17 Run-Time Management of Lisp Parallelism and the Hierarchical Task Graph Program Representation

M. Furnari
Istituto de Cibernetica
C. Polychronopoulos
University of Illinois at Urbana-Champaign

Abstract

This paper suggests how to extend the Hierarchical Task Graph program representation, and its execution model to address the LISP code parallelization problems. The advantages of this approach lie in the fact of not annotating LISP programs, and in accounting for run-time scheduling policies all into a unified environment. We start by reviewing the problem of run-time parallelism management, first in the imperative languages setting, and next in LISP setting. Before describing the Hierarchical Task Graph (HTG) program representation [7], we review the basic notions of control and data dependence analysis. Finally we describe how to modify the HTG and its execution model to take into account for the Lazy Task Creation model [19], and the Sponsor Model for exploiting speculative computations in LISP [20].

1 Introduction

The general applicability and use of parallel processing depends critically on the effective and efficient exploitation of parallelism at all granularity levels. However, when a parallel algorithm is manually coded, the resulting program often produces tasks of a finer grain than that which the implementation can exploit efficiently. To overcome these difficulties the parallelization task has been split into two problems, the first one involves the discovering of parallel opportunities in a program, the second deals with the efficient management of the execution of the parallelized code.

To cope with the problem of discovering the parallel opportunities in a program, two approaches have been pursued, either using parallelizing compilers (restructuring systems), or using parallel language extensions (annotating systems).

To improve the execution of parallelized code, tools for building tasks of acceptable granularity have been developed. They are essentially based on a cheap way to create processes. But fine-tuning may be necessary in some cases to maximize performance, with possible
drawbacks such as the cost in programming effort and the program clarity. Furthermore, this often involves calibration of program parameters through experimentation, and this work may be repeated for a different target machine or data set. Or, worst yet, it may become impractical when the program runs in a multiprogramming/multiprocessing environment. The complexity of parallel programming makes fine performance-tuning impossible at the programmer level, and in a general-purpose environment.

The key to solve this problem is the ability of the compiler to:

- compile the program once and package parallel threads at all granularity levels, and
- adjust the granularity of parallel threads created at run-time, and based on the size and type of the rest of the workload present in the system.

In fact, to be really usable the parallelization must enable the user to run simultaneously several parallel programs on the same machine without a significant drop on parallel program performance.

For the imperative languages there are many efforts addressing this problem, for example, Polychronopoulos [22] suggests moving the scheduling policy inside the program, by introducing the notion of autoscheduling. For the Lisp-like languages Mohr proposed in [19] the Lazy Task Creation strategy; Osborne in [20] proposed the sponsor model to cope with symbolic speculative computations, and Pehoushek in [21] proposed to split the global ready processes queue on each processors to reduce contention on accessing it.

In this paper we describe the use of an extension of the autoscheduling technique, the Hierarchical Task Graph program representation[7], and its corresponding computational model, in the Lisp realm. Starting from the algorithm described in [22], we show step by step how to modify it to accommodate the lazy tasks creation computation model [19] and the touching sponsor for the speculative computations [20]. The main advantage of this approach lies in avoiding to annotate the Lisp programs.

In Section 2 we report on the current status of the parallel Lisp extensions, and the strategies used to manage task granularity at run-time. Attention is focussed on the lazy task creation proposed in [19], and the sponsor model for speculative computations proposed in [20].

In Section 3 we review and summarize the most important notions of control and data dependence analysis. Next, in Section 4 we describe the efforts in using this framework to parallelize imperative languages. Special attention will be payed to the problem of automatically discovering and managing at run-time different parallel constructs, with a dynamic and varying granularity. In this context we review the HTG, and the task allocation algorithm proposed in [22]. Finally, in Section 5, we extend the previous proposal to Lisp, showing how the HTG is a good candidate for synthesizing both the lazy task creation, and the sponsor model avoiding the use of ad hoc techniques such as described in [19,20].

2 Parallelism and Lisp-Like languages

In order to cope with parallelism in Lisp two approaches have been pursued:

- the first one, extends Lisp with parallel constructs to help the compiler to parallelize a Lisp program. In other words, the parallelism is annotated by the added language constructs (annotating systems) [9,10,3,4,5].