MATHEMATICAL PROBLEMS IN COMPUTATION THEORY
BANACH CENTER PUBLICATIONS, VOLUME 21
PWN - POLISH SCIENTIFIC PUBLISHERS
WARSAW 1988

SYNCHRONOUS PARALLELIZATIONS OF SERIAL COMPUTER PROGRAMS

K. CULIK

Computer Science Dept., Wayne State University, Detroit, Michigan, U.S.A.

1. Motivation (Supercomputer and Parallelization)

The Supercomputing Research Center sponsored a Workshop on Parallel Architectures and Algorithms in July 1985 [11], where about ten various supercomputers and candidates of supercomputers were described,·advertized and discussed. All of them were *multiprocessors*, i.e. *parallel computers*, and the number of processors varied between 4 and 64000. The point is that all designers found necessary to exploit a parallelization of algorithms to utilize all (or many) processors available to achieve very high speed (a *supercomputer* is just a very fast computer) a very schort execution time of programs.

A programming of a supercomputer is such a programming of a parallel computer with $N > 1$ processors which leads to an algorithm having the shortest *execution time*, eT_N, possible (with N processors), or the greatest *speedup*, $S_N = eT_N/eT_1$, where eT_1 is the execution time using one single processor. The processor *efficiency* is defined by $E_N = S_N/N$ [8].

The superprogramming is expected to be more difficult than a usual serial programming because of the additional requirement concerning the minimal execution time which is often called an *execution efficiency*. Therefore a conflict between the readability and the execution efficiency [7] is revived. Here the execution efficiency must not be sacrificed to the readability as a structured programming methodology admits [13].

A crucial question is how powerful one processor is. The CONNECTION machine (T. Knight from MIT, and Thinking Machines Corp) has 64000 *bit processors*.

The BUTTERFLY (BBN Lab) and the HYPERCUBE (J. Fox from Cal Tech, and Intel) computers have 128 and 64, processors, respectively. Each processor (as a serial computer) consists of a *cluster of basic processors* for the execution of basic algorithms (of arithmetics and propositional logic) together with the local memory. Therefore any coordination among particular clusters must be done through message passing, and the programming language C is used. Behind these architectures is a new VLSI technology (microcomputers).

The ULTRA computer (J. Schwartz from NYU, and IBM RP3) contains many basic processors, and a shared (global) memory is assumed. It is an architecture reflecting the classical concept of MIMD (multiple instruction multiple data).

The HEP system (B. Smith from Denelcor) has five clusters.

The DATAFLOW machine (J. Gurd from Manchester Univ., U.K.) has twenty processing units and one ring, etc.

There are two different ways how to design algorithms suitable for parallel computers and supercomputers. A programmer himself should design it using a suitable programming language. It can be an *extension of a serial language* (e.g. HEP-FORTRAN or CRAY-FORTRAN) or a *concurrent programming language* (R. Gehani from Bell Lab. is designing CONCURRENT C), or SISAL (Stream Iteration Single Assignment Language), a functional language developed by Livermore Lab. (J. McGraw) and Manchester University, U.K. (J. Gurd) [10] for parallel computers.

In the following an usual (serial) algorithm, Alg (or a serial program) is assumed and the ultimate goal consists of designing a parallelizing compiler which should detect whether or not the given algorithm is parallelizable at all, and if it is then such parallelization of Alg will be found for a prescribed N that $eT_N(\text{Alg})$ is minimal with respects to a defined set of parallelizations of Alg.

On one hand side a parallelizing compiler may be an alternative for the case that new programming languages mentioned above will not be accepted by the programmer community.

On the other hand side the two ways, parallel languages and parallelizing compilers, are, in fact, complementary, as a programmer wants to consider a parallelizm in a *procedure level* (close to the problem solving) and should not be bothered by a parallelizm in an *instruction level* (concerning actual execution).

In Section 2 basic concepts and notation are introduced concerning control flow algorithms, and their representation by usual (serial) control graphs.

Section 3 presents a generalization of serial instructions to parallel instructions of a given width (saying how many processors are needed). The concept of control graph with parallel instructions is introduced.

Section 4 contains four elementary parallelizing construction which are studied and used to transform a serial control graph into a control graph with parallel instructions when preserving serial (synchronous) mode of execution.

Further in Section 5 several types (weak, strong, full) of parallelization of a serial control graph are defined and studied. The main source of parallelization is found in the fact that we are often using functions (and predicates) with arity > 1, i.e. with two or more arguments.

Section 6 presents some conclusions. Two measures of execution times of control graphs (without loops) with serial and parallel instructions are introduced and clarified on some examples. Several topics for futher research are presented.

2. Basic concepts and notation

No computer program in a higher level programming language like PL/I, Pascal, ADA, C, etc., is executed itself. It must be translated into a machine code by a compiler. The compilation is usually done via a lower level programming language, called an *intermediate code*, which is (almost) independent on any source language on one side and, on the other side, it does not reflect any special hardware features of a particular machine. A crucial point is that the execution meaning (and the verification also) of a higher level program is defined by its intermediate code (in which no nested statements or expressions are admitted).

An intermediate code is a sequence of instructions of a few types, the conditional and unconditional jumps are instructions needed only because of the linear form, in which each instruction has at most one successor and predecessor. One can give up the linear form and use ordered pairs of instructions, called *control edges*, to represent the order of execution (in a pair (a, b) the instruction a is supposed to be executed sooner than b). Then one abstracted from a computer program a concept of a *computer algorithm*, called a *control flow algorithm*, with respect to a given set of instructions, Instr. It is a prescription concerning a finite multiset of instructions saying: (1) which instruction should be executed in the next step (and with which augment values) at the start; and, inductively, in the next step (2) if an instruction has been executed and it is not a stop instruction, then it is prescribed uniquely which is the next instruction to be executed (and with which argument values).

It is a mathematical concept of a computer algorithm (introduced by a language independent definition) with respect to Instr, where, obviously, instructions themselves are also computer algorithms but *basic ones* (the concept of a set in set theory is introduced similarly).

The concept of control flow algorithm was introduced in [5] and very conveniently represented by a directed graph $CG = (V, C, r, s)$, $C \subseteq V \times V$ with the start $r \in V$ and the stop $s \in V$, together with two labelings Φ: $V - \{s\} \to$ Instr, and Γ: $C \to$ the set of truth values T and F. CG was called a *flow diagram originally*, but it will be called a *control graph* here (as it is necessary to differentiate also a *flow of data* and to introduce a data graph).

Each instruction from Instr is determined by a name of a basic algorithm from Basis and by some variables chosen from a set of all variables Var (as individual names of its inputs and outputs). E.g. $+$ belongs to Basis and if X, Y, $Z \in \mathrm{Var}$, then $X := Y+Z$ belongs to Instr, etc. It is always assumed that $+$ is interpreted (obviously as the algorithm of addition of arithmetics).

The basis is assumed to contain all basic algorithms of arithmetics corresponding to arithmetic operations and to arithmetic relations, all propositional logic operations and relations, some monoid operations (e.g. concatenation) and relations (e.g. to be a substring of characters), etc., in full accordance to the usual basic data types.

Then a computer is a *finite many sorted algorithmic structure* by which an usual many sorted (relational) structure is determined as it is assumed that by each basic algorithm the corresponding partial operation (a function) or relation (a predicate) is determined when the algorithm is executed.

In this context the set Var is viewed as the memory of such computer, as a naming of a value-object by a variable is conceptually the same thing as a storing that value at the variable viewed as a memory location.

There are all together only five different types of instructions: (1) an assignment ($X := Y+Z$); (2) an input instruction ($X :=$), (3) an output instruction ($:= Y$), (4) a procedure call (CALL SUB(Z, Y; X, W)) where Z and Y are input arguments while X, W are output arguments; and (5) a test ($Y < Z$).

