A comparison between MPI and OpenMP Branch-and-Bound Skeletons

Isabel Dorta, Coromoto León and Casiano Rodríguez
Departamento de Estadística, I.O. y Computación
Universidad de La Laguna
Edificio de Física y Matemáticas
E-38271 La Laguna, Tenerife, Spain

E-mail: `(isadorta, cleon, crguezl)@ull.es`

Abstract

This article describes and compares two parallel implementations of Branch-and-Bound skeletons. Using the C++ programming language, the user has to specify the type of the problem, the type of the solution and the specific characteristics of the Branch-and-Bound technique. This information is combined with the provided resolution skeletons to obtain a distributed and a shared parallel programs. MPI has been used to develop the Message Passing algorithm and for the Shared Memory one OpenMP has been chosen. Computational results for the 0/1 Knapsack Problem on a Sunfire 6800 SMP, a Origin 3000 and a PCs cluster are presented.

1 Introduction

Branch-and-Bound (BnB) is a widely and effective technique for solving hard optimization problems. It determines the optimum-cost solution of a problem through a selective exploration of a solution tree. The tree internal nodes correspond to different states of the solution search and the leaves correspond to feasible solutions. Some of the main reasons for which this technique is used are: First, many applications that fit that paradigm are highly computationally intensive and thus can benefit from multiple computers to improve their performance through parallelism. Second, an application following this paradigm can easily be divided into large numbers of coarse-grain subproblems (tasks). And third, each of those subproblems are highly asynchronous and self-contained, and there are limited communication among them. A subproblem receives information from other subproblem at the moment of its creation and it passes its results to other subproblem at the time of its termination. These three properties make the chosen paradigm suitable for execution in a network environment, where the cost of communication is high and must be avoided by using large numbers of independent coarse-grain subproblems.

This work presents two skeletons to solve optimization problems using the Branch-and-Bound technique. The implementation of the skeletons have been made in C++. Sequential code and parallel code of the invariant part of the solvers are provided for this paradigm. Parallel skeletons can be classified into *task* and *data* ones, taking into account the kind of parallelism used. In the *task* ones a skeleton creates a system of communicating processes. In the *data* case, a skeleton works on a distributed data structure, performing the same operations on some or all elements of this structure. There are in the literature several studies and implementations of *general purpose* skeletons using Object Oriented paradigms [1], [2], [4]. This work is focused in the Branch-and-Bound skeletons. For this paradigm have been developed several parallel tools. PPBB (Portable Parallel Branch-and-Bound Library) [13] and PUBB (Parallelization Utility for Branch and Bound algorithms) [10] propose implementations in the C programming language. BOB [5], PICO (An Object-Oriented Framework for Parallel Branch-and-Bound) [3] and COIN (Common Optimization Interface for Optimization Research) [8] are developed in C++ and in all cases a hierarchy of classes is provided and the user has to extend it to solve his/her particular problem.

In our proposal there are two different parts: One that implements the *resolution pattern* provided by the library and a part which the user has to complete with the particular characteristics of the *problem to solve*, that will be used by the resolution pattern. Required classes are used to store the basic data of the algorithm. BnB uses the class `Problem` to define the minimal standard interface to represent a problem, and the class `Solution` to typify a solution. The class `Subproblem` represents the area of

Figure 1. Distributed Scheme: Master

```
busy[0] = 1; for i = 1, nProcs { busy[i] = 0;}
idle = nProcs - 1;
//Send initial subproblem to first idle slave
auxSp = sp.initSubProblem();
outputPackect.send(firstIdle,
                   auxSp,   // initial subproblem
                   bestSol, // bestSolution
                   sol);    // current solution
idle--;
IDLE2WORKING(busy,firstIdle);       // mark this slave like working
while (idle < (groupSize-1)) {      // while there are working slaves
   recv(source, flag);
   while(flag) {
       if (SOLVE_TAG) {             // receive the final solution
         inputPacket.recv(source,
                          bestSol,  // best solution
                          sol);     // current solution
       }
       if (BnB_TAG) {               // receive a slave request
         inputPacket.recv(source,
                          high,     // upper bound asociated to the problem
                          nSlaves); // receive the number of requiered slaves
         if ( high > bestSol){      // the problem must be branched
           total= ((nSlaves <= idle)?nSlaves:idle);
           for i = 1, total { idle--; IDLE2WORKING(busy,i); }
           outputPacket.send(source,
                             total,       // number of assigned slaves
                             bestSol,     // best Solution
                             1,..., total // slaves identifiers
                            );
         }
         else { // the problem must be bounded
           outputPackted.send(source, DONE);
         }
       }
       if (IDLE_TAG) {     // receive the signal of an idle slave
         inputPacket.recv(source, IDLE);
         idle++;
         WORKING2IDLE(busy,source); // mark this slave like idle
       }
       recv(source, flag);
   } // while (flag)
} // while (idle < (groupSize-1))
// Send the ending message
for i = 1, groupSize { outputPacket.send(i, END); }
```

