A Comparison of Parallel Search Algorithms based on Tree Splitting

Paolo Ciancarini

Technical Report UBLCS-94-14

May 1994

Laboratory for Computer Science
University of Bologna
Piazza di Porta S. Donato, 5
40127 Bologna (Italy)

The University of Bologna Laboratory for Computer Science Research Technical Reports are available via anonymous FTP from the area `ftp.cs.unibo.it:/pub/TR/UBLCS` in compressed PostScript format. Abstracts are available from the same host in the directory `/pub/TR/UBLCS/ABSTRACTS` in plain text format. All local authors can be reached via e-mail at the address *last-name*`@cs.unibo.it`.

UBLCS Technical Report Series

93-1 *Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanism*, by Ö. Babaog;ˇlu and K. Marzullo, January 1993.

93-2 *Understanding Non-Blocking Atomic Commitment*, by Ö. Babaog;ˇlu and S. Toueg, January 1993.

93-3 *Anchors and Paths in a Hypertext Publishing System*, by C. Maioli and F. Vitali, February 1993.

93-4 *A Formalization of Priority Inversion*, by Ö. Babaog;ˇlu, K. Marzullo and F. Schneider, March 1993.

93-5 *Some Modifications to the Dexter Model for the Formal Description of Hypertexts*, by S. Lamberti, C. Maioli and F. Vitali, April 1993.

93-6 *Versioning Issues in a Collaborative Distributed Hypertext System*, by C. Maioli, S. Sola and F. Vitali, April 1993.

93-7 *Distributed Programming with Logic Tuple Spaces*, by P. Ciancarini, April 1993.

93-8 *Coordinating Rule-Based Software Processes with ESP*, by P. Ciancarini, April 1993.

93-9 *What is Logic Programming good for in Software Engineering*, by P. Ciancarini and G. Levi, April 1993.

93-10 *Scheduling Real Time Tasks: A Performance Study*, by F. Panzieri, L. Donatiello and L. Poretti, May 1993.

93-11 *Specification and Detection of Behavioral Patterns in Distributed Computations*, by Ö. Babaog;ˇlu and M. Raynal, May 1993.

93-12 *Interaction Systems II: The Practice of Optimal Reductions*, by A. Asperti and C. Laneve, May 1993.

93-13 *Reliability Analysis of Tree-Based Structures and its Application to Fault-Tolerant VLSI Systems*, by Marco Roccetti, June 1993.

93-14 *Guard Compilation in Logic Shared Dataspace Languages*, by M. Garpari, June 1993.

93-15 *Data Algorithm: A Numerical Method to Extract Shape Information from Gray Scale Images*, by R. Davoli and F. Tamburini, June 1993.

93-16 *Towards Performance Evaluation in Process Algebra*, by R. Gorrieri and M. Roccetti, July 1993.

93-17 *Split and ST Bisimulation Semantics*, by R. Gorrieri and C. Laneve, July 1993.

93-18 *Multilanguage Interoperability*, by G. Attardi and M. Gaspari, July 1993.

93-19 *HERMES: an Expert System for the Prognosis of Hepatic Diseases*, by I. Bonfà, C. Maioli, F. Sarti, G.L. Milandri and P.R. Dal Monte, September 1993

93-20 *An Information Flow Security Property for CCS*, by R. Focardi and R. Gorrieri, October 1993.

93-21 *A Classification of Security Properties*, by R. Focardi and R. Gorrieri, October 1993.

93-22 *Real Time Systems: A Tutorial*, by F. Panzieri and R. Davoli, October 1993.

93-23 *A Scalable Architecture for Reliable Distributed Multimedia Applications*, by F. Panzieri and M. Roccetti, October 1993.

93-24 *Wide-Area Distribution Issues in Hypertext Systems*, by C. Maioli, S. Sola and F. Vitali, October 1993.

93-25 *On Relating Some Models for Concurrency*, by P. Degano, R. Gorrieri and S. Vigna, October 1993.

93-26 *Axiomatising ST Bisimulation Equivalence*, by N. Busi, R. van Glabbeek and R. Gorrieri, December 1993.

93-27 *A Theory of Processes with Durational Actions*, by R. Gorrieri, M. Roccetti and E. Stancampiano, December 1993.

A Comparison of Parallel Search Algorithms based on Tree Splitting

Paolo Ciancarini[1]

Technical Report UBLCS-94-14

May 1994

Abstract

We have used a distributed programming environment to compare a number of parallel search algorithms in the domain of chess. The environment runs on a network of workstations; it is based on Linda for coordination of the search and on GNUChess for chess knowledge. In order to make a practical comparison, we have built several parallel programs using different search algorithms. We have formalized and classified the concept of parallel search algorithms based game tree splitting. The algorithms have been classified in two main classes: static and dynamic. We show that on a workstation network static algorithms are more efficient than dynamic ones. Our results confirm and broaden older works by Marsland, Campbell, Schaeffer, and others.

1. E-mail: cianca@cs.unibo.it

1. Introduction

The alphabeta algorithm is the basis of most sequential programs that search game trees. Its efficiency is decisive in chess playing programs, whose playing strength can be precisely measured using the Elo system [Elo86]. For instance, some experiments found that improving 5 times the running time of such an algorithm the playing strength of a chess programs improves about 200 Elo points [Tho82]. One could expect that multiprocessors and multicomputers should be very helpful in building stronger chess programs. However, it is well known that the efficient parallelization of alphabeta is a difficult problem [FinFis82]. Several different ways to parallelism in chess playing programs have been explored; a short survey is found in [BalVRe86]. The main methods are:

i) Parallel aspiration search [Bau78]: the alphabeta window is partitioned in a number of contiguous segments, that are used by different processors to explore the same game tree; this is a form of OR-parallelism.

ii) Parallel evaluation by special hardware; *e.g.*, HITECH uses parallel hardware for move generation [Ebe86], while Deep Thought [Hsu90] uses parallel hardware for evaluating positions.

iii) Mapping the search space on the processor set by a hash function: this is advantageous if the search space contains several duplicate states; *e.g.*, [Sti91] describes how the whole search space of some chess endings was mapped on a Connection Machine.

iv) Parallel search of "splitted" game tree: the game tree is decomposed assigning its nodes to different search processes.

The last method is the most widely studied and used. For instance, in [MarCam82] several search algorithms were described and compared; other experiments with different algorithms are described in [MOS86]. However, after those seminal papers we found no newer survey works on comparing parallel algorithms, although new ones have been invented. We decided to test the flexibility and the performance of a modern parallel language in implementing such algorithms on a general purpose network.

Thus, in this paper we compare some parallel search algorithms based on tree splitting, implementing them all in the same hw/sw environment, *i.e.,* C-Linda [SCA90] on a network of workstations. We have developed a chess-oriented software library based on a publicly available program, namely GNUChess (courtesy of Open Software Foundation). Using these tools we have built several parallel playing programs, embedding different parallel search algorithms; in order to simplify comparison with results reported in other papers, our experiments were made using a standard set of tests [BraKop82].

