Automatic Extraction of Functional Parallelism from Ordinary Programs[a]

Milind Girkar, Constantine D. Polychronopoulos

Presented by Andreas Wuest, ETH Zurich

(awuest@student.ethz.ch)

June 14, 2005

[a] Automatic Extraction of Functional Parallelism from Ordinary Programs, M. Girkar, C. D. Polychronopoulos, IEEE Transactions on Parallel and Distributed Systems, vol. 3, no. 2, March 1992, pp. 166-178.

OUTLINE

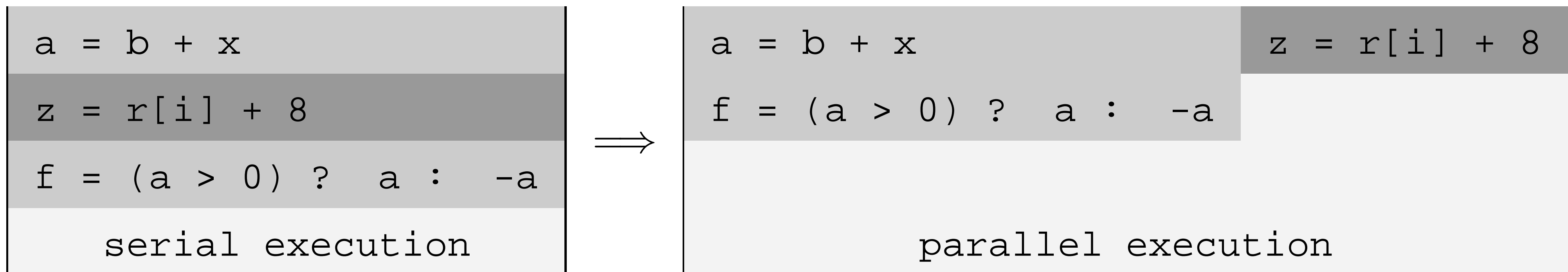
1. Introduction
2. Objectives of Automatic Parallelism Extraction
3. Proposed Method
4. Further Examples
5. Future Prospects
6. Questions

DIFFERENCE BETWEEN PARALLELISM AND CONCURRENCY

Parallelism: parallelism denotes parallel execution not defined by but also not affecting the semantics of a program. Low-level property.

Concurrency: concurrency denotes parallelism at the conceptual level, modelling e.g. an aeroplane which possesses lots of independent and parallel working units. High-level property.

- An ordinary serial program is only a partial order of statements, and execution can therefore happen in arbitrary order, as long as the data and control dependencies are observed. This leaves space for an enormous amount of extremely fine grained parallelism.



- In fact, what is happening in today's processors is that machine instructions already get rescheduled, making use of the actual state space of a program, which can only be done dynamically.
- This scheme could be raised to the software level, detecting dependencies statically, finding the smallest independent instruction sequences, and then schedule them at runtime exploiting the knowledge about the actual variable assignments and control decisions.

- Parallelism and concurrency are not mutually exclusive! In fact, we need both forms of simultaneousness: concurrency to model systems with simultaneously working agents, and parallelism to speed up the execution of such programs (but also to speed up programs with only a single conceptual thread of execution).

WHY PARALLELISM

- Circumvent blocking system calls
- Speed up computations
- But: in the last decade we have seen a proliferation of parallel machines also for the desktop market!
- In near future, we will reach the megahertz limit: (serial) execution of an atomic instruction cannot be made faster. The only solution will be to execute instructions of the same program in parallel, making use of several processors per machine.

- For this to achieve, not only scientific applications have to be able to run in parallel, but also normal applications like an email client, a window manager, or even the operating system itself.
- Threading does not help: threading is on the conceptual level (concurrency), and the programmer has to model the program so that it can run several tasks in parallel. But operating on the conceptual level leads to only few parallelism, and is often further reduced by the various threads being blocked in system calls.

