PALS: An Or-Parallel Implementation of Prolog on Beowulf Architectures

K. Villaverde[1], E. Pontelli[1], H. Guo[3], and G. Gupta[2]

[1] Dept. Computer Science, New Mexico State University
{kvillave,epontell}@cs.nmsu.edu
[2] Dept. Computer Science, Univ. Texas at Dallas
gupta@utdallas.edu
[3] Dept. Computer Science, SUNY Stony Brook

Abstract. This paper describes the development of the *PALS* system, an implementation of Prolog that efficiently exploits or-parallelism on *share-nothing* platforms. PALS makes use of a novel technique, called *incremental stack-splitting*. The technique builds on the stack-splitting approach, which in turn is an evolution of the stack-copying method used in a variety of parallel logic systems. This is the first distributed implementation based on the stack-splitting method ever realized. Experimental results obtained on a Beowulf system are presented and analyzed.

1 Introduction

Or-parallelism (OP) arises from the non-determinism implicit in the process of reducing a given subgoal using different clauses of the program. The non-deterministic structure of a logic programming execution is commonly depicted in the form of a *search tree* (a.k.a. *or-tree*). Each internal node represents a choice-point, i.e., an execution point where multiple clauses are available to reduce the selected subgoal. Leaves of the tree represent either failure points (i.e., resolvents where the selected subgoal does not have a matching clause) or success points (i.e., solutions to the initial goal). A sequential computation boils down to traversal of this search tree according to some predefined search strategy. While a sequential execution attempts to use one clause at the time to reduce each subgoal, eventually using backtracking to explore the use of alternative clauses, OP allows the use of different threads of execution (*computing agents*) to concurrently explore distinct alternatives emanating from a choice-point. If an unexplored branch (i.e., an untried clause to resolve a selected subgoal) 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. Intuitively, OP allows the concurrent search of alternative solutions to the original goal. The importance of the research on efficient techniques for handling OP arises from the generality of the problem—technology originally developed

P. Codognet (Ed.): ICLP 2001, LNCS 2237, pp. 27–42, 2001.

for parallel execution of Prolog has found application in areas such as constraint programming (e.g., [17,13]) and non-monotonic reasoning (e.g., [14]).

Most research on OP execution of Prolog 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 OP from Prolog programs on *Distributed Memory Architectures (DMPs)*—i.e., architectures that do not provide any centralized memory resource. The techniques we propose are immediately applicable to other systems based on the same underlying model, e.g., constraint programming [17] and non-monotonic reasoning [14] systems. Other proposals for OP on DMPs have also been recently proposed [8,18,3].

Experimental [1] and theoretical studies [15] have also demonstrated that *stack-copying*, and in particular *incremental* stack-copying, is one of the most effective implementation techniques for exploiting OP 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. Incremental stack-copying has been used to implement or-parallel Prolog efficiently in a variety of systems (e.g., MUSE [1], YAP [16]), as well as to exploit parallelism from constraint systems [17] and non-monotonic reasoning systems [14]. In order to further reduce the communication during stack-copying and make its implementation efficient on share-nothing 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 DMP platform—specifically a Pentium-based Beowulf—along with a novel scheme to combine incremental copying with stack-splitting on DMPs. The *incremental stack-splitting* scheme is based on a procedure which labels choice-points and then compares the labels to determine the fragments of memory areas that need to be exchanged between agents. We also describe a scheduling scheme which is suitable to be used with this novel incremental stack-splitting scheme. Both the incremental stack-splitting and the scheduling schemes described have been implemented in the *PALS* system, a message-passing OP implementation of Prolog. In this paper we present performance results obtained from this implementation. To our knowledge, PALS is the first OP implementation of Prolog on a Beowulf architecture (built from off-the-shelf components).

2 Stack-Splitting

Relatively few efforts [18,9,3,8,7,6] have been devoted to implementing logic programming systems on DMPs. Some of the older proposals (e.g., [7,6]) relied on variations of stack-copying, while the most recent proposals (e.g., [8,18]) make use of alternative schemes. 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 DMPs. 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.

2.1 The Need for a Different Stack-Copying Model

Traditional stack-copying relies on idle agents copying data structures from busy agents in order to obtain new tasks. In traditional stack-copying, as implemented in MUSE, backtracking on a choice-point which has been shared between two or more agents, requires acquiring exclusive access to the corresponding *shared frame*. Shared frames are associated to each copied choice-point and used to maintain 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 explore the same alternative. This solution works well on SMMs—where mutual exclusion is implemented using *locks*. However, on a DMP this process is a source of overhead, since the shared area becomes a bottleneck [4].

