Second International Workshop on Parallel Processing for Artificial Intelligence (PPAI-93), 29. August 1993, Chambery, France.
(to be published by Elsevier in 1994)

Initialization of Parallel Branch-and-bound Algorithms

Dominik Henrich

Institute for Real-Time Computer Systems and Robotics, Computer Science Department University of Karlsruhe, Kaiserstrasse 12, D-76128 Karlsruhe, Germany, e-mail: dhenrich@ira.uka.de

Abstract

Four different initialization methods for parallel Branch-and-bound algorithms are described and compared with reference to several criteria. A formal analysis of their idle times and efficiency follows. It indicates that the efficiency of three methods depends on the branching factor of the search tree. Furthermore, the fourth method offers the best efficiency of the overall algorithm when a centralized OPEN set is used. Experimental results by a PRAM simulation support these statements.

Keywords: parallel processing, static load balancing, optimization, combinatorial algorithms, idle times, efficiency

1. Introduction

Branch-and-bound (B&B) is a well-known and general combinatorial optimization technique which is used especially for NP-complete problems where no special purpose algorithm can be developed. Because of the high problem complexity, parallel implementations are required for speeding up the computations. Algorithms with a high and stable efficiency, i.e., utilization of the processing elements (PEs), are desired when variable number of processors is used.

For comparisons, *speed-up* is a commonly used measure. It compares the run time of the algorithm when different numbers of PEs are used. Here, the speed-up is computed by dividing the run time of the parallel algorithm with one PE by the run time of the same parallel algorithm with multiple PEs. The *efficiency* is the speed-up divided by the number of PEs used in the parallel case. It is a measure for the average utilization of PEs during the total run time. High efficiencies, close to one, are desired with different numbers of usable PEs. They indicate a good exploitation of the available parallelism.

A brief description of the parallel B&B algorithm is given in the next section. Thereby, a typical problem of parallel processing search trees is formulated. Different initialization methods for solving this problem are stated in Section 3. For these methods, the occurring idle times and the resulting efficiency are analysed formally in Section 4. Finally, in Section 5, the analysis is verified by experimental results.

2. Parallel Branch-and-bound

The discrete combinatorial optimization problem is to find a vector $x = (x_1, ..., x_n)$, $x_i \in \mathbb{N}$, which minimizes a criterion function $f(x)$. In addition, a set of constraints is given which has to be met by the solution vector. This set can be subdivided into explicit and implicit constraints. The implicit constraints describe the relationships between the variables x_i , the explicit ones are the value ranges of the variables. The set of vectors fulfilling the explicit constraints define the search space which will be represented here as a search tree.

For describing the parallel B&B algorithm solving this optimization problem, we use a common version of the sequential B&B formulation as a basis. In Fig. 1, a formulation in Pascal is stated in ref. [12]. Thereby, the initialization is hidden behind a function call and is discussed in detail in the following sections.

There are two main data structures on which the single PEs are working. First, a set of nodes, called OPEN, stores the partial problems not yet investigated. Second, the incumbent z holds the best known solution so far. Both of these data structures may be kept centralized with access of all PEs or may be distributed over the PEs. In case of a distributed OPEN set, the set is divided into subsets which are then located on the single PEs. The kind of implementation is adapted to the underlying computer architecture. In this paper, we assume a distributed OPEN set, except for the parallel random access machine (PRAM).

With these data structures, the parallel B&B works quite similarly to the serial algorithm. All the PEs execute the main loop of the B&B in parallel, synchronously or asynchronously. Each PE grabs a node from an OPEN set, expands it, and inserts the evaluated successors back into OPEN. If necessary, the incumbent is updated.

