Parallel Branch and Bound Algorithm - A comparison between serial, OpenMP and MPI implementations

Parallel Branch and Bound Algorithm - A comparison between serial, OpenMP and MPI implementations

Lucio Barreto and Michael Bauer

Department of Computer Science

Middlesex College - The University of Western Ontario – London, ON – Canada

{lbarret6, bauer}@csd.uwo.ca

Abstract. This paper presents a comparison of an extended version of the regular Branch and Bound algorithm previously implemented in serial with a new parallel implementation, using both MPI (distributed memory parallel model) and OpenMP (shared memory parallel model). The branch-and-bound algorithm is an enumerative optimization technique, where finding a solution to a mixed integer programming (MIP) problem is based on the construction of a tree where nodes represent candidate problems and branches represent the new restrictions to be considered. Through this tree all integer solutions of the feasible region of the problem are listed explicitly or implicitly ensuring that all the optimal solutions will be found. A common approach to solve such problems is to convert sub-problems of the mixed integer problem to linear programming problems, thereby eliminating some of the integer constraints, and then trying to solve that problem using an existing linear program approach. The paper describes the general branch and bound algorithm used and provides details on the implementation and the results of the comparison.

1. Introduction

Integer Programming problems (IP) are special cases of optimization problems where the variables can only assume integer values. Mixed Integer Programming problems (MIP) are special cases where only some of the variables are restricted to integer values. Optimization problems with integer variables can also be linear or nonlinear, depending on the terms of their objective function and their constraints. However, in general, the terms IP and MIP are almost always associated with problems that have linear features. The optimization algorithms considered in this work focus on mixed integer programming, where some variables assume integer values and while other variables take on continuous values.

In considering only problems with linear constraints and linear objective functions, there are important differences between IP and MIP in relation to Linear Programming (LP):

- LP – there are necessary and sufficient conditions of optimality proved theoretically that can be used to efficiently test whether a given feasible solution is an optimal solution. These conditions are used to develop algebraic methods such as the simplex method.
- IP/MIP – there are no known optimality conditions to test whether a given feasible solution is optimal. It is necessary to perform the implicit or explicit comparison of all feasible solutions of the problem.

Approaches to solve IP and MIP problems are numerous, usually specialized in accordance with a particular application. We can divide these approaches into two broad families, with distinct characteristics:

- Heuristic Optimization – may get good results for problems in which classical optimization methods might fail (such as many integer variables and very complex constraints), but there is no guarantee of optimality. Examples of such approaches are genetic algorithms [5][8] and tabu search [3][4].
- Classical Optimization – for convex problems it is guaranteed that the solution is optimal. In this family there are the methods of enumeration, such as the zero-one enumeration algorithm [1] and the branch-and-bound algorithm [2], used in this work.

We focus specifically on using a branch-and-bound approach for finding an optimal solution to MIP problems, including being able to find all optimal solutions when there are multiple ones. The branch-and-bound algorithm is an enumerative optimization technique, where finding a solution to an MIP problem is based on the construction of a tree where nodes represent candidate problems (sub-problems) and branches represent the new restrictions to be considered. Through this tree, all integer solutions of the feasible region of the problem are listed explicitly or implicitly ensuring that all the optimal solutions will be found.

A common approach to solve such problems is to convert the sub-problems that arise in solving integer linear programming problems to linear programming problems, that is, with no integer constraints, and to solve those problems using an existing linear programming solver. The resulting solution can be used to set bounds on the possible solutions or to help select variables to create restrictions that can be used to create sub-problems. The paper presents a comparison of a branch and bound algorithm implemented with three different approaches: serial, shared memory model (OpenMP) and distributed memory model (MPI). The implemented branch and bound algorithm uses the GNU/GLPK [9] as the Linear Programming (LP) solver to find optimal solutions to mixed integer linear problems.

The reminder of the paper is organized as follows. Section 2 describes some important theoretical aspects of branch-and-bound. Section 3 describes the branch-and-bound algorithm itself. Section 4 describes the computational implementation. Section 5 shows a comparison of the approaches. Section 6 presents conclusions about this work and finally Section 7 identifies future work.

2. Theoretical Aspects

The branch-and-bound algorithm is an enumerative technique, in which a solution is found based on the construction of a tree in which nodes represent the problem candidates and branches represent the new restrictions to be considered. Through this tree, all integer solutions of the problem feasible region are listed explicitly or implicitly ensuring that all the optimal solutions will be found.

The overall structure of the branch-and-bound algorithm has three key elements, separation, relaxation and pruning [2]. Separation uses the tactic of “divide and conquer” in order to solve the problem (P). In order to find the solution of P, it is decomposed into two or more descendant sub-problems, generating a list of candidate problems (CP). At a subsequent step in the algorithm, a candidate is selected from the list of candidate problems and the algorithm tries to solve that problem. If a solution cannot be found to that problem, that problem is again decomposed and its descendants are added to the list of candidate problems. If the selected problem can be solved, then a new solution is obtained. The objective function value of this new solution is then compared with the value of the incumbent solution, which is the best feasible solution known so far. If the new solution is better than the incumbent solution, it becomes the new incumbent. Then, the algorithm returns to the list and selects the next candidate. This procedure is repeated until the list is empty, and the solution of the problem is taken as the final incumbent solution.