Nevertheless, stack-copying has been recognized as one of the best representation methodologies to support OP in a DMP setting [9,3,7,6]. 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. Other environment representation schemes proposed for OP require more extensive sharing of data structures and seem less suitable to support execution on DMPs (although some recent efforts for adapting the binding array scheme to DMPs—through the use of distributed shared-memory—have been studied [18,8]). 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 [3,6,9]. This methodology transfers between agents only the highest choice-point (i.e., closer to the root) in the computation or-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. Furthermore, this is guaranteed to be the only choice-point with unexplored alternatives shared between agents. 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 can 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]. This is especially true for stack-copying because: (i) the number of copying operations is minimized; and, (ii) the alternatives in the choice-points copied are "cheap" sources of additional work, available via backtracking. However, the shared nature of choice-points is a major drawback for stack-copying on DMPs.

2.2 Stack-Splitting Copying Model

In the stack-copying approach, 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. The shared frame is locked while the alternative is selected to achieve this effect. 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. 1). 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*.

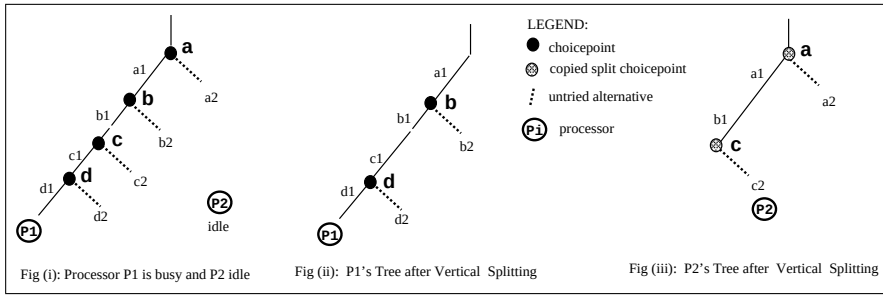

Fig. 1. Splitting of Choice-points

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 independent of each other, and hence communication is minimized during execution. This makes the stack-splitting technique highly suitable for DMPs. Observe that alternative splitting strategies may also be designed—e.g., dividing the alternatives within each choice-point between the two agents [11].

The shared frames in the stack-copying technique are used to maintain global information related to scheduling. The shared frames provide a global description of the or-tree, and each shared frame records which agent is working in which part of the tree. This last piece of information is needed to support scheduling in stack-copying systems—work is taken from the agent that is "closer" in the or-tree, thus reducing the amount of information to be copied. The shared frames ensure accessibility of this information to all agents, providing a consistent view of the computation. However, under stack-splitting the shared frames no longer exist; scheduling and work-load information will have to be maintained in some other way. They could be kept in a global shared area, as in the case of SMMs—e.g., by building a representation of the or-tree—or distributed over multiple

agents and accessed by message passing in case of DMPs. 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) in the correct order. As in the case of scheduling, some information regarding global ordering of choice-points needs to be maintained to execute order-sensitive predicates in the correct order. In this paper however we do not handle side-effects and order sensitive predicates. 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.

Stack-splitting has the potential to improve locality of computation, reduce communication between agents, and improve cache behavior. Indeed, the SMM implementation of stack-splitting described in [11] achieves on many benchmarks better speedups than traditional stack copying. The ability to reuse the same technology on both SMMs and DMPs is also a key to development of Prolog systems on *Clusters of SMMs*, i.e., distributed systems with SMMs as nodes.

2.3 Incremental Stack-Copying

Traditional stack-copying requires agents which share work to transfer a complete copy of the data structures 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 amounts of memory, 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 transfer only the difference between the data areas of the two agents. Incremental stack-copying, in a shared-memory context, is relatively simple to realize—the shared frames can be used to identify which choice-points are in common and which are not [1].

In the rest of the paper we describe a complete implementation of stack-splitting on a DMP 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 speed-ups 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 MPI as communication platform. The ALS Prolog engine is based on the Warren Abstract Machine (WAM).

3 Incremental Stack-Splitting

During stack-splitting, all WAM data areas, 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*. In order to figure out the incremental part that only needs to be copied during incremental stack-splitting, 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.

