Computer Science • 15 (3) 2014 http://dx.doi.org/10.7494/csci.2014.15.3.239

Bo Tian
Mikhail Posypkin

EFFICIENT IMPLEMENTATION OF BRANCH-AND-BOUND METHOD ON DESKTOP GRIDS

Abstract *The Berkeley Open Infrastructure for Network Computing (BOINC) is an open-source middleware system for volunteer and desktop grid computing. In this paper, we propose BNBTEST, a BOINC version of the distributed branch-and-bound method. The crucial issues of the distributed branch-and-bound method are traversing the search tree and loading the balance. We developed a subtask packaging method and three different subtask distribution strategies to solve these.*

Keywords BOINC, branch-and-bound method, distributed computing, volunteer computing, desktop grid

1. Introduction

Many problems in the areas of operations research, and artificial intelligence can be defined as combinatorial optimization problems. The branch-and-bound method (B&B) is a universal and well known algorithmic technique for solving problems of that type. The root of the tree is the original problem, and the other nodes represent subproblems to be solved. Though the algorithm considerably decreases the computational time required to explore the entire solution space, running time remains unbearable. Using parallel or distributed processing is one of the most popular ways to resolve this issue. The implementation of B&B algorithms on parallel machines was studied in numerous papers [11, 13, 15, 20, 24–26]. All of these solvers are based on parallel computation frameworks that are flexible and only useful for tightly-coupled or shared-memory distributed systems.

Over the last decade, we have observed an emergent growth of new HPC platform volunteer computing grids or desktop grids (DGs) [17]. Unlike conventional parallel computers, this platform has not been sufficiently explored as a target for branch-and-bound methods. DGs are a highly dynamic and heterogeneous distributed computing platform. BOINC [9] is one of the typical DGs platforms, which has been developed by a team based at the Space Sciences Laboratory (SSL) at the University of California. It was originally developed to support the SETI@home [10] project before it became useful as a platform for other distributed applications in areas as diverse as mathematics, medicine, molecular biology, climatology, and astrophysics. BOINC has recently become widely popular, in both theory and practice. Devising an efficient B&B implementation for BOINC is a challenging and practically important problem. The approach proposed in our paper addresses this issue.

We implemented a branch-and-bound solver for the NP-hard 0-1 knapsack problem [8]. The classical knapsack problem is defined as follows: given a set of n items, each item j having an integer profit p_j and an integer weight w_j, one needs to choose a subset of items such that their overall profit is maximized while the overall weight does not exceed the given capacity c. The knapsack problem is stated as the following integer programming model:

$$\begin{aligned} & \max \sum_{i=1}^{n} p_i x_i \\ \text{subject to} \quad & \sum_{i=1}^{n} w_i x_i \leqslant c \\ \text{where} \quad & x_i \in \{0,1\},\ i = 1,2,\ldots n \end{aligned} \tag{1}$$

It is worth noting that our approach is not specific to the knapsack problem, and we will use it to implement other branch-and-bound algorithms. The knapsack problem was chosen as one of the most basic and well-studied optimization problems for illustrative purposes.

This paper is organized in the following manner. In Section II, we review the distributed branch-and-bound approach in more detail as well as a survey of existing work, while Section III describes the BOINC framework. A high level description of BNBTEST is given in Section IV, and details of BNBTEST implementation are provided in Section V. We will show experimental evaluation in Section VI, and finally conclude our work in Section VII.

2. Distributed branch-and-bound

Branch-and-bound [18] is a universal and well-known technique for solving optimization problems. In a nutshell, it interprets the input problem as the root of a search tree. Then, two basic operations are recursively executed: branching the problem (node) into several smaller (hopefully easier) problems, or bounding (pruning) the tree node. The bounding can happen due to two reasons: either the problem has become easy enough to be directly solved or one can prove that this node (and hence, its descendants) cannot contribute to the optimal solution. At any point during the search tree traversal, all subproblems can be processed independently. The only shared resource is the incumbent. Hence, processing the search tree in a distributed fashion is very natural and has been studied for decades.

