Performance Analysis of the BiqBin solver for the maximum stable set problem

A. Shamakina\textsuperscript{a,∗}, T. Hrga\textsuperscript{b,†}, J. Povh\textsuperscript{b,‡}

\textsuperscript{a}High-Performance Computing Center Stuttgart (HLRS), University of Stuttgart, Nobelstrasse 19, 70569 Stuttgart, Germany
\textsuperscript{b}University of Ljubljana, Faculty of Mechanical Engineering, Aškerčeva cesta 6, 1000 Ljubljana, Slovenia

Abstract

The BiqBin application is a high-performance solver for linearly constrained binary quadratic problems. This software is open source and available as an online solver. In this White Paper, we report on optimizations of the branch of the BiqBin code designed for solving the maximum stable set problem, which were implemented in the course of the PRACE Preparatory Access Type D project “BiqBin solver” in the time frame of September 2018 to February 2020. This research on the BiqBin code consists of a performance analysis and implementation of a prototype. Optimizing the original BiqBin code v1.0 to a new version v2.0 allowed to increase the speedup 6.2 times for a use case on 192 MPI processes. Further analysis of the BiqBin v2.0 showed a bad parallel efficiency due to the high proportion of MPI communications. All suggested optimization ideas were demonstrated on the prototype.

Keywords: BiqBin, maximum stable set problem, branch and bound algorithm, hybrid programming model, overlapping

Introduction

A wide range of mathematical optimization problems having a quadratic objective function and binary encapsulated variables exist, such as combinatorial optimization to real life logic programs. Such problems are in general NP-hard, so we can solve instances of relevant sizes only approximately, using a combination of heuristic and meta heuristic algorithms. However, to test these approximation algorithms one still needs a ground truth for small or medium size problems, so algorithms to solve such problems for a proven optimum are highly needed. In a joint Slovenian-Austrian research project High-Performance Solver for Binary Quadratic Problems was designed such algorithm called BiqBin and implemented it in a C++ code [1]. A special branch of the code is designed to solve the so called maximum stable set problem. It is developed to run in parallel on HPC systems and can solve instances of the size of 300 vertices in an input graph.

The aim of the PRACE type D project was to produce a performance analysis, detect bottlenecks and optimize the source code. The performance analysis includes a strong scaling experiment on a PRACE Tier-1 system. The finding and fixing of bottlenecks allow improving the parallelism of the code and enabling large simulations with instances of size up to 500 vertices in an input graph.

In this paper we provide an overview of performance analysis and optimization for the BiqBin code. In Section 1 we describe the BiqBin solver and the main algorithm behind it. Section 2 is devoted to the hardware and software used for experiments. In Section 3 the focus of analysis for the BiqBin v1.0 is presented. The strong scalability is also shown. The full performance analysis for the BiqBin
v2.0 code is presented in Section 4. Section 5 deals with the implementation of the prototype. In the last section we conclude the optimization project.

1. BiqBin project

The main objective of the BiqBin project is to develop a best possible solver for linearly constrained non-convex quadratic programs with binary variables, that will be fully parallelized and will be adapted to run on large high-performance computers [1]. A special branch of the code is designed to solve a special problem case, called the stable set problem. We are given an unweighted graph with a set of vertices and edges. The subset of vertices is called stable if none of the vertices in this set are connected by an edge. The maximum stable set problem asks to find the largest stable set in the given graph. This maximum cardinality is referred to as the stability number or independence number of a graph. The problem is modelled as a binary integer problem and the main method for solving it is the branch and bound algorithm (B&B) [14]. Each node of this dynamically growing binary tree corresponds to branching of one binary variable (setting it to 0 or 1). For this selection, we compute an upper and lower bound on the optimal solution. In our case one semidefinite programming problem, which is a computationally demanding task, because it involves solving large systems of linear equations, needs to be solved in order to obtain the upper bound for the optimum value. Also, a heuristic algorithm is used to obtain the lower bound for the optimum value.

For computing the upper bound we use the theta-function of a graph which was introduced by Lovász [15]. This number is the optimum of a semidefinite program and has the property that it is an upper bound on the stability number. We have written and used two solvers (ADMM – Alternating Direction Method of Multipliers [16] and IPM – interior-point method [17]) tailored for the underlying semidefinite program that needs to be solved in each node of the branch and bound tree. Numerical results show that the density of the underlying graph is the key parameter guiding the choice of SDP solver. For sparse graphs with edge density below 10%, the interior-point method is the algorithm of choice. For other instances the ADMM method behaves better and the run-time for performing the relaxation is much less.

