Abstract

Data flow analysis is the prerequisite of performing optimizations such as code motion of partial redundant expressions on imperative sequential programs. To apply these transformations to parallel imperative programs, the notion of data flow must be extended to concurrent programs. The additional parallel source language features are: shared memory and nested parallel statements (PAR). The underlying interleaving semantics of the concurrently-executed processes result in the so-called state space explosion which on first appearance prevents the computation of the meet over all path solution needed for data flow analysis.

For the class of bit-vector data flow problems we can show that for the computation of the meet over all path solution, not all interleavings are needed. Based on that, we can give simple data flow equations representing the data flow effects of the PAR statement. The definition of a parallel control flow graph leads to an efficient extension of Kill-dal's algorithm to compute the data flow of a concurrent program. The time complexity is the same as for analyzing a comparable sequential program.

Keywords: Data flow analysis, parallel languages.

1 Introduction and Motivation

The key tool for attacking the Grand Challenge Problems is parallelism. Parallel hardware is becoming more available and cheaper, but to program these devices is still a difficult task. Hence high-level programming languages are needed which have enough expressive power to implement parallel algorithms. But as usual with high-level languages, when translating them to machine code, some inefficiencies are introduced by the compiler. Therefore compilers have to perform so-called optimizations which improve the program. We may distinguish broadly between two kinds:

- transformations performed on the input language level such as mapping of SIMD programs to MIMD machines, removal of unnecessary synchronization and communication, clustering of processes, data placement etc. and
- transformations performed on the machine language level, such as common subexpression elimination, constant folding, dead code elimination, code motion etc.

For the rest of this paper we have only the second kind of optimizing transformations in mind.

Optimizing a program requires analyzing it, and this is often done by solving data flow equations for the program. Traditional data flow analysis (DFA) methods [10] are designed for sequential programs. Hence they may fail when applied to the control flow of parallel programs as shown in [14]. We give another example showing the problems when "low level" optimizations such as instruction scheduling are performed on parallel programs.

1.1 The Problem

Data flow analysis is more or less the estimation of the effects caused by program statements. This estimation is based on two things: an abstraction of the information needed as prerequisite for the optimizing transformation, and the propagation of the information along the statements of the source program. The information is usually represented by the elements of a semi-lattice[1]. The effect of a single program statement is then a function over these semi-lattice values. One execution of a program (up to the point of consideration) represents an execution path. The propagation is modeled by applying these functions in the order given by the statements of such an execution path. Since we are looking for the "worst case information" (only this guarantees that the transformation is correct for all execution paths leading to this program point) we have to consider the Meet Over all Paths of these information. [11] formalized this idea and gave an efficient algorithm to compute the data flow information for all points of a program.

The data flow information of two statement sequences, without any branches, executed concurrently, is given by the meet of the information of all interleavings[2] of the two sequences. It is clear that this may lead to a "state space explosion" [5], which makes it, on a very first view, intractable to compute the data flow information implied by a PAR statement.

This work has been funded by the ESPRIT project COMPARE, contract number 5399.

---

[1] E.g. boolean values for "the information is present or not".
[2] The topological sorting of the simple statements from both sequences.
1.2 Contribution of this Work

Hence we have to ask:

- which data flow information is valid after the PAR statement, and
- which information is valid before each statement in a process body?

The main contributions of our paper are:

- The lattice-theoretic based data flow framework is extended to cope with parallel programs. The proposed extension is valid only for the large class of one-bit (also known as bit-vector) data flow problems. They are based on a two-element semi-lattice.
- Simple bit-vector data flow equations are derived representing the data flow effects of the PAR statement.
- The Parallel Control Flow Graph is defined and used as a base for an extension of the well known and efficient iterative data flow analysis algorithm.

Based on these results, data flow analysis of parallel programs is possible and efficient. Then traditional optimizations may be applied to these programs without any restriction.

1.3 Related Work

Current approaches in analyzing the data flow of parallel programs have either a restricted model of shared memory, or even disallow it.

[17] investigates the data flow of communicating processes, but these do not share memory: processes communicate solely through synchronous channels. [19] describes an efficient method of computing the Static Single Assignment Form for explicitly parallel programs with wait clauses. The parallel sections must be data-independent, except where explicit synchronization is used. The same is true for [20, 26] who introduce a Parallel Control Flow Graph and the Parallel Precedence Graph which may form the basis of concrete optimizing algorithms.

[6] presents data flow equations for parallel programs, both with and without synchronization. But this work is restricted to PCF FORTRAN programs, which means that access to shared variables is done only at synchronization points. For process start and process end a copy in/copy out semantics is assumed. An intuitive but not formal derivation of the data flow equations is given, which solve only the matching definition problem.

[4] extends abstract interpretation to cope with communicating sequential processes. The problem there is that the resulting state space explodes. [3] applies abstract interpretation to analysis of parallel programs, but bases the semantics on a labeled transition system. [5] attacks this problem using the stubborn set theory [21] which decreases the state set using some heuristics. Hence the analysis is accurate for some examples, and less accurate for others.

[23] presents the basic idea of how the number of interleaveings may be reduced; the parallel program is represented by its structure tree. [25] uses ideas of this report to prove the Hierarchical Coincidence Theorem which is based on a functional representation [18] of the problem. The full version of this paper is available as [24].

2 Machine Model

We assume that other phases of the compiler have done the more "high-level" transformations already, and hence our investigation is based on an imperative language with explicit control flow parallelism, dynamic process creation, and shared memory. As a computing model we assume a MIMD (multiple instruction, multiple data) system, where each process is executed on a separate logical processor. Each processor runs independently of each other and has its own set of registers, which are invisible for other processors. All processors access a shared memory. The access to a single memory cell is atomic, i.e. at a given time only one process may read or write a given cell. We assume an interleaving semantics for the execution of the program with respect to the memory accesses.

3 The Sample Language

A simple imperative language will be used in this paper, which has loops, conditional statements, and a statement to execute other statements in parallel (explicit control flow parallelism). Replicators allow dynamic process creation, and processes share memory.

