Efficient Techniques for Distributed Implementation of Search-based AI Systems*

Gopal Gupta and Enrico Pontelli
Laboratory for Logic, Databases and Advanced Programming
Department of Computer Science
New Mexico State University
Las Cruces, NM 88003, USA
http://www.cs.nmsu.edu/lldap

Abstract

We study the problem of exploiting parallelism from search-based AI systems on distributed machines. We propose stack-splitting, a technique for implementing or-parallelism, which when coupled with appropriate scheduling strategies leads to: (i) reduced communication during distributed execution; and, (ii) distribution of larger grain-sized work to processors. The modified technique can also be implemented on shared memory machines and should be quite competitive with existing methods. Indeed, an implementation has been carried out on shared memory machines, and the results are reported here.

1. Introduction

Artificial Intelligence (AI) is an active field of research, that has found applications in diverse areas. The field of AI is very broad and one can find several types of AI systems: those based on neural networks, those based on tree/graph search, image recognition systems, etc. In this paper we are primarily interested in AI systems that rely on traversing a large search-space, looking for a solution that satisfies certain criteria [17, 16, 18]. We refer to such systems as search-based AI systems. Game playing programs, expert systems, constraint solving applications, and discourse analysis systems are example of such search-based AI systems. Such search-based AI systems can take a lot of time to find a solution, as the search space can be enormous. Given the compute-intensive nature of search-based AI systems, parallel execution is an obvious technique that comes to mind to speed-up the search. In fact, considerable research has been done [18, 16, 19, 17, 6] on exploiting parallelism from search-based AI system. Two approaches have been generally been followed: (i) techniques have been developed and implemented for extracting parallelism from specific AI systems (e.g., [13, 14]), (ii) techniques have been developed and implemented for extracting parallelism from language constructs in programming languages that are typically used for coding AI applications (e.g., Prolog, or Lisp) [6, 11]. In both cases, it is the operation of searching the solution-space that is parallelized. It should also be mentioned that most work on exploiting parallelism falls under (ii). Nearly not as much work has been done on (i), for the obvious reason that (ii) represents a more general approach. Within (ii) considerable work has been done on parallelizing Prolog. In the rest of the paper, we will present our techniques and results in the context of parallel Prolog, though they can equally well be applied to specific AI systems that incorporate searching, as well as other languages that incorporate search mechanisms to facilitate programming of search-based AI applications.

Implementing search (parallel or sequential) requires that we have a representation of the search space in the computer's memory. This representation is usually a tree—called the *search tree*. Each node of this tree represents a branch point from where multiple branches emanate. These branches may lead to further nodes, which may yet split into other branches, and so on. The nodes, or branch points, are termed *choice points*, and the branches are termed *alternatives*, if we were to use Prolog's terminology [6].

The obvious way to search this tree in parallel is to have multiple processors explore the different branches of the search-tree in parallel [18, 16, 17, 19]. Given a search tree, the model of computation that is typically employed is as follows. Multiple processors traverse the search tree looking for unexplored branches. If an unexplored branch, i.e., an unexplored alternative in a choice point, is found, then the processor will select it and begin its execution. The processor will stop either if it fails, i.e., it determines that the solution cannot lie on that branch, or if it finds a solution.

*Authors are partially supported by NSF grants CCR 96-25358, INT 95-15256, CDA 97-29848, HRD 96-28450, EIA 98-10732, and CCR-9900320 and a grant from the Fullbright US-Spain Program.

In case of failure, or if the solution found is not acceptable to the user, the processor will *backtrack*, i.e., move back up in the tree, looking for other choice points with untried alternatives to explore. This process of traversing the tree in parallel is complicated by the need of guaranteeing proper synchronization between processors, e.g., to guarantee that no two processors selects the same alternative for execution.

This form of search-based parallelism is commonly termed *or-parallelism*. Efficient implementation of or-parallelism has been extensively investigated in the context of AI systems [18, 19, 17] as well as for the Prolog language [10]. In sequential implementations of search-based AI systems or Prolog, typically one branch of the tree resides on the stack of the processor at any given time. This simplifies implementation quite significantly—e.g., backtracking is reduced to a simple *pop* operation on the main stack. However, in case of parallel systems, multiple branches of the tree co-exist at the same time, making the parallel implementation complex. Efficient management of these co-existing branches is quite a difficult problem, and is referred to as the *environment management problem* [10].

