The paper presents an algorithm which makes use of two core techniques for dealing with data dependencies while parallelising an input program: privatization and sequencing.
Privatization means that for each variable, which is affected by race conditions between different threads of the program, a private copy is allocated. This private copy doesn't suffer the race-time condition anymore, since at most one thread has concurrent access to it. What makes privatization difficult is the fact, that private variables, which are used in a block, might have to copy-in the value assigned to in a prior block (with respect to the control-flow-graph) or have to copy-out the value for use in a post block. This especially hard for parallel loops, because the presented approach uses statically (or syntactically) analysis, which limits the possible degree of privatization. For example for parallel loops only the value of the last iteration can be copied out, but what if the loop terminates early? In this case the privatization doesn't seem to be applicable there, leaving the loop sequential. Even more restricting is the fact, that there is no whatever mechanism, which allows to copy-in values from previous iterations of the parallel loop. This situation makes this kind of parallel loop construction algorithm for most applications unusable.
Sequencing isn't a parallelisation technique, it's rather the opposite. The algorithm at first tries to exploit as much parallelism as possible by transforming all loops to parallel loops and wrapping all statements in parallel-executable blocks. Through sequencing the algorithm afterwards recreates sequential blocks out of the parallel blocks, which are enforced by data dependencies and cannot be privatized. For loop-carried dependencies the entire loop which carries this dependency has to be sequentialized. For loop-independent dependencies the algorithm tries to find the lowest common ancestor in the control dependency tree, such that identically control-dependent children of this ancestor are put in the same sequential block. Nevertheless this approach is used as the last possibility, if there isn' t any kind of privatization applicable.

Summary I1

The paper we discuss this week is about an approach for automatically generating nested, fork-join parallelism in sequential programs. As the programming language has to support features for generating forks and joins it can be implemented in a language like OCCAM or Parallel Fortran. In order to integrate concurrency the authors introduce two new constructs ("DOALL", "COBEGIN" / "COEND") to express instructions that can be executed in parallel. Their meaning is that loop iterations of a "DOALL"-loop and the statements or blocks within blocks beginning with "COBEGIN" and ending with "COEND" can be executed in parallel. Furthermore they need an additional construct "PRIVATE" to make loop-iterations access distinct instances of certain variable. But in my opinion the paper doesn't make this point really clear so I didn't get this.
As in sequential programs there are statements which depend on other statements the authors have a look on control dependence and on data dependence. They define "post-domination" and "control-dependence" on nodes in a control flow graph and introduce loop-carried, loop-independent, flow and storage-related data which is needed for the algorithm.
When the algorithm is executed it first constructs the control and data dependence graphs, then initializes all loops to "DOALL" and the so called "SEQSET" of all nodes to themselves. After that it tries to eliminate dependences between statements by privatizing. But there are also dependences that cannot be eliminated by privatization. For that reason the authors use another algorithm which solves this problem by reducing parallelism. It gets a dependence on a variable and the control and data dependence graphs as input and outputs the reduction in parallelism that satisfies this dependence.
Applying this algorithms on a sequential program we get a fork-join-parallelized version of it.

Questions / Opinion:
In my opinion this approach is a quite optimistic one, because in most of the programs the dependences are so complex that the parallelized part of the programs would be very small. But since this paper was already released in 1989 it can be seen as one of the first attempts for automated parallelization.

The authors present an algorithm to automatically transform a sequential program into a parallel one, that uses fork-join parallelism and process nesting. The algorithm gets the control-flow- and data-dependence-graph of the sequential program as input, and returns a representation of the parallelized variant in a concurrent execution model. The algorithm introduces privatization to resolve storage-related dependencies.

On page 9 the authors say that satisfying dependences for which privatization fails, other dependences may become satisfied. I would like to see an example for this, and also a reason why privatization does not always increase parallelism.

To me, the algorithm seems pretty simple. For simple programs the algorithm might perform well. However, I think that for more complex scenarios the algorithm is not able to efficiently parallelize. Data dependencies are only resolved by simple privatization, which in many cases is not applicable. Especially loops often suffer from loop-carried dependencies, avoiding application of the algorithm.
On page 4, the authors say to apply the COPYOUT tag to a variable being assigned inside a loop, a recurrence system has to be solved at run-time to determine which process would issue the last write to that variable. Furthermore, they say that therefore static analysis must assume that the last iteration of the loop writes the final value for the variable.
I think that the presented algorithm could be combined with other optimization techniques, such as constant propagation or program slicing, to gain more information, and by that more parallelism. More advanced techniques to handle dependencies inside loops would be of great benefit; to me this seems to be the bottle-neck of the algorithm.

