Using simulation for performance analysis and visualization of parallel Branch-and-Bound methods *

Y.G. Evtushenko[1], I.V. Golubeva[1], Y.V. Orlov[1], M.A. Posypkin[1]

Dorodnicyn Computing Centre, FRC CSC RAS[1]

The Branch-and-Bound (B&B) is a fundamental algorithmic scheme for a large variety of global optimization methods. For many problems B&B requires the amount of computing resources far beyond the power of a single-CPU workstation thus making parallelization almost inevitable. The approach proposed in this paper allows one to evaluate load balancing algorithms for parallel B&B with various numbers of processors, sizes of the search tree, the characteristics of the supercomputer's interconnect. The proposed approach was implemented as a special tool that simulates the process of resolution of the optimization problem by B&B method as a stochastic tree branching process. Data exchanges are modeled using the concept of logical time. The user-friendly graphical interface can render both real traces and ones produced by the simulator. It provides efficient visualization of the CPU's load, data exchanges and progress of the optimization process.

Keywords: performance analysis and simulation, parallel computing, global optimization, branch-and-bound methods, load balancing.

1. Introduction

The Branch-and-Bound method (B&B) is one of the main approaches to the resolution of mathematical programming problems [1,2]. In contrast to heuristic and stochastic methods, B&B ensures the accuracy of the found solutions and, in some cases, can solve the problem exactly. For realistic problems B&B can consume computational and time resources, significantly exceeding the available capacity. Parallel computing can be used to speedup and reduce the memory requirements for B&B implementation. Balancing computational load between processors plays an important role in the parallel implementation of global optimization methods[3-5]. Typically load balancing means transmission of jobs from one processor (core) to another along the computations.

Today most powerful supercomputers contain 10^6 computational cores and this number continues to grow thus making load balancing a very challenging problem. There is a clear demand for deep and systematic study and comparison of various load distribution strategies. Performing such evaluation on a real multiprocessor computing system requires multiple runs on the very expensive equipment. We propose to use the simulator for these purposes. The simulator allows one to study performance of load balancing algorithms with various numbers of processors, sizes of the search tree, the characteristics of the supercomputer's interconnect. The process of resolution of the optimization problem by B&B method is replaced by a stochastic branching process. Data exchange and computations are modeled using the concept of logical time.

Another important problem is an adequate visualization of the algorithm performance. To address this issue we developed a user friendly graphical interface that enables convenient performance analysis through processor load charts communication tables and aggregate statistics. The developed tool could serve as a good problem specific addition to other performance analysis and supercomputer's monitoring software [6-8].

* Supported by Ministry of Science and Education of Republic of Kazakhstan, project 0115PK00554, Russian Fund for Basic Research, project16-07-00458 A, Leading Scientific Schools project NSH-8860.2016.1, Project I.33 of RAS

2. Distributed memory Branch-and-Bound implementation

The goal of global optimization (GO) is to find an extreme (minimal or maximal) value $f^* = f(x^*)$ of an objective function $f(x)$ on a feasible domain $X \subseteq R^n$. The value f^* and feasible point $x^* \in X$ are called optimum and optimal solution respectively. Without loss of generality one can consider only minimization problems: $f(x) \to \min,\ x \in X$.

The Branch-and-Bound (B&B) is a general name for methods to split an initial problem into subproblems which are sooner or later eliminated by bounding rules. Bounding rules determine whether a subproblem can yield a solution better than the best solution found so far. The latter is called *the incumbent solution*. Bounding is often done by comparing lower and upper bounds: a subproblem can be pruned if the lower bound for its objective is larger or equal to the current upper bound, i.e. incumbent solution.

Numerous Branch-and-Bound algorithms were developed for different global optimization problems. Some of them were very successful for particular problem kinds, e.g. Travelling Salesman or Knapsack problems. However for many problems Branch-and-Bound methods require the amount of computing resources beyond the power of a single-CPU workstation. Fortunately Branch-and-Bound is highly suitable for parallel and distributed computing: after splitting the parts of the solution space can be processed independently and simultaneously.

Another great advantage of B&B methods is that the general scheme does not significantly vary from one problem to another. The splitting and bounding rules may differ while keeping the general scheme almost intact. The direct consequence of this is the possibility to separate problem-independent and problem-specific parts. Such separation saves a lot of efforts when implementing a new problem or a new method. This is especially true for tools targeted at parallel and distributed environments because the "parallel" part is reused for different optimization problems. We follow this approach in our tools: the computing space management, the work-distribution and communication among application processes is problem-independent.