- We therefore have to:
  - strictly separate parallelism from concurrency
  - keep parallelisation a low-level concern, handled by the compiler
  - not clutter the programmer with thinking about parallelism (the programmer must not add additional directives to guide the compiler through parallelisation)
  - find efficient methods for switching between separate parallel tasks
  - integrate all that into a modern and widely used compiler

LEVELS OF PARALLELISM

- **Loop-level parallelism:** distribute a loop on one or multiple arrays over multiple processors
- **Data-parallelism:** subsume equal operations on different data (or loops as well) to one instruction (for execution on vector machines)
- **Task-level parallelism:** general parallelisation. For example parallel execution of subroutine calls, of independent loops, or even independent basic blocks. This is the main concern of this research area.

MAIN OBJECTIVES

- Sound automatic detection of data and control dependencies
- Automatic extraction of structured and unstructured parallelism, down to the basic-block level
- Efficient scheduling of the independent tasks at runtime
- Make this work for PRAMs and in future maybe also for distributed machines

INGREDIENTS

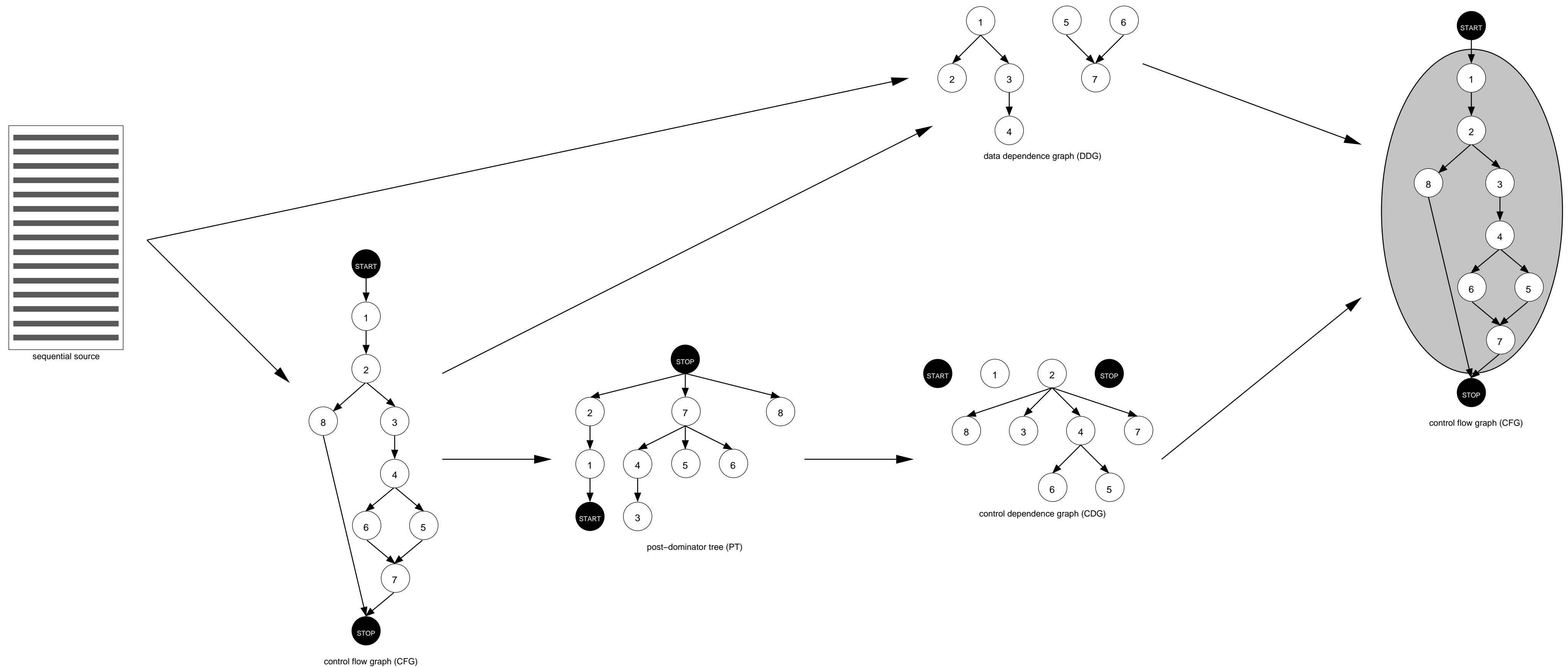
- We need a datastructure to represent parallel tasks and capture the dependencies, as well as their scope (covered in this paper)
- We need an efficient scheduling algorithm to execute the code contained in this data structure (covered in other papers)