The usual way to carry out separation of an integer programming problem is through contradictory constraints in a single integer variable (separation variable or branching variable). Thus, from the

original problem (called node zero), two new descendant sub-problems are created, which are easier to solve than the original one, since a constraint was added to the separation variable. Each generated node has an associated candidate sub-problem and each branch indicates the addition of a constraint related to the variable used in the separation. Therefore, as the algorithm moves down in the tree, the viable region of the generated descendants becomes more restricted.

The most common approach to relaxation is the elimination of the integral constraints (integer variables) where the integer or mixed integer problem is converted to a general LP. Relaxation assumes that the original integer variables may take fractional values and then the resulting LP problem is solved. The obtained optimal solution usually has several variables with non-integer values. Among these variables, one must be selected for separation.

Once the separation and inclusion of new descendants in candidates list is completed, the algorithm must select from among the stored candidates the next to be evaluated and, if necessary, separated successively until the linear problem solution becomes integer, or unfeasible or worse than the incumbent, meaning that the candidate sub-problem can be removed from the list (pruned), not producing any more descendants. This procedure repeats until the list of candidates' sub-problems becomes empty.

3. Branch and bound Algorithm

Consider the MIP (P) whose general form is given by:

$$
(P)\quad
\begin{aligned}
&\text{minimize} && z = \mathbf{c}^{\mathrm{T}}\mathbf{x} && (1)\\
&\text{subject to} && \mathbf{A}\mathbf{x} \le \mathbf{b} && (2)\\
& && x_i \text{ integer } \forall i \in I &&\\
& && x_j \text{ real } \forall j \in J &&
\end{aligned}
$$

Where **c** is the vector of costs, **x** is the vector of variables integers (x_i) and real (x_j), **A** is the constraint coefficient matrix, **b** is the vector of right side of the inequalities, I is the set of integer variables and J is the set of continuous variables. The branch-and-bound algorithm to solve (P) has the following steps:

1. **Start-up**: set the number of active nodes $n = 0$, set the first incumbent and initialize the list of candidate sub-problems with the original problem (P).
2. **Convergence test**: if the candidate list is empty then this means that the process is over and the current incumbent solution is the optimal solution of the problem. Otherwise, continue.
3. **Candidate selection**: among the candidate sub-problems not yet pruned, choose the one that will be the next to be evaluated and remove it from the list. Solve the LP problem related to the relaxed selected problem $\left(CP_R^k\right)$ and store the optimal solution as a lower bound for all its descendants, $z_{\inf}^k = z^*\left(CP_R^k\right)$.
4. **Pruning tests**: the sub-problem $\left(CP^k\right)$ may be pruned if it meets one of the following conditions:
   a) if (CP_R^k) has no feasible solution;
   b) if $z_{\inf}^k > z^*$, where z^* is the actual incumbent value;
   c) if the optimal solution of (CP_R^k) is integer and feasible in $\left(CP^k\right)$. In this case, if the optimal value is lower than the incumbent value, make $z^* = z_{\inf}^k$ and apply the previous test (b) for all candidates' sub-problems not yet pruned.

   If the candidate sub-problem $\left(CP^k\right)$ was pruned, return to the Step 2.

5. **Separation**: from the sub-problem $\left(CP^k\right)$, select a variable for separation from those that are integer and still have continuous value. For the chosen variable x_i, whose current value is x_i^*, generate two new descendant sub-problems and add them to the candidate list. The new sub-problems are generated by adding to $\left(CP^k\right)$ the following restrictions:

$$(PC^{n+1}): x_i \le \left\lfloor x_i^* \right\rfloor \tag{3}$$

$$(PC^{n+2}): x_i \ge \left\lfloor x_i^* \right\rfloor + 1 \tag{4}$$

where $\left\lfloor x_i^* \right\rfloor$ is the largest integer not greater than x_i^*. Set $n = n + 2$ and return to Step 3.

3.1. Performance Enhancement

The algorithm efficiency is directly related to the method of selecting the next candidate sub-problem that will be evaluated (Step 3) and the separation variable (Step 5). Moreover, the existence of a good initial incumbent increases the effectiveness of the pruning test (b), thereby reducing the number of candidate sub-problems that need to be evaluated.

Although there is no systematic technique to determine which one of the candidate sub-problems leads more quickly to the solution, some heuristic rules can be used, such as search by creation order, by depth, by the relaxed solution value, by the solution estimated value, among others.

A very common technique used to select the candidate sub-problem is the LIFO rule (Last In, First Out), which produces a depth search. This strategy allows that the descendant sub-problem be solved from the previous problem (because they differ in only by a single variable) and minimizes the memory requirements to store the candidates' information. The way which these selections are made directly influences the number of nodes that need to be evaluated, determining, thus, the computational effort.

