Author's Retrospective for: Array Privatization for Parallel Execution of Loops

Zhiyuan Li
Department of Computer Science
Purdue University
250 North University Street
West Lafayette, Indiana, USA
zhiyuanli@purdue.edu

ABSTRACT

Array privatization, as a program transform to increase the opportunity for loop parallelization, was introduced at a time when numerical programs dominated the parallel computing world. Today, with parallel processors becoming ubiquitous, the computer industry faces a new challenge, i.e. how to best utilize hardware parallelism for the next generation of main-stream applications. The author re-examines the technique of array privatization both in its historical context and in view of new developments in recent years.

Original paper:
http://doi.acm.org/10.1145/143369.143426

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors – *Compilers, run-time environments;* F.3.2 [**Logics And Meanings of Programs**]: Semantics of Programming Languages – *Program Analysis*

Keywords: Program parallelization; loops; memory allocation

1. BACKGROUND

In the fall of 1990, I joined the compiler group of the Center for Supercomputing Research and Development (CSRD) in University of Illinois, Urbana-Champaign (UIUC). Under the leadership of the director, David Kuck, CSRD was designing and building the Cedar multiprocessor cluster [9]. There were groups working on its hardware, the operating system, the compiler, performance analysis tools, numerical libraries and many other issues concerning supercomputing. Many results from the Cedar project were ground-breaking.

The compiler group, led by David Padua, was busy designing and implementing a Fortran dialect called Cedar Fortran [8] that provides syntax for multiple levels of parallelism and data locality. In order to understand how to make it easier to convert existing numerical programs for shared-memory parallel computers, the compiler group also started to hand parallelize a suite of benchmark programs contributed by the PERFECT Club [2]. Continuing UIUC's long tradition of pioneering automatic parallelization techniques, the compiler group wanted to find out what kind of new techniques might be needed to automatically convert the PERFECT benchmark programs into efficient parallel codes.

The initial experience from this hand parallelization effort was presented at International Conference on Parallel Processing in 1991 and a more definitive report was later published in 1993 [6]. These reports identify several new techniques that are instrumental for parallelizing the PERFECT benchmarks, with array privatization being one of the more important.

The basic idea for array privatization is quite simple. If a loop operates on different parts of arrays in different iterations, then these iterations can be executed simultaneously on multiple processors. Often, the intermediate results are stored in arrays that are rewritten from iteration to iteration. Array privatization creates a private copy of such arrays for each processor, such that writes from iterations executing in parallel do not cause conflicts. This transformation is especially important because arrays are the most common data structures in numerical programs to represent physical quantities.

To anyone who is not the author of the original program, determining which arrays can be safely privatized can be a tedious task. One must determine whether the lifetime of an array (or a part of it) may extend across the iterations of the targeted loop. If so, then privatizing that array will cause an incorrect computation result. Moreover, it is necessary to determine whether the values written to a privatized array in the last loop iteration remain live after the parallel loop exits. If so, these values must be copied back to the original shared array to make them accessible to all processors. It is safe to copy the entire array out, but unnecessary copying can incur high overhead. The program scope to be analyzed for array privatization can be extensive, making the analysis both laborious and error-prone. Obviously, there is a strong incentive to automate the analysis.

Prior to the hand parallelization experiment by the CSRD compiler group, the IBM Ptran project team published a framework for automatically converting a sequential program into parallel code using the fork/join shared-memory model [1]. Among the ideas presented within that framework was privatizing scalar variables to remove data dependencies caused by storage conflicts [3].

The idea to duplicate variables in order to increase parallelism dates back to the *scalar expansion* technique for auto-

ICS 25th Anniversary Volume. 2014
ACM 978-1-4503-2840-1/14/06.
http://dx.doi.org/10.1145/2591635.2591648.

matic vectorization [10], which removes data dependencies due to iterative read/write operations on a single scalar by letting such operations be performed on elements of a new array instead. Conceptually, data dependencies caused by storage conflicts can be eliminated completely by fully expanding all variables, including arrays, that are responsible for such conflicts. It is unknown how to automatically perform such conversion in the general case, but under a set of ideal conditions automatic *array expansion* is possible.

In practice, however, it is obvious that excessive variable expansion incurs overheads that are unacceptable. Increasing a single array dimension can increase the storage for that array by an order of magnitude. In 1990's, computers simply did not have enough memory to allow full array expansion. Moreover, storage increases tend to make it difficult to efficiently use faster but smaller memory devices, such as caches, which is perhaps the bigger issue today. The technique of privatization, by making one copy of the variable for each processor, increases loop level parallelism at the minimum storage cost.