Our parallel library for global optimization BnB-Solver [9] is built on top of MPI [10] which implies that parallel processes communicate via message-passing. Each process do three basic kinds of activity: performing steps of B&B method, sending data and receiving data. Transmitted data consists of sub-problems and/or incumbent solutions and commands. Exchanging sub-problems performs computations redistribution among processes in order to make the load more or less even. Sending incumbents ensures fast error propagation among parallel processes.

According to the aforementioned concepts managing the resolution process including data exchanges can be encapsulated in a special component called the *scheduler*. The problem-specific part is managed by another component – the *solver* that provides methods to solve the problem, read its state (the number of subproblems in a queue) fetch and extract subproblems. Sending and receiving of subproblems is implemented by the *communicator* component. These parts are composed together by a special bridge class that invokes respective methods of the scheduler, the solver or communicator (Figure 1). The proposed approach separates the managing part from implementation details part thereby providing an opportunity for an independent schedulers testing and verification.

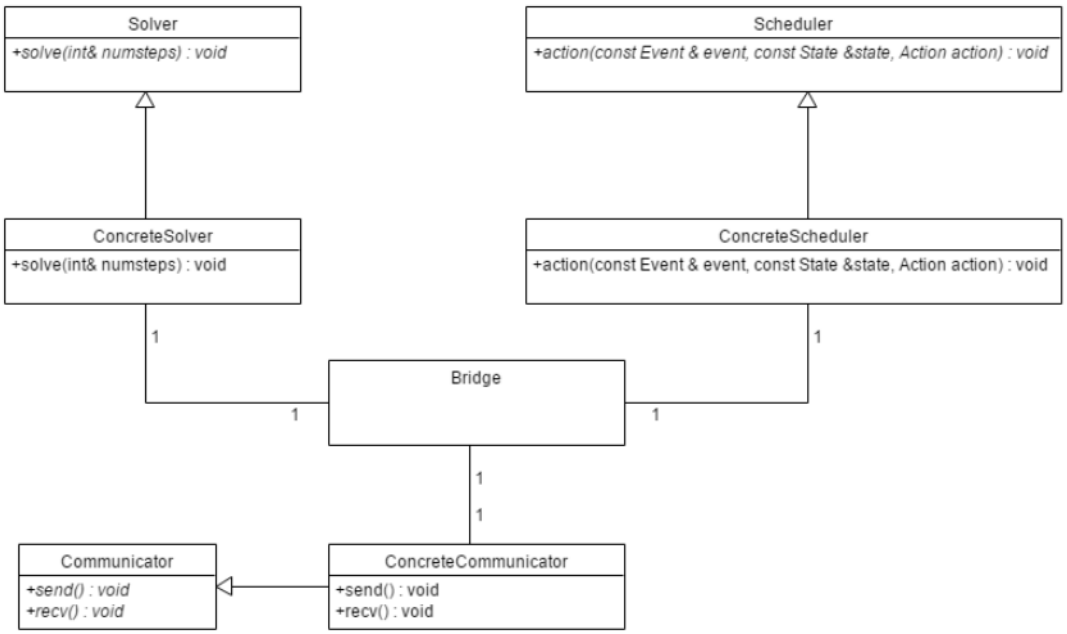

Fig 1. Class diagram of fundamental interactions

The scheduler is a finite state machine that accepts *events* and issues *actions*. Possible events and actions are listed in Tables 1, 2 respectively. The bridge invokes method `action()` of the scheduler class that accepts an event and the solver state as input parameters and generates an action on output. Then the bridge invokes the methods associated with the action of the solver or communicator.

Table 1. Event types

Event type	Arguments	Description
ERROR	Error code	An error occurred.
START		The beginning of computations.
DONE	The real number of steps done	The requested number of steps done.
SENT	The number of transmitted items	The requested sending message action done.
DATA_ARRIVED	The process that sent the data	The receive command finished and the requested data received.
COMMAND_ARRIVED	The process that sent the data	The command arrived.

Table 2. Action types

Action type	Arguments	Description
SOLVE	Number of steps	Perform given number of B&B steps.
EXIT		Terminate the process.
SEND_COMMAND	The receiver process number, command number and arguments	Send the given command to the specified process.
SEND_SUBS_AND_RECORDS	The receiver process number, number of subproblems to transmit.	Send the specified number of commands and the incumbent solution to the specified process.

RECV	The id of process the data is waited from.	Issues the receive command and waits for the message.