Usually, a non-optimal solution is computed by a quick heuristic algorithm before the B&B computes the optimum. This first solution serves as upper bound for the cost estimation of the intermediate nodes. If the

```
procedure Branch_and_bound(root :node);
set   OPEN;            /* set for unsolved nodes */
node  x, y,            /* x, y nodes of search tree
*/
      z;           /* z incumbent* /
begin
      if solution(root) then z := root;
      else z := ∅;                  /* f(z) = ∞ *)
      Init_B&B(OPEN, root);
      while OPEN contains node x with g(x)<f(z) do
            x := select(OPEN);      /* h(x) = min */
            for all successors y of x do begin
                  if solution(y) and f(y)<f(z) then
                        z := y;
                  if not leaf(y) and g(y)<f(z) then
                        OPEN := OPEN + {y};
            end;
      end;
      if solution(z) then return(z);
      else return(failure);
end;
```

Fig. 1: B&B algorithm in Pascal notation.

lower bound of a generated successor is evaluated to be worse than the costs of the heuristic solution, then this successor may be pruned.

In contrast to the serial algorithm, three different phases of processing can be identified in the parallel B&B. These are the start-up phase, the working phase, and the shut-down phase. In the *start-up phase*, the B&B starts with the root node exploring the search tree. Down to a certain depth of the search tree, there are fewer nodes than PEs. Therefore, some of the PEs are idle waiting to receive a node. In the *working phase*, OPEN stores many nodes and all processors are supplied with work. In the *shut-down phase*, some PEs are idle again because only few branches of the search tree are left to be investigated and therefore, compared with the number of PEs, nodes are lacking.

One problem arising in parallel B&B algorithms is typical for parallel processing search trees, especially when they are built up during processing. If we assume that a synchronous parallelization at node level, not all PEs can be used from the very beginning. The search tree has to be explored node by node, and the amount of work which has to be done increases with each iteration. Starting with one initial node (the root node of the search tree), in the next iteration the number of nodes to be considered is multiplied by the branching factor b of the tree. The number of nodes of the search tree, and therefore, the number of active PEs can increase at most by b^d, where d is the current depth. The efficiency of the total algorithm is reduced, since not all PEs are used from the beginning.

The reduction of the efficiency by this kind of initialization is a problem-inherent feature. Furthermore, it cannot be compensated for, though full PE utilization is reached after several iterations. This problem even occurs with an idealized parallel computer: the PRAM. With real parallel computers, the problem is worse due to the communication necessary to distribute the generated nodes equally over the PEs. At least for these three reasons, the problem of Initialization is worth being investigated.

3. Initialization Methods

In this section, four different variants for the initialization of the parallel B&B algorithm are described and analysed. Generally, B&B initialization covers the task of static load balancing. The aim is to provide each PE with a certain amount of work. Ideally, the total (future) work to be done is distributed equally over the PEs. Otherwise, additional effort for dynamic load balancing slows the algorithm down. But in tree search, the future work is hardly to predict without any domain knowledge. Therefore, about one node is provided for each PE (static load sharing).

3.1. Root Initialization

Root initialization is the most common approach when parallel B&B is described. This may be due to the fact that (1) the initialization is not the main topic of the description, (2) a computational model was used which ignores the communication between the PEs (e.g., PRAM), (3) or the initialization has directly been taken over from the serial implementation of B&B.

When applying Root Initialization, the root of the B&B-tree is inserted in one of the local OPEN-subsets. On each PE, the main loop of the B&B is running. Thus, the PE with the root node inserted grabs this node and expands it. The successors of the root node are then distributed according the load balancing scheme applied in the algorithm. The other PE receiving a node proceeds in the same way. After a while, the total OPEN set provides at least one node for eachPE and the whole B&B goes from the start-up phase into the working phase.

It is assumed that the B&B is working on a search tree with a constant branching factor. Thus, all PEs can receive at least one tree node after a certain amount of time. This amount depends on the number of used PEs in a logarithmic way. Processing a search tree with a branching factor b using p PEs, all PEs can receive one node in at least $\log_b p$ iterations. Additionally, the concrete term depends on the use load balancing mechanism.

This initialization has the advantage, that the load balancing method of the main algorithm is used in the start-up phase, too. Thus, the Root Initialization is very simple to implement and no additional code is necessary. On the other hand, a relatively cost-intensive distribution of the successor nodes is used. Each distribution is combined with a necessary communication overhead. Furthermore, many of the PEs are idle and waiting to receive a node.

