A Tool to Display Array Access Patterns in OpenMP Programs

Oscar R. Hernandez[1], Chunhua Liao[1], and Barbara M. Chapman[1]

Computer Science Department
University of Houston
4800 Calhoun Rd.
Houston, TX 77204-3010
{oscar, liaoch, chapman}@cs.uh.edu

Abstract. A program analysis tool can play an important role in helping users understand and improve OpenMP codes. Array privatization is one of the most effective ways to improve the performance and scalability of OpenMP programs. In this paper we present an extension to the Open64 compiler and the Dragon tool, a program analysis tool built on top of this compiler, to enable them to collect and represent information on the manner in which threads access the elements of shared arrays at run time. This information can be useful to the programmer for restructuring their code to maximize data locality, reducing false sharing, identifying program errors (as a result of unintended true sharing) or accomplishing aggressive privatization.

1 Introduction

OpenMP is a de facto standard for shared memory programming that can be used to program SMPs and distributed shared memory systems. OpenMP follows the wellknown fork-join model, a parallel execution model where teams of threads are created when a program enters a parallel region [16] and terminated upon its completion. Studies [15] have shown that good array privatizations techniques are one of the most effective ways to improve OpenMP performance and scalability. The reason for this is that private data is local to a thread, and systematic privatization will minimize data sharing among threads and thus the cost of cache coherency mechanisms and the long latency incurred when fetching data updated elsewhere in the system. Achieving good privatization in OpenMP might require extensive rearrangement of code, particularly when this is applied to the entire application, as is required by the OpenMP SPMD style [8][9]. Unfortunately this is not an easy task. Prior to an effort of this kind, it is crucial to know how the arrays are being accessed by the executing threads, and determine which regions of arrays are shared between multiple threads. A number of tools have been developed that analyze memory references; however, none of them summarize accesses to OpenMP shared data and display that information to help the user privatize or otherwise study and potentially reorganize memory accesses.

In this paper we present the enhancements made to the Open64 compiler [11] and the Dragon tool [5], a program analysis tool built on top of this compiler, to enable them to collect and represent information on the manner in which threads access the elements of shared arrays at run time. We also describe how we have accomplished this task. In the next section, we briefly describe the current functionality of Dragon and give an overview of Dragon and those analysis modules that we use for the implementation of the tool. Then in Section 3 we focus on the problems of determining and representing array sections in interprocedural code regions. After that we explain how OpenMP is lowered in our version of the Open64 compiler, and how this affects the region calculation and our instrumentation strategy. As part of this section, we provide a simple case study that illustrates our extension to the Dragon tool. Finally, we discuss related work, and present some conclusions and future work.

2 Dragon and Open64

The Dragon analysis tool is a production tool for developing, understanding, and maintaining large scale sequential and parallel programs. It is built on top of our enhanced version of Open64 compiler [11] to displays program analysis information in a intuitive way to application developers who need to understand a given source code in depth. The Open64 compiler was originally developed by Silicon Graphics Inc. and is currently maintained by Intel. It is an optimizing compiler suite for Linux/Intel IA-64 systems. Our version of Open64, Open64.UH, has refined interprocedural analsis module and accepts special flags to extract various program information used in Dragon. The input languages for Dragon are FORTRAN 77/90, C, and OpenMP/MPI. Current supported program analsis information includes static/dynamic call graph, flow graph, data dependence results, automatic performance and feedback instrumentation.

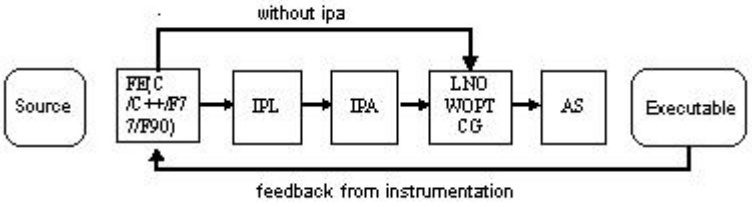

Fig. 1. The modules of Open64 compiler. The user can select interprocedural analysis using the (-ipa flag) or go directly to the backend, where the loop nest and global optimizer reside. The user has the option to automatically instrument a program and obtain feedback files for one or more runs; this information is used for further optimizations.