It seems to me like the algorithm introduces many short fork-join parts, every time adding small overhead to the program. I would like to know whether the speedup gained by parallelism is greater than the introduced overhead by fork/join.
Another problem I see there is that the algorithm might create to many processes while parallelizing. This can lower the speedup, since it puts increased pressure on the host system's scheduler.

I am also disappointed that they did not present any benchmark results. They only say "it has proven to be efficient". Can this somehow be verified?

First Summary

This paper presents an algorithm for automatically parallelizing dependence-based representation of sequential programs into nested fork-join constructs.The resulting parallel programs form a model that allows arbitrary nesting of processes which all together achieve the concurrent execution of loop iterations and statements that are contained in the sequential program; the constructs for expressing parallelism are the DOALL loops and COBEGIN...COEND statements. This algorithm based on two graphs: control and data dependence graphs, which together represent constraints that must be satisfied by any transformation of the original sequential program.The control dependence graph determine whether other statements conditionally will be executed ,whrease the data dependences establish a relative ordering among statements of a procedure .This algorithm also embodies two techniques for dealing with data dependences:sequencing,privatization.The technique of privatization is used to avoid the loss of parallelism by describing conditions under which a storage-related dependence can be eliminated and that through introducing private variables ,where the allocating process accesses a distinct instance of each variable declared private.Privatization declares extra storage at compile-time which can be allocated only where parallelism is actually achieved at run-time.This algorithm takes time proportional to D(deg + log*D) + N, where N is the total number of data dependences in the given set , D is the total number of loop-independent dependences that are in the set and deg is the maximal out-degree of any node in the control dependence graph.Thus, the resulting parallel program achieve ideal parallelism model that satisfies all dependences that have been processed by this algorithm.

My questions:

1)I miss understand this idea: Statements immediately nested inside an irreducible interval cannot be parallelized and must appear in their original, sequential order.

2)I miss understand the loop-independent dependence.

3)why the more efficient algorithm requires just a single pass over the post-dominator tree of the control flow graph?

Automatic Generation of Nested, Fork-Join Parallelism: Summary

This paper talks about handling data dependences and control dependences in the programs for parallelization. Two techniques namely sequencing and privatization have been discussed in detail for data dependence handling. In this model, program is divided into many parallel segments each of which is forked into a process, and in the end results are combined. Hence the name Fork-join parallelism.

Initially author describes about the parallel constructs like CO-BEGIN, CO-END, DOALL, PRIVATE which are added to the normal sequential language constructs to report potential for concurrency. PRIVATE construct causes creation of new versions of the same variable in a particular parallel block just like Single Static Assignment programs. Values are copied in and out of these versions using constructs COPYIN, COPYOUT.

Authors then describe about the concept of identical control dependence which is pivotal for the parallelization algorithm described later. Acyclic control dependence subgraph of the program is used in sequencing and privatization. Author identifies different kinds of data dependences in the programs based on the pattern of data access in subsequent iteration of the loop or flow of value between two statements.

Algorithm for parallelization: This makes use of concepts discussed above. Nodes in the control graph is divided into many sets. Each such set consists of nodes that are identically control dependent to each other. Nodes in each such set can be executed in parallel. All loops are marked DOALL. For each data dependence, check and mark if it is possible to carry out privatization of the variables(try-privatization algorithm). If that is not possible carry out sequencing. Carry out privatization to the marked dependences.

Sequencing: In case of loop carried dependence, loop is made sequential. In case of loop independent dependence, Lowest common ancestor L of the nodes in the dependence is found.. Sequence sets of children of L is merged thereby marking them as executable in parallel.

Try-privatization algorithm: This returns the degree of privatization needed for the data dependency given. For flow dependence it returns 0, indicating privatization is not possible. For loop carried dependence , liveness of the variable is checked to determine the degree of privatization needed.

Open questions:
1. Relation between identical control dependence and potential for parallelization.
2. Idea behind try-privatization algorithm.

Automatic Generation of Nested, Fork-Join Parallelism

Thomas Karos

April 20, 2014

1 Summary

In one sentence, this paper gives a sequential language, augments it by parallel constructs and gives an algorithm to gain parallelism out of a purely sequential program by applying those new parallel constructs inside the sequential program.

