

Workout

Combinatorial search

within the seminar Parallel Programming and Parallel Algorithms

Felix Potthoff

Professor: Prof. Dr. Herbert Kuchen

Assistant: Christian Hermanns

Institut für Wirtschaftsinformatik

Inhaltsverzeichnis

1	Introduction	1
2	Basics	2
3	Combinatorial Search	3
3.1	Combinatorial Problems	3
3.2	Sequential Algorithms	3
3.2.1	Divide & Conquer	3
3.2.2	Backtrack	5
3.2.3	α - β -Pruning	7
3.3	Parallel Algorithms	12
3.3.1	Parallel Divide and Conquer	12
3.3.2	Parallel Backtrack	13
3.3.3	Parallel α - β -Search	17
3.4	Speedup anomalies	19
4	Summary	21
A	Functions	22
	Literaturverzeichnis	23

1 Introduction

Search algorithms are fundamental problem solving methods in fields like computer science and operations research.[PP95] They could be used in many practical applications. In computer science for instance, applications such as database, expert or robot control systems. Even for example in AI learning, planning or natural language understanding are suitable fields.[PM07][p. 1][NK87][p.1]

A variety of different search algorithms exists, because most of them are specialised for a certain kind of problem. Therefore, chapter 3 introduces different sequential search algorithms.

Different classes of search problems could be defined, such as path-finding problems, two-player games and constraint satisfaction problems. For instance classical examples of path-finding problems are combinatorial optimization problems like Rubik's cube or Eight Puzzle. Chess, Backgammon, and the "Game of Othello" belong to the class of two player games, while a classic example of a constraint satisfaction problem is the eight-queens problem.[PP95]

Clusters of the shared-memory architectural style have become popular nowadays as well as hyper threading and multicore processors. Hardware is getting cheaper and consequently, shared memory and parallel programming models could be used cost-effectively to speed up searches as a serious competitive environment to message passing.[PM07][p.1][NK87][p.1]

To speed up search the sequential approaches are adopted and used to develop parallel variants in chapter 3.3 which are based on the work of [Qu04].

In this chapter the Divide & Conquer algorithm serves as a simple example to understand the general issues of parallel processing. Furthermore, in the following section, two parallel variants of Backtrack are introduced. At the end of chapter three different approaches for parallel α - β algorithm are presented.

Roughly considered, the impression could arise, that m parallel processors working for the same search can speed it up m times. But chapter 3.3 will show that several processors have to deal with more overhead than one single processor, which reduces the speedup. Section 3.4 will give a short introduction of speedup anomalies in parallel formulations.

Chapter 4 summarises the algorithm and their advantages and disadvantages.

2 Basics

Basically a graph consists of a set of vertices and a set of edges which connects the vertices. A path is a collection of edges, which connects two vertices. Each edge may appear only once in a path.[Di03][p. 7] A cycle persists, when a path from an arbitrary node to itself exists. A tree is a special graph, which has no loops and only at most a single edge between each pair of vertices.[Ko09][p. 16-18] Each tree has a certain branching factor. It denotes the number of child nodes assigned to each parent node. In balanced trees this branching factor is the same for each node. That means each parent node has the same number of children.

A vertex preceding another vertex on a path from the root is called a ancestor. The immediate ancestor is called child of the node. In contrast to an ancestor a vertex which follows another is called a descendent. Particularly an immediate ancestor is called a parent. A leaf is a vertex without any child.[Ko09][p.19]

The number of vertices is called the order and the number of edges the size of a graph.[Ko09][p. 16]

A common technique to represent decision and optimisation problems is the use of trees. There are several kinds of search trees which are more or less suited for a certain problem.

A game tree consists of vertices denoting different game layouts and edges being the possible moves. Game trees are suitable to represent two player games like chess. For example in a chess game, edges to children represent the possible moves of chess pieces and the children are the resulting board positions.[PM07][p.1] The root would take the initial board position where each figure stands within the two lines on the player's board side.[Qu04][p.388]

In an AND-Tree the child nodes are related to sub problems which have to be solved in order to solve the initial problem. This implicit that each leaf node has to be explored to find the solution to the initial problem.

Also in OR-trees the vertices represent sub problems. But to solve such an or-node, only one children has to deliver a solution to its sub problem.

A AND/OR-Tree includes or-nodes and even and-nodes. The behaviour is straight forward. At an or-node one solution of any child is enough to solve the nodes sub problem. At an and-node every children has to deliver the solution to its sub problem, before the and-nodes problem is solved.

A graph with a designated initial node and several possible goals as leaf-nodes is called a state space tree. [GV99][S.1]

3 Combinatorial Search

3.1 Combinatorial Problems

Combinatorial search problems could be roughly categorised:

- *Decision problems* are intended to find one possible solution for a problem.
- *Optimisation problems* find the best solution for a given problem.

Combinatorial search finds solutions for such diverse problems as assigning crews to airline flights, human voice understanding, or playing games such as chess-games. For a delivery service, for instance, finding the best route is an essential problem, which can be solved by using combinatorial search. (Well known as “Travelling Salesman Problem“)

As mentioned above a suitable representation for such problems are those trees presented in the last chapter. The solution is finding one or more paths through the tree.[Qu04]

A common technique to solve a problem, particularly an optimisation problem is finding a path through an or-tree.[GV99][S.1]

In the next section, sequential algorithms “Divide & Conquer“, “Backtrack“, and “ α - β -Pruning“ are described. In chapter 3.3, “Divide & Conquer“, “Backtrack“ and “ α - β -Search“ are extended to be executable in parallel by several processors, or several distributed computers. Note that not all algorithms are independent from each other. For instance α - β -Pruning is a variant of “Backtrack Search“.

