Parallel Game Tree Search on SIMD Machines

Holger Hopp and Peter Sanders

University of Karlsruhe, 76128 Karlsruhe, Germany
Tel: (49) 721 6084336
Fax: (49) 721 698675
E-mail: {hhopp,sanders}@ira.uka.de

Abstract. We describe an approach to the parallelization of game tree search on SIMD machines. It turns out that the single-instruction restriction of SIMD-machines is not a big obstacle for achieving efficiency. We achieve speedups up to 5850 on a 16K processor MasPar MP-1 if the search trees are sufficiently large and if there are no strong move ordering heuristics. To our best knowledge, the largest speedups previously reported (usually on MIMD machines) are more than an order of magnitude smaller.

Keywords: Parallel game tree search, load balancing, program transformations for SIMD.

1 Introduction

Two-player games with complete information like chess have always been an active area of reasearch in artificial intelligence because they constitute a nontrivial but easy to specify problem area. Since the game tree search algorithms used for most game implementations are very computation intensive, games are also an interesting "benchmark" problem for parallel computing.

The main challenge for parallelization is that the strong tree pruning heuristics like $\alpha\beta$-search used in the sequential case produce very irregular search trees with little or no immediately available parallelism. Early work on parallel game tree search was therefore not able to achieve speedups above 5–10 (e.g. [6]). More recent work is able to exploit todays MIMD machines with processor (PE) numbers up to 1024 [4, 9]. However, a lot of special tuning appears to be necessary such that it is not clear how far the current techniques will lead. Using even larger scale parallelism as available on SIMD-machines has largely failed so far [2].

A principal objective of the work described here is to investigate how game tree search can be parallelized on even larger numbers of PEs. In addition, we use $\alpha\beta$-search as a case study how algorithms with relatively complex control flow can be implemented by an efficient data parallel program. Since the performance of real games like chess or Othello depends on many application specific details not directly related to the $\alpha\beta$-algorithm we restrict ourselves to synthetic game-trees. By changing the parameters of the tree generating process, we are able to model a wide spectrum of possible scenarios.

The structure of this paper is as follows: We first line out in section 2 how sequential $\alpha\beta$-search can be implemented on a SIMD machine without incurring a large overhead to the synchronous control flow. Section 3 describes the key parts of the parallel implementation with a strong emphasis on load balancing. Then, we experimentally evaluate some of the more important aspects of our algorithm in section 4. Finally, section 5 summarizes and discusses the key results.

2 Sequential $\alpha\beta$-Search on SIMD Machines

We now decribe how a recursive function with a highly data dependent control flow such as $\alpha\beta$-search can be decomposed into simple operations which can effectively be executed on a SIMD machine like the MasPar MP-1. The MP-1 consists of a central control unit for computing global values and broadcasting instructions and a large number of simple 4-bit processing elements (PEs) with a local memory. (As opposed to some other SIMD machines, each PE is able to use locally generated addresses). The PEs are interconnected by a router which is able to route arbitrary permutations, a 2D-mesh and a broadcast bus. In addition, the system programming language MPL (an extension of ANSI-C) offers micro-coded primitives for collective operations like reductions and (segmented) prefix operations [13, 14]. We use a configuration with 16384 PEs with 16 KB of memory each.

Figure 1 shows pseudocode for the negamax-variant of recursive $\alpha\beta$-search (closely following [16]). Inputs are a position in the game and a range of results (α, β) (*search window*) which can still influence the overall result (Initially $(-\infty, \infty)$). Output is the quality of the position (Its *value*). The value of a leaf (a final position or a position at some maximal search

```
FUNCTION αβ (position J; integer α,β): integer;
    VAR j,w,value: integer;
BEGIN
    determine successors J.1 ... J.w;
    IF w = 0
        THEN RETURN g(J);              (* leaf evaluation *)
    END ;
    value := α;
    FOR j:=1 TO w DO
        value := max (value, -αβ(J.j, -β, -value));
        IF value ≥ β
            THEN RETURN value;         (* cut *)
        END ;
    END ;
    RETURN value;
END ;
```

Fig. 1. $\alpha\beta$-algorithm (negamax-variant)

depth) is its heuristic evaluation. The value of an interior node is the value of the position after making the "best" move which can be found by evaluating the successors from the point of view of the opponent. Once this value exceeds the upper bound β, no additional search can influence the overall result and we can *prune* the remaining subtrees by returning immediately. The search windows of the recursive calls are set according to the old search window and the best move found so far.

