Comparison between C, C++ and Java implementations of Branch-and-Bound Skeletons

Isabel Dorta, Olga Francisco, Coromoto León
Universidad de La Laguna, Dept. de Estadística, Investigación Operativa y Computación,
La Laguna, Spain, 38271
isadorta@ull.es, cleon@ull.es

Abstract

This article present skeletons to solve Optimization Problems using the Branch-and-Bound technique. The skeletons user is provided with the possibility to solve its problems, as much of sequential form as of parallel and distributed forms without having to modify its code. The skeleton has been implemented using three different programming languages: C, C++ and Java. The first part of our proposal compare the different languages implementations. The second part of our proposal consists of a comparison between the parallel and distributed tools to implement the parallel and distributed versions. An algorithm for the resolution of the classic 0-1 Knapsack Problem has been implemented using the three implementations of skeletons proposed. The parallel implementations have been made using MPI and Java Sockets. Some computational results of the comparison of the languages are presented.

Keywords: Branch-and-Bound Technique, Skeletons, Oriented Object Languages, Imperative Languages

1 INTRODUCTION

An Algorithmic Skeleton must be understand as a set of procedures that compose the structure to use in the development of programs for the resolution of a given problem using a particular algorithmic technique. They provide an important advantage by comparison to a direct implementation of the algorithm from the beginning, not only in terms of code reuse but also in methodology and concept clarity. In general, the software that supply skeletons presents declarations of empty classes. The user must fill these empty classes to adapt the given scheme for the resolution of a particular problem.

To solve combinatorial optimization problems, the best element of some finite set of feasible solutions has to be found. In general, all possibilities have to be explored in a search tree. The simplest technique consists of the enumeration of all possible solutions. However, other approaches to deal with the resolution of problems such as: Branch-and-Bound or Dynamic Programming have been developed. The Branch-and-Bound technique determines the optimum-cost solution of a problem through a selective exploration of a solution tree. The internal tree nodes correspond to different states of the solution search and the “leaves” correspond to feasible solutions.

Since research on Branch-and-Bound algorithms began, these algorithms have been considered one of the suitable tools for parallel processing. Several parallel libraries have been developed for this technique. PPBB (Portable Parallel Branch-and-Bound Library) [9] and PUBB (Parallelization Utility for Branch and Bound algorithms) [8] propose implementations in the C programming language. BOB [4], PICO (An Object-Oriented Framework for Parallel Branch-and-Bound) [3] and COIN (Common Optimization Interface for Optimization Research) [7] 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 this paper, we present three implementations to solve optimization problems using the branch-and-bound technique. They could be used on as much of networks of computers under Linux, like of personal computers multiprocessor. The libraries are generic and based on skeletons. Our skeletons have been carried out in the C, C++ and Java languages. The parallel implementations have been made using MPI (Message Passing Interface), the standard tool in message passing programming and the distributed implementation uses Java Sockets.

The main objective of our proposal is to compare the skeletons implemented using C, C++ and Java languages. An algorithm for the resolution of the classic 0-1 Knapsack Problem has been implemented using the three skeletons proposed. Branch-and-Bound Technique applied to solve the 0/1 Knapsack Problem is described in second section. The sequential implementations of the skeletons implemented are presented in third section. The parallel implementation of the skeletons is described in fourth section. Some computational results are presented in fifth section. Finally, in the sixth section the conclusions and the future work are presented.

2 DEFINITION OF BRANCH-AND-BOUND TECHNIQUE TO SOLVE O/1 KNAPSACK PROBLEM

The 0-1 Knapsack Problem has been studied extensively during the past decades. Several exact algorithms for its resolution can be found in the literature and for this reason appears in many real applications with practical importance. This problem is considered NP-hard.