OVERVIEW

A simple method:

```
1  public boolean checkString(byte[] a) {
2          boolean stringOK;
3          int len = a.length;                              node 1
4          if (len > 0) {                                   node 2
5                  int last = len - 1;                      node 3
6                  if (a[last] == '\0') {                   node 4
7                          stringOK = true;                 node 5
8                  } else {
9                          stringOK = false;                node 6
10                 }
11                 return StringOK;                         node 7
12         } else return false;                             node 8
13 }
```

PROCESS OF BUILDING THE HIERARCHICAL TASK GRAPH (HTG)



DEPENDENCIES

There are two types of dependencies which must be considered:

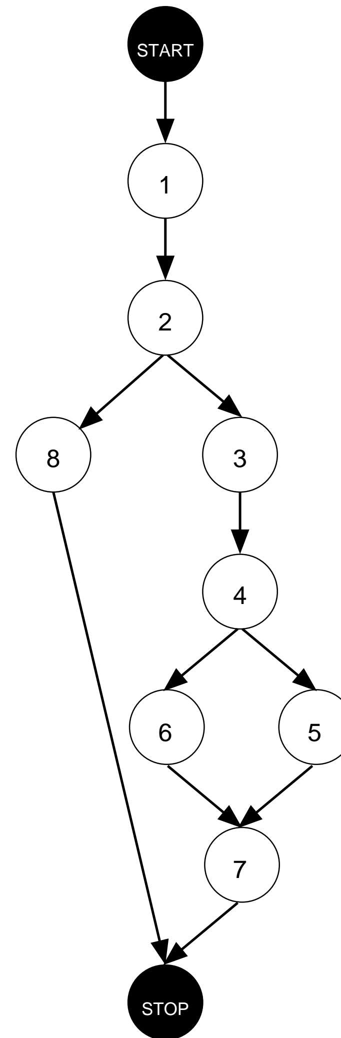
- **Control dependency:** dependencies which arise from *conditional jumps*, i.e. `if` statements, and condition evaluation in `for`, `do` and `while` loops. In our example, node 2 (amongst others) performs a conditional jump. The evaluation of condition `len > 0` determines if node 3 and its descendants, or node 8 should be executed. Therefore, node 3 is said to be *control dependent* on node 2.
- **Data dependency:** dependencies which arise from *uses* of previously assigned variables. In our example, node 2 (amongst others) is a use of variable `len`, which has been assigned in node 1. Therefore, node 2 must not be executed in parallel or before node 1.

CONTROL FLOW GRAPH (CFG)

Definition: a *control flow graph* is a directed graph $CFG = (V, E)$ with unique nodes $START$ and $STOP \in V$ such that there exists a path from $START$ to every node $x \in V$ and a path from every node to $STOP$. $START$ has no incoming arcs and $STOP$ has no outgoing arcs.

- the CFG may be cyclic (if it contains e.g. `for` or `while` loops, or `goto`s)
- a method can be converted into a CFG by splitting it up in its basic blocks, and then adding a $START$ node at the beginning of a method and a $STOP$ node at the end of the method

CONTROL FLOW GRAPH (CFG)



