Automatic Data Layout With Read–Only Replication and Memory Constraints *

Ulrich Kremer

Department of Computer Science
Rutgers University
email: uli@cs.rutgers.edu

Abstract. The memory requirement characteristics of a data layout are particularly important for applications that are executed on a parallel machine mainly because of the amount of main memory that the machine provides, rather than its computation power. It may not be feasible to execute such a memory intensive program on a conventional uniprocessor due to the lack of the necessary memory resources.
Data layouts that specify arrays with multiple read–only copies — each copy with a different data mapping — can significantly reduce the overall execution time of a program since otherwise necessary communication is avoided. However, read–only replication increases a program's memory requirements and therefore should only be applied selectively, in particular for memory intensive applications. This short paper discusses an extension to our previous framework for automatic data layout that considers read–only data replication and minimizes the overall execution time under given memory constraints.

1 Introduction

Multiprocessor architectures provide not only computing cycles, but also large amounts of main memory. Using a parallel machine instead of a uniprocessor is often the only choice for running memory intensive applications. In fact, some researchers argue that machines with a large main memory should always have multiple processors in order to make cost–effective use of the memory's capacity and bandwidth [WH95].

Many scientific applications contain array variables that are much less frequently modified than they are referenced (*glacial variables* [AW96]). A *read–only region* of an array variable is a program region where the array is not modified and all references to the array within the region refer to the same array value. Read–only copies of a glacial array within its read–only regions can reduce program execution times significantly if the array is needed with different mappings within the read–only regions.

Figure 1 shows the benefits and tradeoffs of read–only replication for an Alternating Directions Implicit (Adi) kernel [Kre96]. Three data layouts were measured: *static column–wise*, *dynamic*, i.e., with transpose operations between dimensional sweeps, and *dynamic with replication*. The data layouts were specified as HPF directives. For the dynamic layouts, explicit transpose operations were inserted in the code. The programs were compiled using IBM's HPF compiler

* This research was supported by DARPA contract DABT 63-93-C-0064 and used computing resources at the Cornell Theory Center.

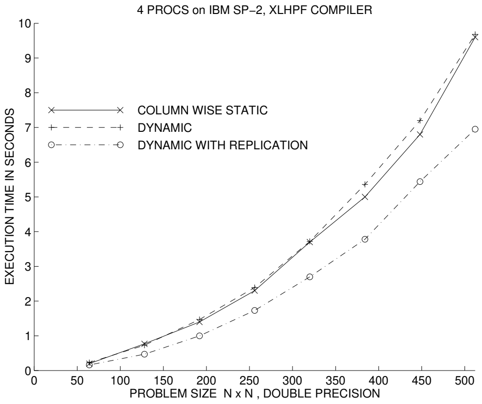

Fig.1: Adi example

	std	**std&rep**	**std&rep&constr**	
			non-res.	res.
sizes of 0–1 problem instances				
#l–vars	40	40	40	
#r–vars	250	870	870	
#constrs	108	278	285	
solution times in milliseconds				
best	10	50	50	40
worst	20	110	100	60
average	13	68	75	50

Fig.2: 0–1 formulations

(xlhpf -O3) and executed on four IBM SP–2 thin nodes. Replication led to a 33% increase in memory usage over the other layouts that kept only a single copy of each array at any given point during program execution. The dynamic data layout with replication is the best for all problem sizes. In the cases where read–only replication is not possible due to the additional memory usage, the static data layout should be selected. For our example, all measured problem sizes fit into memory.

2 Framework for Automatic Data Layout

The *data layout graph* (DLG) is a central program representation of our previous framework. A node in the DLG represents a candidate data layout in the data layout search space of a phase. A phase identifies operations on arrays between which remappings may be profitable. DLG edges represent possible remappings between candidate layouts. Nodes and edges are weighted with their estimated execution times. A solution to the *data layout selection problem* picks exactly one candidate data layout for each phase such that the overall cost of the resulting path is minimal. Finding an optimal solution has been shown to be NP–complete [Kre95].

The previous DLG construction algorithm was based on the assumption that only a single copy of an array can exist at any time during program execution. Therefore, two references to the same array in different phases generate remapping edges in the DLG if there is an execution path from the first reference to the second that does not contain any other reference to the array. In other words, the candidate layout that specifies a data mapping for the second reference *inherits* the candidate layout from the first reference on some execution path. For any given execution path, this inheritance relation is unique. This property does not hold any more in the presense of read–only replication. Inside a read–only region for an array, a candidate layout may be inherited from any reference to the array on an execution path. As a consequence, the new framework inserts additional DLG edges for read–only arrays and their read–only regions.