Since the size and structure of the branch-and-bound tree are not known in advance, the even distribution of computations among processors is a challenging task. Load balancing has been comprehensively studied for tightly-coupled multiprocessors. Most efficient schemes use intensive communication among processors to approach uniform distribution. Unfortunately, this approach is not suitable for volunteer desktop grids where direct communications among computing nodes are normally not allowed. The implementation of branch-and-bound algorithms on the grid was also studied to some extent. The solution for distributed systems consisting of several clusters connected via wide-area networks (WAN) was proposed in [2, 3]. In [23], the branch-and-bound framework was implemented via Ninf-G middleware that provides secure communication over WANs and LANs. The system efficiently utilizes the hierarchical nature of distributed systems: good results were reported for different optimization problems. The work distribution is managed on two levels: at the top level, the work is assigned to master processes, while at the second level, master processes distribute the work among their slaves. The system uses intra-cluster communication implemented via Ninf-G middleware.

Another approach for a computational environment comprising of several supercomputers was studied in [4]. The proposed software called MALLBA is aimed at solving arbitrary global optimization problems by exact heuristic and hybrid methods. To be independent on a particular middleware, MALLBA uses its own set of communication and process management routines. Different optimization algorithms were implemented as different skeletons with a common interface. Such an approach reduces efforts needed to implement new problems. Successful results for some problems were reported.

The BNB-Grid framework proposed in [1] is suitable for utilizing heterogeneous computing resources, supports exact and heuristic search strategies, and runs on distributed systems consisting of different nodes ranging from PCs to large, publicly-available supercomputers. The toolset efficiently copes with difficulties arising in such systems: software diversity, the unreliability of nodes, and different ways of submitting jobs. The distinctive feature of BNB-Grid is the use of different communication packages on different levels: on the top level, we use ICE middleware coupled with TCP/IP sockets, and within a single computing element, either MPI or POSIX Thread libraries are used. Such an approach imposes minimal requirements on the computing element software and efficiently utilizes the communication facilities of each node by using a native communication mechanism.

The software packages mentioned above used proprietary middleware aimed at grids comprising moderate number of powerful computer nodes; e.g., supercomputers. Though these approaches present some useful ideas, they are generally not suitable for desktop grids because the latter is based on standardized middleware, and the number of computing nodes could be very large (thousands and millions of PCs). The approach closest to ours was proposed in [5]. The authors developed a grid-enabled implementation of the branch-and-bound method for computational grids based on Condor [19] middleware. The suggested approach uses a centralized load-balancing strategy: the master keeps a queue of sub-problems and periodically sends them to free-working nodes (slaves). When a sub-problem is sent to the slave, it is either completely solved or the resolution process is stopped after a given number tmax of seconds while unprocessed subproblems are sent back to the master. By adjusting tmax, the systems can control the arrival rate of new sub-problems to the master, preventing memory overflow and avoiding performance degradation due to a bottleneck in the master. Authors reported successful results for several hard quadratic assignment instances.

Condor has been proven to be a good tool for organizing corporate grids which comprise the resources of a department or institution, but it is not aimed at volunteer computing. World-wide volunteer grids differ from corporate grids in that communication latency can be very high due to the fact that all data is stored in file systems, and clients can be connected to the master through slow Internet links and be separated by many intermediate hosts. Unlike Condor, volunteer grids offer directional one-way communication from clients to the master. Clients request new jobs from the master within a specified time, which can be quite long: minutes or even hours. The mentioned observations suggest that data traffic should be kept as small as possible. Thus, we decided to let clients always solve the sub-problems to the end, thus avoiding sending back the remaining unprocessed sub-problems. High latency also implies that the parcels should be large. To fulfill this requirement, our systems packs several sub-problems into one parcel rather than exchanging individual subproblems as in [5]. In the sequel, we evaluate and compare several strategies of aggregating sub-problems to parcels (work-units). Figure 1 shows an example of a distributed search tree in BNBTEST.