The Open64 intermediate representation, called WHIRL, has five different levels, starting with very high level (VHL) WHIRL, and serves as the common interface among all its components. Each optimization phase is designed to work at a specific level of WHIRL. Our extensions to Dragon use information primarily from the VHL and High Level (HL) WHIRL phases, which preserve high level control flow constructs, such as loops and arrays, as well as OpenMP directives, which are preserved as compiler pragmas. It is very important for our purposes to work in the high level representation because the arrays are still represented explicitly as high level constructs (before they become addresses/pointers), allowing the compiler to perform high level array region analysis calculations. The Open64 compiler consists of five modules as shown in Fig. 1. Multiple frontends (FE) parse C/Fortran programs and translate them into VHL WHIRL. If interprocedural analysis is invoked, then IPL (the local part of interprocedural analysis) first gathers data flow analysis information from each procedure, and then summarizes and saves it in files. Array accesses are summarized into array regions for each individual procedure. Then, the main IPA module generates the call graph and performs interprocedural analysis and transformations based on the call graph data structure. In this phase, the array regions are propagated to determine how the arrays are being accessed across the entire program. If the program is instrumented, the inteprocedural analyzer is able to exploit runtime feedback to optimize the callgraph, in particular to decide when to clone procedures with constant parameters, and for inlining purposes. Our extensions to Open64 were mainly concentrated in the interprocedural analyzer module (IPA), where we retrieve static array regions calculations and in the instrumentation libraries, which we have used to instrument the program at the control flow level using array region information. We will explain this instrumentation process in the following sections and will also show how we have used the analysis within the compiler. The backend module consists of the loop nest optimizer (LNO), the medium-level code optimizer (WOPT), and the code generator (CG). We do not discuss these further here.

3 Array Regions and Interprocedural Analysis

Most of the data processed by scientific programs are stored in arrays. Therefore a good method to represent and manipulate array access information is key to the success of interprocedural data flow analysis. The classical analysis methods [18, 19] treat an array as a whole and only use two bits (one for DEF and the other for USE) to represent the accesses to an array. They are very efficient in terms of storage space and computation complexity. However, they are too coarse for many optimizations. Several more accurate methods have been proposed to address this problem. They are largely categorized into exact and summary methods. Exact methods, also called reference-list-based, maintain information about each reference to the elements of an array. Linearization [4] and Atom Images [7] are two implementations of this idea. They are precise, but are very expensive in terms of representation size and operations on them. On the

other hand, summary methods require storage space which is independent of the size of the access sets. Well-known strategies for summarizing array regions are Regular Sections [21], Bounded Regular Sections [20] using triplet notations, and Linear-constraint methods such as Data Access Descriptors [22] and Regions [23]. They use geometrical spaces containing accessed array elements to approximate and represent access information rather than storing individual references. Each summary method has its benefits and drawbacks in terms of complexity and accuracy. Flexible shapes usually mean high precision but also complex computation. For example, Triolet's Regions method is fairly accurate because it applies a set of linear constraints to express a convex region. However, the standard operations (union and intersection) to form and compare such regions are very time-consuming.

Open64 adopts Triolet's Regions to perform its array access analysis at both intraprocedural (LNO) and interprocedural levels (IPA). The IPA array regions are summarized and propagated from local region information generated in LNO for global and formal parameter arrays. An IPA level region is created for each type of access mode (USE, DEF, FORMAL (formal parameter) or PASSED(actual parameter)) for each array variable at procedural level, which in turn includes detailed region information at each array access site in form of Projected Regions. A Projected Region consists of a set of triplet notations [LowerBound, UpperBound, Stride](Projected Nodes in Open64) for each dimension. System of linear equations are formed from constraints for each axis of an array using $lower \leq axis \leq upper$. To propagate array region information, there is a mapping between caller and callee. Open64 first performs a reshape analysis and checks if there are aliases and global variables involved. It then propagates information about formal parameters used as symbolic terms in array region summaries, which later will be used to trigger cloning, if possible. A region could be MESSY or UNPROJECTED if it encounters non-linear access or unknown values of bounds.

Figure 2 gives an example of the array region analysis in Open64 using a simple Jacobi code. In this subroutine, *uold* is a local array and u is a formal parameter array. The DEF region of *uold, (1:n:1,1:m:1)* comes from only one definition site while its USE region *(1:n:1,1:m:1)* is the result of summarizing 6 usage sites. The boundary constraints are used to form system of equations for each region. For example, *u(2:n-1:1,2:m-1:1)DEF* corresponds to two linear equations: $2 \leq dim1 \leq n-1$ and $2 \leq dim2 \leq m-1$.