This paper has the following structure: in Sect. 2 we define the concept of static and dynamic game tree splitting; in Sect. 3 we discuss how parallel search algorithms can be programmed in Linda; we summarise our experiments on static splitting algorithms in Sect.4, whereas Sect.5 contains data for dynamic splitting algorithms. In Sect.6 we make a number of comparisons with results obtained by other researchers.

2. A classification of splitting criteria

There are several ways for handling a game tree during a parallel search, that basically differ for the splitting rules they use. We define some concepts in order to highlight properties of such splitting rules. Consider the following definitions:

Def. *Splitting*: Let A be a game tree and $N=\{N_1, N_2, \ldots, N_n\}$ its set of nodes; let $P=\{P_1, P_2, \ldots, P_m\}$ a set of processes. A *splitting* for A in P is a function $D:N\rightarrow P$ which maps each node of A to a process in P.

Def. *Explorer and supervisor processes*: Let D be a splitting for the game tree A to the set P of processes; let N be the nodes of A and N_i a generic node ($N_i \in N$). We call *explorer* of N_i (according to D) the process $D(N_i)$. Moreover, let *father*: $N\rightarrow N$ be a function that associates a node with its predecessor node (we define the root node as predecessor of itself); we call *supervisor* of N_i (according to D) the process $D(father(N_i))$.

After a splitting the evaluation of a subtree is assigned to a specific process named *explorer*. For instance, let (N_i,P_j) be a pair in D; since N_i fully identifies the subtree S_i of which it is the root, the

minimax value for subtree S_i must be computed by explorer P_j. To evaluate a subtree S_i the explorer P_j has to know the minimax values of the subtrees whose roots are the successors of N_i; function D defines an explorer for each of such subtrees as well. Let N_s be one of N_i's successors and P_s=D(N_s) the corresponding explorer. Which relation is established between P_j and P_s? There is a dependence of P_s from P_j because the minimax value of the subtree assigned to P_s is used by P_j, and P_s must report to this process. We call P_j *supervisor* of N_s because it manages N_s'minimax value.
Thus, a game tree splitting establishes a hierarchic relationship among processes. Such a relationship changes dynamically (i.e., during the tree visit) depending on the relationship among subtrees visited by the processes at any time. In fact, it may happen that, in a subsequent stage of the visit, P_i will have to communicate to P_s the result of its search because the subtree it visited is included in the subtree explored by P_s .

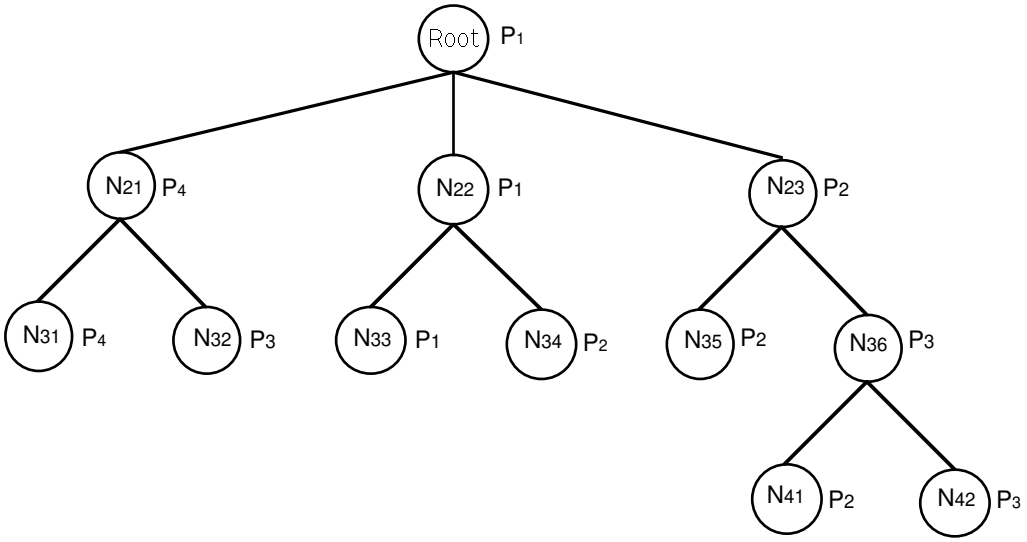

Fig.1 **Splitting a game tree**

Fig.1 shows a game tree splitting; nodes are labelled with the name of the explorer of the subtree of which they are root. Some processes are explorers of several subtrees; nevertheless, at any instant during the search, they will be engaged in the visit of no more than one subtree. Therefore, the description of a splitting should include the visit order followed by each process; this information represents the "dynamic" component of the splitting. In Fig.2 it is shown an example (with reference to the game tree in Fig.1).

P_1: $N_{33} \rightarrow N_{22} \rightarrow$ Root
P_2: $N_{35} \rightarrow N_{41} \rightarrow N_{23} \rightarrow N_{34}$
P_3: $N_{32} \rightarrow N_{42} \rightarrow N_{36}$
P_4: $N_{31} \rightarrow N_{21}$

Fig.2 **Splitting: evaluation order by every process**

In general, it is impossible to fix the visiting order of every subtree, because it depends on the processes' relative speeds. Nevertheless, there are some synchronisation points marking the stages of

the visit: a process can completely search a subtree S only after having received the minimax values of all S'subtrees. Fig.3 depicts a snapshot of the search of game tree in Fig.1, with the splitting specified in Fig.2.

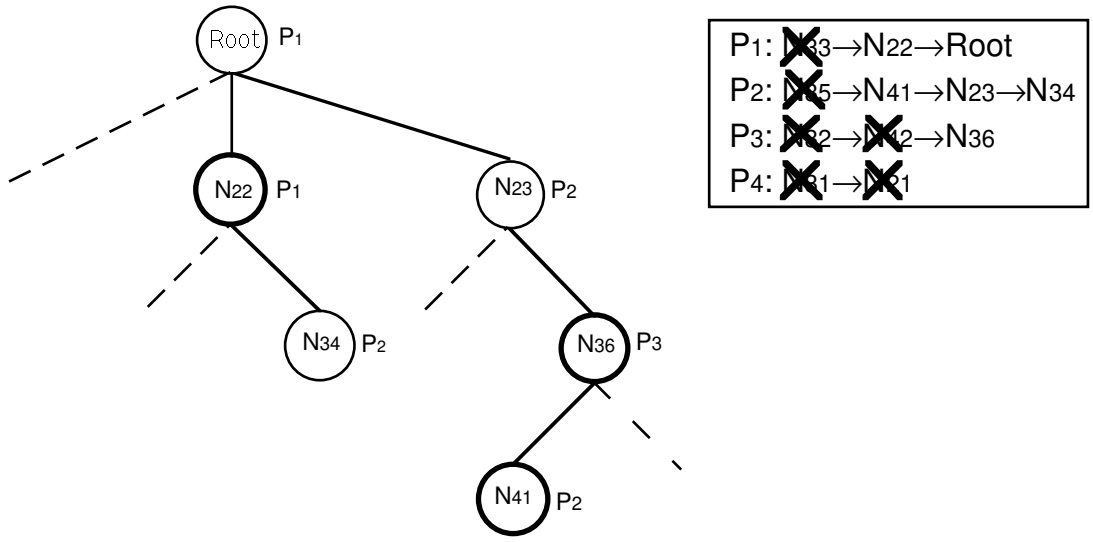

Fig.3 **Snapshot of a splitting during search**

In Fig.3 process P4 is idle because it has completed its own list of tasks; P1 is waiting for N34 minimax value from process P2; likewise, P3 is waiting for N41 value from P2. Therefore, P2 is the only process working at present: it is evaluating N41. There is a hierarchic relation between P3 and P2: it will be reversed when, having completed N36 evaluation, P3 sends a message to P2 .
The main problem in searching a game tree by a splitting method is to decide when the subtree-process association should be established. Usually, it makes no sense to allocate subtrees to processes before search starts, because of the following reasons:

- it is necessary to know the whole tree;
- the $\alpha\beta$ algorithm property of cutting subtrees makes useless the visit of some of them. It is impossible to determine *a priori* the identity of these subtrees and then to obtain at run-time a uniform load distribution.

At search time we usually have the following scenery: some "lucky" processes complete search before others (*e.g.*, because of better cuts) and will remain idle. Every solution for avoiding idleness must be dynamic, *i.e.*, it must provide mechanisms able, during the search, to reassign to idle processes subtrees already assigned to overloaded processes. Therefore, a classification of splitting methods in static and dynamic is not really meaningful, because actually the only plausible solution is dynamic. On the other hand, it makes sense to distinguish different approaches according to the time they establish nodes where it will occur a splitting.

Def. *Splitting node*: Let D be a splitting for game tree A; let N_i be a node of A; N_i is a *splitting node* (according to D) $\Leftrightarrow \exists N_j$: N_i=father(N_j) and $D(N_j) \neq D(N_i)$.

A splitting node is one at which it is established the generation of a parallel search task, *i.e.*, where some of its subtrees are visited by processes different from its explorer. We can now state the following classification:

- a *static* splitting algorithm decides all the splitting nodes before the search starts;
- a *dynamic* algorithm decides some splitting nodes only during the search.

3. Parallel search on a network of workstations

High-level parallel programming languages are gaining momentum, because they simplify the task of building software for the novel architectures with several processors. Parallel search algorithms are good tests for these new languages.
Parallel search algorithms have been programmed in several languages; for instance, in [ABD82] we find an algorithm based on semaphore primitives; in [Eli90] the Orca language was used. In this paper we will use Linda, a coordination language that consists of a small set of parallel programming primitives that can be added to any sequential language [CarGel90]. These primitives associatively access a *tuple space* shared among processes.
There are several possible software architectures that can all be programmed in Linda; one of the most useful is the *master-worker* model. According to this model tasks created during the parallel evaluation of a program are dynamically distributed among processors. The parallel program should be structured as follows:

- a *master* process generates tasks and coordinates the collection of solutions;
- several identical *worker* processes pick and execute tasks and return task results to the master; when a task is terminated, a new task can be chosen;
- in the tuple space there are two main kinds of tuples: active tuples for tasks (we call this multiset the agenda), and passive tuples that represent task results.

The master-worker model allows to structure the supervisor of a node as a master process that coordinates several explorers as workers. This is simple to model: in the following program only the root is a splitting node.

```
int master_main(int n_worker,int depth)
{
position *root,*successor;
int nmoves,minimax,i,master(),worker();
for (i=0;i<n_worker;i++)
   eval ("worker",worker(depth));  /* create n workers */
while (!end())
   {
   load_new_position (&root); /* a new game tree */
   nmoves=genmoves(root,&successor);
   if (nmoves==0)   /* no search to do? */
      return(evaluate(root));
   out ("root",*root); /* put new tree in agenda */
   out ("sync",n_worker);
   for (i=0;i<n_worker;i++)
      out ("job",NEW_POSITION); /* new job */
   in ("sync",0); /* master continues only after all workers are ready
*/
   minimax=master(n_worker,nmoves,depth,successor);/* search */
   in ("root",*root);  /* delete visited tree */
   }
for (i=0;i<n_worker;i++)
   {
   out("job",QUIT);  /* end of task */
   in("worker", 0);  /* delete worker */
   }
return (0);
}

int master(int n_worker,int nmoves)
{
int nmoves,local_score,subtree,free,best;
local_score=-INFINITE;  /* initialize local score */
out ("score", local_score);  /* initialize global score */
free = n_worker;
subtree = 1;
```

```
while ((subtree <= nmoves) || (free<n_worker))
                  /* loop terminates when job done */
   if ((subtree <= nmoves) && (free>0))
            /* more tasks in agenda and a worker is idle? */
      { out("job",subtree); /* insert search task in agenda */
      free--;
      subtree++;
      }
   else     /* wait completion of a task */
      {
      in ("result",?value,?r_subtree);  /* get result */
      if (value>local_score)  /* improved score? */
         { local_score=value;     /* local update of score */
         in ("score",?int);
         out("score",local_score); /* global update of score */
         best=r_subtree;
         }
      free++;}
in ("score",?int); /* to avoid interference with next search */
return (local_score);
}
```

Fig.4 **Core algorithm: master**

Creating and terminating an active tuple for each task is an expensive activity in Network Linda, thus we have defined a general worker structure that can survive the end of a task and can be specialized when it is necessary.

```
int worker(int depth)
{
position *root,*successor,*tree_pointer;
int nmoves,subtree,score,value,job_type,quit,s;
quit=false;
while(!quit)
   {
   in("job",?job_type);
   switch (job_type)
      {
      case QUIT:  /* terminate */
         quit=true;
         break;
      case NEW_POSITION: /* new game tree */
         in("position",?*root);
         in("sync",?s);
         out("sync",s-1);
         nmoves=genmoves(root,&successor);
         break;
      default: /* sequential αβ */
         subtree=job_type;
         tree_pointer=successor+subtree-1;
         makemove(tree_pointer);
         rd("score",?score);  /* update local score */
         value=-alphabeta(tree_pointer,-INFINITE,-score,depth);
```

```
            out("result",value,subtree);
            undomove(tree_pointer);
            break;
         }
      }
   return (0);}
```

Fig.5 **Core algorithm: worker**

Our goal is to compare some parallel search strategies with their sequential versions. We have developed and tested different programs all based on such an algorithm. The experiments we describe used the Bratko-Kopec positions [BraKop82] to measure the speedup on an increasing number of workstations.
The most important value we measured is time T to search a game tree. For sequential versions, this is defined as the interval between the root invocation of $\alpha\beta$ and its termination (we actually measure CPU time used for such an evaluation). For the parallel versions, we let the master to measure time between the distribution of the root node and the collection of its minimax value (for such an evaluation we used real time; the network was used during unloaded hours). Other parameters we recorded are shown in Fig.6.

