J Glob Optim (2017) 69:547–560
DOI 10.1007/s10898-017-0508-y

On parallel Branch and Bound frameworks for Global Optimization

Juan F. R. Herrera[1] · José M. G. Salmerón[2] · Eligius M. T. Hendrix[3,4] · Rafael Asenjo[3] · Leocadio G. Casado[2]

Received: 11 January 2016 / Accepted: 1 March 2017 / Published online: 10 March 2017
© The Author(s) 2017. This article is published with open access at Springerlink.com

Abstract Branch and Bound (B&B) algorithms are known to exhibit an irregularity of the search tree. Therefore, developing a parallel approach for this kind of algorithms is a challenge. The efficiency of a B&B algorithm depends on the chosen Branching, Bounding, Selection, Rejection, and Termination rules. The question we investigate is how the chosen platform consisting of programming language, used libraries, or skeletons influences programming effort and algorithm performance. Selection rule and data management structures are usually hidden to programmers for frameworks with a high level of abstraction, as well as the load balancing strategy, when the algorithm is run in parallel. We investigate the question by implementing a multidimensional Global Optimization B&B algorithm with the help of three frameworks with a different level of abstraction (from more to less): Bobpp, Threading Building Blocks (TBB), and a customized Pthread implementation. The following has been found. The Bobpp implementation is easy to code, but exhibits the poorest scalability. On the contrast, the TBB and Pthread implementations scale almost linearly on the used platform. The TBB approach shows a slightly better productivity.

✉ Eligius M. T. Hendrix
eligius.hendrix@wur.nl

Juan F. R. Herrera
j.herrera@epcc.ed.ac.uk

José M. G. Salmerón
josemanuel@ual.es

Rafael Asenjo
asenjo@uma.es

Leocadio G. Casado
leo@ual.es

1 EPCC, The University of Edinburgh, Edinburgh, UK

2 Informatics Department, University of Almeria (ceiA3), Almería, Spain

3 Department of Computer Architecture, Universidad de Málaga, Málaga, Spain

4 Operations Research and Logistics, Wageningen University, Wageningen, The Netherlands

Keywords Branch-and-Bound · Load balancing · Shared-memory · Framework · TBB

Mathematics Subject Classification 68P05 · 68W10 · 90C57

1 Introduction

A Branch and Bound (B&B) algorithm is a commonly-used method to solve Global Optimization (GO) problems in a deterministic way [20,26,27,38]. This method iteratively divides the search space and discards those areas where a global solution cannot be found. The procedure generates a search tree, where the root node is the initial search space or feasible area. The algorithm explores nodes or branches of this tree, that represent subspaces of the initial search space. Before a node is branched, it is checked whether it can contain a solution better than the best one found so far by the algorithm using upper and/or lower bounds on its local solution. The node is discarded instead if it cannot.

B&B algorithms make use of a working set where the pending nodes are stored to be processed. The number of elements and the structure of the working set usually depend on the selection method used by the algorithm [28]. A stack or LIFO (Last In First Out) data structure is suitable for depth-first search. For best-first search, one can use several approaches: sorted linked-lists, AVL trees, priority queues, etc. Depth-first search requires less memory than best-first search but depth-first may steer the search towards a non-optimal subspace for some time.

The computational burden to solve a GO problem usually increases exponentially with the dimension of the search space due to the performed exhaustive search. Thanks to parallel computation, a larger number of this type of problems can be solved in a reasonable computational time [29,39]. Parallel B&B algorithms can be classified based on several characteristics [13]. One of them is the distinction between the use of single and multiple working sets. Multiple working sets seem appropriate when the number of processing units is large, because each process can work independently on its working set, avoiding bottlenecks in concurrent access to a shared single set. A hybrid approach may use multiple working sets, where each set can be associated to several processes. Similar hybridization can be applied to selection rules. Hence, the number of possible choices is large.

Literature discusses several strategies to parallelize a B&B algorithm, among others: (1) Parallel evaluation of a node; (2) Parallel processing of nodes of the search tree, i.e., the search tree is built in parallel; and (3) A combination of the strategies (1) and (2). Strategy (1) is suitable for problems where the computational burden of evaluating a node is high. Strategy (2) seems appropriate when the number of generated nodes is high and the burden to evaluate a node is low. The GO problem considered here belongs to the latter class of problems [18]. Therefore, the three approaches studied here will traverse the search tree in parallel.

The question is how the chosen platform influences the ease of programming and whether it affects the performance of a B&B algorithm. To investigate this question, we implemented the same B&B algorithm on three libraries with a varying level of abstraction: (1) Bobpp is a high-level library that greatly simplifies the programming effort; (2) TBB can be regarded as a medium-level abstraction in which the programmer manually has to deal with the task coding, but task execution and scheduling is automatically carried out by the runtime; and (3) Pthread is the lowest-level counterpart in which the user has to tackle both

the task and scheduling coding. Bobpp has been chosen for this study because this framework has been recently used by researchers to solve optimization problems [2,22–24]. It is a representative of specifically dedicated frameworks for parallel B&B of high abstraction level.