The scheduler can trace all transitions from one state to another, actions, events and their arguments. If logging is enabled the traces of all processes are collected, merged and written to file system. Then these traces can be processed and visualized by GUI described below.

3. Simulation of parallel B&B

The simulator was designed for convenient fast and efficient performance testing of parallel schedulers. The simulator uses the real scheduler which is taken intact from the library and provides 'fake' implementations of the solver and the communicator. This approach enables the rapid testing of the schedulers on large trees and thousands of processors because the time consuming resolution steps and communications are substituted by formal actions which take nearly zero time.

The parallel processing is simulated serially. For each simulated process the instance of the scheduler is created. The simulator cyclically iterates through these instances and invokes `action()` methods. If the action is `SOLVE` then the specified number of steps is simulated and the logical clock is increased according to the modelled time. The B&B method is substituted by a random branching process where the node generates two new nodes with a probability decreasing with distance between the tree root and the node. When the node reaches the maximal tree depth the probability becomes zero. Thus the maximal tree depth controls the size of the whole tree. The time of solving is modelled using the simple formula $t = nt_s$ where n is the number of performed steps and t_s is the time of one step.

The data transmission is simulated using the concept of logical clock [11]. When the `SEND_SUBS_AND_RECORDS` command is issued the communicator object stores the message and its timestamp obtained by increasing the current time on a process by the modelled time of a message transmission. The time required to transmit the message is computed by the following formula:

$$t = St_p + L + S/B,$$

where S is the size of the message, t_p is time needed for packing a unit of data at a sender process, L is the network latency, defined as the time needed to transfer the minimal amount of data throughout network and B is the bandwidth – the amount of data transmitted through the network in a unit of time.

When the `RECV` command is issued by a scheduler the recipient process the communicator looks up for available messages for this process and if one is encountered it compares the logical time on a recipient t_R with the message time stamp t_S. The logical time on a recipient is adjusted to the maximum of these values and the obtained value is increased by time required to unpack the message:

$$t_R = \max(t_R, t_S) + St_u,$$

where S is the size of the message, t_u is time needed for unpacking a unit of data on the recipient.

During the simulation all events and actions are logged. The log files contain all information about logical time of various simulated events. This information is used by graphical user front-end described in the next section.

4. Graphical front-end

The log files are not suitable for direct analysis by a human. The graphical front-end is aimed at user-friendly graphical visualization and performance analysis of traces produced by either simulator or the real solver. Based on the collected traces the GUI performs the following activities:
– visualizes processors' loads;
– visualizes data exchange among processors;
– computes aggregated performance information such as speedup and efficiency.

Figure 2 shows the window demonstrating processor load plots for individual processors. At the bottom of the window there is a slider similar to one used in multimedia players. It allows an easy and natural navigation throughout the trace. Such representation is convenient for a moderate number of processors. However for hundreds and thousands of processors it can be very inefficient. For such cases BNB-Visualizer provides the processor grid (Figure 3) which scales well. Blue color is used for depicting computations, red color marks processors blocked in the receiving state. Green color means the processor is sending data.