Most parallel implementation of parallel AI systems and parallel Prolog systems have focused on shared-memory machines. Very few attempts have been made to realize such implementations on scalable distributed memory machines. It should be noted that the most efficient or-parallel execution models devised for shared memory machines do not scale up to distributed memory machines, highlighting the difficulty of realizing or-parallel systems on distributed machines.

In this paper we present a method for implementing or-parallelism on distributed memory machines. This method, called *stack-splitting*, reuses efficient implementation mechanisms devised for or-parallel systems on shared-memory multiprocessor to obtain scalable implementation of or-parallelism on distributed memory multiprocessors. This allows us to support or-parallelism on distributed memory architectures with reduced communication and without giving up the use of scheduling mechanisms that have been found to work well for or-parallelism. Stack-splitting has the potential to: (i) improve locality of computation, reduce communication between parallel threads, and increase memory access efficiency (e.g., improve the caching behavior). (ii) allow the use of better scheduling strategies (specifically scheduling on bottom-most choice point [1, 4]) to be realized even in distributed memory implementations of or-parallelism.

In this paper we also present results from implementing stack-splitting on top of the Muse method [1], one of the most efficient method for implementing or-parallelism on shared-memory machines.

2. Implementing Or-parallelism

A major problem in implementing or-parallelism is that multiple branches of the search tree are active simultaneously, each of which may produce a solution or may fail. Each of these branches may potentially bind a variable created earlier during the execution. In normal sequential execution, where only one branch of the search tree is active at any given time, the binding for the variable created by that branch is stored in the memory location allocated for that variable. During backtracking, this binding is removed—during the *untrailing* phase—so as to free the memory location for use by the next branch.

However, during or-parallel execution, this memory location will have to be turned into a *set of locations* shared between processors, or some other means would have to be devised to store the multiple bindings that may exist simultaneously. In addition, we should be able to efficiently distinguish the binding that is applicable to each branch, when it needs to be accessed later in that branch. This problem of maintaining multiple bindings efficiently is called the *multiple environments representation* problem. An extensive discussion can be found in [10] and a complexity-theoretic analysis of the problem is presented in [21]. Numerous solutions have been devised to solve the multiple environments representation problem, and a survey of these techniques can be found in [10].

Stack-copying [1] has been one of the most successful approaches for solving the multiple environments representation problem. It has been incorporated in the Muse or-parallel system [1]. In this approach, processors working in or-parallel maintain a *separate* but *identical* address space, i.e., they allocate their data areas starting at the same logical addresses. Whenever a processor $\mathcal{A}$ working in or-parallel becomes idle, it will start looking for unexplored alternatives generated by some other processor $\mathcal{B}$. Once a choice point p with unexplored alternatives is detected in the computation tree $\mathcal{T}_\mathcal{B}$ generated by $\mathcal{B}$, then $\mathcal{A}$ will create a local copy of $\mathcal{T}_\mathcal{B}$ and restart the computation by backtracking over p and executing one of its unexplored alternatives. The fact that all the the processors working on or-parallel maintain an identical logical address space reduces the creation of a local copy of $\mathcal{T}_\mathcal{B}$ to a simple block memory copying operation (Figure 1).

However, the stack-copying operation is slightly more involved than simply copying data structures, as the choice points have to be copied to an area accessible to all processors. This is important because the set of untried alternatives is now shared between the two processors, and if this set is not accessed in a mutually exclusive way then two processors may execute the same alternative. Thus, after copying, the choice point will be transferred to a shared area. Using the terminology used by Muse, we will refer to

