Supporting SPMD Execution for Dynamic Data Structures

A. Rogers
Princeton University, Princeton, New Jersey

J. Reppy
AT&T Bell Labs

L. Hendren
McGill University, Montréal, Québec, Canada

Abstract
In this paper, we address the problem of supporting SPMD execution of programs that use recursively-defined dynamic data structures on distributed memory machines. The techniques developed for supporting SPMD execution of array-based programs rely on the fact that arrays are statically defined and directly addressable. As a result, these techniques do not apply to recursive data structures, which are neither statically defined nor directly addressable. We propose a three pronged approach. First, we describe a simple mechanism for migrating a thread of control based on the layout of heap allocated data. Second, we explain how to introduce parallelism into the model using a technique based on futures and lazy task creation[21]. Third, we present the compiler analyses and parallelization techniques that are required to exploit the proposed mechanism.

1 Introduction
Compiling for distributed memory machines has been a very active area of research in recent years[4, 6, 16, 18, 19, 24, 25, 26, 29]. Much of this work has concentrated on array-based programs that use arrays as their primary data structure and loops as their primary control structure. These programs tend to have the property that the arrays can be partitioned into relatively independent pieces and therefore operations performed on these pieces can proceed in parallel. It is this property of scientific programs that has led to impressive results in the development vectorizing and parallelizing compilers[1, 2, 22, 28]. More recently this property has been exploited by researchers investigating methods for automatically generating parallel programs for SPMD (Single-Program, Multiple-Data) execution on distributed memory machines. In this paper, we address the problem of the automatic generation of SPMD parallel programs that operate on recursively-defined dynamic data structures. Such programs typically use list-like or tree-like data structures, and have recursive procedures as their primary control structure.

From a compilation standpoint, the most important property of a distributed memory machine is that each processor has its own address space; non-local references are satisfied through explicitly passed messages, which are expensive. Therefore, arranging a computation so that most references are local is crucial to producing efficient code. The aforementioned properties of scientific programs make them ideal applications for distributed memory machines. Each group of related data can be placed on a separate processor, which allows operations on independent groups to be done in parallel with little interprocessor communication.

The key insight underlying recently developed methods for automatically parallelizing programs for distributed memory machines is that the layout of a program's data should determine how the work in the program is assigned to processors. Typically, the programmer specifies a mapping of the program's data onto the target machine and the compiler uses this mapping to decompose the program into processes. The

\*This work was supported, in part, by NSF Grant ASC-9110766 and by DARPA and ONR under contract number N00014-91-J-4039.

\*The research supported, in part, by FCAR, NSERC, and the McGill Faculty of Graduate Studies and Research.
simplest compilation strategy, sometimes called runtime resolution, inserts code to determine at runtime which processor needs to execute a particular statement. Different policies for allocating work are possible but the most popular is the ownership rule: the work of an assignment statement (v := o), including the computation of o, is assigned to the processor owns that v. Control statements such as conditionals and loop are executed by all processors. The code produced by this method can be improved substantially using the arsenal of techniques developed for vectorizing compilers, such as data dependence analysis and loop restructuring[2, 22, 28].

Runtime resolution works because arrays are static in nature, that is, names are available for all elements of an array at compile-time. To determine the processor responsible for a given array element, the programmer supplied mapping function is applied to the array element’s global name. Since every processor knows the global name of every array element, this test can be done locally without communication. Techniques for improving runtime resolution code rely on the fact the expressions used to reference array elements tend to be very simple and have nice mathematical properties.

Now let us return to our problem of parallelizing programs that use dynamic data structures. These programs often exhibit the required property that their data structures can be partitioned into relatively independent pieces. For example, a tree can be recursively partitioned into smaller, independent sub-trees, and a list can be recursively partitioned into its head and its tail. Furthermore, often this partitioning can be used to distribute parallel tasks over the sub-pieces. One such natural parallel sub-division arises in many divide-and-conquer programs. Unfortunately, the techniques used for scientific programs do not work for dynamic data structures. The first problem is that determining that operations on a dynamic data structure are independent is more difficult than determining that operations on an array are independent. This is partially due to the fact the nodes of a dynamic data structure do not have compile-time names and therefore references to a structure do not share the nice mathematical properties of array references. Secondly, recursive functions, rather than loops with their easily pardonable index sets, are the primary control structure for use with dynamic data structures. Finally, without compile-time names the mapping of nodes to processors cannot be done statically, and the owner of a node cannot be determined, in general, without interprocessor communication.

A recent paper by Gupta[7] suggests a mechanism for addressing the problem of global names so that an approach similar to runtime resolution can be used. In his approach, a global name is assigned to every element of a dynamic data structure and this name is made known to all processors. To accomplish this, a name is assigned to each node as it is added to a data structure. This name is determined by the node’s position in the structure and is registered with all processors as part of adding it to the structure. The mapping of a node to a processor is also based on its position in the tree. As an example, a breadth-first numbering of the nodes might be used as a naming scheme for a binary tree. Once every processor has a name for the nodes in a data structure, it can traverse the structure without further communication.

It is important to note that this new way of naming dynamic data structures leads to restrictions on how the dynamic data structures are constructed. For example, because the name of a node is determined by its position, only one node can be added to a structure at a time. Another ramification of Gupta’s naming scheme is that node names may have to be reassigned when a new node is introduced. For example, consider a list in which a node’s name is simply its position in the list. If a node is added to the front of the list, the rest of the list’s nodes will have to be renamed to reflect their change in position.

Applying runtime resolution to dynamic data structures is difficult because this strategy was developed for statically-defined, directly-addressable, rectangular arrays. Dynamic data structures are neither statically defined, nor directly addressable. We propose a more dynamic approach that is better matched to the dynamic nature of the data structures themselves. As in earlier approaches, the programmer is responsible for mapping data to processors. However, we propose that this be done at runtime, by specifying a processor name with every memory allocation request. In addition, rather than making each processor decide if it owns the data, we migrate the thread of computation to the processor that owns the data. Thus, as a dynamic structure is traversed recursively the computation will migrate to the processor that owns that part of the structure. Before presenting our three-pronged approach to the problem of supporting