The input data for the BiqBin solver is an unweighted graph with a set of vertices and edges. We generated random graphs for experiments by a MATLAB [4] script by specifying the number of nodes and a density. The node number and the density determine the problem size. The density is translated to the number of edges in the graph. If the density is between 5% and 25%, the B&B algorithm shows a good scaling. As an output the BiqBin solver returns the stability number of an input graph.

The BiqBin solver was written in C++11 [5]. The parallel programming model based on MPI and the Intel Compiler 2018a has been used. The solver also links with external libraries such as the Armadillo library [6] and the Intel® Math Kernel Library (Intel® MKL) [7]. Armadillo v9.100.5 is a C++ library for numerical linear algebra and scientific computing. The library routines were used to implement bounding methods that solve semidefinite programming relaxations in each B&B node.

2. Hardware and Software Used for Optimization

The analysis was performed on the Salomon cluster at IT4I that consists of 1008 computational nodes of which 576 are regular compute nodes and 432 accelerated nodes. Each node is a powerful x86-64 computer, equipped with 24 cores (two twelve-core Intel Xeon E5-2680v3 processors) and 128 GB RAM. The nodes are interlinked by high speed InfiniBand and Ethernet networks. All nodes share 0.5 PB /home NFS disk storage to store the user files. Users may use a DDN Lustre shared storage with capacity of 1.69 PB which is available for the scratch project data. Analysis was done mainly on one and two compute nodes.

The code was instrumented with Extrae v3.6.1 [3] using the libmpitrace.so library and PAPI v5.5.1 [9] hardware counters instructions (PAPI_TOT_INS), cycles (PAPI_TOT_CYC). An Extrae trace was visualized by Paraver v4.8.1 [10].
3. BiqBin v1.0

3.1. Application Structure of the BiqBin v1.0

Initially, we created an Extrae trace for the original BiqBin code (v1.0) on 8 nodes. The specification of the measurement setup is shown in Table 1.

The runtime of the BiqBin v1.0 application was 24.61 seconds and the runtime with instrumentation was 19.94 seconds. Two launches for the same application may vary in time, because the solving process is not deterministic. In this situation we see that the runtime was even decreased having the additional instrumentation in place. In general, the Extrae instrumentation does not significantly affect the application runtime, we checked that for a set of experiments on the different numbers of compute nodes.

<table>
<thead>
<tr>
<th>Measurement Setup</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>8</td>
</tr>
<tr>
<td>Total number of MPI processes</td>
<td>192</td>
</tr>
<tr>
<td>OpenMP threads per MPI process</td>
<td>1</td>
</tr>
<tr>
<td>Input data (“middle” task)</td>
<td>sparse_graph150_density10</td>
</tr>
</tbody>
</table>

Table 1: Specification of the measurement setup for Extrae tool.

A timeline view of the BiqBin v1.0 application is presented in Figure 1. Time goes from left to right, MPI processes from top to bottom. Colours indicate the state of the processes.

A timeline view of the BiqBin v1.0 application is presented in Figure 1. Time goes from left to right, MPI processes from top to bottom. Colours indicate the state of the MPI processes: an initialization part at the beginning (A), followed by computational part (B) and a short finalization part at the end (C). The yellow-green colour in part A represents *MPI_Init*. Part B starts with orange colour, that is
MPI_Bcast. Blue colour indicates computations, red colour is MPI_Send and MPI_Recv and MPI waiting time. Part C consists only of MPI_Finalize shown in yellow-green colour.

The times for each part are reported in Table 2 which shows that 88 percent of the total time is spend in the computational part B. Therefore, we select the computational part as focus of analysis. Note that blue colour at the beginning of part B has the form of a stair, because the MPI master process generates and sends subgraphs to MPI workers, which serialize this starting phase. Accordingly, there is a delay time for MPI workers.

Table 2: Times spent in the parts of the BiqBin v1.0 in seconds.

<table>
<thead>
<tr>
<th>Trace (192 MPI processes)</th>
<th>Part A</th>
<th>Part B</th>
<th>Part C</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.17</td>
<td>17.61</td>
<td>1.16</td>
<td>19.94</td>
</tr>
</tbody>
</table>

Furthermore, the average running (useful) time is only 10.22 percent and the average idle time is 89.78 percent. MPI communication shows a significant overhead. Particularly MPI_Recv which is 80.47 percent of the all runtime. Here it is also important that the BiqBin v1.0 application uses the master/worker scheme. The master process serves as a load coordinator and does only MPI communications as shown by a straight red line in part B in Figure 1. The worker processes perform the calculations.