Each occurrence of X and W in the previous instruction examples is called a *defining occurrence* while each occurrence of Y and Z is called an *applied occurrence*. In words: each variable occurring either on the left-hand side of an assignment, or in an input instruction, or as an output in a call is called a defining occurrence while occurrences elsewhere in instructions are called applied occurrences.

A variable $X \in \mathrm{Var}$ is called an *input variable* of $CG = \langle V, C, r, s\rangle$ if

(2.1) (i) either there exists a path $(v_1 = r, v_2, \ldots, v_n)$, $n \geqslant 1$, in CG such that there is an applied occurrence of X in $\Phi(v_n)$, and if $n > 1$, then there is no defining occurrence of X in $\Phi(v_i)$ for $i = 2, \ldots, n-1$;

(ii) or X occurs in an input instruction $\Phi(w)$ and after removing w the X would satisfy (2.1) (i).

A variable $Y \in \mathrm{Var}$ is called an *output variable* of CG if

(2.2) (i) either there exists a $w \in V$ such that Y has its defining occurrence in $\Phi(w)$ and for each path $(v_1 = w, v_2, \ldots, v_m = s)$, $m \geqslant 2$, in CG the following holds: Y has no applied occurrence in $\Phi(v_i)$ for each $i = 2, 3, \ldots, m-1$;

(ii) or Y occurs in an output instruction $\Phi(v)$ and after removing v the Y would satisfy (2.2) (1).

COROLLARY 2.1. *A vertex* $v \in V$ *from* (2.2) (i) *does not belong to any strong component of* CG.

Let $\mathrm{In}\,p_{CG}$, $\mathrm{Out}\,p_{CG}$ be the set of all input, output variables of CG, respectively.

It is assumed that all input and output variables of a CG are determined either by requirements (2.1) (i) and (2.2) (i) (when procedures are concerned) or by requirements (2.1) (ii) and (2.2) (ii) (when main procedure, i.e., program, is concerned).

If the original computer program contains some procedures, then the corresponding control flow algorithm is represented by several control graphs (only one corresponds to the main procedure and all remaining ones represent the procedures). Obviously we are interested to parallelize each procedure as well.

Let an applied occurrence of a variable which satisfies (2.1) (i) be called its *input occurrence*, and, similarly, let a defining occurrence of a variable which satisfies (2.2) (i) be called its *output occurrence*.

Let F_{CG} be a function computed by $CG = \langle V, C, r, s, \Phi, \Gamma\rangle$ (under an assumed interpretation) and let $\mathrm{Dom}\,F_{CG} \subseteq D_1 \times D_2 \times \ldots \times D_p$ be its domain when $\mathrm{In}\,p_{CG} = \{X_1, X_2, \ldots, X_p\}$, $p \geqslant 1$, and D_i are assumed sets of values for $i = 1, 2, \ldots, p$.

By the following requirement we want to exclude (*semantically*) *superfluous* vertices (and their instructions) and edges, which are those which either are never executed, or their new values are never exploited (notice that in another interpretation of the same CG different semantically superfluous instructions may be concerned):

(2.3) (i) for each $v \in V - \{s\}$ there exists an initialization Init of CG such that $(\mathrm{Init}(X_1), \ldots, \mathrm{Init}(X_p)) \in \mathrm{Dom}\,F_{CG}$, and the path $(v_1 = v, v_2, \ldots, v_n = s)$ which underlies the execution sequence ExSeq (CG, Init) satisfies:

(a) there is an i, $1 \leqslant i < n$, such that $v_i = v$, and therefore $\Phi(v)$ is executed;

(b) if Z is a variable which has a defining occurrence in $\Phi(v)$, then there exists j, $i < j < n$, such that Z has its applied occurrence in $\Phi(v_j)$, and there is no defining occurrence of Z in $\Phi(v_h)$, where $i < h < j$ except $Z \in \mathrm{Out}\,p_{CG}$, and Z has in $\Phi(v)$ its output occurrence.

(ii) for each $(v, w) \in C$, where $w \neq s$, there exists an initialization Init of CG such that $(\mathrm{Init}(X_1), \ldots, \mathrm{Init}(X_p)) \in \mathrm{Dom}\,F_{CG}$, and the path

$(v_1 = r, v_2, \ldots, v_n = s)$ which underlies the execution sequence EXSeq (CG, Init) satisfies: there exists i, $1 \leqslant i < n-1$, such that $v_i = v$ and $v_{i+1} = w$, and therefore both $\Phi(v)$ and $\Phi(w)$ are executed.

3. Serial control flow algorithm with parallel instructions

Considering an instruction level parallelization of a (serial) *control flow algorithm* represented by its (*serial*) *control graph* CG one is instrested in another (serial) control graph CG' which has the same input variables and the same output variables as the CG has, and which is function equivalent with CG and admits an execution of several instructions simultaneously (in parallel) on various processors in one step (synchronously), when a MIMD machine with global memory, GM, is assumed.

Two questions arise: (A) which serial instructions can be executed in parallel and when, or *what are parallel instructions*, and (B) which control graphs CG' with parallel instructions can be viewed as *parallelizations of a given* CG. Then the problem is to find such parallelization of CG which has the shortest execution time.

(A) If Instr is a set of all instructions determined by a Basis and by a set of all variables, from GM, one says that two instructions $a \in$Instr and $b \in$Instr are *compatible* (for parallel execution) if

(3.1) (i) either $a \equiv b$, i.e., they are equal as character strings;

(ii) or $a \not\equiv b$, and they are not assigning a value to the same variable from GM, or, in other words, they do not contain a defining occurrence of the same variable.

None of the following instructions $X := A+B$, GET LIST(X), $X := A - B$, CALL SUB$(A, B; X, Y)$ is compatible with any other. On the other hand each test and each output instruction is compatible with any other instruction.

A multiset $[a_1, a_2, \ldots, a_k]$ of mutually compatible instructions from Instr where $k \geqslant 1$ is called a *parallel instruction of the width k*, or, a *k-step*. Let ParInstr be the set of all parallel instructions of arbitrary width, and let ParInstrWMT be the set of all parallel instructions which contain at most one test. Thus Instr $\subseteq$ ParInstrWMT $\subseteq$ ParInstr.

E.g. $[X < Y, Y < Z, X := A+B] \in$ParInstr-ParInstrWMT and it is a 3-step while $[X < Y, X := A+B] \in$ParInstrWMT and it is a 2-step assuming the tests $X < Y$, $Y < Z$ and the assignment statement $X := A+B$ belong to Instr.

One says that a $CG = \langle V, C, r, s, \Phi, \Gamma \rangle$ represents a (*serial*) *control flow algorithm with parallel instructions* (from ParInstr) if

(3.2) (i) $v \in V - \{s\} \Rightarrow \Phi(v) \in$ ParInstr;

(ii) $\Phi(v) = [a_1, \ldots, a_n]$ and $a_1, \ldots, a_k$ are all tests from $\Phi(v)$, where $1 \leqslant k \leqslant n \Rightarrow$ there are m successors $w_1, \ldots, w_m$ of v in CG and Γ: $\{(v, w_1), \ldots, (v, w_m)\} \rightarrow$ partition of $\{T, F\}^k$ into $m > 1$ (not empty) classes, is a one-to-one mapping.

A Boolean function of $a_1, \ldots, a_k$ assumed in (3.2) (ii) for an arbitrary large k seems to be not directly suitable for hardware realization. Therefore the ParInstrWMT has been differentiated, and often the requirement (3.2) will be simplified to

(3.3) $$v \in V - \{s\} \Rightarrow \Phi(v) \in \text{ParInstrWMT}.$$

(B) In answering the second question one has to notice that the requirement of function equivalence of a given serial CG and its parallelization CG' with parallel instructions is necessary but it is not sufficient.

