Automatic Parallelization: An Overview of Fundamental Compiler Techniques

Introduction
There is shared and distributed memory parallelism that can be exploited, and the difference is in how the threads communicate. In shared threads communicate with reads and writes to shared memory and in distributed memory, processes do it with messages. All applications can be parallelized through a process using SIMD, SPMD, or MIMD. SIMD occurs when multiple processors execute the same instruction on different data elements. This results in simpler hardware because a single control unit can control many functional unit. The SPMD execution has multiple copies of the same program each independently execution in its own data. Distributed memory execution is an example of SPMD. Shared memory can also be thought of as SPMD though (difference?) MIMD programs have different code on different processors such as a multithreaded program where not all threads are identical.

We can turn one of these forms into a function of the other, for example, merging all the MIMD programs it can be executed as a SPMD. And a SPMD program can be executed on a SIMD machine but will require many processing elements. Basically, we need to look at all these different kinds of parallelism and try to find a way that the compiler can automatically provide speedup for this(?) Representation of a program is shown as an abstract syntax tree or a control flow graph.

The goal is that the compiler takes the sequential program and translates it into a parallel program that makes use of all the cores and parallelism built in.

Dependence Analysis
To determine which parts of a program can execute in parallel three techniques that are used to determine when two references access the same storage area dependence, use-def, and alias analysis.

Constant Propagation attempts to find out what values in a program are constants. It works by traverse the CFG and is a forward data flow analysis. There is a lot of technical proofs and transfer functions about how the exact algorithm works by essentially first construct a CFG, then associate transfer functions with the edge of the graph. The TF depends on the statement or program itself. And third, iterate until no more changes occur. This algorithm somehow always stops and the solution is unique regardless of traversal.
Alias Analysis is done to find the different names but which an object in memory is referenced. Their definition of object in the paper just means any collection of memory that can be referenced in while by the program. In doing data dependence analysis there are four kinds of dependences, a flow, an anti an output and an input. The flow exist when a value is written into an array and read later. The Anti exists when a value is read from an array and write to that array. The output is when a value is written to an array and overwritten and input happens across writes.

Control dependence is just the compiler ensuring that the operation is performed at runtime as it normally would is is not performed when it normally shouldn’t it, basically nothing should change and that is the control dependence. Use-def chains is by definition a structure that has a use of a variable and all definition of that variable that can reach the use without any definitions in the middle.

In parallel programs dependence analysis is needed to build conflict graphs to ensure no conflict in parallel execution. The CG is a way of viewing the dependencies between the different parts of the program.

**Program Parallelization**

Programs can be parallelized with acyclic and cyclic dependency graphs. This is done by creating a graph for the loop, finding strongly connected components, marking nodes..and a bunch of other steps but essentially the point of this is to create a deterministic(?) way to do the actual parallelism once it has been found.

If we are generating code for vector hardware, we need to partition the statements in the original loop into a sequence of loops each with a single operation by the vector unit. Then the smaller loops can be transformed to execute vector operations on data. They also talk about loop interchange as a way to make another loop the inner most loop if a lop other than the most inner is easier to vectorize.

Produce consumer synchronization is just as it sounds, similar to what we learned in OS classes and it’s just a way to find synchronization with memory to make sure that all the tasks can be “scheduled” in parallel without issue. This method can also be optimized by reducing the number of synchronization operations needed to sync all the dependencies in a loop. This optimize lower overhead of synchronizing them and allows a post/wait pair (one of them) to synchronize more than one dependence. Parallelization recursive programs are non privileged when compared to the other examples that they give like the simple while loop parallelization and requires far more attention be paid to how the program execution and flow is.

Instruction level parallelism can be exploited in loops with software pipelining