3.2. *Enumerative Initialization*

The second initialization method is very similar to the previous one. It is used by the algorithms in refs. [16, 10]. The main difference is the broadcast of the root node to every PE at the beginning of the initialization. Thereafter, each PE has the same node in its OPEN-subset. This root node is expanded by each PE according to the main loop of the sequential algorithm. Altogether, the p PEs proceed until at least p nodes rest in the OPEN-subset of each PE. Then the i-th PE keeps the i-th node of the set and deletes the rest. With this node, the PE continues in the local B&B algorithm and the total B&B process changes from the start-up phase into the working phase. In summary, the nodes necessary for initialization of each PE are quasi enumerated by all the PEs.

Additionally, there is a strategy necessary to preserve the correctness of the algorithm. This problem occurs if more nodes than PEs available are generated in the last step of the start-up phase. In this case, the above initialization would discard the unprocessed nodes which could possibly lead to the only optimal solution. An admissible method works in a round robin fashion and distributes the unprocessed nodes over the PEs according to their identifier.

With Enumerative Initialization, the number of iterations in the start-up phase is similar to the Root Initialization. For generating at least p nodes, about $\log_b p$ steps are necessary if a constant branching factor b is assumed. Although no improvement could be achieved in the number of iterations, the Enumerative Initialization has advantages. If the search tree is expanded by all PEs in the start-up phase, every PE computes its "local root" node without any further communication. All the PEs are working from the very beginning and are not waiting to receive a node. On the other hand, there is redundant work because many PEs are processing the same nodes and doing identical evaluation.

3.3. *Selective Initialization*

The Selective Initialization is a circular method first introduced by ref. [2] and used by ref. [7] later on. It prevents the disadvantages of the Enumeration Initialization by a local node selection strategy of the single PEs.

This method starts, analogously to the Enumerative Initialization, with broadcasting the root node over all the PEs. But instead of generating the whole search tree down to a certain depth, each PE generates only one single path. The paths of the PEs are distinct and they lead to the proper node for initializing the PE. To generate this path, each PE cycles in an expansion-selection loop for a certain time. Thereby, the active node (the root node at the beginning) is expanded and its successors generated. Instead of storing all the successors in the OPEN-subset, only one node of the successors is investigated in the sequel.

The selection of this node is done by a circular method, see ref. [2]. Thereby, every PE which participates at the B&B algorithm has a unique identification number *pu_id*. The number of current PEs taking the same path up to now is indicated by *ncp*. Assume that *size* indicates the number of successor nodes which have been generated, then the PE *pu_id* chooses the node denoted by *g* with

$$g = ((pu\_id - 1)\,\mathrm{MOD}\, size) + 1 \tag{1}$$

In the next iteration, the variables are updated by the following equations[1]:

$$delta = \begin{cases} 1, & g \le ncp\,\mathrm{MOD}\, size \\ 0, & \text{otherwise} \end{cases} \tag{2}$$

$$pu\_id = \lceil pu\_id / size \rceil \tag{3}$$

$$ncp = \lfloor ncp / size \rfloor + delta \tag{4}$$

The loop ends, when *size* becomes greater or equal to *ncp*. Then each PE takes a number of successor equal to $\lfloor size\ /\ ncp \rfloor$. The last PE takes the rest of the nodes.

Compared with Root Initialization, this method has the same advantage as the preceding one: only little communication. Additionally, the PEs are doing less redundant work. Only the first part of the path of one PE is identical with some other PE. In every iteration, the PEs of one path are partitioned into the branches of the path. For this reason, the overall efficiency of this initialization is greater than the previous methods.

On the other hand, the selection of nodes is not as good as with the previous methods. In contrast to the serial B&B, this initialization generates the nodes more or less independently of each other. The lower bound of the nodes are not sorted or compared. To summarize, the PEs are not initialized with the *p* best nodes.

In the circular method, nothing is said about the case when *size* equals zero. This happens if no successor of a node can be generated or all successors are pruned by the upper bound. Furthermore, no load balancing is applied in the start-up phase. Thus, the PEs with no successor will become idle until they enter the working phase. Altogether, this method has slightly more idle times as the previous two.