no explored solutions. Its method `branch(pbm, sps)` generates from the current subproblem the subset of subproblems to be explored. The `lower_bound(pbm,sol)` and `upper_bound(pbm,sol)` subproblem methods calculate a lower and upper bound respectively of the objective function for a given problem. Furthermore the user must specify in the definition of the `Problem` class whether the problem to solve is a maximization or minimization problem. Once the user has represented the problem, he/she obtains for free two parallel solvers without any additional effort. The solvers are implemented by the provided class `Solver`. In the class hierarchy there are one skeleton implemented using MPI [12] and other using OpenMP [7]. This is one of the main contributions of this work. With the same specification of resolution, the user obtains two parallel algorithms: one using the message passing paradigm and other with the shared memory one.

The rest of the article is organized as follows. The second and third paragraphs describe the message passing and the shared memory schemes, respectively. The computational results appear in the section fourth, where both schemes are applied to the resolution of the 0-1 Knapsack Problem. Finally, in the fifth section the conclusions and the future work are presented.

Figure 2. Distributed Scheme: Slave Code

```
while (1) {
  recv(source, flag);
  while (flag) {
    if (END_TAG){                 // receive the finishing message
      inputPacket.recv(MASTER, END); return;
    }
    if (PBM_TAG){                 // receive the problem to be branched
       inputPacket.recv(source,    // receive from a slave or the Master:
                        auxSp,     //   the initial subproblem
                        bestSol,   //   the best solution value
                        sol);      //   the current solution
       auxSol = sol;
       bqueue.insert(auxSp);       // insert the subproblem in the local queue
       while(!bqueue.empty()) {
         auxSp = bqueue.remove();  // pop a problem from the local queue
         high = auxSp.upper_bound(pbm,auxSol);    // upper bound
         if ( high > bestSol ) {
           low = auxSp.lower_bound(pbm,auxSol);   // lower bound
           if ( low > bestSol ) {
             bestSol = low;
             sol = auxSol;
             outputPacket.send(MASTER,    // send to the Master:
                               SOLVE_TAG, //   the problem is solved
                               bestSol,   //   the best solution value
                               sol);      //   the solution vector
           }
           if ( high != low ) {
             rSlaves = bqueue.getNumberOfNodes(); // calculate the number of required slaves
             op.send(MASTER,                      // send to the Master:
                     BnB_TAG,                     //   the request of help
                     high,                        //   the upper bound
                     rSlaves);                    //   the number of requiered slaves
             inputPacket.recv(MASTER,                  // receive from the Master:
                              nfSlaves,                //   the number of assigned slaves
                              bestSol,                 //   the best solution value
                              rank {1,..., nfSlaves}); // the identifiers of the slaves
             if ( nfSlaves >= 0) {
               auxSp.branch(pbm,bqueue);    // branch and save in the local queue
               for i=0, nfSlaves{           // send subproblems to the assigned slaves
                 auxSp = bqueue.remove();
                 outputPacket.send(rank,    // send to the slave:
                                   PBM_TAG, //   a subproblem to be solved
                                   auxSp,   //   the subproblem information
                                   bestSol, //   the best solution value
                                   sol);    //   the solution vector
               }
             } // if nfSlaves == DONE the problem is bounded (cut)
       } } }
       outputPacket.send(MASTER, IDLE_TAG);  // send to Master the signal to say I am idle
     }
     recv(source, flag);
   } // while (flag)
} // while(1)
```

2 Message Passing Skeleton

This section describes the design and implementation of the BnB message passing resolution pattern. We have followed an iterative approach for the design of the sequential skeleton. The parallel version uses a Master/Slave scheme.