3.2 Sequential Algorithms

3.2.1 Divide & Conquer

The idea of “Divide & Conquer“ is quite simple. The main problem is broken up into sub problems. Each of these sub problems is solved by “Divide & Conquer“ relatively independent. The last step combines all sub problems iteratively to solve the initial problem.

The algorithm can be represented by an AND-Tree. That means, every sub problem which is represented by a node has to be explored to solve the initial problem. “Divide & Conquer“ algorithms are designed to find one possible solution.

A prominent example is the “Quicksort“ algorithm, which is briefly introduced in the following section to demonstrate the “Divide & Conquer“ idea. On the one hand

“Quicksort“ is space efficient and needs only $n \log(n)$ operations on the average. On the other hand the algorithm needs $O(N^2)$ operations in the worst case.[Se88][p.103] The “Quicksort“ algorithm is designed to sort a set of numbers. A possible imple-

Parameters:

A - An array to sort
start - Startposition
stop - Endposition

```

1: Quicksort(A, left, right)
2:   if left < right then
3:     seperator ← Partition(A, left, right)
4:     Quicksort(A, left, seperator - 1)
5:     Quicksort(A, seperator, right)
6:   endif
    
```

Algorithm 1: Quicksort-Algorithm

mentation modelled after [Se78][p. 848] is shown below in Example 1. The algorithm is started with the initial problem say an unsorted array $A = [4, 3, 1, 2, 8, 6, 5, 9, 7]$. This initial step is represented in a tree by the root. Figure 1 shows the whole tree for the sorting.

In the second step, see line 3 the algorithm calls the “Partition“ function. Pseudo-

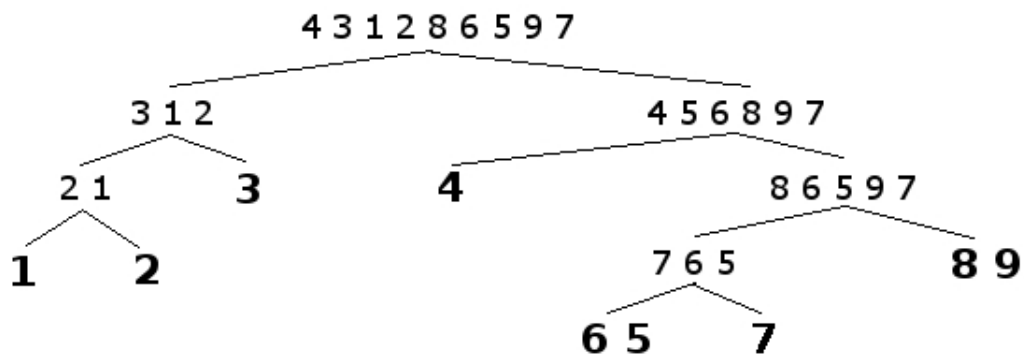


Figure 1: Quicksort

code for the “Partition“ function is separated out to the appendix. It ensures the sorting of the partitions. That means the function returns a list-element sometimes

called pivot. The pivot element divides the array in to partitions. Each element on the right side of the pivot element is greater and each element on the left is smaller. The index of the pivot element is stored to **separator** .

With this **separator** the Quicksort algorithm starts two recursions. One with the left and one with the right partition respectively. Each recursion step corresponds to a child node or subtree respectively. For instance, the first calls of the two recursions produce the two branches rooted by the the left child [3, 1, 2] and the right [4, 5, 6, 8, 9, 7] in Figure 1. Each partition can be processed independent. That is the first hint that such an algorithm is suited for parallel processing, because the sub problems can be processed relatively independent. The algorithm breaks up the whole array in the same manner until there is only one “number“ left or a sorted partition. In the search tree the leaf nodes represent these endpoints. In Figure 1 the leafs are printed bold. If the algorithm reaches such a leaf node, the recursion stops and the found sub result is given back up the tree, to be combined to the final solution. Final solution means the ordered set of numbers.

3.2.2 Backtrack

Consider the problem of finding a single solution in a state-space tree containing one or more solutions. “Backtrack“, also called Depth-First-Search, is a widely used technique for solving such problems, which is storage efficient.[NK92][p. 2]

It uses linear amount of memory with respect to the depth of a tree. “Backtrack“ is designed to find any solution.[GV99][S.2]

The fundamental concept of backtracking is to go deep in the tree until a solution is found or no exploration is possible. A “Backtrack“ algorithm starts at the root, generates all children and continues the search at an arbitrary one. If it is impossible to move further down, the algorithm backtracks to the nearest previous node, where a child node is not explored.

A well known problem which is suited to be solved by the “Backtrack“ algorithm is the n-queens problem. The task is to place n queens on a chess board with nxn fields, so that they could not attack each other. Two queens could attack each other when they are in the same line, column or diagonal to each other.

In the following the 4-queens problem is used to demonstrate the working of the “Backtrack“ algorithm. A possible solution of this problem is shown in Figure 2.