The PAR statement executes all processes specified by ProcessBody in parallel and independently. The process executing a PAR statement is suspended until all child processes have terminated. A ProcessBody is a list of statements, which may be replicated. That is: max(UppBound - LowBound + 1, 0) processes are forked which all execute the statements following the replicator. Each replicated process gets its private copy of the replicator variable var, which has in each replicated process a unique value in the range [LowBound .. UpBound]. Replicated processes are also called asynchronous for-all loops in other languages. All variables can be accessed in each process. No automatic synchronization is done for the access.

Stmt ::= PAR ProcessBody/"/1" END.
ProcessBody ::= [Replicator] Stmt/"/;".
Replicator ::= "/["/var/"="/LowBound TO UpBound"]".
LowBound ::= Expr.
UppBound ::= Expr.
Expr ::= usual expressions.
X/Y is a list of X's separated by a Y. [X] stands for an optional X part.

4 Lattice-Theoretic Background of Data Flow Analysis

This section gives the lattice-theoretic background of data flow analysis and follows [9]. It may be skipped by the reader familiar with the notation.

The source program under consideration is represented as a (sequential) control flow graph:

```
Definition 1 A control flow graph is a triple G = (N, E, no), where N is a finite set of nodes (which contains a list of simple statements, such as assignments). E ⊆ N × N is a set of ordered edges between these nodes and no the unique initial node.
```

\(^3\)A processor may be implemented via a time sharing system.
\(^6\)PAR statements may be nested.
\(^7\)Section 8 defines the parallel version of a control flow graph.
A path from \(n_1\) to \(n_k\) is a sequence of nodes \(n_1, n_2, \ldots, n_k\), such that for \(1 \leq i < k\) all edges \((n_i, n_{i+1}) \in E\). Such a path has length \(k\).

For a node \(n\) \(\text{pred}[n] \cup \text{succ}[n]\) is the set of predecessors (successors) defined as: \(\text{pred}[n] = \{n' : (n', n) \in E\}\), and \(\text{succ}[n] = \{n' : (n, n') \in E\}\).

All nodes of a control flow graph are reachable from the initial node, i.e., there is a path from \(n_0\) to each node \(n\). path\(\{n\}\) is the set of all paths from the initial node to \(n\). path\(\{n\}\) is the set of paths from \(n_0\) up to all predecessors of \(n\).

The data flow information is represented as a semi-lattice:

**Definition 2** A semi-lattice \((L, \sqcap)\) is a set \(L\) with a binary meet operation \(\sqcap\) such that for all \(a, b, c \in L\) the following holds:

\[
\begin{align*}
a \sqcap a &= a & \text{Idempotent} \\
a \sqcap b &= b \sqcap a & \text{Commutative} \\
a \sqcap (b \sqcap c) &= (a \sqcap b) \sqcap c & \text{Associative}
\end{align*}
\]

For two elements \(a, b \in L\), we define:

\[
\begin{align*}
     a \sqcap b &\iff a \sqcap b = a \\
     a \sqcup b &\iff a \sqcap b = a \land a \neq b \\
     a \sqsubseteq b &\iff a \sqcap b = b \\
     a \sqsubset b &\iff a \sqcap b = b \land a \neq b
\end{align*}
\]

(L, \(\sqcap\)) has a zero-element \(\bot\) (bottom), if \(\forall x \in L : x \sqcap \bot = \bot\) and a one-element \(\top\) (top), if \(\forall x \in L : x \sqcap \top = x\). From now on we assume that \((L, \sqcap)\) has a zero-element, but not necessary a one-element. We can extend the \(\sqcap\) operation:

\[
\bigwedge_{i=1}^{n} x_i = x_1 \sqcap x_2 \sqcap \ldots \sqcap x_n \text{ with } x \in \mathbb{G}
\]

A sequence of elements \(x_1, x_2, \ldots, x_n\) of \(L\) is called a chain, if \(\forall 1 \leq i < n : x_i \sqsubseteq x_{i+1}\). \((L, \sqcap)\) is called bounded if for all \(x \in L\) there is a constant \(b_x\), such that each chain starting with \(x\) has length at most \(b_x\). If \((L, \sqcap)\) is bounded, we can define for each countably infinite set \(S\) of elements of \(L\):

\[
\bigwedge_{i \in S} x = \lim_{n \to \infty} \bigwedge_{i=1}^{n} x_i.
\]

If \((L, \sqcap)\) is bounded, there is a number \(m\) with \(\bigwedge_{x \in S} x = \bigwedge_{i=1}^{m} x_i\).

How a simple program statement transforms, by its symbolic execution, the data flow information valid before its execution, is described by a transfer function:

**Definition 3** Let \((L, \sqcap)\) be a bounded semi-lattice. A set \(F\) of functions on \(L\) is called a monotone data flow space associated with \(L\), if the conditions \([M1] - [M4]\) are satisfied. If also \([M5]\) is valid, it is called a distributive data flow space associated with \(L\).

\[\text{[M1]}\]
All functions \(f \in F\) are monotone:
\[
\forall x, y \in L : f(x \sqcap y) \sqsubseteq f(x) \sqcap f(y).
\]
This is equivalent to:
\[
\forall x, y \in L : x \sqcap y = f(x) \sqcap f(y).
\]

\[\text{[M2]}\]
There is an identity function \(id \in F\) with:
\[
\forall x \in L : id(x) = x.
\]

\[\text{[M3]}\]
\(F\) closed under composition:
\[
\forall f, g \in F : f \circ g \in F.
\]

\[\text{[M4]}\]
\(L\) is the closure of \(\\{(\bot)\}\) with respect to the \(\sqcap\) operation and application of functions in \(F\).

\[\text{[M5]}\]
All functions are distributive:
\[
\forall x, y \in L : f(x \sqcap y) = f(x) \sqcap f(y).
\]

A monotone data flow framework is defined as:

**Definition 4** A monotone data flow framework is a triple \(D = (L, \sqcap, F)\) where \((L, \sqcap)\) is a bounded semi-lattice and \(F\) is a monotone function space associated with \(L\). An instance of a monotone data flow framework is a pair \(I = (G, M)\) where \(G = (N, E, n_0)\) is a control flow graph and \(M : N \to F\) is a labeling which maps each node from \(N\) onto a function of \(F\).

If \(D\) is a distributive data flow framework, \(D\) is called a distributive data flow framework.