Procs	processors used for the search (master+workers)
T	total search time in seconds for all 24 positions
N	total searched nodes for all 24 positions
S	speedup $(= \frac{T_1}{T_{nr.proc}})$
E	relative efficiency $(= \frac{S}{nr.proc})$
Fpm	average production factor
Fpmw	average production factor for workers
SO	search overhead $(= \frac{N_{nr.proc}}{N_1} - 1)$
SS	search speed $(= \frac{N}{T})$
SSS	speed gain $(= \frac{VR_{nr.proc}}{VR_1})$

Fig 6 *Parameters we measured in the experiments*

All experiments were done fixing a search depth of 5 plies, for comparison with other papers that used the same search depth. We have used 11 Sun Sparcstations in periods when they were free of other tasks.

4. Static splitting

Let us discuss preliminarly a few results we can expect from our experiments. Let B the branching factor typical of chess game trees, and N_w the number of processors available for explorers:

- the core algorithm efficiency should be proportional to the ratio B/N_w [MarCam82];
- for a game tree with B=40, an $\alpha\beta$ search is equivalent to a minimax search of a tree with branch factor B=7 [Gil72]. Thus, if we had N_w=40 explorers, the ideal average speedup should be 7.

Procs	T(sec)	N (nodes)	S	E	Fpm	Fpmw	SO	SS (nodes/sec)	Sss
1	8301	5347545	---	---	---	---	---	644	---
3	4571	5754203	1.82	0.61	0.66	0.98	0.08	1259	1.95
5	2770	6599460	3.00	0.60	0.75	0.93	0.23	2382	3.70
7	2242	7236560	3.70	0.53	0.71	0.83	0.35	3228	5.01
9	1988	7876769	4.18	0.46	0.68	0.77	0.47	3962	6.15
11	1786	8521288	4.65	0.42	0.66	0.73	0.59	4771	7.41

Fig. 7 *Results of experiments for core algorithm* (basic master worker)

In Fig. 7 we show data for the core algorithm. The reported speedup is low; moreover, values for E tell that adding processors we increase all overheads:

- search overhead: compare real speedup S with ideal speedup S_{ss}. For instance, with 11 processors we have a 37% loss (S_{ss}=7.41 vs S=4.65). This loss is caused by minimum cooperation. We also remark that search overhead increases with more processors because all workers start with value = -∞ , whereas the sequential search has a good approximation of the value starting from the second subtree.
- communication and synchronization overhead: F_{pm} increases until 5 processors (75%) then decreases; for 11 processors the F_{pm} value is almost the same as with 3 processors (66%). This is explained by the fact that the master is idle most of the time, so if we use one process per processor a processor is idle as well.

Procs	T (sec)	N (nodes)	S	E	Fpm	SO	SS (nodes/sec)	Sss
1	8301	5347545	---	---	---	---	644	---
3	4544	5860668	1.83	0.61	0.70	0.10	1290	2.00
5	3173	6330053	2.62	0.52	0.66	0.18	1995	3.10
7	2779	6867688	2.99	0.43	0.61	0.28	2471	3.84
9	2517	7310806	3.30	0.37	0.57	0.37	2905	4.51
11	2310	7990369	3.59	0.33	0.54	0.49	3459	5.37

Fig.8 *Static splitting, master does part of the search*

We should allocate a worker on the master processor; unfortunately, Fig.8 shows that this is not a good idea, because a master process should be able to generate new tasks rapidly when other processors become idle: when a processor is shared by the master and a worker the answer times for master become too slow.

Procs	T (sec)	N (nodi)	S	E	Fpm	SO	SS (nodi/sec)	Sss
1	8301	5347545	---	---	---	---	644	---
3	3298	6156027	2.52	0.84	0.96	0.15	1867	2.90
5	2382	6945194	3.48	0.70	0.91	0.30	2916	4.53
7	2064	7524137	4.02	0.57	0.81	0.41	3645	5.66
9	1798	8186074	4.62	0.51	0.77	0.53	4553	7.07
11	1751	8746752	4.74	0.43	0.71	0.64	4995	7.75

Fig.9 *Static splitting, no master (democratic cooperation)*

The core algorithm can be rewritten to get a more efficient program: there is no need to maintain alive the master if the workers are able to handle the results. We call this variant of master-worker "democratic coperation". Data shown in Fig.9 show an improvement with respect to the pure master-worker paradigm. However, such data also show that search overhead increased, because a) there is

one more process cooperating in the search, and b) each worker starts executing a new task more rapidly, so it is less likely that it can use data from other explorers that finish while it is idle. The most important consideration is that the algorithms get similar results as we increase processors. In fact, let P be the total number of processors and E(P) number of exploring processors; let C(P)be: $C(P) = \frac{E_{master\ -\ worker}(P)}{E_{democratic}(P)} = \frac{P-1}{P}$. The ratio among production factors is proportional to C(P), whereas the ratio among value of search overhead is inversely proportional; since $\lim_{p\to\infty} C(P) = 1 \quad and \quad \lim_{p\to\infty} \frac{1}{C(P)} = 1$ the two algorithms are asyntotically equivalent.

4.1 Ordered game trees: iterative deepening and top level sorting

In the algorithms presented above there is no sorting of nodes; however, it is well known that the execution of αβ algorithm improves if game tree nodes are ordered [MarCam82]. We have implemented such a sorting, and the resulting search algorithm has been tested with the following parameters for iterative deepening: INIT=2-plies, MAX=5-plies; thresold for parallel search: 3-plies. The sequential algorithm we used for comparison is αβ with iterative deepening and same INIT and MAX.

Procs	T (sec)	N (nodes)	S	E	Fpm	SO	SS (nodes/sec)	Sss
1	6740	4529971	---	---	---	---	672	---
3	2884	5415125	2.34	0.78	0.94	0.20	1878	2.79
5	2229	6307062	3.02	0.60	0.87	0.39	2830	4.21
7	2007	7317370	3.36	0.48	0.80	0.62	3646	5.42
9	1839	8187769	3.67	0.41	0.75	0.81	4452	6.62
11	1825	8980505	3.69	0.34	0.71	0.98	4921	7.32

Fig.10 *Basic algorithm, iterative deepening and top-level sorting*

The most impressive result in Fig.10 is the increasing of search overhead SO (98% with 11 processors). The number of visited nodes is larger for 9 and 11 processors. On the other hand, the sequential version benefits from the new heuristics: the visited nodes are decreasd of 17%. We explain this with the fact that parallel search initially cannot gain any advantage from top level sorting, since all processes start simultaneously with initial score equal to -∞.
Moreover, with respect to no iterative deepening, in this case visited nodes include also all internal nodes: this explains the total count with 11 processors. However, the average production factor Fpm decreases of 2%: there is a larger synchronization overhead due to search in the internal tree.

4.2 Reducing the search overhead

Search overhead is caused:

- by lack of sharing of score during search;
- initial score = -∞ for the first Nw tasks (Nw = # explorers).

We try to overcome such problems introducing a tuple storing a global score that all local searches can query. The global score tuple that can be read at any moment, but we must be careful with reading such a tuple, because it may be a very expensive operation. Thus, we try to make only "useful" readings: after each reading of the global score the explorer puts in the tuple space a private tuple. When a worker modifies the global score it removes all private tuples, so that they can know when an update was done. In C-Linda `out` operations are local and inexpensive, whereas `in` are based on broadcasting and are expensive. Deletion of private tuples is expensive as well, but it does not happen frequently. Fig.11 shows the evaluation of such an algorithm.