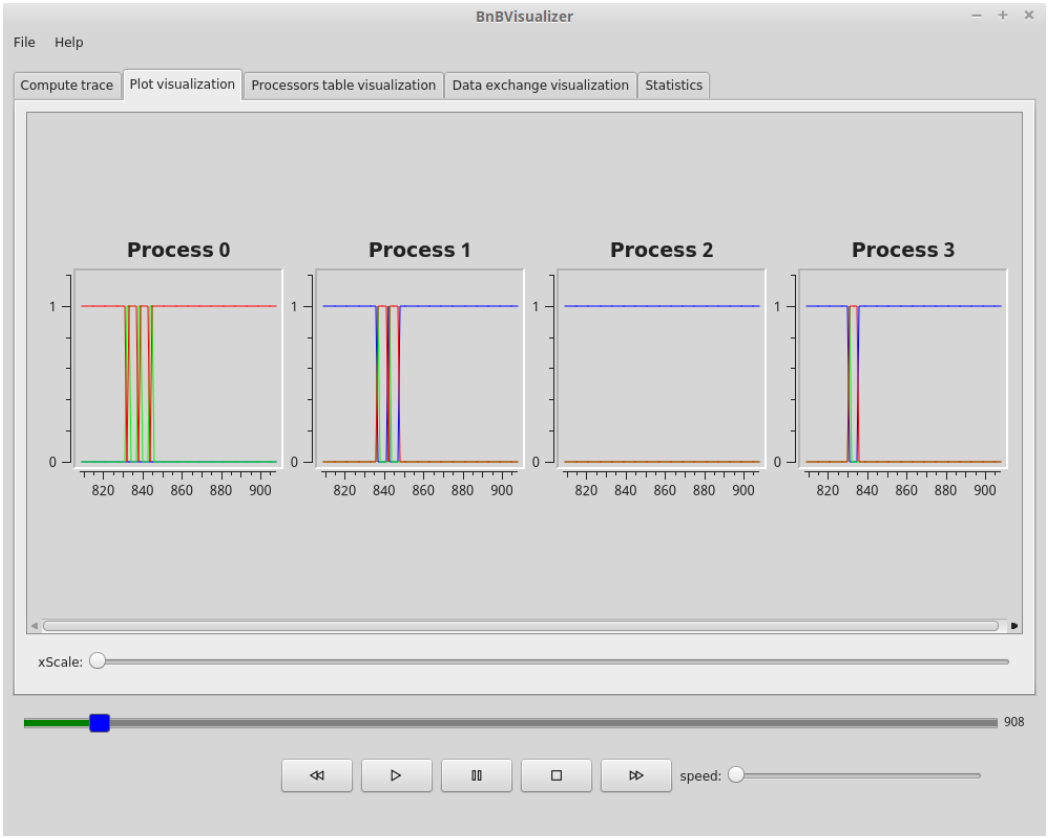

Fig 2. Processors' load plots

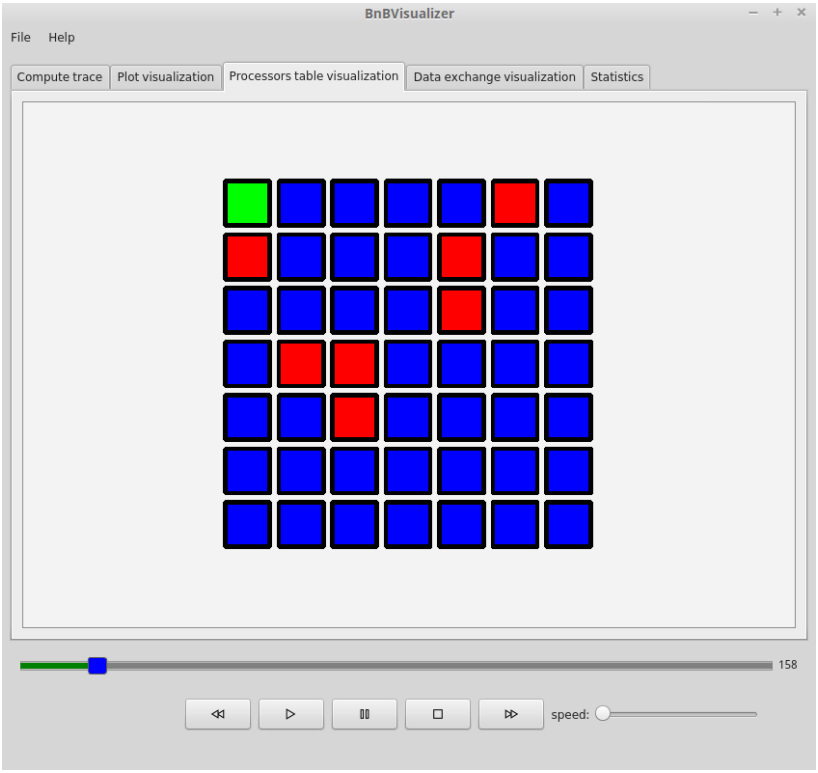

Fig 3. Processors' grid

Communications are visualized using two-dimensional chart where processors are aligned along horizontal (senders) and vertical (receivers) axes. The receive actions are visualized by a horizontal blue line and the send action is represented by a vertical green line (Fig. 4). At the Figure lines (1) and (2) correspond to a successful message transmission from the process 9 to the process 0. Line (3) depicts the unsatisfied send issued by the process 0.