On the other hand, there are methods that use estimates of the objective function value to select from among all candidates the most promising sub-problem. This rule minimizes the total number of problems to be evaluated, but at the same time can drastically increase the memory requirements, since the successive LP problems may not have the same similarity that exists in the depth search.

3.2. Variable Selection

When the solution of the relaxed candidate (PC_R^j) has several integer variables with continuous value, one of them must be selected to be separated. An inadequate variable choice implies evaluation of several descendants' sub-problems that could be eliminated by pruning their predecessor. There is no systematic technique to identify the optimal separation variable, but there are empirical rules that indicate which variables are most attractive. If simplicity is the goal, the separation variable can be selected from a predetermined sequence using the objective function coefficients, highest to lowest cost, for example, or being more specific depending on the problem's features. A more effective alternative is based on the search for the variable that has the highest value in terms of the estimated increase of the objective function, which can be obtained from one of the following techniques:

[MAX;MAX]: always choose the variable which causes the greatest degradation in the objective function in order to quickly obtain a descendant sub-problem that might be pruned. For the candidate sub-problem node k, the selection method for the separation variable j follows the expression below:

$$\max\left\{\max(P_i^- f_i^j, P_i^+ (1 - f_i^j))\right\}$$

[MAX;MIN]: choose the variable which the minimum variation causes the maximum impact. This ensures that both descendants contribute to pruning the sub-problem. For the candidate sub-problem node k, the selection method for the separation variable j follows the expression below:

$$\max\{\min(P_i^- f_i^j, P_i^+(1-f_i^j))\}$$

[MIN;MAX]: choose the variable which the maximum variation causes the minimum impact. Similarly, both descendants contribute to pruning this sub-problem. For the candidate sub-problem node k, the selection method for the separation variable j follows the expression below:

$$\min\{\max(P_i^- f_i^j, P_i^+(1-f_i^j))\}$$

[MIN;MIN]: always choose the variable which causes the lowest degradation in the objective function in order to quickly obtain a viable solution. This is the most conservative method. For the candidate sub-problem node k, the selection method for the separation variable j follows the expression below:

$$\min\{\min(P_i^- f_i^j, P_i^+(1-f_i^j))\}$$

Another issue involved in variable selection is the possibility of separating a variable which was assigned an integer value. It often occurs in a branch-and-bound execution that a specific variable gets an integer value prematurely, i.e., without any limits to impose that integer value. In this case, according to Step 5 of the branch-and-bound algorithm, this variable would not be selected for separation (because it is integer), and if all others variables were also integers, the solution would be considered viable, not generating descendants. This would generate a fail in order to guarantee the reaching of all alternative solutions to the problem. By separating the variables with integer values which still have a lower bound minor than its upper bound, the algorithm guarantees that all integer variables involved in the problem will be evaluated in an implicit or explicit way, exhausting all possible separation opportunities and allowing that all alternative solutions to the problem may be found.

3.3. Multiple Optimal Solutions

Some problems may have multiple optimal integer solutions, such as optimization models that describe the power networks expansion problem for both transmission [12] and distribution [6][7]. In this case, obtaining all the solutions is very important. Although the objective function value is the same for all solutions, one solution may be greater than the other when taking into account other factors that were not explicitly included in the optimization model. If these other factors have some relevance, one may also want to retain sub-optimal solutions. This can be easily accomplished when solutions found are stored in an ordered list, as used in this work.

To ensure that all solutions are found, it is necessary to make changes in the pruning (Step 4) and separation (Step 5) processes. In Step 4 (c), one solution is considered integer when all the integer variables cannot assume other values, i.e., when they all have inequality constraints with upper and lower bounds both integers and identical. Thus, the Step 4 (c) of the conventional algorithm should be replaced by:

c) if the optimal solution of (CP_R^k) is feasible in $\left(CP^k\right)$ and all integer variables in the optimal solution of (CP_R^k) have no degree of freedom – i.e. $\mathbf{x}_k^{\text{lb}} = \mathbf{x}_k^{\text{ub}}$, $\forall k \in I$, where $\mathbf{x}_k^{\text{lb}}$ and $\mathbf{x}_k^{\text{ub}}$ are the lower and the upper bound vectors of integer variables of candidate problem (CP_R^k), respectively. In this case, if the optimal value is lower than the incumbent value, make

$z^* = z_{\text{inf}}^k$ and apply the previous test (b) for all candidates' sub-problems not pruned yet. Otherwise, if the optimal value is equal to the incumbent value, store the new alternative solution.

Notice that the components of vectors $\mathbf{x}_k^{\text{lb}}$ and $\mathbf{x}_k^{\text{ub}}$ are always integer values generated by the separation process, according to equations (3) and (4). In Step 5, in addition to using the variables that have continuous values, it is also necessary to consider separating the integer variables that have some degree of freedom, i.e., with $x_i^{\text{lb}} < x_i^{\text{ub}}$.