So, the paper gives a first glance on how to automatically introduce parallel constructs into a sequential program without breaking the semantics and without changing the core structure of the program (given by blocks, loops etc.). This is done on the base of a quite simple language consisting of statements, if-branches, for-loops and begin... end blocks. The parallel elements are a doall-loop which is allowed to execute all iterations in parallel, and a cobegin... coend block which allows to have all blocks that are immediate children of it to be executed in parallel.

The given algorithm works based on two main pieces of input information, once the data dependence graph and the flow dependence graph for the given program. The first one connects statements to "former" statements on which their outcome depends (according to the assigned variables). This data dependence information is further augmented by some flags that indicate how this data dependence relates to control flow, for example whether the dependence only holds inside on iteration of a loop (e.g. locally defined values) or spans over several iterations (counters, buffer content etc.). The flow dependence graph gives information about where which statement has influence on the control flow of the program.

Based on that information, the presented algorithm proceeds in multiple steps: Initially, it assumes perfect parallelizability by tagging each loop as doall-loop and each block as cobegin... coend block. Now, for each dependence, it re-establishes the sequential relation, if the current state potentially breaks the original – sequential – semantics, by either sequentialization or privatization. The first thing is roughly spoken resetting (all) the constructs that cause the distruction of semantics. The second one is more tricky. It enables parallelism by giving a thread a private copy of some value. If that is possible (e.g. because a variable is reused or reassigned without overlapping effects), it also indicates whether this private value must be supplied for later computations by copyout and whether this value needs to be initialized with a former value (copyin). The result is a program that is annotated with information about which parts can be executed in parallel and which variables need to be copied to do so.

2 Open Questions

- How far can this algorithm be extended? For example, is it possible to make it recognize modulo counters in order to parallelize loops that touch each and every second array element.
- The algorithm does not say anything about the costs of parallelization. Is that reasonable? On which bases can a runtime environment or a compiler decide whether to take the privatization option and fork or simply stay sequential?

Automatic Generation of Nested, Fork–Join Parallelism
===

The authors present an algorithm which, given a imperative program and its associated control flow and data dependence graph yields an equivalent program using fork/join parallelism. Their approach starts by setting all statements in a loop to be run concurrently. Afterwards, to satisfy dependencies, statements are either put into sequence again, or privatized. The authors consider loop-caried, loop-independent, flow and storage-related data dependencies. Privatization basically copies values, so that processes can work with private instances of variables. This aproach only parallelizes statements in reducible intervalls. The authors mention that node splitting would allow to obtain fully reducible programs; it remains unclear why they don't use it.

- Is privatization of any use when the to be transformed program is already in SSA form?
- What is the overhead of tagging variables with COPYIN and COPYOUT?
- Are all sections marked as parallelizable concurrently executed? If so, doesn't that cause too much overhead/pressure on the scheduler?
- The given examples don't contain memory accesses. What happens in their presence?
- What exactly is meant by "reducible intervall"?
- Which modifications would be necessary to support more control structure, like break/continue statements?
- Are there any techniques that can deal with flow dependencies? Could speculization help in this case?
- The sequence algorithm works on whole loops. Isn't that wastefull if the loop does also some further work, which doesn't affect the dependence, and which could in turn be run in parallel?
- The authors discard synchronization primitives as too costly (and their system does not support them anyway). Could using them be benefitical, though, especially in the case where a dependency only occurs in one branch of a switch statements, which is seldomly executed? If so, could the algorithm be easil augmented to make use of them?

Paper I1

This paper introduces an algorithm to automatically parallelize a program. To do that it first of all introduces new language constructs: the DOALL loops and the COBEGIN…COEND statement. Those mark sections which can be executed in parallel. DOALL loops are used to execute loop iterations in parallel (basically if a loop needs 4 iterations sequentially, those 4 iterations can be executed in parallel).

The COBEGIN/-END statement is used to signal possible parallel execution of statements inside this block.

Of course this can lead to data races so simply executing everything in parallel is not the way to go. To be able to do that a new construct has to be introduced: the PRIVATE statement. This is used to signal the program to access a distinct instance of a variable to avoid those problems.

The algorithm itself works using a dependency analysis and based on those information which parts can be marked with “DOALL” and “COBEGIN/-END”. The pseudocode describes its behavior roughly like this:

All loops are converted to DOALL loops and a mysterious “SEQSET” which is used to keep track of the control dependencies is initialized and calculated. Afterwards some more dependencies are tried to get resolved by privatizing. If I managed to understand the algorithm correctly, the COBEGIN/-END blocks are inserted at the nodes of the SEQSET which are independent to other nodes.