A stronger requirement is needed such that the structure of CG and its execution meaning would be reflected. The execution equivalence of CG and CG' preserves the execution time and therefore it is too strong [2], [3].

We proposed here a constructive answer. Various elementary parallelizing constructions, EPCs, will be defined, and a parallelization CG' of CG will be a (serial) control graph with parallel instructions which is obtained from CG by finite number of applications of these EPCs.

4. Elementary parallelizing constructions

An *elementary parallelizing construction*, EPC, concerns

(1) a control graph $CG = \langle V, C, r, s, \Phi, \Gamma \rangle$ with parallel (or serial) instructions;

(2) one control edge $(v, w) \in C$ such that $w \neq s$ and $\text{id}g_{CG}(w) = 1$; and

(3) the parallel instructions $\Phi(v) = [a_1, \ldots, a_p]$ and $\Phi(w) = [b_1, \ldots, b_j, \ldots, b_q]$, where $1 \leqslant p$ and $1 \leqslant j \leqslant q$.

An EPC consists of constructing a new $CG' = \langle V', C', r', s', \Phi', \Gamma' \rangle$ by eliminating b_j from $\Phi(w)$ and adding it to $\Phi(v)$, and, also, eventually, of some other changes.

According to whether or not b_j is a test and whether or not there is a test among $a_1, \ldots, a_p$ one differentiates four types of EPC as follows: *nn-type* means b_j is *not* a test and there is *no* test in $\Phi(v)$; *nt-type* means b_j is *not* a test but there *is a test in* $\Phi(v)$; *tn-type* means that b_j *is a test* but there is *no* test in $\Phi(v)$; and finally, *tt-type* means that b_j *is a test* and there *is a test* in $\Phi(v)$. Obviously the *tt*-type is excluded when only ParInstrWMT are considered.

A *nn-construction* consists of the following change of a *CG* to a *CG'* presented in Fig. 4.1. On the left is case when $q > 1$ and on the right $q = 1$.

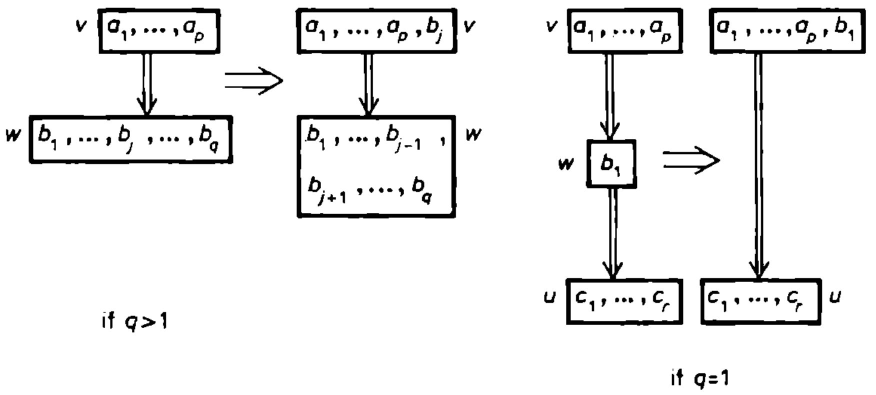

Fig. 4.1

It can be described formally as follows: if $q > 1$ then $\langle V', C', r', s'\rangle = \langle V, C, r, s\rangle$, $\Phi' = \Phi$ except $\Phi'(v) = \Phi(v) \cup \{b_j\}$, $\Phi'(w) = \Phi(w) - \{b_j\}$, and $\Gamma' = \Gamma$; if $q = 1$ then $V' = V - \{w\}$, $C' = (C - \{(v, w), (w, u)\}) \cup \{(v, u)\}$, $r' = r$, $s' = s$, $\Phi' = \Phi$ except $\Phi'(v) = \Phi(v) \cup \{b_1\}$, and $\Gamma' = \Gamma$.

LEMMA 4.1. *Let CG' be a control graph with parallel instructions obtained from a CG by a nn-EPC assuming* (1)–(3). *If CG satisfies*:

(4.1) (i) *if A_i is a variable which has its defining occurrence in a_i, $1 \leqslant i \leqslant p$, then A_i has no applied occurrence in b_j*;

(ii) *b_j is compatible with a_i for each $i = 1, 2, \ldots, p$*;

(iii) *if B_j is a variable which has its defining occurrence in b_j, then B_j has no applied occurrence in any b_h, where $h \neq j$ and $i \leqslant h \leqslant q$,*

then $\mathrm{Inp}_{CG'} = \mathrm{Inp}_{CG}$, $\mathrm{Out}_{PCG'} = \mathrm{Out}_{PCG}$, *and CG' and CG are function equivalent.*

Proof (a sketch). According to definition (2.1) and (2.2), and with respect to requirement (4.1) a *nn*-EPC does not change input and output variables. Therefore one may consider both *CG* and *CG'* being initialized by the same initialization and executed. The initial segments of the two underlying paths are identical and the vertex v either belongs to both or to none. In the second case it is same path and therefore the same execution sequence. If it is infinite then it is undefined, and if it terminates in a stop vertex the resultation is determined.

In the first case two possibilities are to be differentiated:

(a) Let $q > 1$. The two paths are identical and therefore their execution sequences differ only in $\Phi(v) = [a_1, \ldots, a_p]$, $\Phi(w) = [b_1, \ldots, b_q]$ and $\Phi'(v) = [a_1, \ldots, a_p, b_j]$, $\Phi'(w) = [b_1, \ldots, b_{j-1}, b_{j+1}, \ldots, b_q]$. Let Q, R, S and Q', R', S' be the corresponding sequences of states of memory.

Obviously $Q = Q'$ which is the state before the execution of $\Phi(v)$ and $\Phi'(v)$. After their execution the only changes of states concern variables $A_1, \ldots, A_p$ and B_j, and $R(A_i) = R'(A_i)$ for each $i = 1, 2, \ldots, p$. Therefore, eventually, $R(B_j) = R'(B_j)$. Now after the execution of $\Phi(w)$ and $\Phi'(w)$ consider $S(B_h)$ and $S'(B_h)$ for $h = 1, 2, \ldots, q$, assuming B_h is the variable having its defining occurrence in b_h.

If $h \neq j$ then in virtue of (4.1) (iii) $S(B_h) = S'(B_h)$ for $h \neq j$, $1 \leqslant h \leqslant q$, and in virtue of (4.1) (i) $S(B_j) = R'(B_j) = S'(B_j)$, which means $S = S'$. Therefore, further, the same execution sequence and the same sequence of states are concerned.

(b) Let $q = 1$. Then the activities concerned are the sequences Q, $\Phi(v)$, R, $\Phi(w)$, S, $\Phi(u)$ and Q', $\Phi'(v)$, R', $\Phi'(u)$. A similar argument as in (a) leads to the conclusion that from $Q = Q'$ follows $S = R'$, which completes the proof.

A *tn-EPC* consists of the following change of *CG* to a *CG'* which is shown in Fig. 4.2 and 4.3.