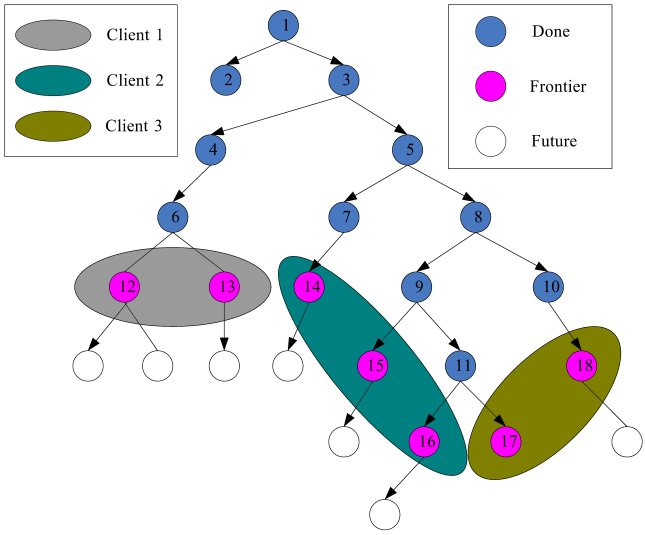

Figure 1. Distributed branch and bound: the frontier is a cut in the search tree separating the completed nodes from the not-yet-explored nodes.

3. BOINC framework

We have built BNBTEST on top of BOINC – a middleware for volunteer grid computing. As almost any distributed software, BNBTEST must cope with the following issues: job distribution, load balancing, parallelism, synchronization, and nondeterminism. The first two points are handled by our system, while the remaining three were implemented by BOINC. The final three are in the scope of this section.

3.1. Parallelism

BOINC supports multi-processor and multi-core executions within the same machine (either standalone or within a cluster). Developers may be able to use OpenCL, MPI, OpenMP, CUDA, languages like Titanium or Cilk, or libraries of multi-threaded numerical "kernels" to develop a multi-threaded app. Also, BOINC supports applications that use co-processors. The supported co-processor types are NVIDIA, AMD, and Intel GPU. As a desktop grid system, BOINC support different kinds of platforms. A platform is a compilation target for BOINC applications – typically, a combination of CPU architecture and an operating system. Each application version is associated with a particular platform. Each project can provide application versions for any set of platforms; the more platforms that are supported, the more hosts that will be able to participate in the project.

BNBTEST just views all multi-cores and multi-processors as individual clients, in order to reduce the complexity of system while increasing efficiency (for example,

if a volunteer computer has 2 processors, each with 4 cores – in BNBTEST, this computer will be seen as $2 * 4 = 8$ cores, each time required 8 workunits from the master server).

3.2. Synchronization

In the BOINC system, there are 'Trickle messages' that allows applications to communicate with the server during the execution of a workunit (job). Messages are XML documents, and they may go from client to server or vice-versa. Trickle messages are asynchronous, ordered, and reliable. Since they are conveyed in scheduler RPC messages, they may not be delivered immediately after being generated, so the communication module is not available in current version of BNBTEST.

3.3. Nondeterminism

Typically, a BOINC server sends 'work unit' to clients, then the clients perform computations and reply to the server. But many things can happen as a result:

- The client computes the result correctly and returns it;
- The client computes the result incorrectly and returns it;
- The client fails to download or upload files;
- The application crashes on the client;
- The client never returns anything because it breaks or stops running BOINC;
- The scheduler isn't able to send the result because it requires more resources than any client has.

In BOINC, there is a validator that decides whether results are correct. We must supply a validator for each application in BNBTEST, and include it in the $\langle daemons \rangle$ section of the configuration file. As we are using BOINC for 'desktop grid' computing (i.e., we trust all the participating hosts), then BOINC supplied a standard validator – "sample_trivial_validator", which requires a strict majority and regards results as equivalent only if they agree byte for byte.