Each board position can be represented by an (x,y)-position for each queen. Consequently representation for the solution in Figure 2 is [(1,2)(2,4)(3,1)(4,3)]. Consi-

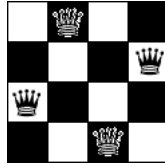


Figure 2: Statespace tree four-queens[PW04]

dering the fact that for a valid solution each queen must be in its own row there is a more efficient way to present a board position. Under assumption that the first queen is always placed in the first column, second always in the second column etc., it is enough to represent the line in which the queen is placed. This results in the representation $[2,4,1,3]$ for the solution of Figure 2 above.

For simplicity the sub trees rooted by node $[3]$ and $[4]$ are omitted, because they

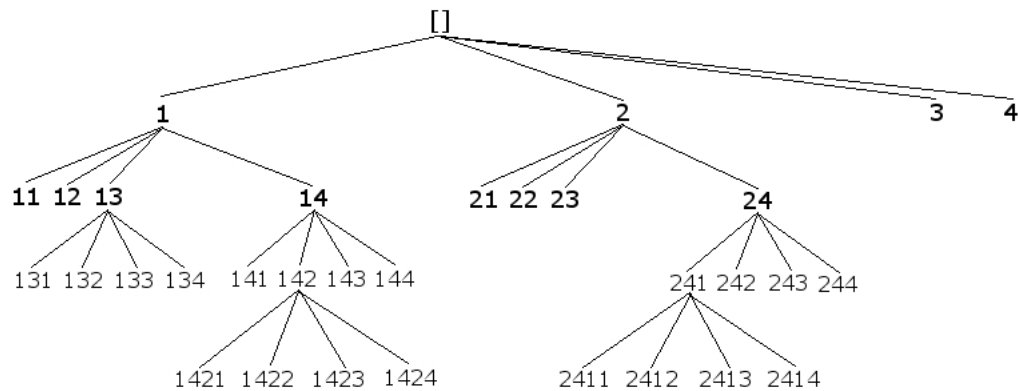


Figure 3: Statespace tree four-queens[PW04]

are only mirror images of the sub trees rooted by $[1]$ and $[2]$.

The algorithm starts with a blank board. Then it explores all possibilities how to place the first queen in one of the four lines. Obviously these are only four. Accordingly the root has four children. After that one of the resulting board positions is chosen for further processing. For example the processing could be continued at node $[1]$ and explore its child nodes $[11][12][13][14]$. Again one arbitrary child node is chosen, for instance the node $[11]$. But this is an invalid board position, because two

queens are not allowed to be in the same line. Therefore the algorithm backtracks to the previous node and choose another child node to continue the search. For instance, the next child node from the left, [12]. The algorithm follows this strategy until it considered all necessary placements.

The “Backtrack“ algorithm cuts off large parts of the tree, compared to a brute force approach which tests each possible board position($16 \times 15 \times 14 \times 13 = 43680$). The presented approach considers significant fewer situations.[PW04][p.1-4]

The algorithm can be implemented with linear use of memory, because only the current alternatives at each level have to be maintained in the memory. A disadvantage might be that the algorithm needs exponential time in the worst case.[Qu04][p. 374]

3.2.3 α - β -Pruning

Game trees sometimes are far too complicated to be evaluated completely. Consider a game tree for chess, which represents all possible moves. The tree would have 38^{84} possible board-positions, i.e. nodes.[Qu04] For this reason an algorithm can stop at a specific level or after a certain amount of time and then estimate the global value. As a common rule, the deeper the search, the better its quality, because the value could likely be estimated more precisely, the deeper the search goes into the tree. If an algorithm has to examine fewer nodes at each level, it obviously can go deeper into tree in a fix amount of time. To guarantee a good search depth, even in more complex trees, α - β -Pruning avoids evaluating nodes which cannot influence the outcome of the search tree. Omitting such path or branches is called pruning. α - β -Pruning can be seen as a depth-first “Branch & Bound“.[GV99][p .5]

The algorithm is based on the assumption that the first player always tries to maximise his outcome. In reaction to this strategy the second player protects himself by minimising the first player’s outcome. The resulting value if both player cleave on this strategy is called mini-max value. Therefore the α - β algorithm uses a lower (α) and an upper (β) bound on the expected value of the tree. The range spanned by these bounds is called search window and used to possibly prune parts of the tree.[Ma86][p.3] Algorithm 2 presents pseudo-code for an α - β -algorithm.[Qu04][p. 393][GV99][p. 5] It follows some simple rules. These rules could be named as the “Mini-Max“-Strategy.

- Set α to a Maxnode’s value if it is greater than the current α . Because the just explored path is better for Player One than the one found so far.
- Set β to the value of a Minnode, if the Minnode’s value is smaller than the current β . Because the just explored path is better for Player Two than the

one found so far.

- If α reaches a higher value as β , the current node represent a situation that is inferior for one player. That means he would never allow the other to reach this state. Because of that no further children of the current node have to be examined.