3.2. Strong Scalability of the BiqBin v1.0

The strong scalability for the sparse_graph250_density10 input file (“big” task) is shown in Figure 2. From the figure it follows that the application has almost ideal scaling.

Figure 2. Strong scalability of the BiqBin v1.0 for the sparse_graph250_density10 input file.

The speedup and efficiency for the sparse_graph250_density10 input file is shown in Figure 3. The application has high efficiency. It is 85.37% even on 64 compute nodes.
3.3. Description of the algorithm in the BiqBin v1.0

To develop a parallel algorithm instead of the sequential branch and bound method we use a master-workers scheme to manage the overall load in which MPI process 0 becomes the master process distributing work to others and carefully managing the status of each worker (idle or busy) by using an array. Each time a branching step is done (fixing one variable to 0 or 1) two new child subproblems are generated. With this procedure, a binary tree is built dynamically throughout the algorithm. In order to solve multiple SDP relaxations in parallel (evaluation of different B&B nodes), workers need to send and receive subproblems. In the BiqBin v1.0 application this is done though the master process. At the beginning of the algorithm the load coordinator distributes the whole graph since the workers need to know this information when receiving instructions on which subproblem to work. Then the original problem is sent to the first worker. After the bounding step (computation of upper and lower bound), the worker generates two subproblems. It keeps one of them, while the other is send back to the load coordinator which decides to whom the subproblem is sent next. In this way the work is shared and different processes can work in parallel.

However, this parallelization scheme has a drawback within the generation of the B&B binary tree. First we start with a root node (level 0). With each branching step, two new child nodes are created and shared among processes. Suppose the number of available processes is \( N \). To fully use all of them we need to reach depth \((\log_2 N)\) rounded above. If the bounding step is not trivial, idle time of some workers can be large and efficiency of the algorithm drops. This algorithm leads to the fact that the part B starts computations in the form of a stairs in Figure 1. This was solved in the BiqBin v2.0.

3.4. Changes in the original algorithm of the BiqBin v1.0

With the second version we addressed two problems encountered in the first phase: Those are large sizes of messages going to and from the load coordinator and the phenomenon seen in Figure 1, where, until a certain depth of the binary tree is reached, some workers are idle and thus resources are wasted.

However, with increased number of workers one load coordinator could not handle so many messages and the communication became a bottleneck. For the subproblems to be send from one worker to another the message has to pass though the load coordinator. To reduce the communication each worker is using its own local queue of subproblems and the message exchange is done without involving the master process. Its job is only to monitor the status of workers and to distribute the best solution found so far. Reducing and properly redirecting the communication between the active workers enables faster pruning of the branch and bound tree and therefore faster termination of the parallel algorithm.

If the number of available workers is large, we need to reach a certain depth of the B&B binary tree in order for the processes to receive some work. Until this happens in the algorithm, workers idle. To fully exploit all the HPC resources available we need a strategy for the worker processes to start
evaluating the nodes of B&B tree as soon as possible. An initial branching strategy is used by the load-coordinator. Based on the number of workers, the master process branches the original problem as long as the number of obtained subproblems equals the number of free workers. These generated subproblems are then distributed among the processes and parallel evaluation of the B&B tree nodes starts.

4. BiqBin v2.0

4.1. Application Structure of the BiqBin v2.0

After this optimization we obtained BiqBin v2.0. A new Extrae trace for the BiqBin v2.0 was created with the same specification as before (Table 1). The Paraver timeline view of the trace is shown in Figure 4.

Figure 4. Paraver timeline view of the 8 nodes BiqBin v2.0 trace. The run was performed with 192 MPI processes. Time goes from left to right, MPI processes from top to bottom. Colours indicate the state of the processes.

The runtime has been significantly reduced from 19.94 to 3.22 seconds. Thus, the speedup is 6.2 times. The times for individual parts are reported in Table 3. The average running (useful) time is 31.18 percent, while the average idle time is 68.82 percent. MPI communication still shows a significant impact. Particularly MPI_Recv which is 40.55 percent of the overall runtime.

Table 3: Times spent in the parts of the BiqBin v2.0 in seconds.

<table>
<thead>
<tr>
<th>Part</th>
<th>Part A</th>
<th>Part B</th>
<th>Part C</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trace (192 MPI processes)</td>
<td>0.31</td>
<td>2.66</td>
<td>0.25</td>
<td>3.22</td>
</tr>
</tbody>
</table>

Here we can notice that the input graph with 150 nodes and density ten is too small for eight compute nodes. There are not enough tree nodes in the dynamic graph, in order to calculate them on 192 MPI processes. In addition, the dynamic graph tiers are not wide enough. Sometimes the width is only 73 nodes and 73 MPI processes perform calculations, while the other processes are waiting.