It makes no sense to use this function directly on a SIMD machine. We first eliminate recursion by explicitly managing a search stack. Then, we must make sure that the program contains only a single loop nest. Else, PEs which have finished executing an interior loop would always have to wait for the last PE to finish – resulting in a very high SIMD overhead. This can be done by decomposing the control flow into *elementary operations* containing no loops which depend on local data. For $\alpha\beta$, we have chosen the following operations:

Leaf evaluation (LE): Make a heuristic evaluation of a leaf.
Node Generation (NG): Generate successors of an interior node.
Tree Movement ($\alpha\beta$): Move up and down the tree by popping and pushing the stack and perform maximization.

The top level control flow is now managed by a *state* variable indicating which operation has to be performed next. If every operation properly sets *state* at its end, the control flow can be implemented by a single synchronous *test loop*.

```
WHILE not finished DO
    IF state = LE THEN Leaf evaluation
    IF state = NG THEN Node generation
    IF state = αβ THEN Move in tree
END
```

There is still some SIMD overhead left but a PE waiting for a particular operation must wait for at most two tests.

Additionally, we are free to choose any sequence of tests and we can test cheap or important operations more frequently. It turns out that carefully scheduling the test loop makes it possible to considerably reduce SIMD overhead. For a general discussion of this technique refer to [18]. Additional implementation details are reported in section 3.5 after we have introduced the remaining operations used for load balancing and other purposes.

3 The Parallel Algorithm

The basis of our approach to parallelize game tree search is that every PE runs a sequential $\alpha\beta$-algorithm. Then, the main task of parallelization is to find a load balancing strategy which is able to supply the PEs with subtrees which are relevant for finding the overall result.

In the load balancing procedure a set of PEs (*masters*) send work to other PEs (*slaves*) always using point to point communication. In order to follow the philosophy of creating elementary operations, the load balancing procedure is small and so each master sends only a single subtree to a slave. To distribute two or more subtrees to slaves, the elementary operation "load balancing" must be performed twice or more.

Distributing every possible subtree immediately to other PEs in order to exploit as much parallelism as possible expands lots of nodes, which are pruned by the sequential $\alpha\beta$-algorithm. The ratio N_{par}/N_{seq} where N_{par}, N_{seq} denote the number of nodes expanded by the parallel respectively the sequential algorithm defines the *search overhead*. The main problem of parallel game tree search is to keep the search overhead small *and* achieve good processor utilization. So in this paper we first describe methods how to decrease the search overhead while achieving high parallelism (section 3.1 and 3.2) and then discuss the choice of connections between masters and slaves (section 3.3).

If the parallel expansion of right subtrees begins too early, they are searched with the completely open search window $(-\infty, \infty)$. In the sequential $\alpha\beta$-algorithm these subtrees are searched with a smaller β-value, which allows more cutoffs. This can be explained with Knuth and Moore's node types [11]:

Definition 1. The **node type** of a node J in a game tree is defined as:

$$type(\epsilon) := 1$$

$$type(J.i) := \begin{cases} 1 & \text{if } type(J) = 1 \text{ and } i = 1 \\ 2 & \text{if } type(J) = 1 \text{ and } i > 1 \\ 3 & \text{if } type(J) = 2 \text{ and } i = 1 \\ 2 & \text{if } type(J) = 3 \\ \text{undefined} & \text{else} \end{cases}$$

The nodes with defined node types are the nodes of the *minimal tree*. This tree contains those nodes that must always be visited by the sequential $\alpha\beta$-algorithm. Type-1-nodes are searched with the open search window $(-\infty, \infty)$, type-2-nodes with a search window $(-\infty, -v)$, and type-3-nodes with a search window (v, ∞). The nodes with undefined node type are searched with a search window (v, w), which leads to cutoffs if $v \geq w$ $(v, w \neq \pm\infty)$.

