Automatic Iteration/Data Partitioning for Distributed Shared Memory Systems

M.A. Navarro
R. Asenjo
E.L. Zapata

June 2000
Technical Report No: UMA-DAC-00/11

Published in:

NATO Advanced Research Workshop on High Performance Computing: Technology and Applications
Cetraro, Italy, June 12-15, 2000

University of Malaga
Department of Computer Architecture
C. Tecnologico • PO Box 4114 • E-29080 Malaga • Spain

Automatic Iteration/Data Partitioning for Distributed Shared Memory Systems

A. Navarro, R. Asenjo and E. Zapata
Dept. of Computer Architecture
University of Málaga, Spain
{angeles,asenjo,ezapata}@ac.uma.es

Abstract

Current parallelizing compilers have reached the maturity in the detection of parallelism for regular codes. However, parallel code generation leaves room for significant improvement. For NUMA machines, where the latency is much bigger when accessing remote memories, parallel code generation should exploit the locality of memory references. This means to select the suitable iteration and data distributions which minimize the communication overhead. This is not the current direction on automatic code generation for Distributed-Shared Memory (DSM) Machines, but we believe that data distribution is still the most influence issue in the efficiency of the parallel code. In order to prove this affirmation, we have conducted a set of experiments, using real codes, where we have explored the limits of the state-of-the-art on automatic parallelization and code generation for DSM machines. We have analyzed the parallel code generation in two parallelizers: PFA, where the target machine was the Origin 2000, and Polaris, where the target system was the Cray T3D. We have compared the efficiencies of the parallel codes generated by each parallelizer with the codes generated by hand, in which we have parallelized the same loops, but we have applied explicit data distribution techniques. The PFA codes for the Origin 2000 are specially inefficient for several reasons, but mainly because the heuristics to migrate pages are very slow for codes with dynamic access patterns, as happens in real codes. When we use the page distribution directives of PFA, the efficiencies of the resultant parallel codes are not much better because the granularity of a page (4Kb) is not suitable for codes that require a cyclic or a block-cyclic data distribution, or codes with halo. On the contrary, with the hand parallelized codes where we have applied data distribution techniques we achieve efficiencies above 60% for 32 processors in the Origin 2000. On the other hand, the Polaris codes for the Cray T3D are more scalable than PFA codes, because Polaris uses the BLOCK data distribution and privatization techniques. However, this is not enough to get efficient parallel codes, because Polaris partially exploits locality and it does not worry about the number of communications. In fact, in the hand codes where we have selected the data distributions that minimize communications, we improve the Polaris efficiencies above 50% for 32 or more processors in the Cray T3D. The experimental results that we have just mentioned and that we will report in detail in this paper, have proved that parallelizing compilers need to implement explicit iteration and data distribution techniques such as we have applied by hand. We believe that a parallelizing compiler can effectively handle the task of finding the iteration/data distributions that minimize communications while balance the computational load as well as generating the parallel code that implements them. In order to generate the same parallel codes that we have obtained by hand, but with the least effort, we have developed an Automatic Iteration/Data Partitioning (AIDP) method.

The second part of the paper is devoted to introducing our AIDP method. In our method we use a notation that allow us to handle general affine and non-affine access functions. We are able to analyze programs with control flow statements such as conditional and iterative statements. We present a locality analysis algorithm that captures in a graph called the *Locality-Communication Graph* or LCG, when is possible to exploit the locality (without communications) of memory references for an array in two nestings, and when it is not possible. In other words, what are the conditions that iterations and data distributions must fulfill for the two nestings, in such a way that all array references are satisfied in the corresponding processor memory. If it is not possible, our method can identify the communication patterns. The compiler use this information to formulate a non-lineal integer programming problem, where the objective function is the minimization of the overhead due to communications and load unbalance. The solution of the problem allow us to select the CYCLIC(k) iteration distribution of each nesting in the code, and to build a generalized bidimensional block-cyclic data distribution for each array in the code. Finally, we will show how to generate the communications, using the `put` primitive and how to generate the parallel code that implement the iteration/data distributions that we have found with our method. The resulting iteration/data distributions obtained by our method are the same that we were using in our hand parallelized codes. That way we achieve the same excelent performances but now, automatically.