4.2. Strong Scalability of the BiqBin v2.0

The strong scalability for the sparse_graph250_density10 input file is shown in Figure 5. From the figure it follows that the application a scaling better than ideal. An explanation of this phenomenon will be given below.
Figure 5. Strong scalability of the BiqBin v2.0 for the sparse_graph250_density10 input file.

The speedup and efficiency for the sparse_graph250_density10 input file is shown in Figure 6. The application has huge efficiency. It is 191.47% even on 64 compute nodes.

Figure 6. Speedup and efficiency of the BiqBin v2.0 for the sparse_graph250_density10 input file.

The B&B algorithm consists of a systematic enumeration of candidate solutions by means of state space search: the set of candidate solutions is thought as forming a rooted tree with the full set at the root. In case of the BiqBin solver the tree is binary. An example of a simplified graph is shown in Figure 7.

Figure 7. An example of a dynamic B&B tree.
Figure 8. A mapping of the graph nodes on 4 MPI ranks and on 8 MPI ranks.

In contrast to the sequential algorithm, the parallel algorithm explores multiple branches of the B&B tree at the same time (Figure 8). This decreases the number of steps and results in faster execution time (speedup) of the parallel algorithm. Suppose that multiple branches are being enumerated by two workers. A Node of the tree can be safely pruned (a subset of feasible solutions can be safely omitted since it does not contain the optimal one), if the difference between upper and lower bound is sufficiently small. In the case of integer weights this is less than one. In order to faster prune the nodes, the relaxations have to produce tight upper bounds close to the optimum as well as good heuristics are needed that produce feasible solutions with high objective values, possibly the optimal ones. If one of the workers quickly finds a stable set (lower bound) that is optimal (but needs to be verified by the algorithm), this helps pruning branches of another worker, thus acting advantageously on the pruning process in the parallel algorithm. This is the reason for superlinear speedup observed with many instances and also reported for similar combinatorial problems in the literature [18].

4.3. Performance Analysis of the BigBin v2.0

Multiple traces with Extrae were generated: on 1 node, 2 and 8 nodes. Table 4 and Table 5 provide the comparative information that was obtained using the basic analysis script [11]. The description of metrics for parallel performance analysis is given in [12]. These metrics are used in the POP (Performance Optimisation and Productivity) project. The metrics are then calculated as efficiencies between 0 and 1, with higher numbers being better. In general, we regard efficiencies above 0.8 as acceptable, whereas lower values indicate performance issues that need to be explored in detail.

Table 4: Overview of the collected raw data.

<table>
<thead>
<tr>
<th></th>
<th>24</th>
<th>48</th>
<th>192</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime (us)</td>
<td>11862139.15</td>
<td>6438986.78</td>
<td>3219500.88</td>
</tr>
<tr>
<td>Runtime (ideal)</td>
<td>11835891.39</td>
<td>6413848.17</td>
<td>10860068.0</td>
</tr>
<tr>
<td>Useful duration (average)</td>
<td>9536899.11</td>
<td>4521459.38</td>
<td>1803719.68</td>
</tr>
<tr>
<td>Useful duration (maximum)</td>
<td>11366230.24</td>
<td>5621986.39</td>
<td>1701823.4</td>
</tr>
<tr>
<td>Useful duration (total)</td>
<td>230064218.54</td>
<td>217838650.13</td>
<td>192714177.87</td>
</tr>
<tr>
<td>Useful duration (ideal, max)</td>
<td>11367549.64</td>
<td>5425698.6</td>
<td>802849.6</td>
</tr>
<tr>
<td>Useful instructions (total)</td>
<td>534967491708</td>
<td>517177405672</td>
<td>886811454908</td>
</tr>
<tr>
<td>Useful cycles (total)</td>
<td>365264293139</td>
<td>348366584251</td>
<td>469094462744</td>
</tr>
</tbody>
</table>

Table 5: Overview of the computed model factors.

