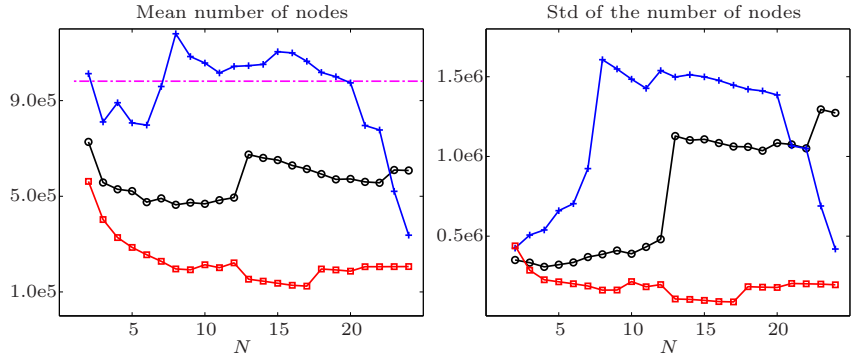
In our experiments, we generate a number of subproblems ranging from 2 to 24, each subproblem being optimized by a single processor. At the end of the experiments, for each problem, the number of nodes explored by each processor is stored, and several measures (average, standard deviation, minimum, maximum, and sum) are computed from the stored values. These measures are finally averaged over all problems in our test set and reported in Figures 1 and 2 versus the number of generated subproblems. Figures 1 and 2 respectively report the results without communications between the processors and with communications. We also report in the results a so-called 'baseline' which consists in the normal optimization of a problem, i.e., starting from the root, the problem is solved to optimality by a single processor. Figures 1 and 2 are just meant to show the trends of the performance measures. The detailed results are given in the form of tables in Appendix A.

A first observation that can be made from the reported results is that our proposed approach, called 'learned', always beats the two trivial approaches in every aspect (considering the same communication setup). The results also highlight the importance of the communications to achieve good performances in parallel B&B. Overall, the mean number of nodes per processor decreases for all partitioning schemes when the number of generated subproblems increases. The same observation can be made for the minimum of the number

of nodes across all processors. The maximum of the number of nodes tends to increase when the number of elements in the partition increases, but only when communications are forbidden. This can be easily understood since the deeper the subproblem is in the optimization tree, the less likely it is to contain a good feasible solution that can prune unpromising branches. Also note that the maximum is the most interesting measure because it conditions the speedup. Indeed, even if the total amount of work required to solve a problem is not equal between the serial and the parallel case, the times needed to complete the optimization is conditioned by the processor that takes the most time. In order to analyze the potential speedups, the maximum number of nodes must thus be compared with the number of nodes in the serial case, i.e., the baseline. The speedups obtained when communications are forbidden are very modest. The situation is otherwise when we allow the processors to communicate. Indeed, in that case, our approach achieves a very interesting 4.22 speedup in comparison with the serial case when the problem is parallelized on 24 cores. This has to be compared with the speedups of 1.36 and 1.58 obtained with the random and balanced partitioning schemes. In other words, the approach that we propose indeed achieves speedups that can have an important impact in practice.

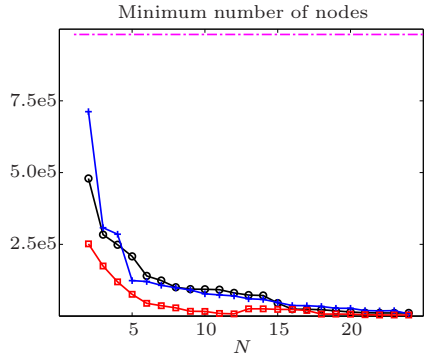
Moreover, it is important to emphasize that the computed speedups are lower bounds on the attainable speedups. Indeed, in this work, we do not perform dynamic load balancing since we only distribute the work to each processor before the optimization starts. If a dynamic load balancing scheme is used to further improve the work equilibrium between the processors, it is conceivable that much higher speedups can be achieved. Indeed, in that case, other measures, such as the mean and the minimum numbers of nodes per processor, should be used to enrich our analysis. Given that the mean and the minimum workload per processor are both very low with our method, it is fair to expect greater gains in computation time if a dynamic load balancing scheme is used, together with our method, in order to redistribute the work from the busiest processor to idle ones.