---

[1] The calculation of *delta* is a corrected version of the formula in [2]

3.4. *Direct Initialization*

The last initialization method described here is the Direct Initialization. It is used for the Vertex-cover problem in ref. [17], and a serial version without exploiting the parallelism is mentioned in ref. [1].

The main strategy is not to built up the search tree explicitly in the start-up phase. Instead, each PE directly computes its local (root) node for initialization similarly to the circular method of Selective Initialization. Because every PE should be served with a distinct node, all the nodes for initialization are taken from a certain depth of the search tree. This depth is determined such , that the amount of nodes in this depth is greater than the number of PEs. Thus, every PE is supplied with at least one node. If there are more nodes than PEs available, then some PEs take multiple nodes at once to ensure the correctness of the algorithm. After initialization, every PE continues with the standard parallel B&B algorithm using its node as a (local) root. In summary, the parallel B&B algorithm starts its usual search in a certain depth of the search tree instead from the root node.

This strategy is only applicable if the structure of the search tree is known in advance. Then, a function for node generation can be set up. It computes the i-th node in a certain depth d_0 with $d_0 = \lceil \log_b p \rceil$.

This approach for initialization includes most of the advantages of the previous two methods, i.e., little communication and no idle times. Additionally, the PEs are not doing any redundant work and, therefore, the efficiency is maximal. On the other hand, possible bad branches of the search tree are not pruned by the upper bound, because no lower bound of the nodes is computed during initialization. The pruning is possible at the end of the start-up phase and will lead to PEs without work. The second disadvantage is the same as in Selective Initialization: the selected nodes are not the p best ones.

3.5. *Comparison*

In this section, the different initialization methods are compared with reference to several criteria (see Table 1). The criteria are selected according to their influence on the quality of the single method. A short description of each criterion follows:

- The first criterion indicates the amount of necessary *communication* between the PEs during the start-up phase.
- Only if the PEs are working on distinct nodes is the potential parallelism exploited. Therefore, the *redundancy* of work is an important criteria. It is measured in the number of multiple processed nodes.
- The *idle times* of the PEs act as an indicator for decreasing efficiency. In our model, they occur if a PE is waiting for work, i.e., receiving a node from another PE.

- An initialization method is called *general* if it is applicable to all discrete optimization problems, which could be solved by the B&B algorithm. As the table shows, not all initialization methods have this property.
- The *variable branching* criterion indicates whether the method is able to handle search trees with non-constant branching factors or whether some branches may be pruned by their poor cost estimation.
- The last criterion specifies whether the nodes have been selected in a *best-first* fashion at the end of the start-up phase.

In Table 1, three marks (+, +/-, -) state how good (good, fair, poor) a criterion fits to one initialization method relative to other methods. To enable a homogeneous evaluation, the first three criteria have to be used in their reverse forms. For example, comparing the redundancy, the Root and Direct Init. evaluate as good, the Selective Init. as fair, and the Enumerative Init. as poor concerning the avoidance of redundancy.

	Root Init.	Enumer-ative Init.	Selective Init.	Direct Init.
no comm.	-	+	+	+
no redundancy	+	-	-/+	+
no idle-times	-	+	-/+	+
generality	+	+	+	-
var. branching	+	+	+	-
best-first	+	+	-	-
references	others	[16, 10]	[2, 7]	[17, 1]

Table 1: Comparison of the four B&B initialization methods

4. Analysis

When investigating the different initialization methods, the resulting efficiency of the overall algorithm is of high interest. Most of the criteria of Section 3.5 more or less influence the efficiency. In this section, we analyse a criterion which is more basic than the others. When regarding the idealised parallel computational model (PRAM), only the idle times of the PEs are relevant. The other criteria vanish because of the special PRAM structure (see below). Regarding real parallel machines, all the problems of the PRAM are included, too, plus several additional ones. Thus, the idle times form a machine-independent and method-inherent feature of the B&B initializations.

For the analysis, we assume a Parallel Random Access Machine with concurrent read and write operations (CRCW-PRAM) as a model. Each PE has access to the global memory in constant time. Thus, there is no need to regard the communication because it can be done through the global memory. In addition, no redundancy of best-first problems have to be considered because the PEs always receive the p best unique nodes from the memory.