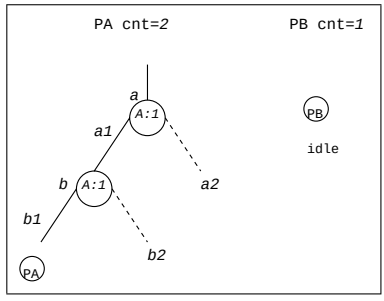

Fig. 2. A Labels its Choice-points

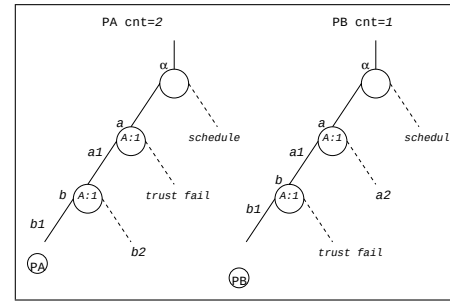

Fig. 3. A Gave Work to B

To perform labeling, each agent maintains a counter. 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: *(1)* agent rank, *(2)* counter, and *(3)* 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. Also, when a parallel choice-point is removed from the stack, its corresponding label is also removed from the label stack (this is actually integrated with the variable untrailing mechanism). Initially, the label stack in each agent is set to *empty*. Intuitively, the label stack keeps a record of changes done to the stacks since the last stack-splitting operation. 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. Process A first creates labels for its two parallel choice-points. These labels have their rank and counter parts as *A:1*. Process A pushes these labels into its label stack (Fig. 2).

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 Section 4. To avoid taking the original alternative of a choice-point, we change its next alternative field to WAM instruction *trust_fail*. See Fig. 3. 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

Assume process W is giving work to process I. Process W will label all its parallel choice-points which have not been labeled before and will push them into its label stack. If process I label stack is empty, then non-incremental stack-copying will need to be performed followed by stack-splitting. Process W sends its complete choice-point stack and its complete label stack to process I. Then stack-splitting is performed on all the parallel choice-points of process W. However, if process I label stack is not empty then process I sends its label stack to process W. Process W compares its label stack against the label stack from I. 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.

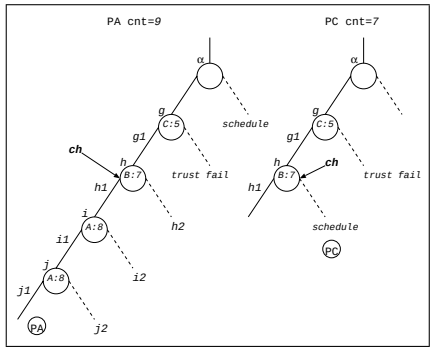

Fig. 4. Labels Comparison

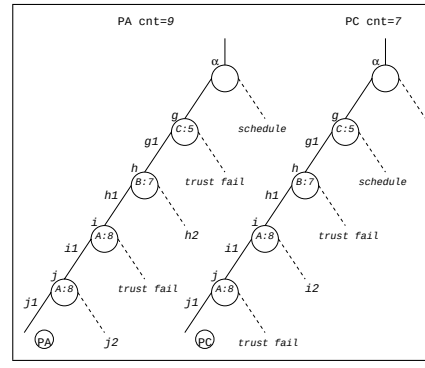

Fig. 5. Proc. A Gave Work to Proc. C

If the choice-point *ch* does not exist, then non-incremental stack-copying will need to be performed followed by stack-splitting, as described before. 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. Process W also sends its label stack starting from the label corresponding to *ch* to the top of its label stack. Stack-splitting is then performed on all the parallel choice-points of W.

We illustrate the above 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 parallel choice-points which have not been labeled before

and then increments its counter. Afterwards, process C sends its label stack to process A. Process A compares its label stack against the label stack of process C and finds the last choice-point *ch* with a common label. Above choice-point *ch*, the Prolog trees of processes A and C are equal. See Fig. 4. Now, process C backtracks to choice-point *ch*. Incremental stack-copying can then take place. Process A sends its choice-point stack starting from choice-point *ch* to the top of its choice-point stack, and stack-splitting is performed (Fig. 5).