Consider the classical 0-1 Knapsack Problem where a subset of N given items has to be introduced in a knapsack of capacity C. Each item has a profit p_i and a weight w_i and the problem is to select a subset of items whose total weight does not exceed C and whose total profit is a maximum. Assume that all input data are positive integers. Introducing the binary decision variables x_i, with $x_i = 1$ if item i is selected, and $x_i = 0$ otherwise, the problem can be formulated as follows:

$$\begin{array}{rc} & max \ \ \sum_{i=1}^{N} p_i x_i \\ subject \ \ to: & \sum_{i=1}^{N} w_i x_i \leq C \\ & x_i \in \{0,1\}, i \in \{1,...,N\} \end{array}$$

The Branch-and-Bound algorithm described by Martello and Toth [5] to solve this problem has been implemented using the C, C++ and Java skeletons. The algorithm requires the elements to be ordered in ascending order according to the weight/profit relation given by:

$$\frac{p_i}{w_i} \geq \frac{p_{i+1}}{w_{i+1}} \quad (i = 1,...,N-1)$$

The bounds used in the implementation are defined as follows:

The *lower_bound* is calculated by including objects in the knapsack until the maximum capacity is reached:

$$\sum_{i=k}^{s} w_i x_i \leq C; \quad x_i \in \{0,1\}, i,k,s \in \{1,...,N\}$$

$$lower_bound = \sum_{i=k}^{s} p_i x_i$$

In a similar way the *upper_bound* is calculated, but in this case a portion of the the last object considered is included to fit the capacity.

$$C_r = \sum_{i=k}^{s} w_i x_i \leq C; \qquad x_i \in \{0,1\}, i,k,s \in \{1,...,N\}$$

$$upper_bound = \sum_{i=k}^{s} p_i x_i + \frac{p_{s+1} \times (C - C_r)}{w_{s+1}}$$

As an example [5] we consider a knapsack with a capacity $C = 102$ and the number of objects $N = 8$ with the following weights and benefits:

$$\begin{array}{rcl} p_k & = & \{15,100,90,60,40,15,10,1\} \\ w_k & = & \{2,20,20,30,40,30,60,10\} \end{array}$$

Figure 1 shows the trace of the algorithm for the resolution of the Knapsack Problem in this example. To solve the problem by Branch-and-Bound a tree in whose root the value of none of the x_i is fixed, and where in each successive level the value of a farther variable is determined by numerical order of the variables has