The "maximal (or worst case) information reaching a program statement" is given by the following

**Definition 5**

\[
\bigwedge_{p \in \text{path}(\mathbb{G})} f_p(\bot)
\]

is called the meet over all path and represents the "maximal (or worst case)" information reaching a node \(n\) of the program. \(f_p\) is the transfer function of the path \(p\) (see below).

5 Properties of Some DFA Frameworks

First we give some properties of bit-vector data flow frameworks \(D^b\) which is then generalized to \(D^0\). At the end of this section then we apply these results to the transfer functions of statements and statement sequences.

5.1 Properties of the Boolean Semi-Lattice

Since we restrict our investigation to the class of bit-vector data flow problems, we give some general results for the boolean semi-lattice:

**Definition 6** The data flow information of an entity is a value of the set \(B = \{\top, \bot\}\). For a given binary meet operation \(\land\), \(\bot \sqsubseteq \top\) must hold.

Observation 1: Obviously there are only two different binary operations which can be the meet operation of a semi-lattice: \(\land\) (boolean and) and \(\lor\) (boolean or)\(^a\). They are given as shown in Table 1.

\[
\begin{array}{c|c|c|c|c|c|c|c}
\hline
& \bot & \top & \bot & \top & \bot & \top & \bot & \top \\
\hline
\land & \bot & \bot & \bot & \bot & \bot & \bot & \top & \top \\
\lor & \bot & \top & \top & \top & \bot & \top & \bot & \top \\
\hline
\end{array}
\]

Table 1: The boolean meet operations.

Observation 2: There are only four functions \(B \to B\): the two constant functions, the identity, and negation.

\[
\begin{align*}
\text{use} & : \bot \mapsto \top, \top \mapsto \top \\
\text{modify} & : \bot \mapsto \bot, \top \mapsto \bot \\
\text{identity} & : \bot \mapsto \bot, \top \mapsto \top \\
\text{negation} & : \bot \mapsto \top, \top \mapsto \bot
\end{align*}
\]

\(^a\)The other possible 14 binary operations over \(B\) do not have the required properties, even if some may have an interesting interpretation such as xor: the data flow information is valid, if it is valid in exact one path.
Obviously, the negation function is not monotone and not distributive. The other three are both monotone and distributive. Often, the constant functions are interpreted, respectively as use, which generates or uses some information, and modify, which modifies or invalidates it.

We finish this section with:

**Observation 3:** For any two-element semi-lattice \((B, \sqcap)\), there is exactly one monotone function space \(\mathcal{F}^B = \{\text{id}, \sqcap\}\) associated with \(B\). It is also distributive: \(\mathcal{D}^B = (B, \sqsubseteq, \mathcal{F}^B)\) is called the one bit data flow analysis framework. There are only two interpretations of the meet operation: the boolean \(\land\) and \(\lor\) operation, respectively. Their DFA interpretation is: the information valid before a node \(n\) must be valid on all \((\land)\) (at least one \((\lor)\)) path reaching \(n\).

### 5.2 Properties of the Function Space \(\mathcal{F}^C\)

A slight generalization of the \(\mathcal{D}^B\) DFA framework is \(\mathcal{D}^C\), for which the following lemma obviously holds:

**Lemma 1** Let \((C, \sqsubset)\) be a bounded semi-lattice, and \(\mathcal{F}^C\) a set of functions \(C \rightarrow C\), such that \(\mathcal{F}^C\) contains only the identity function \(\text{id}\) and for each element \(c\) of \(C\) its constant function \(\text{const}_c\) with \(\forall x \in C: \text{const}_c(x) = c\). Then \(\mathcal{F}^C\) is a distributive function space, and the corresponding constant data flow framework \(\mathcal{D}^C\) is distributive.

We consider now “composition chains” of functions \(f_1 \circ f_2 \circ \cdots \circ f_n(x)\) and show some properties. The following lemmata help us to compute the data flow information which is valid after a \(\text{PAR}\) statement: if for such a composition chain a predicate \(P\) holds for all \(x \in C\), then there is a function \(f_i\) in it, such that \(P\) holds for all \(x\), and all “following” \(f_j, j > i\) do not invalidate the predicate.

**Lemma 2** Let \(f_1, \ldots, f_n \in \mathcal{F}^C\) and \(P\) a predicate over \(C\).
\[
\forall x \in C: P[f_n \circ f_{n-1} \circ \cdots \circ f_1(x)] \iff \exists 1 \leq i \leq n: \forall x \in C: P[f_i(x)] \land \forall i < j \leq n: \forall x \in C: P[f_j(x)]
\]

The lemma can be proved using induction over the number \(n\) of functions in the composition chain. The next lemma is a corollary of the previous one:

**Lemma 3** Let \(f_1, \ldots, f_n \in \mathcal{F}^C\). Then: \(\exists 1 \leq i \leq n: \forall x \in C: \text{def}_{f_i} \circ f_1 \circ \cdots \circ f_n(x) = f_i(x)\) and \(\forall i < j \leq n: f_j = \text{id}\)

The next lemmata state that under some circumstances the order of the functions of a composition chain may be changed and still return the same value.

**Lemma 4** Let \(f_1, \ldots, f_n \in \mathcal{F}^C\). From \(\exists 1 \leq i \leq n: \forall x \in C: f_n \circ f_{n-1} \circ \cdots \circ f_1(x) = f_i(x)\), it follows that for an arbitrary permutation \((k_1, \ldots, k_n)\) of the numbers \(1, \ldots, i - 1\) holds:
\[
f_n \circ f_{n-1} \circ \cdots \circ f_1(x) = f_n \circ f_{n-1} \circ \cdots \circ f_i \circ f_{i+1} \circ \cdots \circ f_n(x)
\]

**Proof (Lemma 4)** If \(f_i\) is a constant function, it returns the same result for all arguments. Hence the order of the functions which form the argument of \(f_i\) is not important. If \(f_i\) is the identity function, all other functions must also be the identity function, otherwise \(f_n \circ f_{n-1} \circ \cdots \circ f_1(x) = f_i(x)\) would not hold for all \(x\).  