[KM75] have introduced four node types for α - β -search trees. Type-1-Nodes are initial nodes which are searched with an open search window. This means the upper and lower bound are equal to $+\infty$ and $-\infty$ respectively. Type-2-Nodes have no upper bound and Type-3-Nodes have no lower bound respectively. Type-4-Nodes are undefined nodes, which are reached by a search instance with both, an upper and a lower bound. Because a chess game-tree is too large to convey the workings of α - β -algorithm an easier hypothetic game is introduced. But note the game has the same properties as chess. It only has fewer alternative situations to consider. Each player has two turns and their turns are iterated. A game-tree for this two player, zero sum game described above is shown in Figure 4. Arbitrary outcomes are chosen for the end situations of the game represented by leaf nodes. Positive numbers indicate units of money won by Player One and negative numbers, units of money won by Player Two. For instance, node “B“ describes that Player Two wins 5 units of money.[Qu04][p.388] Initially the algorithm starts with $\alpha = -\infty$ and

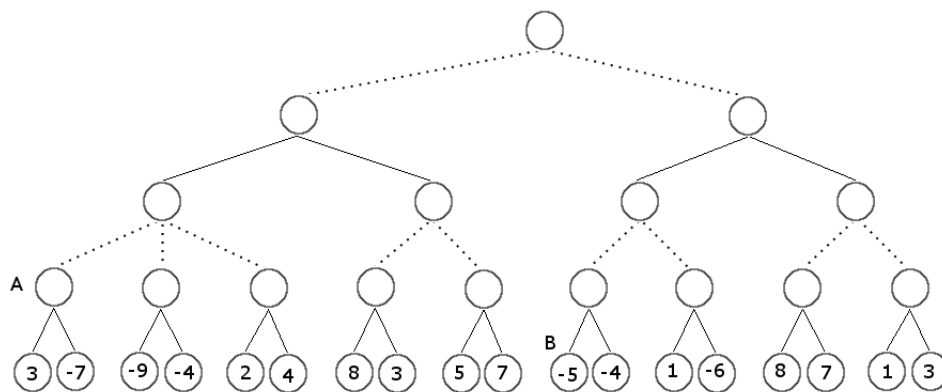


Figure 4: Game tree for hypothetical two player game

$\beta = +\infty$. [GV99][S.5] For each generated node the algorithm is called recursively with the window which is valid for the node, i.e. the sub tree rooted by the node. As shown in line 12 of Algorithm 2 the α - β -Search is called with the current α , β - values and the found subtree. The algorithm has to explore only nodes within the range

of the values. α - β -Search traverses over all nodes of a tree, if no pruning appears. Note, the search window for the first child is always the same as the parent's search window. That means that in every situation the left most subtree of size two has to be evaluated completely. For instance see the node "A" in Figure 4, all its children have to be explored. No pruning can appear at this state of the processing.[Qu04] If the algorithm has explored all necessary children of a Max-node, it returns its current α -value. In case of a Min-Node the β -value is returned. If during the processing the value of α is greater than or equal to β , then the cut-off is set to true. Accordingly the "while-loop" between Line 10 and 21 is skipped. That means no further nodes of the current subtree are evaluated. In first part of Algorithm 2 necessary variables are defined. The shown implementation of α - β algorithm always stops at a specific search depth. To see how the

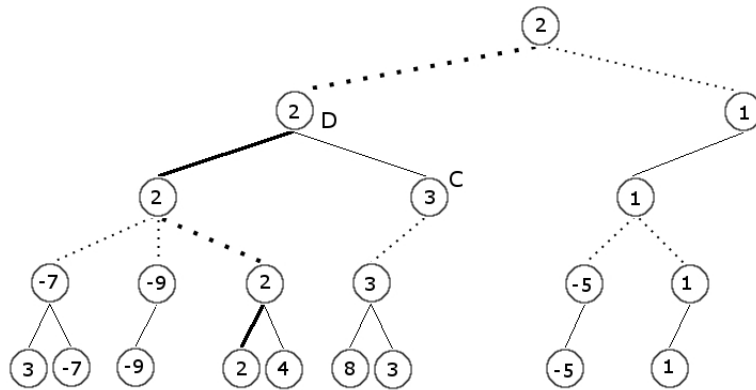


Figure 5: Game tree for hypothetical two player game with pruning[Qu04]

pruning works for a specific example see Figure 5. The game tree shows only the nodes of the tree in Figure 4 which would be visited by the α - β -algorithm. The path which corresponds to the "Mini-Max"-Strategy is printed with bold edges. Pruning appears for example at Node "C" the α -value is greater than β . At this point it is not necessary to explore any further child node, because Player Two would never allow Player One to reach the path, corresponding to the second child of node "D". He would choose the bolded path at node "D" to minimise Player One's outcome. The most pruning appears in a perfect ordered tree. Because the best solution is always examined first. In such a case the α - β -algorithm has to examine only approximately twice of the square root of all nodes, which would be explored without pruning. Obviously this is the best case and in this particular situation the effective branching factor is reduced from b to $\sqrt[2]{b}$. Correspondingly, the algorithm can go twice as deep in the game tree as the minimax algorithm.