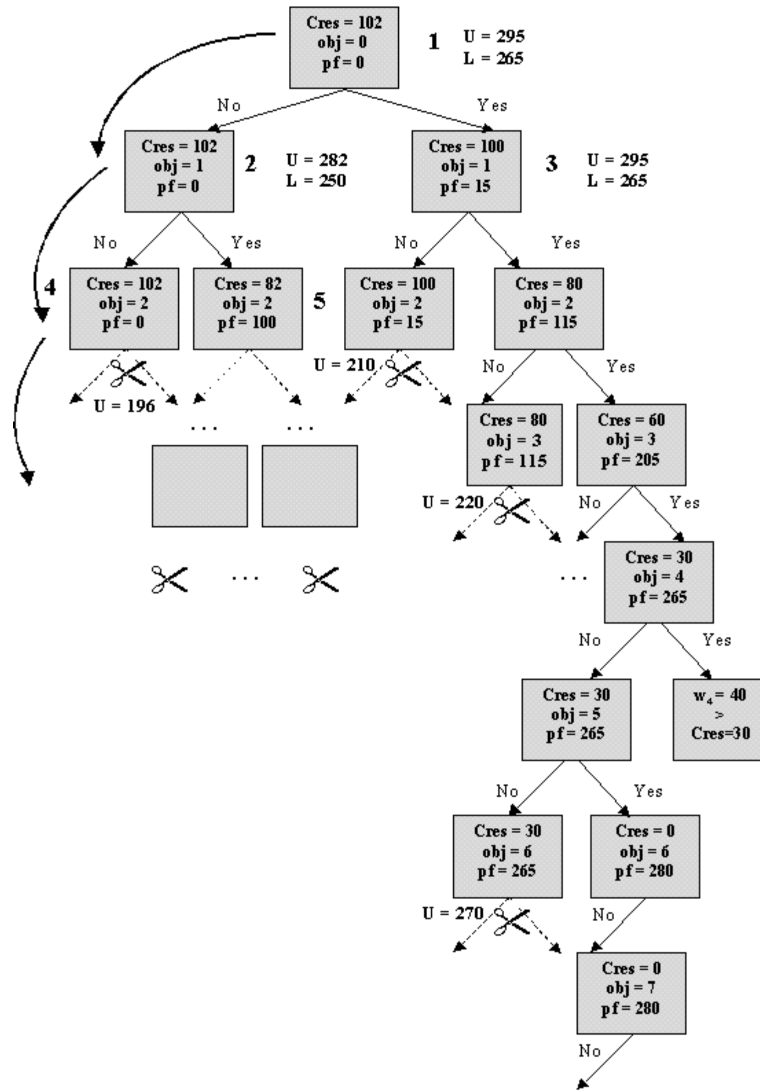

Figure 1: Search Tree of an Example of Knapsack Problem

to be explored. Each node that is explored produces two farther nodes, depending on whether the following object in the knapsack is introduced or not. If a node is generated, the upper bound and a lower bound of the solution value are calculated, which can be obtained completing the partially specified capacity, and these bounds are used to cut useless branches and to guide the exploration of the tree. The optimum solution vector is $x = (1, 1, 1, 1, 0, 1, 0, 0)$ and the value of the objective function is $z = 280$.

A Branch-and-Bound algorithm always converges in finite steps regardless of the search function. However, its efficiency and the required storage space are highly dependent on the search function. We introduce the following important search functions used in practice and their characteristics:

- *Breadth-first search* expands the search space on a level by level basis. A priority function that always chooses to expand the problem with the smallest depth value will lead to a breadth-first expansion of the search space. An advantage of a breadth-first search is that it guarantees to find the solution of minimum depth in the tree. In a problem where solutions may be found at different depths this may be one of the optimization criteria.

- *Depth-first search* attempts to generate a solution by performing search levels down the tree. After expanding a given problem, the algorithm attempts to expand one of its newly generated offsprings. One way of implementing this is to always select the problem-state of greatest depth in the tree for

expansion. An advantage of a depth-first search is that it often has the smallest storage requirements of any search strategy. This strategy also attempts to generate an initial solution quickly so that this can be used for pruning sections of the tree.

In general terms, it can be said that the breadth-first searches explore the nodes in the same order in which they are created, it means that they use a queue (FIFO list) to store the nodes that have been generated but have not been examined yet. The depth-first searches explore the nodes in inverse order to that of their creation, employing a stack (LIFO list) to store the nodes that are being generated but have not been examined yet.

3 SKELETONS USER INTERFACES

In the MaLLBa skeletons, different users appear (see figure 2): *Skeleton programmer* is the person who designs and implements the skeleton. He or she decides the set of classes the skeleton offers and requires. *Skeleton filler* is the person who adapts the real problem and the heuristic to the skeleton classes filling the details left by the skeletons programmer. *Final user* is the person who uses the fulfilled skeletons to get an approximated solution for an instance.

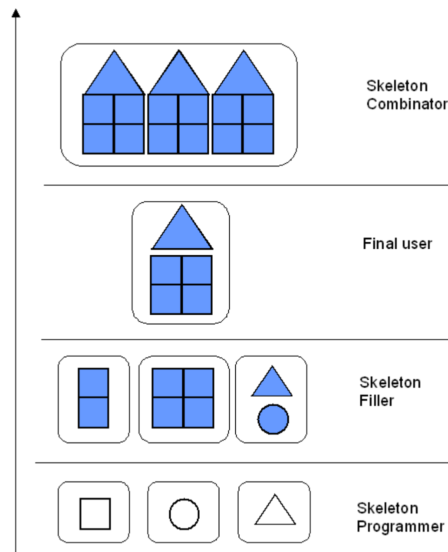

Figure 2: Skeleton Users

The skeletons are specially target at two different objectives: provide the final user with a friendly environment sharing their common knowledge, and simplify the programming of the skeleton filler. In a MaLLBa skeleton two principal parts are distinguished: 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* and that will be used by the resolution pattern. There is an intuitive relationship between the participating entities in the resolution pattern and the classes which will be implemented by the user.

The part provided by the skeleton, that is, the resolution pattern, is implemented through classes. These classes are denominated *provided classes* and appear in the code with the qualifier *provides*. The part which the user fills in with their particular problem is implemented through classes labelled with the qualifier *requires*, and will be named *required* classes.

The adjustment that has been accomplished by the user consists of two steps. First, the problem has to be represented through data structures and then, using them, the user has to implement the required functionalities of the classes. These functionalities will be invoked from the particular resolution pattern (because the interface for such classes is known) so that, when the application has been completed, the expected functionalities applied to the particular problem are obtained. Figure 3 shows a UML scheme of MaLLBa for the Branch-and-Bound technique.

