

Incremental Stack-Splitting Mechanisms for Efficient Parallel Implementation of Search-based AI Systems

K. Villaverde, E. Pontelli, H. Guo
Dept. Computer Science
New Mexico State University
{kvillave,hguo,epontell}@cs.nmsu.edu

G. Gupta
Dept. Computer Science
University of Texas at Dallas
gupta@utdallas.edu

Abstract

Incremental stack-copying is a technique which has been successfully used to support efficient parallel execution of a variety of search-based AI systems—e.g., logic-based and constraint-based systems. The idea of incremental stack-copying is to only copy the difference between the data areas of two agents, instead of copying them entirely, when distributing parallel work. In order to further reduce the communication during stack-copying and make its implementation efficient on message-passing platforms, a new technique, called stack-splitting, has recently been proposed. In this paper, we describe a scheme to effectively combine stack-splitting with incremental stack copying, to achieve superior parallel performance in a non-shared memory environment. We also describe a scheduling scheme for this incremental stack-splitting strategy. These techniques are currently being implemented in the PALS system—a parallel constraint logic programming system.

1. Introduction

Search is a popular problem solving paradigm used in many AI systems. Such *search-based AI systems* traverse the solution space (typically modeled as a tree) looking for an instance (solution) which satisfies the criteria laid out by the programmer. Examples of search-based AI systems can be found in application areas such as Expert Systems, Game playing, constraint programming, planning, and logic-based systems [15, 4]. Such search-based AI systems can spend a large amount of time looking for a solution, due to the high dimensional nature of the search space and the lack of effective and general search heuristics. Even if good heuristics were found to prune the search space, the narrow down space can still be quite large. For example, researchers in the area of planning are constantly striving to generate plans within acceptable time constraint, and are often forced to

impose limitations (e.g., maintain a small number of feasible actions) in order to obtain a solution in a reasonable amount of time [4]. Even then, the number of possible plans to consider is astronomical. Given the compute-intensive nature of search-based AI systems, the use of parallelism to improve systems' performance becomes a natural choice. Indeed, a considerable number of proposals have emerged over the years to exploit parallelism from this class of applications [16, 15, 12, 17, 14]. In particular, considerable emphasis has been placed on exploiting parallelism from the operational semantics of the programming languages commonly used to code these classes of applications (e.g., constraint programming, logic programming, and functional programming) [12, 13, 22]. This approach has the advantage of being general enough to warrant the ability to extract parallelism from virtually any application written in any of these paradigms, typically in a way that is transparent or semi-transparent to the application programmer. 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 some form of non-determinism to facilitate programming of search-based AI applications (e.g., constraint programming and rule-based programming).

Execution in a search-based system is centered around the development and exploration of a *search tree*. Each internal node represents an execution point where multiple alternatives are available to continue with the execution. Leaves of the tree represent either failure points (i.e., execution points which cannot lead to any solution) or success points (i.e., solutions to the problem). Adopting logic programming terminology, we will refer to the internal nodes in the tree as *choice-points*. A sequential computation boils down to traversal of this search tree according to some pre-defined search strategy; languages like Prolog mostly adopt a fixed strategy (a left-to-right, depth-first traversal strategy), while others may adapt their search strategy according to the structure of each individual computation (e.g.,

constraint programming languages). This structuring of the computation suggests a natural way for the exploitation of parallelism: multiple processors (*computing agents*) can explore the different branches of the search-tree in parallel [15, 16]. These multiple agents traverse the search tree looking for unexplored branches. If an unexplored branch (i.e., an unexplored alternative) is found, the agent picks it up and begins execution. This agent will stop either if it fails (reaches a failing leaf), or if it finds a solution. In case of failure, or if the solution found is not acceptable to the user, the agent will *backtrack*, i.e., move back up in the tree, looking for other choice-points with untried alternatives to explore. The agents may need to synchronize if they access the same node in the tree. 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 [15, 16, 17] as well as for the Prolog language [10, 12].

Most research on or-parallel execution of non-deterministic languages so far has focused on techniques aimed at shared-memory multiprocessors (SMMs). In this paper we are concerned with the development of execution models for exploitation of or-parallelism from Prolog programs on Message Passing Platforms (MPPs). The techniques we propose are immediately applicable to other systems based on the same underlying model, e.g., constraint [20] and non-monotonic reasoning [17] systems.