Finally, the last set of experiments that we propose focuses on the parallel optimization of our set of test problems when the number k of generated subproblems is greater than the number N of processors. In order to do so, we generate a certain number k of subproblems with each considered partitioning scheme. Then, the subproblems are distributed across the N available processors, either by randomly attributing k/N subproblems to each processor (for the random and balanced partitioning schemes), or by using Algorithm 2 (for the learned partitioning scheme). Note that the number of subproblems generated with the learned partitioning scheme is at most k, but may be less given that Algorithm 1 provides another stopping criterion that can be used to stop the generation of subproblems before the limit k is reached. Table 4 reports, in the case where the number of generated subproblems is greater than the number of processors, the same performance measures as those presented in Figures 1 and 2. Note that, in this case, the performance measures (average, standard deviation, etc.) are computed from the total number of nodes

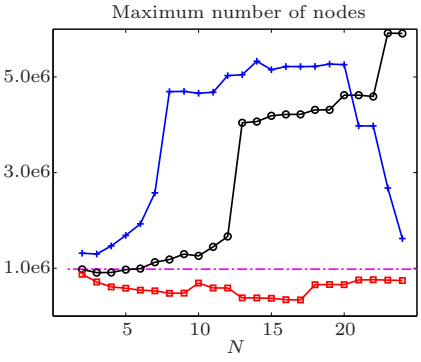


(a) Mean number of nodes per processor

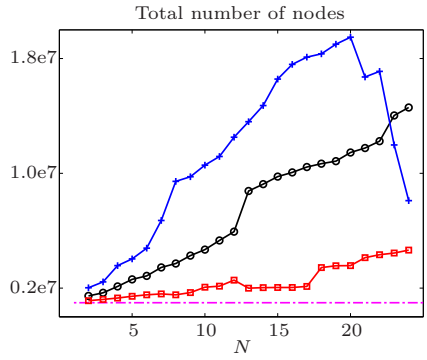
(b) Standard deviation of the number of nodes per processor



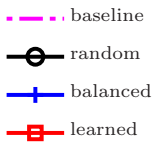
(c) Minimum number of nodes per processor



(d) Maximum number of nodes per processor

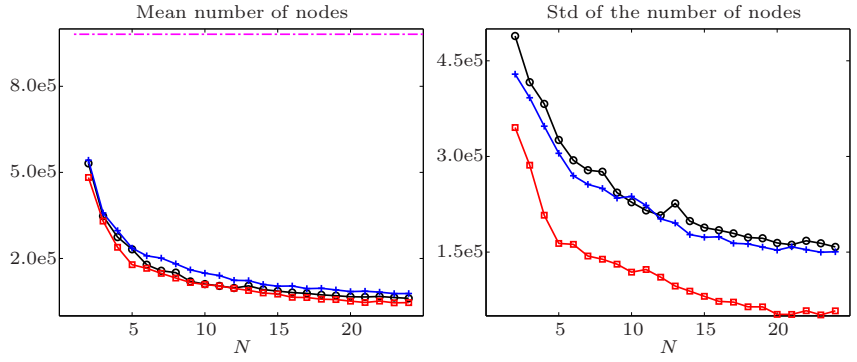


(e) Total number of nodes



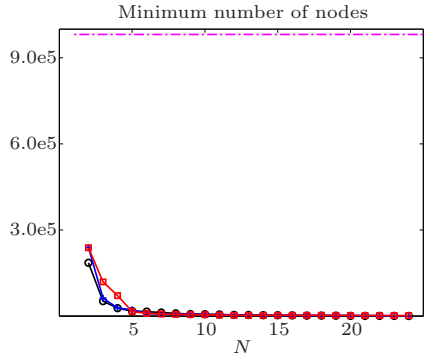
(f) Legend

Fig. 1 Parallel optimization without communications results with an increasing number of generated subproblems (k) and an increasing number of processors (N) (with the number of subproblems being equal to the number of processors, i.e., $k = N$). The different figures report the average, the standard deviation, the minimum, the maximum and the sum of the numbers of nodes that each processor explores until optimality is reached. The values shown in the figures are averages over all problems in our set of test problems.