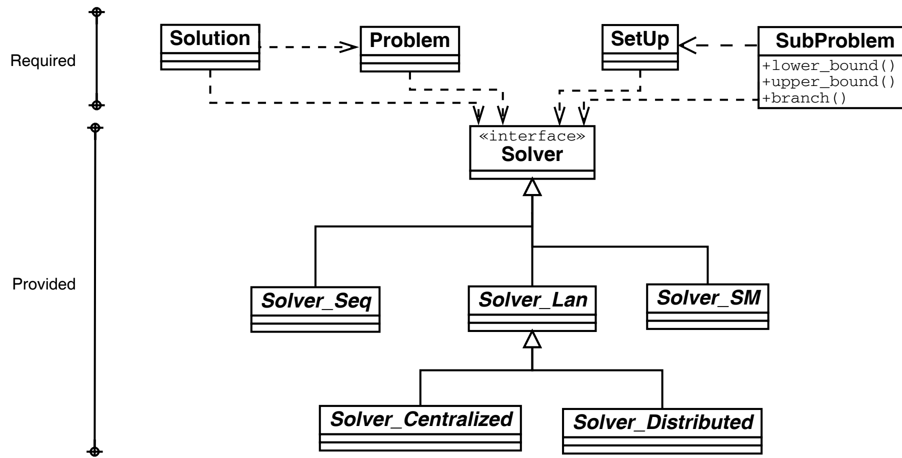

Figure 3: UML Scheme of MaLLBa::BnB

```
void lowerUpper (info spb, double L, U) {
   number i, weig, prof;
   if (spb.CRest < 0) {
      L = -INFINITY; U = -INFINITY;
   }
   else {
      i = spb.obj; weig = 0;
      prof = spb.profit;
      while ( (i<N) && (weig <= spb.CRest) ) {
         if ( w[i] <= (spb.CRest-weig) ) {
            weig += w[i]; prof += p[i];
         }
         i++;
      }
      L = prof;     /* lower bound */
      for (i = spb.obj, weig = 0, prof = spb.profit; i < N, weig <= spb.CRest; i++) {
         weig += w[i]; prof += p[i];
      }
      if (i != spb.obj) {
         i--; weig -= w[i];  prof -= p[i];
      }
      if ((i == spb.obj) && (spb.obj == (N-1)))
         U = prof;  /* upper bound */
      else
         U = prof + (p[i]*(spb.CRest - weig))/w[i];
}  }
```

Figure 4: Bound method of C skeleton

4 SEQUENTIAL IMPLEMENTATIONS

The following figures show the sequential iterative C code used to solve the 0-1 Knapsack Problem through Branch-and-Bound algorithm for a maximization problem. The bound function *lowerUpper* is defined in Figure 4. We use the same function to calculate the lower and upper bounds. The lower bound is defined as the maximum benefit that can be obtained from a given subproblem, while the upper bound includes the proportional part of the benefit of the last object that could not be inserted into the knapsack. The *branch* function that studies the insertion of the object k or considers not inserting it is presented in Figure 5. The function *knap* is presented in Figure 6. It proceeds by repeating the test of live subproblems. Departing from a subproblem *sp* the use of the Branch-and-Bound technique is considered to find the best solution *bestSol*. The first subproblem is extracted from the queue, its upper and lower bounds are calculated and supposing it improves the bestSol, it is updated. It continues with the process of extracting problems from

```
void branch (info spb ) {
   info spNO, spYES;
   number newC, newPf;
   number nextObject = spb.obj + 1;
   if (nextObject <= N ) {
      spNO.obj = nextObject;
      spNO.CRest = spb.CRest;
      spNO.profit = spb.profit;
      insert(spNO);
      newC = spb.CRest - w[spb.obj];
      if (newC >= 0) {
         newPf = spb.profit + p[spb.obj];
         spYES.obj = nextObject;
         spYES.CRest = newC;
         spYES.profit = newPf;
         insert(spYES);
}  }  }
```

Figure 5: branch method of C skeleton.

```
number knap() {
   double L, U;
   info sp;
   while (!empty()) {
      sp = extract();
      lowerUpper (sp, L, U);
      if (bestSol < L) bestSol = L;
      if (bestSol < U) branch(sp);
   }
   return (bestSol);
}
```

Figure 6: knap method of C skeleton.

the queue and proving the bestSol until the queue is empty. If the problem is still unresolved, it is branched and the branch function inserts the new subproblems into the queue. The execution of the C skeleton is carried out in the *main()* function presented in Figure 7. The function knap returns the value of the best solution founded.