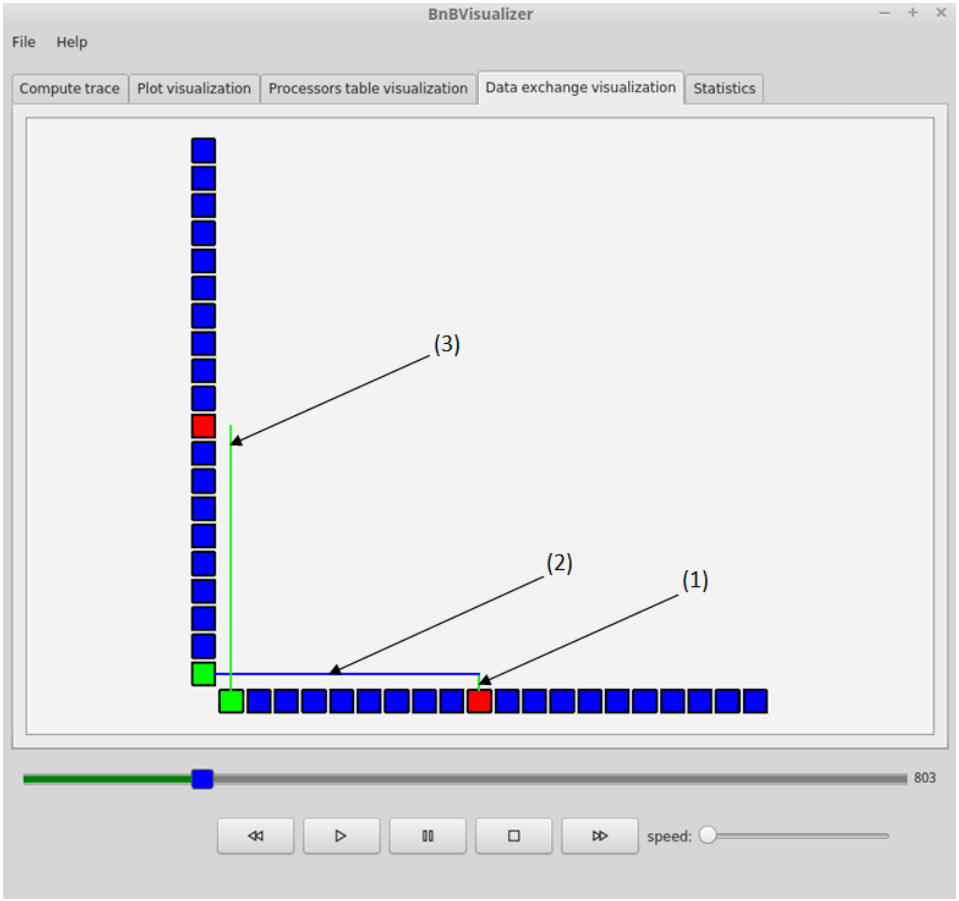

Fig 4. Communications visualization

The cumulative information about the processors' usage and performance metrics is shown in a separate tab (Figure 5). This performance chart shows the number of processors occupied at the given moment of time (blue color) and the number of free processors (green color).

5. Experiments

5.1 Case study I: selecting best parameters for adaptive load balancing

The simulator was used to study the comparative performances of a family of load balancing algorithms working as follows. At the initial phase the 1st (*master)* processor generates some number of sub-problems. At the second stage each of remaining processors (*slaves*) gets a sub-problem from the master and starts its resolution. The solution process on a slave is interrupted each *T* iterations and then the slave sends *S* sub-problems or less to the master. If there are remaining sub-problems on a slave it resumes B&B method. The master processor stops receiving sub-problems from slaves when the number of sub-problems in its pool exceeds *M* and resumes receiving when it drops below *m.* This is done by setting parameter *S* to 0 or to its original value.

Figure 5 shows the performance chart for small values of *T*. The very intensive data exchange among parallel processes doesn't yield good performance because of large communication expenses.