In the PRAM, OPEN is kept in the global shared memory. Therefore, the first three initialization methods reduce to one method. It is the development of the search tree beginning from the root. The root is inserted in the global OPEN set as initialization. Then the nodes in OPEN are successively expanded and the search tree is built up. This procedure is the same for the first three initialization methods. On the other hand, the fourth initialization method directly generates the appropriate nodes in a certain depth of the tree. Thus the start-up phase is shortened.

4.1. Idle Times

With special consideration of the three phases of the B&B, we will take a closer look into the start-up phase in the following paragraphs. In this phase, the first three initialization methods build up the search tree successively. Each PE grabs a node from OPEN and expands it. With a constant branching factor of b, the algorithm can expand at most b^d nodes in depth d. By definition, the start-up phase ends when there is at least one node for each PE. This is the case in the depth d_0, with

$$d_0 = \lceil \log_b p \rceil \tag{5}$$

From the very beginning of the algorithm, all of the p PEs are available and in every iteration all nodes in OPEN can be expanded. Thus, the start-up phase takes $d_0 + 1$ iterations, beginning the first iteration in depth zero.

In every single iteration, p nodes could potentially be expanded. So, the amount of expanded nodes cab be as high as $p(d_0 + 1)$. But not all PEs can do a reasonable amount of work during the start-up phase. Some of the PEs are working and the rest are idle waiting to receive work. Let a_i and b_i denote the number of idle and working PEs in iteration i, respectively. Additionally, we define the sums over all iterations (depth zero up to d_0) of the start-up phase by

$$A = \sum_{d=0}^{d_0} a_d \quad \text{and} \quad B = \sum_{d=0}^{d_0} b_d \tag{6}$$

Clearly, adding these sums of working and idle PEs over the total start-up phase will give the total amount of potential expandable nodes in this phase:

$$A + B = p(d_0 + 1) \tag{7}$$

The number of working PEs is determined by the number of expandable nodes per each iteration. Assuming that the iteration corresponds with depth d+1 in the search tree, which holds in the start-up phase, then b_i can be calculated by min(p, b^{i+1}). Thus, the sum of working times B is

$$B = \sum_{i=1}^{d_0+1} \min(p, b^{i-1}). \tag{8}$$

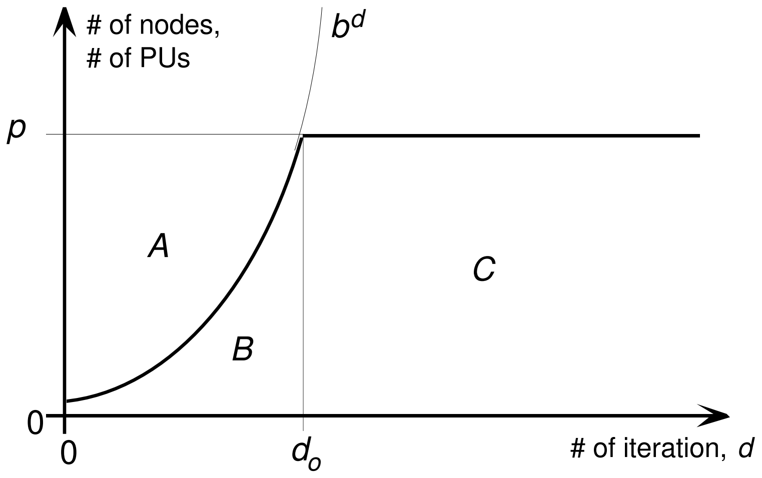

Fig. 2: Different quantities of PE times in parallel B&B algorithm are indicated by following areas: A*: idle times,* B: *working times in start-up phase,* C*: working times in working phase.*

Accordingly, the sum A can be calculated by

$$A = p(d_0 + 1) - \sum_{i=0}^{d_0} \min(p, b^i) \tag{9}$$

By choosing the number of PEs p with $p = b^k$, for some $k > 0$, this equation can be simplified. The minimum operator is superfluous and a closed form of the second term is given through the geometric sum. The branching factor b is greater than 1. Otherwise there will be only a pathological search tree. Additionally, we view A as a function depending only on the number of PEs p, and the branching factor b. With this, the sum of idle PEs over all start-up iterations is

$$A(p,b) = p(1 + \log_b p) - \frac{pb - 1}{b - 1} \tag{10}$$

In Fig. 2, all the above-mentioned quantities are shown. The maximum number of expandable nodes of the search tree and the number of working PEs are plotted against the iteration. The total amount of idle and working PE times are indicated by A and B, respectively. The area C denotes the ideal working phase of the B&B, where about p nodes are expanded in each iteration.