3.2 Incremental Stack-Splitting: Challenges

Sequential Choice-points: The first issue has to do with sequential choice-points that are located among the parallel choice-points shared by two agents. If the alternatives of these choice-points are kept in both processes, we may have repeated or wrong computations. Hence, the alternatives of these choice-points should only be kept in one process (e.g., the one giving work). If the alternatives are kept in the process giving work, then the process that is receiving work should change the next alternative field of these choice-points to the instruction *trust_fail* to avoid taking the original alternatives of these choice-points.
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 splitting process. This can be fixed by having the process giving work create a stack of all these conditional variables along with their bindings. This stack will then be sent to the process receiving work so that it can update the bindings.
Garbage Collection: When garbage collection takes place, relocation of choice-points may also occur. Hence, the labels in our label stack may no longer label the correct parallel choice-points. Therefore, we need to modify our labeling procedure so that when garbage collection on an agent takes place, the label stack of this agent is invalidated. The next time this process gives work, non-incremental stack-copying will have to take place. This solution is analogous to the one adopted in the original implementation of the MUSE system [1].

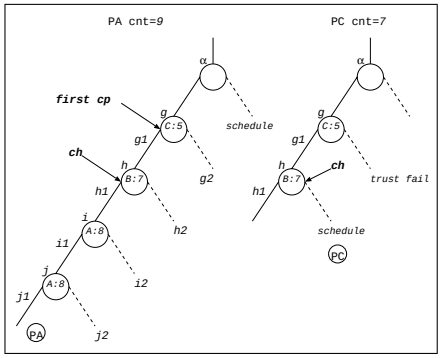

Fig. 6. Copy Nextclause from *first cp* to *ch*

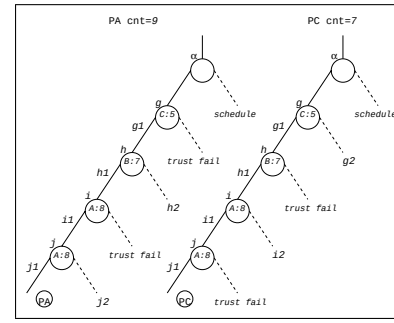

Fig. 7. C Received Next-Clause Fields

Next Clause Fields: The fourth issue arises when the next clause fields of the parallel choice-points between the first parallel choice-point *first cp* and the last choice-point *ch* with a common label in the agent giving work are not the same compared to the ones in the agent receiving work. This situation occurs after several copying and splitting operations. In this case, we cannot just copy the part of the choice-point stack between choice-point *ch* and the top of the stack and then perform the splitting. This is because the splitting will not be performed correctly. For example, suppose that in our previous example when process C requests work from process A, we have this situation (Fig. 6). We can see that choice-point *g* should be given to process C. But process C does not have the right next clause field for this choice-point. The problem can be solved by having the process giving work send all the next clause fields between its first choice-point *first cp* and choice-point *ch* to the process receiving work. Then the splitting of all parallel choice-points can take place correctly. See Fig. 7.

4 Scheduling

The main objective of a scheduling strategy is to balance the amount of parallel work done by different agents. Additionally, work distribution among agents should be done with minimal communication overhead. These two goals are somewhat at odds with each other, since achieving perfect balance may result in a very complex scheduling strategy with considerable communication overhead, while a simple scheduling strategy which re-distributes work less often will incur low communication overhead but poor balancing of work. Therefore, it is obvious that there is an intrinsic contradiction between distributing parallel work as even as possible and minimizing the distribution overhead. Thus our main goal is to find a trade-off point that results in a reasonable scheduling strategy.

We adopt a simple distributed algorithm to implement a scheduling strategy in PALS. A 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 request 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. 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.

The distributed scheduling algorithm mainly consists of two parts: one is for the scheduling agent, and the other is for the running 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. The running agent's algorithm can be briefly described as follows: each incoming message can be either a `Send_LoadInfo` message—i.e., a notification of a change in load for some processors—or a `Request_Work` message—i.e., a request for sharing, which is accepted if the local load is above a given threshold. At fixed time intervals (which can be selected at initialization of the system) the agent examines the content of its message queue for eventual pending messages. `Send_LoadInfo` messages are quickly processed to update the local view of the overall load in the system. Messages of the type `Request_Work` are handled as described above. Observe that the concrete implementation actually checks for the presence of the two types of messages with different frequency (i.e., request for work messages are considered less frequently than requests for load update).

5 Implementation and Performance

Stack-Splitting: The stack-splitting procedure has been implemented by modifying the commercial ALS Prolog system, using the MPI library for message passing. The only major data structures added to the ALS system are: the label stack, the load vector, and buffers in order transfer information. The whole system runs on a truly distributed machine (a network of 32 Pentium II nodes connected by Myrinet-SAN Switches). All communication—during scheduling, copying, splitting, etc.— is done using explicit message passing via MPI.