Procs	T (sec)	N (nodes)	S	E	Fpm	SO	SS (nodes/sec)	Sss
1	6740	4529971	---	---	---	---	672	---
3	2689	4995937	2.51	0.84	0.94	0.10	1858	2.76
5	1854	5346599	3.64	0.73	0.88	0.18	2884	4.29
7	1485	5634915	4.54	0.65	0.81	0.24	3795	5.65
9	1315	5936185	5.13	0.57	0.77	0.31	4514	6.72
11	1280	6726213	5.27	0.48	0.74	0.48	5255	7.82

Fig.11 *Score sharing*

This table shows that we have achieved the goal of reducing search overhead SO (62% with 9 processors); the speedup with 11 processors was improved of 42% (3.69 → 5.27). Values for Fpm are approximated: they do not take in account overhead due to updating the local score. We can evaluate such an overhead as follows:

• with P processors, let Fpmr be the real production factor and OC due to score sharing; we have that:

$$S_{VR} = P \cdot F_{pmr} \Rightarrow F_{pmr} = \frac{S_{VR}}{P}$$

$$F_{pm} = F_{pmr} + OC \Rightarrow OC = F_{pm} - F_{pmr}$$

• in particular, with 11 processors:

$$F_{pmr}(11) = \frac{S_{VR}}{P} = \frac{7.82}{11} = 0.71$$

$$OC(11) = F_{pm} - F_{pmr} = 0.74 - 0.71 = 0.03$$

Communication overhead is not negligible, however the improvements in search overhead are worth such a price.

4.3 PVSplit

We have tested PVSplit with iterative deepening (INIT=2-plies and MAX=5-plies) and top-level sorting; moreover, THRESOLD=3-plies. We have made two sets of experiments, with or without global score.

procs	T (sec)	N (nodes)	S	E	Fpm	SO	SS (nodes/sec)	Sss
1	6740	4529971	---	---	---	---	672	---
3	2693	4949981	2.50	0.83	0.91	0.09	1838	2.73
5	1931	5349308	3.49	0.70	0.82	0.18	2770	4.12
7	1659	5657469	4.06	0.58	0.75	0.25	3410	5.07
9	1502	6130143	4.49	0.50	0.67	0.35	4081	6.07
11	1446	6445828	4.66	0.42	0.60	0.42	4458	6.63

Fig. 12 *PVSplit without score sharing*

PVSplit reduces search overhead with respect to score sharing (see Fig.11), however its efficiency is lower. Compare the production factors: PVSplit gives very low values (14% with 11 processors). The reason is the synchronization necessary along the principal variation; we can reduce such an overhead with a more balanced load distribution.

We also have to take into account the fixed cost of each splitting: communication overhead has to be multiplied for each node in the principal variation that is used for parallel splitting.

Procs	T (sec)	N (nodes)	S	E	Fpm	SO	SS (nodes/sec)	Sss
1	6740	4529971	---	---	---	---	672	---
3	2640	4770665	2.55	0.85	0.91	0.05	1807	2.69
5	1797	5020905	3.75	0.75	0.84	0.11	2794	4.16
7	1443	5228542	4.67	0.67	0.77	0.15	3623	5.39
9	1255	5358664	5.37	0.60	0.71	0.18	4270	6.35
11	1210	5704328	5.57	0.51	0.64	0.26	4714	7.01

Fig.13 *PVSplit with score sharing*

Applying score sharing to PVSplit we reduce search overhead without loss (Fig.13): in fact, these heuristics handle different parts of the game tree:

- PVSplit is used in search of subtrees 2..P (if we use P processors)
- score sharing is used for subtrees after the Pth subtree.

This algorithm gives the best speedups among those we have tested. We remark that there is a high synchronization overhead caused by nodes along the principal variation.

5. Dynamic splitting

The main problem with static splitting is load distribution. The master-worker model is based on sequential jobs that are perfomed without attempting to parallelize them and are assigned without estimating the computational costs. Since for parallel $\alpha\beta$ we cannot schedule jobs aiming at minimizing idle times, there is no way of dynamically balancing the load.

5.1 Dynamic job allocation

A first solution to this problem consists of avoiding “atomic” sequential jobs. We defined a variant of the core algorithm in which when there are no more jobs in the agenda the workers that finish their jobs help the busy ones. This means that a busy worker accepts help and splits its search tree, becoming a supervisor. We will apply this idea only to children of the root node. Since we cannot forecast how long the evaluation of a subtree will take, nor which nodes will be assigned to other processors, we classify this a dynamic splitting algorithm.
There are several choice to implement cooperation among workers: we have implemented the following one.

- An explorer E_f completes a search job and finds an empty agenda; however, there are other workers still busy. E_f notifies to be free to a busy worker E_r ;
- E_r decomposes its subtree S, and sends tasks to free explorers;

Processes that cooperate to the search are dynamically partitioned in three sets:

- set F of processes that completed their tasks and become free;
- set R_p of processes that distribute their subtrees;
- set R_s of processes that sequentially evaluate their subtrees.