<table>
<thead>
<tr>
<th></th>
<th>24</th>
<th>48</th>
<th>192</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel efficiency</td>
<td>86.81%</td>
<td>70.22%</td>
<td>31.18%</td>
</tr>
<tr>
<td>Load balance</td>
<td>84.34%</td>
<td>90.83%</td>
<td>59.81%</td>
</tr>
<tr>
<td>Communication efficiency</td>
<td>95.82%</td>
<td>77.99%</td>
<td>52.84%</td>
</tr>
<tr>
<td>Serialization efficiency</td>
<td>96.02%</td>
<td>84.60%</td>
<td>-</td>
</tr>
<tr>
<td>Transfer efficiency</td>
<td>99.79%</td>
<td>92.10%</td>
<td>-</td>
</tr>
<tr>
<td>Computation scalability</td>
<td>108.00%</td>
<td>106.01%</td>
<td>119.38%</td>
</tr>
<tr>
<td>Global efficiency</td>
<td>88.81%</td>
<td>74.44%</td>
<td>37.22%</td>
</tr>
<tr>
<td>IPC scalability</td>
<td>100.00%</td>
<td>101.38%</td>
<td>117.47%</td>
</tr>
<tr>
<td>Instruction scalability</td>
<td>100.00%</td>
<td>103.43%</td>
<td>66.38%</td>
</tr>
<tr>
<td>Frequency scalability</td>
<td>100.00%</td>
<td>101.10%</td>
<td>153.29%</td>
</tr>
<tr>
<td>Speedup</td>
<td>1.00</td>
<td>1.84</td>
<td>3.68</td>
</tr>
<tr>
<td>Average IPC</td>
<td>1.46</td>
<td>1.48</td>
<td>1.72</td>
</tr>
<tr>
<td>Average frequency (GHz)</td>
<td>1.59</td>
<td>1.61</td>
<td>2.43</td>
</tr>
</tbody>
</table>
4.4. Efficiency

According to the Table 5, the parallel efficiency is good on 1 compute node (80.81%), a little bit worse on 2 nodes (70.22%) and very bad on 8 nodes (31.18%). The value of the parallel efficiency depends on the load balance and the communication efficiency.

4.5. Load Balance

According to Table 5, the BiqBin application has some issues with the load balance. The load balance equals to 59.01 %. All processes have an almost equal amount of work. However, the sizes of individual pieces of work are different. This is due to the solver used for computing the optimal value of SDP relaxations. Computation of the theta function is used in the bounding routine and the choice between ADMM and interior-point method is based on the density of the underlying graph that determines the subproblem. ADMM is used when the density is above 10 %, whereas IPM is used otherwise. The density of the test instance in Figure 9 is around 10 %. When branching is done some of the edges and vertices are removed and the obtained subproblems vary in density. Thus some workers use a different method for computing the upper bound than others and time spent evaluating nodes is distinct. Hence for some instances the parallel algorithm has uneven load balance.

Figure 9. Paraver histogram showing the useful duration.

A Paraver histogram of the useful computation duration (user function code) is shown in Figure 9. This histogram was created only for the useful duration for the trace part with the computation. Each useful duration has a size in microseconds. All sizes are divided into time ranges. The time ranges of useful duration are sorted and increases from left to right in the range [1 207 312.83 – 3 219 500.88 us], the rank ID goes from top to bottom (from 0 to 192) and colours indicate the aggregated useful duration in this time range. Thus, the maximum of the aggregated useful duration (a rectangle of a blue colour circled in red) equals to 986 577.85 us. That means that MPI Rank 25 has a longer aggregated useful duration than other MPI processes.

The timeline in Figure 10 shows also the useful duration, but here there are separate values of the useful duration. The panel contains information about time ranges in microseconds and colours. MPI rank 25 (thread 1.26.1 in Figure 10) has bigger pieces of work (orange colour) than the rest: 986 573.85 microseconds. This is also the reason of the load imbalance in the BiqBin solver.
4.6. Computational Performance

The Intel Xeon E5-2680v3 processor allows the execution of multiple Instructions Per Cycle (IPC) as they have multiple execution ports, which can operate in parallel, e.g., floating point addition and multiplication as well as a logical operation can be executed at the same time. The utilization of the CPU features can be measured with the IPC metric. The Xeon E5-2680v3 has 6 internal ports and is capable to execute in the best case three different arithmetic operations per cycle. Therefore, the IPC limit for mixed floating point computation and logic code is three and for floating point arithmetic two. To get an impression of the CPU utilization and computational performance of the BiqBin, the IPC is determined for the useful code with PAPI counters. Only the application specific code is considered, as e.g., MPI library calls do internally busy looping and would thus distort the IPC value of the application. In general, IPC values of one and above are considered as good.