Section 2 describes the chosen Global Optimization B&B algorithm for the experiments. Section 3 discusses the libraries and skeletons used to facilitate parallel programming of B&B algorithms. The options on the range of working sets and selection rules to ease the load balancing between the processor units are discussed for the Bobpp framework in Sect. 3.1, TBB in Sect. 3.2 and a custom coding with low-level POSIX threads in Sect. 3.3. A test bed of instances with their required constant for bounding purposes is presented in the results Sect. 4. We report on the measured ease of programming and the consequences of the chosen frameworks on the algorithm performance. Section 5 reports on the findings of the investigation.

2 A B&B Global Optimization algorithm

The objective of the algorithm is to find a minimum point x^*, such that $f^* = f(x^*) = \min_{x \in X} f(x)$, where the search space $X \subset \mathbb{R}^n$ is a non-empty hyper-rectangle, sometimes called a box constrained area, i.e., there exist lower and upper bounds for each dimension. The objective function under consideration does not need to be differentiable nor everywhere (Lipschitz)-continuous.

Algorithm 1 as described in [18], uses a simplicial partition, see [15]. Therefore, the box-constrained area is sub-divided into $n!$ simplices by a standard method [35]. The evaluated points in the search space are the vertices of the simplices and the best function value found thus far f^U is stored as the general upper bound of f^*. Simplex division is performed bisecting the simplex over its longest edge. A lower bound f^L, given by the description in [18], of objective function f is calculated for each simplex based on the function values at its vertices.

Algorithm 1 Simplicial B&B algorithm

Require: $X,\ f,\ K, \delta$
1: Partition X into simplices $S_k,\ k = 1, \dots, n!$
2: Start the working set as $\Lambda := \{S_k : k = 1, \dots, n!\}$
3: The set of evaluated vertices $V := \{v_i \in S_k \in \Lambda\}$
4: Set $f^U := \min_{v \in V} f(v)$ and $x^U := \arg\min_{v \in V} f(v)$
5: Determine lower bounds $f_k^L = f^L(S_k)$ based on K
6: **while** $\Lambda \neq \emptyset$ **do**
7: Extract a simplex S from Λ
8: Bisect S into $S1$ and $S2$ generating x
9: **if** $x \notin V$ **then**
10: Add x to V
11: **if** $f(x) < f^U$ **then**
12: Set $f^U := f(x)$ and $x^U := x$
13: Remove all S_k from Λ with $f_k^L > f^U - \delta$
14: **end if**
15: **end if**
16: Determine lower bounds $f^L(S1)$ and $f^L(S2)$
17: Store $S1$ in Λ if $f^L(S1) \leq f^U - \delta$
18: Store $S2$ in Λ if $f^L(S2) \leq f^U - \delta$
19: **end while**
20: **return** x^U, f^U

As selection rule, depth-first search is used in order to reduce memory requirements and facilitate the memory management [17]. The selection rule will be discussed in Sect. 3. A simplex S is deleted if $f^L(S) > f^U - \delta$, where δ is the search accuracy, i.e. the procedure guarantees that the best function value returned by the algorithm does not deviate more than δ from the optimum function value f^*.

The bounding is based on a so-called upper fitting according to [3]. Consider the objective function f with a global minimum f^* on box-constrained area X. Given a global minimum point x^*, let scalar K be such that

$$K \geq \max_{x \in X} \frac{|f(x) - f^*|}{\|x - x^*\|}, \tag{1}$$

where $\|\cdot\|$ denotes the Euclidean norm. The function $f^* + K\|x - x^*\|$ is an upper fitting according to [3] for an arbitrary $x \in X$. Consider a set of evaluated points $x_i \in X$ with function values $f_i = f(x_i)$, then the area below

$$\varphi(x) = \max_i \{f_i - K\|x - x_i\|\} \tag{2}$$

cannot contain the global minimum (x^*, f^*). Let $f^U = \min_i f_i$ be the best function value of all evaluated points, i.e., an upper bound of f^*. Then the area $\{x \in X : \varphi(x) > f^U\}$ cannot contain the global minimum point x^*.

Now consider a simplex S with evaluated vertices $v_0, v_1, \ldots, v_n$, where $f_i = f(v_i)$. To determine the existence of optimal solution x^* in S, each evaluated vertex (v_i, f_i) provides a cutting cone:

$$\varphi_i(x) := f_i - K\|x - v_i\|. \tag{3}$$

Let Φ be defined by

$$\Phi(S) = \min_{x \in S} \max_i \varphi_i(x). \tag{4}$$

If $f^U < \Phi(S)$, then simplex S cannot contain the global minimum point x^*, and therefore S can be rejected. Notice that $\Phi(S)$ is a lower bound of f^* if S contains the minimum point x^*.