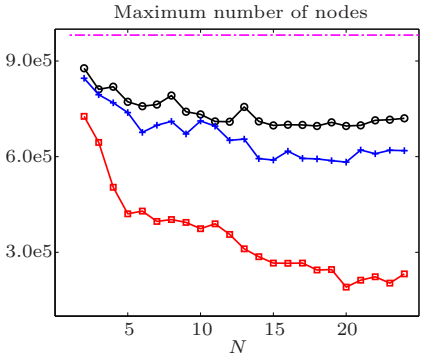


(a) Mean number of nodes per processor

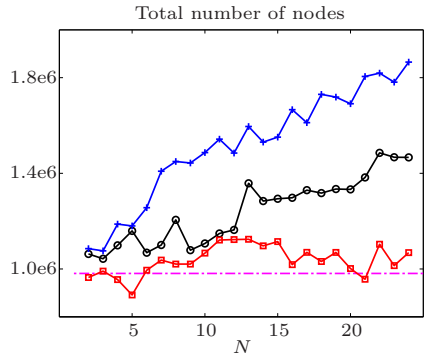
(b) Standard deviation of the number of nodes per processor



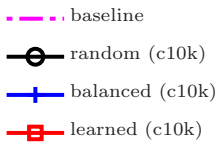
(c) Minimum number of nodes per processor



(d) Maximum number of nodes per processor



(e) Total number of nodes



(f) Legend

Fig. 2 Parallel optimization with communications (every 10,000 nodes) results with an increasing number of generated subproblems (k) and an increasing number of processors (N) (with the number of subproblems being equal to the number of processors, i.e., $k = N$). The different figures report the average, the standard deviation, the minimum, the maximum and the sum of the numbers of nodes that each processor explores until optimality is reached. The values shown in the figures are averages over all problems in our set of test problems.

explored by each processor, which corresponds to the sum of the number of nodes required by each subproblem attributed to the given worker. The total number of nodes is thus an image of the total amount of work that a processor carries out.

Table 4 indicates first that the parallel optimization without communications does not compare very well with the single threaded case. Indeed, the maximum number of nodes that a processor performs is always greater than in the single threaded case. This implies that the parallel optimization does not terminate before the single threaded B&B. However, the approach that we propose compares favorably with the trivial partitioning schemes. Indeed, with our method, the mean number of nodes that a processor explores decreases, as well as the standard deviation. This implies that, on average, the number of nodes per processor is less than for a single threaded optimization. Things are different when communications are allowed between the processors. Indeed, while the trivial partitioning schemes still perform very poorly compared to the single threaded optimization, the proposed method, on the other hand, exhibits very interesting performances. Indeed, in that case, the obtained speedup between the single threaded baseline and the learned partitioning scheme is equal to 4.07 or 5.61, depending on the number of processors. Similarly to the case without communications, the mean number of nodes and the standard deviation per processor are reduced compared to the trivial partitioning schemes, which shows that the load is acceptably balanced between the workers.

Finally, it is interesting to compare the situation where each processor is assigned only one subproblem, with the situation where each processor is responsible for multiple subproblems. This analysis can be carried out by comparing the results from Table 4 with those from Tables 5-9. Comparing both sets of results for 12 and 24 processors yields the following observations. First, the mean and the total number of nodes are roughly equal between both setups. Additionally, we see that the standard deviation per processor decreases when more subproblems are assigned to a single worker, which is a display of better load balance. Lastly, we observe that the difference between the maximum and the minimum number of nodes decreases when the processors are assigned more than one subproblem, which tends, again, to show that the work is better balanced in that case. Overall, the conclusion that can be drawn is that allocating more than one subproblem to each processor does not increase the total amount of work to be done, but sensibly reduces the unbalance between the workers.

6 Conclusions and future work

In this paper, we proposed a new approach to split the optimization of a single problem into several parallel parts with the goal that the amount of work given to each processor is well balanced between the workers. The approach consists in creating a complexity function, with the use of machine learning techniques,

Table 4 Parallel optimization results when the number of generated subproblems (k) is greater than the number of processors (N). The table reports the average, the standard deviation, the minimum, the maximum and the sum (over all processors) of the numbers of nodes that each processor explores until optimality is reached. The values shown in the tables are averages over all problems in our set of test problems. Note that, for the learned partitioning scheme, the k indicates the maximum number of elements in the partition. The real number of generated subproblems is different for each problem and depends on the stopping criterion given in Algorithm 1.