4 Extensions to Dragon and Open64: OpenMP Early Lowering, Array Regions Calculations and Instrumentation

Our extensions to Dragon to provide this new functionality uses the static symbolic analysis and array data flow analysis within the compiler to instrument the program strategically at the control flow level using array region information. Our goal is to calculate the how threads access the array regions in an

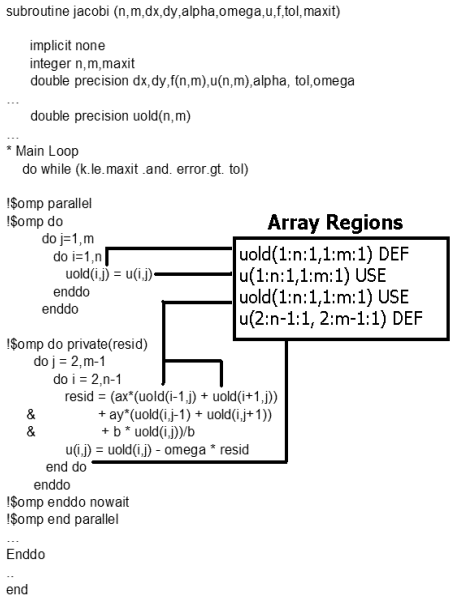

Fig. 2. Open64 calculation of the regions in a Jacobi OpenMP program. Note that there is one region for each access mode per array for a subroutine and it is given in triplet notation.

OpenMP code. To do so requires us to take into consideration how OpenMP is lowered in Open64, since many of the arrays change in scope and definitions (from local variables to global/interprocedural variables, in the case of shared variables). This is the result of the compilation phase where OpenMP directives are transformed into multi-threaded code. It typically involves outlining parallel regions by turning them into procedures containing the code in the region, making shared data available by turning them into global variables or passing them as arguments to the parallel region procedures, redefining private data as local variables to parallel region procedures, and inserting new loop bounds with scheduling calls for parallel loops. [17] discusses a similar strategy for lowering OpenMP in a source to source compiler.

Our extensions to Open64 force the compiler to lower OpenMP directives before the Array region analysis take place. This allow us to accurately calculate the array regions accessed on a thread by thread basis. Since shared data becomes global in Open64's OpenMP lowering process, the array regions accessed are propagated across the procedure boundaries, allowing us to determine how shared arrays are accessed in the entire program. This information helps us to figure out what information we need to gather at runtime and to instantiate any symbolic expressions that describe array Regions. Information such as un-

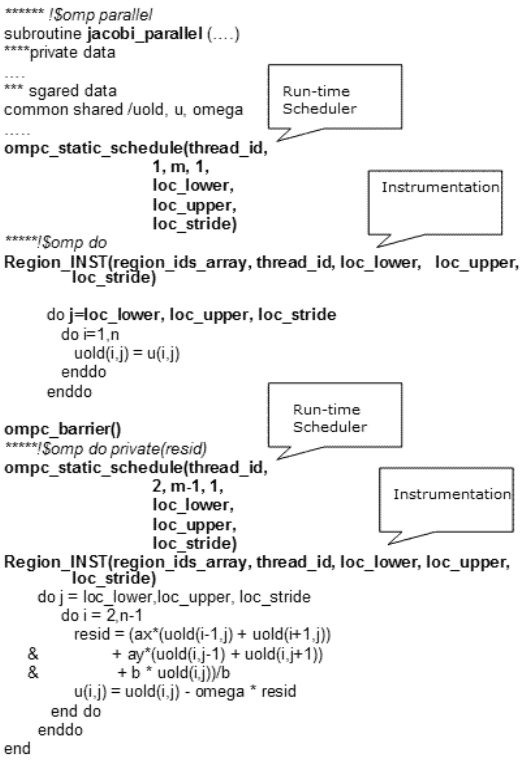

Fig. 3. Open64 lowering of a Jacobi OpenMP program to explicit multithreading code. Note how the parallel region is translated to a procedure, shared variables become global variables, and the loop bounds of the parallel loops are assigned by a run-time scheduler.

known loop bounds etc. In Figure 3 we show how the Jacobi program looks after OpenMP lowering.

Figure 3 shows how an OpenMP program is lowered. It also illustrates how a parallel region is translated to a procedure that every thread participating in the computation of the parallel region will execute. The run-time schedule assigns temporary upper and lower bounds and a stride to each thread. Since the Jacobi program uses a static scheduling scheme, each thread will get, as far as possible, an even amount of work. This means that the iteration space is divided equally among the threads. Figure 4 shows how the lowering of OpenMP affects the regions of the program in comparison to the original ones in Figure 2. The loop boundaries of the parallel loops are assigned to the threads at run-time.