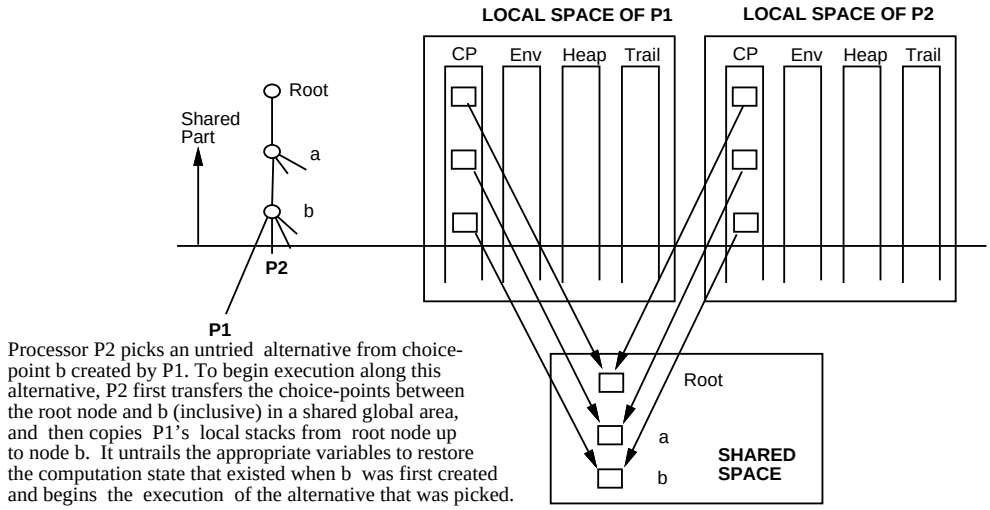

Figure 1. Stack-copying based Or-parallelism

a choice point transferred to the shared memory area as a *shared frame*. Both the processor that copies and the processor being copied from, will replace their choice points with a pointer to the appropriate shared frame. Due to involvement of shared frames, this whole operation of obtaining work from another processor is termed *sharing* of or-parallel work. In order to reduce the number of sharing operations performed (since each sharing operation may involve a considerable amount of overhead), unexplored alternatives are always picked from the *bottom-most* choice point in the tree; during the sharing operation all the choice points between the bottom-most choice point and the top-most choice point are shared between the two processors. This means that in each sharing operation we try to maximize the amount of work shared between the two processors. Furthermore, in order to reduce the amount of information transferred during the sharing operation, copying is done *incrementally*, i.e., only the *difference* between $\mathcal{T}_{\mathcal{A}}$ and $\mathcal{T}_{\mathcal{B}}$ is actually copied.

3. The Stack-Splitting Model

A major reason for the success of the Muse method is that it performs *scheduling on bottom-most choice point*, as mentioned earlier. That is, an idle processor picks work (an untried alternative) from the bottom-most choice point of an or-branch. The stack segments upwards of this choice point are copied before the exploration of this alternative is begun. The copied stack segments may contain other choice points with untried alternatives. These alternatives will be tried via standard backtracking on the copied segments (of course, they may be picked by other processors looking for work as well). Thus, a significant amount of work potentially becomes available to the copying processor every time a copying operation is performed.

The shared frames in the shared memory space have to be accessed in a mutually exclusive manner, to make sure that the same alternative is not tried by two processors that have copies of the same stack-segment. This solution for building an or-parallel system based on the shared frames works fine on a shared memory multiprocessor, however, on a distributed memory machine it becomes a source of significant overhead, as the operation of accessing the shared area becomes a bottleneck. This is because sharing of information in a distributed memory machine leads to frequent exchange of messages and hence considerable overhead. Centralized data structures, such as the shared frames, are, not unexpectedly, expensive to realize in a distributed setting. Nevertheless, stack copying has been considered by most researchers as the best environment representation methodology to support or-parallelism in a distributed memory setting [5, 2]. This is because while the choice points are shared, at least all other data-structures, such as the environment, the trail, and the heap, are not. However, the fact that the choice points are shared is a major drawback for a distributed implementation of stack-copying. So the question we wish to consider is: can we avoid this sharing of choice points while doing bottom-most scheduling?

3.1. Copying with Stack Splitting

In the stack-copying technique the primary reason why a choice point has to be shared is because we want to make sure that the selection of its untried alternatives by various active processors is serialized, so that no two processors select the same alternative. The shared frame is locked while

the alternative is picked, to guarantee this property. However, there are other simple ways of ensuring that no alternative is simultaneously selected by multiple processors: we can *split* the untried alternatives of a choice point between the two copies of the choice point stack. We call this operation *Choice Point Stack Splitting* or simply stack-splitting. This will ensure that no two processors pick the same alternative—since no alternative is visible to more than one processor at a time.

We can envision different schemes for splitting the set of alternatives between the two choice points—e.g., each choice point receives half of the alternatives, or the partitioning can be guided by additional information regarding the unexplored computation, such as granularity and likelihood of failure. In addition, the need for a shared frame, as a critical section to protect the alternatives from multiple executions, has disappeared, as each stack copy has a choice point, though their contents differ in terms of which unexplored alternatives they contain. All the choice points can be evenly split in this way during the copying operation. The choice point stack-splitting operation is illustrated in figure 2.