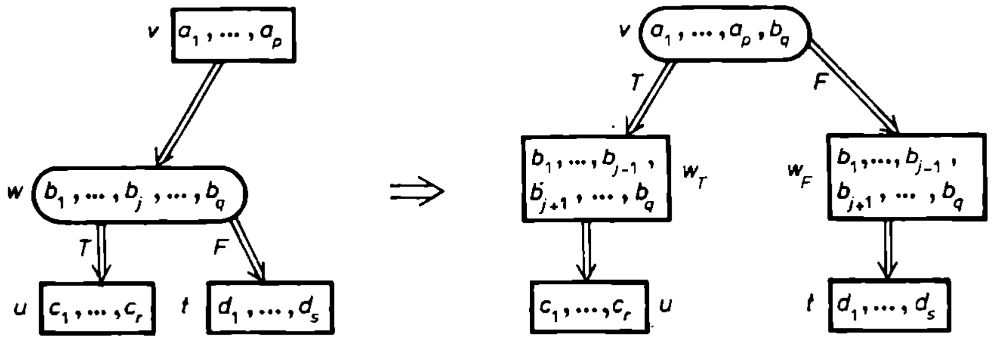

Fig. 4.2

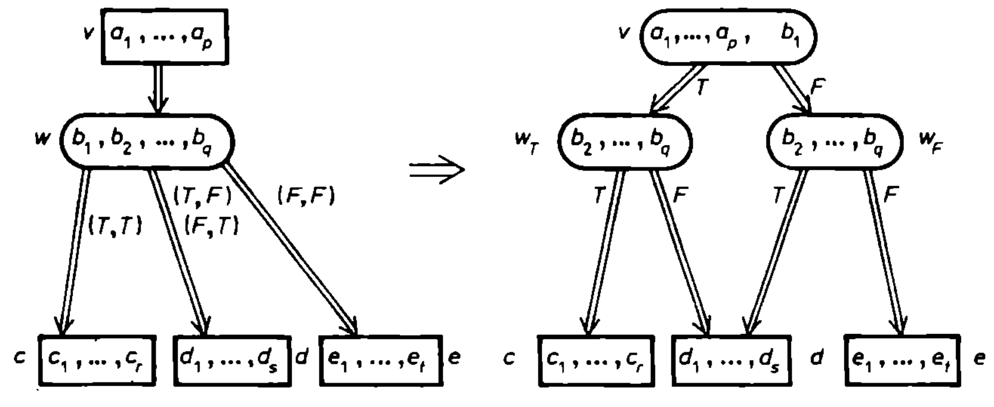

Fig. 4.3

Fig. 4.2 is the case when b_j is the single test in $\Phi(w)$ while in Fig. 4.3 is the case when there are two tests in $\Phi(w)$, b_1 and b_2, and therefore the edges leaving w are labeled by a pair of truth values, and one concrete case is considered when $\operatorname{odg}(w) = 3$. In general case there are $k > 1$ tests in $\Phi(w)$ and edges leaving w are labeled by k tuples of truth value, $\operatorname{odg}(w) < 2^k$. Then one of the tests, b_j, is removed from $\Phi(w)$ and the vertex w is split into two vertices w_T, w_F which both are labeled by the same $\Phi'(w)$, and the edges leaving them terminate in successors of w, and are labeled by $(k-1)$-tuples of truth values accordingly.

LEMMA 4.2 *Let CG' be a control graph with parallel instructions obtained from a CG by a tn-EPC assuming* (1)–(3). *If CG satisfies* (4.1) (i), *then* $\mathrm{Inp}_{CG} = \mathrm{Inp}_{CG'}$, $\mathrm{Outp}_{CG} = \mathrm{Outp}_{CG'}$, *and CG' and CG are function equivalent.*

Proof. Let $b_1, \ldots, b_k$ be all tests in $\Phi(w)$, where $k \geqslant 1$ and let $w_1, \ldots, w_m$ be all successors of w, where $2 \leqslant m < 2^k$, and let b_1 be removed from $\Phi(w)$ and added to $\Phi(v)$. A *tn*-construction does not change input and output variables. Assuming the same initalization of CG and CG' one determines paths with the same initial segments differing only as follows: v, w, w_1 in CG and v, w_T, w_i or v, w_F, w_i in CG'. As only one test has been moved the states of memory are without any change at all. Therefore it remains to show that all the tests are executed with the same result truth values. But it is clear. If the execution of w in CG leads to a k-tuple $(tv_1, \ldots, tv_k)$, thus $\Gamma(w, w_i) = (tv_1, \ldots, tv_k)$, then either $tv_1 = T$ and $\Gamma'(v, w_t) = T$, $\Gamma'(w_t, w_i) = (tv_2, \ldots, tv_k)$ is choosen, or $tv_1 = F$ and $\Gamma'(v, w_F) = F$, $\Gamma'(w_F, w_i) = (tv_2, \ldots, tv_k)$ is choosen, which completes the proof.

A *nt-construction* consists of the following change of CG to a CG' which are presented in Fig. 4.4 and 4.5.

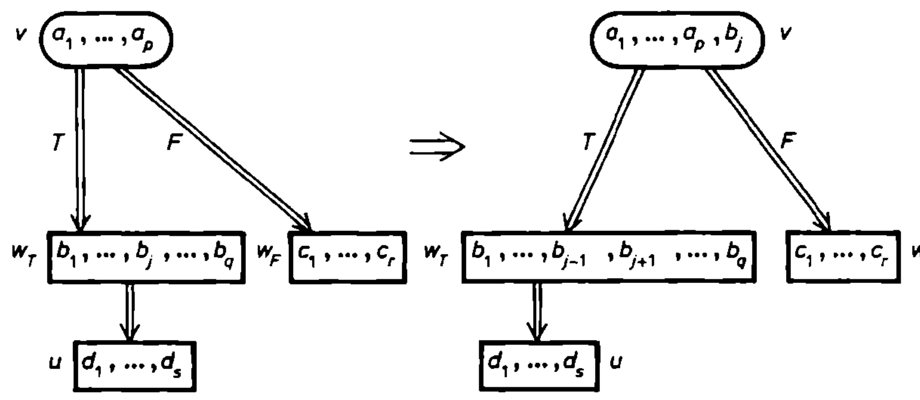

Fig. 4.4

In Fig. 4.4 there is only one test among the instruction from $\Phi(v)$ while in Fig. 4.5 there are two tests in $\Phi(v)$ and one particular way of a distribution of pairs of truth values among the edges leaving v. In both figures $q > 1$ is assumed. If $q = 1$ then the corresponding vertex w disappears in CG' and the corresponding edge leaving v terminates in u directly. The general

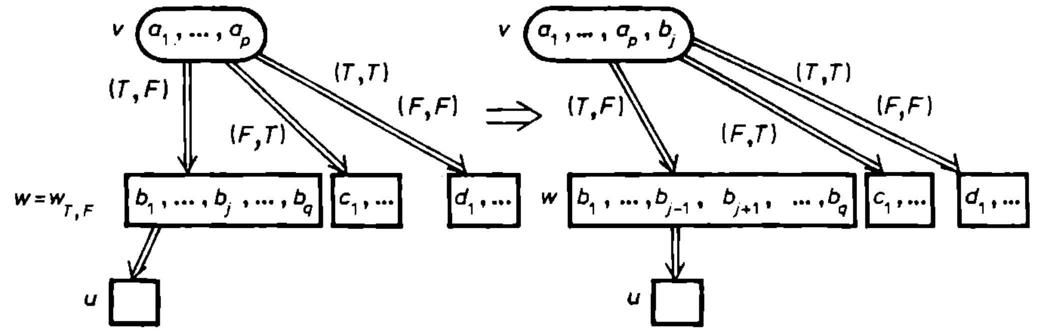

Fig. 4.5

case when there are $k \geqslant 1$ tests in $\Phi(v)$ and $\Gamma(v, w) = (tv_1, \ldots, tv_k)$ is described in a similar way.

