Notes on Data Dependences and Reordering Transformations, as a precursor to Data Locality Transformations

CS4961
September 13, 2011

Today's lecture is about how to determine if a reordering of a computation preserves its meaning. We formulate the notion of preserving meaning through the concept of a data dependence. The notion of data dependence relates to the conditions outlined by Bernstein in 1966 for when if was safe to execute two processes in parallel. The safety criteria focused on properties of memory accesses (assuming memory is shared between processes), identifying any overlap in data accessed by the two processes that would potentially cause the parallel computation to produce an incorrect result.

Bernstein's conditions (1966): I_j is the set of memory locations read by process P_j, and O_j the set updated by process P_j. To execute P_j and another process P_k in parallel,

$$I_j \cap O_k = \phi$$
$$I_k \cap O_j = \phi$$
$$O_j \cap O_k = \phi$$

Observe that there is one condition missing here, among the 4 possible combinations of the input and output of P_i and P_j. The inputs can be overlapping, and it is safe, since neither's memory state is being modified.

Now the related definition of a data dependence.

- **Data Dependence:** Two memory accesses are involved in a *data dependence* if they may refer to the same memory location and one of the accesses is a write.

Load-store classification:

Expressed in terms of load-store order in the sequential program, we now provide some definitions of different types of dependences on memory accesses to the same location.

1. True dependence (read after write): The first access stores into a location that is later read by the second access. ***X = …; … = X;***
2. Anti-dependence (write after read): The first access reads from a location into which the second access later stores. ***… = X; X = …;***
3. Output dependence (write after write): Both accesses write to a location, and the ordering of the writes must be preserved so that any later accesses read the correct value. **X= …; X = …;**

This classification is taken from the parallelizing compiler literature. So the role of a compiler is to analyze memory accesses to pinpoint the dependences and determine whether parallelization is safe. Compilers must be conservative in deciding whether two accesses are to the same memory location. Compilers also perform other *reordering transformations* on the code to make parallelization safe or more efficient (e.g., have coarser granularity). Now we will look at a few definitions to understand how compilers reason about parallelization and other reordering transformations.

A data dependence can either be between two distinct program statements or two different dynamic executions of the same program statement. Much of the parallelizing compiler literature focuses on dependence analysis of loop nest computations, as loops offer ready sources of balanced and scalable parallel computations. As we will discuss in the next lecture, dependence analysis can be formulated on iteration instances in the iteration space of a multi-dimensional loop nest. Here are a few definitions that help formalize how parallelizing compilers reason about the safety of parallelization.

- **Definition 2.5 (equivalence of computations):** Two computations are equivalent if, on the same inputs, they produce identical outputs and the outputs are executed in the same order.
- **Definition 2.6 (reordering transformation):** A reordering transformation changes the order of statement execution without adding or deleting any statement executions.
- **Definition 2.7 (preserving dependences):** A reordering transformation preserves a dependence if it preserves the relative execution order of the dependences' source and sink.
- **Fundamental Theorem of Dependence:** any reordering transformation that preserves every dependence in a program preserves the meaning of that program and is a valid transformation.

Reduction computations
In the absence of synchronization, parallelizing the pairwise sum and count 3s examples, as in the textbook, actually violates the notion of a valid transformation, since there is an inherent dependence on the variable that accumulates the final result. However, these computations can be parallelized because they are *reductions*.

Definition: A *reduction computation* computes a result that represents a reduction in the dimensionality of the input data. A reduction computation exhibits data dependences, but if the operation on the input data is *associative*, it is safe to reorder the operations and execute them in parallel. A reduction computation has the structure, ***"result = result op …"*** Operation ***op*** must be associative.

References
Bernstein, A. J. (October 1966). "Program Analysis for Parallel Processing,' IEEE Trans. on Electronic Computers". EC-15, pp. 757–62.

Optimizing Compilers for Modern Architectures: A Dependence-Based Approach, Allen and Kennedy, 2002, Ch. 2.

For next time, we will look at a more formal definition of dependences, and work through a set of reordering transformations.