3.2. Advantages of Stack-splitting

The major advantage of stack-splitting is that scheduling on bottom-most can still be used without incurring huge communication overheads. Essentially, after splitting, the different or-parallel threads become fairly independent of each other, and hence communication is minimized during execution. In particular, backtracking on a node that has been copied from a different processor does not require anymore the use of mutual exclusion. This makes the stack-splitting technique highly suitable for distributed memory machines. The possibility of parameterizing the splitting of the alternatives based on additional semantic information (granularity, non-failure, user annotations) can further reduce the likelihood of additional communications due to scheduling.

3.3. Overheads of Stack-splitting

The shared frames in the regular stack-copying technique is also a place where global information related to scheduling and work-load is kept. The shared frames provide a globally accessible description of the or-tree, and each shared frame keeps information that allows one to determine which processor is working in which part of the tree. This last piece of information is of particular importance to support the kind of scheduling typically used in stack-copying systems—work is taken from the processor that is "closer" in the computation tree, thus reducing the amount of information to be copied—since the "distance" between the processors, and the hence the difference between their stacks, is minimized. The shared nature of the frames ensures accessibility to this information to all processors, all of whom see a consistent picture. However, because the shared frame no longer exists under the stack-splitting schema, scheduling and work-load information will have to be maintained in some other way. It could be kept in a global shared area similar to the case of shared memory machines (e.g., by building a representation of the or-tree), or distributed over multiple processors and accessed by message passing in case of non-shared memory machines. The management of scheduling in a distributed memory system will require communication between processors anyway; the use of stack-splitting allows scheduling on bottom-most and is expected to reduce the amount of scheduling-related communication needed. In particular, access to non-local information is needed only when a processor runs out of local work, and not at each backtracking step (as in the case of standard stack copying).

Thus, stack-splitting does not completely remove the need of a shared description of the computation tree. Nevertheless, the use of stack-splitting can mitigate the impact of accessing logically shared resources—e.g., stack-splitting allows scheduling on bottom-most which, in general, reduces the number of calls to the scheduler [1].

3.4. The Cost of Stack-splitting

Let us next consider the cost of the stack-splitting operation. The stack-copying operation in the stack-splitting technique is a little involved, though only slightly more than in regular (*a lá* Muse) stack-copying. In regular stack-copying, the original choice point stack is traversed and the choice points transferred to the shared area. This operation involves only those choice points that have never been shared before—if a choice point is already shared, then its copy already resides in the global shared memory-area. The update of the actual entries in the choice point stacks of the the processors takes place only after the appropriate choice points have been copied to the global shared area.

In the stack splitting technique, after the copying is done, we need to traverse both the stacks, splitting the untried alternatives in the choice points of the two stacks. In the case of shared memory implementations, this operation is expected to be considerably cheaper than transferring the choice point to the shared area. The actual splitting can be represented by a simple pair of indices that refer to the list of alternatives (which is static and shared by all the processors). In the case of distributed memory implementations, the situation is similar: since each processor maintains a local copy of the code, the splitting can be performed by communicating to the copying processor which alternatives it can execute for each choice point (e.g., described as a pair

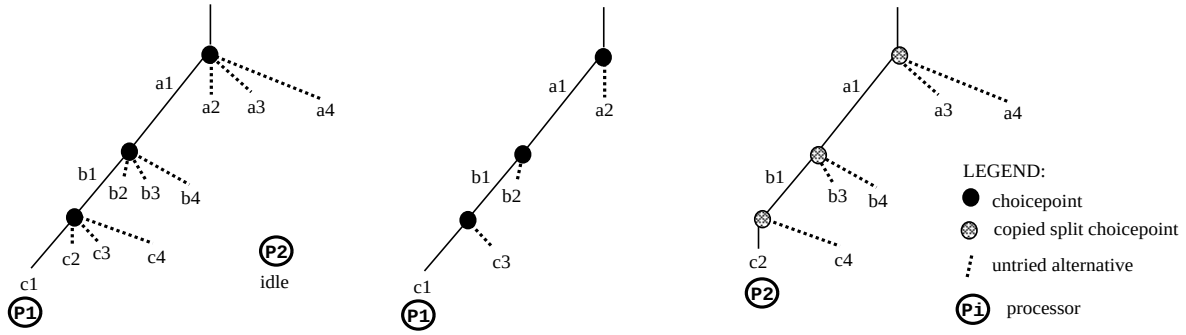

Fig (i): Processor P1 is busy and P2 idle

Fig (ii): P1's Tree after Stack Splitting

Fig (iii): P2's Tree after Stack Splitting

Figure 2. Stack-splitting based or-parallelism

of pointers to the static list of alternatives).