The code in figure 1 shows the tasks for the Master. The generation of new subproblems and the evaluation of the results of each of them are completely separated of the individual processing of each subtask. The Master is responsible of the coordination between subtasks. The Master has a data structure *busy* where registers the occupation state

Figure 3. Shared Memory scheme

```
// shared variables  {bqueue, bstemp, soltemp, data}
// private variables {auxSol, high, low}
// the initial subproblem is already inserted in the global shared queue
while(!bqueue.empty()) {
  nn = bqueue.getNumberOfNodes();   // calculate the number of necessary threads
  nt = (nn > maxthread)?maxthread:nn;
  data = new SubProblem[nt];        // compute the subproblem for each thread
  for (int j = 0; j < nt; j++)
    data[j] = bqueue.remove();
  set.num.threads(nt);              // establish the right number of threads
  parallel forall (i = 0; i < nt; i++) {
    high = data[i].upper_bound(pbm,auxSol);
    if ( high > bstemp ) {
      if ( low > bstemp ) {        // critical region
        // only one thread can change the value at any time
        bstemp = low;
        soltemp = auxSol;
      }
      if ( high != low ) {        // critical region
        // just one thread can insert subproblems in the queue at any time
        data[i].branch(pbm,bqueue);
} } } }
bestSol = bstemp;
sol = soltemp;
```

of each slaves (line 1); at the beginning all the slaves are idle. The initial subproblem, the best solution and the best value of the objective function are sent to an idle slave (lines 3-10). While there are idle slaves the Master receives information from them and decides the next action to apply depending if the problem is solved (line 14), if there are a slaves request (line 19) or if the slave has not work to do (line 36). If the problem is solved, the solution is received and stored (line 15). When the master receives a request of certain number of slaves, it is followed with the upper bound value (lines 21-22). If the upper bound value is better than the actual value of the best solution, the answer to the slave includes the number of slaves that can help to solve its problem (lines 26-30). In other case, the answer indicates that it is not necessary to work in this subtree (line 33). When the number of idle slaves is equal to the initial value, the search process finishes, then the Master notifies the slaves to end the work (line 45).

A slave (see figure 2) works bounding the received problem (lines 8-11). New subproblems are generated calling to the `branch` method (line 38). The slave asks for help (line 33) to the Master. If no free slaves are provided, the slave continues working locally. In other case, it removes subproblems from its local queue and sends them directly to other slaves.

The scheme presented in this article is not a centralized scheme in the sense of the Master do not distribute the work. A slave is who must distribute the subproblems between other slaves. On the other hand, the Master is who checks if there are free slaves or not, for this reason the proposed scheme is not an absolutely distributed one. The Master and Slave codes are *provided*. The provided classes are those which the user has to call when uses a skeleton, that is, only has to use them. Those classes are: `Setup` and `Solver`. The `Setup` class groups the configuration parameters of the skeleton. For example, this class specifies whether the search of the solution area will be a depth-first, a best-first or a breadth-first or also if the scheme must be the distributed or the shared memory, etc. The `Solver` class implements the strategy to follow, in this case, the Message Passing Skeleton, and maintains updated information of the state of the exploration during the execution. The execution is carried out through a call to the `run()` method. This class is specified by `Solver_Seq`, `Solver_Lan` and `Solver_SM` in the class hierarchy. To choose a given resolution pattern, the user must instantiate the corresponding class in the `main()` method. The implementation of the distributed memory skeleton uses `MPI_Send` and `MPI_Recv` to send and receive message respectively. The main loop in the Master and Slave codes are implemented using `MPI_IProbe`. When a message is received, its status is used to classify what kind of work should be done: finish, receive a problem for branching, receive a request of slaves, etc.

Table 1. Sunfire 6800. MPI implementation. The sequential average time is 1529.91 seconds.

Procs	Average	Min	Max	Speedup-Av	Speedup-Max	Speedup-Min
2	1560.15	1554.38	1569.08	0.98	0.98	0.98
3	1185.65	1177.28	1204.49	1.29	1.30	1.27
4	767.02	625.81	896.83	1.99	2.44	1.71
8	412.41	378.02	462.95	3.71	4.05	3.30
16	303.66	284.78	315.77	5.04	5.37	4.85
24	250.10	239.37	258.37	6.12	6.39	5.92

Table 2. Origin 3000. MPI implementation. The sequential average time in seconds is 867.50.