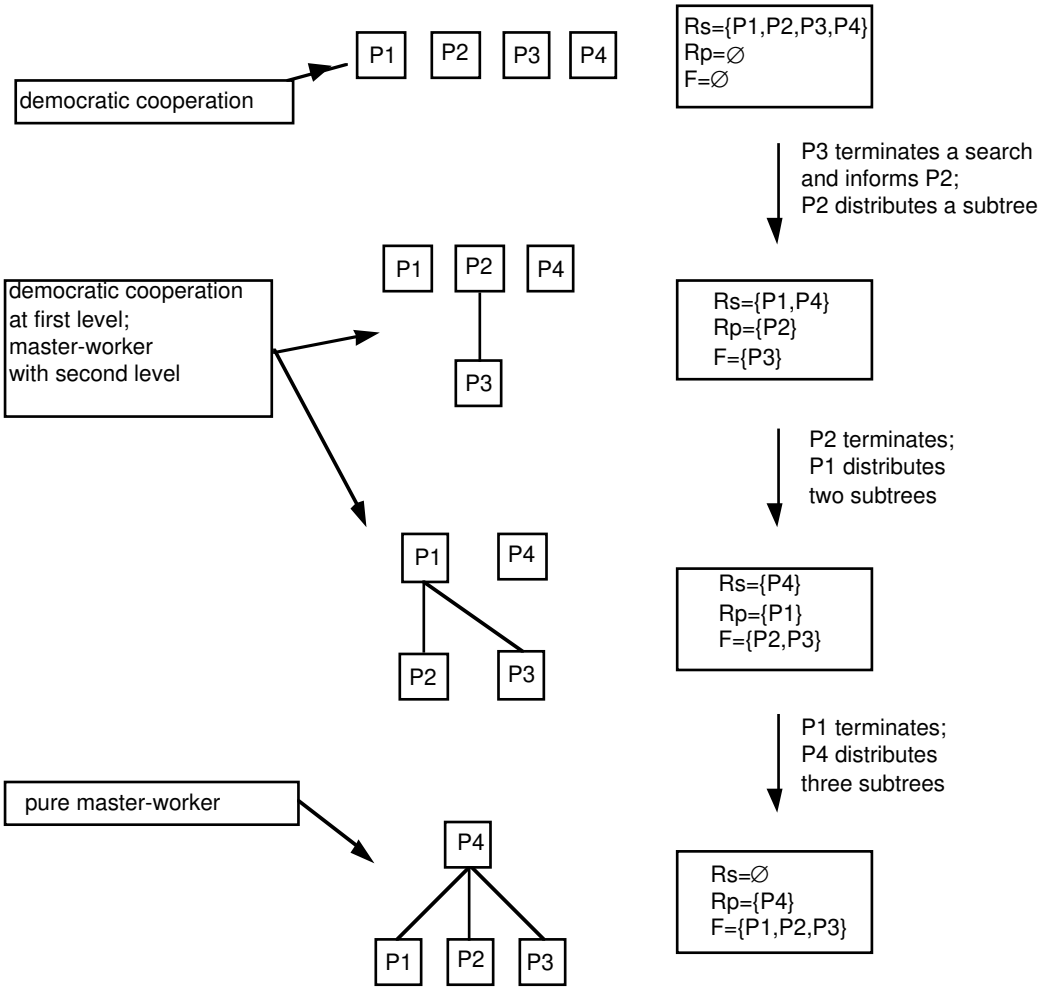

Fig.14 **Dynamic splitting**

In this way we put a limit to the splittings that can happen at run-time, that are expensive: at most they are #F. Thus, the algorithm executed by each worker on its own subtree amounts to classic alphabeta until another worker declares to be available to help, after that the master-worker model is used. The following table summarizes the results obtained with this algorithm.

Procs	T (sec)	N (nodes)	S	E	Fpm	SO	SS (nodes/sec)	Sss
1	6740	4529971	---	---	---	---	672	---
3	3019	5251659	2.23	0.74	0.93	0.16	1740	2.59
5	2216	6322002	3.04	0.61	0.92	0.40	2853	4.24
7	1921	7291629	3.51	0.50	0.91	0.61	3796	5.65
9	1721	8300115	3.92	0.44	0.91	0.83	4823	7.18
11	1600	9439299	4.21	0.38	0.90	1.08	5900	8.78

Fig.15 *Dynamic splitting of top level*

Compare this table with Fig.9. The average production factor is improved (0.71→0.90 with 11 processors): this means that load balancing was better. More interestingly, such parameter is constant: the difference between 3 and 11 processors amounts only to 3%, so that there is little room to improve such values applying dynamic splitting also at lower levels of the game tree. Although search overhead is very high (108% with 11 processors), this waste does not overcome the gain in synchronisation overhead.
We have also tested a dynamic version of PVSplit, enriched with iterative deepening and top level sorting; no global score is implemented (Fig.16).

Procs	T (sec)	N (nodes)	S	E	Fpm	SO	SS (nodes/sec)	Ss
1	6740	4529971	---	---	---	---	672	---
3	2532	4899016	2.66	0.89	0.90	0.08	1935	2.88
5	1805	5453978	3.73	0.75	0.88	0.20	3022	4.50
7	1486	5886412	4.54	0.65	0.87	0.30	3961	5.89
9	1315	6386703	5.13	0.57	0.84	0.41	4857	7.23
11	1208	6789922	5.58	0.51	0.80	0.50	5621	8.36

Fig.16 *Dynamic PVSplit*

Search overhead is low, and efficiency is good: the speedup S with 11 processors was improved of 24% with respect to data shown in Fig.11.

5.2 A discussion of dynamic splitting

The algorithm described in Sect. 5.1 is not really representative of dynamic splitting methods. A more general definition of such a concept should answer to the following questions:

- when a supervisor process P visiting a node N should create a new parallel task?
- who executes the task?
- which data can be use to make a splitting decision?

Consider the game tree in Fig.17; process P is responsible of its evaluation because it was given the root. According to the depth-first rule used in $\alpha\beta$, we have the following sequence for successors: N_1, N_2, N_3.

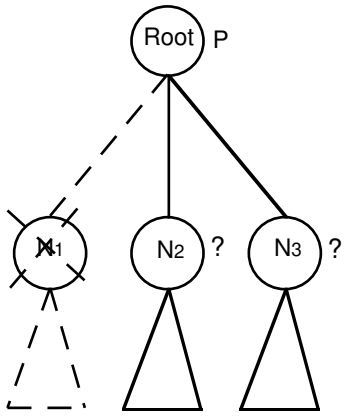

Fig.17 **Problems with dynamic splitting: who has to evaluate N2 and N3?**

Suppose the subtree with root N_1 has been evaluated; if no cuts were produced P has to start the visit of N_2, i.e. it generates the task of visiting N_2. It can happen that P itself is assigned to this task, so

that only when such a visit is completed N3 will be evaluated, unless a cut is produced. If P were assigned all successors of the root node shown in Fig.17, then actually no splitting would have been done in the root. Instead, suppose that N2 is given to a process Q different from P; then, P should not block itself waiting for Q, but should continue the visit and allocate a task for the evaluation of N3. In this way we have a splitting only if this is really necessary; moreover, we improve control on search overhead and load balancing.
An important decision when using of this kind of algorithm is the knowledge necessary for a splitting. There are two kinds of knowledge that are necessary for a process P to take a splitting decision on a node N:

- local knowledge, *i.e*. data owned by the process P itself that is going to decide a splitting. For instance, knowledge on the level of N in the game tree, or on the size of the subtree whose root is N, is local.
- global knowledge, *i.e*. data that have to be asked to the operating environment. For instance, how many processors are idle or how many tasks are still in the agenda are examples of global knowledge.

Suppose that in order to decide if N is a splitting node we only use knowledge local to P, assigning N and its subtree to P itself: the decision will be very rapid. However, load balancing will rapidly deteriorate. Thus, we need to combine both form of knowledge; we achieve this taking a decision in two steps:

i) let P be the supervisor of N; we make a local test to see if P itself can take charge of N;
ii) if not, we use global knowledge to choose a different supervisor. This implies communicating with other processes.

We put all global data in tuples: both the task to be assigned to a worker, corresponding to a splitting, and data to make splitting decisions. A node N becomes a splitting node when search of one of its children is given to another process. A splitting decision needs data that can be either local or non-local. In general non-local data are more expensive to acquire, thus we classify splitting rules built in the Boolean function as local or global. Local rules say when it is not convenient to start a parallel task; if local rules require splitting, then global data are acquired.
This is a very partial list of criteria based on local data:

- subtree size: small subtrees are not distributed. Subtree size should be measured in terms of nodes, but this figure is not known before the visit; however, it is possible to estimate such a quantity knowing the tree depth and the wideness of the initial $\alpha\beta$ window [Bau78]. A measure simpler to obtain is given by the tree depth [Eli90].
- "young-brothers-wait" [FMMV89]: aims at reducing search overhead if the game tree is ordered. The first successor of a node has to be completely evaluated before proceeding in the visit. Thus a node N has to be evaluated by its supervisor if it is the first to be visited. A generalisation of this method consists of avoiding the distribution of the first K (K>1) successors. For ordered game trees the best moves are with high probability "on the left", thus if we give priority to this side we get better $\alpha\beta$ windows for continuing the search in parallel.
- limiting the max. number of children visited in parallel: the idea is to get a good amount of information from incomplete searches. The supervisor only need to know how many nodes were distributed but not yet entirely searched.

