Automatic Parallelization: An Overview of Fundamental Compiler Techniques (4-7)

Transformations
Transformations that involve reduction and recurrence recognition allow dependences to be ignored as the operations on data are commutative. One way to do this is loop peeling and splitting. Peeling involves removing a small number of iterations from a loop and reproducing them as straight-line code before/after the loop. Splitting does the same thing but after a number of iterations are peeled, executing them in a second loop is the best way to generate code.
Loop skewing lets the loop be parallelized by changing the direction vector of a dependence. The opposite of something like this is to do loop normalization, or loop deskewing.

Can also concatenate one vector onto another one by use of an induction variable which is a function of the loop induction variable. Forward substitution is when variables that are repeatedly used are assigned into a temporary area. This can be a body of a loop, expression, parts of an expression. Scalar expansion eliminates the dependencies that forward substitution but uses storage to accomplish this instead of using up computation time. This kind of method can also be applied to arrays that also uses temp variables in a loop, and they can be parallelization pretty well and is called array privatization. Node splitting is used for flow and dependencies that leverages the reuse of storage. They add extra storage to save the read locations the read location can be moved around the loop and break the cycle. This method and array privatization fall in the same category of targeting anti-dependence, node splitting is used when the A-D exists at run time and the global state must be maintained.

Array privatization is useful when a variable is assigned a value used within that iteration. Reduction recognition breaks dependences and enables parallelization through by exploiting associativity. Although most loops can’t be parallelization because of the data dependencies, the most important techniques are scalar expansion and array privatization because they follow common programming idioms such as using scalars and array in nested loop to hold temp values it leads to anti/output dependences that aren’t fundamental to the computation but are just insufficient resources being used, uses aforementioned methods dramatically reduce dependences.

Transformations (Recursive)
Loop blocking breaks the iteration space of a loop into blocks and is used to split loops into chunks of computation or as one step in performing loop tiling for locality purposes. By unrolling loops you allow a wider window for optimizations while not changing the order of fetches and stores is always legal to perform. Loop distribution, or fusion/fission merge multiple loops into a monolithic
one by using inverse transforms. Loop reversal just reverses the order in which the iterations of the loop will be executed, i.e. from 0 to n-1 vs. n-1 to 0. This reverse the direction of dependences and is useful to fuse loops that might be illegal to fuse (?!). Loop interchange looks to increase the temporal/spatial locality of accesses and interchanges the positions of the loop so the inner is not in the outer loop. Aside from just locality exploit (i.e. array access may occur in wrong order b/c library reuse), using this method if the inner loop is parallel interchanging it with outer decreases the number of times the parallel loop is instantiated at runtime and increases the work in the parallel loop. Tiling is used when loop has inefficient accesses to an array because of inefficient ordering. You can interchange as a first step but that isn’t enough, you also need to tile the iteration to cover the original iteration space and reorder so all array regions accessed in a tile fit into the cache. Need to use this for matrix multiply and transpose operations. To account for linear transforms of iteration spaces, i.e. skewing, reversal, and interchange (tiling, strip mining, fusion, cannot). We can use a unimodular matrix that represents the effects of skewing, reversal, and interchange to give an optimal loop nest. (?!)

**Distributed Machine Compilation**

Distributed memory machines have nodes (multicore processors) that are connected through internet and usually just execute single program multiple data programs (SPSD). Communication is done through messages and collective communication specified by the compiler or the user because the computations are not fully independent.

The problems that need to be solve for this programming paradigm to work are data distribution, computation data by reference, computing partitioning, generation communication, and handling programming languages that are targeted for distributed machines. In distributed memory machines parallelism is exploited by having communication go across multiple processors by distributing the data across the processors using methods like replicated, block, cyclic, and cyclic-block size. Can perform replicated distribution where all processes own every element of an array. Cyclic distribution (load balance) spreads the elements across process in a round robin while block cyclic distribution emphasizes load balance and allows benefits of both. Block-cyclic distribution just has the benefits of both (?!). Partitioning is needed as the loop bounds can only be shrunk to the cover the iteration of the computation by each process. Generating communication is needed to have a single coherent view of the data being maintained. And it is beneficial to move the communication out of the inner most loops to maximize parallel and min the overhead of communication.

**Diophantine Equations**

Essentially equations where domains, range and coefficients are ints and is indeterminate polynomials. Most of the problems in compilers and solving for dependencies seems to be solving these kinds of diophantine equations. Imagine a matrix with a bunch of operations that need to be solved as a set of linear equations.