

Techniques for Reducing the Overhead of Run-time Parallelization

Hao Yu and Lawrence Rauchwerger *

Dept. of Computer Science
Texas A&M University
College Station, TX 77843-3112
{h0y8494, rwerger}@cs.tamu.edu

Abstract

Current parallelizing compilers cannot identify a significant fraction of parallelizable loops because they have complex or statically insufficiently defined access patterns. As parallelizable loops arise frequently in practice, we have introduced a novel framework for their identification: speculative parallelization. While we have previously shown that this method is inherently scalable its practical success depends on the fraction of ideal speedup that can be obtained on modest to moderately large parallel machines. Maximum parallelism can be obtained only through a minimization of the run-time overhead of the method, which in turn depends on its level of integration within a classic restructuring compiler and on its adaptation to characteristics of the parallelized application. We present several compiler and run-time techniques designed specifically for optimizing the run-time parallelization of sparse applications. We show how we minimize the run-time overhead associated with the speculative parallelization of sparse applications by using static control flow information to reduce the number of memory references that have to be collected at run-time. We then present heuristics to speculate on the type and data structures used by the program and thus reduce the memory requirements needed for tracing the sparse access patterns. We present an implementation in the Polaris infrastructure and experimental results.

1 Run-Time Parallelization Requires Static Compiler Analysis

To achieve a high level of performance for a particular program on today's supercomputers, software developers are often forced to tediously hand-code optimizations tailored to a specific machine. Such hand-coding is difficult, increases the possibility of error over sequential programming, and the resulting code may not be portable to other machines. The large human effort that is involved in parallelizing code is still keeping parallel programming a task for highly qualified scientists and has kept it from entering mainstream computing. The only avenue for bringing parallel processing to every desktop is to make parallel programming as easy (or as difficult) as programming current uniprocessor systems. This can be achieved through good programming languages and, mainly, through automatic compilation.

* Research supported in part by NSF CAREER Award CCR-9734471, NSF Grant ACI-9872126, DOE ASCI ASAP Level 2 Grant B347886 and a Hewlett-Packard Equipment Grant

Restructuring, or parallelizing, compilers address this need by detecting and exploiting parallelism in sequential programs written in conventional languages as well as parallel languages (e.g., HPF). Although compiler techniques for the automatic detection of parallelism have been studied extensively over the last two decades (see, e.g., [9, 18]), current parallelizing compilers cannot extract a significant fraction of the available parallelism in a loop if it has a complex and/or statically insufficiently defined access pattern. Typical examples are complex simulations such as SPICE [8], DYNA-3D [17], GAUSSIAN [6], CHARMM [1].

In previous work [13] we have shown that a viable method to improve the results of classic, static automatic parallelization is to employ run-time techniques that can trace 'relevant' memory references and decide whether a loop is parallel or not. Run-time techniques can succeed where static compilation fails because they have access to the input data. For example, input dependent or dynamic data distribution, memory accesses guarded by run-time dependent conditions, and subscript expressions can all be analyzed unambiguously at run-time. In contrast, at compile-time the access pattern of some programs cannot be determined, sometimes due to limitations in the current analysis algorithms but most often because the necessary information is just not available, i.e., the access pattern is a function of the input data. For example, compilers usually conservatively assume data dependences in the presence of subscripted subscripts. Although more powerful analysis techniques could remove this last limitation when the index arrays are computed using only statically-known values, nothing can be done at compile-time when the index arrays are a function of the input data [4, 15, 19].

In [11] we have presented the general principles of run-time parallelization implementation. Briefly, such run-time parallelization can be effective, i.e., obtain a large fraction of the available speedup by reducing the associated run-time overhead. This can be achieved through a careful exploitation of *all or most* available *partial* static information by the compiler to generate a minimal run-time activity (for reference tracing and subsequent analysis). To achieve significant performance gains both compiler and run-time techniques need to take into account the specific characteristic of the applications. While it is difficult and maybe, for now, impractical to specialize the compilation technology to each individual code, we have found two important classes of reference patterns that need to be treated quite differently: dense and sparse accesses.

In our previous work we have mostly discussed how to efficiently implement the LRPD test for the dense case. In this paper we will emphasize the compiler and run-time techniques required by sparse applications. As we will show later, the run-time disambiguation of sparse reference patterns requires a rather different new implementation and presents serious challenges in obtaining good speedups.

We will first present some more generally applicable techniques to reduce the run-time overhead of run-time testing through shadow reference aggregation. More specifically we will show how we can reduce the number and instances of memory references traced during execution by using statically available control- and data-flow information. Then we will present specific shadow structures for sparse access patterns. Finally we will present experimental results obtained through implementation in the Polaris infrastructure to illustrate the benefits of our techniques.

2 Foundational Work - The LRPD Test for Dense Problems

We have developed several techniques [13, 12, 14] that can detect and exploit loop level parallelism in various cases encountered in irregular applications: (i) a speculative method to detect fully parallel loops (The LRPD Test), (ii) an inspector/executor technique to compute wavefronts (sequences of mutually independent sets of iterations that can be executed in parallel) and (iii) a technique for parallelizing *while* loops (do loops with an unknown number of iterations and/or containing linked list traversals). In this paper we will mostly refer to the LRPD test and how it is used to detect fully parallel loops. To make this paper self-contained we will now briefly describe a simplified version of the speculative LRPD test.

2.1 The LRPD Test

The LRPD test speculatively executes a loop in parallel and tests subsequently if any data dependences could have occurred. If the test fails, the loop is re-executed in a safe manner, e.g., sequentially. To qualify more parallel loops, *array privatization* and *reduction parallelization* can be speculatively applied and their validity tested after loop termination.¹ For simplicity, reduction parallelization is not shown in the example below; it is tested in a similar manner as independence and privatization.

Consider a *do* loop for which the compiler cannot statically determine the access pattern of a shared array A (Fig. 1(a)). We allocate the shadow arrays for marking the write accesses, A_w , and the read accesses, A_r , and an array A_{np} , for flagging non-privatizable elements. The loop is augmented with code (Fig. 1(b)) that will mark during speculative execution the shadow arrays every time A is referenced (based on specific rules). The result of the marking can be seen in Fig. 1(c). The first time an element of A is written during an iteration, the corresponding element in the write shadow array A_w is marked. If, during any iteration, an element in A is read, but never written, then the corresponding element in the read shadow array A_r is marked. Another shadow array A_{np} is used to flag the elements of A that *cannot* be privatized: an element in A_{np} is marked if the corresponding element in A is both read and written, and is read first, in any iteration.

A post-execution analysis, illustrated in Fig. 1(c), determines whether there were any cross-iteration dependencies between statements referencing A as follows. If $\text{any}(A_w(\cdot) \wedge A_r(\cdot))^2$ is true, then there is at least one flow- or anti-dependence that was not removed by privatizing A (some element is read and written in different iterations). If $\text{any}(A_{np}(\cdot))$ is true, then A is not privatizable (some element is read before being written in an iteration). If *Atw*, the total number of writes marked during the parallel execution,

¹ *Privatization* creates, for each processor cooperating on the execution of the loop, private copies of the program variables. A shared variable is privatizable if it is always written in an iteration before it is read, e.g., many temporary variables. A *reduction variable* is a variable used in one operation of the form $x = x \otimes exp$, where \otimes is an associative and commutative operator and x does not occur in exp or anywhere else in the loop. There are known transformations for implementing reductions in parallel [16, 7, 5].

² any returns the “OR” of its vector operand’s elements, i.e., $\text{any}(v(1 : n)) = (v(1) \vee v(2) \vee \dots \vee v(n))$.