Experimental [1] and theoretical studies [18] have demonstrated that *stack-copying*, and in particular *incremental* stack-copying, is one of the most effective implementation technique for exploiting or-parallelism that one can devise. Stack-copying allows sharing of work between parallel agents by copying the state of one agent (which owns unexploited tasks) to another agent (which is currently idle). The idea of *incremental* stack-copying is to only copy the *difference* between the state of two agents, instead of entirely copying the complete state. Incremental stack-copying has been used to implement or-parallel Prolog efficiently in a variety of systems (e.g., MUSE [1], YAP [19]), as well as to exploit parallelism from other declarative systems [22, 20, 17].

In order to further reduce the communication during stack-copying and make its implementation efficient on message-passing platforms, a new technique, called stack-splitting, has recently been proposed [11]. In this paper, we describe the first ever concrete implementation of stack-splitting on a message-passing platform, along with a novel scheme to combine incremental copying with stack-splitting on MPPs. The *incremental stack-splitting* scheme is based on a procedure which labels parallel choice-points and then compares the labels to determine the fragments of data and control areas that need to be exchanged between agents. Furthermore, we describe a scheduling scheme

which is suitable to be used with this novel incremental stack-splitting scheme. The techniques described are currently being implemented in the *PALS* system, a message-passing or-parallel implementation of Prolog. In this paper we present performance results from this implementation.

2. Stack-splitting

Most research related to or-parallel implementation of non-deterministic languages so far has focused on techniques aimed at shared-memory multiprocessors (SMMs) [12, 18]. Relatively fewer efforts [21, 9, 2, 8, 7, 6] have been devoted to implementing logic and constraint programming systems on message passing platforms. Out of these efforts only a small number have been implemented as working prototypes, and even fewer have produced acceptable speed-ups. Existing techniques developed for SMMs are mostly inadequate for the needs of MPPs.¹ In fact, most implementation methods require sharing of data and/or control stacks to work correctly. Even if the need to share data stacks is eliminated—as in *stack-copying*—the need to share the control stack still exists. The control stack can be large and frequently accessed, thus degrading the performance on MPPs.

Stack-splitting is a modification of the stack-copying method and is designed to support *or-parallelism* (OP) on MPPs. To our knowledge, stack-splitting is the first model which satisfies the basic optimality criteria for OP [10, 18] and provides separation between control areas. Stack-splitting allows one to avoid sharing of the control stack, thus allowing efficient techniques devised for SMMs to be implemented on MPPs without excessive performance degradation. Stack-splitting has the potential to improve locality of computation, reduce communication between agents, and improve cache behavior. Furthermore, it allows better scheduling strategies—e.g., *scheduling on bottom-most choice-point*—even in MPP implementations of OP.

2.1. The Need for a Different Stack-Copying Model

Traditional stack-copying [1] behaves as illustrated in Fig. 1. Agent *A* provides a copy of its data structures to agent *B*. Agent *B* performs a simple backtracking to the last choice-point copied, allowing Agent *B* to restart computation with one of the unexplored alternatives.

In the traditional stack-copying technique, as implemented in the MUSE and YAP systems, backtracking on a choice-point which has been shared between two or more agents, requires acquiring exclusive access to the corresponding *shared frame* (see Fig. 1). Shared frames are associated to each copied choice-point and used to maintain

¹Adaptation of models with higher degree of sharing to MPPs, using distributed shared memory techniques, have recently been proposed [21].

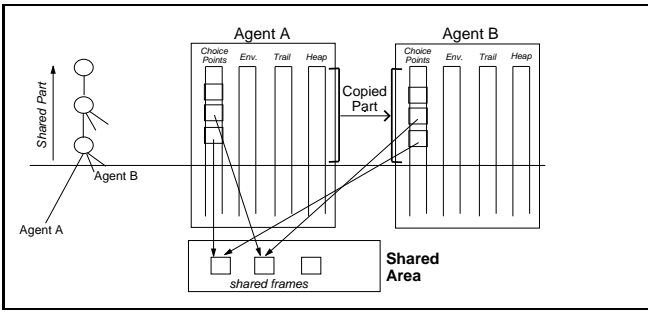


Figure 1. Stack-copying

a shared representation of the alternatives available in such choice-point. The use of shared frames with mutually exclusive access guarantees that no two agents try to explore the same alternative. This solution works fine on SMMs—where mutual exclusion is easily implemented using *locks*. However, on a MPP this process is a source of overhead—access to the shared area becomes a bottleneck [3]. Sharing of information in a MPP leads to frequent exchange of messages and hence considerable overhead. Centralized data structures, such as shared frames, are expensive to realize in a distributed setting.