The following lemmata state some properties of the value of composition chains. They answer the question which information is valid before a statement inside a \(\text{PAR}\) statement.

**Lemma 5** Let \(f, g \in \mathcal{F}^C\). Then for arbitrary \(x \in C:\)
\[
x \sqcap f(x) \sqcap g(f(x)) = x \sqcap f(x) \sqcap g(x)
\]

**Proof (Lemma 5)** If \(g = \text{id}\), then: \(x \sqcap f(x) \sqcap g(f(x)) = x \sqcap f(x) \sqcap g(x)\).

Using induction over the number \(n\) of functions in the composition chain we can conclude:

**Lemma 6** Let \(f_1, \ldots, f_n \in \mathcal{F}^C\). Then for arbitrary \(x \in C: x \sqcap f_1(x) \sqcup f_2 \circ f_1(x) \sqcup \ldots \sqcup f_n \circ f_{n-1} \circ \cdots \circ f_1(x) = x \sqcap f_1(x) \sqcup f_2 \circ f_1(x) \sqcup \ldots \sqcup f_n(x)\)

### 5.3 Properties of Statement Sequences and Composition Chains

From now on, we consider only \(\mathcal{D}^C\). We now connect the functions to a statement sequence, which represents an execution path. Then we state properties of the function composition chains, if two (or more) statement sequences are executed in parallel. This is modeled by considering the set of topological sortings of the statements contained in the sequences.

**Definition 7** Let \(s_1, \ldots, s_n\) be simple statements, as assignments, which are executed in the order \(p \equiv (s_1; \ldots; s_n)\). Let \(f_n \in \mathcal{F}^C\) be the transfer function connected to the statement \(s_n\): \(f_n(x) = f_{n-1} \circ f_n \circ f_{n-1} \circ \cdots \circ f_1(x)\)

**Definition 8** Let \(s_1', \ldots, s_r'\) and \(s_1'', \ldots, s_r''\) be simple statements, which are executed in the order \(p' \equiv (s_1'; \ldots; s_r')\) and \(p'' \equiv (s_1''; \ldots; s_r'')\), respectively. \(\text{TopSorts}(p', p'')\) is the set of statement sequences which result from a topological sorting of the two sequences \(p'\) and \(p''\). For two statements \(s_i', s_i''\) no order is defined, and \(s_i'\) must be executed before \(s_i''\) (also for the statements \(s_i''\)).

**Lemma 7** Let \(s_1', \ldots, s_r'\) and \(s_1'', \ldots, s_r''\) be simple statements, which are executed in the order \(p' \equiv (s_1'; \ldots; s_r')\) and \(p'' \equiv (s_1''; \ldots; s_r'')\), and \(f_p, f_{p'} \in \mathcal{F}^C\) the functions corresponding to the statements \(s_i'\) and \(s_i''\), respectively. The following holds:
\[
\bigcap_{p \in \text{TopSorts}(p', p'')} f_p(x) = f_{p'} \cap f_{p'}(x)
\]

Where \(p \cap q\) is the concatenation of two statement sequences \(p\) and \(q\).

**Proof (Lemma 7)** We prove this lemma in several steps:

1. Let \(p \equiv (s_1; \ldots; s_n) \in \text{TopSorts}(p', p'')\). With Lemma 3 we have a \(1 \leq i \leq n\) where \(f_p(x) = f_{p'}(x)\) and \(\forall i < j \leq n: f_j = \text{id}\). If there are several such \(i\), we choose the largest one. Now \(s_i\), the statement determining the value of the path \(p\), may be contained either in \(p'\) or \(p''\).

Note: To be a monotone function space associated with \(S\), all three functions are needed.

\[f_i\] is an abbreviation of \(f_{t_i}\).
The proposition is proved, by reordering the statements of the program, this simplifies the presentation of the data flow analysis of the statement in two steps:

1. If the maximum (worst case) data flow information of some sequential programs, which is defined analogously.

2. Capital letters S denote composite statements, whereas small letters s denote simple ones.

3. Proposition: If $s_i \in p'$, then for all $x$:

   $f_p(x) = f_{p';p''}(x)$.

   Proof: The proposition is proved, by reordering the sequence $p$ stepwise: The statements $s_{k_1}, s_{k_2} \in p$ with $1 \leq k_1, k_2 < i$ may be reordered in a way that all statements $s_{k_1}, s_{k_2} \in p'$ are placed before $s_{k_1} \in p''$ and still fulfill the order constraints of $p'$ and $p''$, respectively. Lemma 4 guarantees that the value of this reordered sequence is still equal to $f_p(x)$.

   "The instructions $s_k \in p'$ with $k > i$ may also be placed before $s_i$, since $f_k = 1$.

   The statements $s_{k_i} \in p''$ with $k > i$ need not be reordered.

   Hence if $s_i \in p'$ then $f_p(x) = f_{p';p''}(x)$. Analogously, if $s_i \in p'$ the $f_p(x) = f_{p';p''}(x)$.

4. Now the statement sequences from seq' and seq'' may be reordered as shown above, while not changing their value. Hence $\prod_{p \in seq'} f_p(x) = f_{p';p''}(x)$ and $\prod_{p \in seq''} f_p(x) = f_{p';p''}(x)$.

5. And so:

   $\prod_{p \in TopSorts(p')} f_p(x) = f_{p';p''}(x)$.

6. Data Flow Analysis of the PAR Statement

   We now solve the data flow analysis problem for the PAR statement in two steps:

   1. Which information is valid before each statement in a process body, and

   2. Which data flow information is valid after the PAR statement.

   But before doing this, we need some more definitions.

6.1 Statement Traces Generated by the PAR Statement

   If the maximum (worst case) data flow information of some statements is given as the meet over all paths, the question arises: how can we compute all paths of a parallel program?

   We now consider statement traces instead of paths in the control flow graph; this simplifies the presentation of the next results.

   Definition 9 A statement trace or statement execution sequence is a list of simple statements in the order they are executed by a single run of a program. The set seq[S] is the set of all traces generated by statement S. $s \in p \in seq[S]$ means that S is a simple statement contained in sequence q.

   It is well known how the set of traces is constructed for sequential programs:

   • For the concatenation of statements $S_1; S_2$ the traces produced from $S_1$ and $S_2$ are concatenated.