The following criteria are based on global knowledge:

- limiting the number of tasks in agenda, aiming at reducing synchronisation overhead. In fact, if tasks were too many with respect to the available processors, the task originators could be strongly slowed. A typical value that can be chosen as maximum is the total number of processors. We put a counter in a single tuple; the counter is incremented by supervisors and decremented by workers.
- splitting only in case of idle processor: we avoid that a task remains too much time in agenda, so that we reduce synchronisation overhead. We need to count how many processors are idle, again with a counter tuple.

Now we are ready to describe a dynamic algorithm parametric wrt the splitting rules.

5.3 A general algorithm for dynamic splitting

We have designed a software architecture that is easy to adapt to different splitting algorithms. Given a splitting criterion, we specialise the core algorithm aiming at maximising the efficiency. We note that we could treat most static splitting algorithms as special cases of trivial dynamic splitting, however we would lose most of the optimisations that were possible because the splitting nodes were well known or in a small number.
The core algorithm for dynamic splitting allows that any node can be a splitting node. Data on arbitrary nodes is much more expensive to transmit and to receive: communication overhead increases, especially on a network of workstations.
Moreover, each task has to be identified independently form the worker that executes it; thus to each node N we associate a signature that will be inherited by all N successors; such an idea was introduced in [Eli90]. When a cut occurs, all parallel search having the same signature will be terminated, because they are now useless.
Each new splitting node is associated to a new signature; since in the core algorithm a parallel search can be started at any level in the tree, nodes of the same subtree obtain different signatures. When a cut is generated for the top-level of a subtree all parallel searches of its branches should terminate, even if the signatures are different. The signature is a distributed data structure whose root is the signature of the game tree root. Suppose that a node N has a signature S inherited from its father; suppose that N is chosen as splitting node: it get s a new signature T.

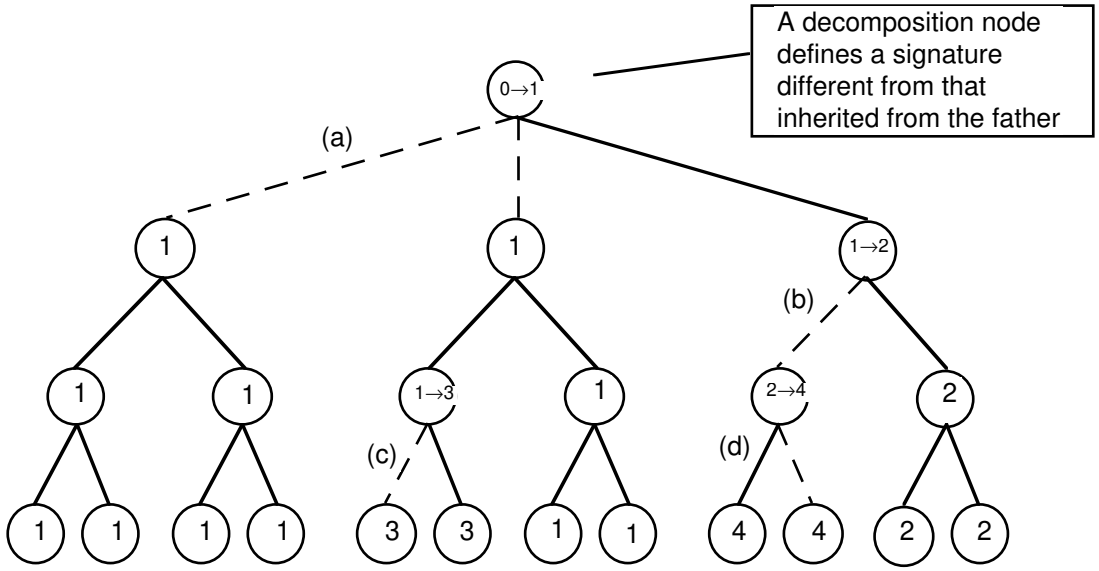

Fig.18 **Assigning a signature**

In Fig.18 it is shown a game tree; dotted lines represents subtrees distributed to other processors; a node label is the local signature. The depicted game tree includes four local splittings. Splitting (a) distributes tasks for searching leftmost subtrees; other splittings distribute a single task only.

5.4 Results

We tested the algorithm for dynamic splitting with several splitting criteria; we found that the most interesting were the following:

- *Policy A*: local condition: young-brothers-wait without distributing small subtrees (depth≤2); global condition: splitting is allowed only if there is an idle worker and the agenda contains (P-1) tasks.
- *Policy B*: no signature tree; splitting is allowed even if there are no idle workers.

The corresponding tests are shown in Figg.19 and 20.

Procs	T (sec)	N (nodes)	S	E	Fpm	SO	SS (nodes/sec)	Sss
1	8301	5347545	---	---	---	---	644	---
3	3797	6206534	2.19	0.73	0.85	0.16	1635	2.54
5	2634	6829036	3.15	0.63	0.82	0.28	2593	4.02
7	2133	7407997	3.89	0.56	0.77	0.39	3473	5.39
9	1878	7810188	4.42	0.49	0.74	0.46	4159	6.46
11	1699	8205839	4.89	0.44	0.70	0.53	4830	7.50

Fig.19 *Young-brothers-wait (Policy A)*

Procs	T (sec)	N (nodes)	S	E	Fpm	SO	SS (nodes/sec)	Sss
1	8301	5347545	---	---	---	---	644	---
3	4629	6476838	1.79	0.60	0.84	0.21	1399	2.17
5	2798	7280335	2.97	0.59	0.88	0.36	2602	4.04
7	2223	8133582	3.73	0.53	0.89	0.52	3659	5.68
9	1972	8541944	4.21	0.47	0.89	0.60	4332	6.72
11	1663	8689459	4.99	0.45	0.89	0.62	5225	8.11

Fig.20 *Young-brothers-wait (Policy B)*

These tables show several differences, although the policies are similar. Policy A gives a smaller search overhead, thanks to the signature tree; moreover, using less processors policy A is better than B. However, the production factor in A deteriorates with more processors, because the global condition it uses reduces the response time for each new task, even if there are "often" idle workers. On the other hand, policy B increments the average number of tasks in the agenda, and it lowers the coupling between the master and the workers. With 11 processors, we have a high production factor (89%). This algorithm results in a finer splitting and higher communication overhead due to task distribution and collection of results. Let us try to analyze such an overhead. Look at the F_{pm} column in Fig.21: such values include also communication overhead. Thus, we should split F_{pm} into F_{pmr} (real productivity) e OC (communication overhead):

Procs	Fpm	Fpmr	OC
3	0.85	0.84	0.01
5	0.88	0.80	0.08
7	0.89	0.80	0.09
9	0.89	0.75	0.14
11	0.89	0.73	0.16

Fig.21 *Policy B: analyizing the average production factor*

Such a table shows that the cost to be paid for communication overhead on a network of workstations is high.