Equation (4) is not easy to determine as shown by Mladineo [25]. Therefore, alternative lower bounds of (4) can be generated in a faster way. We use two of them and take the best (highest) value. An easy-to-evaluate alternative is to consider the best value of $\min_{x \in S} \varphi_i(x)$ over the vertices i. This results in a lower bound

$$\underline{\Phi}_1(S) = \max_i \left\{ f_i - K \max_j \|v_j - v_i\| \right\}. \tag{5}$$

The second lower bound is based on the more elaborate analysis of infeasibility spheres in [5] and developed to non-optimality spheres in [16]. It says that S cannot contain an optimal point if it is covered completely by so-called non-optimality spheres. According to [5], if there exists a point $c \in S$ such that

$$f_i - K\|c - v_i\| > f^U \; i = 0, \ldots, n, \tag{6}$$

then S is completely covered and cannot contain x^*. This means that any interior point c of S provides a lower bound $\min_i \{f_i - K \max_j \|c - v_i\|\}$. Instead of trying to optimize the lower bound over c, we generate an easy-to-produce weighted average based on the radii of the spheres. Consider that $f_i > f^U$, otherwise S can contain an optimum point. Let

$$\lambda_i = \frac{K}{f_i - f^U} \tag{7}$$

and take

$$c = \frac{1}{\sum_j \lambda_j} \sum_i \lambda_i v_i. \tag{8}$$

A second lower bound based on (6) is

$$\underline{\Phi}_2(S) = \min_i \{f_i - K\|c - v_i\|\}. \tag{9}$$

The final lower bound we consider in this paper for the B&B is the best value $f^L(S) = \max\{\underline{\Phi}_1(S), \underline{\Phi}_2(S)\}$.

3 Parallel Branch-and-Bound

Parallelizations of Branch-and-Bound algorithms have been widely studied for a large number of applications and machine architectures. A highly-cited survey was performed in 1994 by Gendron and Crainic [13].

One of the goals of a parallelization is to achieve a trade-off between reduction of parallel overhead and maintaining the cores busy by doing useful work. Parallel overhead is caused among others by communications, memory management and inclusion of new code to handle the parallelism. Regarding the useful work, a parallel version of the algorithm should perform the same computations or evaluations as the sequential one. The Search Overhead Factor (SOF) is defined as the ratio between the work done by the parallel and the sequential versions. Two different types of anomalies can occur [19,21]:

- Accelerating anomalies, where a sharper upper bound f^U is found in earlier stages of the parallel algorithm, than for the sequential version. This reduces the number of evaluated nodes of the search tree.
- Detrimental anomalies, where the parallel version visits more branches of the search tree than the sequential one due to the unawareness of the update of the upper bound f^U.

In general, the use of best-first search and its variations leads to less anomalies than depth-first search, but requires more memory. High memory requirement slows the execution down due to data structure management and the speed/amount of the different levels of the memory hierarchy in the system. Detrimental anomalies also increase execution time, because more sub-problems are evaluated. Notice that initiation of global upper bound f^U with the global minimum f^* causes the number of evaluated sub-problems to be the same and not to depend on the selection rule.

Load balancing among the process units of the parallel system is one of the main problems that arise due to parallel exploration of the search space. Parallel B&B algorithms usually perform a load balancing strategy. The Relative Load Imbalance (RLI) measures the effectiveness of the load balancing strategy [33]. Let W_{tot} be the total work (number of evaluated simplices) performed by p working units (threads) and W_i, $i = 0, \ldots, p-1$, the work performed by working unit i such that $\sum_{i=0}^{p-1} W_i = W_{tot}$. Let $W_{max} = \max_i W_i$. Then, the Relative Load Imbalance is defined as

$$\text{RLI} = 1 - \frac{W_{tot}}{pW_{max}}. \tag{10}$$

RLI is defined on the interval $[0, 1 - (1/p)]$; a value close to zero shows a small load imbalance. For the sake of simplicity, the value of RLI shown in Sect. 4 is normalized towards a range of [0, 100].

Many frameworks have been proposed since 1994 to facilitate the development of parallel B&B algorithms, such as PPBB [37], ZRAM [4], Symphony [31], PICO [9], PeBBL [10], BCP [34], ALPS [40], Bobpp [8], and MallBA [1], to name a few. These frameworks can be classified according to the provided methods (B&B, Branch and Cut, Dynamic Programming, etc.) and the techniques used to code the algorithm (programming language, parallelization libraries, etc). An overview of some of these frameworks can be found in [7]. The use of a framework is not always the best approach in terms of efficiency. A framework offers a general skeleton to code a specific instance of the algorithm [30]. In some cases, the developer is not concerned about aspects like the data structure or the way in which the search is performed. In general, it is difficult for a framework to provide the best performance compared to a custom developed algorithm, because some characteristics of the problem are specific and they are not taken into account by the framework.

B&B methods are more efficient on multicore than on GPU or FPGA systems for problems with few arithmetic computations and challenging memory handling due to the size of the search tree [11]. Our study focuses on this case. We consider three approaches with different levels of abstraction. The first one is based on the Bobpp framework, the second makes use of the Thread Building Blocks (TBB) library and the last one is based on an in-house low-level Pthread library.