4. Computational Implementation

The branch-and-bound algorithm was implemented in C++ and was adapted to three different versions: one serial and two parallel. The two parallel implementations respectively used OpenMP [11] and MPI [10]. The parallel versions are based on the serial approach, just making use of the necessary parallel calls. However, all approaches use the same programming logic.

In the system, several parameters may be configured, such as:

- maximum size of the branch-and-bound tree;
- different tolerances for integer and equality;
- selection criteria of the separation variable;
- criteria to select a node;
- minimum depth to open the tree[1];
- occupation percentage of the available space in the tree to exchange sort order of the active nodes in order to preserve available memory;
- possibility of separation in integer variables.

The central data structure used in the branch-and-bound algorithm stores all the information that characterizes each candidate sub-problem (active node). The information associated with each node of the solution tree is divided into two groups:

- active nodes – information related to the candidate sub-problems;
- tree nodes – information of the predecessors nodes that were separated (using a contradictory constraint) generating the active nodes.

In this work, each node stores only the information relating to the last separation; the rest of the information is associated with the tree nodes. This procedure avoids data redundancy.

The candidate sub-problems, when generated, are inserted into the lists of active nodes; several lists are maintained depending on a particular sort criterion. The available ordering possibilities on active nodes are as follows:

- node number (LIFO, Last In First Out);
- node depth;
- node relaxed solution (LP value);
- node estimated solution (using pseudo-costs);
- node estimated solution weighted with depth (deepest nodes are more important).

[1] Before adopting the chosen strategy for selecting the candidate node, the program will expand the tree breadth-first, until all active nodes have the same depth as the minimum chosen in the configuration.

Since all lists are circularly-linked, the algorithm can select the node that has any of the extreme values (minimum or maximum, in each of the criteria) directly. Moreover, as all lists are simultaneously updated, it is possible to switch between any criteria during the solution process without the need for reordering, which have high computational cost.

Whenever a candidate node is pruned (satisfying some of the criteria presented in Step 4 of the algorithm in Section 3) it is removed from the list of active nodes, together with all its predecessors that do not possess active descendants, recursively, from the node that was pruned towards the source node of the tree.

As an example, consider the problem P:

$$(P)\begin{cases} \min & v = 5n_{12} + 2n_{13} + 2n_{23} + \beta \\ s.a. & 350n_{12} + 400n_{13} + \beta \geq 400 \\ & 350n_{12} + 210n_{23} + \beta \geq 200 \\ & \beta \geq 0 \\ & n_{ij} \geq 0 \quad \forall ij \\ & n_{12}, n_{13} \text{ e } n_{23} \text{ integers} \end{cases}$$

The branch-and-bound algorithm for this problem can be represented by Figure 1, where each node is represented for a circle with a number that indicates its generation's order (index *i* of the algorithm). In the side of each node there is the relaxed solution and if it that solution is "integer" or worse than the incumbent, it is related to a pruned node, represented by a colourful circle.