Parameter:

`pos` - Position

`β` - Upper Bound `α` - Lower Bound

`depth` - Search depth `maxc` - Maximum possible moves

Variables:

`max` - Children of `pos` in gametree

`cutoff` - Set to `TRUE` when ok to prune

`i` - for iteration `width` - number of legal moves

Alpha_Beta(`pos`, `α` , `β` , `depth`)

```

1:   if depth ≤ 0 then
2:     return (Evaluate( pos )) - Evaluate terminal node
3:   endif
4:   width Generate_Moves(pos )
5:   if width = 0 then
6:     return (Evaluate(pos )) - No legal move
7:   endif
8:   cutoff ← TRUE and 9:   i ← 1
10:  while (i ≤ width) and (cutoff = TRUE) do
11:    val ← Alpha_Beta(c[i],  $\alpha$ ,  $\beta$ , depth-1)
12:    if MAX_Node(pos) and val >  $\alpha$  then
13:       $\alpha$  ← val
14:    elseif MIN_Node(pos) and val <  $\beta$  then
15:       $\beta$  ← val
16:    endif
17:    if  $\alpha$  >  $\beta$  then
18:      cutoff ← TRUE
19:    endif
20:    i ← i + 1
21:  endwhile
22:  if MAX_Node(pos) then
23: return  $\alpha$ 
24:   else return  $\beta$ 
25:  endif
26: end

```

Algorithm 2: Sequential α - β -algorithm

Experimental evidence could be shown that the α - β algorithm often searches no more than fifty percent of the nodes which would be explored in an perfect ordered tree. In other words normally α - β algorithm leads to much higher performance than a Mini-Max-Algorithm.

Several enhancements are available to speedup α - β search. In the following, two approaches will be presented. The “Aspiration Search“ tries to reduce the amount of expanded notes by taking narrower search windows. The bounds for the windows are calculated by the forecast values for the mini-max-value. If the actual mini-max-value falls into the window, no further work is necessary and the speedup can be high. A disadvantage of this approach is that the initial bounds sometimes do not embed the actual mini-max-value. In this case the search has to be repeated with corrected bounds.[Ma86]

As the name “Iterative Deepening“ already says explores this enhancement of the game tree only until a certain depth. By using this strategy the time which is needed for a search can be controlled. The depth can be raised iteratively. Such a search is called a (x)ply, where x is the number smaller than the depth of the tree. A (x)ply search explores the tree till level x+1.

As a further enhancement the value which is returned by such a search can be used as the centre of the window for a “D-ply Aspiration Search“.

3.3 Parallel Algorithms

3.3.1 Parallel Divide and Conquer

As pointed out in chapter 3.2.1 “Divide & Conquer“ algorithm seems to be suited for parallel implementation. In the following two approaches for processing on multi computers and on multi processors is presented.

In case of an implementation for multi processors a central stack can be used to store all the unsolved problems. All processor work on this single stack. If one processor has divided a problem, it can put the resulting sub problems on the stack. Each “free“ processor can take a problem from the top of the stack, in order to solve it. Because of this constellation the stack operates as an effective workload balancing, but has the disadvantage that the stack might get a bottleneck when the number of processes increase.

As mentioned above, the “Divide & Conquer“ algorithm can also be executed by using different computers. For this task two approaches are available. The easiest way is to store the original problem and the final solution in the memory of a single process analogous to the approach for multi processors. A search on such a parallel architecture can be divided into three phases:

- phase 1: The original problem and the arising sub problems are divided and propagated to the parallel computers.
- phase 2: All parallel computers have a task to work on.
- phase 3: Some computers combine results and others are idle.

In the first and third phase fewer tasks than processors are present and only in the second phase it is possible that all processors are busy by computing a task.

Appropriate the speedup of this approach is limited by this overhead for propagation and combination. The size of the problem is also limited by the resources of the central processor.

To avoid these problems the next approach divides the original sub problems and the solutions among the different memories of the parallel processors. This eliminates the overhead for propagation of the first approach. Another advantage is the good scalability of the problem size. It can grow with the number of processors. A disadvantage is the difficulty to balance the workload among the processors, because no implicit mechanism controls the allocation of the processors. The workload has to be handled by communication, the overhead of this approach.

3.3.2 Parallel Backtrack

Because of a high problem complexity it is necessary to speed up the Backtrack algorithm. Therefore, it can be performed in a parallel manner. A straight-forward approach in case of a balanced tree is to split it up into sub trees and divide these sub trees among the processors. That means each processor goes down the tree until a certain level to start its processing. Each process works on one of the sub trees, which are rooted by one a node at this level. The overhead for a processor is to iterate to the “start“ level. If the search depth is smaller than twice of the start depth the overhead is quite small. The overall overhead grows only linear with the number of processors.

To demonstrate how the search can be speeded up by using this approach, a tree with branching factor 3, 10 level to search and 5 available processors is used. If the parallel computing starts at the root, only one node is available. Appropriate only one process could work and the parallel approach is as fast as the sequential one. If the parallel search starts at level 1, three sub trees are rooted by nodes. Each sub tree can be processed by its own processor. The resulting speedup is nearly three. The initial overhead for the processors to go to level 1 reduces the speedup marginal. In case that the parallel processing starts at level 2, [Qu04] reported a speed up of about $9/2 = 4,5$. If there is no level at which the number of processes is equal to the number of nodes like for 5 processors in this example, the sequential “Backtrack“ search is used until a level “m“ where enough sub trees are available to provide one task for each processor.[Qu04][p.375]

This strategy is good for balanced trees, but usually most state space trees are not balanced. In such a case the strategy just mentioned is very poor, because some processes have to proceed larger sub trees than others. That means some processes are still busy while others had a small sub tree to exam and stay idle.

Another approach tries to overcome this problem by using a sequential search in the first step. This search should be as deep enough that each process can examine a large number of sub problems. This approach is based on the assumption that the differences between the processing times are smaller when processes handle a larger part of the tree, rather than a small part. The basic idea is, the more sub trees handled by a process the better is the speedup.