The benchmarks used to test our system are standard benchmarks drawn from the pool of programs frequently used to evaluate OP systems (e.g., *Queens*, *Knight*, *Solitaire*). The benchmarks selected are simple but provide sufficiently different program structures to validate the parallel engine. The timing results in seconds from our incremental stack-splitting system are presented in Table 1. The modifications made to the ALS WAM are very localized and reduced to the minimum. This has allowed us to keep a clean design—that can be easily ported to other WAM-based implementations—and to contain the parallel overhead—our engine on a single processor is on average 5% slower than ALS WAM. The corresponding speed-ups are presented in Fig. 8 (with label *incremental*).

Note that for benchmarks with substantial running time the speed-ups are quite good, while for programs with smaller running time the speed-ups deteri-

Table 1. Timings for Incremental Stack-Splitting (Time in sec.)

Benchmark	**# Processors**					
	1	**2**	**4**	**8**	**16**	**32**
Knight	159.950	81.615	40.929	20.754	10.939	8.248
Send More	61.817	32.953	17.317	8.931	4.923	3.916
8 Puzzle	27.810	15.387	8.442	10.522	3.128	5.940
Solitaire	5.909	3.538	1.811	1.003	0.628	0.535
10 Queens	4.572	2.418	1.380	0.821	1.043	0.905
Hamilton	3.175	1.807	0.952	0.610	0.458	0.486
Map Coloring	1.113	0.702	0.430	0.319	0.318	0.348
8 Queens	0.185	0.162	0.166	0.208	0.169	0.180

orate. This is consistent with our belief that DMP implementations should be used for parallelizing programs with coarse-grained parallelism. For programs with small running times, there is not enough work to offset the communication costs on DMPs. Nevertheless, our system is reasonably efficient, given that even for small benchmarks it can produce speed-ups. It is also interesting to observe that in no cases we have observed slow-downs due to parallel execution—thanks to simple granularity control mechanisms embedded in the scheduler. For some benchmarks the speedup graphs are somewhat irregular – specially the *8 Puzzle*. We believe that the reason behind this hides in the scheduling strategy used.

One of the objectives of the experiments performed is to validate the effectiveness of incremental stack-splitting for efficient exploitation of parallelism on DMPs. In particular, there are two aspects that we were interested in exploring: *(i)* verifying the effectiveness of stack-splitting versus a more "direct" implementation of stack-copying (i.e., keeping single copies of choice-points around the system); *(ii)* verifying the impact of *incremental* splitting. Validity of stack-splitting vs. stack-copying can be inferred from the experiments described in the next subsection: a direct implementation of stack-copying would produce the same amount of communication traffic as some of the variations of scheduling tested, and thus incur the same kind of problems described next. In order to evaluate the impact of incrementality, we have measured the performance of the system on the selected benchmarks without the use of incremental splitting—i.e., each time a sharing operation takes place, a complete copy of the data areas is performed. The results obtained from this experiment are in Fig. 8: the figure compares the speed-ups observed with and without incremental copying. We can observe that incremental stack-splitting obtains higher speed-ups than the non-incremental stack-copying. The difference is more significant in benchmarks with a large number of choice-points, where incrementality is applied more frequently.

Scheduling: One of the major reasons to adopt stack-splitting is the ability to perform scheduling on bottom-most choice-point. Other DMP implementations of OP have resorted to scheduling on the top-most choice-point, where only