Of course privatizing cannot eliminate every dependency. Code which is dependent on each other due to the control flow is such a case.

Summary of Automatic Generation of Nested, Fork-Join Parallelism

The paper Automatic Generation of Nested, Fork-Join Parallelism presents an algorithm that can automatically parallelize programs into fork-join programs. The fork-join programs are a composition of statements and loops of sequention programs. The data dependencies in the problems are dealt by this algorithm using two different techniques:
1. Sequencing: this process reduces parallelism;
2. Privatization: this process uses private variables to remove dependencies.
This algorithm takes as input an analyzed sequential procedure and its control and data dependency graphs. The algorithm then looks for parallelism in the loops, or applied node splitting to find fully reducible programs.
This algorithm is a first of its kind which can automatically generate parallelism for general models.
A single instruction stream, referred here as a process, exists before, after and during every execution. To allow for one process to execute multiple concurrent processes two additional constructs are introduced, they are DOALL and COBEGIN...COEND. The DOALL is a cognitive sequential iterative process which allows for concurrent execution of iterations. It is a fork of iterations with corresponding joins at the completion. Where as, the COBEGIN...COEND construct is similar to a BEGIN...END construct except that closely nested blocks are executed concurrently. COBEGIN is the fork and COEND is the corresponding join.
A control dependence graph lists all the conditions that affect execution of a statement. It is the control flow of a program. The data dependence graph is a directed graph whose nodes are the nodes of the control flow graph and the edges represent execution of read or write. Together the data and control dependence graphs put forward all the necessary contraints that need to be satisfied after the changing of the sequential program into a fork-join program.
This algorithm provides for a good basis for compiler analysis and optimization. The privatization technique eliminates storage related dependencies without loss of parallelism. The worst case complexity of the algorithm is D(deg + log*D) + N, where N is the total number of data dependences in the given set , D is the total number of loop-independent dependences that are in the set and deg is the maximal out-degree of any node in the control dependence graph.

Summary

The paper describes an algorithm for transforming sequential programs into concurrent programs using nested-fork/join parallelism. The resulting programs will execute as much code in parallel as is feasible, they consist of sequential parts called processes that can spawn new processes on their own (nesting), multiple spawned processes may be executed in parallel (fork) but the execution of processes following their parent process will be postponed until all forked processes have terminated (join).
The authors impose an abstract programming system based on PTRAN which they claim to be adaptable to other systems, this adaptation is not described further. The assumptions the authors make are: Programs are run with a global shared memory where variables can be defined as private so they are only accessible to the declaring process and its nested processes. Three new language constructs are introduced:
DOALL, similar to a common loop but each "iteration" is performed in parallel with no defined order;
COBEGIN...COEND, a block which executes all immediate nested blocks or statements in parallel;
PRIVATE, declaration of a variable that is confined to the be used by a single process and its nested processes, may be combined with COPYIN or COPYOUT which initialize the variables value before forking or export its value for later use after joining.
The authors classify existing data dependences using four different annotations:
Loop-carried: Data is being written in one iteration and referenced in another.
Loop-independent: Data is only written and read within complete iterations.
Flow: Data is written at one point and read later.
Storage-related: Data is read or written by one statement and overwritten at a later point.
The described algorithm works roughly as follows:
1. Build the control-flow-graph, control-dependence-tree and the data-dependence-tree of the original program.
2. Initialize all loops in the original program as DOALL constructs, initialize all nodes in the CDT as execution-independent among them.
3. For all data-dependences check if they can be satisfied by privatizing. If this is not possible sequence them: If the dependence is loop-carried mark the carrying loops for sequential execution, if it is loop-independent find the lowest common ancestor in the CDT and mark its two children leading to the depending statements for sequential execution.
4. Privatize all data-dependences which can be privatized:
    - Flow dependences can not be privatized
    - If the dependence is loop-carried it can only be privatized if its variable is introduced within the loop. COPYOUT is required when its variable is used after the loop is executed.
    - If the dependence is loop-independent it may be privatized depending on the other storage-related dependences.

Possible discussion questions

- How can this approach be adapted to other languages/memory systems? (e.g Java/C/Python/Functional languages)
- Does the assumption that a COPYOUT has to be performed by the last iteration in a DOALL construct always hold?
- What data-dependences could be unprofitably eliminated by privatization if performed before sequencing?
- Why does renaming not increase parallelism for this algorithm?