Expanding a node in parallel before knowing α,β-bounds will make this subtree much larger than in the sequential case, because there are less opportunities for cutoffs. For example, expanding a right successor of the root (a type-2 node) in parallel with a search window $(-\infty, \infty)$ will effectively change this node to a type-1-node, and it also changes all (indirect) successor node types accordingly.

3.1 The Young Brothers Wait Concept

In order to decrease the search overhead, we adopt the *Young Brothers Wait Concept* (YBWC) introduced in [4, 5]: A successor $v.J$ of a node v must not be expanded before the leftmost brother $v.1$ is completely evaluated. The YBWC significantly reduces the search overhead, but it decreases the available parallelism.

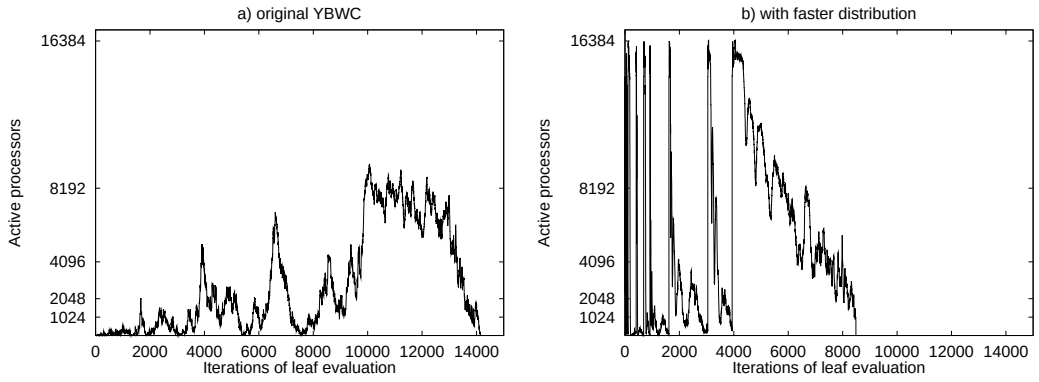

Fig. 2. Active processors

For example, figure 2 a) shows the number of active PEs while expanding a width 7, depth 15 synthetic game tree. In this example, YBWC reduces the search overhead from 236.4 to 2.9, but it also reduces the average processor utilization from 95.2% to 18.8%. The reason is that the YBWC only generates sufficient parallelism in the last phases of the search. When the minimax value of a type-1-node is evaluated, only one PE is active. These nodes are called *synchronization nodes*. Other nodes produce weaker synchronizations, resulting in an overall utilization curve like figure 2 a).

The second approach in [4], the YBWC-1-2, slightly relaxes YBWC. It only delays the parallel expansion at type-1, type-2 and some type-undefined nodes. This increases the average processor utilization, but it also increases search overhead. YBWC-1-2 does not avoid synchronization nodes. The synchronizations are necessary to expand the right successors of a type-1 node with a defined β-value.

3.2 Faster Distribution at Synchronization Nodes

It is important to reactivate all inactive PEs immediately after a synchronization node. A good choice is to distribute all branches of the *right minimal tree*, the unsearched subtree of the minimal tree rooted at the synchronization node. The right minimal tree can be distributed quickly, without communication. Each PE generates the subtree for which it is responsible using only the root node and its PE number [3].

The root's negamax value is evaluated in several sequential phases, one phase for each synchronization node (bottom-up, shown in figure 3). Parallelism is only used within the phases. A preceding phase has evaluated the negamax value of the preceding synchronization node. The current phase evaluates the negamax value of the current synchronization node as follows: The right minimal tree of the synchronization node is distributed to the PEs without communication. After this initialization, the main loop with the dynamic load balancing routine takes over until the negamax value of the synchronization node is known.

