Implementation of a Run Time System to support parallelization of Partially Parallel loops using R-LRPD Test

CS738 Project

Pranav Garg (Y5313), Virajith Jalaparthi (Y5510)

INTRODUCTION

Static parallelization of loops is not possible if they have complex or statically insufficiently defined access patterns. Run time parallelization is the only solution in such cases. The Recursive-LRPD test helps to dynamically extract and exploit the maximum available parallelism present in any loop. In this test, we speculatively execute a loop as doall, and then apply a fully parallel data dependence test to check for cross processor dependencies. If the test fails, all the iterations before the earliest detected dependence are committed and the process recurses on the remaining iterations. The R-LRPD test thus gives far better speed ups than the LRPD test, in which after the detection of any cross processor dependencies, the entire loop is executed sequentially. With the R-LRPD test we get proportional speed ups for any available parallelism in the loop and in the worst case, the parallelized program works at least as fast as the sequential program modulo run-time overheads. We implemented a library supporting the parallelization of loops using the R-LRPD test. We first present the R-LRPD test and some implementation issues followed by the experimental results on our program suite.

IMPLEMENTATION

First the array involving cross-iteration dependencies is identified and its private copies are made in every processor. All the iterations are then speculatively executed in parallel during which the shadow arrays are used to track the access pattern of the array. The library method mark_write() marks for every array write, the memory location within the shadow array indexed by the processor number and the array reference. Similarly, the method mark_read() marks the memory location within the shadow array only if the array index referenced had not been earlier written to by the same processor. If the reference pattern in a processor for any array index is of the form (Read* | (Write | Read)*), it may result in a conflict if the same index had been written to in any previous iteration. This condition is checked in the method analyze() and the earliest dependence violation, caused by speculation, is recorded. The modifications caused by all the iterations before this reported dependence are committed and the shadow arrays and other data structures are re-initialized. This process is then recursively carried out for the remaining iterations. The R-LRPD test thus breaks a partially parallel loop into a number of fully parallel loops which are executed one after another. Fig 1 indicates a test case and Fig. 2 represents the parallel code with method calls to the functions implemented in the library.
EXPERIMENTAL RESULTS

The speed up reported by a test program depended on the way iterations were distributed on the processors. We tested our programs for two simple strategies – Redistribution (RD) and Non-Redistribution (NRD). In the Non-Redistribution strategy, after analyzing the conflicts, the remaining iterations are scheduled only on the processors having wrong data. The processors whose iterations were committed are left idle for this time. Since the same iterations are scheduled on a processor at every step, the programs instrumented with this strategy show a better time locality. On the other hand, in the Redistribution strategy, we redistribute the remaining iterations across all the available processors. Though in loops with a high level of parallelism the RD strategy might be faster, in most of the cases it uncovered new dependencies which were satisfied before by executing on the same processor.

Our test case consisted of a loop with 1000 iterations with randomly generated read and write array references. The speed up was measured for two machines: Intel(R) Core(TM)2 CPU @ 2.13GHz and 2MB cache, and an 8-core Intel Xeon CPU @ 3GHz and 2MB cache. A comparison between the performances of the RD and the NRD strategies (Fig. 3 and Fig. 4) suggests that for partially parallel loops, Non-Redistribution generally speeds up the program better as compared to Redistribution. For the case of the 2 core machine, the test cases 1,2,3,4 and 5 have 0, 1, 4, 6 and 9 conflicts resp. and in the case of the 8 core machine, the number of conflicts is 0, 6, 10, 16 and 40 resp. The graphs in Fig. 5 and Fig. 6 indicate the performance of the parallel code on the two machines. The performance depends on the degree of parallelism in a loop which is quantified by the number of recursions required in the R-LRPD test. As the number of cores increases, the chunk size per processor decreases. This results in a greater number of conflicts indicated by a slowdown of the parallel code on Intel Xeon for the worst case. On the dual-core machine however, the worst case is almost as fast as the sequential execution of the loop, which reaffirms the superiority of this test over other run-time parallelization methods.
Figure 3 Speed ups for various test cases on Intel Core 2 Processor

Figure 4 Speed ups for various test cases on Intel Xeon Processor (8 Cores)

Figure 5 Speed ups for varying degree of parallelism on Intel Core 2 Processor

Figure 6 Speed ups for varying degree of parallelism on Intel Xeon Processor (8 Cores)