	N	k	Mean	Std	Min	Max	Sum
Single	1	1	9.81e+05	-	9.81e+05	9.81e+05	9.81e+05
Without communication							
random	12	240	7.94e+06	5.70e+06	1.73e+06	1.97e+07	9.53e+07
random	24	480	6.29e+06	5.76e+06	1.00e+06	2.14e+07	1.51e+08
balanced	12	240	1.55e+07	8.81e+06	4.53e+06	3.33e+07	1.86e+08
balanced	24	480	1.25e+07	8.15e+06	3.28e+06	3.79e+07	3.00e+08
learned	12	240	5.96e+05	2.38e+05	1.85e+05	9.89e+05	7.16e+06
learned	24	480	5.54e+05	3.03e+05	1.22e+05	1.18e+06	1.33e+07
With communication (primal bound, every 10,000 nodes)							
random	12	240	1.67e+05	1.89e+05	4.22e+04	6.94e+05	2.00e+06
random	24	480	7.98e+04	1.59e+05	1.03e+04	7.60e+05	1.91e+06
balanced	12	240	1.99e+05	1.50e+05	5.22e+04	5.47e+05	2.39e+06
balanced	24	480	1.44e+05	1.27e+05	3.29e+04	5.73e+05	3.46e+06
learned	12	240	9.38e+04	6.14e+04	1.90e+04	2.41e+05	1.12e+06
learned	24	480	4.58e+04	3.95e+04	6.80e+03	1.75e+05	1.10e+06

that is able to estimate the number of nodes, hence the amount of work, that a subproblem of the original problem requires in order to be fully optimized. To this end, we develop a set of features that are used to characterize a given subproblem in the B&B tree, and use these features as input of the learned complexity function in order to predict the expected number of nodes required to solve this subproblem to optimality. These estimates are then used to create a partition of the original optimization tree so that one or several elements of the partition can be given to each processor. The experiments show that our approach succeeds in balancing the amount of work between the processors and that interesting speedups can be achieved with little effort.

Further research orientations include the development of more relevant features that would better grasp the dynamics of the considered problems in order to better predict the subproblem size. Another research direction is to implement the proposed framework on massively parallel computers to understand how the speedups and processor utilizations change when the original work is split into a very large number of independent parts.

Finally, let us emphasize the fact that, although, in this paper, we totally focus on a single class of MIP problems, the same framework can be transposed to any class of problems with minor adaptations of the proposed features.

Acknowledgements This work was funded by the Dysco Interuniversity Attraction Pole (IAP) of the Belgian Science Policy Office and the Pascal2 Network of Excellence of the European Union. AMA's thesis is funded by a FRIA scholarship from the Fonds de la

Recherche Scientifique-FNRS (F.R.S.-FNRS). The scientific responsibility rests with the authors.

References

Breiman L (2001) Random forests. Machine learning 45(1):5–32

Dorta I, Leon C, Rodriguez C (2004) Parallel branch-and-bound skeletons: Message passing and shared memory implementations. In: Parallel Processing and Applied Mathematics, Springer, pp 286–291

Eckstein J, Phillips CA, Hart WE (2001) Pico: An object-oriented framework for parallel branch and bound. Studies in Computational Mathematics 8:219–265

El-Dessouki OI, Huen WH (1980) Distributed enumeration on between computers. IEEE Transactions on Computers 29(9):818–825

Gendron B, Crainic TG (1994) Parallel branch-and-branch algorithms: Survey and synthesis. Operations Research 42(6):1042–1066

Karp R, Zhang Y (1988) A randomized parallel branch-and-bound procedure. In: Proceedings of the twentieth annual ACM symposium on Theory of computing, ACM, pp 290–300

Kumar V, Rao VN (1987) Parallel depth first search. part ii. analysis. International Journal of Parallel Programming 16(6):501–519