Figure 8 shows the Java sequential skeleton implemented using a breadth-first search. The Java Branch-and-Bound Skeleton requires the following classes: *Problem*, *Solution* and *Subproblem*. The class *Problem* corresponds to the definition of a problem instance; the class *Solution* corresponds to the definition of a solution (feasible or not) of a problem instance and the class *Subproblem* is a new abstraction that represents the not explored solution area. This class requires the following methods:

- *lower_bound()*: Given a problem and a subproblem, this function computes a lower bound of the best objective function value that could be obtained for a given subproblem.
- *upper_bound()*: Given a problem and a subproblem, this function computes an upper bound of the best objective function value that could be obtained for a given subproblem.
- *branch()*: Given a problem, a solution and a subproblem, makes a partition of the current solution area. This implies to make a decision about the current subproblem that will generate a set of subproblem instances to be explored. This set of subproblems generated should be inserted in the data structure *queue* using the method *insert*.
- *solve()*: return the best solution obtained until that moment, it means, the solution with the value returned by *lower_bound()*.

We are interested to know the behavior of the algorithm. Concretely, we would like to know the number of nodes generated of the search tree and the number of these nodes that are visited to find the best solution.

```
int main (int argc, char ** argv) {
   number sol;
   readKnap(data);
   spini.obj = 0;
   spini.CRest = M;
   spini.profit = 0;
   insert(spini);
   bs = knap();
   printf("bestSol lf", bs);
   return 1;
}
```

Figure 7: main method of C skeleton.

```
public class SolverSeqQueue {
  LinkedList spList;
  long Visitednodes = 0, Generatednodes = 0;

  public SolverSeqQueue(){
    spList = new LinkedList();
  }

  public Solution run(Problem pbm) {
    double bestActual = -1, upper, lower;
    SubProblem actualsp, anteriorsp;
    Solution sol = null;
    spList.add (pbm.generateSubProblem());
    while (!spList.isEmpty()) {
        actualsp = (SubProblem) spList.getFirst();
        Generatednodes++;
        if ((upper = actualsp.upper_bound(pbm)) > bestActual) {
           Visitednodes++;
           if ((lower = actualsp.lower_bound(pbm)) > bestActual) {
              bestActual = lower;
              sol = actualsp.solve(pbm);
           }
           if (upper != lower)
              actualsp.branch(pbm, this);
        }
        spList.removeFirst();
    }
    System.out.println("Generated Nodes: " + Generatednodes + "Visited Nodes: " + Visitednodes );
    return sol;
} }
```

Figure 8: Java Sequential Skeleton.

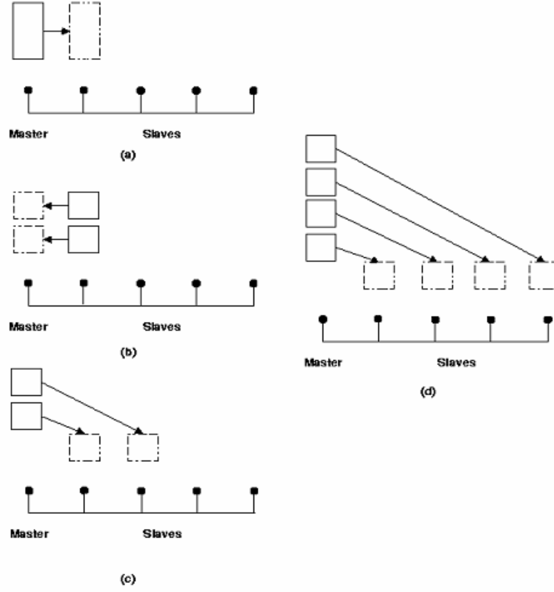

Figure 9: Phases of the Master-Slave Paradigm

For this reason, we have added two new variables in the provided class *SolverSeqQueue* to store the number of generated nodes *Generatednodes* and the number of visited nodes *Visitednodes*. The final value of these variables will be known when finish the method *run()* of class SolverSeqQueue. The number of generated nodes is increased when a subproblem is extracted from the data structure, in this case a queue. The number of visited nodes is increased in the case of the upper bound is not going to prune the actual subproblem being studied.