LEMMA 4.3. *Let CG′ be a control graph with parallel instructions obtained from a CG by a nt-construction assuming* (1)–(3). *If either the* (4.1) (i)–(iii) *together with* (4.1) (iv) *is satisfied, where*

(4.1) (iv) *if there is a defining occurrence of a variable B_j in b_j, then each path $(v_1, v_2, \ldots, v_m = s)$ such that v_1 is a successor of v and $v_1 \neq w$ satisfies the following requirement: either B_j does not occur in any $\Phi(v_i)$ for $i = 1, 2, \ldots, m-1$, or it does and if h, $1 \leqslant h < m-1$, is the smallest index such that B_j occurs in $\Phi(v_h)$, then there is no applied occurrence of B_j in $\Phi(v_h)$ (it means that the previous value of B_j will not be used in the execution sequence $\Phi(v_1), \ldots, \Phi(v_{m-1})$ at all), or b_j is an output instruction and* (4.1) (i) *is satisfied,*

then $\mathrm{Inp}_{CG} = \mathrm{Inp}_{CG'}$, $\mathrm{Out}p_{CG} = \mathrm{Out}p_{CG'}$, *and CG is function equivalent with CG′.*

If (4.1) (iv) *is not satisfied, then there exists an interpretation of CG and CG′ (viewed as schemes) such that CG and CG′ are not function equivalent.*

Proof. If b_j is an output instruction, then the assertion follows immediately from definition (2.2) and from the properties of a *nt*-construction. If b_j is not an output instruction (and it is not a test), then a *nt*-construction preserves input and output variables and therefore we can consider an initialization of both *CG* and *CG′* by the same values of input variables. The corresponding paths are identical (except the case when $q = 1$ and w is omitted in *CG′*) and the concerned segment is v, w. If Q, R, S are three consecutive states in *CG*, and Q', R', S' in *CG′*, then one assumes $Q = Q'$. After the execution of $\Phi(v)$ and of $\Phi'(v)$ one has $R(A_i) = R'(A_i)$ for $i = 1, 2, \ldots, p$, but $R(B_j)$ and $R'(B_j)$ may differ. As $\Gamma(v, w) = \Gamma'(v, w)$ according to (4.1) (i) and (iii) $S(B_h) = S'(B_j)$ for $h = 1, 2, \ldots, q$ and $h \neq j$. Further $S(B_j) = R'(B_j) = S'(B_j)$ and therefore $S = S'$.

Now consider another initialization such that the concerned sequence is v, w', where $w' \neq w$ and w' is a successor of v.

If $(v, w' = v_1, v_2, \ldots, v_m = s)$ is a path considered in (4.1) (iv) and h is the index concerned then let $S_0, S_1, \ldots, S_h$ be a corresponding sequence of states in CG and $S'_0, S'_1, \ldots, S'_h$ in CG'. Assuming $S_0 = S'_0$, one executes $\Phi(v)$ and $\Phi'(v)$ to obtain $S_1(A_i) = S'_1(A_i)$ for $i = 1, 2, \ldots, p$, but $S_1(B_j)$ may differ from $S'_1(B_j)$. In virtue of (4.1) (iv) $S_2 = S'_2$ except, eventually, $S_2(B_j)$ may differ from $S'_2(B_j)$ but the value B_j is not used at any time until it is changed to $S'_h(B_j)$ after the execution of $\Phi'(v_h)$ in CG'. Therefore although $S_i \neq S'_i$ for $i = 1, \ldots, h-1$, it is $S_h = S'_h$, which completes the proof of the functional equivalence.

If (4.1) (iv) is not satisfied, then in Fig. 4.6 is an example showing that CG and CG' are not function equivalent. It complets the proof of Lemma 4.3.

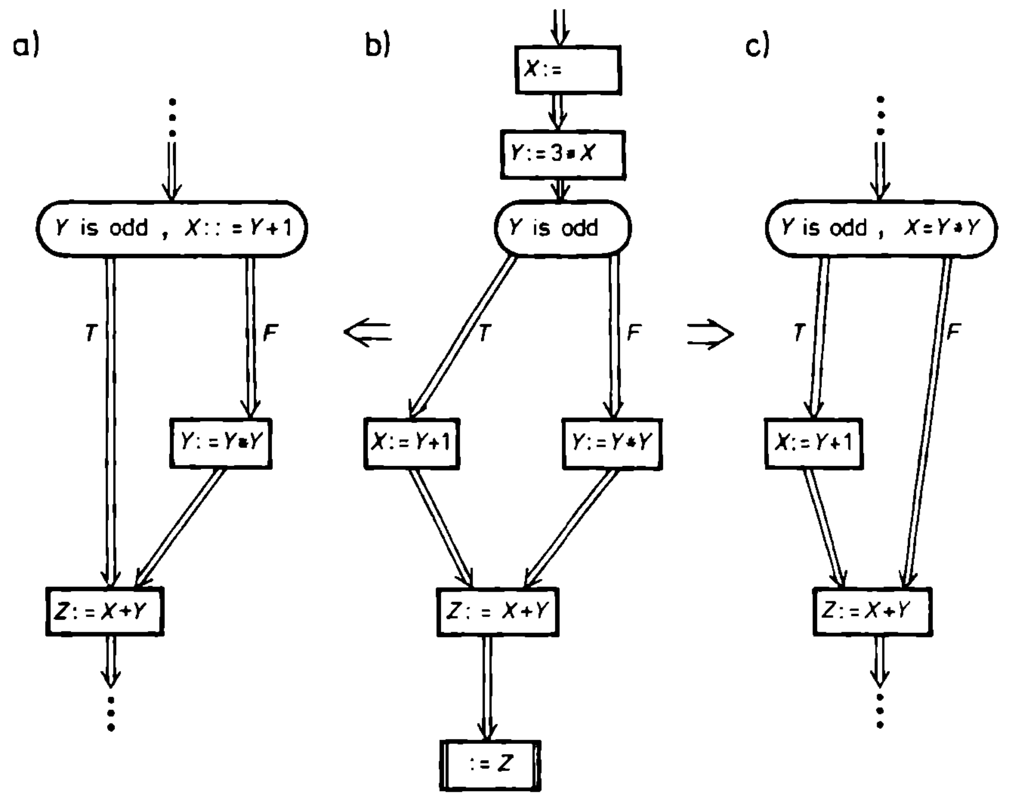

Fig. 4.6

If a serial CG in Fig. 4.6 (b) is initialized by $X = 3$ the result is $Z = 19$, and Fig. 4.6 (c) is the result of a nt-construction applied to Fig. 4.6 (b) which does not satisfy (4.1) (iv). If Fig. 4.6 (c) is initialized by $X = 3$ the result is $Z = 163$. Similarly Fig. 4.6 (a) is obtained by a nt-construction from Fig. 4.6 (b). If $X = 2$ is the initialization, then the results $Z = 38$, and $Z = 43$ show again that the function equivalence is not preserved.

There is another important fact concerning a nt-construction, namely, that the instruction b_j (which has been moved to a test) will be executed unnecessarilly for some truth values of the test. Theoretically it is necessary to differentiate a *parallelization which excludes unnecessary execution* of some instructions (the *nn*-type and *tn*-type) and which *requires unnecessary execution* (the *nt*-type).

A *tt-construction* consists of the following change of a CG into CG' described in Fig. 4.7 and 4.8:

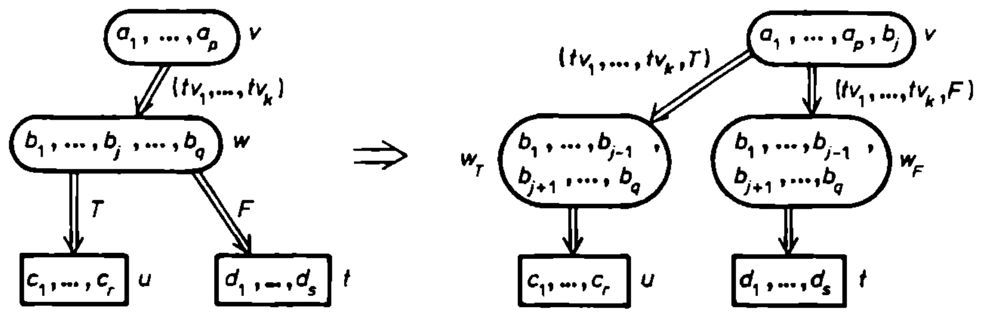

Fig. 4.7

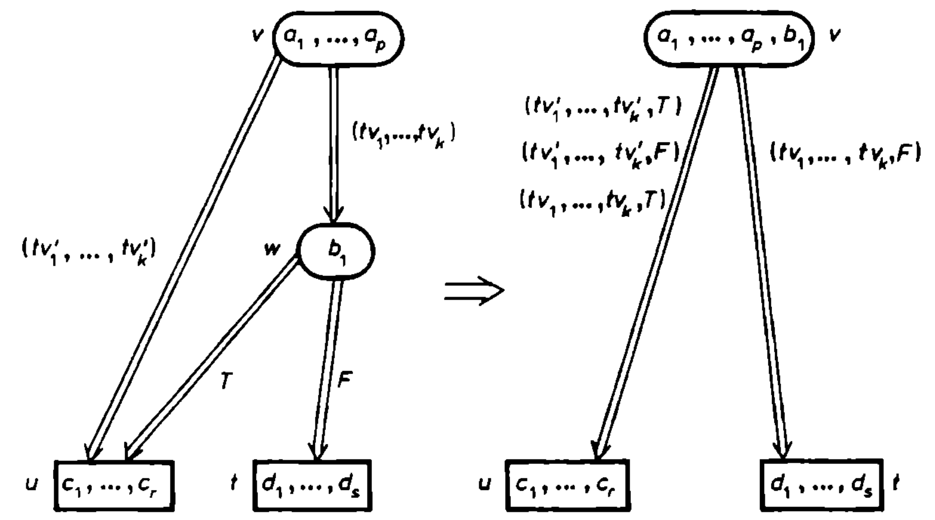

Fig. 4.8

There are $k \geqslant 1$ tests in $\Phi(v)$ in Fig. 4.7, $q > 1$ and only test, b_j, in $\Phi(w)$, while in Fig. 4.8 the case $q = 1$ (thus b_1 is the test) is assumed. In general case when there are $k > 1$ tests in $\Phi(v)$ and $h > 1$ tests in $\Phi(w)$ the labelings of edges leaving v and w should be changed accordingly. An example when $h = 2$ and b_1 is the test to be moved is presented in Fig. 4.9.

LEMMA 4.4. *Let CG' be a control graph with parallel instructions obtained from a CG by a tt-construction assuming* (1)–(3). *If* (4.1) (i) *holds, then* $\mathrm{Inp}_{CG'} = \mathrm{Inp}_{CG}$, $\mathrm{Outp}_{CG'} = \mathrm{Outp}_{CG}$, *and CG' is function equivalent with CG.*

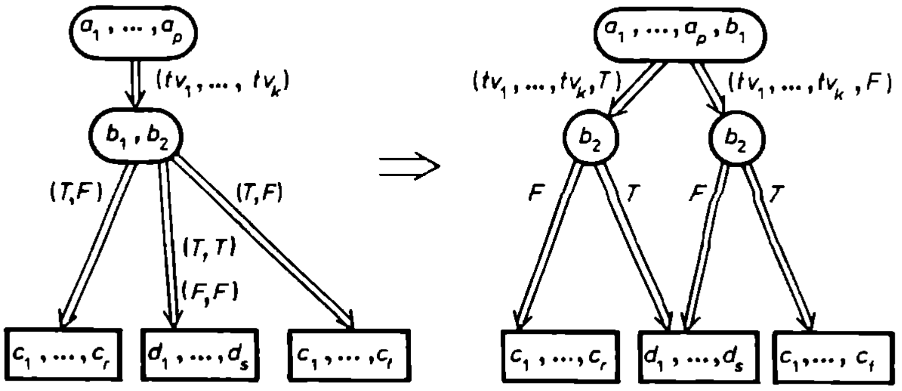

Fig. 4.9

Proof. As only a test is concerned the states of memory need not to be investigated as they are not changed by a *tt*-construction. It remains to check the order of execution of the non-tests but it follows from the transformations of the edge labeling immediately.

A *tt*-construction (similarly as a *nt*-construction) requires unnecessary execution of the test b_j in all cases when $\Phi'(v)$ is executed, with the result $(tv^*, \dots, tv_k^*, tv_{k+1}^*)$ different from either $(tv_1, \dots, tv_k, T)$ or $(tv_1, \dots, tv_k, F)$. It is reflected by the fact that always the concerned edge is labeled by both $(k+1)$-tuples $(tv_1^*, \dots, tv_k^*, T)$ and $(tv_1^*, \dots, tv_k^*, F)$.

5. Parallelization and parallelizability of serial control graphs

A control graph with parallel instructions is said to be *excluding unnecessary execution* if its execution does not require unnecessary execution of any (serial) instruction (see *nt*- and *tt*-construction in Section 4).

A control graph CG' with parallel instructions is called a *full parallelization of a (serial) CG* if

(5.1) there exists a finite sequence $CG = CG_1, CG_2, \dots, CG_n = CG'$, of control graphs with parallel (or serial) instructions such that each CG_i where $1 < i \leqslant n$ is obtained from CG_{i-1} by one of the following constructions:

(i) a splitting of a vertex;

(ii) a *nn*-construction satisfying (4.1) (i)–(iii);

(iii) a *tn*-construction satisfying (4.1) (i);

(iv) a *nt*-construction satisfying (4.1) (i)–(iv);

(v) a *tt*-construction satisfying (4.1) (i);

when at least one of elementary parallelizing constructions (ii)–(v) is used.

If any *tt*-construction of (v) is not used, then CG' is called a *strong parallelization of CG*, and if both *tt*- and *nt*-constructions of (v) and (iv) are not permitted then CG' is called a *weak parallelization of CG*.

LEMMA 5.1. *Each weak parallelization CG' of a serial control graph CG is a control graph with parallel instructions which excludes unnecessary execution, and each parallel instruction of which belongs to* ParInstrWMT. *Further* $\mathrm{Inp}_{CG'} = \mathrm{Inp}_{CG}$, $\mathrm{Outp}_{CG'} = \mathrm{Outp}_{CG}$ *and CG' and CG are function equivalent.*

The proof follows from definition (5.1) and Lemma 4.1 and 4.2 immediately.

A (serial) control graph is called *weakly parallelizable* if there exists at least one its weak parallelization.

THEOREM 5.2. *If a serial control graph $CG = \langle V, C, r, s, \Phi, \Gamma\rangle$ without I/0-instructions satisfies:*

(5.2) *there exists a control edge $(v, w) \in CG$ where $v, w \in V - \{s\}$ such that $\Phi(v)$ is not a test, and if X is a variable which has a defining occurrence in $\Phi(v)$ then X does not have an applied occurrence in $\Phi(w)$.*

then CG is weakly parallelizable for each interpretation of CG.

If (5.2) *is not satisfied then CG need not to be weakly parallelizable and there exists such interpretation of CG in which CG is not weakly parallelizable.*

Proof. If (5.2) is satisfied and $\mathrm{id}g_{CG}(w) = 1$, then requirement (4.1) (i)–(iii) is satisfied (as $\Phi(v)$ and $\Phi(w)$ are just serial instructions). Therefore, according to whether $\Phi(w)$ is not a test or it is a test a CG' obtained from CG by Lemma 4.1 or 4.2, respectively, is a weak parallelization of CG.

If $\mathrm{id}g_{CG}(w) > 1$ then one can split the vertex w into w' and w'' such that $\mathrm{id}g_{CG'}(w') = 1$, $\mathrm{id}g_{CG^*}(w'') = \mathrm{id}g_{CG}(w) - 1$, where CG^* is the result of the splitting. It reduces the situation to the previous case (as the splitting is a legal step to get a weak parallelization according to (5.1)).

As above there is no use of any interpretation the assertion holds for each interpretation (the same one for CG and CG').