3.4. Limitations of job execution times

The Volunteer Grid (Desktop Grid) requires work sent to volunteer computers to be returned within a set time limit. This is to ensure the overall project batches do not get delayed. At the same time, this facilitates the participation of devices that are on only a few hours per day; e.g., home computers can process a project in the background while performing email messaging, web browsing, and other housekeeping chores. The principle is that every cycle counts and each work unit eventually does get completed. Frequent checkpoints will let these jobs resume very near to where they were shut down the previous time.

BOINC does not run work in a deadline order. Normally, BOINC schedules tasks in the order they were received. Rush jobs will show messages like "Running-High Priority" and others like "Waiting to Run" if paused or preempted during the

general state of Earliest deadline first (EDF), by some referred to as panic state when the order of processing is strictly according to which jobs need to be completed the soonest, irrespective of the project. In BNBTEST, we set Minimum execution times at 2 minutes and Maximum at 30 minutes in order to achieve good performance.

4. BNBTEST overview

BNBTEST uses two ways to traverse the search tree – Depth-first search (DFS) and Breadth-first search (BFS). It is known that a BFS requires more memory space, while a DFS can use more time [12].

As in any other BOINC application, BNBTEST has two parts: a master part (or simply 'master') working on a master server, and a client part that works on clients. Initially, the master reads the problem data and runs the sequential BFS solver locally (in an attempt to generate a large frontier) to provide adequate amounts for the client workstation. The master workstation will stop running the sequential BFS solver until the amount reaches an upper threshold, or the given number of iterations is done. Using a BFS on the master allows us to generate a sufficient amount of sub-problems in a short period of time.

Next, unexplored sub-problems of the frontier are packed in workunits that are sent to clients. The client workstation solves sub-problems in the workunit by a sequential DFS solver. Figure 2 shows the parallelization and distribution strategy across machines and cores. Standard BOINC policy assures that more-powerful machines will get more workunits.

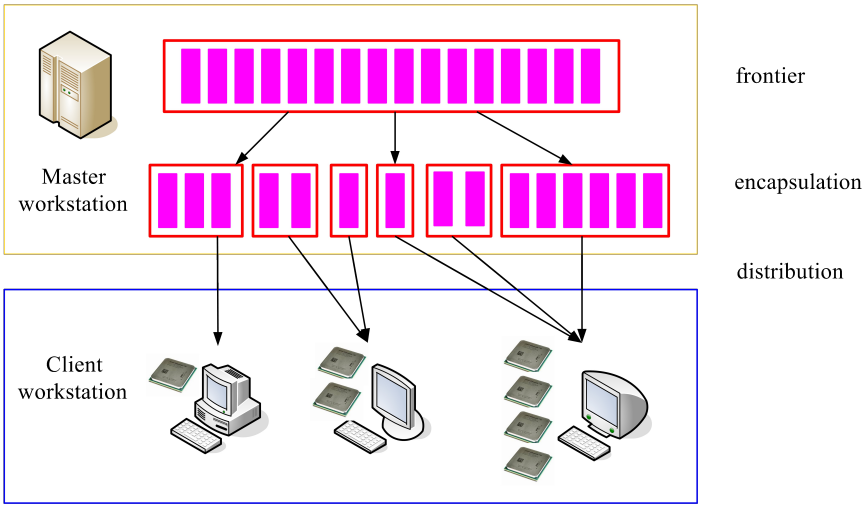

Figure 2. Parallelization and distribution strategy with a similar approach across machines and cores.

5. Implementation

Although the basic structure of the BNBTEST search and distribution algorithm is quite simple, there are several details we have to deal with in order to ensure correctness and obtain good performance.

5.1. Traversing the tree

As noted in the previous section, there are two standard ways of traversing a search tree. BFS processes subproblems that have a lower layer number (closer to the root) first. This allows broad exploration of the search space and may eventually lead to a large (exponential) number of subproblems. In contrast, DFS explores deeper nodes first. This rule ensures that there are at most $\lambda\beta$ open problems at any time, where λ is the maximum height of the tree and β the maximum branching factor (number of children) of a node [6].