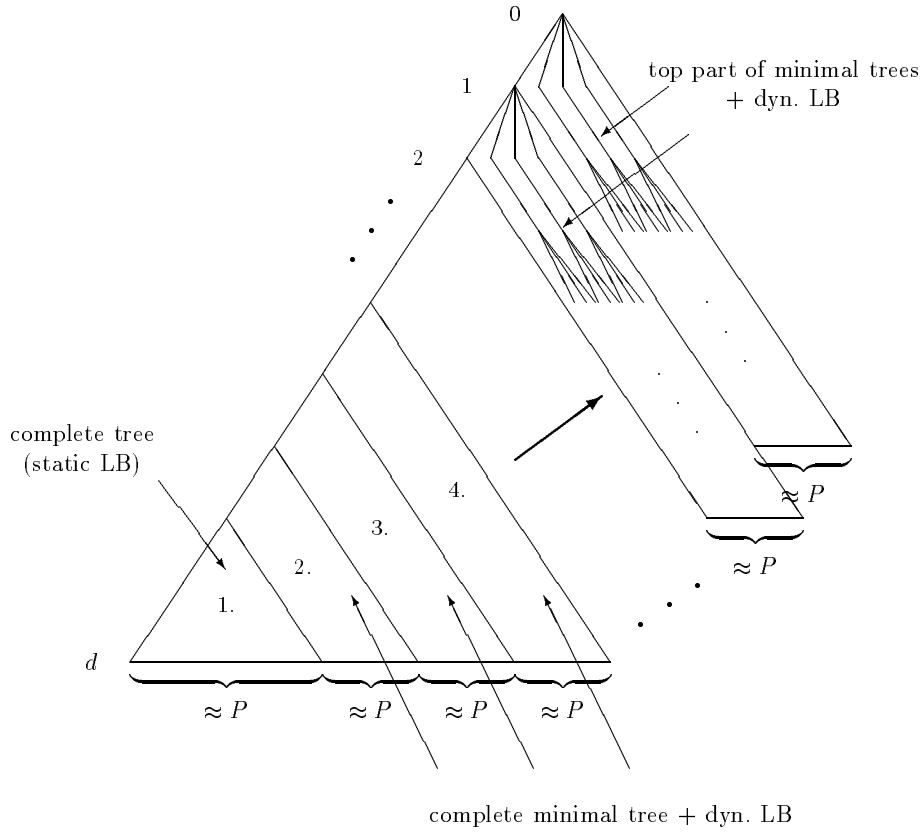

Fig. 3. Phases of the parallel algorithm with faster distribution

In the first phases the right minimal trees (symbolized by the trapezoids in figure 3) are very small. We can distribute the complete minimax tree for the first few phases at once (the triangle in figure 3) and compute its value using a sequence of segmented prefix-min/max-operations. In intermediate phases we can at least distribute the right minimal tree. Later, we distribute the right minimal tree up to a certain depth. The complete algorithm is more complex, for a more detailed description of this algorithm refer to [8].

The faster distribution at synchronization nodes has reduced the number of iterations in the example in figure 2 from 14144 to 8487. The average processor utilization increased from 18.8% to 34.7%. The search time was 39% shorter compared to the pure YBWC.

3.3 Random Polling and Rendezvous Distribution

We made experiments with two algorithms to select communication partners:

Random Polling Every load balancing step determines a new permutation. A master-slave connection is established, if the master has work available for the slave, and the slave is idle.

As shown in [17], a random shift, i. e. connecting PE i with PE $i + k \bmod P$ guarantees a very effective load distribution. Shifts can be routed more efficiently than general permutations on the MasPar.

Rendezvous Distribution The masters and slaves are matched using the rendezvous method introduced in [7] and improved in [15, 10]. The slaves (= idle PEs) are enumerated, yielding an index i, then each slave sends its processor ID to PE i. These IDs are read by the masters to get a definite slave. All potential masters are sorted by "urgency" of the subtrees which can be delivered to slaves. To decrease the cost of the sorting procedure it is useful to sort in a small range, such that a bucket sort algorithm can be used [8].

Random Polling is consuming less time than rendezvous distribution, because it rarely achieves a processor utilization of 100% and it is not sensitive to the urgency of the transmitted subtree. Overall, random polling is faster for small game trees, but rendezvous distribution is faster for large trees, where the processor utilization is high. The division in phases makes it possible to use random polling in the first phases, and rendezvous distribution for the last phases.