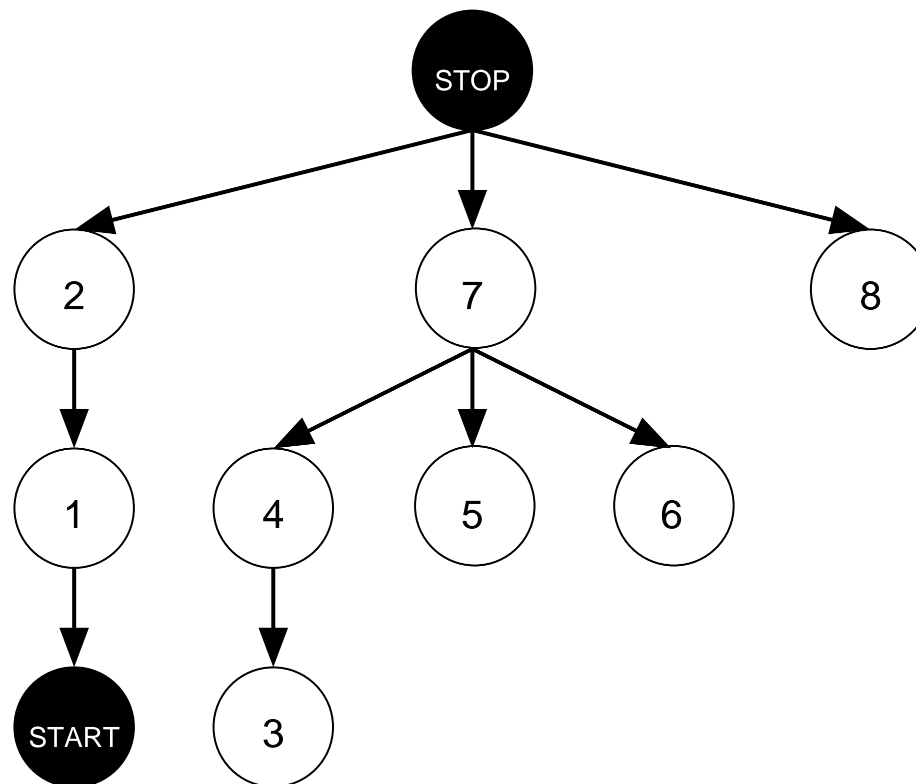
control flow graph (CFG)

POST-DOMINATOR TREE (PDT)

Definition: a *post-dominator tree* is a tree (directed acyclic graph), rooted at the $STOP$ node. If a node x is an *immediate post-dominator* of node y, then y is a direct descendant of x.

- the *post-domination relationship* $y\Delta_p x$ holds iff every path from x to $STOP$ (not including x) contains y. A node never post-dominates itself.
- the *immediate post-dominator* is the least element in the chain of post-dominators of a node x
- since $STOP\Delta_p x$ holds, the set of post-dominators for every node x is nonempty (except $x = STOP$)
- all nodes except $STOP$ have a unique immediate post-dominator

POST-DOMINATOR TREE (PDT)



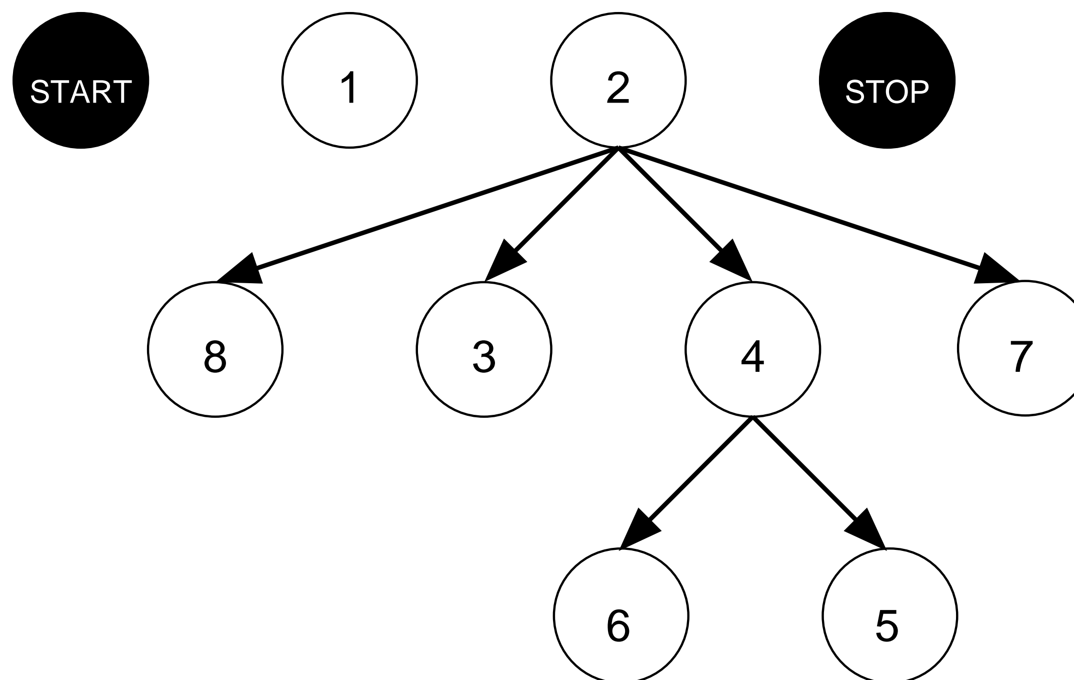
post−dominator tree (PT)