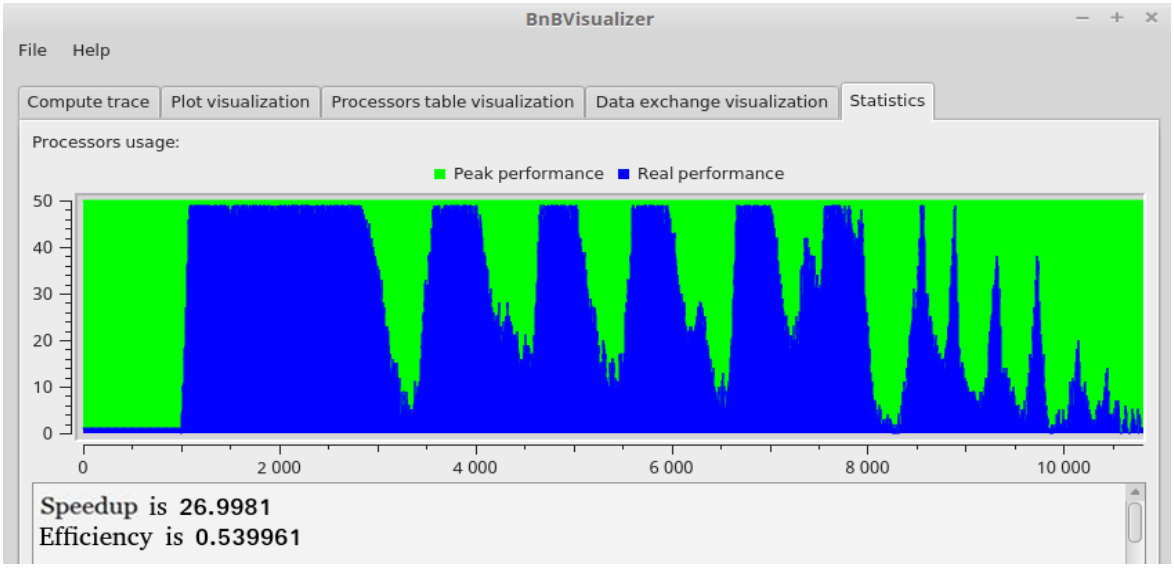

Fig 5. The performance for small values of T

For moderate values of T the performance is better but we can see significant performance losses at the final stage of the algorithm (Fig. 6). In the middle of the computational process the load balance is good but at the terminal stage it is quite bad.

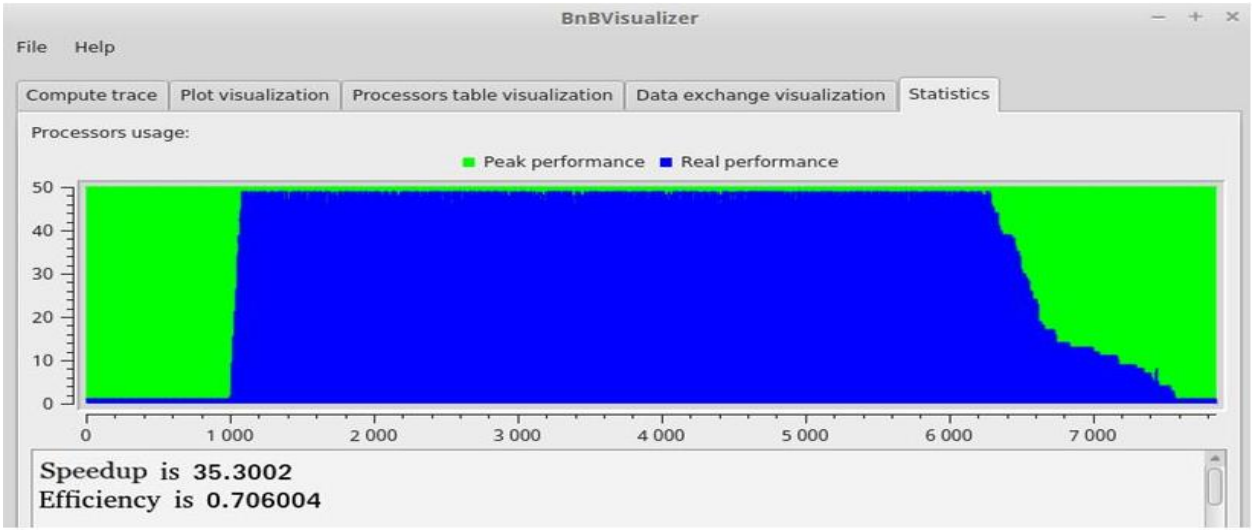

Fig. 6. The performance for moderate values of T

The natural solution to avoid such performance losses is to introduce dynamic adaptation: when the number of subproblems on the master drops below the number of free processors the parameter T is decreased in 10 times. Thus at the middle of computations when the deman for load redistribution is small T is kept relatively large. At the final stage T dicreases in order to provide good load balancing among process throught intesive excahnge of subproblems. This leads to a better performance (Fig. 7).