Lai TH, Sahni S (1984) Anomalies in parallel branch-and-bound algorithms. Communications of the ACM 27(6):594–602

Laursen PS (1994) Can parallel branch and bound without communication be effective? SIAM Journal on Optimization 4(2):288–296

Linderoth JT (1998) Topics in parallel integer optimization. PhD thesis, Georgia Institute of Technology

Otten L, Dechter R (2012) A case study in complexity estimation: Towards parallel branch-and-bound over graphical models. In: Proceedings of the Twenty-Eighth Conference on Uncertainty in Artificial Intelligence, pp 665–674

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

Rao VN, Kumar V (1987) Parallel depth first search. part i. implementation. International Journal of Parallel Programming 16(6):479–499

Wah B, Yu CF (1985) Stochastic modeling of branch-and-bound algorithms with best-first search. Software Engineering, IEEE Transactions on SE-11(9):922–934

Yang MK, Das CR (1994) Evaluation of a parallel branch-and-bound algorithm on a class of multiprocessors. Parallel and Distributed Systems, IEEE Transactions on 5(1):74–86

A Complete experimental results

This appendix contains the detailed experimental results obtained when our test problems are optimized by a parallel B&B. We report the mean, standard deviation, minimum, and maximum numbers of nodes observed on each processor, together with the sum of the number of nodes of each processor. The results are then averaged for all problems and reported in the following tables. Note that, in this case, the number k of generated subproblems is equal to the number N of processors.

Table 5 Mean number of nodes

		Without comm.			With comm. (every 10k nodes)		
N	baseline	random	balanced	learned	random	balanced	learned
2	9.81e+05	7.27e+05	1.01e+06	5.62e+05	5.31e+05	5.42e+05	4.82e+05
3	9.81e+05	5.58e+05	8.11e+05	4.03e+05	3.48e+05	3.59e+05	3.30e+05
4	9.81e+05	5.29e+05	8.91e+05	3.27e+05	2.75e+05	2.97e+05	2.39e+05
5	9.81e+05	5.22e+05	8.07e+05	2.86e+05	2.32e+05	2.36e+05	1.78e+05
6	9.81e+05	4.75e+05	7.98e+05	2.56e+05	1.78e+05	2.09e+05	1.66e+05
7	9.81e+05	4.91e+05	9.59e+05	2.28e+05	1.57e+05	2.01e+05	1.48e+05
8	9.81e+05	4.64e+05	1.18e+06	1.96e+05	1.51e+05	1.81e+05	1.32e+05
9	9.81e+05	4.74e+05	1.08e+06	1.93e+05	1.20e+05	1.60e+05	1.16e+05
10	9.81e+05	4.68e+05	1.06e+06	2.14e+05	1.11e+05	1.49e+05	1.09e+05
11	9.81e+05	4.84e+05	1.02e+06	2.02e+05	1.04e+05	1.40e+05	1.04e+05
12	9.81e+05	4.94e+05	1.04e+06	2.22e+05	9.69e+04	1.24e+05	9.57e+04
13	9.81e+05	6.74e+05	1.05e+06	1.53e+05	1.04e+05	1.23e+05	8.86e+04
14	9.81e+05	6.60e+05	1.05e+06	1.45e+05	9.17e+04	1.09e+05	7.99e+04
15	9.81e+05	6.51e+05	1.10e+06	1.36e+05	8.62e+04	1.03e+05	7.60e+04
16	9.81e+05	6.29e+05	1.10e+06	1.28e+05	8.11e+04	1.04e+05	6.46e+04
17	9.81e+05	6.14e+05	1.06e+06	1.24e+05	7.82e+04	9.48e+04	6.40e+04
18	9.81e+05	5.93e+05	1.02e+06	1.96e+05	7.32e+04	9.61e+04	5.82e+04
19	9.81e+05	5.71e+05	1.00e+06	1.92e+05	7.02e+04	9.05e+04	5.70e+04
20	9.81e+05	5.72e+05	9.74e+05	1.87e+05	6.66e+04	8.45e+04	5.12e+04
21	9.81e+05	5.60e+05	7.96e+05	2.06e+05	6.58e+04	8.59e+04	4.67e+04
22	9.81e+05	5.56e+05	7.77e+05	2.06e+05	6.75e+04	8.27e+04	5.14e+04
23	9.81e+05	6.10e+05	5.21e+05	2.06e+05	6.38e+04	7.74e+04	4.54e+04
24	9.81e+05	6.08e+05	3.37e+05	2.06e+05	6.11e+04	7.77e+04	4.61e+04