3.1 Bobpp framework

Bobpp is a C++ framework that facilitates the creation of sequential and parallel solvers based on search tree algorithms, such as Divide-and-Conquer and Branch-and-Bound [8]. The framework has been used to solve Quadratic Assignment Problems [12], Constrained Programming [22–24], and biochemical simulation [2]. The possible parallelizations are based on Pthreads, MPI or Athapascan/Kaapi libraries. Here, we focus on the behaviour of the B&B algorithms for sequential and threaded versions using Pthreads.

Bobpp provides a set of C++ templates or skeletons on top of which the user has to code some classes in order to configure the behavior of the algorithm. These templates are aimed at facilitating the development of search algorithms over irregular and dynamic data structures. The developer may use the example classes provided in the framework, but may also reimplement these classes to code a more specific algorithm. The Bobpp framework allows different ways to schedule the search (depth-first search, best-first search, etc.) as well as provides templates for different data structures. For instance, our B&B implementation selects depth-first search and relies on a priority queue that serves as a working set storing the simplices that have to be processed. Using one global set, one of the threads extracts a node from the set. Then, it is divided according to the branching rule, generating two or more children. In case the termination rule is not satisfied, the generated nodes are evaluated and added to the set. Otherwise, the nodes are discarded. In order to avoid the bottleneck caused by accessing the same set, several sets (priority queues) can be used. The dynamic load balancing strategy followed in Bobpp relies on work-stealing among sets in order to achieve better performance. The optimal number of sets is difficult to predict, because it depends on the problem characteristics and system resources.

3.2 Threading Building Blocks (TBB)

The Intel® Threading Building Blocks (TBB) library provides a productive framework to develop parallel applications in C++ [32]. TBB facilitates the development of loop and task-

based algorithms with high performance and scalability, even for fine-grained and irregular applications.

TBB class `task_group` is a high-level interface that allows to create groups of potentially parallel tasks from functors or lambda expressions. TBB class `task` is a low-level interface that provides more control but is less user-friendly.

Nodes in a B&B search tree seamlessly map onto tasks. Each thread keeps a "ready pool" of ready-to-run tasks. TBB features a work-stealing task scheduler that automatically takes care of the load balancing among pools. From the developer point of view, using a task-based approach could be simpler than using a threaded-based approach, because the user does not need to code the data structure to store and schedule the pending tasks. The developer only has to spawn the tasks and the task scheduler decides when to execute them.

Potential parallelism is typically exploited by a split/join pattern. Two basic patterns of split/join are supported. The most efficient, but also programming demanding, is the continuation-passing pattern, in which the programmer constructs an explicit "continuation" task. The parent task creates child tasks and specifies a continuation task to be executed when the children complete. The continuation inherits the parent's ancestor. The parent task then exits; it does not block waiting for its children. The children sub-sequently run, and after they (or their continuations) finish, the continuation task starts running. This pattern has been used to develop the B&B algorithm previously described in Sect. 2.

In addition to the productive programming interface and to the work-stealing load balancing, TBB features two additional advantages when it comes to parallel tree traversals. First, TBB exhibits a good trade-off between breadth-first and depth-first traversals of the tree. The first one leverages parallelism, while the second avoids too much memory consumption. TBB relies on breadth-first only when stealing work, but otherwise it is biased towards going deep in its branch until the cut-off criterion [36] (not used in the experimentation) is met and that way, the remaining sub-tree is processed sequentially (in order to avoid generating too many fine-grained tasks). Second, TBB can efficiently keep all the cores busy without oversubscribing them. This is, in TBB only one thread/worker per core should be created, but the programmer is responsible of coding an algorithm that generates enough tasks to feed all the workers.

3.3 Pthreads model

This model is based on the dynamic generation/destruction of threads with (asynchronous) multiple sets [13]. Each thread handles its own working set. This strategy was used to suffer less from memory contention problems than a single set, where the working set is shared by all threads [6].

The execution starts creating one thread in charge of the complete search space. In this model, a thread can create a new thread if there is enough work to share, up to the maximum number of threads `MaxThreads`, defined by the user. The newly generated thread will receive half of the simplices stored in its parent [6]. The best upper bound f^U is shared between the threads using a global shared variable. A thread dies when it ends its assigned work.

The time a core is waiting for a new thread, is given by the time needed by a thread to: (1) check that the number of threads is less than `MaxThreads`; (2) divide its working set; (3) create a new thread; and (4) migrate the new thread to an idle process unit (or less overheaded unit, in case of having more threads than processing units). The migration of threads is done by the Operating System, which is out of the scope of this study. Depth-first search using a stack has been used as selection rule.

4 Experimental results

We first provide the design of experiments in Sect. 4.1. Then the performance of the algorithm on the various platforms is compared in Sect. 4.2. Finally Sect. 4.3 compares the ease of programming for the platforms.

4.1 Design of experiments

Algorithms have been coded in C/C++ and compiled using `gcc` version 4.8.1 with Intel® TBB 4.1 Update 2 library. Experiments have been conducted on a node of the BullX machine, that features two Intel® Xeon® E5-2620 (Sandy Bridge) at 2 GHz with eight cores each (16 cores total), with 20 MB of L3 cache and 64 GB of RAM. The Operating System is Ubuntu Server 12.04. Default OS thread manager was used, without thread affinity.