Each process works on its own partition in the same manner. The algorithm for this processing is shown in Algorithm 3. To perform the algorithm, six constants and variables are used, which could be seen at the beginning of Algorithm 3. They define the process environment. For instance the constant `id` is the individual rank

of the process. This `id` is used to identify which nodes should be explored by the process. `Moves` stores the steps the process has made so far. The remaining constants are valid for all processes equally. The `max_depth` is the maximum of steps, which a process is allowed to go deeper in the tree and `p` is the number of all processors. The `cutoff_count` is the level at which the processes start their work.

The algorithm is designed recursively. That means, if the process discovers a new node, the sub tree rooted by this node is processed independently by the “Backtrack“ algorithm. This recursive call is seen in line 16. The function “Unmake_Move“ is the “backtracking“ which was discussed for the sequential algorithm.

Until the process finds a solution, it records the moves it makes and stores them in the variable `moves` .

Each process iterates over the tree nodes to `cutoff_depth` and numbers the nodes found on that level. Based on the number of a node and the process’ id, the sub tree rooted by the node is or is not explored. Considering that at the `cutoff_depth` 8 nodes are found, the `selection_function` is `mod p`, as shown in line 8 of Algorithm 3 and four processors are used. Process 0 would work on the nodes 4 and 8.

This approach can speed up the processing, but also has the disadvantage that it finds every solution. That is because there is no communication between the processes. Each process search its partition of the tree until it found a solution. Sometimes only one possible solution is needed. For such a type of answer the approach is inefficient. Obviously the algorithm should stop as soon as possible if an arbitrary process has found a solution. For this purpose a process can send a message when it found a solution and each process can terminate in case of one of the following events:

- it finds a solution and sends a message
- it receives a message
- it has finished its part of the search space tree.

Normally, if a process tries to exit before another process tries to send a message it leads to a runtime error. To overcome this problem, the process has to ensure that no other messages are on the way.

Dijkstra et. al. have invented an algorithm, called “Distribution termination detection“, to detect if messages are pending a system and if all processes are inactive. If and only if these two conditions are fulfilled, a process is allowed to exit.

Dijkstra et. al.’s algorithm requires that each process has a message-count and a colour. The colour can be either black or white. The processors are also arranged in

Global Constants:

- `cutoff_count` - Count of nodes at cutoff depth
- `cutoff_depth` - Depth at which subtrees are divided among processes
- `max_depth` - Depth to which state space is searched
- `moves` - Records position in search tree (i.e., moves made so far)
- `id` - Process rank
- `p` - Number of processes

```

Parallel_Backtrack(board, level):
1:   if level = max_depth then
2:     if IsSolution(board) then
3:       Print_Solution(moves)
4:     endif
5:   else
6:     if level = cutoff_depth then
7:       cutoff ← cutoff_count + 1
8:       if cutoff_count mod p ≠ id then
9:         return
10:      endif
11:    endif
12:    possible_moves ← Count_Moves(board)
13:    for i ← to possible_moves do
14:      Make_Move(board, i)
15:      moves[level] ← i
16:      Parallel_Backtrack(board, level + 1)
17:      Unmake_Move(board, i)
18:    endfor
19:  endif
20:  return
    
```

Algorithm 3: Parallel Backtrack Algorithm

a so-called successor ring and a token is available which also has a message count and a colour like the processes.

Figure 6 shows the general structure of a succession with 5 processors. The circles represent processes and the numbers inside the circles are their message counts. The square represents the token, also with its message count inside.

A process increments its message count if it receives and decrements its message count when it sends a message. At the beginning all processes and the token have a message count of zero and their colours are white.

To check the system's state a process has to pass the token around this ring.

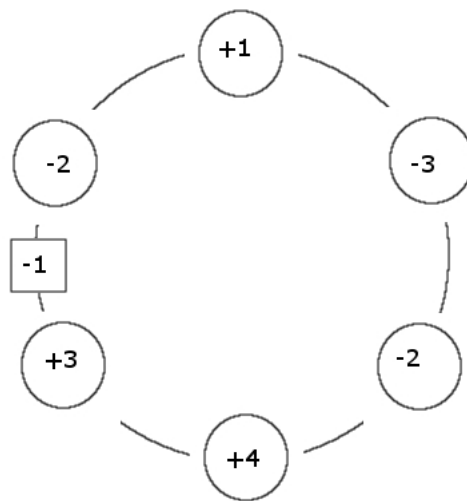


Figure 6: Successorring for five Processors

Generally if a process receives the token, it stops working as soon as possible, because it knows that another process has found a solution. The process holds the token until it has stopped its activity. Meanwhile the process is still active and holds the token, it turns the token's colour to black if its own colour is black. After finishing its activity it turns its own and the tokens colour from back to white. Independent of its initial activity, the process finally adds its message count to the token's message count and sends it to the next process in the successorring. All processes work on and forward the token in the same manner until the token finally returns to the ancestor process. If all following conditions are fulfilled, the algorithm is able to exit:

- the token is back at the ancestor
- the message count of the token is zero
- the colour of the token is white

- the message count of the ancestor process is zero.

Receiving or sending the token do not count as a message, consequently the message-count is not incremented or decremented in case of such an event.

3.3.3 Parallel α - β -Search

In the following section three variants of parallel α - β search are presented:

- Parallel “Aspiration Search“
- Parallel “Subtree Evaluation“
- “Distributed Tree Search“

The parallel “Aspiration Search“ defines as many disjoint search windows as processors are available. The narrower the windows the higher is the number of nodes which are potentially pruned. Every processor will never finish later than a single processor with the search window $(-\infty, +\infty)$.