4.2. Efficiency

With the quantification of idle times caused by the initialization, it is easy to estimate the efficiency of the overall algorithm. Therefore, it is assumed that the complete (balanced) search tree has to be explored for solving the problem, which is certainly unfavourable. With d_{max} indicating the tree depth, the total number of nodes to be investigated is calculated by $b^{d_{max}}$.

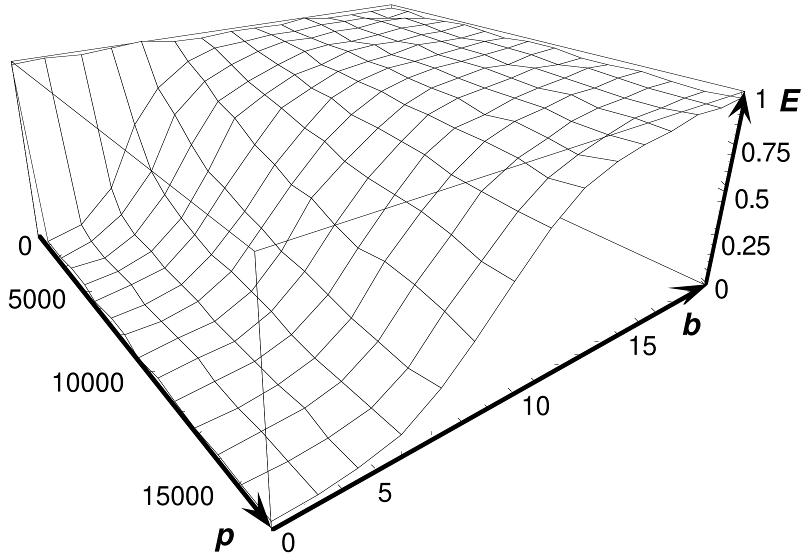

Fig. 3: Efficiency of the overall B&B algorithm when one of the first three initialization methods is used.

This corresponds to the iteration number of the serial algorithm. With the parallel B&B, the amount of idle times has to be added. Thus, the number of iterations I is

$$I(p) = \left\lceil \frac{b^{d_{max}} + A(p,b)}{p} \right\rceil \tag{11}$$

Thereby, the term A is chosen according to the last section. The speed-up S is calculated by $S(p) = I(1) / I(p)$ and the efficiency by $E(p) = S(p) / p$. Thus, the overall efficiency E of the algorithm is

$$E(p) = \frac{b^{d_{max}}}{p} \left\lceil \frac{p}{A(p,b) + b^{d_{max}}} \right\rceil \tag{12}$$

Notice that the length of the start-up phase depends on the branching factor b and on the number of processors p. In Fig. 3, the efficiency is plotted against the number of processors p and the branching factor b. Thereby, the depth d_{max} of the search tree is 20. The efficiency increases when the branching factor becomes greater than a certain threshold depending on p.

In the previous analyses, the occurring idle times and the efficiency of the B&B start-up phase are formulated. Thereby, two facts can be observed. First, the idle times increase with the number of PEs, assuming that the problem instance chosen is complex enough. Second, by increasing the branching factor of the search tree, this idle times can be reduced. Therefore, if there is a choice between different enumeration schemes, the schema generating the broader tree is preferable. This holds only if one of the first three initialization methods is used.

