KYBERNETIKA — VOLUME *12* (1976), NUMBER 1

Asynchronous Serioparallel Execution of the Loop

PAVEL KUNC

A solution of the task to execute loops in serioparallel mode is given. It is assumed that number of iterations is (much) higher then the number of processors available. The processors are considered to be identical. Features of suitable hardware and operating system are introduced.

Introduction

Many papers devoted to parallel processing have appeared in last ten years. The problem of parallel programming was discussed both from theoretical and practical point of view. Very important article interested in logical parallelism, determinism and equivalence of parallel processes is e.g. [4]. More practically oriented papers consider objective programming languages and systems demands [3] and many of them are influenced by Dijkstra's semaphores [2]. The present paper is not solving problems of parallelism in general; it accepts the assumption, stated in [1], that "most parallelism in normal programs can be found in loops". There is no doubt that the number of parallely executable loops would remarkably increase if the programs were written with regard to this opportunity.

The task is strictly formulated first, then a solution in the form of on illustrative example written in Algol 60 is given and, at last, the demands on operation code and operating system, which make the solution to be effectively used, are discussed.

1. DEFINITIONS

An important notion in parallel programming is a *unit of parallelism* — a part of a program regarded as a whole by the system controlling parallel execution. In our case it is a loop body. The indexed variable to be processed is determined by a current value of a control variable. This value changes always before the loop body starts to be executed again and it must be accessible during this execution. Obviously the operation changing the value of the control variable and storing it in the "save" variable must make the control variable unaccessible to all the other operations, i.e. it is *indivisible* with regard to the control variable.

A *processor* is a facility, which performs control and arithmetic unit functions of a typical serial computer. It can fetch an instruction and its operands from the memory, perform the instruction and store its result.

CPU of our hypothetical *computer* consists of a finite number of identical processors and a common memory. We consider the only one and homogeneous memory.

A *loop* with a *control variable* is defined as a multiple execution of a consequence of steps i, ii, iii or i, ii, iv, where:

(i) the control variable is set to next value,

(ii) test of the control variable on a boundary value, (the result defines the choice of the next step),

(iii) loop. body execution

(iv) branch to a defined address

The number of executions of the consequence i, ii, iii is called *iteration number* of the loop and denoted n.

The loop is *parallelly executable* unless a memory location exists, the value of it is during two executions of the loop body for two different values of the control variable either changed in both cases or changed in one case and refered to in the other. (This definition satisfies conditions stated in [4], the conditions for parallel executability of Fortran-like loops are given in [1].)

2. THE TASK

Let's have a parallelly executable loop, the iteration number of it is stored a variable N. Let a variable K contain the current number of available processors. Our goal is to define a way which

– enables the processors to share the iterations,

– assures that the processors will be maximally used by executing of user's program.

Without loss of generality we can also assume the processors to be numbered 1, 2, ..., k (and we shall denote them $P_1, \ldots, P_k$) and the user's program to be executed by P_1 until the parallel loop occurs.

Let's assume following procedures available:

$W(I, J)$ performs $I := I + 1$; $J := I$ and is indivisible with regard to I.

FORK (P, ADR) "occupies" processor P, causes it to start its activity with the statement labelled *ADR* and decreases K on 1.

*CALL*1 $(P, ADR1, ADR2)$ stands for the following demands on *OS*:

- release P (this is an obvious demand on releasing the processors currently executing the user's program),
- store *ADR*2 (the address next to the loop),
- start parallel execution on all available processors at *ADR*1.

*CALL*2 (P) simulates *OS* call with demand on releasing P.

BRANCH(ADR) establishes waiting branch beginning with *ADR*.

RELEASE (P) cancels previous occupation of P and increases K on 1.

Note 1. *FORK* procedure does not respect Algol rules of locality of labels.

Note 2. K is maintained by *OS* and its value is equal to the current number of available processors.

Note 3. "Occupation": *A processor cannot be started without previous occupation.* An occupied processor must be released before another occupation.

The following loop evidently meets parallel executability conditions:

```
PROG: begin
        I:= 0;

ZPET: I:= I + 1;
        if I > N then goto DALE;
        X[I]:= if Y[I] > 0·0 then sqrt( Y[I]) else 0·0;
        go to ZPET
        end;

DALE:
```

The execution of translated program can be described be means of augmented Algol 60 as follows:

```
PROG: begin
        begin comment performed by P [1];
        I:= 0;
        CALL 1 (P[1], ZPET, DALE)
        end;
        begin integer C;
              begin performed by OS;
              RELEASE(P[1]);
              BRANCH(DALE);
              C := K;
              for p := 1 step 1 until C do
              FORK(P[p], ZPET)
              end;
              begin integer J[1 : p];
                    begin comment performed by all P[p];
                    ZPET: W(I, J[p]);
                    if J[p] > N then go to A4;
                    X[J[p]] := if Y[J[p]] > 0·0 then sqrt(Y[J[p]]) else 0·0;
                    go to ZPET;
                    A4: C := C − 1;
                    A5: If C ≠ 0 then go to A5;
                    CALL2 (P[p])
                    end
              end
        end;
        begin comment performed by OS for all processors;
        RELEASE P([p])
        end;
        begin comment performed by OS for a free P[q];
        FORK(P[q], DALE)
        end
        end PROG;
DALE:
```

4. DEMANDED HARDWARE AND OPERATING SYSTEM FEATURES

We have already expressed these demands having issued special procedures at the beginning of chapter 3. Level of detail used in this paper does not permit, stating a strict boundary between machine code and operating system share with respect to the control of parallelism. Let's define following rules.

Machine code must contain instructions for:

– operation *RELEASE*,

– starting parallel branch on a free processor (operation *FORK*),

– performing an operation like *W* indivisible with regard to a given variable.

The basic operating system is assumed to satisfy following conditions:

– uses one processor for its purpose only,

– does not contain branches parallel to each other,

– works with memory locations local only to *OS* itself,

- can define global (e.g. C) or local (e.g. $J[p]$) variables so that user's program branches can use them without effect on program corectness,
- can realize *BRANCH* operation.

5. SUMMARY

Let's summarize features of our solution.

Advantage:

a) Operating systems intervences only twice (independently of the values of N and K).

b) K is a variable, not a constant.

c) Processors are working asynchronously and are maximally used by performing user's program instructions.

d) The solution is invariant to the "rest of program" – the original program can contain branches parallel to the loop or not.

e) Only one indivisible general operation is assumed.

Disadvantage:

a) Another processor released during loop exucution can't be used for this purpose,

b) There is a limit of asynchronism – the processors must be released (logically) at the same time.

Evidently, the practical value of the presented solution may be proved in connection with some real multiprocessing system. A great advantage is a fact that parallel executability of a loop can be relatively easily found out, especially in higher programming languages. An asynchronous serioparallel loop can be used, then, in case of purely serial source programs.

(Received November 1, 1974.)

REFERENCES

[1] J. L. Baer: A Survey on Multiprocessing. Technical Report, Univ. of Washington, Seatle, May 1972.

[2] R. M. Karp, R. E. Miller: Parallel Program Schemata. JCSS *3* (1969).

[3] E. W. Dijkstra: Cooperating Sequential Process. Progr. Languages, Academic Press, 1968.

[4] A. J. Bernstein: Analysis of Programs for Parallel Processing. IEEE Trans. on C, *15* (Oct. 1966).

Ing. Pavel Kunc: Výzkumný ústav matematických strojů (Research Institute for Mathematical Machines), Lužná 2, 160 00 Praha 6. Czechoslovakia.