[Qu04] reported that experiments with the parallel “Aspiration Search“ for chess games lead to two major cognitions. The first cognition is that the maximum speedup is typically five or six regardless of the number of the processors. The second cognition is that the parallel “Aspiration Search“ can sometimes lead to super linear speedup when two or three processors are being used.[Qu04][p.396]

The concept of super linear speedup is presented in the next section.

Parallel “Subtree Evaluation“ is another way to compute α - β -Search with the use of several parallel processors. Each processor is allowed to examine its own independent sub tree. This strategy produces two kinds of overheads:

- Search overhead: This overhead depends on the number of nodes which have to be explored to set up the parallelism.
- Communication overhead: This overhead is due to the time to organise the process performing the search.

Both of these overheads are correlated with each other. It is possible to reduce one overhead by expanding the other. The correlation could be demonstrated under assumption of a perfect ordered tree. Without communication between the processors the parallel algorithm prunes much fewer nodes than the sequential one. All parallel processor start the search of their sub trees with the search-window $(-\infty, +\infty)$, where the sequential algorithm can use narrower bounds from searches of preceding

sub trees.

To eliminate the search overhead the processors need topical values for α and β . Without the search overhead the parallel algorithm prunes as many nodes as the sequential search. One approach to achieve this goal is to delay the search of Type-2-Nodes until all searches of Type-1-Nodes have delivered.[Qu04][p.397]

[Qu04] pointed out that practice has shown that delaying the search of some subtrees could reduce the search overhead significantly. Based on this findings the "Distributed Tree Search" (DTS) approach by "Ferguson and Korf" is presented.

In the DTS approach every processor is controlled by a process. The processes are assigned to nodes. At the beginning only one process is working and allocated to the root. At that time this single process controls all processors.

When a process is assigned to a non-terminal node it generates the children and the corresponding legal moves. Then the process assigns processors to the generated nodes based upon a processor allocation strategy. A valid allocation strategy could be the following:

- If the algorithm reaches a Type-1-Node, it allocates all processors to the left-most children.
- When a child node returns the cutoff bounds of its search, the freed processors are allocated to unexplored child nodes in a breadth first manner.
- If all processors are assigned to child nodes, cutoff bounds exists and the process reaches a Type-2 or a Type-3-Node, a new process for each child node is created. Every process has at least one processor.

In case that a process is assigned to a terminal node, it returns the value of that node and the assigned processor back to the parent process. After that the sub process is able to terminate. If a child process finished its work, it reacts like a process assigned to a terminal node. Finished work means, that all sub processes have returned their value and their processors. The process itself sends a message with its α and β values and all processors to its parent.

Parent processes suspend their operations until receiving a message from a child or their parent process.

After a parent received a message it reallocates the freed processes to slower processes which have not finished their work. This results in efficient work load balancing. The awaked parent process may also send α - β values to its children. The advantage of sending the α - β values, is that the sub process could stop its work if their partition could not lead to an optimal mini-max value.

The algorithm ends when the root process terminates. The algorithm has three advantages compared with the parallel “Aspiration Search“:

- All processors stay busy, because blocked processes share a processor with one of its child processes.
- An awaked parent process gains more priority than processes deeper in the search tree.
- If one processor is assigned to a node, the normal α - β search algorithm can be applied.

In order to test the DTS algorithm, “Ferguson and Korf“ have implemented an α - β search for the “Game of Othello“. The board of the game has 8-by-8 grid. The player’s discs have a white and a black side. It is a deterministic, perfect information, zero-sum game of strategy between two players, like chess.[Le95][p. 2]

On the one hand they reported, a speed up of about 10 on 32 processors, for their algorithm. On the other hand the algorithm explores 3 times more nodes than the sequential algorithm.[FK88][p.131]

3.4 Speedup anomalies

Two different speedup anomalies in parallel formulation exist. Sometimes an algorithm with p processors does less work compared to the sequential algorithm. This anomalous behavior is called acceleration anomalie. In contrast, a deceleration anomaly exists, if the parallel algorithm with more than one processor does more work than the sequential one and the speedup is smaller than the number of processors. The situation that the speedup for the parallel formulation compared to the sequential one is greater than the number of processes used, is reffered by super linear speedup.[GV99][p.5,6]

[NK92] reported a super linear speed up for parallel Deep-First-Search if no heuristic information is available to order the successors of a node and the distribution of solutions is nonuniform.

As seen in section 3.3.3, α - β search can sometimes lead to superlinear speedup if 2 or 3 processors are used.

In general the search overhead factor could be a measure for this anomalies. The search overhead factor is formulated as W_p/W_s , where W_p is the number of nodes search by the parallel and W_s the number od nodes searched by the sequential algorithm. On the average the search overhead factor sould be greater or equal to 1.

An average search overhead smaller 1 indicates that the sequential alorithm is not optimal.

4 Summary

It could be seen that combinatorial search is applicable for a variety of decision and optimisation problems on discrete mathematical structures.

Several parallel approaches exist. An effective workload and the minimality are the major issues, which each parallel approach has to deal with. Minimality means the algorithms should explore as few nodes as possible and have only little communication overhead. The competition of parallel search is the effective balancing between communication and initial overhead and the effective workload.

