Understanding Parallelism - Loops*

Charles Severance
Kevin Dowd

Loops are the center of activity for many applications, so there is often a high payback for simplifying or moving calculations outside, into the computational suburbs. Early compilers for parallel architectures used pattern matching to identify the bounds of their loops. This limitation meant that a hand-constructed loop using if-statements and goto-statements would not be correctly identified as a loop. Because modern compilers use data flow graphs, it's practical to identify loops as a particular subset of nodes in the flow graph. To a data flow graph, a hand constructed loop looks the same as a compiler-generated loop. Optimizations can therefore be applied to either type of loop.

Once we have identified the loops, we can apply the same kinds of data-flow analysis we applied above. Among the things we are looking for are calculations that are unchanging within the loop and variables that change in a predictable (linear) fashion from iteration to iteration.

How does the compiler identify a loop in the flow graph? Fundamentally, two conditions have to be met:

- A given node has to dominate all other nodes within the suspected loop. This means that all paths to any node in the loop have to pass through one particular node, the dominator. The dominator node forms the header at the top of the loop.
- There has to be a cycle in the graph. Given a dominator, if we can find a path back to it from one of the nodes it dominates, we have a loop. This path back is known as the *back edge* of the loop.

The flow graph in Figure 1 (Flowgraph with a loop in it) contains one loop and one red herring. You can see that node B dominates every node below it in the subset of the flow graph. That satisfies Condition 1 (list, p. 1) and makes it a candidate for a loop header. There is a path from `E` to `B`, and `B` dominates `E`, so that makes it a back edge, satisfying Condition 2 (list, p. 1). Therefore, the nodes `B`, `C`, `D`, and `E` form a loop. The loop goes through an array of linked list start pointers and traverses the lists to determine the total number of nodes in all lists. Letters to the extreme right correspond to the basic block numbers in the flow graph.

*Version 1.3: Aug 25, 2010 11:20 am -0500
†http://creativecommons.org/licenses/by/3.0/

Flowgraph with a loop in it

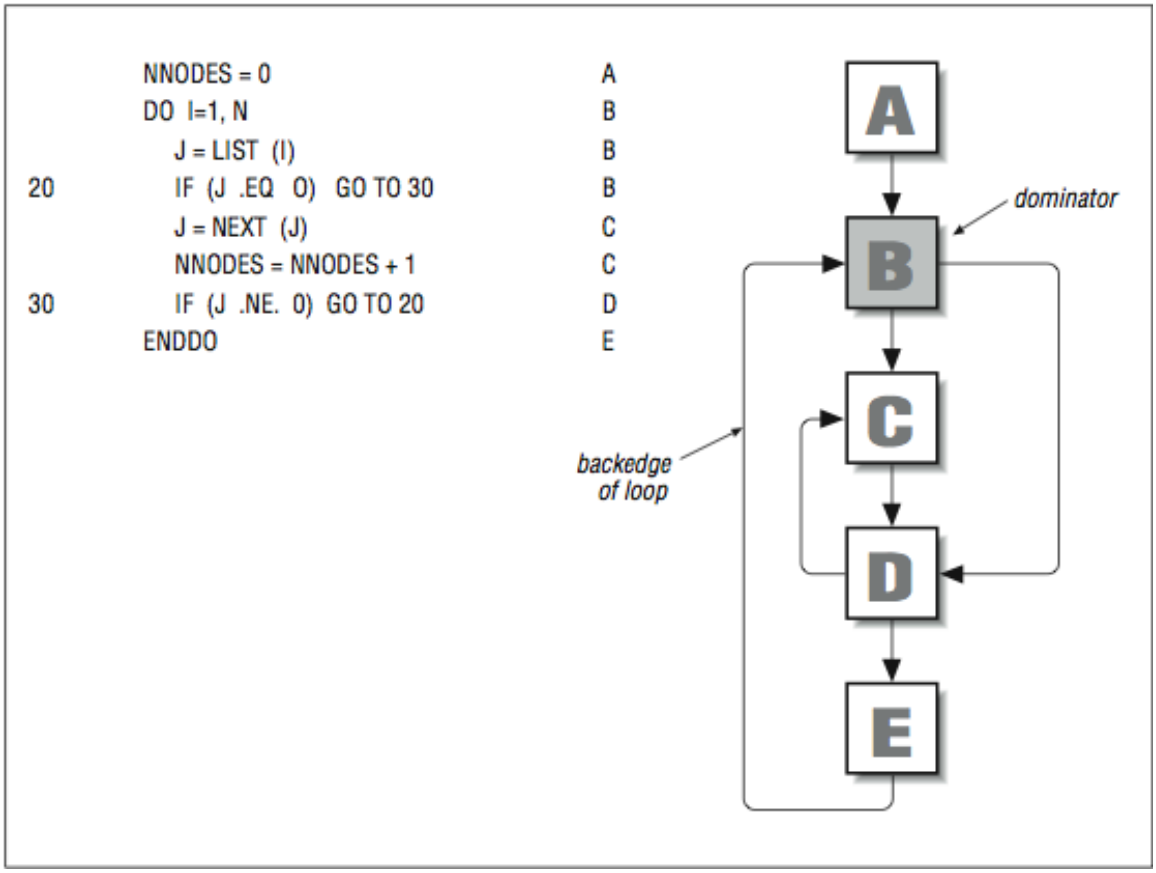

Figure 1

At first glance, it appears that the nodes `C` and `D` form a loop too. The problem is that `C` doesn't dominate `D` (and vice versa), because entry to either can be made from `B`, so condition 1 (list, p. 1) isn't satisfied. Generally, the flow graphs that come from code segments written with even the weakest appreciation for a structured design offer better loop candidates.

After identifying a loop, the compiler can concentrate on that portion of the flow graph, looking for instructions to remove or push to the outside. Certain types of subexpressions, such as those found in array index expressions, can be simplified if they change in a predictable fashion from one iteration to the next.

In the continuing quest for parallelism, loops are generally our best sources for large amounts of parallelism. However, loops also provide new opportunities for those parallelism-killing dependencies.