The test bed developed in [18] was used to have a varying tree size. The test instances can be found in Table 1. Each test function is five-dimensional and is indexed by a number. The used accuracy is $\delta = 0.05\left(\overline{f} - f^*\right)$, where $\overline{f}$ is the maximum function value on the search space. The accuracy has been chosen such that the resulting tree for the largest instance still fits in a desktop computer.

4.2 Performance on the various platforms

The numerical results in Table 1 show the computational effort in terms of the number of evaluated simplices (N. Eval. S.) and the wall-clock time in seconds (Time) for the sequential implementation in the Bobpp framework and for the custom-made implementation in serial C. The number of evaluated simplices is similar in both cases. It is not the same, because there is no selection criterion defined to choose between two simplices of the same depth in the search tree. Regarding the execution time, Bobpp is from two to seven times slower than the serial C version.

Table 2 shows the parallel performance of the Bobpp implementation varying the number pq of priority queues and number of threads. For 2, 4, and 8 threads, the performance is higher if the number of priority queues matches the number of threads. However, the optimal

Table 1 Sequential execution time of the Bobpp and custom-made implementation

No.	Function name	Bobpp version		Custom-made version	
		N. Eval. S.	Time	N. Eval. S.	Time
1	Ackley	1,033,107,720	8281.4	1,010,945,400	1178.6
2	Dixon & Price	123,575,854	299.4	123,575,850	96.0
3	Holzman	219,996,634	518.6	219,996,634	241.7
4	MaxMod	1,877,094,680	3109.1	1,877,094,680	1389.7
5	Perm	166,839,972	2458.2	166,831,502	443.7
6	Pinter	989,052,844	9168.5	989,052,844	1378.4
7	Quintic	261,000,328	670.0	261,009,818	206.4
8	Rosenbrock	175,614,988	433.4	175,613,436	136.4
9	Schwefel 1.2	87,628,502	197.3	87,628,502	68.2
10	Zakharov	603,693,472	1537.2	603,678,276	471.3

Table 2 Elapsed time of the Bobpp version varying the number of priority queues (pq)

	16 threads			8 threads			4 threads			2 threads	
pq	16	8	1	8	4	1	4	2	1	2	1
1	1203	655	1714	1125	1239	1762	2207	2316	2713	4511	4959
2	175	61	204	64	76	198	96	114	210	184	305
3	272	107	363	112	127	346	226	199	370	313	545
4	1493	911	3097	900	1034	3010	1220	1493	3035	2028	4358
5	245	166	272	316	326	367	630	648	681	1264	1336
6	985	687	1641	1225	1323	1796	2417	2524	2860	4874	5340
7	331	128	425	138	171	411	216	263	449	397	663
8	227	86	286	92	109	281	145	173	300	257	444
9	115	43	144	85	53	137	68	84	147	121	210
10	799	292	992	314	363	951	443	600	1045	918	1499

number of priority queues is 8 for 16 threads. For 16 threads and $pq = 16$, workload balancing overhead picks up and in this case RLI (see Sect. 3) is higher (between 6 and 17%), whereas RLI is always less than 9% for the other cases. The workload imbalance is small, but the high cost of the dynamic workload balancing hinders the parallel performance.

Overall, Tables 1 and 2 show a quite poor performance of the Bobpp version on the used platform. Its relative speedup[1] for 16 threads and $pq = 8$ is less than 6, but for test functions 1, 5 and 6 it is between 12 and 14. However, the speedup, that uses the best sequential time as reference (last column of Table 1), using 16 threads is just between 1.5 and 2.2 for all test functions, as can be seen in Fig. 1a. In this figure, the speedup at "$x = 0$ threads" represents the fastest serial version and serves as a reference for the speedup degradation of Bobpp with 1 thread (ranging from 0.14 to 0.47).

Figure 1a, c, e show the real speedup of the Pthread and TBB versions for the test functions varying the number of threads. Again the best sequential version is used as reference. TBB performs slightly better than Pthread, although both implementations show near-linear speedup. A speedup slightly below one for a single thread is shown, explicitly quantifying the parallel overhead incurred by the Pthreads and TBB implementations, respectively.

Figure 1b, d, f show the CPU Usage Histogram of the three implementations, that represents the CPU usage in terms of number of active cores during the execution of the algorithm. Bars represent the fraction of the total time during which a given number of cores are simultaneously running. The idle bar represents the fraction of time during which all cores are waiting, i.e., no thread is running. The histogram has been obtained by profiling test problem number 2 (Dixon & Price) using 16 threads (with $pq = 8$ for Bobpp). The profiler indicates that the performance of Bobpp suffers from a high spin time (45%). The executions for $pq = 1$ and $pq = 16$ show an even larger spin time of 82 and 60%, respectively.

Both in Pthread and TBB implementations, the only synchronization point between threads is due to the update of the upper bound (stored in a global variable and protected with a lock). For the used test bed, this update is not a frequent operation. This fact, along with an efficient load balancing policy, results in an almost linear speedup for both implementations. The small difference in speedup between Pthread and TBB can be analysed studying Fig. 1d, f, where it is noticeable that TBB achieves full utilization of the 16 cores. On the contrary,

[1] Computed w.r.t. the sequential Bobpp version, i.e. using the fourth column of Table 1 as the sequential time.