CONTROL DEPENDENCE GRAPH (CDG)

Definition: a *control dependence graph* of a $CFG = (V, E)$ is a directed labelled graph $CDG = (CV, CE)$ such that $CV = V$ and edges $(x, y) \in CE$ are labelled $x - a$ iff x is *control dependent* on y ($x\delta_c y$) by the branching decision $x - a$.

- node y is *control dependent* $y\delta_c x$ on node x with label $x - a$ iff $y \not\Delta_p x$ and it exists a nonnull path $P =< x, a, \cdots, y >$, such that for any $z \in P$ (excluding x and y) $y\Delta_p z$ holds

CONTROL DEPENDENCE GRAPH (CDG)



control dependence graph (CDG)

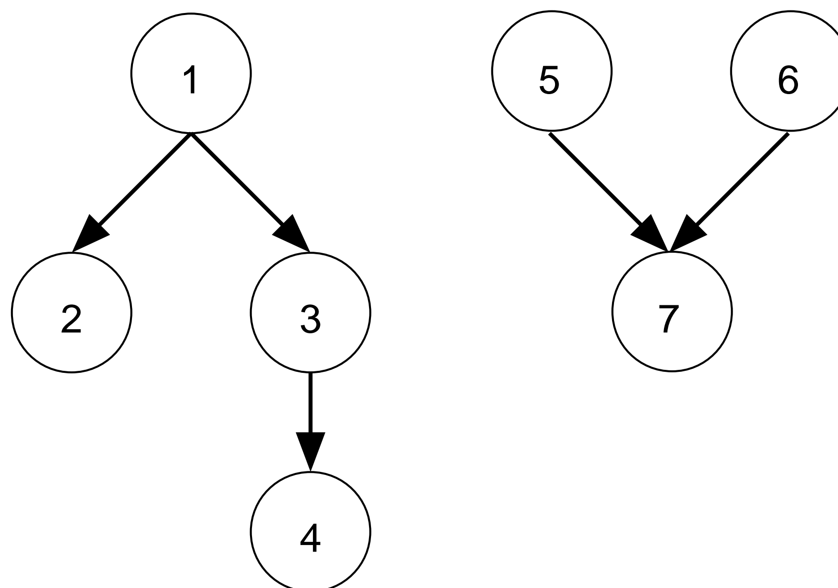
DATA DEPENDENCE GRAPH (DDG)

Definition: a *data dependence graph* of a $CFG = (V, E)$ is a directed labelled graph $DDG = (DV, DE)$ such that $DV = V$ and $(x, y) \in DE$ if $x\delta_d y$.

- if x and y conflict with each other, then y is *data dependent* on x, denoted by $x\delta_d y$.
- therefore, exactly one of the following can occur between two distinct nodes x and y:

1. y is reachable from x ($x\delta_d y$)
2. x is reachable from y ($y\delta_d x$)
3. x is not reachable from y, and y is not reachable from x (conflict does not matter since x and y will not be executed together)

DATA DEPENDENCE GRAPH (DDG)



data dependence graph (DDG)