Table 6 Standard deviation of the number of nodes

		Without comm.			With comm. (every 10k nodes)		
N	baseline	random	balanced	learned	random	balanced	learned
2	-	3.50e+05	4.25e+05	4.39e+05	4.89e+05	4.29e+05	3.45e+05
3	-	3.34e+05	5.07e+05	2.87e+05	4.16e+05	3.92e+05	2.86e+05
4	-	3.07e+05	5.39e+05	2.26e+05	3.82e+05	3.47e+05	2.08e+05
5	-	3.21e+05	6.60e+05	2.14e+05	3.26e+05	3.05e+05	1.64e+05
6	-	3.35e+05	7.04e+05	2.01e+05	2.94e+05	2.70e+05	1.62e+05
7	-	3.69e+05	9.24e+05	1.87e+05	2.78e+05	2.56e+05	1.44e+05
8	-	3.86e+05	1.61e+06	1.61e+05	2.76e+05	2.50e+05	1.39e+05
9	-	4.09e+05	1.55e+06	1.62e+05	2.43e+05	2.35e+05	1.31e+05
10	-	3.90e+05	1.48e+06	2.15e+05	2.28e+05	2.37e+05	1.19e+05
11	-	4.32e+05	1.43e+06	1.82e+05	2.15e+05	2.23e+05	1.23e+05
12	-	4.81e+05	1.54e+06	1.96e+05	2.07e+05	2.02e+05	1.11e+05
13	-	1.13e+06	1.50e+06	1.06e+05	2.26e+05	1.96e+05	9.67e+04
14	-	1.10e+06	1.51e+06	1.03e+05	1.99e+05	1.77e+05	8.87e+04
15	-	1.11e+06	1.50e+06	9.69e+04	1.88e+05	1.73e+05	8.06e+04
16	-	1.08e+06	1.48e+06	8.94e+04	1.84e+05	1.74e+05	7.27e+04
17	-	1.06e+06	1.45e+06	8.76e+04	1.79e+05	1.64e+05	7.16e+04
18	-	1.06e+06	1.42e+06	1.83e+05	1.73e+05	1.63e+05	6.42e+04
19	-	1.04e+06	1.41e+06	1.80e+05	1.72e+05	1.58e+05	6.40e+04
20	-	1.08e+06	1.39e+06	1.78e+05	1.64e+05	1.53e+05	5.23e+04
21	-	1.08e+06	1.07e+06	2.03e+05	1.61e+05	1.58e+05	5.23e+04
22	-	1.05e+06	1.05e+06	2.01e+05	1.68e+05	1.54e+05	5.81e+04
23	-	1.29e+06	6.89e+05	1.99e+05	1.64e+05	1.50e+05	5.14e+04
24	-	1.27e+06	4.19e+05	1.95e+05	1.58e+05	1.51e+05	5.79e+04