In Figure 11 the IPC ranges are sorted and increase from left to right (from 0.05 to 2.73), the rank ID goes from top to bottom (from 1 to 192), colours indicate the aggregated useful durations with appropriate IPC. The maximum IPC value of the BiqBin solver is in the range [2.53, 2.54]. This is a very good indicator. There is also a line from IPC values from the range [2.30, 2.32]. This line tells about the quality of IPC balance. The average IPC value found for all runs is in the range [2.48, 2.61], which is considered as good.
Figure 11. Paraver histogram showing the IPC.

4.7. Communications

The MPI call statistics is shown in Table 6. On average, each process spends only 32.53% of the total runtime outside MPI. The most significant MPI operation is MPI_Recv (40.55%). It is due to the processes are waiting for new work, it is connected to the problem of enough data to be distributed.

The master process behaviour is different from worker process behaviour. The rank 0 spends 37.82% outside MPI and 53.18% of the runtime on MPI_Recv. The rank 0 calls MPI_Recv 829 times (Table 7), the average number of MPI_Recv calls for the workers MPI processes is 18.12.

Table 6: MPI call statistics for the runtime (in percents).

<table>
<thead>
<tr>
<th></th>
<th>Outside MPI</th>
<th>MPI_Send</th>
<th>MPI_Recv</th>
<th>MPI_Bcast</th>
<th>MPI_Comm_rank</th>
<th>MPI_Comm_size</th>
<th>MPI_Init</th>
<th>MPI_Finalize</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>32.53</td>
<td>0.08</td>
<td>40.55</td>
<td>15.14</td>
<td>0.00</td>
<td>0.06</td>
<td>4.59</td>
<td>7.64</td>
</tr>
<tr>
<td>Max</td>
<td>62.27</td>
<td>1.17</td>
<td>57.26</td>
<td>15.33</td>
<td>0.01</td>
<td>0.1</td>
<td>9.15</td>
<td>7.72</td>
</tr>
<tr>
<td>Min</td>
<td>17.31</td>
<td>0.00</td>
<td>14.76</td>
<td>0.00</td>
<td>0.00</td>
<td>0.03</td>
<td>0.00</td>
<td>7.58</td>
</tr>
<tr>
<td>Avg/Max</td>
<td>0.52</td>
<td>0.07</td>
<td>0.71</td>
<td>0.99</td>
<td>0.31</td>
<td>0.6</td>
<td>0.5</td>
<td>0.99</td>
</tr>
</tbody>
</table>

Table 7: Average number of MPI calls.

<table>
<thead>
<tr>
<th></th>
<th>Outside MPI</th>
<th>MPI_Send</th>
<th>MPI_Recv</th>
<th>MPI_Bcast</th>
<th>MPI_Comm_rank</th>
<th>MPI_Comm_size</th>
<th>MPI_Init</th>
<th>MPI_Finalize</th>
</tr>
</thead>
<tbody>
<tr>
<td>Master</td>
<td>2816</td>
<td>1976</td>
<td>829</td>
<td>6</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Workers</td>
<td>41.23</td>
<td>12.11</td>
<td>18.12</td>
<td>6</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

4.8. I/O

The application spends on an output only 7 770.27 microseconds. Each MPI process calls the output function once at the end of the application.
4.9. Summary

The BiqBin v2 application has a clear structure with initialization, computation and finalization. The initial analysis of the BiqBin application for the “middle” size of an input graph based on the MPI trace showed that it has:

- bad parallel efficiency (31.18%);
- bad load balance (59.01%);
- high IPC (average value is 2.55);
- on average, each process spends 68.47% of the runtime on MPI communications;
- the output takes non-essential part of the runtime.

Most of the application runtime is spend on MPI. It is recommended to optimize the B&B algorithm in the following way:

1) decrease the number of MPI communication inside each compute node by using OpenMP programming model;
2) decrease the number of MPI communication between compute nodes.

5. Prototype

The code optimisation of the BiqBin solver suggests significant changes in the B&B algorithm as well as the structure of the application. Therefore, it was decided to demonstrate the main ideas on a prototype which is a proof of concept. These ideas proposed by the PRACE project expert were not implemented in the BiqBin application. However, a hybrid implementation can significantly improve the BiqBin scalability for a large input graph and decrease the number of MPI communications.

The differences of the BiqBin v2.0 and the prototype are presented in Table 8. In particularly, the prototype uses a hybrid programming model (MPI+OpenMP). It allows to run only 1 MPI process per compute node. MPI rank 0 on the compute node 0 is the master process. MPI processes on other nodes are the MPI workers. The master process coordinates the work of the MPI Workers (master-worker model). MPI processes communicate between compute nodes. In addition, there are 24 OpenMP threads per node in the prototype. They perform computations. The use of OpenMP threads allows to reduce the number of MPI processes in total. Results of these computations are stored on the node. The MPI Worker of each node analyses the results and, if necessary, transfers results to the MPI master.