Fig. 1 Speedup and CPU usage histogram for Bobpp, Pthread and TBB implementations. **a** Speedup for Bobpp. **b** CPU usage histogram for Bobpp. **c** Speedup for Pthread. **d** CPU usage histogram for Pthread. **e** Speedup for TBB. **f** CPU usage histogram for TBB

Pthread leaves one, two or even all cores idle for a small fraction of time which may be convenient if cooling or energy saving are taken into account. This behaviour is due to the dynamic thread creation implemented in the Pthread approach.

It is worth mentioning that in TBB, it is more challenging to monitor some statistics of the execution. For instance, the Pthread version efficiently keeps track of the number of evaluated simplices by using per-thread private variables. However, in TBB, tasks are not

tied to a particular thread so per-thread privatization of a variable requires expensive system calls. A straightforward alternative is to store the number of evaluated simplices as a global atomic variable. However, the frequent and concurrent optimization of this single global memory position can kill the scalability of the code, mainly due to cache invalidations, but also due to contention in the access to the atomic variable. Therefore, such statistics have been deactivated in the production version of the TBB implementation in order to collect the data in Fig. 1.

4.3 Programmability of the implementations

Measuring the programmability or *ease of programming* to the user of the investigated B&B implementations is a challenge. For this purpose we follow the methodology proposed in [14], which defines three quantitative metrics: the SLOC (*Source Lines Of Code*), the CC (*Cyclomatic Complexity*) and the PE (*Programming effort*).

When computing the SLOC, comments and empty lines are excluded. This metric probably depends more on the user programming style than the other two metrics. In general, we can assume that higher values for this metric correspond to more error prone and difficult to maintain codes. The CC metric is defined as the number of linearly independent paths through the code. Finally, the PE parameter is defined as a function of the number of unique operands, unique operators, total operands and total operators found in a code. The operands correspond to constants and identifiers, while the symbols or combinations of symbols that affect the value of operands constitute the operators. CC and PE metrics can measure the programming effort required to implement an algorithm: higher values mean that it is more complex for a programmer to code the algorithm.

Table 3 shows the Programmability metrics for Bobpp, TBB and Pthread codes, without considering auxiliary functions that are common to the three implementations (like function evaluation, bounding logic, etc.). The table also provides the ratio of the Pthread metrics versus the TBB metrics and the TBB metrics versus the Bobpp metrics. Although the Pthread implementation has 10% more SLOC and 32% more Cyclomatic Complexity than TBB, the former exhibits 5% less programming effort than the latter. This is because in TBB we have to declare the `task` and `continuation task` classes, which slightly increases the number of operands and operators. On the other hand, the Pthread version requires more conditional instructions in order to manage the threads creation/destruction and synchronization, which translates into the mentioned 32% higher level of CC. Regarding the comparison between TBB and Bobpp, they are pretty similar in terms of SLOC and PE. However, Bobpp is 93% easier in terms of CC. Clearly, the Bobpp framework successfully encapsulates the deep-first search and cut-off logic that can not be totally hidden in TBB and not at all in Pthread. This translates into 1.93× more code-paths in TBB and 2.55× more in Pthreads.

Table 3 Programability metrics for the Bopbb, TBB and Pthread implementations

Implementation	SLOC	CC	PE
Bobpp	327	29	1,474,377.56
TBB	323	56	1,486,205.33
Pthread	357	74	1,409,007.80
Ratio Pthread/TBB	1.10	1.32	0.95
Ratio TBB/Bobpp	0.98	1.93	1.01

5 Conclusion

This paper compares three parallel implementations using different abstraction levels of a Global Optimization Branch-and-Bound algorithm in performance on a test bed of instances and on ease of programming. Features like the selection rule, load balancing method and customizable number of working sets are important. Depth-first search is the strategy which has least memory requirement. This fact leads to less memory management and thus less execution time. The use of a single data structure is not appropriate, because memory contention arises when several threads access a position at the same time. The highest abstraction code based on the Bobpp framework obtains a low speedup for most of the test problems. The lowest abstraction code based on Pthreads uses a load balancing method inherit to dynamic thread creation and obtains a similar performance as the middle abstraction code based on TBB. Using a dynamic number of threads opens the possibility to adapt the parallel level of the application to the current available computational resources during run-time.

To wrap up, we consider TBB an interesting tradeoff between ease of programming and parallel performance. For the evaluated B&B problem, TBB is significantly faster and more scalable than Bobpp with a slight increment in the programming effort. On the other hand, coding this problem in Pthread does not add parallel performance nor ease of programming w.r.t. TBB. We believe, that these results also hold for other similar B&B problems.

The use of a dynamic number of threads in future TBB auto-tuned versions is an appealing approach to be studied as well as the decision when to use more than one thread per queue. Additionally, an interesting future research question is the effect of using a cut-off to limit the parallelism until certain level of the search tree in order to reduce the parallel overhead in the TBB and Pthreaded versions.