Two different approaches of parallel “Divide & Conquer“ have been presented. The easiest was to maintain all problems in the memory of a single processor. Then the speedup is bounded to the processor’s capacities. The second approach divides the problem among the different memories. The speedup can be much higher, but it could be difficult to balance the workload.

Parallel “Backtrack“ was shown as a more common approach because it is able to find every solution or only one possible. For instance, the approach of parallel “Backtrack“ where the problems and sub problems are held in different memories. It has the advantage to use linear memory in respect to the search depth. It was pointed out that search trees are often imbalanced which reduces the speedup. The more the imbalance the less the speedup. In order to balance the workload, many sub trees are assigned to each process. This is based on the assumption that processes with many subtrees have nearly the same processing-time. Even Dijkstra’s “distributed termination detection“ was introduced to avoid run-time errors. But the discussed parallel “Backtrack“ did not use the knowledge about the problem to avoid exploring unnecessary sub trees.

Chapter 3.2.3 introduces α - β -pruning, an approach which is able to cut off large parts of the search tree. In the best-case this cutoff allows the algorithm to go twice as deep in the search tree as a straight forward mini-max search without pruning. [Qu04] described it as the preferred method to evaluate game trees.

It could be pointed out that parallel algorithms can speed up the combinational search but sometimes accompany with the lost of the special advantages of the sequential computation. Each particular situation has to be analysed to find the right, i.e. the most efficient and fastest algorithm.

A Functions

```
Partition(A, left, right)
1:  i ← left
2:  j ← right + 1
3:  pivot ← A[left]
4:  loop
5:    loop
6:      i ← i + 1
7:    while(A[i] < pivot) repeat
8:    loop
9:      j ← j - 1
10:   while(A[j] > pivot) repeat
11:   until(j < i)
12:   Switch(A[i], A[j])
13:   repeat
14:   Switch(A[left], A[j])
15:   return(j)
```

Algorithm 4: Partition function for Quicksort

Literatur

- [He93] Dominik Henrich: *Initialization of Parallel Branch-and-bound Algorithms*, Second International Workshop on Parallel Processing for Artificial Intelligence (PPAI-93), 29. August 1993, Chambery, France.
- [PM07] Plamenka Borovska, Milena Lazarova: *Efficiency of parallel Minimax Algorithm for Game Tree Search*, International Conference on Computer Systems and Technologies, 2007.
- [GV99] Ananth Grama, Vipini Kumar: *State of the Art in Parallel Search Techniques for Discrete Optimization Problems*, IEEE Transactions on Knowledge and Data Engineering, Vol. 11, No. 1, 1999.
- [Qu04] Michael J Quinn: *Parallel Programming with C with MPI and OpenMP*, p. 369-399, 2004.
- [KB95] G. Kondrak, P. v. Beek: *A theoretical evaluation of selected backtracking algorithms*, 1995.
- [Se88] R. Sedgewick: *Algorithms*, Addison Wesley, 1988.
- [Se78] R. Sedgewick: *Implementing Quicksort Programs*, Communications of the ACM, Vol. 21 No. 10, 1978.
- [KM75] D. Knuth, W. Moore: *An analysis of alpha-beta pruning*, Artificial Intelligence, 6(4):293-326,1997.
- [HS95] Holger Hopp, Peter Sanders: *Parallel Tree Search on SIMD Machines*, Proceedings of the Second International Workshop on Parallel Algorithms for Irregularly Structured Problems, Vol. 980, p. 349-361, 1995.
- [Ma86] T.A. Marsland: *A review of Gadme-tree pruning*, ICCA Journal 9, 1, p.3-19, 1986.
- [NK87] V. Nageshwara Rao, Vipin Kumar, K.Ramesh: *A parallel implementation of Iterative-Deepening-A**, 1987.
- [Ko09] Kolaczyk, E. D.: *Statistical Analysis of Network Data*, Methods and Models, New York, Springer, 2009.
- [Di03] R. Diestel: *Graphentheorie*, Springer, 2003.

- [Le95] Anton Leouski: *Learning of Position Evaluation in the Game of Othello*, 1995.
- [NK92] V. Nageshwara Rao, Vipin Kumar: *On the Efficiency of Parallel Backtracking*, 1992.
- [PP95] Panos M. Pardalos, Leonidas S. Pitsoulis, T. Mavridou, Mauricio G. C. Resende: *Parallel Search for Combinatorial Optimization: Genetic Algorithms, Simulated Annealing, Tabu Search and GRASP*, Lecture Notes In Computer Science; Vol. 980, p. 317-331, 1995.
- [FK88] C.Ferguson, Richard E. Korf: *Distributed Tree Search and its Application to Alpha-beta Pruning* AAAI-88 Proceedings, 1988.
- [PW04] Hilary Priestley and Martin Ward and H. A. Priestley and St. Giles and M. P. Ward and South Rd: *A Multipurpose Backtracking Algorithm*, 1994.

Ich versichere hiermit, dass ich meine Seminararbeit „*Combinatorial search*“ selbstständig und ohne fremde Hilfe angefertigt habe, und dass ich alle von anderen Autoren wörtlich übernommenen Stellen wie auch die sich an die Gedankengänge anderer Autoren eng anlehnenden Ausführungen meiner Arbeit besonders gekennzeichnet und die Quellen zitiert habe.

Münster, den (20.12.2009)

Felix Pothhoff