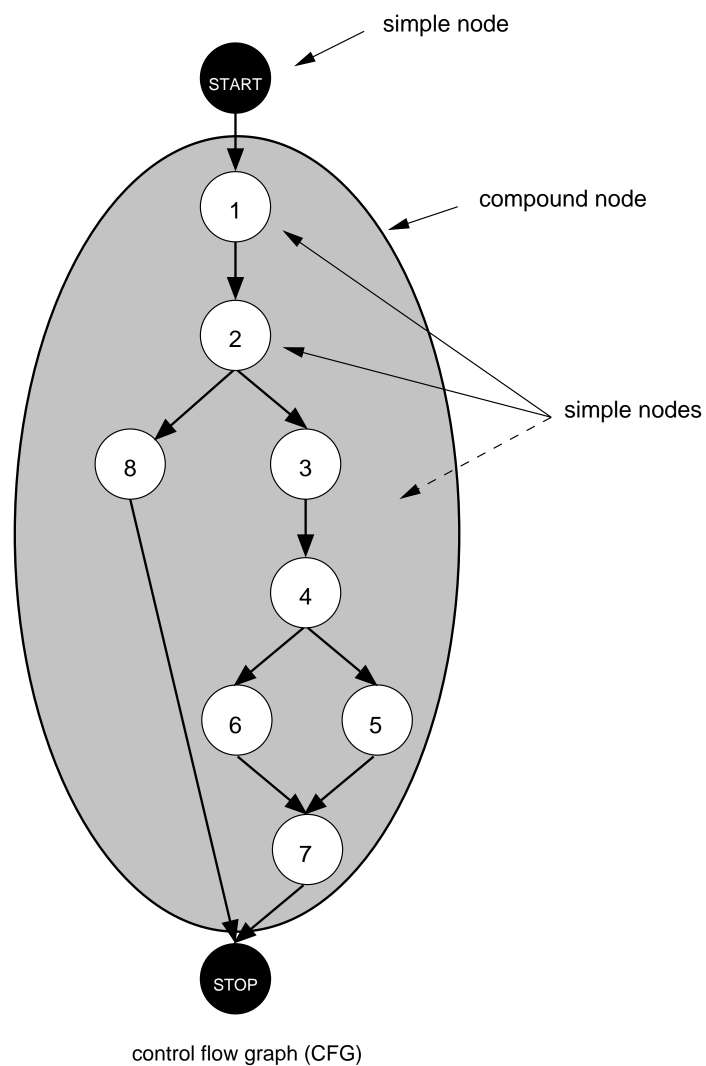
HIERARCHICAL TASK GRAPH (HTG)

Definition: a *hierarchical task graph* is a directed acyclic graph $HTG = (HV, HE)$ with unique nodes $START$ and $STOP \in HV$ such that there exists a path from $START$ to every node $x \in HV$ and a path from every node to $STOP$. $START$ has no incoming arcs and $STOP$ has no outgoing arcs.

The HTG is composed of the following nodes:

- *simple node*: node representing a task which has no subtasks
- *compound node*: node representing a task that consists of other tasks
- *loop node*: node representing a task that is a loop whose iteration body is an HTG

HIERARCHICAL TASK GRAPH (HTG)