We have not defined "urgent" subtrees yet. *Urgency* is relevant for selecting a unexpanded subtree on the master PE, and for sorting in the rendezvous distribution. A good subtree selection strategy is to select the highest unexpanded subtree in the game tree. If several subtrees on the same tree level are available, take the leftmost one. This method achieves good load sharing, but it expands a lot of nodes which are not expanded by the sequential $\alpha\beta$-algorithm. YBWC distinctly decreases this speculative computation, but does not even come close to eliminating it.

Other simple urgency functions besides "top-down, left-right" failed in our experiments. The selection strategy "bottom-up, left-right" reduces the number of expanded nodes by 30–40%, but it is slower by about 50% since it generates too little, and only very fine grained parallelism. The selection strategy "left-right, top-down", which prefers left subtrees regardless of their tree level, incurs more node expansions (7%) as well as a higher run time (50%).

3.4 Further Improvements

This subsection briefly explains additional elementary operations used for improving the performance. For a more detailed description of these algorithms refer to [8].

The elementary operation "passing α,β-values" sends new α,β-values to slave PEs. This makes more cutoffs possible and frees PEs working on pruned subtrees. For all but very small game trees this operation decreases the runtime.

The second improvement is only rarely useful. The *stack merging* operation is used to decrease the number of waiting PEs. A PE has the state *waiting*, if it has fully expanded its subtree and must wait for other PEs which expand other subtrees. The operation merges stacks of waiting PEs to the PEs they are waiting for and frees the waiting PEs for new work. This operation is very expensive, so a low testing frequency in the main loop is necessary. In very large game trees this operation decreases the run time by 0–2%, for smaller and normal-sized game trees stack merging is useless. In experiments the maximum number of waiting PEs was only 700 (of 16384) – too small a number to make this operation useful.

3.5 Scheduling of elementary operations

What elementary operations are necessary for a SIMD game tree search? We need operations to implement the sequential $\alpha\beta$-algorithm (*search operations*) and operations to implement the parallel parts (*management operations*), which use communication between PEs.

We splitted the sequential parts into three elementary operations (section 2): *node generation* (NG), *leaf evaluation* (LE), and *moving within the game tree* ($\alpha\beta$). For management we mentioned the elementary operations *load balancing* (LB), *passing α,β-values* (PV), and *stack merging* (SM). The forth management operation *report result* (RR) sends results from the slaves to their masters.

The relative frequency of NG and LE strongly depends on the kind of the game tree to be searched. Narrow and well ordered game trees need more node generations, others need more leaf evaluations. It is also important whether the search depth is even or odd, particularly for wide game trees. The third criterion is the ratio of execution times of NG and LE. The more expensive operation should be tested less frequently.

The tree moving operation $\alpha\beta$ is very short, so it is a good choice to increase its testing frequency. For example, for some classes of wide trees the sequence

NG; $\alpha\beta$; $\alpha\beta$; LE; $\alpha\beta$; $\alpha\beta$; LE; $\alpha\beta$; $\alpha\beta$;

was a good choice, but for a narrower tree a sequence with two times more NGs than LEs was good. For very wide trees it was favorable to test LE up to four times more often than NG.

A testing sequence with NG, LE, and $\alpha\beta$ is called *basic sequence*. Finding an optimal basic sequence is an important part of SIMD game tree search. Its choice can change the run time by a factor of 2–3. Fortunately, for real game trees it is usually known how wide or how well ordered they are. A good basic sequence can be found with a few experiments. The influence of the tree depth on performance is very small, the influence of the tree width and the NG/LE execution time ratio is larger.

Now a complete testing sequence is created by combining the basic sequence with the management operations. The RR operation is cheap and should be tested often. The YBWC-blocking and deblocking demands a frequent load balancing. In our experiments it was a

good choice to call the LB operation every 1-5 basic sequences (*BS*), depending on the game tree size and shape. This is such a high frequency, that more adaptive strategies for triggering [15, 10] are not worth the additional expense for counting active PEs. The PV and SM operations should be tested more rarely. A complete testing sequence looks like

LOOP *BS*; RR; *BS*; RR; LB; *BS*; RR; *BS*; RR; PV; LB; **END** .