5 PARALLEL IMPLEMENTATIONS

To parallelize our skeletons for the Branch-and-Bound technique we have implemented a master-slave strategy. These parallel implementations are composed of the following basic processes:

- Master process: There is only one master process in the skeletons. It contains all the information about the states space and the status of each slave process, e.d., busy or idle.
- Slave process: the number of slave processes is specified in an initiation parameter. Each *slave process* performs all the computations to solve/evaluate a subproblem.

Figure 9 shows the Master and the Slaves tasks [1, 2]. The generation of new subproblems and the evaluation of the results of each of them are completely separate from the individual processing of each subtask. The Master is responsible for the coordination between subtasks. The Master has a data structure which registers the occupational state of each slave; at the beginning all the slaves are idle. The Master assigns the first subproblem to the first idle slave in phase *a*. While there are idle slaves the Master receives information from them and decides the next action to apply depending on whether the problem is solved, or there is a slaves request or whether the slave is idle. In phase *b*, the slave branches the subproblem into two subproblems and asks the Master for help. When the master receives a slave request from a slave, it is followed with the upper bound value. 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 the problem, as it is represented in phase *c*. Otherwise, the answer indicates that it is not necessary to work in this subtree. If no free slaves are available, the slave continues working locally. Otherwise, it removes subproblems from its local queue and sends them directly to other slaves, phase *d*. When the number of idle slaves is equal to the initial value, the search process finishes, then the Master notifies the slaves to finish work.

	FIFO			LIFO		
Size	Time	Generated_Nodes	Visited_Nodes	Time	Generated_Nodes	Visited_Nodes
500	0,043	783	393	0,147	1752	1422
1000	0,878	6282	3446	2,948	14840	13680
2000	4,728	13512	11647	4,722	13512	11647
3000	8,127	15966	9043	10,458	18879	16611
4000	61,609	81631	57537	457,518	577613	569333
5000	25,643	35781	17996	33,343	36432	32939
10000	64,329	37886	22108	-	-	-

Table 1: FIFO vs LIFO using Java Skeleton

	Time in seconds	
Number of objects	C++ Skeleton	Java Skeletons
500	0,049	0,043
1.000	0,825	0,878
2.000	4,579	4,728
3.000	6,755	8,127
4.000	58,421	61,609
5.000	23,954	25,643
10.000	54,286	64,329

Table 2: C++ vs Java using FIFO Structure

The implementation of the skeletons use *MPI_Send* and *MPI_Recv*, to send and receive messages respectively [6]. 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 bounding and branching, receive a request from slaves, etc.

6 COMPUTATIONAL RESULTS

We accomplished a study of the time required by the sequential and parallel Branch-and-Bound Skeletons using the C, C++ and Java languages.

The experiments have been done on two different clusters:

- An heterogeneous 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 and 32 Gbyte of hard disk each. This cluster of PCs belongs to the Parallel Computing Group of La Laguna University.
- IBM CLX/768 (IBM Linux Cluster), 384 2 PE per node Intel Xeon Pentium IV 3 GHz 512 KB cache, 788 GB of memory each, and 5.5 TB of hard disk. This cluster belongs to CINECA.

The software used in the PCs cluster the operating system was Debian Linux version 2.2.19 (herbert@gondolin). The C and C++ compilers were GNU gcc version 2.7.2.3 and the *mpich* version was 1.2.0, and the compiler of Java was J2SE 5.0. The software used in the CLX/768 was the mpiCC compiler of C++ and the mpicc compiler of C based on the MPICH-GM version of MPI (myrinet enabled MPI).

Table 1 shows the times in seconds obtained executing the Java implementation when a FIFO and a LIFO data structure is used to solve the Knapsack Problem for number of objects between [500,10.000]. Also, this table shows the number of generated nodes and visited nodes when both data structures are used.

Table 2 presents the comparison between C++ and Java languages for different sizes of the knapsack problem for the sequential case using FIFO data structure; these problems were generated for sizes between [500,10.000]. The algorithm computes the best solution and the solution vector in all cases.