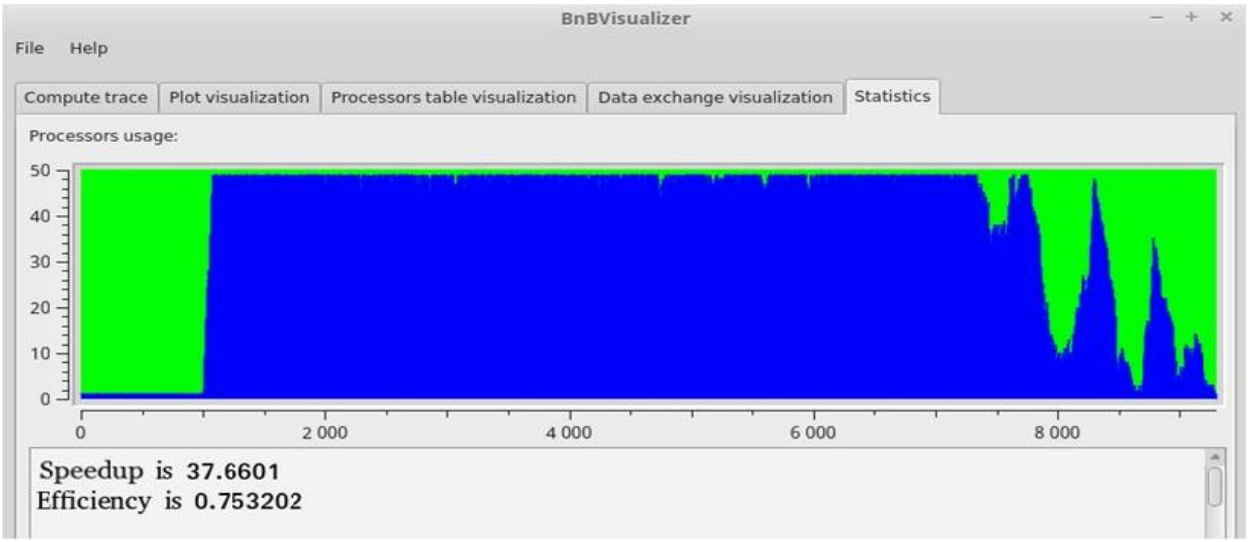

Fig. 7. The performance for dynamic adaptation of *T*

5.2 Case study II: studying performance of parallel frontal algorithm

In the second case study we simulated the simplest possible load balancing scheme – *frontal branching*. In this approach the master performs T B&B steps thereby producing a number of subproblems. Then each sub-problem is sent to the respective slave and is solved completely. The results are collected and the best found solution is selected and supplied to the user. The number of available cores is supposed to be larger than the number of subproblems generated by the master.

Theoretical studies [12,13] for a particular case of B&B method suggest that the speedup of frontal branching is a unimodal function of the threshold value T. We used simulator to check whether this is the general behavior and assess the influence of the network latency. The simulator was run in batch mode on random trees with maximal depth varied from 30 to 50 and with different values of T from 100 to 1000. The results showed that though the behavior is not necessary strictly unimodal the trend is obvious: there is a value of T where the speedup reaches its maximum and then starts to decrease.

Figure 8 shows the plot of the speedup as a function of T for a random tree of depth 40. Two graphs show the speedup as a function of T for zero latency (red) and non-zero latency (blue). We observe quasi-unimodal behavior for both cases. As expected the speedup for non-zero latency is less than for zero latency case.

8. Conclusions

The paper discussed the simulator of parallel Branch-and-Bound method that can be used for a deep study and comparison of load balancing algorithms. Though the simulation can't completely replace the testing on a real multiprocessor it can significantly reduce the number of expensive runs on a supercomputer. Since the traces produced by the simulator follow the same format as the parallel solver the graphical front-end supports performance visualization for both the simulator and the optimization library. The simulator can run in batch mode to perform large-scale simulation for comprehensive performance analysis, e.g. produce scalability charts [14].

In the future we are going to implement more sophisticated hierarchical interconnect models in our tool and perform a comprehensive analysis and comparison of various load balancing algorithms.