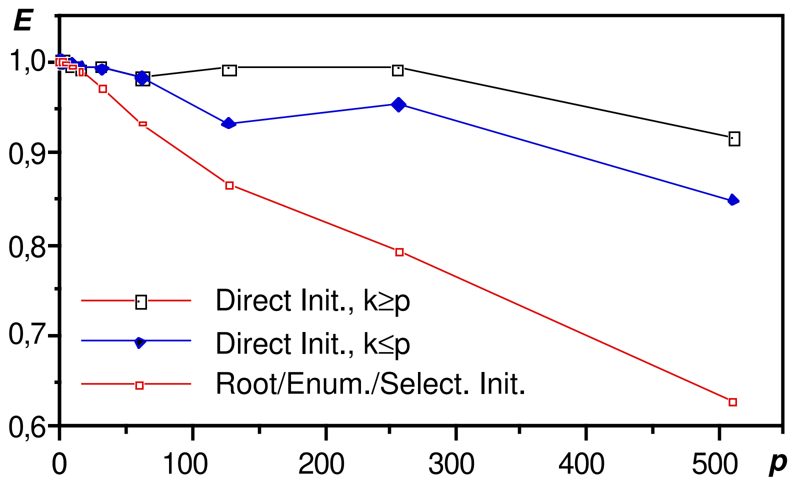

Fig. 4: Comparison of different initialization methods in a PRAM simulation. E*: efficiency,* p*: number of processors.*

Considering the Direct Initialization, the behaviour of the start-up and working phase cannot be analysed in that way. With this method, there is no start-up of this kind. Thus, there are no idle times in the start-up. But, because of the missing start-up phase, the following working phase is influenced. Thus, the overall behaviour of the different initialization methods will be compared by experiments outlined in the next section.

5. Experimental Results

We solve a typical scheduling problem, as an application domain of the parallel B&B. A fixed number n of elementary jobs with different processing times operate on m identical machines. Thereby, the single job is not preemptable and there are no precedence relations between the jobs. The optimization problem is to find the sequence of jobs on each machine which minimizes the total processing time (makespan) of the system.

The B&B algorithm uses partial (incomplete) schedules as intermediate nodes of the search tree. The branching factor b and the depth d_{max} of this tree are determined by the number n of machines and the number m of jobs, respectively. The search tree is built up by successively placing single jobs on all machines. Thereby, the next job is selected according the maximum processing time. The lower bound of each partial schedule is computed by its makespan plus the average processing time still needed. Very long jobs can lengthen the second term. The B&B selects the nodes from OPEN in a best-first fashion. Conflicts are solved by depth-first prioritization in the search tree: Deeper nodes and more left nodes are preferred.

The experiments were performed by a PRAM simulation, cf. Section 4. The problem solved consists of three machines ($b = m = 3$) and twelve jobs

($d_{max} = n = 12$) with different processing times (4, 4, 4, 5, 5, 5, 5, 6, 6, 6, 6, 6). Thus, the search tree has a total size of about $7.9*10^5$ nodes. The implemented B&B expands in average by about 6.000 nodes. In Fig. 4, the two types of the presented initialization methods are compared by solving this problem. The efficiency of the Root Initialization and the Direct Initialization are plotted against increasing number of processors. The latter method is split up generating more ($k \geq p$) or less ($k \leq p$) initial nodes than PEs available.

The comparison of the three initialization methods supports the results of the analysis, given in Section 4. The values of the Root Initialization correspond to one section of the efficiency in the plot of Fig. 3. The efficiency of the Direct Initialization is better than the previous one throughout, though it decreases with increasing p, too. Comparing the different numbers k of expanded nodes in the first iteration, the method with more nodes than PEs ($k > p$) is preferable.

6. Conclusion

The results indicate that the differences of the single initialization methods are significant. The performance of the overall algorithm is influenced by the method chosen for the start-up phase. The efficiency of the first three methods (Root, Enumerative, and Selective Init.) depends on the branching factor of the search tree. An enumeration scheme generating a broad search tree is preferable.

For the Direct Initialization, the PRAM analysis and simulation show that it has the best efficiency, assuming it is applicable. This holds, especially if more nodes than processors are generated in the first iteration. One drawback of the Direct Initialization is the lack of heuristic pruning. After initialization of a distributed OPEN set, due to pruning some PEs will become idle until the dynamic load balancing will provide them with nodes again. The fraction of pruned nodes depends on the application domain, thus, the final decision pro or contra Direct Initialization cannot be done in general.