Now the problem resides in determining the array regions accesses on a per-thread basis. For this purpose, we instrumentthe parallel loops nests that have access to shared arrays. Figure 3 also shows how instrumentation calls are in-

serted in the lowered version of the Jacobi program. The arguments for the RegionInstrument calls include the thread ID, procedure ID, basic block ID, array region ID, and a list of terms/variables needed, to determine the array region boundaries.

Figure 5 shows the results after running the instrumented Jacobi program using four threads. In this code, you can see which portion of the array *UOLD* is being accessed by each of the four threads. The overlap areas among the regions indicate where the threads share data. This information can help the program-

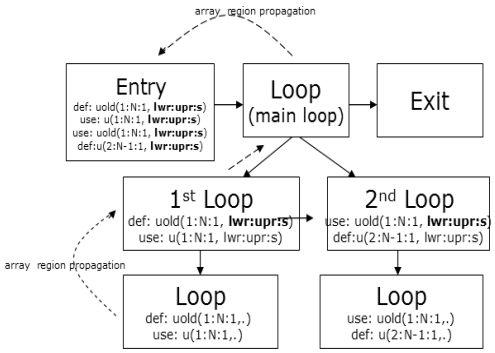

Fig. 4. The array regions of the OpenMP-lowered version of Jacobi. Note the difference between the boundaries of the regions here and the original ones.

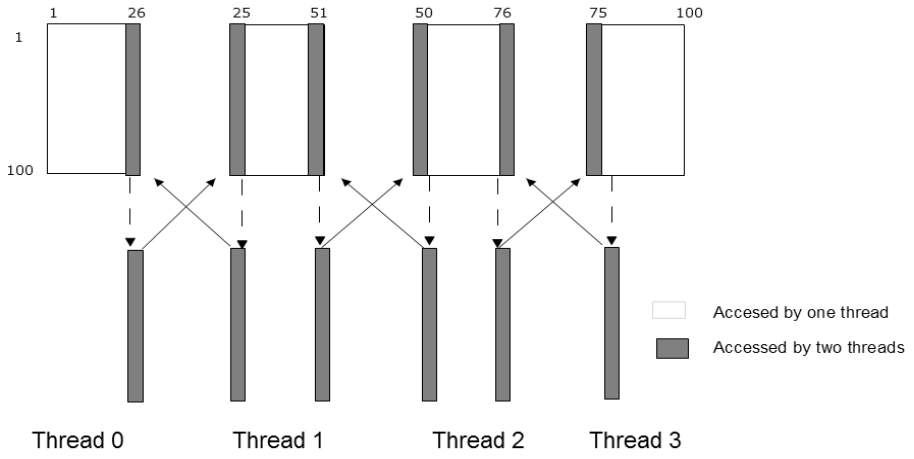

Fig. 5. The different portions of the array *UOLD* accessed by the different threads. The overlap area of the regions is the subregion where the threads share data. The array portions of the regions that are accessed by one thread only can safely be privatized

mer decide how to partition shared arrays to maximize privatization. [15], [9] presents an example illustrating how this can be accomplished and describes their experimental results.

5 Related work

A variety of tools provide detailed information on the execution behavior of a sequential or parallel program, but with different goals and approaches. VTune [6] and PAPI [2] retrieve hardware counter information to poll or take samples to find out how the application is executing, focusing on memory issues such as cache misses. The information retrieved is low level and cannot be used to give a higher-level view of how OpenMP shared arrays are accessed. On the other hand, tools such as MemSpy [10], which map memory accesses to source code and try to understand how to adapt an application to the current memory subsystem, do not summarize array accesses and present them in an intuitive way to the programmer. Intel's ThreadChecker [6] focuses on semantic problems in OpenMP to detect data races, uninitialized private data, and more, but does not emphasize shared data accesses. MetaSim [14] and Dyninst [3] provide a framework to extract the memory signature of programs and try to match the application to the best hardware profile available, but they rely on simulations and explore other problems. Our approach is based on combined static and dynamic analysis and instrumentation to retrieve runtime information for summarizing and displaying the array regions accessed at the thread level.