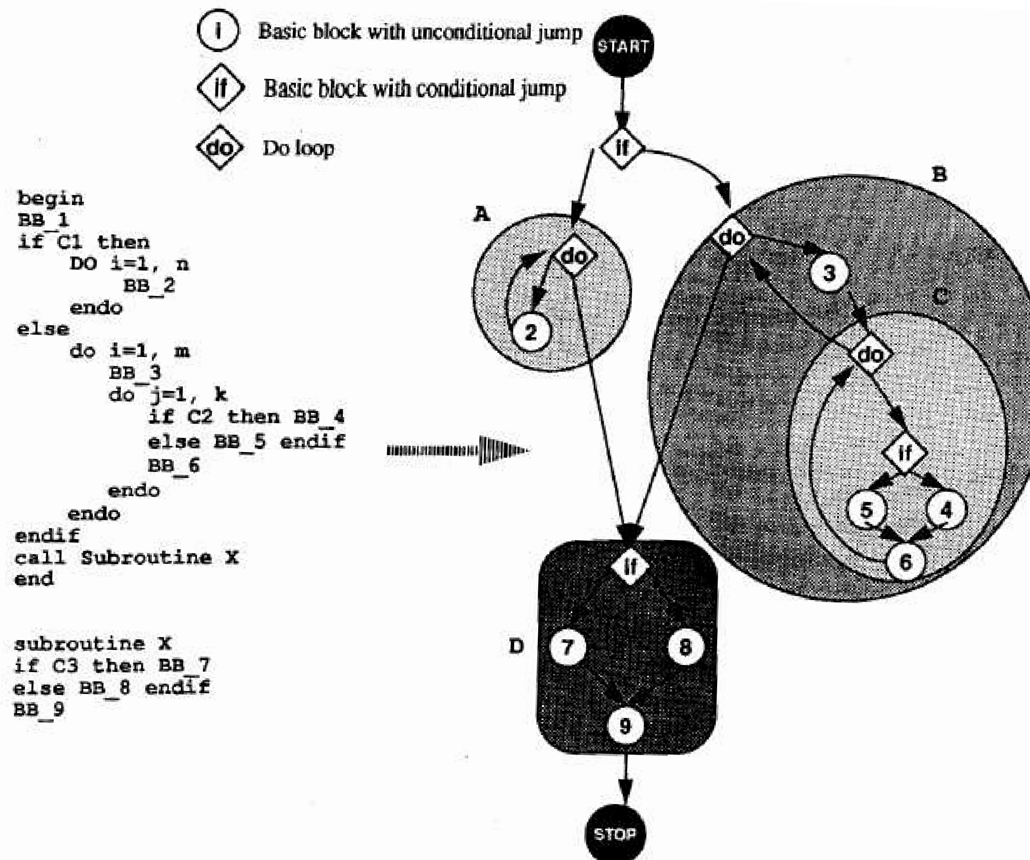
control flow graph (CFG)

EXECUTION CONDITIONS

Definition: an *execution condition* c_x for a node x is a boolean expression, which, when $\mathtt{true}$, denotes that node x is ready to be executed.

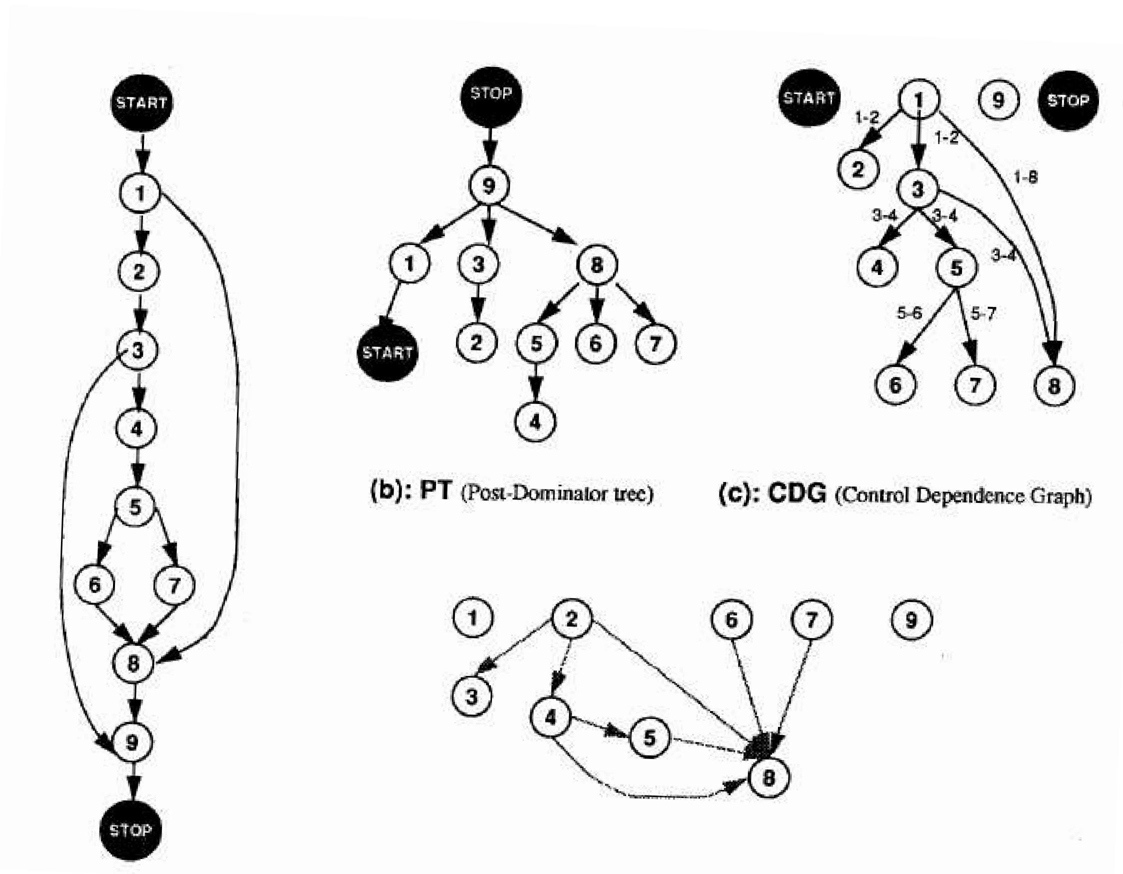
The derivation of the (optimised) execution conditions is easy to see. Since e.g. node 7 is control dependent on node 2, and data dependent on nodes 5 and 6, the execution condition therefore is $(4 - 5 \wedge 5) \vee (4 - 6 \wedge 6)$, i.e. node 5 must have finished execution if branch $4 - 5$ was taken or node 6 must have finished execution if branch $4 - 6$ was taken.