7. Acknowledgements

This research work was founded by the Deutsche Forschungsgemeinschaft (DFG) with a stipend in the frame of the "Graduiertenkolleg". The work was performed at the Institute for Real-Time Computer Systems and Robotics, Prof. Dr.-Ing. U. Rembold and Prof. Dr.-Ing. R. Dillmann, University of Karlsruhe, D-76128 Karlsruhe, Germany.

8. References

[1] Abdelrahman T. S. and T. N. Mudge, 1988, "Parallel branch and bound algorithms on hypercube multiprocessors", Proceedings of the 1988 ACM Conference on Lisp and Functional Programming, pp. 1492-1499.

[2] El-Dessouki O., Huen W. H., 1980, "Distributed enumeration on network computers", IEEE Trans. on Computers, vol. 29, pp. 818-825.

[3] Huang S.-R and Larry S. Davis, "Parallel Iterative A* Search: An Admissible Distributed Heuristic Search Algorithm", Proceedings of the eleventh International Joint Conference on Artificial Intelligence, 1989, pp. 23-29.

[4] Imai M., Fukumara T., Yoshida Y., 1979, "A parallelized branch-and-bound algorithm: Implementation and efficiency", Systems-Computers-Control, vol. 10, no. 3, pp. 62-70.

[5] Janakiram V. K., et al., 1988, "A randomized parallel branch-and-bound algorithm", Int. Jour. of Parallel Programming, vol. 17, no. 3, pp. 277-301.

[6] Lai T-H., Sprague A., 1985, "Performance of Parallel Branch-and-Bound Algorithms", IEEE Transactions on Computers, vol. C-34, no. 10, pp. 962-964, MAG Lab papers, no. 33.

[7] Ma R. P., Tsung F. S., Ma M. H., 1988, "A dynamic load balancer for a parallel branch-and-bound algorithm", Proc. of the 3rd Conf. on Hypercubes Concurrent, Computers, and Applications, Pasadena, CA, pp. 1505-1513.

[8] Miller D. L., Pekney J. F., 1989, "Results form al parallel branch-and-bound algorithm for solving large symmetric traveling salesman problems", Operations Research Letters, vol. 8, pp. 129-135.

[9] Mohan J., 1983, "Experience with two parallel programs solving the traveling salesman problem", Proc. of the Int. Conf. on Parallel Processing, Bellaire, Michigan, Aug. 1983, pp. 191-193, IEEE Comp. Soc., Washington, D. C.

[10] Pargas R. P., Wooster E. D., 1988, "Branch-and-bound algorithms on n hypercube", Proc. of the 3rd Conf. on Hypercube, Concurrent Computers, and Applications, Pasadena.

[11] Quinn M. J., Deo N., 1986, "An upper bound for the speed-up of parallel best-bound branch-and-bound algorithms", BIT, vol. 26, no. 1, pp. 35-43.

[12] Roucairol C., 1988, "Parallel branch and bound algorithms: An Overwiew", Proc. of the Int. Workshop on Parallel and Distributed Algorithms, Gers, France, pp. 153-163.

[13] Schwan K. and B. Blake and W. Bo and J. Gawkowski, 1989, "Global Data and Control in Multicomputers: Operating Systems Primitives and Experimentation with a Parallel Branch-and-Bound Algorithm", Concurrency: Practice and Experience, vol. 2, pp. 191-218, vol. 1

[14] Sprague A. D., 1991, "Wild anomalies in parallel branch-and-bound", Tech. Rep.91-04, CIS, UAB, Birmingham.

[15] Taudes A., Netousek T., 1991, "Implementing branch-and-bound algorithsm on a cluster of workstations", eds: Grauer M., Pressmar D. B., Parallel cpmputing and mathematical optimization, Proceedings, Springer.

[16] Vornberger O., 1986, "Implementing Brach-and-bound in a ring of processors", Proc. of CONPAR 86, Lecture Notes on Computer Sci. 237, Springer.

[17] Vornberger O., 1987, "Load balancing in a network of transputers", Second Int. Workshop on Distributed Algorithms, Amsterdam, pp. 116-126.