   • For an IF statement, the traces generated by the THEN and ELSE parts are united.

   • For loops, all n-fold concatenations of the traces produced by the loop body are united.

   Since we assumed an interleaving semantics for the execution of the statements of our PAR statement, it is obvious that the topological sorting of the traces produced by the process bodies, determines all traces of the PAR statement.

   Definition 10 For a given trace p the set prefixes(p) is the set of all prefixes of p. It includes the empty trace and the entire trace p. prefixes(P) is the extension to a set of traces P, so that it contains all prefixes of all traces of $p \in P$. For a statement S, prefixes[S] is the abbreviation of prefixes( seq[S] ).

   Definition 7 is now extended for arbitrary (composite) statements.

   Definition 11 Let $S, S_1, \ldots, S_n$ be arbitrary (composite) statements: $f_S(x) = \prod_{p \in seq[S]} f_p(x)$.

   $f_{S_1;S_2;\ldots;S_n}(x) = f_{S_n \circ f_{S_{n-1}} \circ \ldots \circ f_{S_1}}(x)$ is the function corresponding to the statement sequence $S_1; \ldots; S_n$.

   An obvious lemma is:

   Lemma 8 Let $S_1, \ldots, S_n$ be arbitrary statements and let be $S \equiv S_1; \ldots; S_n$. Then for distributive functions $f \in F$: $f_{S_1; S_2; \ldots; S_n}(x) = f_{S}(x)$, while for monotone $f \in F$, only $f_{S_1; S_2; \ldots; S_n}(x) \subseteq f_{S}(x)$ holds.

   Definition 12 Let $s$ be a statement inside a (nested) PAR statement. sib[s] is the set of all simple statements which could possibly be executed in parallel to $s$. If s is part of a replicated process body, all statements of this process body are also contained in sib[s].

   Note that the property $s' \in sib[s]$ of $s'$ is a purely syntactical one, which can easily be determined from the source program.

6.2 Which Data Flow Information Reaches a Statement

   Let $s$ be a statement inside a PAR statement. By definition the information valid before $s$ is given by the meet over all traces reaching $s$. To answer the question which these traces are, let us examine the following example:

   PAR $s_1; s_2; \ldots; s_n$ END. $s$ may now be the first statement executed in an interleaving produced by this PAR statement.

   On the other hand, $s_1$ may be executed before $s$ in another interleaving, or $s_1; s_2$ are executed before $s$, etc. Hence the set of statements executed before $s$ is given by prefixes($\langle s_1; \ldots; s_n \rangle$). Using lemma 6 we can conclude that:

   $\prod_{p \in prefixes(\langle s_1; \ldots; s_n \rangle)} f_p(x) = x \cap \prod_{i=1}^{n} f_{s_i}(x)$. If the process body $S$ of PAR $s$ is only executed in parallel to $s$ produces more than one execution sequence, the set of execution sequences reaching $s$ is given by:

   $\cup_{p \in prefixes[seq[S]]} \text{prefixes}(p)$, and hence:

   $\prod_{p \in prefixes[seq[S]]} f_p(x) = x \cap \prod f_{s_i}(x)$ is a statement from $s \setminus f_{s_i}(x)$.

   The fact that there may be statements, which are always executed before $s$, does not influence our considerations, since the argument $x$ of the function $f_p$ reflects the
information reaching s, if no parallel statements are executed before s.

Since PAR statements may be nested, the following theorem is a consequence of the above:

**Theorem 1** Let s be a simple statement in a program, and sib[s] the set of simple statements possibly executed in parallel to s. Then for $D^2$ data flow problems, the following information is valid before s:

$$x \cap \bigcap_{t \in \text{sib}[s]} f_t(x),$$

where x is the information valid before the PAR statement.

If we use the following definition, we can restate the theorem so that it is easier to use as a base for an implementation.

**Definition 13** For statement s, we define $\text{in}^+[s]$ as the information reaching s on a “sequential” trace from the program entry. That is, none of siblings of s are executed before s. And let $\text{in}^+[s]$ be the information reaching s if all possible traces are considered.

If a function $f_t \in \mathcal{F}^t$, $t \in \text{sib}[s]$ is a constant function, then the value of $f_t(x)$ is independent of x: $f_t(x) = f_t$. Otherwise $f_t$ is the identity, and $f_t(x) = x$. Then $\text{in}^+[s]$ is simply the value of x, and the Theorem 1 can be restated as:

$$\text{in}^+[s] = \text{in}^+[s] \cap \bigcap_{t \in \text{sib}[s]} f_t$$

(1)

### 6.3 Which Information is Valid After the PAR Statement

Lemma 7 is the cornerstone for the following theorems. After extending it to sets of paths we obtain:

**Theorem 2** For the $D^2$ data flow problems and the PAR $S_k | S_l \text{ END}$ statement the following holds:

$$\bigcap_{p \in \text{TopSort}[S_k: S_l]} f_p(x) = \bigcap_{p \in \text{TopSort}[S_k]} f_p(x) \cap \bigcap_{p \in \text{TopSort}[S_l]} f_p(x).$$

To extend this theorem for PAR statements with more than two process words, we define:

**Definition 14** The set of simple permutations $\text{s_perm}$ of the numbers $1, \ldots, n$ is given by: $\text{s_perm}(1, n) = x_e f$

$$\{(1, 2, \ldots, n, 1), (n, 1, 2, \ldots, n-1, 1), (1, 2, \ldots, n-1, 2, \ldots, 1), (1, 2, 3, \ldots, n, n-1)\}.$$ That is, the $i^{th}$ number is exchanged with the last number in the sequence: $1, 2, \ldots, n$. $i \equiv (i_1, i_2, \ldots, i_n)$ denotes an element of this set. Note that $\text{s_perm}(1, n)$ has only n elements.