Thus, in both cases we expect the sharing operation to have comparable complexity; a slight delay may occur in the shared memory case, due to the need of performing a traversal of the choice point stack in both the processors. On the other hand, in stack-splitting the two traversals (one in the copying processor and one in the processor from where we are copying) can be overlapped—in the original stack copying scheme the copying processor is instead suspended until the other processor has completed the sharing operation. However, if the stack being copied is itself a copy of some other stack, then unlike regular stack-copying, we still need to traverse both the source and the target stacks and split the choice points. In such cases, the cost of the sharing operation will be slightly higher than the cost of copying in regular stack-copying.

Once a processor picks work from another processor, it will look for work again only after it finishes the exploration of this alternatives, as well as all the alternatives it acquired via stack-splitting. Incremental copying and other optimizations developed for Muse still apply to stack-splitting, though some extra work is needed. Each processor has knowledge of the parts of its stack which are shared (this information is available locally to the processor). These shared parts, if possible, should not be immediately deallocated on backtracking; otherwise, when work is picked from other processors, these shared parts will have to be copied again.

3.5. Optimizing Stack-splitting Cost

The cost incurred in splitting the untried alternatives between the copied stack and the stack from which the copy is made, can be eliminated by amortizing it over the operation of picking untried alternatives during backtracking, as shown next.

In the modified approach, no traversal and modification of the choice points is done during the copying operation. The untried alternatives are organized as a binary tree (see Figure 3). Note that the binary alternatives can be efficiently maintained in an array, using standard techniques found in any data-structures textbook. In addition, each choice point maintains the "copying distance" from the very first original choice point as a bit string. This number is initially 0 when the computation begins. When stack-splitting takes place and a choice point whose bit string is n is copied from, then the new choice point's bit string is $n1$ (1 tagged to bit string n), while the old choice point's bit string is changed to $n0$ (0 tagged to bit string n). When a processor backtracks to a choice point, it will use its bit string to navigate in the tree of untried alternatives, and find the alternatives that it is responsible for. For example, if the bit-string of a processor is 10, then it means that all the alternatives in the left subtree of the right subtree of the binary tree are to be executed by that processor.

However, it is not very clear which of the two strategies—incurring cost of splitting at copying time *vs* amortizing the cost over the operation of picking untried alternatives—would be more efficient. In case of amortization, the cost of picking an alternative from a choice point is, of course, now slightly higher, as the binary tree of choice points needs to be traversed to find the right alternative.

3.6. Applicability and Effectiveness

Stack splitting essentially approximates static work distribution, as the untried alternatives are split at the time of picking work. If the choice points that are split are balanced, then we can expect good performance. Thus, we should expect to see good performance when the choice points generated by the computation that are parallelized contain a large number of alternatives. This is the case for applications which fetch data from databases and for most generate & test type of applications.

For choice points with a small number of alternatives, the stack-splitting scheme is more susceptible to problems cre-

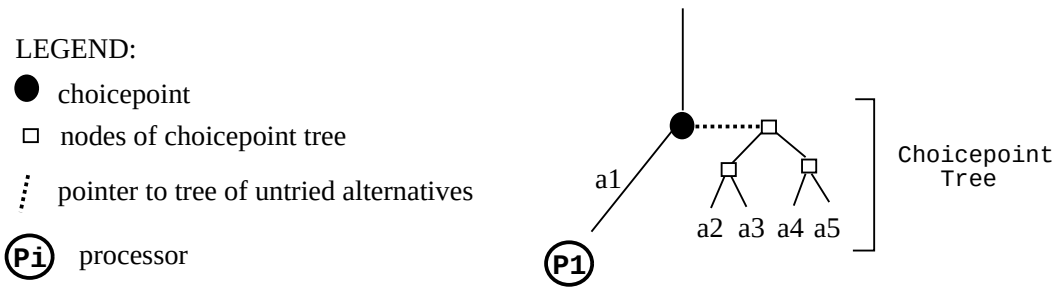

Figure 3. Amortizing Splitting Overhead

ated by the static work distribution strategy that implicitly results from it: for example, in cases where or-parallelism is extracted from choice points with only two alternatives. Such choice points arise quite frequently, since many programs generate or-parallelism from predicates like member and select:

```
member(X,[X | _]).
member(X,[_ | Y]) :-
        member(X,Y).

select(X,[X | Y],Y).
select(X,[Y | Z],[Y | R]) :-
       select(X,Z,R).
```