Unfortunately, a number of serious challenges stand in the way when one tries to automatically privatize arrays. Some of them are common to all analyses that deal with array references, e.g. how to build a mathematical system that is sufficiently precise to model the analysis problem and yet can be efficiently solved. There is a unique challenge to array privatization, however. Its underlying mathematical system may simultaneously involve all references to the targeted array. This is in contrast to the traditional *data dependence analysis* that answers the question whether array accesses from different iterations interfere with each other. For data dependence analysis, it is sufficient to examine array references one pair at a time, which greatly simplifies both the mathematical system and the information gathering to build the system.

At the highest level, there are two alternative approaches to the construction of the mathematical system for array privatization, which we now discuss.

2. THE TOP-LEVEL DESIGN REEXAMINED

The paper revisited here takes an approach that can be viewed as *write-centric*, and it corresponds to a forward direction in which the mathematical system is built based on program analysis and is solved incrementally. An opposite approach is *read-centric*, corresponding to backward propagation. In a later experiment to actually build a compiler prototype, my student, Junjie Gu, and I took the latter approach [7].

The write-centric approach computes the *cover set* for the array targeted for privatization at every program point in the loop. This is the set of array elements that are known to have been definitely written, no matter what execution path is taken when the specific program point is reached. Naturally, this set must be computed in the forward direction, with every write reference examined for the array elements being written. Note that this set will be parameterized by the execution conditions of the current program point, including the index variables of all nested loops (within the targeted loop) containing the program point.

When analyzing a sequence of program statements, the cover set can be expanded (by union operations) on-the-fly. If a read reference is encountered, its upwardly exposed elements can be determined by subtracting the current cover set from the set of array elements that are read. Different cover sets, however, must be computed for different execution paths until they are intersected.

The read-centric approach propagates, in the backward direction, the set of array elements that may be upwardly exposed to the beginning of any arbitrary iteration of the targeted loop. When a write reference is encountered, the elements written are subtracted from the set being propagated. The difference is propagated further. An obvious potential advantage is that, if most of the time the difference is either empty or in a simple shape, then the cost to propagate the upwardly exposed set is low. In comparison, the write-centric approach must propagate all elements that are written in the targeted loop, which may consume much more storage and time. However, the read-centric approach has a potential down side: if most of the time the set subtraction results in a non-convex set, we will need to decompose the result into several convex sets, because the solvers of the mathematical systems normally require the sets to be convex. Hence, we may end up with propagating a long list of fragmented convex sets. In the experiment mentioned above [7], however, a set representation is devised to allow lazy computation of set differences. A set subtraction is registered but not actually performed until it cannot be postponed any further. The experiment shows that for the PERFECT benchmarks the subtraction operations can often be avoided this way.

3. OTHER DEVELOPMENTS

Numerous efforts have been undertaken by researchers to make array access analyses efficient. Progress has been made on several fronts, some of which are recounted below, as they have direct impact on the quality and efficiency of automatic array privatization.

Set representation.

The set of array elements needs to be represented in a way that allows set operations to be performed efficiently. For array privatization, one can borrow methods originally proposed for interprocedural data dependence analysis. Often, the set of elements accessed by a single array reference forms a *regular array section* and can be denoted by a triplet *(lower bound, upper bound, stride)* [12, 4]. During automatic array privatization, both the cover sets for write references and the sets of upwardly exposed elements can be represented as a list of regular array sections. It is often beneficial to maintain a list of postponed set subtraction operations, as mentioned above, instead of eagerly performing the subtraction and storing the fragmented result as a list of regular array sections.

Solving the underlying mathematical systems.

The problem of determining whether a set of written array elements overlaps a set of upwardly exposed elements can usually be reduced to the problem of determining the feasibility of a set of integer linear inequalities. Such linear inequalities contain loop index variables and possibly other symbols appearing in array subscripts and loop limits. This is an integer programming problem and in general can be very time consuming to solve. Numerous approximation methods have been proposed but are not discussed here. It is worth pointing out that a more comprehensive mathemat-

ical model that incorporates execution path conditions may be needed, in practice, for some cases. Today, logical system solvers are mature enough to solve such more complicated systems.

Finding relationship between different symbols.