Acknowledgements This work has been funded by grants TIN2014-53522-REDT (CAPAP-H5 network) and TIN2015-66680 from the Spanish Ministry, and grants P11-TIC-7176 and P12-TIC-301 from Junta de Andalucía, in part financed by the European Regional Development Fund (ERDF). J.M.G. Salmerón is a fellow of the Spanish FPU program.

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

References

1. Alba, E., Almeida, F., Blesa, M., Cabeza, J., Cotta, C., Daz, M., Dorta, I., Gabarr, J., Len, C., Luna, J., Moreno, L., Pablos, C., Petit, J., Rojas, A., Xhafa, F.: Mallba: a library of skeletons for combinatorial optimisation. In: Monien, B., Feldmann, R. (eds.) Euro-Par 2002 Parallel Processing. Lecture Notes in Computer Science, vol. 2400, pp. 927–932. Springer, Berlin (2002)
2. Amar, P., Baillieul, M., Barth, D., LeCun, B., Quessette, F., Vial, S.: Parallel biological in silico simulation. In: Czachórski, T., Gelenbe, E., Lent, R. (eds.) Information Sciences and Systems 2014: Proceedings of the 29th International Symposium on Computer and Information Sciences, pp. 387–394. Springer, Cham (2014). doi:10.1007/978-3-319-09465-6_40
3. Baritompa, W.: Customizing methods for global optimization, a geometric viewpoint. J. Glob. Optim. **3**(2), 193–212 (1993)
4. Brüngger, A., Marzetta, A., Fukuda, K., Nievergelt, J.: The parallel search bench ZRAM and its applications. Ann. Op. Res. **90**, 45–63 (1999)
5. Casado, L.G., Hendrix, E.M.T., García, I.: Infeasibility spheres for finding robust solutions of blending problems with quadratic constraints. J. Glob. Optim. **39**(4), 577–593 (2007)

6. Casado, L.G., Martínez, J.A., García, I., Hendrix, E.M.T.: Branch-and-Bound interval global optimization on shared memory multiprocessors. Optim. Method Softw. **23**(5), 689–701 (2008)
7. Crainic, T.G., Le Cun, B., Roucairol, C.: Parallel branch-and-bound algorithms. In: Parallel Combinatorial Optimization, pp. 1–28. Wiley (2006). doi:10.1002/9780470053928.ch1
8. Djerrah, A., Le Cun, B., Cung, V.D., Roucairol, C.: Bob++: framework for solving optimization problems with branch-and-bound methods. In: 2006 15th IEEE International Conference on High Performance Distributed Computing, pp. 369–370 (2006). doi:10.1109/HPDC.2006.1652188
9. Eckstein, J., Phillips, C.A., Hart, W.E.: Inherently parallel algorithms in feasibility and optimization and their applications, studies in computational mathematics. In: Dan Butnariu, Y.C. (ed.) Pico: An Object-Oriented Framework for Parallel Branch and Bound, vol. 8, pp. 219–265. Elsevier, Amsterdam (2001)
10. Eckstein, J., Hart, W.E., Phillips, C.A.: PEBBL: an object-oriented framework for scalable parallel Branch and Bound. Math. Program. Comput. **7**(4), 429–469 (2015)
11. Escobar, F.A., Chang, X., Valderrama, C.: Suitability analysis of FPGAs for heterogeneous platforms in HPC. IEEE Trans. Parallel. Distrib. **27**, 600–612 (2016). doi:10.1109/TPDS.2015.2407896
12. Galea, F., Le Cun, B.: Bob++ : a framework for exact combinatorial optimization methods on parallel machines. In: PGCO'2007 as Part of HPCS'07, pp. 779–785 (2007)
13. Gendron, B., Crainic, T.G.: Parallel Branch-and-Bound algorithms: survey and synthesis. Oper. Res. **42**(6), 1042–1066 (1994)
14. González, C.H., Fraguela, B.B.: A generic algorithm template for divide-and-conquer in multicore systems. In: 2010 IEEE 12th International Conference on High Performance Computing and Communications (HPCC), pp. 79–88 (2010). doi:10.1109/HPCC.2010.24
15. Hendrix, E.M.T., Tóth, B.G.: Introduction to Nonlinear and Global Optimization. Springer, New York (2010)
16. Hendrix, E.M.T., Casado, L.G., Amaral, P.: Global Optimization simplex bisection revisited based on considerations by Reiner Horst. In: Murgante, B., et al. (eds.) Computational Science and its Applications ICCSA 2012. Lecture Notes in Computer Science, vol. 7335, pp. 159–173. Springer, Heidelberg (2012)
17. Herrera, J.F.R., Casado, L.G., Hendrix, E.M.T., Paulavičius, R., Žilinskas, J.: Dynamic and hierarchical Load-Balancing techniques applied to parallel branch-and-bound methods. In: 2013 Eighth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing, pp. 497–502 (2013). doi:10.1109/3PGCIC.2013.85
18. Herrera, J.F.R., Casado, L.G., Hendrix, E.M.T., García, I.: Heuristics for longest edge selection in simplicial Branch and Bound. In: Gervasi, O., et al. (eds.) Computational Science and Its Applications—ICCSA 2015, pp. 445–456. Springer, Berlin (2015)
19. Lai, T.H., Sahni, S.: Anomalies in parallel Branch-and-Bound algorithms. Commun. ACM **27**(6), 594–602 (1984)
20. Lawler, E.L., Wood, D.E.: Branch-and-Bound methods: a survey. Oper. Res. **14**(4), 699–719 (1966)
21. Li, G.J., Wah, B.W.: Coping with anomalies in parallel Branch-and-Bound algorithms. IEEE Trans. Comput. **35**(6), 568–573 (1986)
22. Menouer, T., Le Cun, B.: Anticipated dynamic load balancing strategy to parallelize constraint programming search. In: 2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum, pp. 1771–1777 (2013). doi:10.1109/IPDPSW.2013.210
23. Menouer, T., Le Cun, B.: A parallelization mixing or-tools/gecode solvers on top of the Bobpp framework. In: 2013 Eighth international conference on P2P, Parallel, Grid, Cloud and Internet Computing, pp. 242–246 (2013). doi:10.1109/3PGCIC.2013.42
24. Menouer, T., Le Cun, B.: Adaptive N to P portfolio for solving constraint programming problems on top of the parallel Bobpp framework. In: 2014 IEEE International Parallel Distributed Processing Symposium Workshops, pp. 1531–1540 (2014). doi:10.1109/IPDPSW.2014.171
25. Mladineo, R.H.: An algorithm for finding the global maximum of a multimodal multivariate function. Math. Program. **34**, 188–200 (1986)
26. Paulavičius, R., Žilinskas, J.: Simplicial Global Optimization. Springer, New York (2014a)
27. Paulavičius, R., Žilinskas, J.: Simplicial Lipschitz optimization without the Lipschitz constant. J. Glob. Optim. **59**(1), 23–40 (2014b)
28. Paulavičius, R., Žilinskas, J., Grothey, A.: Investigation of selection strategies in Branch and Bound algorithm with simplicial partitions and combination of Lipschitz bounds. Optim. Lett. **4**(2), 173–183 (2010)
29. Paulavičius, R., Žilinskas, J., Grothey, A.: Parallel Branch and Bound for Global Optimization with combination of Lipschitz bounds. Optim. Methods Softw. **26**(3), 487–498 (2011)
30. Poldner, M., Kuchen, H.: Algorithmic skeletons for Branch and Bound. In: Filipe, J., Shishkov, B., Helfert, M. (eds.) Software and Data Technologies, Communications in Computer and Information Science, vol. 10, pp. 204–219. Springer, Berlin (2008)