Both these predicates generate choice points with only two alternatives each—thus, at the time of sharing, a single alternative will be available in each choice point. The different alternatives are spread across different choice points. Stack splitting would assign all the alternatives to the copying processor, thus leaving the original processor without local work. However, the problems raised by such situations can be solved using a number of techniques discussed in [9]. Most significant of these is the technique of *vertical* splitting of the choice points. In vertical splitting each processor is given all the alternatives of alternate choice points. Thus, in this case, the alternatives are not split, rather the list of choice points available is split between the two processors [9].

3.7. Performance Evaluation

The stack-splitting technique has been implemented by modifying the Muse or-parallel system, which is itself realized on top of the SICStus Prolog (SICStus 2.1) system from the Swedish Institute of Computer Science. The first prototype of stack-splitting has been developed on shared-memory architectures. The goal of this prototype is to perform a preliminary feasibility study of the ideas discussed in this paper. Porting on distributed-memory architectures is currently in progress.

The timing results in seconds along with speedups obtained are shown in Table 1 and Table 2. All the benchmarks for which results are reported involve search. The benchmarks used are classical benchmarks used for evaluating the parallel behavior of or-parallel systems—and they have been mostly taken from the benchmarks pool of or-parallel systems Muse and Aurora.

The results reported in this paper have been obtained on: (i) A Pentium-Pro 200Mhz 4-node shared memory workstation, running Solaris 2.7 (software compiled using gcc); (ii) An 8-node Sequent shared memory system, running Dynix. The Sequent hardware is relatively old and slow with respect to current industry standards. Nevertheless it gives a good feeling for the parallel behavior of the system and it represents the largest parallel shared memory system currently available to us. Experiments on Pentium-based and Sparc-based parallel systems have provided comparable speedups.

The results presented in Table 1 illustrates the execution times and speedups observed on the Sequent system. In particular the table presents for each benchmark

- the execution times (in seconds)—the figures reported are the average execution times over a sequence of 10 runs;
- the relative speedups obtained for different number of processors;

Additionally, the table presents the results obtained using the Muse or-parallel system and the or-parallel system based on Stack Splitting. Table 2 presents the results (time in milliseconds) obtained on the Pentium-pro hardware. It can be observed that the speedups on the two systems are very similar.

All benchmarks have been executed by requiring the system to exploit parallelism only from selected promising predicates, and by declaring all other predicates sequential (i.e., non-parallelizable). We believe this situation better reflects the kind of behavior needed to guarantee adequate performance in a distributed execution (which remains the

Benchmark	# Processors							
	1		3		5		7	
	Muse	SS	Muse	SS	Muse	SS	Muse	SS
Tina	21.4	22.5	25.1 (0.85)	10.1 (2.22)	25.1 (0.85)	8.3 (2.71)	25.0 (0.85)	7.6 (2.97)
Large	131.7	144.9	118.0 (1.12)	54.3 (2.67)	120.1 (1.1)	38.8 (3.73)	118.4 (1.11)	32.3 (4.49)
Queens1	56.8	63.6	49.8 (1.14)	39.5 (1.61)	42.9 (1.32)	36.1 (1.76)	42.1 (1.35)	34.4(1.85)
FQueens	51.3	59.0	18.8 (2.72)	24.5 (2.41)	10.8 (4.77)	13.1 (4.51)	7.5 (6.88)	9.9 (5.95)
Salt	98.8	104.7	38.7 (2.55)	36.2 (2.89)	25.1 (3.93)	23.3 (4.5)	18.6 (5.32)	18.2 (5.76)
Solitaire	22.9	24.4	8.4 (2.71)	8.7 (2.8)	5.3 (4.29)	5.6 (4.37)	4.6 (4.95)	4.1 (5.91)
Houses	37.1	41.2	13.7 (2.71)	14.5 (2.84)	8.9 (4.18)	9.3 (4.45)	6.7 (5.53)	6.9 (5.92)
constraint	67.0	69.7	25.0 (2.68)	25.7 (2.71)	15.2 (4.42)	15.5 (4.49)	10.6 (6.35)	10.8 (6.44)

Table 1. Execution Times on Sequent: Muse vs. Stack-Splitting

ultimate goal of the stack-splitting model). All the benchmarks have been executed using all-solutions queries.

