Available online at www.sciencedirect.com
ScienceDirect
J. Parallel Distrib. Comput. 66 (2006) 1267–1293
Journal of Parallel and Distributed Computing
www.elsevier.com/locate/jpdc

Stack splitting: A technique for efficient exploitation of search parallelism on share-nothing platforms ☆

Enrico Pontelli[a,*], Karen Villaverde[a], Hai-Feng Guo[b], Gopal Gupta[c]

[a] *Department of Computer Science, New Mexico State University, Las Cruces, NM 88003, USA*
[b] *Department of Computer Science, University of Nebraska at Omaha, Omaha, NE, USA*
[c] *Department of Computer Science, University of Texas at Dallas, Richardson, TX, USA*

Received 2 August 2004; received in revised form 6 September 2005; accepted 7 May 2006
Available online 10 July 2006

Abstract

We study the problem of exploiting *parallelism* from search-based AI systems on *share-nothing* platforms, i.e., platforms where different machines do not have access to any form of shared memory. We propose a novel environment representation technique, called *stack-splitting*, which is a modification of the well-known *stack-copying* technique, that enables the efficient exploitation of or-parallelism from AI systems on distributed-memory machines. Stack-splitting, coupled with appropriate scheduling strategies, leads to reduced communication during distributed execution and effective distribution of larger grain-sized work to processors. The novel technique can also be implemented on shared-memory machines and it is quite competitive. In this paper we present a distributed implementation of or-parallelism based on stack-splitting including results. Our results suggest that stack-splitting is an effective technique for obtaining high performance parallel AI systems on shared-memory as well as distributed-memory multiprocessors.

Keywords: Logic programming; Artificial intelligence; Or-Parallelism; Beowulf clusters

> Krishna picked up a twig, pointed it towards Bhima, and split it into two through the middle.
>
> — *The Mahabharata*

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 exploring a large search-space, looking for a solution that satisfies certain criteria [35,30,36]. 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. 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, the use of parallel processing to speedup the search is an obvious path to follow to obtain high execution efficiency. In fact, considerable research has been done on exploiting parallelism from search-based AI system [26,30,32,33,35–37]. Two approaches have been generally followed: (i) techniques have been developed and implemented for extracting parallelism from specific AI systems (e.g., [32]), (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 and Lisp) [26,27]. In both cases, it is the operation of searching the solution-space that is parallelized. It should also be mentioned that most work done so far for exploiting parallelism in AI largely falls under (ii). Nearly not as much work has been done on (i), for the

☆ This paper is an extended and revised version of the paper "Incremental Stack-Splitting Mechanisms for Efficient Parallel Implementation of Search-based AI Systems", appeared in the Proceedings of the IEEE Computer Society International Conference on Parallel Processing (ICPP) 2001.

* Corresponding author. Fax: +1 505 646 1002.
E-mail addresses: epontell@cs.nmsu.edu (E. Pontelli), kvillave@cs.nmsu.edu (K. Villaverde), haifengguo@mail.unomaha.edu (H.-F. Guo), gupta@utdallas.edu (G. Gupta).

0743-7315/$ - see front matter © 2006 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2006.05.002

obvious reason that (ii) represents a more general approach. Within (ii) considerable work has been done on parallelizing Prolog, the most popular logic programming language. In the rest of the paper, we will present our techniques and results in the context of parallel logic programming systems, specifically parallel Prolog systems, though they can equally well be applied to specific AI systems that incorporate search, as well as other languages that incorporate search mechanisms to facilitate programming of search-based AI applications. An example of the former kind is described in [43] where parallelism is exploited from a non-monotonic reasoning system.

Implementing search (parallel or sequential) requires that we have a representation of the search space, which is usually a tree (called a *search tree*), in the memory. 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 [26]. The obvious way to search this tree in parallel is to have multiple processors explore the different branches of this search tree in parallel [30,35–37]. 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) is found, the processor picks it up and begins execution. This processor will stop either if it fails (reaches a dead-end), i.e., some condition is not met and hence the processor determines that the solution cannot lie on that branch, or if it finds a solution. 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. The processors have 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 [35–37] as well as for the Prolog language [26].

In sequential implementations of search-based AI systems or Prolog, typically one branch of the tree resides on the stacks at any given time. This simplifies implementation quite significantly. However, in the 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 a difficult problem, and it is referred to as the *multiple environments management problem* [21,22].

Most research so far has focused on techniques targeted towards shared-memory symmetric multiprocessors (SMPs). Relatively fewer attempts [17,3,13,11,9,48] have been made to implement Prolog systems on share-nothing platforms. Only a small number of these proposals have been implemented as working prototypes, and even fewer have produced significant speedups. Various implementations rely on out-of-date and fairly specialized architectures, such as transputers [3,9,11], while others have been developed only as simulators [48]. Existing techniques developed for SMPs are inadequate for the needs of distributed share-nothing platforms. In fact, most implementation methods require sharing of data and control stacks to work correctly. Even if the need to share data stacks is eliminated—as in *stack copying* [2] or in *stack recomputation* [14]—the need to share parts of the control stack still exists. The control stack can be large, thus degrading the performance on distributed-memory platforms. One can conclude that the state-of-the-art of parallel Prolog systems on share-nothing platforms is far behind that of parallel Prolog systems on SMPs.

In this paper we present a method for exploiting the or-parallelism found in AI systems and Prolog on share-nothing platforms. This method, called stack-splitting, allows the efficient techniques devised for realizing or-parallelism on shared-memory multiprocessor machines to also be effectively implemented on distributed-memory multiprocessor machines, by naturally incorporating efficient mechanisms for scheduling work and reducing communication overhead [25]. Stack-splitting has the potential to:

- improve locality of computation, reduce communication between parallel threads, and increase memory access efficiency (e.g., better cache behavior);
- allow better scheduling strategies (specifically *scheduling on bottom-most choice-point*) to be deployed even in distributed-memory implementations of or-parallelism. This is not the case for many of the models proposed for or-parallel execution of Prolog programs on share-nothing multiprocessors, where proven efficient scheduling techniques [2] have been sacrificed in favor of less effective techniques to reduce the communication overhead (e.g., *scheduling on top-most choice-point*) [11,3].

The fact that our techniques for implementing search-based systems on share-nothing architectures are based on simple modifications of efficient techniques that have proved to work on SMPs gives us confidence that they will work better than many approaches previously proposed. Our intuition has been confirmed by the results we have obtained from our parallel implementations.

In this paper we present details of our stack-splitting technique. We describe a prototype implementation of stack-splitting on shared-memory multiprocessors and on a Linux-based Beowulf cluster. We describe the results of the extensive cache-level performance evaluation of the shared-memory implementation. This performance evaluation was done in order to understand the low-level behavior of stack-splitting. The results were in turn used to estimate the level of performance obtainable on a distributed-memory multiprocessor before embarking on an implementation of stack-splitting on such scalable machines. It is important to stress that our is the first ever implementation of Prolog on Beowulf platform to be developed.

Our implementation and performance studies indicate that stack-splitting can be efficiently implemented on SMPs as well as share-nothing platforms. On SMPs, we anticipated that the method's performance will be comparable to stack-copying [2]—our implementation shows that our method's performance is, in fact, better. The real benefit of the method is apparent on share-nothing platforms, on which efficient implementation becomes possible. The ability to reuse the same technology on