6. Conclusions

We have compared a number of parallel search algorithms in the domain of chess. We used Linda to develop two basic programs, one for static splitting and one for dynamic splitting. Their performances were progressively improved, especially using democratic cooperation with a low number of processors. For all algorithms we have sistematically measured the speedup S. We have seen that the theoretically optimum speedup S=N with N processors is impossible to reach for the following reasons:

- commmunication overhead, given by messages betwen processors;
- search overhead, given by duplication of subtree evaluation;
- synchronisation overhead, given by bad load balancing.

The most annoying problem is the coupling among these overheads: if we tried to reduce one, we got always an increase in another. For instance, the algorithm with dynamic splitting of the game tree top level (Fig.15) gives the best load balancing (Fpm=90% with 11 processors), but also the worst search overhead (SO=108% with 11 processors).
Thus, we searched for a tradeoff among the overheads. Our results say that the best results are given by PVSplit combined either with score sharing (S=5.57 with 11 processors; see Fig.13) or with dynamic distribution (S=5.58 with 11 processors; see Fig.16).
Our results can easily be compared with results obtained by other researchers. For instance, in [MOS86] a speedup of 3.66 with 5 workstations is reported; such a measure was obtained with a depth of 7 plies on the Bratko-Kopec test; with a shallower depth (5 plies) we got a speedup of 3.49 (3.75 with score sharing). In [Sch89] a speedup of 4.78 with 7 processors, depth of 7 plies was obtained for DPVSplit; we got a speedup of 4.54 (depth = 5 plies). These results seem to say that the use of the Linda programming model does not introduce any special overhead for this class of programs.
On the other hand, we got bad comparison results for algorithms based on dynamic distribution of search. However, the problem seems to be the hardware architecture, not the programming environment; in fact, we found the following results in literature:

- [Eli90] reports that the program Oracol, based on dynamic splitting and written in Orca, gets a maximum speedup of 5.5 on a multiprocessor with shared memory and 10 processors.
- [FMM92] presents a dynamic algorithm based on "young-brothers-wait". On a transputer with 256 processors, the authors report a surprising speedup of 126 (depth 8-plies).

These results suggest that dynamic splitting algorithms can offer good performances, but only on shared memory or on massive parallel systems where communication and fine-grain parallelism are cheap. It would be interesting to test out general dynamic splitting algorithm on these architectures. Also, it would be interesting to verify in massively parallel machines the asyntotic convergence postulated in [MOS86] for static splitting methods .

Acknowledgements. This paper has been partially supported by ESPRIT BRA Project BROADCAST and by Italian CNR.

Bibliography

[ABD82] S.Akl, D.Barnard, R.Doran, "Design, Analysis, and Implementation of a Parallel Tree Search Algorithm", *IEEE Transactions on Pattern Analysis and Machine Intelligence,* 4(2), (1982), 192-203.

[BalvRe86] H.Bal, R.vanRenesse, "A summary of parallel alpha-beta search results", *ICCA Journal*, (Sep.1986), 146-149.

[Bau78] G.Baudet, "The design and analysis of algorithms for asynchronous multiprocessors", Ph.D. Thesis, CS Dept., Carnegie-Mellon Univ., (Apr.1978).

[BraKop82] I.Bratko, D.Kopec, "The Bratko-Kopec experiment: a comparison of human and computer performance in chess", *Advances in Computer Chess 3*, M Clarke (Ed.), (1982), 57-72, Pergamon Press.

[CarGel90] N.Carriero, D.Gelernter, *How to write parallel programs. A first course*, MIT Press, 1990.

[Ebe86] C.Ebeling, *All the Right Moves: A VLSI Architecture for Chess*, MIT Press, 1986.

[Eli90] R.Elias, "Oracol, A Chess Problem Solver in Orca", Dept. of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, (1990).

[Elo86] A.Elo, The Rating of Chessplayers: Past and Present, 2nd Ed., Arco, 1986.

[FMM91] R.Feldmann, P.Mysliwietz, B.Monien, "A Fully Distributed Chess Program", *Advances in computer chess* 6, D. Beal (Ed.), (1991), 1-27.

[FMM92] R.Feldmann, P.Mysliwietz, B.Monien, "Experiments with a Fully-Distributed Chess Program", *Heuristic Programming in AI*, J.VanDenHerik and V.Allis (Eds.), Hellis Horwood, (1992), 72-87.

[FelOtt88] E.Felten, S.Otto, "A Highly Parallel Chess Program", *Proc. of the International Conference on Fifth Generation Computer Systems*, (1988), 1001-1009.

[FisFin80] J.Fishburn, R.Finkel, "Parallelism in Alpha-Beta Search on Arachne", Technical Report 394, CS. Dept., Univ. of Wisconsin, Madison, (Jul.1980).

[FinFis82] J.Fishburn, R.Finkel, "Parallelism in alpha-beta search", *Artificial Intelligence*, Vol. 19, (1982), 89-106.

[FMMV89] R.Feldmann, P.Mysliwietz, B.Monien, O.Vornberger, "Distributed Game-Tree Search", *ICCA Journal*, Vol.12, No.2, (1989), 65-73.

[Gil72] J.Gillogly, "The Technology Chess Program", *Artificial Intelligence*, Vol. 3, (1972), 145-163.

[Hsu90] F.Hsu, "Large scale parallelization of alpha-beta search: an algorithmic and architectural study with computer chess", Ph.D. thesis, Carnegie Mellon University, Pittsburgh, (Feb. 1990).

[SCA90] "C-Linda Reference Manual", Scientific Computing Associates Inc., New Haven, Connecticut, (1990).

[MBS91] T.Marsland, T.Breitkreutz, S.Sutphen, "A network multiprocessor for experiments in parallelism", *Concurrency: Practice and Experience*, 3(3), (Jun.1991), 203-219.

[MarCam82] T.Marsland, M.Campbell, "Parallel Search of Strongly Ordered Game Trees", *ACM Computing Surveys*, Vol.14, No.4, (Dec.1982), 533-551.

[MOS86] T.Marsland, M.Olafsson,J.Schaeffer, "Multiprocessor tree-search experiments", *Adavances in Computer Chess 4*, Pergamon Press, (1986), 37-51.

[Sch89] J.Schaeffer, "Distributed Game-Tree Searching", *Journal of Parallel and Distributed Computing*, 6, (1989), 90-114.

[Sti91] L.Stiller, "Group Graphs and Computational Symmetry on Massively Parallel Architecture", *The Journal of Supercomputing*, 5, (1991), 99-117.

[Tho82] K.Thompson, "Computer Chess Strength", *Advances in Computer Chess 3*, M.R.B. Clarke (Ed.), Pergamon Press, (1982), 55-56.