Procs	Average	Min	Max	Speedup-Av	Speedup-Max	Speedup-Min
2	933.07	932.67	933.80	0.93	0.93	0.93
3	743.65	740.73	745.29	1.17	1.17	1.16
4	454.77	399.86	492.43	1.91	2.17	1.76
8	251.82	236.94	266.96	3.44	3.66	3.25
16	186.18	174.24	192.80	4.66	4.98	4.50
24	152.49	144.29	167.92	5.69	6.01	5.17
32	151.09	144.69	166.46	5.74	6.00	5.21

3 Shared Memory Skeleton

OpenMP supports two kinds of worksharing constructs to speedup parallel programs: the *for* pragma and the *sections* pragma. Neither of these constructs can be utilized to parallelize access to list or tree structured data [9].

The algorithm proposed in this paper works with a global shared queue of tasks (*bqueue*) implemented using a linked data structure. For the implementation of the BnB shared memory resolution pattern we have followed a very simple scheme (see figure 3). First, the number of threads (*nt*) is calculated and established (lines 5-6 and 10). Then *nt* subproblems are removed from the queue and assigned to each thread (lines 7-9). In the parallel region from line 11 to 22 each assigned thread works in its own subproblem. The lower and upper bounds are calculated. The best solution value and the solution vector must be modified carefully, because only one thread can change the variable at any time. Next the sentences in lines 16 and 17 must be in a critical region. The same special care must be take into account when a thread tries to insert a new subproblem in the global shared queue (line 21).

The generic shared memory code outlined in figure 3 is instantiated for the OpenMP API as follows: The `omp_set_num_threads` is used to establish the number of threads. The `parallel for` pragma substitutes the generic parallel forall. Finally, the OpenMP pragma `critical` has been used for the implementation of the two critical regions.

4 Computational Results

The algorithm for the resolution of the classic Knapsack 0/1 problem described by Martelo and Toth [6] has been implemented using the Branch-and-Bound skeleton. In this section we analyze the experimental behavior for this problem on sets of randomly generated test problems. Since the difficulty of such problems is affected by the correlation between profits and weights, we considered the strongly correlated ones.

The experiments have been done on three different machines:

- Sunfire 6800 SMP, with the following configuration: 24 750 MHz UltraSPARC-III processors, 48 Gbyte of shared memory each and 120 Gbyte of disk space per system.
- Origin 3000, whose configuration is 160 600 MHz MIPS R14000 processors, 1 Gbyte of memory each and 900 Gbyte of disk.
- An heterogenous Cluster of PCs, which was configured with 2 750 MHz AMD Duron Processors, 4 800 MHz AMD Duron Processors, 7 500 MHz AMD-K6 3D processors, 256 Mbyte of memory each one and 32 Gbyte of hard disk each one.

The software used in the Sunfire 6800 SMP was the mpCC compiler of C++ and the UNICOS/mk tool. In the

Table 3. PCs cluster. MPI implementation. The sequential average time is 5,357.02 seconds.

Procs	Average	Min	Max	Speedup-Av	Speedup-Max	Speedup-Min
2	11,390.69	11,343.45	11,434.42	0.47	0.47	0.47
3	9,344.12	6,175.15	10,044.11	0.57	0.87	0.53
4	5,162.14	4,538.29	6,195.14	1.04	1.18	0.86
8	3,722.09	3,478.06	4,642.56	1.44	1.54	1.15
10	3,518.24	3,299.57	3,699.64	1.52	1.62	1.45

Origin 3000 was used the MPISpro CC compiler of C++ (version 7.3.1.2m) and IRIX MPI. Finally, in the PCs cluster the operating system was Debian Linux version 2.2.19 (herbert@gondolin), the C++ compiler was GNU gcc version 2.7.2.3 and the *mpich* version was 1.2.0.

Tables 1, 2 and 3 were obtained using MPI. The first and the second tables show the speedup results of five executions of the 0/1 Knapsack Problem randomly generated for size 100,000, while the third one was obtained through ten executions. Only the optimal value of the objective function is calculated. The solution vector where this value is obtained is omitted. The first column contains the number of processors. The second column shows the average time in seconds. The column labelled "Speedup-Av" presents the speedup for the average times. The third and fifth columns give the minimum times (seconds) and the associated speedup whereas the fourth and sixth columns show the maximum. Figure 4 shows the same results graphically.