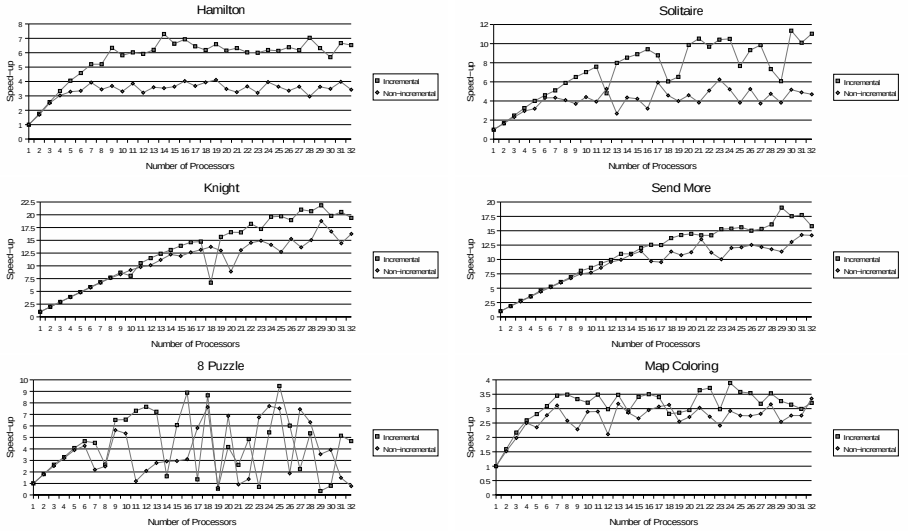

Fig. 8. Incremental Stack-Splitting vs. Non-Incremental Stack-Splitting

the oldest choice-point with unexplored alternatives is exchanged between processors. Top-most scheduling will share only one choice-point at the time, thus relieving the engine from the need of controlling access to shared choice-points.

To validate the effectiveness of our claim, we have developed a top-most scheduler for our system and compared its performance with that of the incremental stack-splitting with bottom-most scheduling. Fig. 9 compares the speed-ups observed using the two different schedulers. In the figure we have reported the behavior only of those benchmarks where significant differences in performance have been recorded. In all other benchmarks, top-most and bottom-most scheduling provide similar results, as a small number of choice-points are cre-

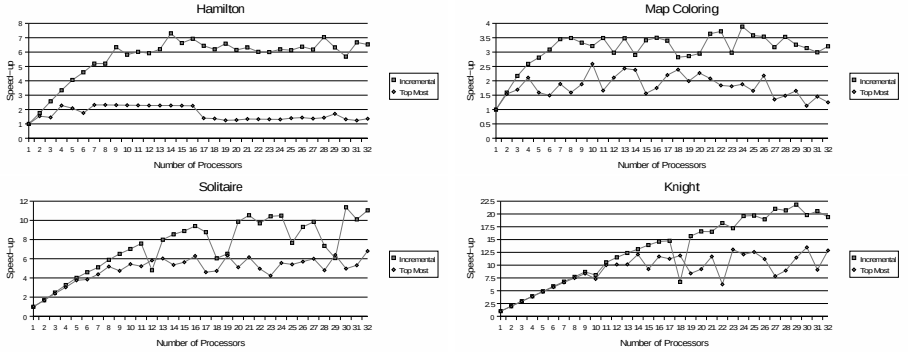

Fig. 9. Incremental Stack-Splitting vs. Top Most Scheduling

ated and only one at a time is shared between processors. As we can observe from Fig. 9, bottom-most scheduling provides a sustained speed-up considerably higher than top-most scheduling. This is due to the reduced number of calls to the scheduler performed during the execution—processors spend a higher fraction of their time doing useful work compared to scheduling on top-most.

Another aspect of our implementation that we are interested in validating is the performance of the distributed scheduler. As mentioned in Sect. 4, our scheduler is based on keeping in each processor an "approximated" view of the load in each other processor. The risk that this method may encounter is that a processor may have out-of-date information concerning the load in other processors, and as a consequence it may try to request work from idle processors or ignore processors that may have unexplored alternatives. Fig. 10 provides some information concerning the number of attempts that a processor needs to perform before receiving work. The figure on the left measures the average number of requests that a processor has to send; as we can see, the number is very small (1 or 2 requests are typically sufficient) and such number is generally better if we adopt bottom-most scheduling. The figure on the right shows the maximum number of requests observed; these numbers tend to grow towards the end of the computation (when less work is available)—nevertheless, typically only one or two processors achieve these maximum values, while the majority of the processors remain close to the average number of attempts.

To further validate our scheduling approach, we have compared it with an alternative scheduling scheme developed in PALS. This alternative scheme is an implementation of a *centralized* scheduling algorithm, designed following the guidelines of the scheduler used in Opera [7]. In the centralized approach, only one processor, called *central*, is in charge of keeping track of the load information. Idle processors send their requests for work directly to the central processor. In turn, the central processor is in charge of implementing a matchmaking algorithm between idle and busy processors. When stack-splitting occurs, only the central processor is informed about the load information update. Fig. 11 compares the speed-ups achieved using centralized scheduling with the speed-ups observed using the distributed scheduling approach.[1] As evident from the figure, the speed-ups observed in centralized scheduling are almost negligible—this is due to the inability of the scheduling method to promptly respond to the requests for new work. Also, the use of a reasonably fast network (Myrinet) leads to the creation of a severe bottleneck at the level of the centralized scheduler.