Nevertheless, stack-copying has been recognized as the best representation methodology to support OP in a MPP setting [8, 9, 2, 7, 6]. This is because, while the choice-points are shared (through the shared frames), at least all the other data-structures, such as the environment, the trail, and the heap are not. To avoid the problem of sharing choice-points in distributed implementations, many developers have reverted back to the *scheduling on top-most choice-point* strategy [8, 9]. This methodology transfers between agents only the highest choice-point (i.e., closer to the root) in the computation tree which contains unexplored alternatives. The reasoning is that untried alternatives of a choice-point created higher up in the or-tree are more likely to generate large subtrees as well as minimize the amount of computation “shared” by different agents. Indeed, a system making use of scheduling on top-most choice-point can be realized without the need of shared frames.

However, if the granularity of the branches in the top-most choice-points does not turn out to be large, then another untried alternative has to be picked and a new copying operation performed. In contrast, in *scheduling on bottom-most choice-point* more work could be found via backtracking, since more choice-points are copied during the same sharing operation. Scheduling on bottom-most choice-point is characterized by the fact that all the choice-points owned by one agent are copied during a sharing operation. Additionally, scheduling on bottom-most is closer to the depth-first search strategy used by sequential systems, and facilitates support of Prolog semantics. Research done

on comparing scheduling strategies indicates that scheduling on bottom-most is superior to scheduling on top-most [5]. However, the shared nature of choice-points is a major drawback for a MPP implementation of stack-copying. The question we consider is: can we avoid sharing of choice-points while keeping scheduling on bottom-most? The answer is affirmative, as is discussed next.

2.2. Stack-Splitting Copying Model

In stack-copying, the primary reason why a choice-point has to be shared is because we want to serialize the selection of untried alternatives, so that no two agents can pick the same alternative. However, there are other simple ways of ensuring the same property: perform a splitting of the choice-points, i.e., each agent is given all the alternatives of alternate choice-points (See Fig. 2). In this case, the list of choice-points is split between the two agents. We call this operation *choice-point stack-splitting* or simply *stack-splitting*. Stack-splitting will ensure that no two agents pick the same alternative. 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 different choice-point. All the choice-points can be evenly split in this way during the copying operation. 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. This makes the stack-splitting technique highly suitable for MPPs.

The shared frames in stack-copying are used to maintain global information related to scheduling. Shared frames are also employed in MUSE [1] to detect the Prolog order of choice-points, needed to execute order-sensitive predicates (e.g., side-effects, extra-logical predicates) in the correct order. However, under stack-splitting the shared frames no longer exist; scheduling and global ordering of choice-points information will have to be maintained in some other way. They could be kept in a global shared area or distributed over multiple agents and accessed by message passing in case of MPPs.

Thus, stack-splitting does not completely remove the need of a shared description of the or-tree. On the other hand, the use of stack-splitting mitigates the impact of accessing shared resources—e.g., stack-splitting allows scheduling on bottom-most which reduces the number of calls to the scheduler.

2.3. Incremental Stack-Copying

Traditional stack-copying requires agents which share work to transfer a complete copy of the data structures rep-

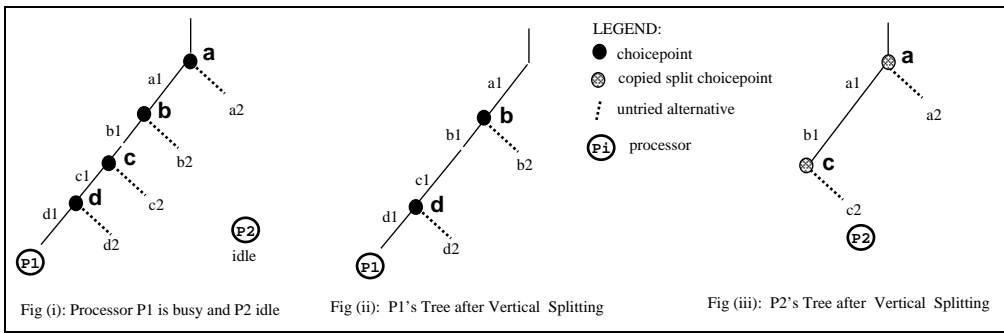


Figure 2. Splitting of Choice-points

representing the status of the computation. In the case of a Prolog computation, this may include transferring most of the choice-points along with copies of the other data areas (trail, heap, environments). Since Prolog computations can make use of large quantities of memory (e.g., generate large structures on the Heap), this copying operation can become quite expensive. Existing stack-copying systems (e.g., MUSE) have introduced a variation of stack-copying, called *Incremental Stack-Copying* [1] which allows to considerably reduce the amount of data transferred during a sharing operation. The idea is to compare the content of the data areas in the two agents involved in a sharing operation, and transfer only the difference between the state of the two agents. This is illustrated in Fig. 3. In Fig. 3(i) we have two agents (P1 and P2) which have 3 choice-points in common (e.g., from a previous sharing operation). P1 owns two additional choice-points with unexplored alternatives while P2 is out of work. If P2 obtains work from P1, then there is no need of copying again the 3 top choice-points (Fig. 3(ii)).