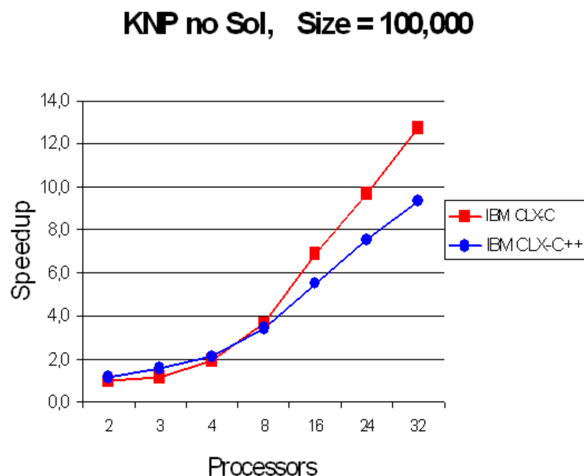

Figure 10: Comparison between Parallel C++ and C Implementations for number of objects 100000

Figure 10 shows the obtained results for the knapsack problem when the number of objects is 100,000 and the solution vector is not calculated. The results of the executions in the CLX/768 for C and C++ implementations are presented.

7 CONCLUSIONS

Three skeletons for the resolution of problems by means of the Branch-and-Bound paradigm has been introduced in this paper. The chosen languages to develop the skeletons have been C, C++ and Java. The high level programming offered has to be emphasized. The schemes used can be easily transformed to solve other combinatorial optimization problems. The skeletons provide a sequential solver and a parallel solver without additional user effort. To the advantages of the oriented object (OO) methodology provides: modularity, re-usable, modifiable, and so on, is added the interpretation facility of the skeleton, mainly because there is quite an intuitive relationship between the entities participating in the resolution pattern and the classes implemented by the skeleton. Also, the language Java offers greater advantages for the development of the components software associated with the geographical distribution.

The results of a comparison of times, number of nodes generated and number of nodes visited for the Java Skeletons using a breadth-first and depth-first search have been presented. In the case of depth-first search, size problems larger of 10.000 objects have not be solved because of memory problems. It can be appreciated that the time needed for the algorithm to solve the problem is related to the number of objects generated and visited.

A comparison of the time required for the C++ and Java sequential skeletons to solve different sizes of the knapsack Problems is presented. The results show C++ sequential skeleton is slightly more efficient than Java sequential skeleton. On the other hand, the results obtained using the C++ parallel skeleton are less efficient than the results obtained using the C parallel skeleton.

A comparison between the parallel implementation using MPI and distributed implementation using Java Sockets is in our agenda.

8 Acknowledgements

This work has been supported by the European Community-Research Infrastructure Action under the FP6 "Structuring the European Research Area" Programme, under the Project HPC-EUROPA (RII3-CT-2003-506079). We also would like to thank to the CICYT projects TIC02-04498-C05-05 by the support received.

References

[1] Dorta, I., León, C., Rodríguez, C., and Rojas, A., *Parallel Skeletons for Divide-and-Conquer and Branch-and-Bound Techniques*. Proceeding of the Eleventh Euromicro Conference on Parallel, Distributed and Network Based Processing. IEEE Computer Society, pages 292–298, 2003.

[2] Dorta, I., León, C., Rodríguez, C., and Rojas, A., *MPI and OpenMP implementations of Branch-and-Bound Skeletons*, Advances in Parallel Computing, Elsevier, vol 13, pp. 185-192, 2004.

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

[4] Lecun, B., and Roucairol, C., *Bob : Branch and bound optimization library*, In INFORMS, Institute For OR and Management Science, 1995.

[5] Martello, S. and Toth, P., *Knapsack Problems Algorithms and Computer Implementations*, John Wiley & Sons Ltd., 1990.

[6] Message Passing Interface Forum. *MPI: A message-passing interface standard*, International Journal of Supercomputing, Applications and High Performance Computing, 8:3–4, 1994.

[7] Ralphs, T.K. and Ladányi, L., *SYMPHONY: A Parallel Framework for Branch, Cut and Price*, http://www.branchandcut.org/SYMPHO-NY, 2000.

[8] Shinano, Y., Higaki, M., and Hirabayashi, R., *A Generalized Utility for Parallel Branch and Bound Algorithms*, Proc. of the 7nd IEEE Symposium on Parallel and Distributed Processing (SPDP'95), IEEE Computer Society Press, pages 392–401, 1995.

[9] Tschöke, S., and Polzer, T., *Portable Parallel Branch-and-Bound Library*, User manual, Technical report D-33095, University of Paderborn, http://www.uni-paderborn.de/ ppbb-lib, 1995.