A volunteer computing grid (or desktop grid) is a very large grid system, with tens of thousands of computers. To ensure that our computational resources are fully utilized, BNBTEST must generate enough subproblems and workunits to keep all machines occupied. This favors BFS in the master workstation, which tends to generate more problems. However, BFS is very memory-intensive, so the required master server of BNBTEST has more processing power and memory. It stands to reason that a central server should be the one most powerful. So, on the master side; we adopt BFS, and in relative terms, on the client side, we use DFS. Using BFS at the client side may result in a very large frontier and, as a consequence, a memory overflow.

5.2. Load balancing

In essence, the goal of BNBTEST is to traverse a rooted tree in parallel. With k machines, a simple algorithm to achieve this would locally split the initial problem into k subproblems, send the subproblems to different machines, and then wait for them to finish. Because a typical branch-and-bound tree is extremely unbalanced, some machines will complete their subproblems much faster than the others.

Given the restrictive communication model imposed by BOINC, BNBTEST must plan ahead of time to avoid such situations. Considering the variance of the search tree, it is difficult to design a general algorithm to generate subproblems evenly. Hence, in BNBTEST, we designed and implemented three different workunit packaging and distribution strategies. If the number of sub-problems in the frontier is S, the total number of workunits is W; then, each workunit has $\lfloor \frac{S}{W} \rfloor$ or $\lfloor \frac{S}{W} \rfloor + 1$ subproblems.

A. Dense strategy: Dense strategy packages the physically close sub-problems from the search tree into W workunits.

$$w_i = \left[s_{\frac{(i-1)S}{W}+1}, s_{\frac{(i-1)S}{W}+2}, \ldots, s_{\frac{iS}{W}} \right] \quad \text{where} \quad i = 1, 2, \ldots, W \tag{2}$$

B. Sparse strategy: Sparse strategy equidistantly picks subproblems then packages them into W workunits.

$$w_i = \left[s_{\frac{(i-1)S}{W}+i}, s_{\frac{iS}{W}+i}, \ldots, s_{\frac{(i-2+\frac{S}{W})S}{W}+i} \right] \quad \text{where} \quad i = 1, 2, \ldots, W \tag{3}$$

C. Random strategy: Random strategy we use Fisher-Yates shuffle [14] to generate a random permutation of a finite set, in plain terms, for randomly shuffling the subproblems. Then package them into W workunits. The pseudo code of random strategy was shown in Algorithm 1.

Algorithm 1: Random strategy

```
Input: S: Number of subproblems
       W: Number of workunits
       L[S]: list of subproblems
Output: L[w]: Output W lists
for i ← S − 1 to 1 do
    j ← random integer with 0 ⩽ j ⩽ i
    swap L[j] and L[i];
S_w ← S/W
S_wl ← S mod W
order ← 0
while order ⩽ S_w l do
    for i ← 0 to S_w + 1 do
        push L_s[i] into L_w[order]
    order++
while S_wl < order < S_w do
    for i ← 0 to S_w do
        push L_s[i] into L_w[order]
    order++
return order
```

6. Experimental evaluation

As a high performance distributed computing platform, BOINC has about 596,224 active computers (hosts) worldwide processing 9.2 petaFLOPS on average as of March, 2013. The BOINC framework is supported by various operating systems, including Microsoft Windows, Mac OS X, Android, GNU/Linux, and FreeBSD. Hence, BOINC has been proven to be stable and robust (we don't need to verify these in the very first step). Also, because this paper is our preliminary experimentation of implement branch and bound on DGs, we are more concerned about system implementation,

load balancing between workunits, and the validity of the results. The robustness of our system will be verified in future work.