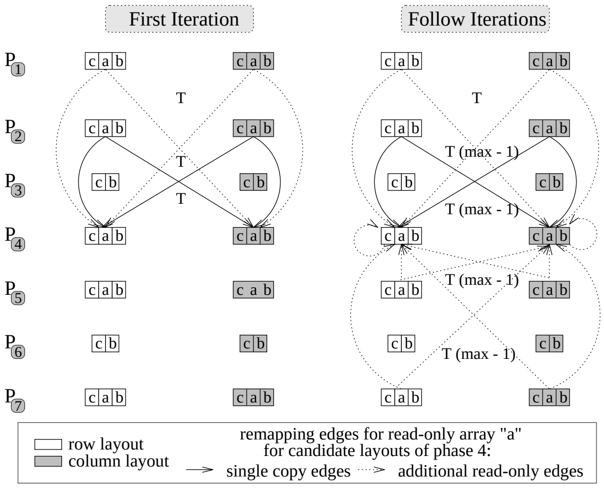

Fig.3: DLG edges for read–only array "a" for phase P_4 in Adi example

In the new framework, all array references are classified as either read–only or not read–only. For array references that are not read–only, the previous problem representation and 0–1 formulation of the data layout selection problem is used. It is important to note that an array may be read–only for only parts of a program. For read–only array references, new remapping constraints have to be introduced to correctly handle the additional read–only DLG edges. The following section discusses the new framework for simple read–only regions.

2.1 Simple Read–Only Regions

A simple read–only region is a subgraph in the *phase control flow graph* (PCFG). The PCFG is an augmented control flow graph where each phase is represented by a single node [Kre95]. A simple read–only region consists of a single phase that defines the value of a read–only variable and a singly nested loop in the PCFG that contains phases referencing the read–only variable. Figure 3 shows partial DLGs for the simple read–only region for array "a" in our Adi example. The region consists of P_1 as the defining phase for array "a" and $P_2, \ldots P_7$ as the body of the loop. To simplify the example, each phase is shown with a search space of two candidate layouts, namely a row and a column layout. Phases are labeled with the names of arrays they reference. Node weights are not shown.

The framework introduces two new sets of edges. The first set covers the case where control enters the loop the first time. The second set represents the situation in subsequent loop iterations. A phase in the loop can inherit a data layout (1) from outside the loop and all phases that reference the read–only array inside the loop before the phase (*first iteration*), and (2) from outside the loop and any phase that references the read–only array inside the loop (*follow iterations*). Figure 3 shows the DLG edges for the two cases for candidate layouts of phase P_4. T is the cost for a single transpose operation and max is the number of loop iterations. Both entities are used to specify edge weights.

The new edges require a reformulation of the previous, single copy 0–1 integer programming formulation of the data layout selection problem. OUT–constraints become inequality constraints. The number of OUT–constraints may increase quadratically in the number of phases in the region. There is no increase in the number of IN–constraints and IN–constraints remain equality constraints. However, their sizes my increase linearly with the number of phases. In addition, new *resource constraints* ensure for each program point that the sum of the sizes of any read–only copies life at the program point does not exceed a predetermined maximum. The formulation of memory constraints assumes that the sizes of all arrays are known at compile or tool invocation time. A more detailed discussion can be found in [Kre96].

2.2 Experiments

To compare the efficiency of the old and new framework, 10 instances of the DLG for our Adi example were generated, where each phase was represented by five possible candidate data layouts. Each problem instance had different node and edge weights. A single array was classified as a read–only variable. In Figure 2, column std refers to our previous 0–1 formulation. Column std&rep is the formulation including the modified remapping constraints for read–only replication. The rightmost column, std&rep&constr includes in addition the resource constraints specifying enough (non-res.) and insufficient memory (res.) to allow replication. The problem instances were solved using *CPLEX*[2], a linear integer programming tool and library, running on a SUN UltraSparc1 (143MHz/64Mbyte). In addition to the solution times, the sizes of the three different 0–1 formulations are given in terms of the number of layout and remapping variables, and number of constraints.

3 Conclusion

In summary, preliminary experiments on a single program kernel showed that the overhead is significant — within a factor of six — as compared to our previous framework which allowed no read–only replication. However, the expected benefits of read–only replication are significant as well. More work will be needed to evaluate the new framework on larger problems and to develop an efficient treatment of complex loops and branch structures.

References

[AW96] T. Autrey and M. Wolfe. Initial results for glacial variable analysis. In *Proceedings of the Nineth Workshop on Languages and Compilers for Parallel Computing*, San Jose, CA, August 1996.

[Kre95] U. Kremer. *Automatic Data Layout for Distributed Memory Machines.* PhD thesis, Rice University, October 1995. Available as CRPC-TR95-559-S.

[Kre96] U. Kremer. Automatic data layout with read–only replication and memory constraints. Technical Report LCSR-TR93-283, Department of Computer Science, Rutgers University, December 1996.

[WH95] D. Wood and M. Hill. Cost–effective parallel computing. *IEEE Computer*, Feb 1995.

[2] *CPLEX* is a trademark of CPLEX Optimization, Inc.