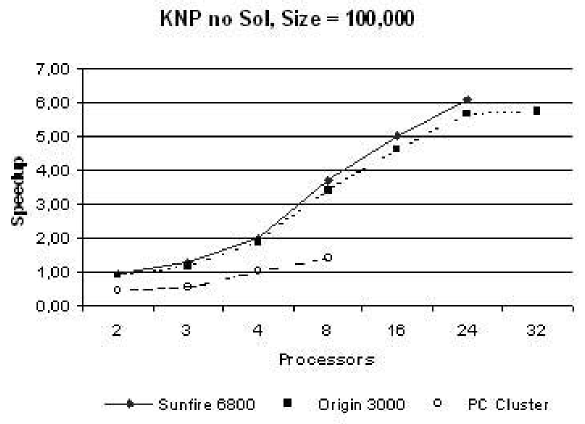

Figure 4. Speedups. No solution vector

Due to the fine grain of the 0/1 knapsack problem, there is no lineal increase in the speed up when the number of processors increase. For large numbers of processors the speed up is poor. For those cases there is still the advantage of being able to manage large size problems, than can not be solved sequentially. The limited performance for two processor systems is due to the Master/Slave scheme followed in the implementation. In this case, one of the processors is the Master and the others are the workers and furthermore, communications and work are not overlapped.

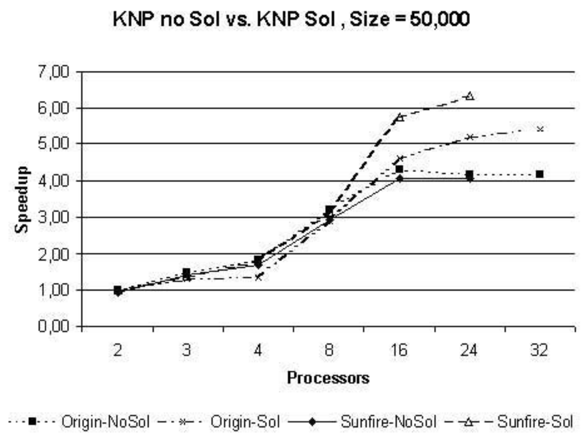

Figure 5. Speedups. With and without solution vector

Figure 5 shows the obtained results for the knapsack problem when the number of object is 50,000 and the solution vector is also calculated. The results of the executions in the Origin 3000 and the Sunfire 6800 are represented. As can be appreciated, when the solution is obtained the speedups are better because the problem to solve is harder, that is, is a coarse-grain problem. For the problems of size 100,000 appears memory allocations problems.

The table 4 shows the results for the problem using the OpenMP skeleton. The solution vector is not calculated in these experiments. Figure 6 compares the results obtained using the MPI and the OpenMP implementations without the solution vector. A similar behavior can be appreciated between both libraries. However, when the number of processors increase the speedup of the OpenMP version decrease while the MPI stays stable.

Table 4. Origin 3000. OpenMP implementation. The sequential average time in seconds is 869,76.

Procs	Average	Min	Max	Speedup-Av	Speedup-Max	Speedup-Min
2	747.85	743.98	751.02	1.16	1.17	1.16
3	530.74	517.88	540.76	1.64	1.68	1.61
4	417.62	411.02	422.59	2.08	2.12	2.06
8	235.10	231.26	237.00	3.70	3.76	3.67
16	177.72	162.00	207.15	4.89	5.37	4.20
24	157.30	151.18	175.67	5.53	5.75	4.95
32	176.43	174.23	179.01	4.93	4.99	4.86

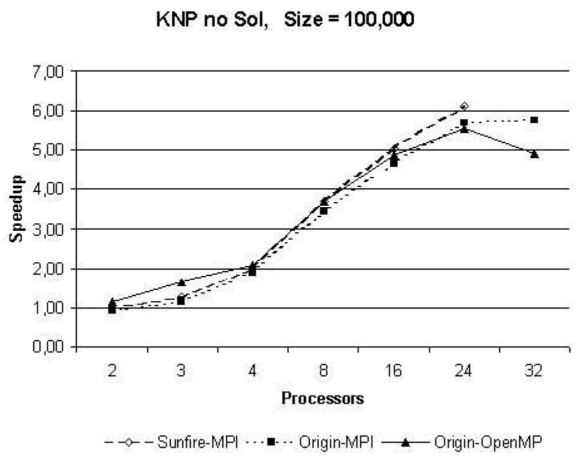

Figure 6. Comparison between MPI and OpenMP

5. Conclusions and Future Works

Two generic schemes for the resolution of problems by means of the Branch-and-Bound paradigm has been introduced in this paper. It has to be emphasized the high level programming offered to the user. By means of only one specification of the interface of his/her problem, obtains two parallel solvers: one using the message passing paradigm and other using the shared memory one. Another advantage is that the user does not need to have any expertise in parallelism.