As can be seen in Table 1, stack-splitting leads in general to better speed-ups. The execution time of the stack-splitting system is occasionally slightly worse than the execution times of Muse—on average the sequential stack-splitting system is 5% to 12% slower than sequential Muse. This is due to: (i) the temporary removal of some sequential optimizations from the stack-splitting system, to facilitate the development of the initial prototype; and, (ii) the presence of one additional comparison during the backtracking phase—needed to distinguish between choice points that have been split and those that have not. The first problem will be solved in the next version of the prototype. The second problem is inherent in the current representation of the alternatives in the choice points used by the SICStus system (on which the stack-splitting system has been developed). All choice points associated to the same predicate share the same list of alternatives. This complicates the implementation of stack-splitting, as the list of alternatives cannot be directly manipulated (as this may potentially affect other choice points as well). The simple alternative of duplicating the list of alternatives proved too inefficient. At present we have introduced two pointers in the choice point to maintain the segment of alternatives list of interest—and this leads to the need of discriminating between split and non-split choice points during backtracking. The adoption of a different, alternative representation in the engine could solve this problem—but requires drastic changes to the basic sequential engine and to the compiler.

In the case of *Tina* benchmark, the Muse system suffers a slowdown irrespective of the number of processors employed. This behavior is rather unusual (and at odds with the speed-ups reported for *Tina* in the literature for Muse). We conjecture it originates from the fact that we have explicitly identified all choice points as parallel or sequential. Stack-splitting is instead capable of extracting parallelism from this benchmark reaching a maximum speedup of about 3. Another interesting benchmark is *Large* which generates a small and balanced number of relatively deep branches—an ideal situation for the splitting approach. Muse obtains marginal speed-up, while stack-splitting produces considerably better speedups. Better parallel behavior is obtained for almost all benchmarks, except *FQueens*. This benchmark generates a single choice point with a very large number of alternatives, and each alternative is small and leads quickly to success or failure. In this case stack-splitting pays the price of a slightly more expensive sharing phase. Nevertheless this is clearly not the kind of situations in which distributed execution is desired.

An implementation of the stack-splitting method is in progress on a distributed network of shared-memory multiprocessors (on a Beowulf Myrinet-based system with Pentium-2 nodes). From the analysis and discussion presented above, it is apparent that stack-splitting should perform well on distributed memory machines, primarily because of better locality and because it leads to reduced communication. A low-level performance study of our shared memory implementation of stack-splitting implementation is in progress using the SimICS Sparc multiprocessor simulator. The low level performance study of caching behavior, locality of access, etc., will give us an indication of what kind of performance to expect from a distributed implementation of stack-splitting.

Benchmark	# Processors		
	1	2	3
Tina	1215	719 (1.69)	535 (2.27)
Large	8093	4422 (1.83)	3065 (2.64)
Queens1	3520	2466 (1.43)	2207 (1.59)
Salt	6117	3274 (1.87)	2155 (2.84)
Solitaire	1364	704 (1.94)	488 (2.79)
Houses	2425	1263 (1.92)	859 (2.82)
constraint	3030	1569 (1.93)	1102 (2.75)

Table 2. Execution Times on Solaris X86

4. Conclusion and Related Work

In this paper, we presented a technique called stack-splitting for implementing or-parallelism and discussed its advantages and disadvantages. Stack-splitting is an improvement of stack-copying. Its main advantage, compared to other well-known techniques for implementing or-parallelism, is that it allows coarse-grain work to be picked up by idle processors and be executed efficiently without incurring excessive communication overhead.

Distributed implementation of AI systems has been a reasonably active area of research. There are several projects in which a specific AI system has been taken and parallelized on distributed memory multiprocessors [15, 22, 8, 12, 7, 17, 16, 18, 19]. Distributed implementation of Prolog have also been attempted [2, 5]. However, none of these systems are very effective in producing speedups over a wide range of benchmarks. Distributed implementations of Prolog have been attempted on Transputer systems (The Opera System [23] and the system of Benjumea and Troya [3]). Of these, Benjumea and Troya's system has produced quite good results. However, both the OPERA system and the Benjumea and Troya's system have been developed on now-obsolete Transputer hardware, and, additionally, both rely on a stack-copying mechanism which will produce poor performance in programs where the task-granularity is small. We hope that our distributed implementation of Prolog based on stack-splitting will be superior to these aforementioned distributed implementations. A distributed parallel implementation of Prolog based on stack-copying, with ALS Prolog system (www.als.com) as the underlying engine, is planned in the near future.