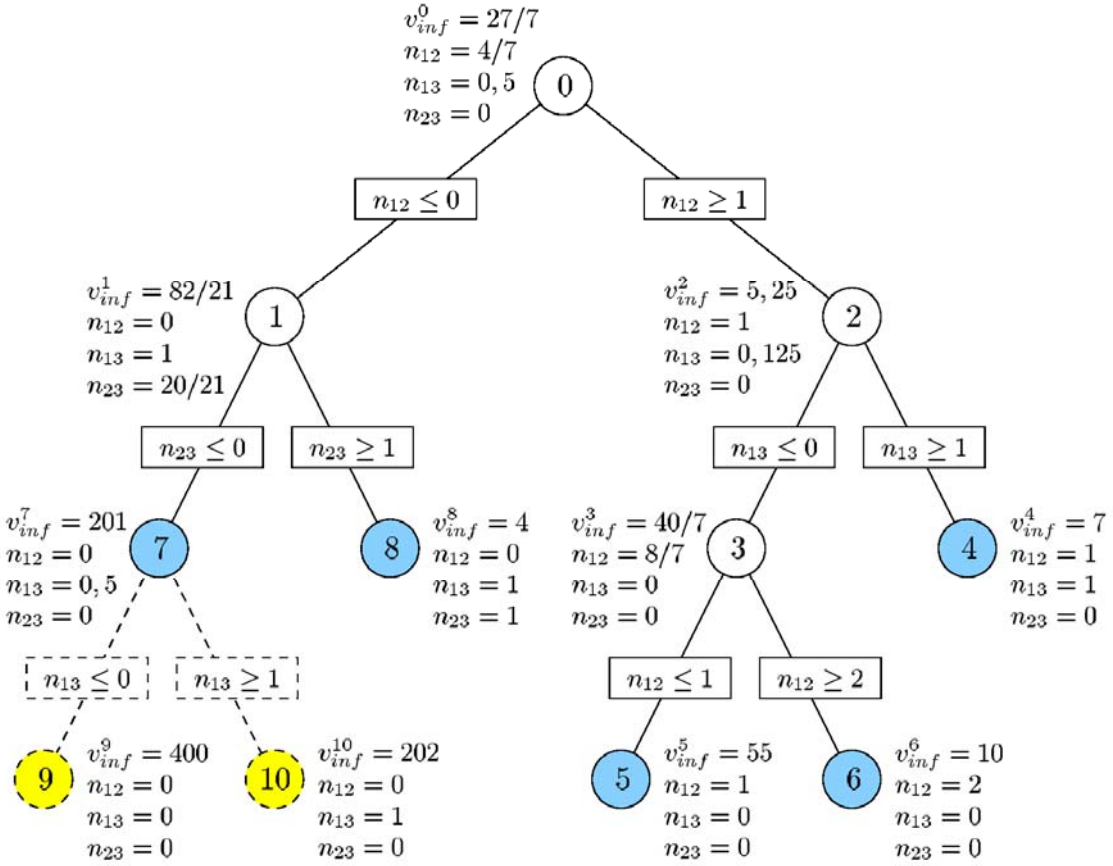

Figure 1. Branch-and-bound intermediate tree.

The constraints that are added for each separation are shown inside the rectangles in the respective branch. Thus, for any node, the associated problem can be determined going back on the tree towards the root node (node 0) and adding the correspondent constraints for that path. For example, the candidate problem represented by node 7 is the original problem *(P)* with the addition of the following constraints:

$$n_{12} \le 0$$
$$n_{23} \le 0$$

The tree starts with Node 0 through the linear problem resolution (P_R), which is obtained from the relaxation of the integer constraints of the expression 1.

$$(P_R)\ \begin{cases} \min\ v = 5n_{12} + 2n_{13} + 2n_{23} + \beta \\ s.a.\ \ 350n_{12} + 400n_{13} + \beta \ge 400 \\ \qquad 350n_{12} + 210n_{23} + \beta \ge 200 \\ \qquad \beta \ge 0 \\ \qquad n_{ij} \ge 0 \quad \forall ij \end{cases}$$

The linear problem solution presents two variables with continuous values: $n_{12} = 4/7$ and $n_{13} = 0.5$. Thus, it does not satisfy the pruning criterion and the process continues with the separation of the problem. The first non-integer variable (n_{12}) is chosen, as the separation variable. In this case, the two descendants' nodes are generated by the constraints $n_{12} \le 0$ (Node 1) and $n_{12} \ge 1$ (Node 2). In this part of the process there are two candidates that may be chosen to be evaluated, represented by Nodes 1 and 2. Supposing that the selecting criterion is to evaluate the last generated node. So, Node 2 will be the chosen one. The equivalent relaxed problem solution – relaxed problem with the constraint $n_{12} \ge 1$ – still has a variable with continuous value ($n_{13} = 0.125$). Then, the problem of the Node 2 should be separated and, in this case, only one variable can be selected, (n_{13}).

After the generation of the Node 2 descendants (Nodes 3 and 4), Node 4 is selected and evaluated. Thus, the first integer solution for the original problem is obtained and this solution becomes the incumbent ($v^* = v^4 = v^4_{\text{inf}} = 7$). As the incumbent value was changed, the remained candidates (Nodes 1 and 3) are evaluated, but it is not possible to make any pruning. Again, the last generated candidate is selected (Node 3) and its variable n_{12} is separated. Both descendants of the Node 3 (Nodes 5 and 6) have integer solutions that are discarded since they are worse than the existing incumbent ($v^5 = v^5_{\text{inf}} = 55$ and $v^6 = v^6_{\text{inf}} = 10$).

The algorithm now goes back to its candidates' list and selects the only available node (Node 1) which is separated in its variable n_{23} generating Nodes 7 and 8. Taking the last one to be evaluated, we obtained an integer solution better than the incumbent, which is updated to $v^* = v^8 = v^8_{\text{inf}} = 4$. Next, Node 7 is evaluated and discarded (with its descendants) since its lower bound is greater than the incumbent ($v^7_{\text{inf}} = 201 > v^*$). Nodes 9 and 10 are represented only to complete the enumeration.

In the way that the process evolved, it was necessary to solve 9 linear problems to guarantee the enumeration of all possible alternatives. The optimal solution for the original problem was obtained after 8 LPs:

$$v^* = 4$$
$$n_{12}^* = 0$$
$$n_{13}^* = 1$$
$$n_{23}^* = 1$$

An important issue regarding performance in enumeration algorithms is related to how good the initial incumbent solution is. A good initial solution significantly increases the efficiency of the pruning test (4b) and so, reduces the number of candidates that need to be evaluated. For the example of Figure 1, an initial incumbent $v^* = 5$, would be enough to immediately prune Node 2 (and all its descendants), reducing then the total number of evaluations.

Considering the data structures used and the criteria for node ordering and selection of the separation variable, the serial branch-and-bound algorithm was implemented according to the flowcharts of Figures 2 and 3. A summary of the parallel implementation can be seen in Figure 4.