4 Experimental Results

In this section we present some experimental results with synthetic game trees. We look at *regular* trees with fixed depth d and width w for every node. This model is frequently used in the literature [16].

The *leaf value distribution* should simulate successor ordering heuristics with different strength. Two leaf value distributions are used here: *Uniformly distributed* trees simulate game trees without heuristics for ordering successor nodes. The probability that successor $J.k$ is the best is the same for all successors, $f(k) = \frac{1}{w}$. *Geometrically distributed* trees simulate game trees with node ordering heuristics. The probability that successor $J.k$ is the best is

$$f(k) = \begin{cases} p(1-p)^{k-1} & \text{if } 1 \leq k < w, \\ (1-p)^{k-1} & \text{if } k = w, \end{cases}$$

where p is a value in the range [0..1]. The higher p, the higher the quality of ordering. Minimal trees are generated with $p = 1.0$. We used algorithms similar to those in [16] to generate these synthetic game trees.

Speedup and efficiency are measured in relation to the sequential $\alpha\beta$-algorithm on the MasPar. The sequential execution times for larger trees are extrapolated from sequential times on a fast sequential machine. If not otherwise mentioned, all 16K PEs are used.

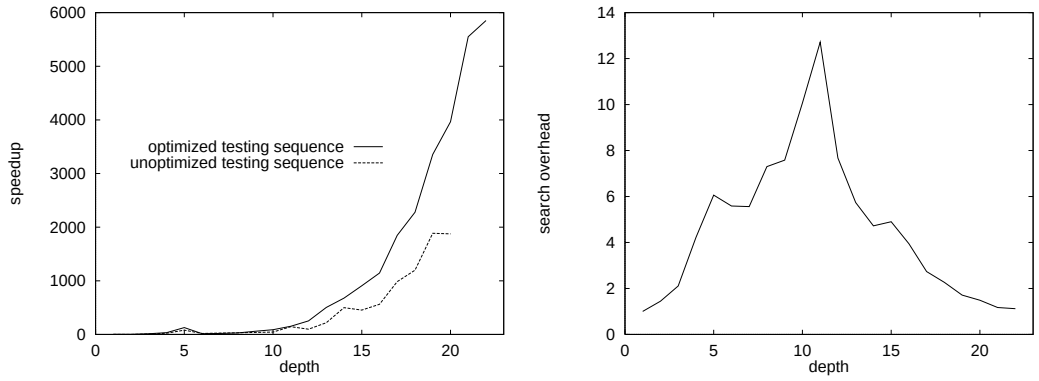

Fig. 4. Uniformly distributed trees (width 7)

Figure 4 shows speedups for uniformly distributed trees of width 7. To achieve high speedups, very large trees must be searched. Some data about a w7d22 tree (width 7, depth 22): The MasPar expands $7.4 \cdot 10^{10}$ nodes in parallel in 1:45 hours, a SPARC 5 (85 MHz) needs 47:24 hours for $6.6 \cdot 10^{10}$. nodes. The MasPar was 27 times faster than the SPARC, but a single PE of the MasPar was 220 times slower than the SPARC. The right curve in figure 4 shows the reasons for the relatively good (for SIMD game tree search) efficiency of 36%. The search overhead is small for the large trees (w7d22: 1.12). The main reason for the small speedups for small trees is the low processor utilization (figure 2), but also the high search overhead of up to 13 (figure 4).

The second speedup curve in figure 4 shows the detrimental effect of suboptimal testing sequences. The low curve is optimized for 0.9-geometrically distributed trees with a more complex leaf evaluation function.