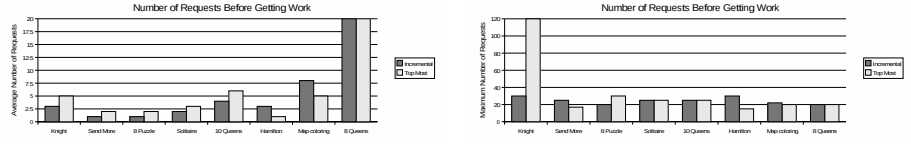

Fig. 10. Average and Maximum Number of Tries to Acquire Work

[1] We had to limit the experiments to a smaller number of CPUs due to unavailability of half of the machine at that time.

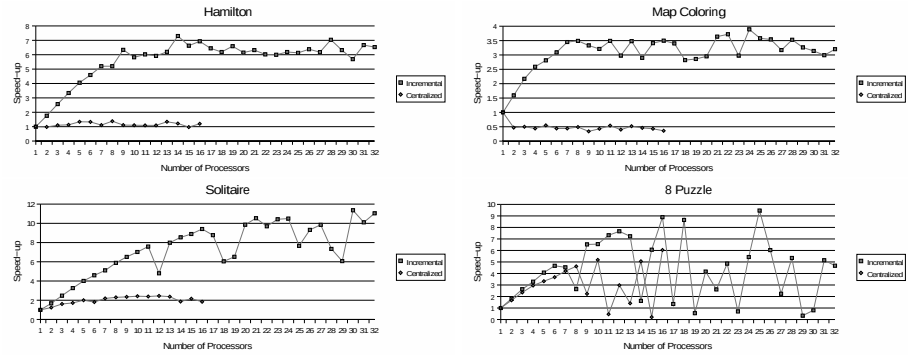

Fig. 11. Incremental Stack-Splitting vs. Centralized Scheduling

The results presented in [5] suggest that random selection of work may also provide a simple and effective alternative when searching for work. We have experimented with this idea, by modifying the scheduler to select any busy processor for scheduling. The idea is to avoid bottleneck situations where multiple idle processors are concentrating their requests for work towards the same busy processor. We have named this new version of the scheduler *Random Scheduler*. In this version, an idle processor searches its load vector for the next processor with load greater than a given small threshold. Fig. 12 compares the speed-ups observed in the Random scheduler with those from the standard bottom-most scheduling with selection of processor with highest load. The results indicate that the Random scheduler is less effective. This suggests that selecting work from the processor with highest load is not a severe bottleneck and sending requests to lightly loaded processors may increase the number of calls to the scheduler.

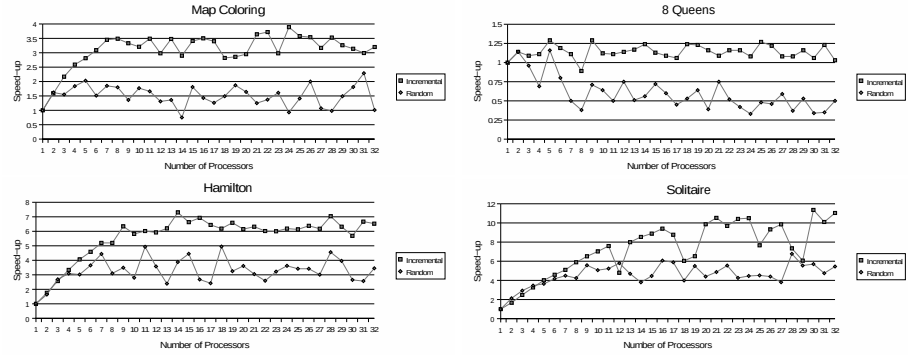

Fig. 12. Incremental Stack-Splitting vs. Random Scheduling

6 Related Work and Conclusions

In this paper we proposed a novel scheme to implement incremental stack-splitting for OP on DMPs. 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. The incremental stack-splitting scheme presented is based on a procedure which labels parallel choice-points and then compares the labels to determine the incremental WAM areas to be copied. Furthermore, we described a scheduling strategy for incremental stack-splitting. The incremental stack-splitting scheme and the scheduling strategy have been implemented in the ALS Prolog system, and performance results from this implementation were reported. To our knowledge, PALS is the first ever or-parallel implementation of Prolog on Beowulf systems.