6 Conclusions and Future Work

Our extensions to the Dragon tool allow us to construct a mapping between OpenMP threads and the regions of arrays that they access. This is accomplished by recalculating the array regions within the compiler after the OpenMP lowering has been performed, and using the region analysis within Open64 to help strategically instrument the application. During execution, the regions can be made precise. There are a variety of potential uses of this information. Our Dragon tool and Open64 compiler version are robust systems that can handle large-scale programs. Further work is needed to evaluate this new feature of the tool on production applications.

References

1. V. Balasundaram and K. Kennedy. A technique for summarizing data access and its use in parallelism enhancing transformations. *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, 41–53, 1989.
2. S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A Scalable Cross-Platform Infrastructure for Application Performance Tuning Using Hardware Counters. *Proc. Supercomputing 2000* , November 2000, Dallas TX.

3. B. Buck and J.K. Hollingsworth. An API for Runtime Code Patching. *Journal of Supercomputing Applications*, 14(4):317–329, 2000.
4. Michael Burke and Ron Cytron. Interprocedural dependence analysis and parallelization. *Proceedings of the 1986 SIGPLAN symposium on Compiler contruction*, 162–175, 1986.
5. Barbara Chapman, Oscar Hernandez, Lei Huang, Tien-hsiung Weng, Zhenying Liu, Laksono Adhianto, Yi Wen. Dragon: An Open64-Based Interactive Program Analysis Tool for Large Applications. *Proceedings of the 4th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT 03)*, 2003.
6. Intel Corporation products for OpenMP. Intel ThreadChecker and VTUNE http://developer.intel.software/products/
7. Zhiyuan Li and Pen-Chung Yew. Efficient interprocedural analysis for program parallelization and restructuring. *Efficient and precise array access analysis*, 24(1):65–109, 2002.
8. Zhenying Liu, Barbara Chapman, Yi Wen, Lei Huang, Tien-hsiung Weng, Oscar Hernandez. Improving the Performance of OpenMP by Array Privatization. *WOMPAT2002, Workshop on OpenMP Applications and Tools*, 224-259, 2002.
9. Zhenying Liu, Barbara Chapman, Yi Wen, Lei Huang, Tien-hsiung Weng, Oscar Hernandez. Analyses for the Translation of OpenMP Codes into SPMD Style with Array Privatization. *WOMPAT2003, Workshop on OpenMP Applications and Tools*, 26-41, 2003.
10. M. Martonosi, A. Gupta, T. Anderson. MemSpy: Analyzing Memory System Bottlenecks in Programs. *SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1–12, 1992 Newport, Rhode Island.
11. The Open64 compiler Website http://open64.sourceforge.net
12. William Pugh. A practical algorithm for exact array dependence analysis. *Commun. ACM*, 35(8):102–114, 1992.
13. Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures, A Dependence-Based approach.* 585–588, Academic Press, 2002
14. A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia and A. Purkayastha. A Framework for Application Performance Modeling and Prediction. *Proceedings of Supercomputing 2002*, November 2002, Baltimore.
15. B. Chapman. F. Bregier, A. Patil and A. Prabhakar. *Achieving Performance under OpenMP on ccNUMA and Software Distributed Shared Memory Systems* Special Issue of Concurrency Practice and Experience, Volume 14, Issue 8-9, September 2001.
16. OpenMP Architecture Review Board. *Official OpenMP Specifications* http://www.openmp.org.
17. M. Soukup. *A source-to-source OpenMP compiler* Master's thesis, University of Toronto, Toronoto, Ontario 2001.
18. J. Barth An interprocedural data flow analysis algorithm *in Conference Record of the Fourth ACM Symposium on the Principles of Programming Languages*,(Los Angeles), Jan. 1977.
19. Keith D. Cooper and Ken Kennedy Efficient computation of flow insensitive interprocedural summary information SIGPLAN Symposium on Compiler Construction 1984: 247-258
20. Paul Havlak and Ken Kennedy An Implementation of Interprocedural Bounded Regular Section Analysis IEEE Transactions on Parallel and Distributed Systems, vol. 2, no. 3, pp. 350-360, 1991.

21. David Callahan and Ken Kennedy Analysis of Interprocedural Side Effects in a Parallel Programming Environment J. Parallel Distrib. Comput. 5(5): 517-550 (1988)
22. Vasanth Balasundaram and Ken Kennedy A Technique for Summarizing Data Access and Its Use in Parallelism Enhancing Transformations PLDI June 1989: 41-53
23. R. Triolet, F. Irigoin and P. Feautrier Direct parallelization of call statements SIGPLAN Symposium on Compiler Construction, July 1986: 176-185