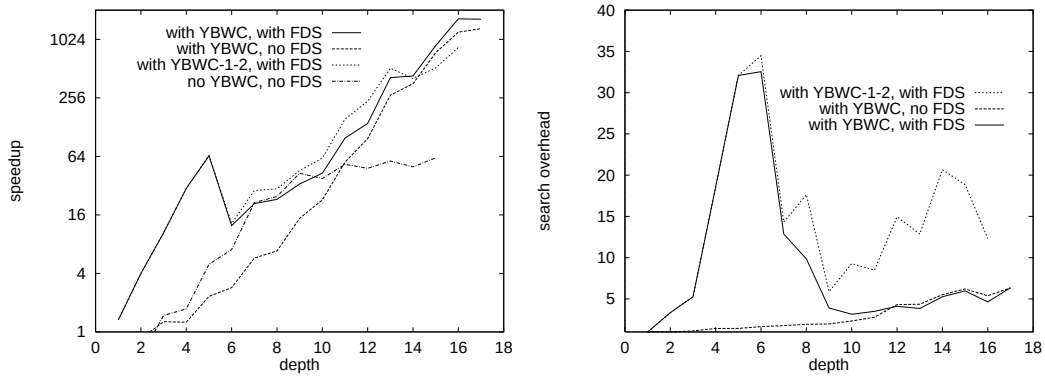

Fig. 5. 0.9-geometrically distributed trees (width 7)

Figure 5 shows speedups for 0.9-geometrically distributed trees of width 7 using different load balancing techniques (logarithmic scale!!). The curves can be divided into four classes of tree sizes:

1. For very small search depths (up to depth 5) it is possible to distribute all leafs to the PEs and evaluate the negamax value with scans, which is done by the faster distribution at synchronization nodes (FDS) from section 3.2. This is an order of magnitude faster than the purely dynamic algorithms.
2. For small search depths (depth 6 to 10) the distribution procedure cannot exhaust the search tree without dynamic load balancing. The speedups are lower than with depth 5, but higher than for pure dynamic algorithms. The algorithm without YBWC still achieves good performance.

   A problem is the drop of speedup at depth 6, because dynamic load balancing is too expensive here. The gap could be filled by using an additional specialized method.

3. For intermediate depths (up to depth 14) YBWC-1-2 with FDS was most successful. The improved PE utilization compared to YBWC makes up for the higher search overhead. The algorithm without YBWC reaches its performance limit.
4. In large trees (depth 15 and more) the pure YBWC algorithms are best. The trees are large enough now to achieve a high processor utilization.

Unfortunately, the speedups for geometrically distributed trees are considerable lower than the speedups for uniformly distributed trees. The reason is the high search overhead of 5–6 shown in figure 5. YBWC cannot decrease the search overhead as effectively as for uniformly distributed trees. Without YBWC the search overhead is up to 160!

We adopt the performance measures introduced in [15] to split the losses of performance into four ratios, which multiplied together yield the efficiency (details in [8]). The four performance ratios (in a range [0..1]) are

fraction of time working The PEs must suspend searching while doing load balancing and other management operations. The *fraction of time working* is the ratio of search time to the total execution time.

processor utilization The average ratio of *working* PEs (which have a subtree for expansion) to the total number of PEs.

raw speed ratio The *raw speed ratio* reflects the inefficiencies of SIMD calculations. Some PEs are inactive in conditional statements, but they are still called working.

work ratio This is the reciprocal of search overhead. It describes performance losses incurred by the expansion of nodes which are not expanded by the sequential algorithm.

Table 1. Ratios and measures

leaf value distribution	uniform		0.9-geometrical	
tree size	w7d16	w7d22	w7d13	w7d17
processor utilization	0.762	0.902	0.577	0.851
fraction of time working	0.825	0.88*	0.790	0.90*
work ratio	0.253	0.893	0.137	0.169
raw speed ratio	0.441	0.50*	0.743	0.78*
efficiency	0.070	0.357	0.046	0.101
speedup	1150	5852	761	1650

Table 1 shows the four performance ratios of four example trees (* = estimated). The losses due to search overhead (work ratio) are the main problem, the SIMD overhead (raw speed ratio) is smaller than one might expected.