A relatively small number of proposals can be found in the literature dealing with execution of Prolog on DMPs. Some of the existing environment representation models proposed (e.g., Conery's Closed Environments) have been designed with distributed memory in mind, but they have never been concretized in actual implementations. Most of the older systems implemented on DMPs [7,6] are based on stack copying and have been designed with respect to a specialized architecture (Transputers). Their schedulers are tailored for this class of architectures and they all resort to top scheduling to reduce communication costs. PDP [3] makes use of a recomputation approach to deal with OP, and has also been developed on Transputers. MUSE version on switch based multiprocessors [2] (e.g., Butterfly) gives good speedups for very coarse grain applications but uses distributed shared-memory techniques. Only in recent years a renovated effort towards developing models for generic DMP architectures have emerged. These include DAOS [8] and Dorpp [18] based on variations of the binding arrays method and relying on distributed shared-memory technology; DAOS has not reported any implementation result, while Dorpp has been executed on simulators (with fairly good results). In contrast to DAOS and Dorpp, we opted to continue using stack copying with a fully distributed scheduler. For comparison of stack-splitting with other existing approaches see [11].

Acknowledgments: We are grateful for the help received from K. Bowen, C. Houpt, and V. Santos Costa. This work has been partially supported by NSF grants CCR-9875279, CCR-9900320, CDA-9729848, EIA-9810732, CCR-9820852, and HRD-9906130, and by a fellowship from the Dept. of Education.

References

1. K.A.M. Ali and R. Karlsson. The Muse Or-parallel Prolog Model and its Performance. In *N. American Conf. on Logic Prog.*, pages 757–776. MIT Press, 1990.
2. K.A.M. Ali, R. Karlsson, and S. Mudambi Performance of Muse on Switch-Based Multiprocessors Machines. *New Generation Computing*, 11(1):81-103, 1992.
3. L. Araujo and J. Ruz. A Parallel Prolog System for Distributed Memory. *J. of Logic Programming*, 33(1):49–79, 1998.
4. H. Babu. Porting MUSE on ipsc860. Master's thesis, NMSU, 1996.

5. A.J. Beaumont and D. H. D. Warren. Scheduling Speculative Work in Or-Parallel Prolog Systems. In *ICLP*, pages 135–149, 1993. MIT Press.
6. V. Benjumea and J.M. Troya. An OR Parallel Prolog Model for Distributed Memory Systems. In *PLILP*, pages 291–301, 1993. Springer Verlag.
7. J. Briat et al. OPERA: Or-Parallel Prolog System on Supernode. In *Implementations of Distributed Prolog*, pages 45–64. J. Wiley & Sons, New York, 1992.
8. L.F. Castro et al. DAOS: Scalable And-Or Parallelism. In *Euro-Par*, pages 899–908, 1999. Springer Verlag.
9. W-K. Foong. *Combining and- and or-parallelism in Logic Programs: a distributed approach*. PhD thesis, University of Melbourne, 1995.
10. G. Gupta and E. Pontelli. Optimization Schemas for Parallel Implementation of Nondeterministic Languages and Systems. In *IPPS*, 1997. IEEE Computer Society.
11. G. Gupta and E. Pontelli. Stack-splitting: A Simple Technique for Implementing Or-parallelism on Distributed Machines. In *ICLP*. MIT Press, pages 290–304, 1999.
12. G. Gupta, E. Pontelli, M. Carlsson, M. Hermenegildo, and K.M. Ali. Parallel Execution of Prolog Programs: a Survey. *ACM TOPLAS*, 2001. (to appear).
13. L. Perron. Search Procedures and Parallelism in Constraint Programming. In *PPDP*, pages 346–360, 1999. Springer Verlag.
14. E. Pontelli and O. El-Kathib. Construction and Optimization of a Parallel Engine for Answer Set Programming. In *PADL*, pages 288–303, 2001. Springer Verlag.
15. D. Ranjan, E. Pontelli, and G. Gupta. On the Complexity of Or-Parallelism. *NGC*, 17(3):285–308, 1999.
16. R. Rocha, F. Silva, and V. Santos Costa. YapOr: an Or-Parallel Prolog System based on Environment Copying. In *EPPIA*, pages 178–193, 1999, Springer Verlag.
17. C. Schulte. Parallel Search Made Simple. In *TRICS*, number TRA9/00, pages 41–57, University of Singapore, 2000.
18. F. Silva and P. Watson. Or-Parallel Prolog on a Distributed Memory Architecture. *Journal of Logic Programming*, 43(2):173–186, 2000.