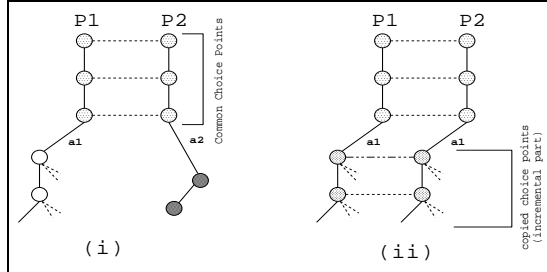


Figure 3. Incremental Stack-Copying

In the rest of the paper we describe a complete implementation of stack-splitting using a message passing platform, analyzing in detail how the various problems mentioned earlier have been tackled. In addition to the basic stack-splitting scheme, we analyze how stack-splitting can be extended to incorporate *incremental copying*, an optimization which has been deemed essential to achieve speedups in various classes of benchmarks. The solution we describe has been developed in a concrete implementation, realized by modifying the engine of a commercial Prolog system (ALS Prolog) and making use of the Message Passing

Interface (MPI) as communication platform. The ALS Prolog system is based on an efficient version of the Warren Abstract Machine (WAM).

3. Incremental Stack-splitting

During stack-splitting, all WAM data areas (the WAM model includes a stack for the choice-points, a stack for environments, a heap for the dynamic creation of terms, and a trail used to support undoing of variable binding during backtracking), except for the code area, are copied from the agent giving work to the idle one. Next, the parallel choice-points are split between the two agents. Blindly copying all the stacks every time an agent shares work with another idle agent can be wasteful, since frequently the two agents already have parts of the stacks in common due to previous copying. We can take advantage of this fact to reduce the amount of copying by performing *incremental copying*, as discussed earlier.

In order to figure out the incremental part to be copied, parallel choice-points will be *labeled*. The goal of the labeling process is to uniquely identify the original “source” of each choice-point (i.e., which agent created it), to allow unambiguous detection of copies of common choice-points.

To perform labeling, each agent maintains a counter. Initially, the counter in each agent is set to 1. The counter is increased by 1 every time the labeling procedure is performed. When a parallel choice-point is copied for the first time, a label for it is created. The label is composed of three parts: (i) agent rank, (ii) counter, and (iii) choice-point address. The agent rank is the rank (i.e., id) of the agent which created the choice-point. The counter is the current value of the labeling counter for the agent generating the labels. The choice-point address is the address of the choice-point which is being labeled. The labels for the parallel choice-points are recorded in a separate *label stack*, in the order they are created. Intuitively, the label stack keeps a record of changes done to the stacks since the last stack-splitting operation. Observe that the choice of maintaining labels in a stack—instead of associating them directly to the choice-points—has been dictated by efficiency reasons.

Let us illustrate the stack-splitting accompanied by labeling with an example. Suppose process A has just created two parallel choice-points and process B is idle. Then Process B requests work from process A. Process A first creates labels for its two parallel choice-points. These labels have their rank and counter parts as $A:1$. Process A then pushes these labels into its label stack. See Fig. 4. Notice that process A incremented its counter to 2 after the labeling procedure was over.

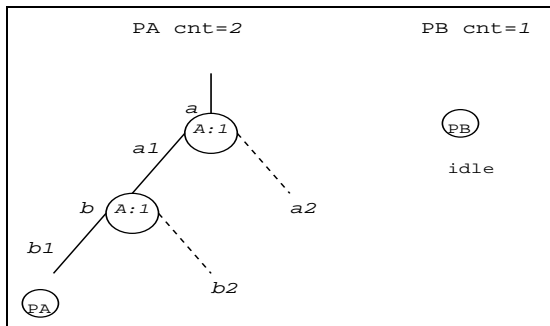


Figure 4. A Does Labeling

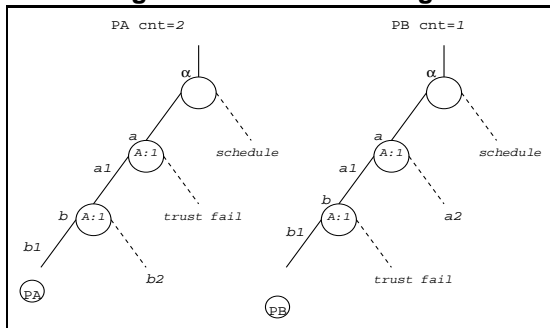


Figure 5. A Gives Work to B

Stack-copying is done next. Process B gets all the parallel choice-points of process A along with process A label stack. Then, stack-splitting takes place: process A will keep the alternative $b2$ but not $a2$, and process B will keep the alternative $a2$ but not $b2$. We have designed a new WAM scheduling instruction which is placed in the next alternative field of the choice-point above which there is no more parallel work. This scheduling instruction implements the scheduling scheme described in Sect. 4. To avoid taking the original alternative of a choice-point, we change its next alternative field to WAM instruction *trust_fail*. See Fig. 5. Afterwards, process B backtracks, removes choice-point b along with its corresponding label in the label stack, and then takes alternative $a2$ of choice-point a .