4.1. Parallel Implementation

4.1.1. MPI

In the MPI implementation, the master node initializes the program, creates a base tree with a list of active nodes and sends those active nodes to the slaves whenever they need a problem to work on. The master also controls the process, including list of solutions, statistics, etc.

While the master is creating a base tree, the slaves also create a copy of the same base tree. Each slave receives an index for a node from the master, then initializes its own tree and starts the B&B. All the calculation is done by the slaves. Once a solution is found, it is sent to the master who updates all slaves.

The master node waits for any message from the slaves and it does no other calculations during the B&B process. Whenever a slave finishes its job, it sends a message to the master asking for a new node and process continues.

As the slaves work in different nodes with their own tree, there is no possibility of conflict in accessing the same part of the memory.

4.1.2. OPENMP

In the OpenMP implementation, there is no concept of master and slaves. The cpu0 node is the one responsible for initializing the program and creating a common tree with a list of active nodes. In this implementation, the nodes have a flag indicating whether or not they are being used. After this creation, all nodes start the B&B requesting the first available node from the list of active nodes. So, each slave works in different part of the same tree. Of course, as the memory is shared, once something happens, the variables are automatically updated.

With both implementations there are possibilities for further improvements and those will be discussed in the Sections 6 and 7.

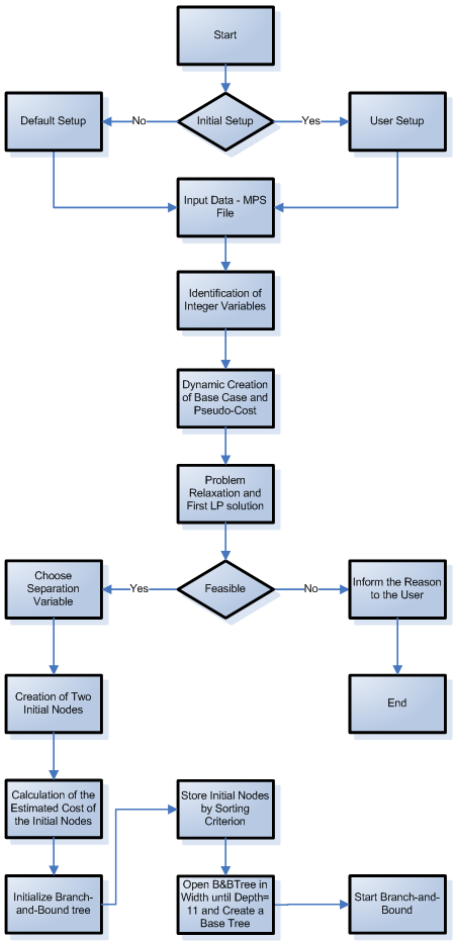

Figure 2. Serial branch-and-bound algorithm – Part 1

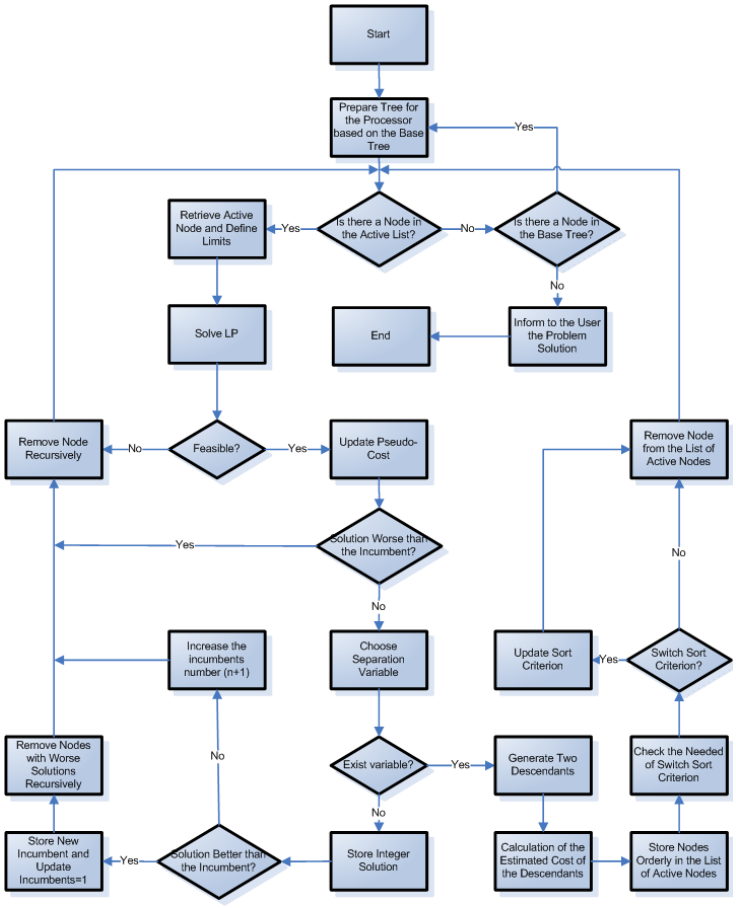

Figure 3. Serial branch-and-bound algorithm – Part 2.