We tested BNBTEST on a small cluster of 15 computers, each with 2–4 GB of RAM and 2-8 core processors running different operation systems (primarily GNU/Linux and Microsoft Windows series).

As already mentioned, our example application is the 0-1 knapsack problem. We focus our experiments on Circle instances circle $\left(\frac{2}{3}\right)$ [14]. The instances are generated such that the profits as function of the weights form an arc of a circle (an ellipsis, actually). The weights are uniformly distributed in $[1, R]$, and for each weight w, the corresponding profit is chosen as $p = \frac{2}{3}\sqrt{4R^2 - (w - 2R)^2}$, here we set $R = 200$. These instances are commonly used for benchmarking sequential solvers [8, 21, 22]. The performance of our solver is competitive with a similar implementation based on Condor Grid, which was reported in [16]. The experiment is aimed at demonstrating the effect of three different strategies and load balancing between workunits. We measured makespan (M_{span}) defined as the time elapsed between the start of the first task of the job and the finish of its last task, i.e.

$$M_{span} = T_{stop} - T_{star} \tag{4}$$

Also, we defined the speedup (S_p) as a ratio between the total amount of (useful) CPU time consumed by the application and the M_{span}. The total useful time T_u for the set of workunits (U) and the speedup S_p are defined as follows:

$$T_u = \sum_{i \in U} T(i) \tag{5}$$

$$S_p = \frac{T_u}{M_{span}} \tag{6}$$

For this experiment, all data is obtained for $cir200$, a circle $\left(\frac{2}{3}\right)$ instance with 200 items. The grain size of the workunit is controlled by the maximum amount of subproblems per workunit (MSW). We choose MSW by $5, 10 and 20$ as different test cases. The maximum number of subproblems generated by the master is limited to 1000. Table 1 shows the makespan (M_{span}), the total useful time (T_u), the speedup (S_p) and execution time in master server (T_m) for each trial in the experiment. By default, all time is shown in seconds.

Table 1

Performance of BNBTEST with 1000 subproblems on cir200.

MSW	Strategies											
	Dense strategy				Sparse strategy				Random strategy			
	M_{span}	T_u	S_p	T_m	M_{span}	T_u	S_p	T_m	M_{span}	T_u	S_p	T_m
5	201	3041	15.131	0.56	172	2761	16.051	2.31	133	2410	18.117	5.5
10	178	2046	14.302	0.43	170	2667	15.69	2.05	148	2223	15.022	5.01
20	191	2495	13.066	0.41	189	2996	15.057	1.92	134	2042	15.238	4.83

We can draw some meaningful conclusions from the table:

- The parallel performance decreases with the increase of grain size when we fixed the total number of subproblems (for all three strategies);
- In most case, random strategy gives a better performance. And the smaller of gain size, the better parallel performance of random strategy. Meanwhile, dense strategy parallel performs always the worst.
- In master workstation, random strategy performs better because of the lower algorithm complexity. Random strategy has the worst T_m since the time cost from shuffle algorithm.

As mentioned in the previous section, a typical branch-and-bound tree is extremely unbalanced, so workunit load-balance is a key indicator. Figure 3 compares the task execution time of three different strategies in the same instance. Obviously, we can see from the figure 3 that the random strategy has the best load-balance, while the dense strategy gives us the worst load balancing.