If we have a closer look at the proof of lemma 7, we see that the order of the statements $s_k$, with $k < n$ has no influence on that result. Hence we have the following lemma:

**Lemma 9** Let $p_1, \ldots, p_n$ statement sequences of simple statements. Then:

$$\bigcap_{p \in \text{TopSort}(p_1, \ldots, p_n)} f_p(x) = \bigcap_{\tau \in \text{s_perm}(1, n)} f_{p_1 \tau \ldots \tau p_n}(x)$$

Note that any other permutation would serve, as long as each statement sequence appears at least once at the end. Hence we have the following theorem:

**Theorem 3** For the $D^2$ data flow problems and the PAR $S_1 | \ldots | S_n \text{ END}$ statement:

$$f_p(x) = \bigcap_{p \in \text{TopSort}[S_1: S_n]} f_p(x)$$

It is easy to see that the result is not changed by a replicated process body $\text{var} := 1 \text{ mod } \text{upb} S$ with $\text{upb} - 1 \text{ mod } \text{upb} > 0$, since for the result only $S$ is important. If $\text{upb} - 1 \text{ mod } \text{upb} > 0$, then it may happen that $S$ is never executed, and the theorem should be adjusted accordingly. We will see in the implementation section 8, how this could easily be done.

### 7 Bit-Vector Implementation

From now on, we consider only the $D^2$ data flow analysis framework.

Until now we considered only one program entity, such as a single program variable or a single expression. When implementing data flow analysis, usually all entities are considered at the same time. Hence we are dealing with sets of informations, valid at a program point. Each entity is coded by a number $1, \ldots, |\text{entities}|$. For the class of one-bit data flow problems $D^2$ there is a quite efficient implementation of sets: the bit-vector.

**Definition 15** A bit-vector is the characteristic function vec of a finite set of of object numbers $1, \ldots, n = |\text{entities}|$, vec: $\{1, \ldots, n\} \rightarrow \{\text{TRUE}, \text{FALSE}\}$ with $1 \leq n$. Usually vec(i) is denoted as vec[i].

The set operations $\cup, \cap, -$ are defined for bit-vectors, by element wise application of the boolean operations $\lor, \land, -. \text{ The set difference is defined by } a \rightarrow b = a \land \neg b.$

The empty set $\emptyset$ is represented by the bit-vector, in which all values are FALSE.

Usually for each statement and basic block the following four sets are defined [1]: $\text{gen}, \text{kill}, \text{in}$ and $\text{out}$. $\text{gen}(\text{kill})$ is the set of information generated (invalidated) by this statement/basic block. $\text{in}$ is the set of information valid immediately before execution of this statement and $\text{out}$ the information valid immediately after that point.

As seen before, the $\cap$ operation is either set union or set intersection. The DFA problems using set union (intersection) as meet operations are called may problems (must problems), since the information must reach a given program point on at least one (on all) paths leading to that point.

We define now the four sets in terms of the transfer functions. Each statement s has a separate transfer function for each entity e, denoted by $f^s_e$.

**Definition 16** Let s be a statement, $\text{gen}_e[s] = \text{TRUE}$ iff $f^s_e = \text{u}$ (use), $\text{kill}_e[s] = \text{TRUE}$ iff $f^s_e = \text{m}$ (modify), $\text{in}_e[s] = \text{TRUE}$ iff $\bigcap_{p \in \text{TopSort}[s]} f^p_e(\bot) = \top$ and $\text{out}_e[s] = \text{TRUE}$ iff $\bigcap_{p \in \text{TopSort}[s]} f^p_e(\bot) = \top$. Obviously: $\text{gen}_e \cap \text{kill}_e = \emptyset$.
We will first formulate the result of Theorem 1 in the form of Equation 1 using the DFA sets. Note that we are now using $D^P$:

In the equation $in^P[e] = in^P_1[e] \cap \bigcap_{f \in \text{DF A}} f_t$, all $f_t$ are constant functions. The right hand side is $T$, iff both parts of it evaluate to $T$. The following equivalences hold:

$\forall t \in \text{DFA} : f_t^* = T \iff \bigwedge_{f_t} f_t^* = T$ 
$\exists t \in \text{DFA} : f_t^* = \bot \iff \bigvee_{f_t} f_t^* = \bot$

Hence the following theorem can be given:

We now distinguish:

$\bigwedge_{f_t} f_t^* = T \iff \bigvee_{f_t} f_t^* = \bot \iff \forall t \in \text{DFA} : f_t^* = \bot \iff \exists t \in \text{DFA} : f_t^* = T$

And we can state the following theorem:

**Theorem 4** The information of a one-bit DFA problem reaching a statement $S$ can be computed by:

$in^P_S = \begin{cases} 
in^P_1 - \bigcup_{f \in \text{DFA}} \text{kill}_t & \text{for } \bigwedge \text{with } in^P, \text{kill}_t = T = \text{TRUE} \\
in^P_1 \cup \bigcup_{f \in \text{DFA}} \text{gen}_t & \text{for } \bigvee \text{with } in^P, \text{kill}_t = T = \text{FALSE} \\
\end{cases}$ (so is the first statement of the program).

The next step is the adaptation of Theorem 3. Before doing so, we restate the equations for in and out in the sequential case, as they may be found e.g. in [1]:

For a simple statement the relation is given by:

$\text{out}_S = \text{gen}_S \cup \text{in}_S - \text{kill}_S$  \(2\)

For sequential composition of statements $S \equiv S_1; S_2$:

$\text{out}_S = \text{out}_2 \cup \text{out}_1 - \text{kill}_2$  \(3\)

For branches in a sequential program:

$\begin{align*}
in_S &= \bigcap_{f \in \text{DFA}} \text{out}_p \\
\text{out}_S &= \text{gen}_S \cup \text{in}_S - \text{kill}_S
\end{align*}$  \(4\)

Using these equations we can compute the right hand side of the equation from Theorem 3: $\bigcap_{f \in \text{DFA}} (1^{x_{1}}) \bigcup \bigcup_{i=1}^{n} (\text{gen}_i - \text{kill}_i)$. We again have to distinguish between if and must problems. Hence the following theorem can be given:

**Theorem 5** The information $\text{out}_S$ of a one-bit DFA problem valid after the PAR $S_i | \ldots | S_i$ EMD statement $S$ can be computed as:

$\text{out}_S = \begin{cases} 
(\bigcup_{i=1}^{n} \text{gen}_i) - (\bigcup_{i=1}^{n} \text{kill}_i) \cup (\text{in}_S - (\bigcup_{i=1}^{n} \text{kill}_i) & \text{for } \bigwedge \\
(\bigcup_{i=1}^{n} \text{gen}_i) \cup (\text{in}_S - (\bigcup_{i=1}^{n} \text{kill}_i) & \text{for } \bigvee
\end{cases}$

where $\text{in}_S$ is the information valid before the PAR statement, and $\text{gen}_i$ and $\text{kill}_i$ are the sets corresponding to the process bodies $S_i$.