CODE TO HTG



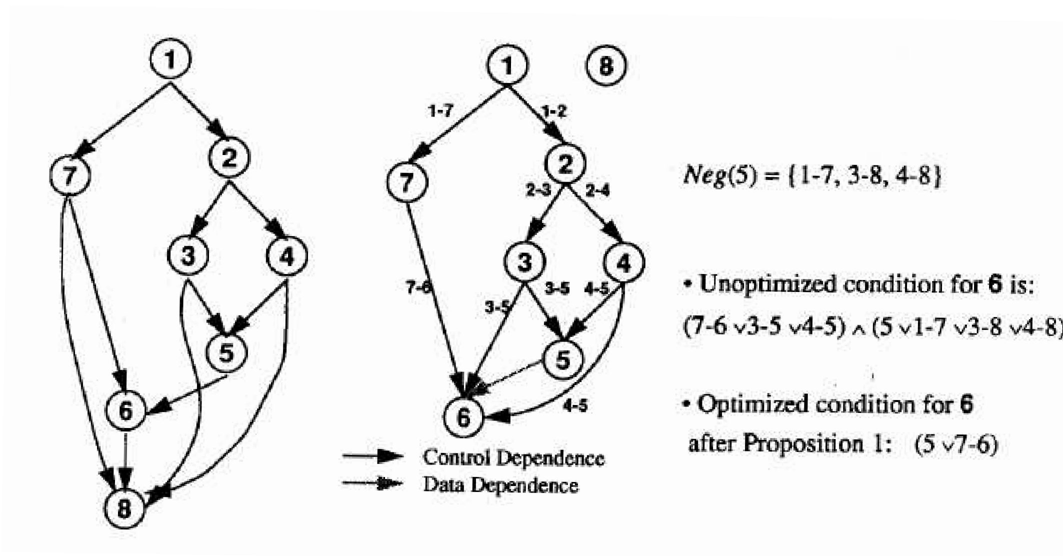
fi gure: Automatic Extraction of Functional Parallelism from Ordinary Programs, M. Girkar, C. D. Polychronopoulos, p. 169.

CFG, PDT, CDG, DDP



fi gure: Automatic Extraction of Functional Parallelism from Ordinary Programs, M. Girkar, C. D. Polychronopoulos, p. 171.

EXECUTION CONDITIONS



fi gure: Automatic Extraction of Functional Parallelism from Ordinary Programs, M. Girkar, C. D. Polychronopoulos, p. 176.

FUTURE PROSPECTS

- Using the HTG as an intermediate representation between compiler front- and backend.
- Developing an efficient scheduler to execute to switch between the different tasks.
- The fine grained execution model might even get further optimised by building processors which support fast task switches, and have caches which can deal with lots of target changes.

Questions?