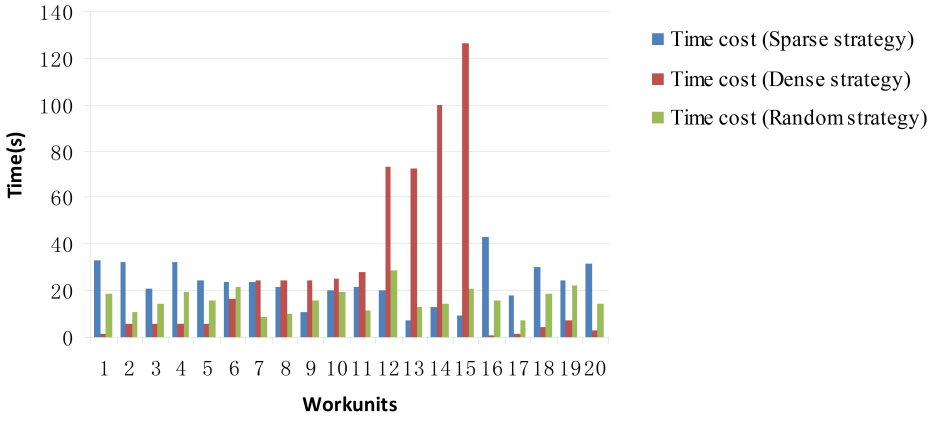

Figure 3. Load-balancing of three strategies.

After the comparison of these three strategies, the random strategy gives us the highest performance in the client workstation but the worst performance in the master workstation. Dense strategy performs better in the master workstation, but always worst in the client workstation; the sparse strategy always has intermediate-level performances in both client and master workstations.

While the size of instance increases, task execution time in the master workstation increases linearly, but it cannot be compared to the increase in the client workstation. For a real large-scale computation, our demonstration is made on an instance cir250, a circle $\left(\frac{2}{3}\right)$ knapsack instance of size 250. Table 2 shows the performance on cir250. The instance well proves this point. The execution time in the master workstation increased by a few seconds, but in the client workstation, it increased 9 times. Based on these different scales of test instances, we recommend the random strategy in most cases.

Table 2

Performance of BNBTEST with 1000 subproblems on cir250.

MSW	Strategies											
	Dense strategy				Sparse strategy				Random strategy			
	M_{span}	T_u	S_p	T_m	M_{span}	T_u	S_p	T_m	M_{span}	T_u	S_p	T_m
5	1821	27991	15.37	2.13	1636	26644	16.277	2.60	1473	27090	18.39	7.21

7. Conclusion

We have introduced BNBTEST, a framework for implementing the branch-and-bound method on a desktop grid system (BOINC), which includes the following features: multi-core parallelization, traversing, delivering the search tree, and load balancing of the workunits. BNBTEST has been proven as a good distributed branch and bound solver. Future work will focus on more-efficient packaging and a distribution strategy, and make the BNBTEST to be a modular middleware, with the user interface for users. Also, we plan to increase our volunteer grid for solve more complex and practical optimization problems.

References

[1] Afanasiev A., Evtushenko Y., Posypkin M.: The Layered Software Infrastructure for Solving Large-scale Optimization Problems on the Grid. *Horizons in Computer Science Research*, vol. 4, pp. 129–144, 2012.

[2] Aida K., Futaka Y., Osumi T.: Parallel Branch and Bound Algorithm with the Hierarchical Master-Worker Paradigm on the Grid. *IPSJ Digital Courier*, vol. 2, pp. 584–597, 2006.

[3] Aida K., Natsume W., Futakata Y.: Distributed computing with hierarchical master-worker paradigm for parallel branch and bound algorithm. In: *Proc. 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2003)*, pp. 156–163, 2003.

[4] Alba E., Almeida F., Blesa M., Cabeza J., Cotta C., Díaz M., Dorta I., Gabarró J., León C., Luna J., Moreno L., Pablos C., Petit J., Rojas A., Xhafa F.: MALLBA: A Library of Skeletons for Combinatorial Optimisation (Research Note). In: *Proceedings of the 8th international Euro-Par Conference on Parallel Processing*, pp. 927–932, 2002.

[5] Anstreicher K., Brixius N., Goux J. P., Linderoth J.: Solving large quadratic assignment problems on computational grids. *Mathematical Programming*, vol. 91, pp. 563–588, 2002.

[6] Budiu M., Delling D., Werneck R. F.: DryadOpt: Branch-and-Bound on Distributed Data-Parallel Execution Engines. In: *IPDPS*, pp. 1278–1289, 2011.