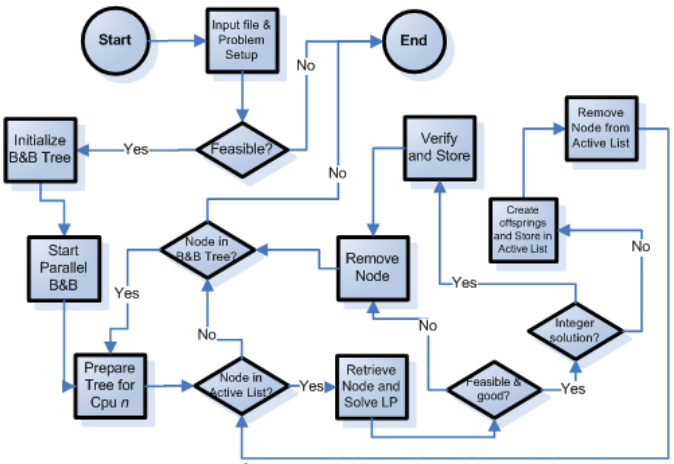

Figure 4. Summary of the parallel branch-and-bound algorithm

In the next section we present results comparing the three implementations for solving the same MIP.

5. Results

The initial comparison of the algorithms to date made use of an optimization problem previously presented [6][7]. The problem is a simplified three-phase electrical distribution network consisting of 18 nodes (2 substations and 16 nodes with loads) and 24 branches operating under 13800V. The topology of this network is shown in Figure 5 in which rectangles denote the substations and circles are the nodes where loads are concentrated. Branches drawn as continuous lines denote the initial network (those with a single line are part of the fixed network and those with double lines are candidates for replacement), and branches drawn as dashed lines are candidates for addition (and are not part of the initial network). The search space has approximately 2^{118} combinations. The results are illustrated in Figure 6. The problem is fully detailed in the above reference.

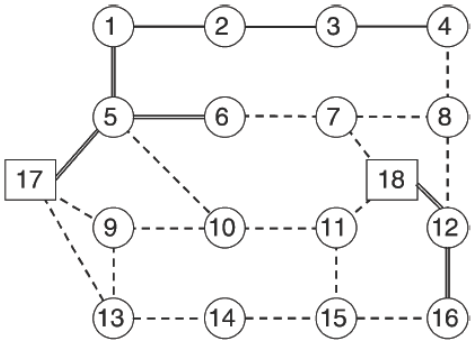

Figure 5: Diagram of the 18-node network. [7]

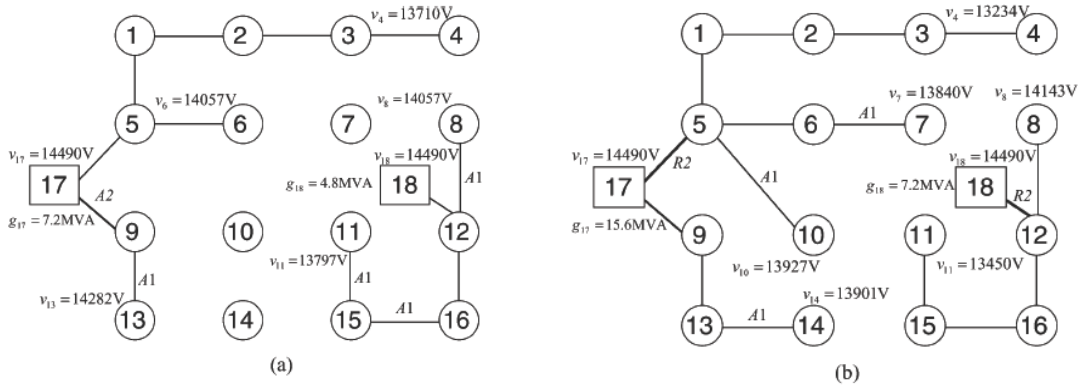

Figure 6: 18-node network - Solution with multistage expansion. [7]

The results for the tests were carried out on two different systems:

- HP Linux cluster running XC 3.1 with 267 nodes, 4 cores each, connected via Myrinet 2g (narwhal.sharcnet.ca) for the MPI and Serial Implementation.
- SUSE Enterprise 10 Linux Cluster running with 128 nodes, 1 core per socket, NUMA (silky.sharcnet.ca) for the OpenMP.

Even though these are different systems with different architectures, the goal of the experiment was to compare the wall time of the parallel implementations with the serial one to support the idea that classic optimization methods can still be used in modern distributed systems for many different areas.

Table 2: Results for the Electrical Distribution Network – Serial Approach

#LPs	Wall Time (s)
70240	7324

Table 3: Results for the Electrical Distribution Network – Parallel Approaches

#Proc	#LPs		Wall Time (s)		Speedup	
	MPI	OpenMP	MPI	OpenMP	MPI	OpenMP
2	91230	65321	6155	6654	1.2x	1.1x
5	86745	47652	4623	6236	1.6x	1.2x
10	84451	39567	2757	5712	2.7x	1.3x
50	72344	26541	976	3854	7.5x	1.9x
100	65645	21454	806	3562	9.1x	2.1x

Table 2 summarizes the execution of the solution to the problem for the serial implementation. The results show the required number of linear programming problems (sub-problems) (#LPs) and the total time required in seconds. The time was measured by the algorithm and corresponds to the total wall clock time.