If (5.2) is not satisfied let us consider an arbitrary $(v, w) \in C$, where $v, w \in V - \{s\}$ and $\Phi(v)$ is not a test, thus it assigns to a variable X, and let X have its applied occurrence in $\Phi(w)$. If $\Phi(w)$ assigns to a variable B (thus a *nn*-construction is concerned), then in any free (Herbrand) interpretation CG and CG' are not function equivalent (as if $\Phi(v) =_{\mathrm{df}} X := f(Y, Z)$ and $\Phi(w) =_{\mathrm{df}} B := g(X, A)$, then $g(X, A) \neq g(F(Y, Z), A)$). If $\Phi(w)$ is a test (a *tn*-construction is concerned) and, e.g. $\Phi(w) =_{\mathrm{df}} p(X, A)$, then one can interpret p in such a way that $\mathrm{int}p(X, A) \neq \mathrm{int}p(f(Y, Z), A)$, which completes the proof.

LEMMA 5.3. *Let $CG = \langle V, C, r, s, \Phi, \Gamma\rangle$ be a serial control graph with Φ: $V - \{s\} \to$ Instr(Basis, Var) but without I/0-instructions, procedures and arrays, which satisfies* (2.3) *and* $|\mathrm{Inp}_{CG}| = |\mathrm{Outp}_{CG}| = 1$.

(a) *If CG is weakly parallelizable and there are no tests in Basis, then there exists at least one basic algorithm in Basis which computes a function with the arity* > 1.

(b) *There exist weakly parallelizable CGs such that all the basic algorithms from Basis compute either unary functions or unary predicates* (*and at least one basic algortihm is a test*).

Proof. (a) By Theorem 5.2 there exists a $(v, w) \in C$, where $w \neq s$ such that (according to the assumptions) $\Phi(v)$ and $\Phi(w)$ are assignments. According to (2.3) there exists a path $(v_1 = r, v_2, \ldots, v_i = v, v_{i+1} = w, \ldots, v_n, v_{n+1} = s)$ in CG and $[\Phi(v), \Phi(w)]$ will be a parallel instruction in CG' (when a *nn*-construction is applied). Let us assume that all basic algorithms from Basis compute unary functions, and let us derive a contradiction from such assumption. Then $\Phi(v) =_{\mathrm{df}} A_j := f_j(B_j)$ for $j = 1, 2, \ldots, n-1$, where $n \geqslant 2$ and f_j are the unary functions.

If $X \in \mathrm{Inp}_{CG}$ and $Y \in \mathrm{Outp}_{CG}$ then $B_1 = X$ and $A_n = Y$ (according to the assumption), and the occurrence of Y in $\Phi(v_n)$ is the only output occurrence of Y in CG. Therefore in virtue of (2.3) (i) (b) $A_j \in \{(B_{j+1}, \ldots, B_n)\}$ for $j = 1, \ldots, n-1$, where $\{(B_{j+1}, \ldots, B_n)\}$ is a multiset, which means $A_{n-1} = B_n$. Considering $A_{n-2} \in \{B_{n-1}, A_{n-1}\}$ one has to exclude $A_{n-2} = A_{n-1}$ in virtue of (2.3) (i) (b) (as after the execution of $\Phi(v_{n-1})$ the value assigned to A_{n-2} would not be exploited), and therefore $A_{n-2} = B_{n-1}$. By an induction one shows $A_j \in \{B_{j+1}, A_{j+1}, \ldots, A_{n-1}\}$ and $A_j = B_{j+1}$ for $j = 1, 2, \ldots, n-1$.

It contradicts assumption (5.1) that the instructions $\Phi(v) =_{\mathrm{df}} A_i := f_i(B_i)$, $\Phi(w) =_{\mathrm{df}} A_{i+i} := f_{i+i}(B_{i+1})$ satisfy $A_i \neq B_{i+i}$ which completes the proof of (a).

(b) Fig. 5.1(a) is an example of a weakly parallelizable CG with one single test, where $f_1, \ldots, f_4$ are arbitrary unary functions and p is a predicate (obviously they are supposed to be interpreted accordingly). Fig. 5.1(b) is a weak parallelization of Fig. 5.1(a) obtained by a *nn*-construction applied to

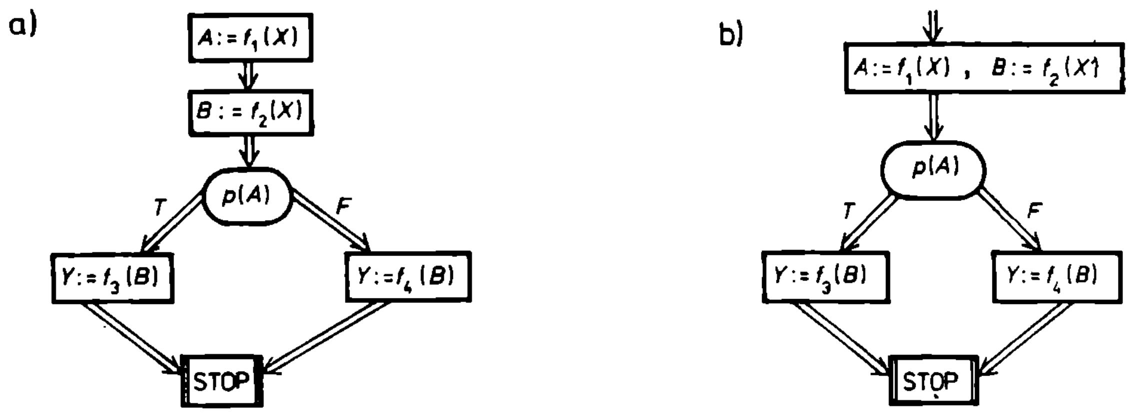

Fig. 5.1 (a), (b)

(v, w), where $v = 1$ and $w = 2$, while Fig. 5.1(c) is a weak parallelization of Fig. 5.1(a) obtained by a *tn*-construction applied to (v, w) when $v = 2$ and $w = 3$.

The control flow algorithm of Fig. 5.1(a) can be represented in a functional programming language as follows:

If $p(f_1(X))$ THEN $f_3(f_2(X))$ ELSE $f_4(f_2(X))$ which shows that $f_2(X)$ had to be computed twice.

A (serial) control graph is called *strongly* (*fully*) *parallelizable* if there exists at least one its strong parallelization which is not a weak parallelization (if there exists at least one its parallelization which is not a strong parallelization).

Neither Fig. 5.1(b) nor Fig. 5.1(c) is weakly parallelizable but Fig. 5.1(b) is strongly parallelizable (which follows from Lemma 4.3 when $v = 3$ and $w = 4$) while Fig. 5.1(c) is not strongly parallelizable. Fig. 5.2 is a strong parallelization of Fig. 5.1(a) and of Fig. 5.1(b) but it is itself no more strongly parallelizable (which follows from Lemma 4.3 immediately).

c)

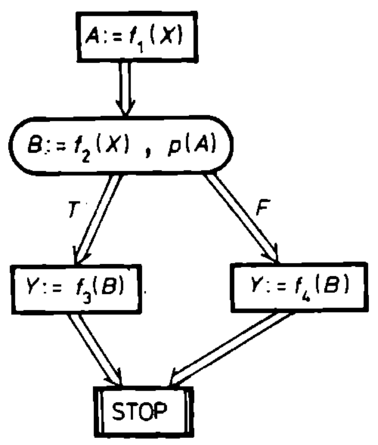

Fig. 5.1(c)

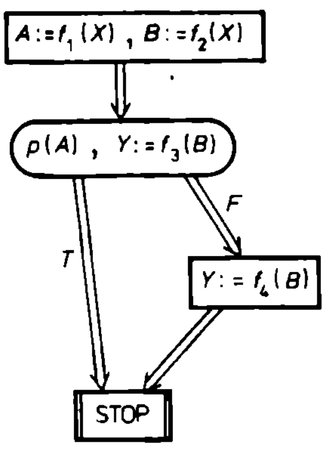

Fig. 5.2

Strong and full parallelizations and parallelizability can be investigated in a similar way as weak parallelization and parallelizability. Intuitively one expects that the desirable parallelizations of serial control graphs will be those which are no more parallelizable. It would be useful to find efficient algorithms to construct unparallelizable parallelizations of given serial control graphs in general, and in particular when some important problems are concerned (and procedures, arrays and *I*/0-instructions are admitted) which are needed in numerical applications or in *AI* applications.