Table 7 Minimum number of nodes

		Without comm.			With comm. (every 10k nodes)		
N	baseline	random	balanced	learned	random	balanced	learned
2	9.81e+05	4.80e+05	7.12e+05	2.52e+05	1.86e+05	2.39e+05	2.38e+05
3	9.81e+05	2.84e+05	3.08e+05	1.74e+05	5.20e+04	6.23e+04	1.19e+05
4	9.81e+05	2.48e+05	2.85e+05	1.19e+05	2.75e+04	2.83e+04	7.11e+04
5	9.81e+05	2.08e+05	1.24e+05	7.58e+04	1.74e+04	2.17e+04	1.60e+04
6	9.81e+05	1.40e+05	1.20e+05	4.45e+04	1.55e+04	9.18e+03	9.09e+03
7	9.81e+05	1.24e+05	1.07e+05	3.61e+04	1.24e+04	6.88e+03	7.27e+03
8	9.81e+05	1.00e+05	9.87e+04	2.86e+04	9.40e+03	6.35e+03	5.87e+03
9	9.81e+05	9.38e+04	9.11e+04	1.67e+04	6.88e+03	7.11e+03	5.32e+03
10	9.81e+05	9.33e+04	7.77e+04	1.61e+04	6.71e+03	5.84e+03	4.73e+03
11	9.81e+05	9.18e+04	7.37e+04	9.23e+03	5.52e+03	4.60e+03	3.53e+03
12	9.81e+05	8.07e+04	7.03e+04	7.70e+03	4.00e+03	4.00e+03	2.67e+03
13	9.81e+05	7.32e+04	6.03e+04	2.53e+04	3.71e+03	3.49e+03	2.58e+03
14	9.81e+05	7.18e+04	5.80e+04	2.52e+04	3.03e+03	2.86e+03	2.52e+03
15	9.81e+05	4.51e+04	4.77e+04	2.32e+04	2.75e+03	2.39e+03	2.49e+03
16	9.81e+05	2.40e+04	3.66e+04	2.31e+04	2.27e+03	2.30e+03	2.53e+03
17	9.81e+05	2.38e+04	3.61e+04	1.96e+04	1.65e+03	2.52e+03	2.44e+03
18	9.81e+05	2.20e+04	3.36e+04	7.06e+03	1.54e+03	2.10e+03	2.49e+03
19	9.81e+05	1.80e+04	2.72e+04	6.91e+03	1.24e+03	2.09e+03	2.30e+03
20	9.81e+05	1.42e+04	2.72e+04	6.66e+03	5.40e+02	2.10e+03	2.02e+03
21	9.81e+05	1.16e+04	1.91e+04	4.77e+03	6.11e+02	1.87e+03	1.99e+03
22	9.81e+05	1.09e+04	1.81e+04	3.74e+03	6.09e+02	1.31e+03	1.71e+03
23	9.81e+05	1.09e+04	1.89e+04	3.74e+03	6.06e+02	1.17e+03	1.79e+03
24	9.81e+05	1.09e+04	7.73e+03	3.69e+03	4.63e+02	3.77e+02	1.62e+03

Table 8 Maximum number of nodes

		Without comm.			With comm. (every 10k nodes)		
N	baseline	random	balanced	learned	random	balanced	learned
2	9.81e+05	9.75e+05	1.31e+06	8.73e+05	8.77e+05	8.46e+05	7.26e+05
3	9.81e+05	9.08e+05	1.30e+06	7.14e+05	8.11e+05	7.94e+05	6.45e+05
4	9.81e+05	9.11e+05	1.47e+06	6.09e+05	8.19e+05	7.69e+05	5.04e+05
5	9.81e+05	9.71e+05	1.69e+06	5.83e+05	7.72e+05	7.38e+05	4.21e+05
6	9.81e+05	9.93e+05	1.93e+06	5.41e+05	7.58e+05	6.76e+05	4.29e+05
7	9.81e+05	1.13e+06	2.58e+06	5.28e+05	7.64e+05	6.99e+05	3.97e+05
8	9.81e+05	1.18e+06	4.69e+06	4.76e+05	7.92e+05	7.11e+05	4.03e+05
9	9.81e+05	1.29e+06	4.70e+06	4.80e+05	7.41e+05	6.71e+05	3.94e+05
10	9.81e+05	1.26e+06	4.66e+06	6.90e+05	7.33e+05	7.12e+05	3.74e+05
11	9.81e+05	1.45e+06	4.68e+06	5.87e+05	7.10e+05	6.96e+05	3.89e+05
12	9.81e+05	1.66e+06	5.03e+06	5.86e+05	7.09e+05	6.51e+05	3.56e+05
13	9.81e+05	4.04e+06	5.05e+06	3.80e+05	7.56e+05	6.55e+05	3.11e+05
14	9.81e+05	4.07e+06	5.33e+06	3.79e+05	7.11e+05	5.94e+05	2.86e+05
15	9.81e+05	4.19e+06	5.15e+06	3.72e+05	6.98e+05	5.89e+05	2.66e+05
16	9.81e+05	4.22e+06	5.22e+06	3.43e+05	7.00e+05	6.17e+05	2.66e+05
17	9.81e+05	4.22e+06	5.22e+06	3.36e+05	7.00e+05	5.95e+05	2.66e+05
18	9.81e+05	4.31e+06	5.22e+06	6.56e+05	6.96e+05	5.93e+05	2.45e+05
19	9.81e+05	4.31e+06	5.27e+06	6.59e+05	7.07e+05	5.87e+05	2.46e+05
20	9.81e+05	4.62e+06	5.26e+06	6.57e+05	6.96e+05	5.83e+05	1.91e+05
21	9.81e+05	4.62e+06	3.98e+06	7.55e+05	6.98e+05	6.21e+05	2.13e+05
22	9.81e+05	4.59e+06	3.98e+06	7.60e+05	7.14e+05	6.09e+05	2.23e+05
23	9.81e+05	5.92e+06	2.68e+06	7.52e+05	7.16e+05	6.20e+05	2.03e+05
24	9.81e+05	5.91e+06	1.62e+06	7.43e+05	7.20e+05	6.19e+05	2.32e+05