[7] Crainic T. G., Cun B. L., Roucairol C.: *Parallel Combinatorial Optimization.* John Wiley & Sons, Inc., 2006.

[8] David P.: Where are the hard knapsack problems? *Computers & Operations Research*, vol. 32, pp. 2271–2284, 2005.

[9] David P. A.: BOINC: a system for public-resource computing and storage. In: *Proceedings of the 5th IEEE/ACM International GRID Workshop*, 2004.

[10] David P. A., Cobb J., Korpela E., Lebofsky M., Werthimer D.: SETI@home: an experiment in public-resource computing. *Communications of the ACM*, vol. 45, pp. 56–61, 2002.

[11] Djerrah A., Cun L. B., Cung V. D., Roucairol C.: Bob++: Framework for solving optimization problems with branch-and-bound methods. In: *Proceedings of the 15th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, pp. 369–370, 2006.

[12] Knuth D. E.: *The Art of Computer Programming.* Addison-Wesley, Boston, 1968.

[13] Drummond L. M. A., Uchoa E., Goncalves A. D., Silva J. M. N., Santos M. C. P., Castro M. C. S.: A grid-enabled distributed branch-andbound algorithm with application on the Steiner problem in graphs. *Parallel Computing*, vol. 32, pp. 629–642, 2006.

[14] Durstenfeld R.: Algorithm 235: random permutation. *Communications of the ACM*, vol. 7, p. 420, 1964.

[15] Eckstein J., Phillips C. A., Hart W. E.: *PICO: An object-oriented framework for parallel branch and bound. Studies in Computational Mathematics*, vol. 8, pp. 219–265, 2001.

[16] Glankwamdee W., Linderoth J.: *Parallel Combinatorial Optimization.* John Wiley & Sons, Inc., 2006.

[17] Kacsuk P., Kovacs J., Farkas Z., Marosi A. C., Balaton Z.: Towards a Powerful European DCI Based on Desktop Grids. *J Grid Computing*, vol. 9, pp. 219–239, 2011.

[18] Land A. H., Doig A. G.: An automatic method of solving discrete programming problems. *Econometrica*, vol. 28, pp. 497–520, 1960.

[19] Litzkow M. J., Livny M., Mutka M. W.: Condor-a hunter of idle workstations. *8th International Conference on Distributed Computing Systems*, pp. 104–111, 1988.

[20] Lüling R., Monien B.: Load balancing for distributed branch and bound algorithms. In: *Parallel Processing Symposium, 1992. Proceedings., Sixth International*, pp. 543–548, 1992.

[21] Martello S., Toth P.: A new algorithm for the 0-1 knapsack problem. *Management Science*, vol. 34, pp. 633–644, 1988.

[22] Martello S., Toth P.: Upper bounds and algorithms for hard 0-1 knapsack problems. *Operations Research*, vol. 45, pp. 768–778, 1997.

[23] Nakada H., Tanaka Y., Matsuoka S., Sekiguchi S.: *Grid Computing: Making the Global Infrastructure a Reality.* John Wiley & Sons Ltd., 2003.

[24] Quinn M.: Analysis and implementation of branch-and-bound algorithms on a hypercube multicomputer. *IEEE Transactions on Computers*, vol. 39, pp. 384–387, 1990.

[25] Shinano Y., Higaki M., Hirabayashi R.: A generalized utility for parallel branch and bound algorithms. In: *IEEE Symposium on Parallel and Distributed Processing*, p. 392, 1995.

[26] Tschöke S., Lüling R., Monien B.: *Solving the traveling salesman problem with a distributed branch-and-bound algorithm on a 1024 processor network. in Proceedings of the 9th Parallel Processing Symposium*, pp. 182–189, 1995.

Affiliations

Bo Tian
Lomonosov Moscow State University, e-mail: yesyestian@gmail.com

Mikhail Posypkin
Lomonosov Moscow State University, e-mail: mposypkin@gmail.com

Received: 26.11.2013
Revised: 5.02.2014
Accepted: 11.02.2014