31. Ralphs, T., Gzelsoy, M.: The symphony callable library for mixed integer programming. In: Golden, B., Raghavan, S., Wasil, E. (eds.) The Next Wave in Computing, Optimization, and Decision Technologies, Operations Research/Computer Science Interfaces Series, vol. 29, pp. 61–76. Springer, Berlin (2005)
32. Reinders, J.: Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism. O'Reilly, Newton (2007)
33. Sakellariou, R., Gurd, J.R.: Compile-time minimisation of load imbalance in loop nests. In: 11th International Conference on Supercomputing, ACM, New York, ICS '97, pp. 277–284 (1997)
34. Saltzman, M.J.: Coin-or: an open-source library for optimization. In: Nielsen, S. (ed.) Programming Languages and Systems in Computational Economics and Finance, Advances in Computational Economics, vol. 18, pp. 3–32. Springer, Berlin (2002)
35. Todd, M.J.: The computation of fixed points and applications. Lecture Notes in Economics and Mathematical Systems, vol. 124. Springer (1976). doi:10.1007/978-3-642-50327-6
36. Tousimojarad, A., Vanderbauwhede, W.: Comparison of three popular parallel programming models on the Intel Xeon Phi. In: Lopes, L., et al. (eds.) Euro-Par 2014: Parallel Processing Workshops. Lecture Notes in Computer Science, vol. 8806, pp. 314–325. Springer, Berlin (2014)
37. Tschoke, S., Polzer, T.: Portable parallel branch-and-bound library user manual, library version 2.0. Tech. rep., University of Paderborn (1996). http://www2.cs.uni-paderborn.de/cs/ag-monien/SOFTWARE/PPBB/documentation.html
38. Žilinskas, J.: Branch and Bound with simplicial partitions for Global Optimization. Math. Modell. Anal. **13**(1), 145–159 (2008)
39. Žilinskas, J.: Parallel Branch and Bound for multidimensional scaling with city-block distances. J. Glob. Optim. **54**(2), 261–274 (2012)
40. Xu, Y., Ralphs, T., Ladnyi, L., Saltzman, M.: Alps: a framework for implementing parallel tree search algorithms. In: Golden, B., Raghavan, S., Wasil, E. (eds.) The Next Wave in Computing, Optimization, and Decision Technologies, Operations Research/Computer Science Interfaces Series, vol. 29, pp. 319–334. Springer, Berlin (2005)