Table 8: Code optimization.

<table>
<thead>
<tr>
<th>№</th>
<th>Changes</th>
<th>BiqBin v2.0</th>
<th>Prototype</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Programming Model</td>
<td>MPI</td>
<td>MPI+OpenMP</td>
</tr>
<tr>
<td>2</td>
<td>#MPI processes per node</td>
<td>24</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>#OpenMP thread per node</td>
<td>-</td>
<td>24</td>
</tr>
<tr>
<td>4</td>
<td>Initialization</td>
<td>MPI_Send, P2P</td>
<td>MPI_Scatter, collective</td>
</tr>
<tr>
<td>5</td>
<td>Main loop of the MPI master</td>
<td>MPI_Recv</td>
<td>MPI_Probe</td>
</tr>
<tr>
<td>6</td>
<td>Data amount information</td>
<td>Extra MPI_Recv</td>
<td>MPI_Get_count</td>
</tr>
<tr>
<td>7</td>
<td>Overlapping computations and communications</td>
<td>-</td>
<td>OpenMP tasks</td>
</tr>
<tr>
<td>8</td>
<td>Structure for free MPI workers</td>
<td>array</td>
<td>queue</td>
</tr>
<tr>
<td>9</td>
<td>Local priority queue with subproblems</td>
<td>not safe</td>
<td>safe</td>
</tr>
<tr>
<td>10</td>
<td>Division into classes</td>
<td>partially</td>
<td>completely</td>
</tr>
</tbody>
</table>
5.1. UML Class Diagram of the Prototype

The prototype code is divided into classes for ease of implementation. An UML class diagram of the prototype is shown in Figure 12. Only important attributes and methods are presented. The arguments for the methods are also skipped.

**Scheduler class** is an auxiliary class for the implementation of scheduler strategies. In the current version of the prototype the Scheduler class contains a method to count redundant subproblems on a compute node. There is also a method to count a number of required workers in order to process these redundant subproblems.

**Safe_queue class** is an auxiliary class for implementation of a queue with the ability to safely get/put elements and so on. The OpenMP threads have access to this queue.

**MPIprocess class** is a basic class that contains typical attributes of an MPI process (a rank, a size, a number of OpenMP threads per MPI process) and special attributes (a copy of GLB, a local GLB, a queue with local subproblems, a scheduler).

**Master class** is a child class for the MPIprocess class. The Master class also has specific attributes: A queue of the free Workers and the main Global Lower Bound (GLB). A Master process can do communication with other processes and local communication is performed by using OpenMP threads.

**Worker class** is a child class for MPIprocess class. The Worker class has no additional attributes. A Worker process can only do communication.

---

Figure 12. UML class diagram of the prototype.
Prototype program is the main program. A header of the Prototype program contains only a description of messages and tags for MPI communications. The Prototype program creates one instance of the Master class and several instances of the Worker class.

5.2. MPI process interaction between compute nodes

The UML Sequence Diagram in Figure 13 shows how MPI processes interact with each other. The actions number 1, 4 and 5 were performed in a parallel mode by using MPI collective calls: MPI_Bcast, MPI_Scatter and MPI_Bcast accordingly. The actions 1-5 are the initialization of the application. Then comes the main loop.

There are three basic options of interaction between the MPI master and the MPI worker:

1) finding a new value of the GLB;
2) getting the free workers;
3) informing about lack of work.

The implementation of this loop is shown in Figure 18.
The UML Sequence Diagram in Figure 14 shows the interaction between OpenMP threads and the MPI worker. One OpenMP thread creates two OpenMP tasks for communication and computation, which are handled by free OpenMP threads. The MPI calls are only in the part of the communication task and are executed by the MPI worker.
5.3. Prototype program

The source code of the Prototype program is shown in Figure 15. MPI rank 0 creates one instance of the Master class. Other ranks create one instance of the Worker class. Then the master and workers do the initialization and running. At the end of the running the master stops all the workers.

Figure 15. The source code of the Prototype.cpp file.