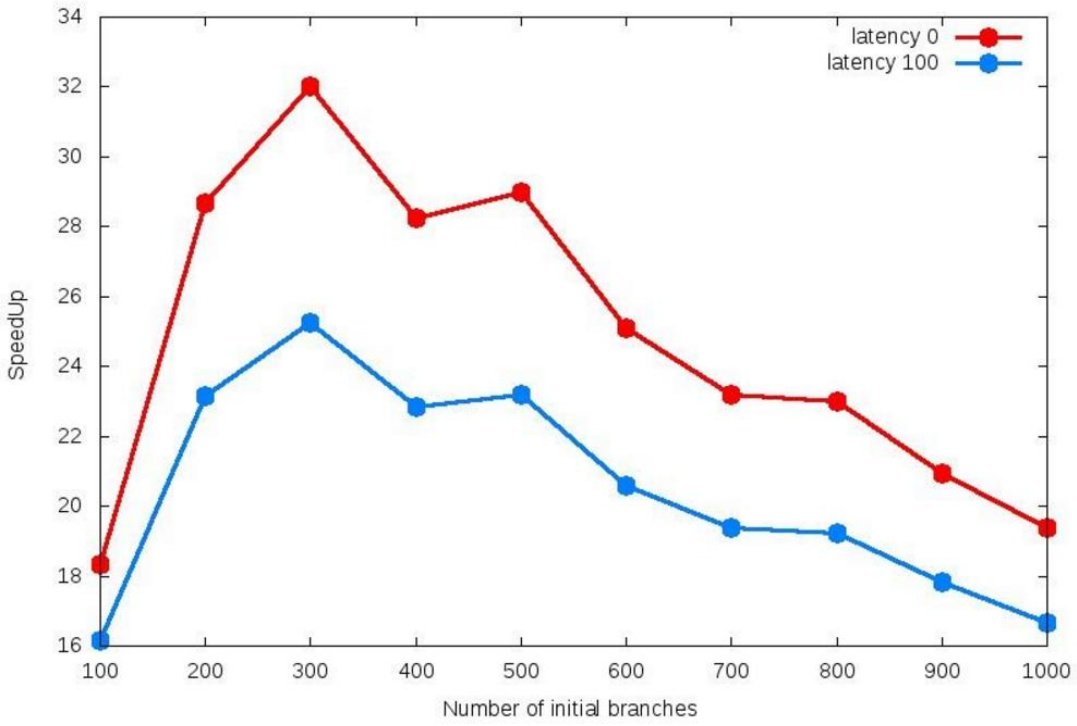

Fig. 8. The speedup as a function of T

References

1. Pardalos P.M., Romeijn E., Tuy H. Recent developments and trends in global optimization // Journal of Computational and Applied Mathematics. 2000. Vol. 124. No. 1-2. P. 209–228.
2. Scholz D. Deterministic global optimization: geometric branch-and-bound methods and their applications. Springer Science & Business Media, 2011. 153 p.
3. Gendron B., Crainic T.G., Parallel Branch-and-Bound Algorithms: Survey and Synthesis // Operations Research. 1994. Vol. 42. No. 6. P. 1042–1066.
4. Lüling R., Monien B., Load balancing for distributed branch & bound algorithms// Proceedings of Sixth International Parallel Processing Symposium. IEEE, 1992. P. 543–548.
5. Barkalov K., Gergel V., Lebedev I. Use of Xeon Phi Coprocessor for Solving Global Optimization Problems // Proceedings of Parallel Computing Technologies. Springer International Publishing, 2015. P. 307–318.
6. Ganglia Monitoring System. URL: http://ganglia.sourceforge.net/ (accessed 12.06.2016)
7. Nagios-the industry standard in IT infrastructure monitoring. URL: https://www.nagios.org (accessed 12.06.2016)
8. Stefanov, K., Voevodin, V., Zhumatiy, S., Voevodin, V. Dynamically Reconfigurable Distributed Modular Monitoring System for Supercomputers (DiMMon) //Procedia Computer Science. 2015. Vol. 66. P. 625–634.
9. Evtushenko Y., Posypkin M., Sigal I. A framework for parallel large-scale global optimization //Computer Science-Research and Development. 2009. Vol. 23, No. 3–4. P. 211–215.
10. Snir M., Otto S. W., Huss-Lederman S., Walker D. W., Dongarra J. MPI, The Complete Reference. Scientific and Engineering Computation. MIT Press, 1996. 140 p.

11. Lamport L. Time, clocks, and the ordering of events in a distributed system //Communications of the ACM. 1978. Vol. 21, No 7. P. 558–565.
12. Kolpakov R. M., Posypkin M. A. Estimating the computational complexity of one variant of parallel realization of the branch-and-bound method for the knapsack problem //Journal of Computer and Systems Sciences International. 2011. Vol. 50, No. 5. P. 756-765.
13. Posypkin M. A., Sigal I. K. Speedup estimates for some variants of the parallel implementations of the branch-and-bound method //Computational Mathematics and Mathematical Physics. 2006. Vol. 46, No. 12. P. 2187-2202.
14. Voevodin V., Antonov A., Dongarra J. AlgoWiki: an Open Encyclopedia of Parallel Algorithmic Features //Supercomputing frontiers and innovations. 2015. Vol. 2, No. 1. P. 4-18.