3.1. Incremental Stack-splitting: The Procedure

In this section we describe how the label stacks are used to compute the incremental part to be copied. Assume pro-

cess W is giving work to process I. Process W will label all its unlabeled parallel choice-points and will push them into its label stack. Process W then increments its counter.

If process I label stack is empty, then non-incremental stack-copying will need to be performed followed by stack-splitting. Process I then tries its new work via backtracking.

However, if process I label stack is not empty then process I sends its label stack to process W which compares it against its own. Comparison will go from the first label entered in the stacks to the last one. The objective is to find the last choice-point ch with a common label. In this way, processes W and I are guaranteed to have the same computation *above* the choice-point ch , while their computations will be different below such choice-point.

If the choice-point ch does not exist, then non-incremental stack-copying will need to be performed followed by stack-splitting. However, if choice-point ch does exist, then process I backtracks to choice-point ch , and performs incremental-copying. Process W sends its choice-point stack starting from choice-point ch to the top of its choice-point stack along with the corresponding label stack. Stack-splitting is then performed. Afterwards, process I tries its new work via backtracking.

We illustrate this procedure by the following example. Suppose process A has three parallel choice-points and process C requests work from A. Process A first labels its last two unlabeled parallel choice-points and then increments its counter. Afterwards, process C sends its label stack to process A. Process A compares the label stacks and finds the last choice-point ch with a common label (Fig. 6).

Now, process C backtracks to choice-point ch . Incremental stack-copying is performed. Then, stack-splitting takes place (Fig. 7). C backtracks to choice-point i and takes alternative $i2$.

3.2. Incremental Stack-splitting: Challenges

Four issues that were not discussed above and which are fundamental for the correct implementation of incremental stack-splitting are discussed below.

Sequential Choice-points: The first issue has to do with the sequential choice-points that are located among the parallel choice-points that will be split between two agents. The alternatives of these choice-points should be kept in only one process otherwise, we may have repeated, useless or wrong computations.

Installation Process: The second issue has to do with the bindings of conditional variables (i.e., variables that may be bound differently in different or-parallel branches) which may not be copied during the incremental stack-splitting process. This can be fixed by having the process giving work create when necessary a stack of all these conditional variables along with their bindings. The stack is then sent

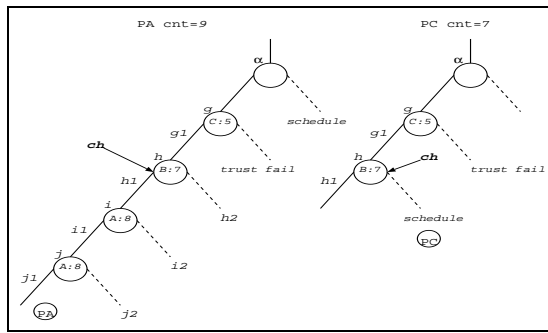


Figure 6. A Compares Labels with C

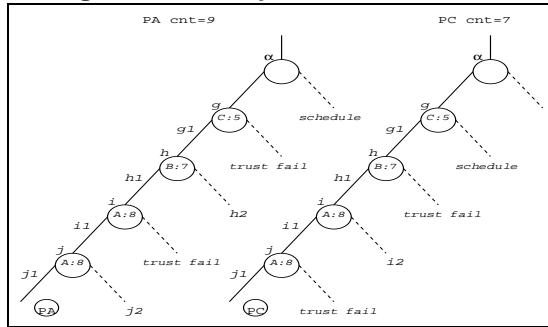


Figure 7. A Gives Work to C

to the process receiving work so that it can perform the installation.

Garbage Collection: The third issue arises when garbage collection takes place. When this happens, relocation of choice-points may also take place. Hence, the labels may no longer label the correct parallel choice-points and the label stack should be invalidated.

Next Clause Fields: The fourth issue arises when the next clause fields of the parallel choice-points above the last common choice-point *ch* are not the same compared to the ones in the agent receiving work. This situation occurs after several copying and splitting operations. Hence, the process giving work should send these next clause fields to the process receiving work. Then the splitting of all parallel choice-points can take place correctly.