A more challenging issue than solving a logical system concerns finding sufficient information to build a useful system. In general, the more constraints we can extract from the program the more precise the model is. If two different symbols appear in the system but we know nothing about their relationship, then the system tends to be a poor approximation of the problem and its solution may be useless. For this reason, we ideally want to find a closed-form expression for every symbol such that the expression contains no symbols except for the program input variables. This way, every useful relationship governing the symbols will be represented in the mathematical system. Unfortunately, in practice we are rarely able to obtain such an ideal representation, which means that there will be "unknown symbols" present in the system. We wish to be able to find useful constraints on such symbols via symbolic substitution. The demand-driven substitution method proposed by Tu and Padua synchronizes the substitution steps for two symbols of interest [13]. This method is found to be especially effective for the purpose of automatic array privatization.

Dynamic methods.

For programs that are difficult to analyze at compile time, dynamic privatization methods have been proposed for arrays [11, 5].

Researchers looking into ways to parallelize non-numerical programs have discovered that non-array aggregate variables, such as dynamically allocated irregular data structures, may also be privatized to improve loop level parallelism. Due to the space limit, these new developments are not discussed here.

4. LOOKING FORWARD

Automatic array privatization is motivated mainly by the desire to automatically parallelize programs. Due to the well-known difficulties with parallelizing legacy programs, whether automatic parallelization has a future has constantly been questioned. One has to ask, however, what the effective alternatives are. Based on the difficulties experienced by many when teaching or learning parallel programming, the question seems to be unanswered. It is important to find an effective way to retain the determinism of computational results offered by sequential programming languages and yet, at the same time, to remove the complications that can obscure the inherent parallelism in a computation problem.

5. REFERENCES

[1] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante. An overview of the PTRAN analysis system for multiprocessing. *The Journal of Parallel and Distributed Computing*, 5(5):617 – 640, October 1988.

[2] M. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, L. Pointer, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orszag, F. Seidl, O. Johnson, and R. Goodrum. The PERFECT Club benchmarks: Effective performance evaluation of supercomputers. *International Journal of Supercomputer Applications*, 3:5–40, 1988.

[3] M. Burke, R. Cytron, J. Ferrante, and W. Hsieh. Automatic generation of nested, fork-join parallelism. *Journal of Supercomputing*, 3(2):71 – 88, 1989.

[4] D. Callahan and K. Kennedy. Analysis of interprocedural side effects in a parallel programming environment. *Journal of Parallel and Distributed Computing*, 5(5):517–550, 1988.

[5] F. Dang, H. Yu, and L. Rauchwerger. The R-LRPD test: Speculative parallelization of partially parallel loops. In *Proceedings of the 16th International Symposium on Parallel and Distributed Processing*, IPDPS '02, pages 20–, Washington, DC, USA, 2002. IEEE Computer Society.

[6] R. Eigenmann, J. Hoeflinger, G. Jaxon, Z. Li, and D. A. Padua. Restructuring Fortran programs for Cedar. *Concurrency - Practice and Experience*, 5(7):553–573, 1993. An early version is in *Proceedings of the 1991 International Conference on Parallel Processing*, pages 57–66, 1991.

[7] J. Gu, Z. Li, and G. Lee. Experience with efficient array data flow analysis for array privatization. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '97, pages 157–167, New York, NY, USA, 1997. ACM.

[8] M. D. Guzzi, J. P. Hoeflinger, D. A. Padua, and D. H. Lawrie. Cedar Fortran and other vector and parallel Fortran dialects. In *Proceedings of the 1988 ACM/IEEE Conference on Supercomputing*, Supercomputing '88, pages 114–121, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[9] D. J. Kuck, E. S. Davidson, D. H. Lawrie, and A. H. Sameh. Parallel supercomputing today and the Cedar approach. *Science*, 231(4741):967 – 974, 1986.

[10] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '81, pages 207–218, New York, NY, USA, 1981. ACM.

[11] L. Rauchwerger and D. Padua. The privatizing DOALL test: a run-time technique for DOALL loop identification and array privatization. In *Proceedings of the 8th international conference on Supercomputing*, ICS '94, pages 33–43, New York, NY, USA, 1994. ACM.

[12] P. F. Rémi Triolet and F. Irigoin. Direct parallelization of call statements. *Proceedings of the ACM Symposium on Compiler Construction*, 1986.

[13] P. Tu and D. Padua. Gated SSA-based demand-driven symbolic analysis for parallelizing compilers. In *Proceedings of the 9th International Conference on Supercomputing*, ICS '95, pages 414–423, New York, NY, USA, 1995. ACM.