Now we have to determine which information is generated and invalidated by a PAR statement as a whole. Again we start with the sequential composition of statements (following [1]): For $S \equiv S_1; S_2$ we have:

$\text{gen}_S = \text{gen}_S \cup \text{gen}_1 - \text{kill}_2$  \(5\)

$\text{kill}_S = \text{kill}_1 \cup \text{kill}_2 - \text{gen}_2$

Hence we can follow the same arguments as for out PAR, with the simplification that the term $in = (\text{kill} \ldots)$ does not exist. We note here that if the “In-Out-problem” is a may-problem, then the “Gen-Problem” is a may-problem too, while the corresponding “Kill-problem” is a must-problem, and vice versa, if the In-Out-problem is a must-problem. The next theorem can be stated:

**Theorem 6** The information of a one-bit DFA problem generated and invalidated by a PAR $S_1 | \ldots | S_n$ EMD statement $S$ can be computed as:

$\text{In-Out-problem } \bigwedge: \text{gen}_S = \bigcup_{i=1}^{n} \text{gen}_i - \bigcup_{i=1}^{n} \text{kill}_i$  \(6\)

$\text{In-Out-problem } \bigvee: \text{kill}_S = \bigcup_{i=1}^{n} \text{kill}_i$  \(7\)

Having all these theorems, we see that we have avoided the state-space explosion problem. These results are given for an “isolated” PAR statement. The next section will put them in the context of a parallel control flow graph and we will see how this gives us an elegant algorithm for computing the data flow information of a parallel program.

**8 The Parallel Control Flow Graph**

The following explanation is based on our implementation of the parallel language Modula-P [22] developed in the COMPARE project. To express parallelism in the intermediate language CCMIR-P3 we define some additional CCMIR statements and define a parallel control flow graph (PCFG).

Looking at the results of this theory, we observe that the PAR statement and its processes are treated by it as a single statement with a complex behavior. The idea for integrating the analysis of a PAR statement is to treat it like other CCMIR statements such as assignment or procedure

---

10 The symbol $X_i$ is abbreviated to $X_i$.

11 Which information is valid before/after a statement.

12 Which information is generated by the statement.

13 Common COMPARE Medium Intermediate Representation, Parallel extension; the intermediate language of the COMPARE compilers.
call, except that it has a more complex DFA behavior. The DFA effects of the PAR statement are determined solely by its process bodies.

The central idea of our parallel control flow graph (PCFG) is that it is a forest of disjoint CFG’s. Each process body and procedure body constitutes a separate CFG. Since jumps into or out of process bodies are forbidden in the source language, there are no jump edges connecting the CFG’s. To form a PCFG of a procedure we connect these separate CFG’s by adding parallel edges between the CFG of a process body and the mirParallel statement containing this process body.

For the program fragment shown in Table 2 the PCFG is given in Figure 1.

After parsing a source program, we obtain a list of all basic blocks, constituting an entire procedure. There is no specific order in this list. From that, we compute for each basic block the list of predecessors and successors (cf. [1]). Each basic block has zero, one or two successors. It has none, if it has as its last statement the EndProcedure or EndProcess statement. It has one, if the last statement is a mirGoto, and two if this is a mirIf statement. Entry basic blocks are marked by the BeginProcedure and BeginProcess CCMIR instructions: their basic blocks have no predecessors. To find the roots of the CFG’s we simply scan the procedure’s list of all basic blocks for basic blocks having no predecessors.

If a replicator specifies that no process should be created, then we draw an extra jump edge from the process body entry to its exit. This solves the problems mentioned in the note to Theorem 3.

Our definition of a PCFG differs from the one given in [26] in the way that their PCFG has two kinds of nodes: ordinary basic blocks and super nodes (or parallel blocks). Such a super node represents the entire process body.

9 Solving the Data Flow Equations of Concurrent Programs

Let’s assume we do not have nested PAR statements. Then we must analyze a program in the following order:

1. Compute gen and kill information for all process bodies and the mirParallel statement itself.
2. Compute gen and kill information for all statements of the procedure’s CFG.
3. Compute in and out for all statements of the statements of the procedure’s CFG.

The dashed lines are parallel edges connecting the separate CFG’s of the process bodies to their mirParallel statement. The grey boxes are CFG’s of the process bodies. The other boxes are part of the procedure body’s CFG.

Figure 1: A PCFG for the program fragment of Table 2.

4. Since we know the exact information reaching the mirParallel statement, we can compute the in and out information of the valid at the statements of the process bodies: the information reaching the mirParallel statement reaches the entry of each process body.

Now it is obvious how to deal with nested PAR statements:

1. Visit the deeply-nested process bodies first and compute their gen and kill information. This corresponds to a depth-first traversal of the PCFG along the parallel edges. This is called inside-out computation of Gen/Kill.
2. Compute gen and kill of all other statements of the procedure’s CFG.
3. Compute in and out for all statements of the procedure’s CFG.
4. Visit the mirParallel statements from outside-in (the reverse order of inside-out) and compute in and out of the process bodies. Outside-in is the top-down traversal of the PCFG along the parallel edges.

This kind of computation of the data flow information is a mixture of the structural [2], [1, page 611] for the effect of the mirParallel statement and iterative (all other statements) method. A similar idea has been presented for DFA of sequential problems by [8].

The computation of the in and out information must be done using an iterative algorithm [11, 1]. At a first glance the same seems to be true for the computation of gen and kill. But this iterative approach for gen and kill solves a broader problem: it computes for each basic block b the set

Note: We have to compute the gen and kill info for a set of basic blocks. In the sequential DFA, this information is not needed.
of \textit{gen} and \textit{kill} information reaching \textit{b}. But we need only the \textit{gen} and \textit{kill} information which is valid at the end of a process body. We can therefore use a simpler algorithm which combines the computation of the local \textit{gen} and \textit{kill} with the computation of the \textit{gen} and \textit{kill} of a set of basic blocks. We visit the basic blocks in the reverse depth first order, which guarantees that at the end of the process we have the same result than using the iterative method.