4. Scheduling

We adopt a simple and fair distributed algorithm to implement a scheduling strategy in the PALS system. A new data structure—the *load vector*—is introduced to indicate the work loads of different agents. The work load of an agent is approximated by the number of parallel choice-points present in its local computation tree. Each agent keeps a work load vector V in its local memory, and the value of $V[i]$ represents the work load of the agent with rank i . Based on the work load vector, an idle agent can re-

quest parallel work from other agent with the greatest work load, so that parallel work can be fairly distributed. The load vector is updated at runtime. When stack-splitting is performed, a `Load_Info` message with updated load information will be broadcasted to all the agents so that each agent has the latest information of work load distribution. Additionally, load information is attached with each incoming message. For example: when a `Request_Work` message is received from agent P_1 , the value of P_1 's work load, 0, can be inferred.

Based on its work load each agent can be in one of two states: *scheduling* state or *running* state. When an agent has some work to do, it is in a running state, otherwise, it is in a scheduling state. An agent that is running, occasionally checks whether there are incoming messages. Two possible types of messages are checked by the running agent: one is `Request_Work` message sent by an idle agent, and the other is `Send_Load_Info` message, which is sent when stack-splitting occurs. The idle agent in scheduling state is also called a scheduling agent. An idle agent wants to get work as soon as possible from another agent, preferably the one that has the largest amount of work. The scheduling agent searches through its local load vector for the agent with the greatest work load, and then sends a `Request_Work` message to that agent asking for work. If all the other agents have no work, then the execution of the current query is finished and the agent halts. When a running agent receives a `Request_Work` message, stack-splitting will be performed if the running agent's work load is greater than the splitting threshold, otherwise, a `Reply_Without_Work` message with a positive work load value will be sent as a reply. If a scheduling agent receives a `Request_Work` message, a `Reply_Without_Work` message with work load 0 will be sent as a reply.

5. Implementation and Performance

The stack-splitting procedure has been implemented on top of the commercial ALS Prolog system using the MPI library for message passing. The whole system runs on a Sun Enterprise 4500 with 14 processors. While the Sun Enterprise is an SMM, it should be noted that all communication—during scheduling, copying, splitting, etc.—is done using messages. Migration of the system to a truly distributed Beowulf machine (a network of Pentium II nodes connected by a Myrinet Switch) is currently under way, and has so far confirmed the experimental results observed on the Sun Enterprise.

The benchmarks that we have used to test our system are the following. The *8 Queens* and *10 Queens* benchmarks consist of placing a number of queens on a chessboard so that no two queens are on the same line. The *Map Coloring* benchmark consists of coloring a planar map. The *Hamil-*

ton benchmark consists of finding a closed path through a graph such that all the nodes of the graph are visited once. The *Knight* benchmark consists of finding a path of knight-moves on a chessboard of size 5 so that every square of the board is visited just once. The *Send More* benchmark consists of solving the classical crypto-arithmetic puzzle. The *Solitaire* benchmark is a solution to the standard game involving a triangular board with pegs and one empty hole. The *8 Puzzle* benchmark is a solution to the puzzle involving a 3-by-3 board with 8 numbered tiles. The *Send More*, *Solitaire*, and *8 Puzzle* benchmarks compute only one solution while the other benchmarks compute all the solutions.

The timing results in seconds from our incremental stack-splitting system, which has just become operational, are presented in Table 1. This system is based on the scheduling strategy described above.

Benchmark	# Agents				
	1	2	4	8	14
<i>8 Queens</i>	0.306	0.198	0.143	0.157	0.149
<i>Map Coloring</i>	2.036	1.298	0.696	0.479	0.430
<i>Hamilton</i>	6.895	3.879	1.940	1.151	0.761
<i>10 Queens</i>	7.575	3.922	2.087	1.378	1.141
<i>Solitaire</i>	12.912	7.598	3.813	2.029	1.335
<i>8 Puzzle</i>	52.945	29.601	15.026	7.845	4.754
<i>Send More</i>	115.183	65.271	31.447	16.496	9.686
<i>Knight</i>	275.737	141.213	70.528	35.539	22.403

Table 1. Incremental Stack-splitting (sec.)

Although the results are very encouraging, it should be noted that they are very preliminary since no performance tuning has been done. For benchmarks with substantial running time (*Knight*, *Send More*, and *8 Puzzle*) the speed-ups are very good. We also observe that for benchmarks with not so substantial but also not very small running time (*Solitaire*, *10 Queens*, and *Hamilton*) the speed-ups are still quite good. Note that for the benchmarks with small running time (*Map coloring* and *8 Queens*) the speed-ups deteriorate. See Fig. 8 under the label *Incremental*.