Prototype.cpp

```c
int main(int argc, char** argv){
    int rank;
    int size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    if (rank == 0){
        Master master(rank, size);
        master.init();
        master.run();
        master.stop();
    } else {
        Worker worker(rank, size);
        worker.init();
        worker.run();
    }
    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Finalize();
    return 0;
}
```

5.4. Initialization of the MPI Master and MPI Workers

During the initialization the master sends the subproblems to the workers. In the BiqBin v2.0 solver the master did that in a loop by using MPI_Send. But this MPI call is a P2P operation. Whenever
possible, it is recommended to replace individual MPI calls with collective MPI calls (Figure 16). Therefore, we used `MPI_Scatter` call. It demands to use an array as an argument (`shbuf`).

Figure 16. The initialization in the BiqBin solver and the Prototype.

5.5. Overlapping computations and communications

We organized the simultaneous execution of the computation and communication inside each MPI process in the prototype (Point 7 in Table 8). The overlapping for the MPI master and workers was implemented by using OpenMP tasks (Figure 17). Each MPI worker has an OpenMP parallel section. Only one OpenMP thread creates two OpenMP tasks: a communication task and a computation task. Free OpenMP threads process these tasks.

The implementation for the MPI master is a little bit harder. The MPI master coordinates MPI workers on other compute nodes and at the same time coordinates the local OpenMP threads on the compute node 0 as MPI worker. Thus, it combines two roles: the MPI master (global communication) and the MPI worker (local communication) within MPI process 0. The MPI process 0 uses for local communication only local variables. It allows also to decrease the MPI communication.

Figure 17. The overlapping computations and communications by using OpenMP tasks in the Prototype.

5.6. Probing incoming communications

Decreasing the amount of the `MPI_Recv` calls can help to improve the overall performance. As an alternative probing is very useful, it serves many purposes such as getting the count of elements we are about to receive, the ID and tags of the processes we are receiving from or if we are actually receiving
nothing [13]. The probing only informs that the process is ready to receive a communication, but does not block the process itself.

In Figure 18 in the BiqBin v2.0 source code there are two MPIRecv calls. The first call (line 6) gives information about the MPI source. The second call (line 14) informs the master about a number of required workers.

In Figure 18 the prototype source code has a MPIProbe call (line 4). From this call the master knows an ID of MPI source and a message. Besides, MPIProbe is a blocking call. The master can do a real receiving in one of the subroutines of the switch statement (line 8-14). For example, in line 4 in the Master::free_workers_handler().

Figure 18. The implementation of the main loop for the MPI master in the BiqBin v2.0 and the Prototype.

5.7. Getting the Information about Data Amount

In the situation when a worker has extra subproblems, it will ask the master about free workers to help. In the BiqBin code the worker performs two MPI_Send and two MPI_Recv calls (Figure 19). It should send the message “SEND_FREEWORKERS”, send a request, receive an exact number of free workers and receive a vector with the free workers.

The worker in the prototype calls MPI_Send only once (line 11). The message variable consists of “SEND_FREEWORKERS” and the worker_request consists of a required workers number. As we already know, the master in the prototype uses an MPI_Probe call, in order to get information about the MPI source and a message tag. Thus, the master recognizes the tag and calls an appropriate subroutine (Master::free_workers_handler()) from Figure 18). The master also knows that the worker will request free workers. Therefore, it sends immediately a vector with free workers. The worker receives this data (line 15, Figure 19). Then, we use MPI_Get_count (line 18) on the received status to retrieve the information we want: the number of elements in the vector. This way, we can allocate only the right number of elements.
6. Conclusion

This research on the BiqBin code consists of two parts: a performance analysis and implementation of the prototype. The overview of the BiqBin v1.0 showed that the application had a problem with an initial generation of graph nodes and their distribution. This has been fixed in the BiqBin v2.0. The speed up was 6.2 times on 192 MPI processes.

The analysis also highlighted that the number of MPI calls have a significant impact on the overall performance and the load balance should be improved. The code optimisation of the BiqBin v2.0 suggests significant changes in the B&B algorithm and the structure of the application. Therefore, it was decided to demonstrate the main ideas on the prototype. The following modifications have been implemented in the prototype source code:

- hybrid programming model (MPI + OpenMP);
- overlapping computations and communications (OpenMP tasks);
- replacement of MPI P2P calls with MPI collective calls;
- using probing incoming communications;
- using `MPI_Get_count` for getting an exact amount of received data;
- division into classes.

The further line of work is to implement the proposed solutions in the BiqBin solver.

References

[13] Introduction to MPI. Probing an incoming communication.

Acknowledgements

This work was financially supported by the PRACE project funded in part by the EU’s Horizon 2020 Research and Innovation programme (2014-2020) under the grant agreements 730913 and 823767.