References

[1] K.A.M. Ali and R. Karlsson. The MUSE Approach to Or-parallel Prolog. In *Int'l J. of Parallel Prog.*, 19(2):129–162, 1990.

[2] L. Araujo and J.J. Ruz. A Parallel Prolog System for Distributed Memory. In *Journal of Logic Programming*, 33(1):49-79, 1997.

[3] V. Benjumea and J. M. Troya. An OR Parallel Prolog Model for Distributed Memory Systems. In *Procs. of PLILP* Springer Verlag, LNCS 714, pp. 291-301, 1993.

[4] A. Beaumont and D.H.D. Warren. Scheduling Parallel Work in Or-parallel Prolog Systems. In *Proc. International Conference on Logic Programming*. pp. 135-150. MIT Press. 1993.

[5] L. F. Castro, C. Geyer et al. DAOS: Distributed And-Or in Scalable Systems. Technical Report. Department of Computer Science, Federal University of Rio Grande del Sul, Brazil, 1998.

[6] J. C. de Kergommeaux, P. Codognet. Parallel Logic Programming Systems: A Survey. In *Computing Surveys*, 26(3): 295-336, 1994.

[7] M. Dixon and J. de Kleer. Massively parallel assumption-based truth maintenance. LNAI, Springer-Verlag. pp. 131–142. 1989.

[8] J. Gu. Parallel Algorithms and Architectures for Very Fast AI Search University of Utah, 1989

[9] G. Gupta and E. Pontelli. Stack-splitting: A Simple Technique for Implementing Or-parallelism and And-parallelism on Distributed Machines. NMSU Tech. Rep. May 1999.

[10] G. Gupta and B. Jayaraman. Analysis of Or-parallel Execution Models. *ACM Transactions On Programming Languages and Systems (ACM TOPLAS)*. Vol 15. No. 4. September 1993. pp. 659-680.

[11] D.A. Kranz, et al. Mul-T: A High-Performance Parallel Lisp. In *ACM Programming Lang. Design and Impl.*, pp. 81-90, 1989.

[12] A. Jindal, R. Overbeek, W. C. Kabat. Exploitation of parallel processing for implementing high-performance deduction systems. *Journal of Automated Reasoning*, 8(1), pp. 23–38, 1992.

[13] J. Hendler et al. Massively Parallel Support for a Case-based Planning System. In Proceedings of the Ninth IEEE Conference on AI Applications, Orlando, Florida, March 1993.

[14] H. Kitano, J. A. Hendler (eds.). *Massive Parallel Artificial Intelligence*, AAAI Press/MIT Press, Menlo Park, 1994.

[15] J.S. Kowalik. *Parallel Computation and Computers for Artificial Intelligence*. Kluwer Academic Publishers. 1987.

[16] V. Kumar, P. S. Gopalakrishnan, L. N. Kanal (eds.), *Parallel Algorithms for Machine Intelligence and Vision*. Springer-Verlag, 1990.

[17] V. Kumar and L. N. Kanal. Parallel Branch-and-Bound Formulation for AND/OR Tree Search. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*. Volume 6, pp. 768–788. 1984.

[18] V. Kumar and J. N. Rao, Parallel Depth-First Search on Multiprocessors Part II: Implementation. In *International Journal of Parallel Programming*, 16(6), pp. 479-499, 1987.

[19] T-H. Lai and S. Sahni. Anomalies in Parallel Branch-and-Bound Algorithms. In *Communications of the ACM*, 27(6): 594-602, 1984.

[20] E. Lusk, D.H.D. Warren, et. al. The Aurora Or-Prolog System. In *New Generation Computing*, Vol. 7, No. 2,3, pp. 243-273, 1990

[21] E. Pontelli, D. Ranjan, G. Gupta. On the Complexity of Or-parallelism. *New Generation Computing*, 1999 (to appear).

[22] N. Takahashi et al. Example-Based Machine Translation on a Massively Parallel Processor. In *Procs. of IJCAI*, 1993.

[23] O. Werner, A. C. Yamin, J. L. V. Barbosa, C. F. R. Geyer. OPERA Project: An Approach Towards Parallelism Exploitation on Logic Programming. In *Procs. of WLP*, pp. 20-23, 1994.