The observations made above are consistent with our belief that parallel systems should be used for parallelizing programs with somewhat substantial running times. For programs with small-running times, there is not enough work to offset the cost of exploiting parallelism. Nevertheless, our system is reasonably efficient, given that even for small benchmarks it can produce reasonable speed-ups.

In order to compare our incremental stack-splitting system we have also implemented two other techniques using *non-incremental* stack-copying. One of these techniques is based on stack-splitting, and the other is based on scheduling on top-most choice-point. Both implementations also used the scheduling mechanisms described above. The speed-ups for these systems are shown in Fig. 8 under the labels *Complete* and *Top*.

Most benchmarks show that the incremental stack-splitting system obtains higher speed-ups than the non-incremental systems. Between the non-incremental systems, the stack-splitting system performs better in most of the benchmarks than the scheduling on top-most choice-point system. This is particularly evident in the case of the *Hamilton* benchmark (Fig. 8). Some of the benchmarks show almost no difference in performance among the three systems. One of the reasons why this is happening is that during the execution of these benchmarks there must be only one or two parallel choice-points which are given away or split per sharing. Analyzing the source code for these benchmarks, we see that we have just one parallel choice-point which contains all the parallel work.

Finally, the incremental stack-splitting system, although still not optimized, introduces a reasonably small overhead with respect to the original sequential ALS Prolog system. Our PALS system, on a single agent, is on average 30% slower than the sequential ALS system.

6. Conclusions and Future Work

In this paper we proposed a novel scheme to implement incremental stack-splitting for or-parallelism on message-passing platforms. The novel method allows to take advantage of the higher locality and independence of computation threads allowed by stack-splitting, without losing the advantages of incremental copying. Furthermore, we also described a scheduling strategy for the incremental stack-splitting based implementation. These techniques have been implemented in the ALS Prolog system, and performance results from this preliminary implementation were reported.

We have almost completed the migration of the PALS system to a Bewoulf distributed memory machine, and the preliminary results confirm the validity of our methodology (these results, which are too preliminary to report here, seem to show speedups that are even better than those for the Sun Enterprise multiprocessor reported in this paper). In order to guarantee a sustained good speed-up on machines with larger number of processors (e.g., our Bewoulf system has 32 nodes), we are exploring alternative scheduling methodologies (e.g., centralized vs. distributed scheduling, top vs. bottom scheduling). We are also performing a low-level performance tuning of the system, and developing methodologies to support side-effects and other extralogical predicates.

Acknowledgments

We are grateful to Ken Bowen and Charles Houpt for providing us access to the ALS system, and to V. Santos Costa for the numerous suggestions. This