Table 9 Total number of nodes

		Without comm.			With comm. (every 10k nodes)		
N	baseline	random	balanced	learned	random	balanced	learned
2	9.81e+05	1.45e+06	2.03e+06	1.12e+06	1.06e+06	1.08e+06	9.64e+05
3	9.81e+05	1.67e+06	2.43e+06	1.21e+06	1.04e+06	1.08e+06	9.91e+05
4	9.81e+05	2.12e+06	3.57e+06	1.31e+06	1.10e+06	1.19e+06	9.56e+05
5	9.81e+05	2.61e+06	4.03e+06	1.43e+06	1.16e+06	1.18e+06	8.91e+05
6	9.81e+05	2.85e+06	4.79e+06	1.53e+06	1.07e+06	1.26e+06	9.95e+05
7	9.81e+05	3.44e+06	6.71e+06	1.60e+06	1.10e+06	1.41e+06	1.04e+06
8	9.81e+05	3.71e+06	9.44e+06	1.53e+06	1.21e+06	1.45e+06	1.02e+06
9	9.81e+05	4.26e+06	9.76e+06	1.69e+06	1.08e+06	1.44e+06	1.02e+06
10	9.81e+05	4.68e+06	1.06e+07	2.06e+06	1.11e+06	1.49e+06	1.07e+06
11	9.81e+05	5.32e+06	1.12e+07	2.14e+06	1.15e+06	1.54e+06	1.12e+06
12	9.81e+05	5.93e+06	1.25e+07	2.56e+06	1.16e+06	1.48e+06	1.12e+06
13	9.81e+05	8.77e+06	1.36e+07	1.99e+06	1.36e+06	1.60e+06	1.12e+06
14	9.81e+05	9.25e+06	1.47e+07	2.03e+06	1.28e+06	1.53e+06	1.10e+06
15	9.81e+05	9.77e+06	1.66e+07	2.04e+06	1.29e+06	1.55e+06	1.11e+06
16	9.81e+05	1.01e+07	1.76e+07	2.04e+06	1.30e+06	1.67e+06	1.02e+06
17	9.81e+05	1.04e+07	1.81e+07	2.11e+06	1.33e+06	1.61e+06	1.07e+06
18	9.81e+05	1.07e+07	1.83e+07	3.43e+06	1.32e+06	1.73e+06	1.03e+06
19	9.81e+05	1.08e+07	1.90e+07	3.56e+06	1.33e+06	1.72e+06	1.07e+06
20	9.81e+05	1.14e+07	1.95e+07	3.57e+06	1.33e+06	1.69e+06	1.00e+06
21	9.81e+05	1.18e+07	1.67e+07	4.12e+06	1.38e+06	1.80e+06	9.57e+05
22	9.81e+05	1.22e+07	1.71e+07	4.33e+06	1.49e+06	1.82e+06	1.10e+06
23	9.81e+05	1.40e+07	1.20e+07	4.45e+06	1.47e+06	1.78e+06	1.01e+06
24	9.81e+05	1.46e+07	8.09e+06	4.65e+06	1.47e+06	1.87e+06	1.07e+06