The algorithm using shared memory is more compact and easier to understand than the message passing one. In this sense, we can conclude that the OpenMP paradigm is more manageable than the MPI one.

The obtained computational results suggest that for a large number of processors the proposed algorithms are not scalable. This is not a general trend in the Brach-and-Bound algorithms. In the case of the algorithms presented in this paper could be consequence of the overload of communications. One line of work to improve this deficiency is to parametrize the number of subproblems that is solved by each processor before doing any request. Another issue to improve is the memory management.

Studing the input data randomly generated for 0-1 knapsack problem, we have discovered that the number of visited nodes is in order n^2. Then one challenge is try to generate a problem with enough grain to have some increase in performance in the parallel case.

In the message passing algorithm presented all the information related with a subproblem is grouped in the same class, "SubProblem". In particular, the path of decisions taken up from the original problem down to that subproblem. All this information travels with the object whenever there is a communication. There is room for optimization regarding to this, since the subproblems share some common past. The natural solution necessarily implies the distribution of the decision vector among the different processors and the introduction of mechanisms to recover that distributed information.

Finally, an approach that could allow a better management of the computational resources, specially in hybrid share-distributed memory architecture, is to combine data and task parallelism. An algorithm that integrates OpenMP and MPI is in our agenda [11].

6. Acknowledgements

This work has been partially supported by the CICYT project TIC02-04498-C05-05. We also have received the support of the European Community-Access to Research Infrastructure action of the Improving Human Potential Programme contract No. HPRI-CT-1999-00026 (TRACS Program at EPCC-Edinburgh Parallel Computing Center). We also would like to thank the CIEMAT for allowing us the access to their computers.

References

[1] Anvik, J., MacDonald, S., Szafron, D., Schaeffer, J.,, Bromling, S. and Tan, K.: Generating Parallel Programs from the Wavefront Design Pattern. In Proceedings of the 7th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'02), Fort Lauderdale, Florida. (2002).

[2] Cole, M.: Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming, submitted to Parallel Computing, (2002)

[3] Eckstein, J., Phillips, C.A., Hart, W.E.: PICO: An Object-Oriented Framework for Parallel Branch and Bound. Rutcor Research Report (2000)

[4] Kuchen, H., A Skeleton Library. Euro-Par 2002, pp. 620-629, (2002).

[5] Le Cun, B., Roucairol, C., The PNN Team: BOB: a Unified Platform for Implementing Branch-and-Bound like Algorithms. Rapport de Recherche n.95/16 (1999)

[6] Martello, S., Toth, P.: Knapsack Problems: Algorithms and Computer Implementations. John Wiley & Sons Ltd, (1990)

[7] OpenMP: C and C++ Application Program Interface Version 1.0. http://www.openmp.org/, October (1998)

[8] Ralphs, T.K., Ladányi, L.: COIN-OR: Common Optimization Interface for Operations Research. COIN/BCP User's Manual. International Business Machines Corporation Report (2001)

[9] Shah S., Haab G., Petersen P., Throop J.: Flexible control structures for parallelism in OpenMP. Concurrency: Practice and Experience 12(12): 1219-1239 (2000)

[10] Shinano, Y., Higaki, M., Hirabayashi, R.: A Generalized Utility for Parallel Branch and Bound Algorithms. IEEE Computer Society Press, 392-401, (1995)

[11] Smith L., Bull, M.: Developmet of mixed mode MPI/OpenMP applications. Scientific Programming 9, 83-98, (2001)

[12] Snir M., Otto S. W., Huss-Lederman S.,Walker D. W. and Dongarra J.: MPI – the Complete Reference. 2nd Edition, MIT Press, (1998)

[13] Tschöke, S., Polzer, T.: Portable Parallel Branch-and-Bound Library. User Manual Library Version 2.0, Paderborn (1995)