Using less than 16K PEs leads to speedups comparable with speedups achieved on MIMD machines. Even for the difficult geometrically distributed trees (w7d13, $p = 0.9$) we achieved a speedup of 141 on 1024 PEs, ignoring the SIMD effects this is comparable with MIMD results [4].

5 Conclusions

The YBWC as an approach to parallel game tree search does have the potential for large scale parallelism. But the game trees have to be very large if redundant work shall not limit the efficiency too much. This problem is particularly severe if the game tree is strongly ordered. Therefore, finding good heuristics for keeping the balance between sufficient parallelism and low search overhead will be a key issue in future research. For real games, additional problems are likely to occur due to hard to parallelize heuristics. For example, transposition tables are so communication intensive that they can only be used in the upper levels of the search trees [4, 9].

A not so severe problem is keeping the PEs busy. The first phases of the search which contain little parallelism due to synchronization nodes can be accelerated using the specialized distribution methods described in section 3.2. This optimization is even more important if a real game is implemented using iterative deepening techniques. The later phases of search contain enough parallelism to be load balanced using standard methods like random polling which are also used in other circumstances [17, 1, 12].

Using SIMD machines for parallel game tree search turned out to be not so much different from using MIMD machines. The negamax variant of $\alpha\beta$-search can be broken down into few simple operations. By carefully scheduling these operations together with the required communication operations, the SIMD overhead can be kept quite small. As long as the node evaluation function does not have a very complex structure, we expect this experience to transfer to real games. However, a philosophy behind SIMD, namely to achieve maximal raw performance by employing massive amounts of slow but cheap PEs is not very successful for game tree search. For difficult to parallelize problems, it is more cost effective to employ fewer but faster PEs even if the theoretical peak performance is lower.

References

1. R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *FOCS*, 1994.
2. V. Cung and L. Gotte. A first step towards the massively parallel game-tree search. In *International Workshop on Parallel Processing for Artificial Intelligence*, pages 88–93, Chambery, 1993. Elsevier.
3. O. I. El-Dessouki and W. H. Huen. Distributed enumeration on between computers. *IEEE Transactions on Computers*, C-29(9):818–825, September 1980.
4. R. Feldmann. *Game Tree Search on Massively Parallel Systems.* PhD thesis, Universität Paderborn, August 1993.
5. R. Feldmann, P. Mysliwietz, and B. Monien. Studying overheads in massively parallel min/max-tree. In *ACM Symposium on Parallel Architectures and Algorithms.* ACM, 1994.
6. R. Finkel and J. Fishburn. Parallelism in alpha-beta search. *Artificial Intelligence*, 19:89–106, 1982.
7. W. D. Hillis. *The Connection Machine.* Series in Artificial Inteligence. MIT Press, Cambridge, MA, 1985.
8. H. Hopp. Parallele Spielbaumsuche auf SIMD-Rechnern. Diplomarbeit, Universität Karlsruhe, Feb. 1995.
9. C. F. Joerg and B. C. Kuszmaul. Massively parallel chess. In *Third DIMACS parallel implementation challenge workshop*, pages 299–308. Rutgers University, 1994.

10. G. Karypis and V. Kumar. Unstructured tree search on SIMD parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 5(10):1057–1072, 1994.
11. D. Knuth and W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.
12. V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing. Design and Analysis of Algorithms.* Benjamin/Cummings, 1994.
13. MasPar Corporation. *MasPar System Overview*, July 1992.
14. MasPar Corporation. *MPL Reference Manual*, May 1993.
15. C. Powley, C. Ferguson, and R. E. Korf. Depth-fist heuristic search on a SIMD machine. *Artificial Intelligence*, 60:199–242, 1993.
16. A. Reinefeld. *Spielbaum-Suchverfahren.* Informatik-Fachberichte, Band 200. Springer-Verlag, 1989.
17. P. Sanders. A detailed analysis of random polling dynamic load balancing. In *International Symposium on Parallel Architectures Algorithms and Networks*, pages 382–389, Kanazawa, Japan, 1994. IEEE.
18. P. Sanders. Emulating MIMD behavior on SIMD machines. In *International Conference Massively Parallel Processing Applications and Development.* Elsevier, 1994.