Table 3 summarizes the execution for both parallel implementations: MPI and OpenMP. The results show the number of processors used (#Proc), the required number of linear programming problems (sub-problems) (#LPs) for each approach related to #Proc, as well as the total time spent across all processors and the calculated speedup. For the tests we used the [MAX;MAX] separation variable selection strategy.

The difference between the number of LPs in the parallel executions is not related to a different strategy, but is due to several other factors. In the MPI implementation, as the memory is distributed and the processors have to exchange messages, each node is creating its own tree and because of that it could take more time to find a good incumbent solution. Many unnecessary nodes are evaluated before finding a good one, since the algorithm has to ensure the enumeration of all possible solutions without pruning anything that could be promising. On the other hand, in the OpenMP approach, fewer nodes were evaluated because the algorithm always found good solutions earlier than the MPI (for this particular problem). Also, the OpenMP implementation is more similar to the serial one. Unfortunately, other issues with the OpenMP implementation were found and are reported in the conclusions.

6. Conclusions

Based on the initial experimental results some conclusions can be drawn:

- The serial implementation works reasonably well.

- The OpenMP implementation always generated fewer sub-problems than the serial, but the speedup was not as good as the MPI implementation. Unfortunately, the chosen LP solver (GLPK) [9] was not "thread safe", which made it impossible to run a shared memory model approach without the use of some explicit mechanisms of control in the implementation. This drastically decreased the performance. Also, the use of a flag in order to control the list of active nodes to avoid race condition has a cost that decreases performance, as well. We plan to address those problems in future work.
- The MPI implementation seemed to provide a more robust way to split the branch and bound search. Though it generated more sub-problems for some cases, it took the least time in all executions and always ran perfectly. It is important to mention here that before the parallel computation starts, the algorithm prepares a base tree for the MPI implementation and the nodes are sent to the slaves whenever they need. This base tree is initially static, which means that there are a specific number of nodes available to be evaluated. When this number is reached, no more nodes are available to be sent and the allocated processors become idle, waiting the others to finish the job. This is a problem that will be addressed in future work.

As indicated, these results are based on our particular problem and, while complex, additional experiments need to be run to better compare the algorithms and approaches to parallel implementation as well as to help improve the implementation. Our initial impression is that the MPI approach works reasonable well and with much less headache than the OpenMP one.

7. Future Work

Our next steps include finding a better way to evaluate nodes in parallel and finding a better way to keep all processors working full time. We also want to test the algorithm using others LP solvers that are thread-safe and faster. Finally, we intend to compare the implementations to several other problems including some from other domains, testing the different separation variable choice that we had implemented as well.

References

[1] Balas, E. (1965). "An additive algorithm for solving linear programs with zero-one variables", Operations Research, Vol. 13, No. 4, pp. 517–546.
[2] Geoffrion, A.M. and R.E. Marsten (1972). "Integer Programming Algorithms: A Framework and State-of-the-Art Survey", Management Science, Vol. 18, No. 9, pp. 465–491.
[3] Glover, F. (1989). "Tabu Search – Part I", ORSA Journal on Computing, Vol. 1, No. 3, pp. 190–206.
[4] Glover, F. (1990). "Tabu Search – Part II", ORSA Journal on Computing, Vol. 2, No. 1, pp. 4–32.
[5] Goldberg, D.E. (1989). "Genetic Algorithms in Search, Optimization and Machine Learning", Addison-Wesley Professional, 432 p.
[6] Haffner, S., L.F. Pereira, L.A. Pereira and L. Barreto (2008a). "Multistage model for distribution expansion planning with distributed generation - Part I: problem formulation". IEEE Transactions on Power Delivery, Vol. 23, No. 2, pp. 915–923.
[7] Haffner, S., L.F. Pereira, L.A. Pereira and L. Barreto (2008b). "Multistage model for distribution expansion planning with distributed generation - Part II: numerical results". IEEE Transactions on Power Delivery, Vol. 23, No. 2, pp. 924–929.
[8] Holland, J.H. (1992). "Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence", The MIT Press, 228 p.
[9] Makhorin, A., (2001). "GLPK Linear Programming Kit Manual" GLPK documentation,

Moscow Aviation Institute, Moscow, Russia, February 2001, available in http://www.gnu.org/software/glpk/glpk.html.

[10] MPI-forum - http://www.mpi-forum.org/

[11] OpenMP - http://www.openmp.org

[12] Romero, R., Monticelli, A., Garcia, A. e Haffner, S. (2002). Test systems and mathematical models for transmission network expansion planning, IEE Proc.-Gener. Transm Distrib., Vol. 1491, No. 1, pp. 27-36.