6. Conclusions and further research

All the concepts introduced in Sections 3–5 and assertions with their proofs in Sections 4–5 show that the study of control flow algorithms must be extended to data flow analysis, a special one [2], not necessarily in all generality [1].

Empirical observation that a very important and often source of parallelizm is the fact that we are using operations and relations with arity > 1 is supported theoretically at least partly (Lemma 5.3).

Another observation can be made from Lemmata 4.2–4.4, namely, that very often tests are suitable for parallelization. In fact the concept of parallel instruction (assuming a synchronous mode of execution of a machine) is a generalization of parallel assignments admitted in [9].

Assuming the execution time $eT(a) > 0$ for each (serial) instruction $a \in \mathrm{Instr}(\mathrm{Basis}, \mathrm{Var})$ is given one can define the execution time of a serial CG in several ways [4] in all generality.

Let us consider two cases concerning CGs without strongly connected subgraphs but with tests: a Worest case, $WeT(CG)$, and an Average case, $AveT(CG)$, which are defined using the concept of a path $br = (v_1 = r, v_2, \ldots, v_n, v_{n+1} = s)$, called a *branch of CG*.

If $eT(br) =_{\mathrm{df}} \sum_{j=1}^{n} eT(\Phi(v_j))$ is the execution time of br and Br_{CG} is the set of all branches of CG (eventually only those which actually are determined by some executions), then

$$WeT(CG) = \max_{br \in Br_{CG}} eT(br); \tag{6.1}$$

and

$$AveT(CG) = \Big(\sum_{br \in Br_{CG}} eT(br)\Big)/|Br_{CG}|. \tag{6.2}$$

If $[a_1, \ldots, a_k] \in \mathrm{ParInstr}$, then

$$eT[a_i, \ldots, a_k] = \max_{1 \leqslant i \leqslant k} eT(a_i). \tag{6.3}$$

Considering Fig. 5.1 and assuming $eT(i) = 1$ for each $i = 1, 2, \ldots, 5$, one obtains WeT_1 (Fig. 5.1(a)) $= AveT_1$ (Fig. 5.1(a)) $= 4$ while WeT_2 (Fig. 5.1(b)) $= AveT_2$ (Fig. 5.1(b)) $= WeT_2$ (Fig. 5.1(c)) $= AveT_2$ (Fig. 5.1(c)) $= 3$ and WeT_2 (Fig. 5.2) $= 3 > AveT_2$ (Fig. 5.2) $= 2.5$, where the index k of WeT_k or $AveT_k$ means the maximal width of a parallel instruction in the corresponding control graph.

If we admit (more adequately than in [8]) that different types of instructions may have different execution times then considering again Fig. 5.1 once can assume, e.g., $eT(i) = i$ for $i = 1, 2, \ldots, 5$. Then WeT_1 (Fig. 5.1(a)) $= 11$, $AveT_1$ (Fig. 5.1(a)) $= 10.5$, WeT_2 (Fig. 5.1(b)) $= 10$, $AveT_2$ (Fig. 5.1(b)) $= 9.5$, WeT_2 (Fig. 5.1(c)) $= 9$, $AveT_2$ (Fig. 5.1(c)) $= 8.5$, WeT_2 (Fig. 5.2) $= 11$, and $AveT_2$ (Fig. 5.2) $= 8.5$.

It makes good sense to compare the previous execution times only because they concern parallelizations of the same serial control graph. Only under these assumptions it is meaningful to ask the question which parallelization (weak, strong of full) of a given serial control graph is optimal, i.e., it has the shortest execution time for a prescribed width of parallel instructions. E.g. if the width is 2 one is looking for optimal 2-parallelizations of the given serial control graph, CG, and only if it is a 2-parallelization CG' of CG one may compute the speedup $WS_2(CG) = WeT_1(CG)/WeT_2(CG')$ or $AvS_2(CG) = AveT_1(CG)/AveT_2(CG')$, and the efficiency $WE_2(CG) = WS_2(CG)/2$ or $AvE_2(CG) = AvS_2(CG)/2$.

All these questions should be investigated to get sufficient insight about parallelizations of serial control graphs.

A crucial answer about allocation of particular processors to particular serial instructions a_i from a parallel one $[a_1, a_2, \ldots, a_k]$ is straightforward (assuming $k \leqslant N$ where N is the number of all available processors): allocate a processor P_i to a_i (for $i = 1, 2, \ldots, k$ (in any permutation) as the synchronous execution rule requires that the execution of all $a_i s$ must be completed before the next parallel instruction may be started (similarly as the execution of a parallel statement in [6]). It is the synchronization requirement.

The allocation of $P_i s$ to $a_i s$ does not reflect any intutitive processes detected during the problem solving, as it is assumed in [6], and in other concurrent programming languages.

If asynchronous mode of execution of processors is assumed then it is desirable to allocate processors to larger parts of a given serial program which is extended to a new parallel flow of control (fork vertex and synchronization vertex [3]). It requires a further information about synchronization concerning the communication among the processors in the instruction level which is not specified in [6].

The arity > 1 of basic algorithms (being the main source of parallelizm) indicates that the classical computation theory based on lambda calculus and on the reduction of n-ary functions to only unary ones cannot be a possible conceptual framework to study parallelizability [12]. Therefore the previous investigations based on the concept of computer algorithm (control flow algorithm) is a contribution to a computer computation theory.

References

[1] A. V. Aho, J. D. Ullman, *Principles of Compiler Design*, Addison-Wesley, 1979.

[2] K. Culik, *Towards a theory of control-flow and data-flow algorithms*, Proceedings of 1985 Int. Conf. on Parallel Processing, Aug. 1985, 341–348.

[3] –, N. K. Tsao, *Programming of Supercomputers*, Computer Science Dept., Wayne State University, Detroit, Proposal, Dec. 1985, 10 p.

[4] – and R. G. Trenary, *Parallelizability Degree of Synchronous and Asynchronous Parallel Programs*, Proc. of Conf. on Robotics, Oakland University, Rochester, Mi., 1984, 237–243.

[5] H. H. Goldstine and J. von Neumann, *Planning and Coding programs for an electronic computer instrument*, Pergamon Press 1963, 80–235.
[6] C. A. R. Hoare, *Communicating Sequential process*, Comm. ACM 21 (1978), 666–677.
[7] D. E. Knuth, *Structured Programming with GOTO Statements*, ACM Computing Surveys, Vol. 6, No. 4, Dec. 1974, 261–301.
[8] D. J. Kuck, *A Survey of Parallel Machine Organization and Programming*, ACM Computing Survey, Vol. 9 1977, 29–59.
[9] Z. Manna, *Mathematical Theory of Computation*, McGraw-Hill, 1974.
[10] J. McGraw et al., SISAL: *Streams and Iteration in a Single Assignment Language*, Language Reference Manual, Version 1.2, Jan. 1985, Livermore Nat. Lab. and University of Manchester, M – 160.
[11] *Workshop on Parallel Architectures and Algorithms*, sponsored by the Supercomputing Research Center in Washington, D. C., July 1985.
[12] M. Schönfinkel, *Über die Bausteine der mathematischen Logik*, Math. Ann. (1924), Vol. 92, p. 305.
[13] N. Wirth, *On the Composition of Well-Structured Programs*, ACM Computing Surveys, Vol. 6, No. 4, Dec. 1974, 247–259.

Presented to the semester
Mathematical Problems in Computation Theory
September 16–December 14, 1985