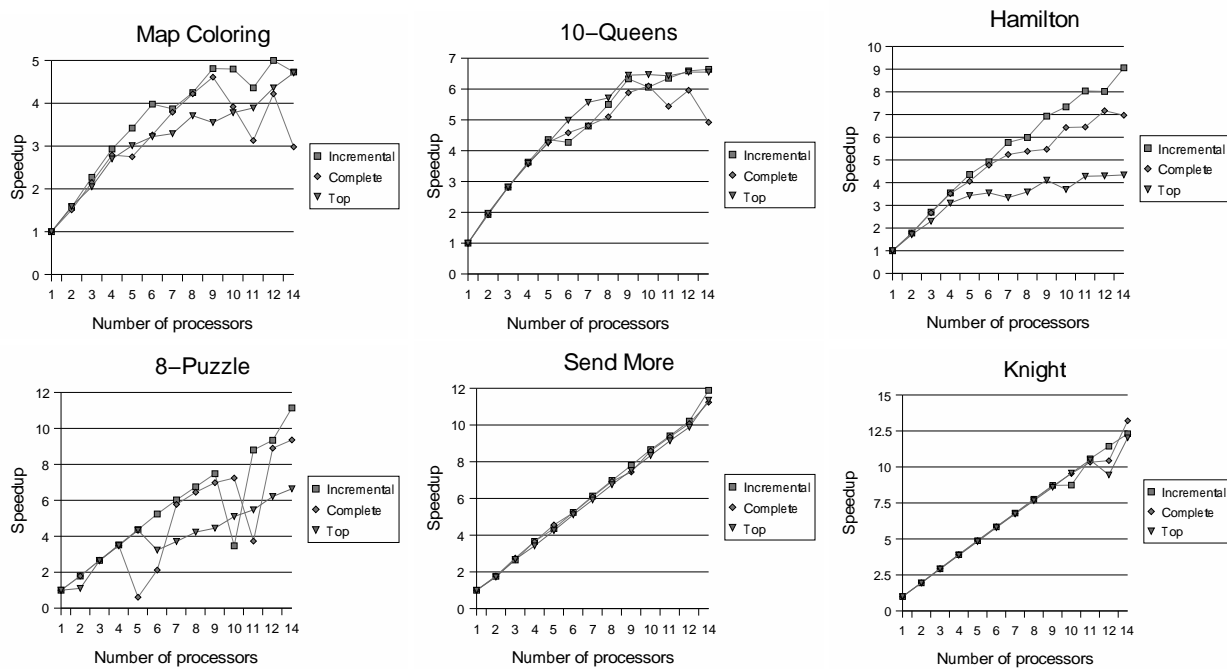


Figure 8. Speed-ups for Selected Benchmarks

work has been supported by NSF grants CCR9875279, CCR9900320, CDA9729848, EIA9810732, CCR9820852, and HRD9906130, and by a graduate fellowship from the US Department of Education.

References

- [1] K. Ali and R. Karlsson. The Muse Or-parallel Prolog Model. In *N. American Conf. Logic Progr.*, MIT Press, 1990.
- [2] L. Araujo and J. Ruz. A Parallel Prolog System for Distributed Memory. *J. Logic Programming*, 33(1), 1998.
- [3] H. Babu. Porting muse on ipsc860. Master's thesis, New Mexico State University, 1996.
- [4] C. Baral, V. Kreinovich, and R. Trejo. Computational Complexity of Planning and Approximate Planning in Presence of Incompleteness. In *IJCAI*, 1999.
- [5] A. Beaumont and D. H. D. Warren. Scheduling Speculative Work in Or-Parallel Prolog Systems. In *Int. Conf. Logic Progr.*, 1993. MIT Press.
- [6] V. Benjumea and J. Troya. An OR Parallel Prolog Model for Distributed Memory Systems. In *PLILP*, 1993. Springer.
- [7] J. Briat et al. OPERA: Or-Parallel Prolog System on Supernode. In *Implementations of Distributed Prolog*, J. Wiley & Sons, 1992.
- [8] L. Castro et al. DAOS: Scalable And-Or Parallelism. In *EuroPar*, 1999. Springer Verlag.
- [9] W.-K. Foong. *Combining and- and or-parallelism in Logic Programs*. PhD thesis, University of Melbourne, 1995.
- [10] G. Gupta and B. Jayaraman. Analysis of Or-parallel Execution Models. *ACM TOPLAS*, 15(4):659–680, 1993.
- [11] G. Gupta and E. Pontelli. Efficient Techniques for Distributed Implementation of Search-based AI Systems. In *ICPP*. IEEE Computer Society, 1999.
- [12] G. Gupta, E. Pontelli et al. Parallel Execution of Prolog Programs: a Survey. *Transactions on Computational Logic*, 2001. (to appear).
- [13] R. Halstead. Implementation of Multilisp: Lisp on a Multiprocessor. In *Symp.on LISP and Functional Programming*, 1984. ACM Press.
- [14] H. Kitano and J. Hendler, editors. *Massive Parallel Artificial Intelligence*. AAAI/MIT Press, 1994.
- [15] V. Kumar and L. Kanal. Parallel Depth-First Search on Multiprocessors. *Int. Journal of Parallel Programming*, 16(6):479–499, 1987.
- [16] T.-H. Lai and S. Sahni. Anomalies in Parallel Branch-and-Bound Algorithms. *CACM*, 27(6):594–602, 1984.
- [17] E. Pontelli and O. El-Kathib. Construction and Optimization of a Parallel Engine for Answer Set Programming. In *Practical Aspects of Declarative Languages*, 2001. Springer.
- [18] D. Ranjan, E. Pontelli, and G. Gupta. On the Complexity of Or-Parallelism. *New Generation Computing*, 17(3), 1999.
- [19] V. Santos Costa, L. Damas, R. Reis, and R. Azevedo. *YAP User's Manual*. University of Porto, 1999.
- [20] C. Schulte. Comparing Trailing and Copying for Constraint Programming. In *Int. Conf. Logic Progr.*, MIT Press, 1999.
- [21] F. Silva and P. Watson. Or-Parallel Prolog on a Distributed Memory Architecture. *J. Logic Programming*, 43(2):173–186, 2000.
- [22] P. Van Hentenryck. Parallel Constraint Satisfaction in Logic Programming: Preliminary Results of CHIP within PEPSys. In *Int. Conf. Logic Programming*, 1989. MIT Press.