10 Complexity of this DFA Algorithm

To estimate the complexity of this algorithm, we use as complexity measure the number of visits of a basic block during the iterative computation of the DFA information. A “comparable” sequential program is one where the PAR statement and its process bodies are executed sequentially.

For the computation of the \textit{gen} and \textit{kill} sets, we have to visit each basic block once, both in the sequential and parallel case.

During the computation of \textit{in} and \textit{out} we apply the iterative algorithm several times to different (and disjoint) sets of basic blocks: first, we compute the DFA information for the basic blocks of the procedure’s CFG. Second, we compute \textit{in} and \textit{out} for the sets of basic blocks corresponding to process bodies. The process bodies are considered in the \textit{outside-in} order of the PAR statement.

The number of iterations needed to compute the DFA information is determined by the loop \textit{nestedness} \cite{5} of the source program.

Since we do not have jump edges between basic blocks of different process bodies, we are always computing the DFA information of disjoint sets of basic blocks. Hence the loop \textit{nestedness} is the same for the parallel and the comparable sequential program. Hence the overall number of basic block visits is equal in the parallel and the comparable sequential program.

As a result the data flow analysis of a parallel program has the same complexity as a comparable sequential one.

11 Current and Future Work

Currently the algorithm of \cite{12} for elimination of partial redundancies is implemented in the COMPARE compiler for the source language \textit{Modula-P} \cite{22} which is an parallel extension of Modula-2.

12 Conclusion

This work shows that data flow analysis of parallel programs is possible, and can be done as efficiently as for sequential programs. The novelty is that there is no restriction in the kind shared memory access, nor in the “accuracy” of the resulting DFA information. Hence it is now possible to apply optimizing transformations, which are well known from the sequential context.

To show this, we proved some nice properties of the semilattice based data flow frameworks $\mathcal{D}^C$ and $\mathcal{D}^S$, which allowed us to reduce the number of interleavings needed for the computation of the \textit{meet over all paths} solution of the DFA problem. Then we extended these results to bit-vectors, and obtained simple set equations, computing the DFA information valid inside and after a PAR statement. Based on that we gave a simple algorithm to compute the DFA information valid at all program points. This algorithm is a slight variant of the usual iterative DFA algorithm, but the basic blocks are visited in a special order: \textit{inside-out} and \textit{outside-in} of the nesting structure of the PAR statement.

Acknowledgments

I want to express many thanks for discussions with Bernhard Steffen and Jens Knoop, both of the University of Passau, and with Helmut Emmelmann.

References

\begin{enumerate}
\item K. Kennedy. A survey of data flow analysis techniques. In \cite{15}, pages 5–34.
\end{enumerate}
A What Might Go Wrong: An Example

This section shows the potential problems, when applying sequential data flow analysis to an explicitly parallel program. And more that running an instruction scheduler may cause a wrong execution of a parallel program. The small program given in Table 4 executes the processes $P_1$ and $P_2$ in parallel. It is intuitively clear that $\text{critical}_1$ and $\text{critical}_2$ are never executed at the same time.

On a first glance this program has a race condition [16], (both processes read and write the shared variables, without some kind of explicit synchronization) but this race condition is an intended one: The variables a and b are used to implement a simple synchronization scheme. Extensive work has been done on analyzing parallel programs for potential races but little work has been done to optimize them.

A simple-minded optimizer could perform the following "optimizations" (which would be correct in sequential contexts):

- Propagate a=0 and b=0 to IF a=0 and IF b=0 respectively.
- Then the expressions could be statically evaluated to TRUE.
- Dead code elimination removes the IF and ELSE parts.

And as consequence, both, $\text{critical}_1$ and $\text{critical}_2$ would be executed.

But even without traditional optimizations performed by the compiler, things might go wrong during instruction scheduling: The non-optimized code of process body $P_i$ on a typical RISC processor is given in Table 5. The instruction scheduler could now decide to reorder the instructions, e.g. to insert another instruction between a register load and an immediately following register use instruction (e.g. ldc 1, r0; st r0, a) which results in the code for $P_i$ shown in Table 3a. In this case, it may happen that $\text{critical}_1$ and $\text{critical}_2$ are both executed, as shown in Table 3b.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|}
\hline
ID & Time & T1 & T2 & T3 & T4 \\
\hline
Processor 1: & ldc 1, r0; & '{r1 = 0}' & st r0, a; & cmp r1, 0; & ... & \text{critical}_1 \\
Processor 2: & ldc 1, r0; & '{r1 = 0}' & st r0, b; & cmp r1, 0; & ... & \text{critical}_2 \\
\hline
\end{tabular}
\caption{Code and execution of reordered code}
\end{table}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
a := 0; b := 0; & \text{PAR} & \\
\hline
a := 1; & \{P_1\} & b := 1; \\
IF a = 0 & THEN \text{critical}_1; & IF a = 0 & THEN \text{critical}_2; \\
ELSE & a := 0; & ELSE & b := 0; \\
\text{END} & \text{ELSE else}_1 & \text{ELSE else}_2 & \text{END} \\
\hline
\end{tabular}
\caption{Part of process synchronization code}
\end{table}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
ldc 1, r0; & \text{Load constant 1 into register r0.} \\
st r0, a & \text{Store r0 in memory at address a.} \\
ld b, r1 & \text{Load content of memory b.} \\
cmp r1, 0 & \text{Compare register with constant.} \\
jeq then1 & \text{Conditional branch to then1.} \\
code of else1 & \\
\hline
\end{tabular}
\caption{Non-optimized code for Process $P_i$}
\end{table}

Even worse, some processors, such as the DEC Alpha, are able to reorder the memory accesses to different addresses to some degree. Hence, even the unchanged code could give the wrong result. To avoid this situation, the DEC Alpha offers a memory barrier instruction, which delays the processor until all pending memory requests are fulfilled. In our example this instructions must follow every memory access, which results in a significant slow-down of the program.